Machine Learning Algorithms for Geometry Processing by Example

by

Evangelos Kalogerakis

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Machine Learning Algorithms for

Geometry Processing by Example

Evangelos Kalogerakis

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2010

This thesis proposes machine learning algorithms for processing geometry by example. Each algorithm takes as input a collection of shapes along with exemplar values of target properties related to shape processing tasks. The goal of the algorithms is to output a function that maps from the shape data to the target properties. The learned functions can be applied to novel input shape data in order to synthesize the target properties with style similar to the training examples. Learning such functions is particularly useful for two different types of geometry processing problems. The first type of problems involves learning functions that map to target properties required for shape interpretation and understanding. The second type of problems involves learning functions that map to geometric attributes of animated shapes required for real-time rendering of dynamic scenes.

With respect to the first type of problems involving shape interpretation and understanding, I demonstrate learning for shape segmentation and line illustration. For shape segmentation, the algorithms learn functions of shape data in order to perform segmentation and recognition of parts in 3D meshes simultaneously. This is in contrast to existing mesh segmentation methods that attempt segmentation without recognition based only on low-level geometric cues. The proposed method does not require any manual parameter tuning and achieves significant improvements in results over the state-of-the-art. For line illustration, the algorithms learn

functions from shape and shading data to hatching properties, given a single exemplar line illustration of a shape. Learning models of such artistic-based properties is extremely challenging, since hatching exhibits significant complexity as a network of overlapping curves of varying orientation, thickness, density, as well as considerable stylistic variation. In contrast to existing algorithms that are hand-tuned or hand-designed from insight and intuition, the proposed technique offers a largely automated and potentially natural workflow for artists.

With respect to the second type of problems involving fast computations of geometric attributes in dynamic scenes, I demonstrate algorithms for learning functions of shape animation parameters that specifically aim at taking advantage of the spatial and temporal coherence in the attribute data. As a result, the learned mappings can be evaluated very efficiently during runtime. This is especially useful when traditional geometric computations are too expensive to re-estimate the shape attributes at each frame. I apply such algorithms to efficiently compute curvature and high-order derivatives of animated surfaces. As a result, curvature-dependent tasks, such as line drawing, which could be previously performed only offline for animated scenes, can now be executed in real-time on modern CPU hardware.

Dedication

To my wife, Olia.

Acknowledgements

First of all, I would like to thank my supervisors, Aaron Hertzmann and Karan Singh. They offered great help, guidance and support during my PhD studies at the University of Toronto. I would also like to thank my PhD committee members, Eugene Fiume and Jos Stam, for their valuable comments and feedback on my thesis. In addition, I would like to thank the external examiner, Szymon Rusinkiewicz, for his detailed comments and recommendation for my thesis.

I would like to thank my fellow lab members Derek Nowrouzezahrai, Simon Breslav, Patricio Simari, and James McCrae for their collaboration in research.

Finally, I would like to thank my wife, Olia Vesselova, my parents and brother for all of their love and support they gave me throughout the years.

Contents

Chapter 1

Introduction

3D shape processing is fundamental to computer graphics, computer-aided design, computer vision, multimedia and several other fields in computer science and engineering. Shape processing deals with transformation and analysis of 3D geometry data, which typically comes in the form of a raw collection of points or/and faces in 3D space. An important component of 3D shape processing usually involves the extraction or synthesis of various shape properties based on its underlying geometry. Despite the significant advances in this field, there is still a lot to be done for inferring shape properties more automatically and efficiently by exploiting the regularities and repeating patterns in geometry data. With the appearance of large repositories of 3D models on the Internet, such issues are becoming increasingly significant.

Consider the example of shape segmentation, illustrated in Figure 1.1. The figure shows the composite images of segment boundaries selected by different people based on the recent study by Chen *et al.* [16]. The goal of a shape processing algorithm for segmentation would be to infer the parts of all these different shapes in a similar way to segmentations performed by humans. To date, nearly all existing shape segmentation methods attempt segmentation without recognition. When the goal of segmentation can be formulated mathematically (e.g., parti-

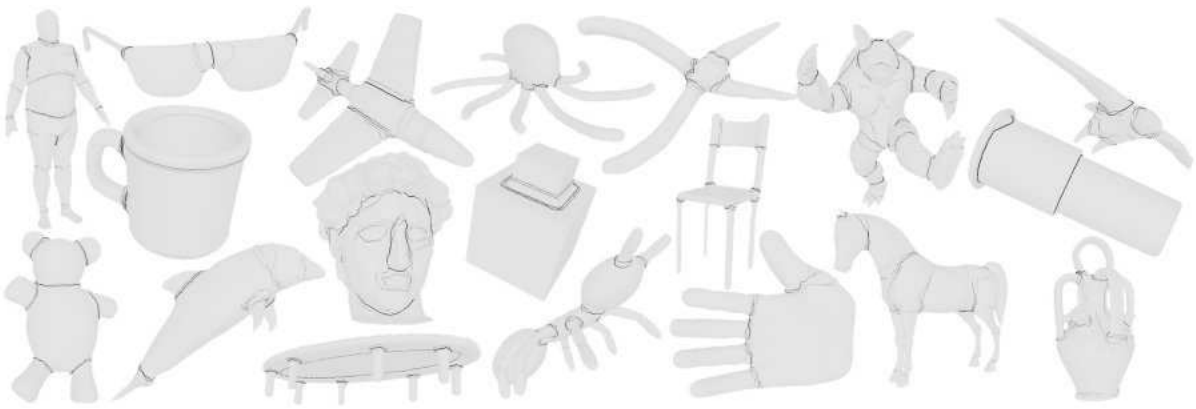


Figure 1.1: *Composite images of segment boundaries selected by different people (the darker the seam the more people have chosen a cut along that edge). This image is taken by the recent survey by Chen* et al. *[16] which considers human segmentations on* 19 *object categories. One example is shown for each of the categories considered in this study.*

tioning into patches of near-constant curvature), low-level geometric cues may be sufficient. However, many tasks require some understanding of the functions or relationships of parts, which are not readily available from low-level geometric cues.

Therefore, an important question in shape processing is how to develop algorithms that would automatically learn to recognize complex patterns in the geometry data, such as parts in shapes as in the above example. Similar to how humans learn by using their past experience, machine learning can be performed based on empirical data, such as from training databases.

This thesis introduces machine learning techniques for a collection of problems in geometry processing. The main ingredient of these algorithms is to learn functions that maps from the shape data to the target properties. The input to the algorithms is a collection of training shapes along with exemplar values of target properties that are relevant to the desired geometry processing tasks. The output is the learned function for each mapping. There are two types of tasks, for which the introduced learning techniques are useful. The first type of tasks involves the computation of properties that require some shape understanding and may also depend on

users' style and preferences. A characteristic example is shape segmentation and part labeling mentioned above. This is a highly nontrivial problem due to the large variability of parts. Previous research has mainly focused on using single geometric criteria, rules or heuristics to find meaningful parts in a shape. However, parts exhibit such large variability that it is unlikely to have satisfactory results for a broad set of shapes based on such approaches. In other words, it is extremely hard to develop a simple mathematical formula or handcraft all possible rules as well as their exceptions in order to detect parts in shapes in general. Instead, much better results can be obtained if a mapping is learned from lots of different shape features to each part based on some representative training examples. The parameters of the mapping as well as the most appropriate features for each part are adaptively selected based on a learned model according the training dataset.

Another characteristic example for this type of problems is artistic rendering of shapes. Here, the stroke locations, orientations, texture and other properties vary not only according to the underlying shape but also according to its shading. In addition, the stroke properties also exhibit enormous variability according to the specific artist's style, communication goals or even mood. Existing approaches employ specific rules with several hand-tuned parameters; however, it is extremely hard to capture all possible styles with hand-tuned models. Machine learning can be also used here to automatically capture several aspects of style based on a few exemplar drawings. Such example-based approaches offer a potentially natural workflow for users. Instead of designing complex user interfaces or requiring users to tune several hard-to-understand parameters, the only workload for users is to provide the machine with a few training examples.

The second type of tasks, where machine learning proves to be particularly useful, involves the computation of shape attributes in animated, dynamic scenes, that exhibit some spatial continuity as well as temporal coherence with respect to the animation parameters. Surface curvature and visibility are examples of such attributes. These shape attributes are known functions of the

input geometry, i.e., they can be estimated with appropriate geometric techniques. However, some of these attributes are very slow to compute geometrically. In this case, machine learning algorithms can exploit their temporal and spatial coherence to learn very compact mappings from animation parameters to these attributes. The learned mappings can be then evaluated very efficiently during runtime, enabling also much faster execution of tasks that depend on these attributes.

The learning techniques for the above two types of tasks could also be combined into a single pipeline, if necessary. First, a target shape property could be inferred for static shapes based on the learning techniques for shape understanding. Then, the property can also be computed efficiently for dynamic scenes using the learning techniques for animated scenes.

1.1 Overview and Contributions

As mentioned above, the goal of the thesis is to develop learning techniques that learn mappings of geometric-based features to target properties. In order to achieve this, first the target properties must be identified. Then, a set of appropriate features must be extracted to form the input space of the mapping. Then, an appropriate learning technique must be designed to match the requirements of this mapping. The thesis describes these considerations for these steps in Chapter 2.

I should strongly emphasize here that machine learning is not and should not be thought of as some monolithic theory. Thus, the thesis does not describe a single, unified workflow or framework for applying learning to geometry processing. There is no single learning algorithm or theory that can be applied to solve all example-based geometry processing problems in general. Each geometry processing problem has its very own characteristics and components, thus, completely different requirements and formulations exist for selecting and developing the

most appropriate learning algorithms. This is very important, because especially in the field of computer graphics, it might be thought that machine learning is about downloading one technique, treating it as a black-box, and then, make it work somehow.

I do however describe the general steps and considerations for applying machine learning to geometry processing. Then, I present the machine learning techniques developed for the above-mentioned types of problems. Each technique has its own contributions to the problem it attempts to solve.

First, I show a learning approach for automatic labeling and segmentation of 3D meshes (Chapter 3). The method obtains state-of-the-art results and is the first to demonstrate effective segmentation and labeling for a broad type of meshes. Mesh labeling itself enables automation of several tasks in computer graphics and computer-aided design that would normally require laborious human intervention. Various applications of mesh labeling for automatic 3D object manufacturing, texturing and character rigging are demonstrated. Second, I show a learning approach for creating line illustrations of 3D models from a single example (Chapter 4). The main contribution of this application is the ability to synthesize detailed line illustrations based on the learned aspects of the artist's hatching style.

With respect to the second type of tasks, I demonstrate a learning technique that computes surface curvature for animated scenes (Chapter 5). The learned model can accurately and efficiently predict surface curvatures and their derivatives during runtime, enabling also real-time object-space rendering of feature lines, such as suggestive contours and apparent ridges. This represents an order-of-magnitude speed-up over the fastest existing algorithms that are capable of estimating curvatures and their derivatives accurately enough for many different types of line drawings.

Finally, Chapter 6 mentions future research directions for applying machine learning techniques for other geometry processing problems.

Limitations: It should be noted that there are also limitations to the learning techniques presented in this thesis. First of all, they require a representative enough training dataset for learning the parameters of each task reliably. However, it might not be always technically feasible to acquire training examples for a problem.

There are also no theoretical guarantees on the generalization performance of the algorithms from a deterministic point of view. In other words, even if a learning algorithm finds a hypothesis that explains the training examples well, it is impossible to deterministically predict the error of the algorithm when it is applied to novel unseen data (unless they are the same as the training data). However, if the learned model performs well on most training data and it is not too complex, it will probably do well on similar test data. This is known as Inductive Learning Hypothesis in the literature of machine learning. In other words, it states that if the hypothesis space is not too complicated and if the training dataset is large enough, the probability of performing much worse on test data than on training data can be bounded. In general, it is important to acquire a training dataset which is large and representative enough of the distribution of the input features and output properties. If this is not possible, then it is likely that any learning technique might overfit the training data and fail.

Finally, the learning techniques of this thesis require considerable time for their training step. In our problems, the learning time was usually several hours. However, once the models are learned, they can be applied to novel input very efficiently. There are also specific limitations to each technique that will be described in detail later in the thesis.

Chapter 2

Machine learning techniques for geometry processing

This chapter presents an overview of the steps and considerations for developing machine learning techniques for geometry processing by example. The key ingredient of these techniques is to learn mappings from shape data to target properties that are relevant to the desired geometry processing tasks. As noted in the previous chapter, there is no single machine learning algorithm or unified framework to be applied for any mapping. Instead, there are several considerations for designing an appropriate learning algorithm or a combination of learning algorithms for a task. In many cases, this also involves intensive experimenting with many different techniques, especially if there are no theoretical evidence for choosing a particular technique. The goal of this chapter is to layout the general strategy for developing learning techniques for geometry processing problems as well as to review several learning techniques that can be applied to these problems depending on the type and characteristics of each mapping. It should be noted that I do not cover every possible learning technique; I mainly focus on the most representative supervised learning techniques and especially the ones that are used in the rest of the thesis.

2.1 Steps for designing learning techniques for geometry processing

Given a collection of input shapes along with exemplar target properties on them, the goal is to develop an appropriate learning technique that would learn a function from the shape data to target properties. Let **s** be the shape data and **t** be the target properties. In this thesis, it is assumed that the shape data are given in the form of triangular meshes. The target properties are represented by continuous or discrete values defined on the mesh triangles, the mesh vertices or the projections of the triangles to the image plane (i.e., the corresponding pixels). For example, a target property could be a symbolic attribute, such as a part label per face or a geometric shape attribute such as the principal curvatures per vertex. The learned function should successfully generalize to novel input shapes $\tilde{\mathbf{s}}$ i.e., correctly output predictions $\tilde{\mathbf{t}}$ for the target properties for shapes that are not the same with the ones used for learning.

There are several steps that can be followed in general to develop an appropriate learning technique for a geometry processing task. Below, I discuss each of them in detail.

Identification of target properties: The first step is to identify the target shape properties that are relevant for the desired geometry processing task and can be learned from training examples. The target properties must exhibit some regularities and repeating patterns that can be explained based on the training data i.e., the training data should come from some unknown probability distribution and should not be completely random. For example, mesh segmentation can be formulated as a problem of assigning a label to each mesh face. In this case, the target properties are the part labels, which are used to perform the segmentation task i.e, the labels of connected components on the mesh induce the mesh segmentation. The part labels strongly depend on the shape properties, hence, they can be learned.

Feature extraction: The shape data needs to be pre-processed and transformed into a space of

features that are expected to be relevant to the target properties. Let $\mathbf{x}$ be the extracted shape features. For example, in the case of mesh segmentation, given a mesh which is a collection of triangles, it would not make sense to use its mesh information as it is (e.g., triangles ids). Finding all the possible relevant shape features for a task might be a nontrivial problem. There might be different relevant features depending on the specific goals of each task and the user's preferences. Therefore, in general, it is better to construct feature vectors out of as many informative features as possible.

For example, the algorithm for learning mesh segmentation in this thesis extracts hundreds of shape features based on several shape descriptors proposed in the computer graphics and vision literature that have been found relevant to segmentation and object recognition. If different features are relevant depending on the dataset or task, it is better to include all different features. During the learning step, feature selection techniques can handle the problem of mining the most relevant features for each task. It is also important to choose features that are as discriminative as possible to predict at least some ranges of values for the target properties. Finally, it is necessary to appropriately scale the features so that they have a similar range of values between different shapes.

Forming training datasets: The next step is to acquire a training dataset that will provide exemplar values for the target properties. The values of the target properties are set by either hand-labeling them or acquiring sensor data. The training dataset should be as representative as possible of the different data that might be encountered during test time. The training examples should also contain consistent values for the target properties $\mathbf{t}$ for each exemplar shape. This means that for the same input $\mathbf{s}$, the target properties should be almost the same; some inconsistencies and noise can be tolerated in general, but inconsistent training examples may result in less reliable learned models. After forming the training dataset, we apply supervised learning techniques described below.

Learning: The main goal of the learning step is to find the function $\mathbf{f}(\mathbf{x};\mathbf{w})$ along with its

parameters **w**, which accurately maps features **x** to target properties **t** and best generalizes to different shape inputs. The mapping may be very high-dimensional or highly non-linear. The identification of the type and characteristics of the mapping is very important for applying the appropriate learning algorithm to the problem.

For example, one thing to examine is if we need to choose a classification or regression technique for the learning problem. If the target property **t** takes continuous values, then the mapping should be expressed as a regression problem (Section 2.2). If the target property is categorical, i.e. it takes values from a finite number of discrete categories, then it should be expressed as a classification problem (Section 2.3).

Then, we examine what model is appropriate for the mapping. Would a linear model be expressive enough to capture the relationship of the features to the target properties or would a non-linear model be more appropriate? This can be decided based on theoretical evidence, intuition, plotting the data, or experimenting with the features (or subsets of them). Sometimes, it is common to express a non-linear mapping as a linear mapping, by simply transforming the features **x** into a higher-dimensional space based on a selected kernel function applied to them. This is known as kernel trick and will be described in more detail in Section 2.2.2.

In addition, if there are interdependencies of the assignments between different target properties, then graphical models could be used to capture these interdependencies. For example, assigning a part label to a vertex strongly depends on the assignments of labels to its neighboring vertices. Graphical models will be briefly presented in Section 2.3.4, and I will mostly focus on Conditional Random Fields for classification.

If the input features **x** form a very high-dimensional space and it is expected that different subsets of features are relevant for different tasks and for different ranges of values of the target property, then a feature selection technique must be used. This decreases the number of parameters of the model, making it sparser. Sparse models generalize better, are more compact

to store, and require less time to be evaluated during runtime. In this thesis, we focus on boosting techniques for regression and classification that efficiently perform automatic feature selection and can handle large numbers of input features (Section 2.4).

Finally, if the dimensionality of the target property is too high and its data points lie close to a manifold of much lower dimensionality because of linear or non-linear interdependencies between the different dimensions of the data, then dimensionality reduction techniques can be used to project them into a lower-dimensional space. This also leads to a more compact model that can be evaluated more efficiently during runtime. Dimensionality reduction techniques will be presented in Section 2.5.

It is also common to develop a learning technique that combines several of the above steps (e.g., dimensionality reduction, feature selection, etc.) along with the necessary adaptations depending on each geometry processing problem. This is the case for all the problems that we will present in this thesis. Thus, developing machine learning techniques for geometry processing does not mean that a black-box technique is simply selected and applied to the data. Instead, the type and characteristics of the mapping should be studied carefully so that appropriate techniques are applied and combined, or even developed from scratch.

Application to novel data: Once the model is learned, the learning algorithms can predict values $\tilde{\mathbf{t}}$ of the target properties for novel shapes, by simply applying the learned mappings. As mentioned earlier, the main goal of learning is successful generalization to novel input.

Experimentation: When developing a learning technique, it is necessary to evaluate its performance. It is also very important to compare its performance with other learning techniques as well as methods that do not use learning (i.e., based on simple rules or heuristics with fixed parameters). For this reason, it is useful to design or acquire a benchmark dataset. There are many types of datasets that can be used for evaluation: a) artificial datasets that are created synthetically based on some simple logic or formulas, b) realistic datasets that are created syn-

thetically based on models with properties similar to what can be found in real problems, and c) real datasets that consist of data representing actual observations in the physical world. In general, it is preferable to evaluate a learning technique on real datasets, so that its behavior is better understood for real-world problems. An issue with real datasets is that they might be very hard to obtain. In addition, they might include some amount of errors and noise. Obviously, it is better when the learning technique can cope with such errors and noise. If there are several outliers and inconsistencies in the acquired dataset, it might be unavoidable to perform some pre-processing to filter out the dataset. However, this step may introduce some bias in the evaluation.

In order to evaluate a supervised learning technique, the dataset needs to be split into a training and a test set i.e., learning will be performed on the training set and then the learned model will be applied to the test data separately. Since it is important to evaluate the learning technique on the whole range of data existing in the dataset, it is better to repetitively apply the learning technique on randomly generated training and test sets, or if time and resources permit, on all possible training and test sets that can be created from the dataset. When comparing different techniques, any kind of quantitative data can be reported regarding the behavior of the proposed algorithm on this problem, such as learning speed, training set error, test set error, performance variance across different training sets and so on. It is always crucial to report the test set error, because this is usually a highly informative measure about the generalization capabilities of the technique.

Examining every possible aspect and specification for benchmarking learning techniques can be a time-consuming and tiresome process for researchers. On the other hand, it is useful to properly benchmark new learning techniques in order to ensure orderly progress in the field. Analyzing all the details to create a proper benchmark is beyond the scope of this thesis: useful studies of benchmarking practices for machine learning techniques can be found in [111, 32].

In the following sections, I will explain some representative learning techniques for regression,

classification and dimensionality reduction. I will also focus on boosting techniques for classification and regression, which I found particularly useful for the applications presented in this thesis. I will also refer to related work in the field of geometry processing that uses learning techniques.

2.2 Regression

The goal of regression is to map the input feature vector $\mathbf{x}$ to one or more continuous target properties $\mathbf{t}$. Curve and surface fitting can be seen as a special case of regression. From this aspect, regression techniques are especially useful for surface reconstruction (e.g. [1, 15, 146, 75, 106]). Regression has been also used for mesh skinning (e.g., [86, 132, 80, 97, 149, 29]).

The majority of machine learning techniques treat regression from a probabilistic point of view i.e., we assume that the target properties can be expressed as deterministic functions $f(\mathbf{x};w)$ of the input features $\mathbf{x}$ plus some noise. Assume for now that we have one target property t and the noise model is Gaussian:

$$t = f(\mathbf{x};\mathbf{w}) + \varepsilon \tag{2.1}$$

where ε is a zero-mean Gaussian random variable with variance σ^2. In this case, given a feature vector input $\mathbf{x}_i$ ($i = 1,2,...N$), the output value of the target property t_i follows the following Gaussian distribution:

$$p(t_i|\mathbf{x}_i,\mathbf{w},\sigma^2) = \mathcal{N}(t_i|f(\mathbf{x};\mathbf{w}),\sigma^2) \tag{2.2}$$

Given N training pairs $\{\mathbf{x}_i,\mathbf{t}_i\}$, the likelihood of all the training target property values given the unknown parameters $\mathbf{w}$ and σ^2 can be expressed as:

$$p(\mathbf{t}|\mathbf{x},\mathbf{w},\sigma^2) = \prod_{i=1}^{N} \mathcal{N}(\mathbf{t}_i|f(\mathbf{x}_i;\mathbf{w}),\sigma^2) \tag{2.3}$$

Maximizing the likelihood of the above data corresponds to maximizing its logarithm. Using the logarithm of the lihelihood is convenient for transforming the products into sums, but also

is also useful for numerical reasons in general:

$$\ln p(\mathbf{t}|\mathbf{x},\mathbf{w},\sigma^2) = -\frac{N}{2}(\ln\sigma^2 + \ln(2\pi)) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(f(\mathbf{x}_i;\mathbf{w}) - \mathbf{t}_i)^2 \tag{2.4}$$

As it can be seen from the above equation, maximizing the log-likelihood corresponds to minimizing the sum of squares of the residuals $r_i^2 = (f(\mathbf{x}_i;\mathbf{w}) - \mathbf{t}_i)^2$. If we assume a linear model, i.e., $f(\mathbf{x}_i;\mathbf{w}) = \mathbf{w}\cdot\mathbf{x}_i$, then we can derive an analytical solution for the parameters $\mathbf{w}_{ML}$ that maximize the log-likelihood, which is essentially the one given from least-squares:

$$\mathbf{w}_{ML} = (\mathbf{x}^T\cdot\mathbf{x})^{-1}\cdot\mathbf{x}^T\cdot\mathbf{t} \tag{2.5}$$

We may also want to fit the model with an offset, which is useful when the average value of the target property is not expected to be 0. In this case, we fit the model $f(\mathbf{x}_i;\mathbf{w}) = \mathbf{w}\cdot\mathbf{x}_i + w_0$. By adding a column of values always equal to 1 to the features: $[\mathbf{x}\ \mathbf{1}]$, the same Equation 2.5 applies; the parameter w_0 is simply incorporated into the vector $\mathbf{w}$ in this case. The parameter w_0 is also known as bias term.

2.2.1 Robust Regression Techniques

A very important remark here is that the least-squares solution correspond to the maximum likelihood solution for the parameters under the assumption that we have gaussian noise on the data. However, if the data follow different probability distributions, or even worse, there are outliers, least-squares is not the most appropriate technique, since in this case it may yield bad estimates of the parameters. Estimating parameters when there are outliers or when the noise follows non-gaussian distributions can be performed using robust statistical estimation techniques. Here, we briefly discuss the most widely used techniques for robust regression.

M-estimation: A popular class of these techniques is the M-estimation techniques [46]. M-estimation aims at minimizing a cost function defined over the residuals:

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{N} \rho(r_i;\mathbf{w}) \tag{2.6}$$

where $\rho(r_i;\mathbf{w})$ is the cost function. Minimizing the above expression leads to maximum-likelihood estimates of the parameters if the cost function is interpreted as the negative log-likelihoods of the assumed probability distributions on the target property values given the unknown parameters. In least squares, the cost function is $\rho(r_i) = r_i^2/\sigma^2$ and the assumed distribution is the gaussian, as mentioned above. Other probability distribution functions assumed for robust M-estimation are the following:

$$p_{Cauchy}(t_i|\mathbf{x}_i,\mathbf{w},\sigma^2) \propto \frac{1}{1+r^2/(2\sigma^2)} \tag{2.7}$$

In this case, the cost function is $\rho(r_i;\mathbf{w}) = \ln(1+r^2/(2\sigma^2))$ and the estimator is known as Cauchy-Lorentzian.

Another popular choice is:

$$p_{Geman}(t_i|\mathbf{x}_i,\mathbf{w},\sigma^2) \propto \exp(-\frac{r^2}{r^2+\sigma^2}) \tag{2.8}$$

In this case, the cost function is $\rho(r_i;\mathbf{w}) = \frac{r^2}{r^2+\sigma^2}$ and the estimator is known as Geman-McLure.

The above both estimators result in penalizing outliers much more than the least-squares approach. However, minimizing such cost functions requires an Iterative Reweighted Least-Squares (IRLS) procedure, which is much slower. In IRLS, a set of weights is kept and updated for the data points and a weighted-least squares problem is solved at each iteration until convergence. There are many other estimators proposed in the literature for M-estimation. An excellent survey on M-estimation techniques for vision can be found here [134]. In [72, 71], I suggested the use of M-estimation for robust estimation of curvature tensors on meshes and point clouds . In figure 2.1, we visualize the principal curvatures estimated using least-squares (left), compared to M-estimation (right). Robust M-estimation yields less noisy curvature values.

Least-Median of Squares: Another robust estimation approach is to minimize the median of

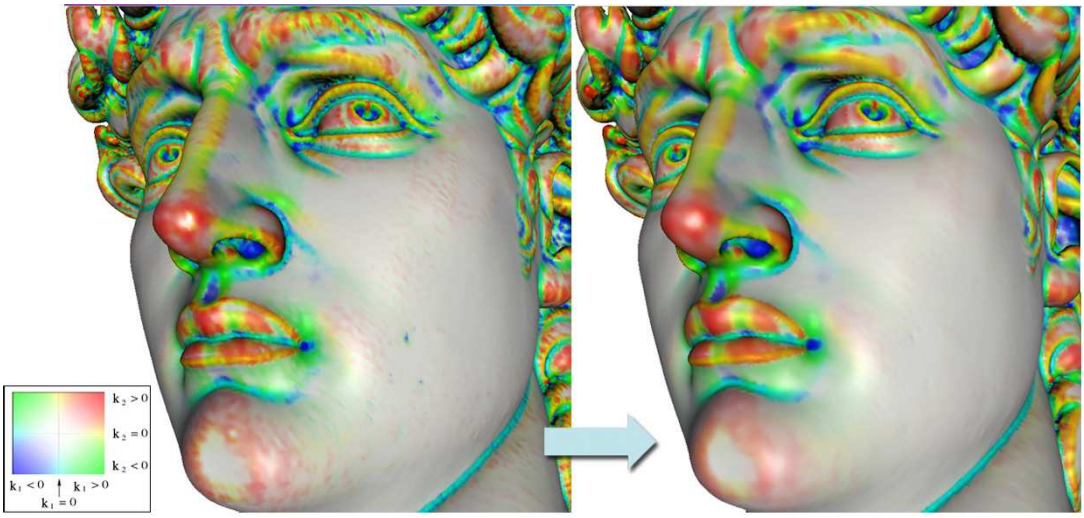


Figure 2.1: Left: *Visualization of principal curvatures estimated by least-squares fitting of the curvature tensor based on finite normal differences ([115]);* **Right:** *Curvature values computed using a robust M-estimator [72] that results in less noisy estimates.*

the squared residuals:

$$\arg\min_{\mathbf{w}} median(r_i^2) \tag{2.9}$$

This technique is called Least-Median of Squares [113] and is especially useful when up to half of the data points are outliers. Minimizing the median of the squared residuals is not trivial, since the median function is not differentiable. Instead, a common approach is to randomly select subsets of training samples, fit the parameters with least-squares for each subset, measure the median of squared residuals over the rest of the points, and finally select the model of the subset that has the least median. Least-Median of Squares has been used for piecewise smooth surface reconstruction [31]. An issue with Least-Median of Squares is that it is computationally expensive and can select suboptimal models when there are relatively few outliers in the data.

Another similar approach is to again randomly select subsets of training pairs, fit the parameters with least-squares for each subset, find the number of the total points that are inliers given a residual threshold, and finally select the model of the subset that has the largest number of inliers. This technique is known as RANSAC [30]. RANSAC has the same issues with Least-

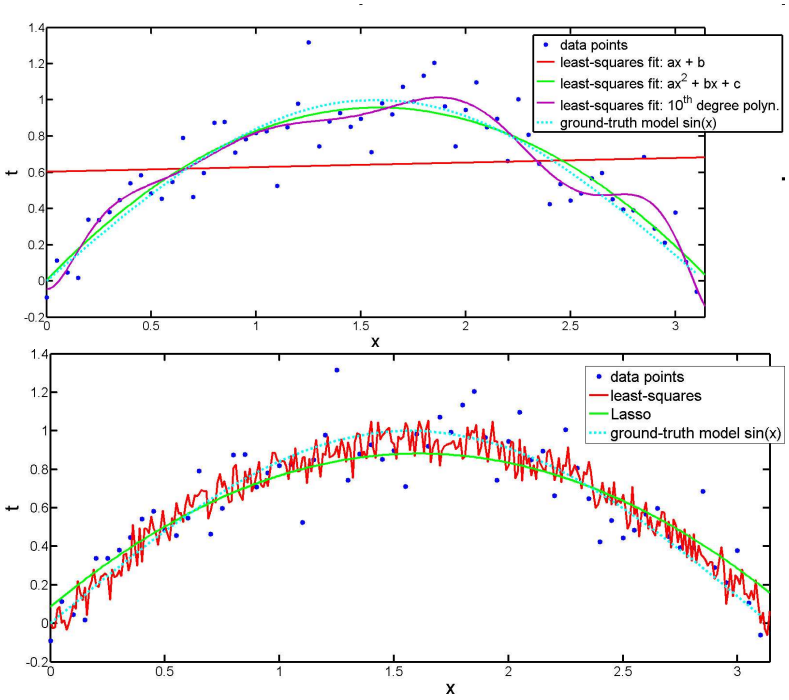


Figure 2.2: Left: *Regression with linear models using linear features (red line), quadratic features (green line),* 10^{th}*-order polynomial features (purple line) on data points generated by a sine function (cyan dotted line) plus some noise. A quadratic curve approximates the ground-truth function very well in contrast to the linear curve that underfits the data and the* 10^{th}*-order polynomial curve that overfits them.* **Right:** *Regression with linear models using input features* $[\mathbf{x}_i^2\ \mathbf{x}_i\ \mathbf{1}\ \varepsilon_1\ \varepsilon_2\ \varepsilon_3]$ *on the same data points. The last three dimensions* $\varepsilon_1, \varepsilon_2, \varepsilon_3$ *contain random numbers, generated by a zero-mean Gaussian distribution with unit-variance. Least-squares (red line) yields non-zero weights on the noise features which are irrelevant to the target properties. Hence, it results in noisy estimates. Using regularization with Lasso (Section 2.2.3) yields almost zeros weights to the noise features and results in a much better approximation the ground-truth function.*

Median of Squares and also requires to tune a parameter for discriminating the inliers.

2.2.2 Non-linear regression

In Equation 2.5, I assumed that $f(\mathbf{x}_i;\mathbf{w}) = \mathbf{w}\cdot\mathbf{x}_i$ i.e., the target property is linearly related to the input feature vector $\mathbf{x}$. However, this might impose several limitations on the model. For example, fitting a linear model to data points generated from a sine function plus some noise is rather inappropriate, as shown in Figure 2.2(top).

A simple trick to enrich the above model is to apply a fixed nonlinear basis on the input features $\mathbf{x}$. In this case, our model can be expressed as a linear combination of basis as follows:

$$f(\mathbf{x}_i;\mathbf{w}) = \mathbf{w}\cdot\phi(\mathbf{x}_i) \tag{2.10}$$

where $\phi(\mathbf{x}_i)$ is the applied basis function. For example, the basis function can be polynomials: $\phi(\mathbf{x}_i) = [\mathbf{x}_i^n\ \mathbf{x}_i^{(n-1)}\ ...\ \mathbf{x}_i^2\ \mathbf{x}_i\ \mathbf{1}]$. For example, fitting a linear model with quadratic features is a much better approximation to the sine function, as shown in Figure 2.2(top). There can be many other choices for the basis functions, such as the gaussian basis:

$$\phi_j(\mathbf{x}_i) = \exp(-(\mathbf{x}_i-\mathbf{m}_j)^T\cdot\mathbf{S}_j^{-1}\cdot(\mathbf{x}_i-\mathbf{m}_j)) \tag{2.11}$$

where $\mathbf{m}_j$ are fixed locations of centers for $j = 1,2,...,M$ Gaussian functions with covariance matrices $\mathbf{S}_j$.

Another example is sigmoid or hyperbolic basis functions that are particularly useful for modeling sharp transitions of the target property with respect to the features:

$$\phi_j(\mathbf{x}_i) = tanh(\mathbf{c}_j\cdot\mathbf{x}_i + c_0) \tag{2.12}$$

where $\mathbf{c}_j$ are fixed parameters that control the rate of the transition. Such basis functions are used in the activation units of neural networks for regression.

We could also select other functions, such as wavelets that have the advantage of being localized in space and frequency, or Fourier bases that have infinite spatial support but specific frequency.

Fortunately, applying fixed bases functions on the input **x** produces models that is still linear with respect to the parameters **w**. In fact, they can be solved with least-squares as above:

$$\mathbf{w}_{ML} = (\mathbf{\Phi}^T \cdot \mathbf{\Phi})^{-1} \cdot \mathbf{\Phi}^T \cdot \mathbf{t} \tag{2.13}$$

where $\mathbf{\Phi}$ is:

$$\mathbf{\Phi} = \begin{bmatrix} \varphi_0(\mathbf{x}_1) & \varphi_1(\mathbf{x}_1) & \dots & \varphi_{M-1}(\mathbf{x}_1) \\ \varphi_0(\mathbf{x}_2) & \varphi_1(\mathbf{x}_2) & \dots & \varphi_{M-1}(\mathbf{x}_2) \\ \dots & \dots & \dots & \dots \\ \varphi_0(\mathbf{x}_N) & \varphi_0(\mathbf{x}_N) & \dots & \varphi_{M-1}(\mathbf{x}_N) \end{bmatrix} \tag{2.14}$$

where N is the number of training examples and M is the number of bases functions.

However, still, having fixed bases function can be a limitation. For example, it would be preferable to consider the locations of the Gaussian basis functions of Equation 2.11 as parameters in our model and learn their optimal locations from the training data. Or it would be preferable to learn the parameters c_j of the hyperbolic tangent functions of equation 2.12. Learning such parameters is common when we perform regression with Neural Networks that are based on sigmoid or hyperbolic tangent activation units. However, these formulations result in models that are non-linear with respect to the parameters, thus linear least-squares cannot be used to solve for the parameters in this case. Instead, non-linear optimization techniques need to be used to minimize the training error. Non-linear techniques might tend to overfit the data, hence, it is important to incorporate regularization while fitting such models, as I will describe below.

2.2.3 Regularization

An important remark regarding least-squares is that minimizing the training error in the data does not necessary yield good models for generalization. Trying to make the training error as little as possible may yield a very complex model that overfits the training data and generalizes

very badly to novel data. As shown in Figure 2.2(top), fitting a 10^{th} order polynomial gives the lowest training error compared to the other fitted models of the figure, but badly approximates the sine function. Complex models usually fit the noise as well, producing a very poor representation of the data that is far from the ground-truth model.

The problem of overfitting also appears when we have a very high-dimensional feature space, where many dimensions might also not be correlated to the target property. Using the least-squares formulation of Equation 2.5 can easily yield large weights on irrelevant dimensions of the input features. Such complex models will also generalize poorly to novel data. An example is shown in Figure 2.2(bottom), where we least-squares fit a linear model on the feature space $[\mathbf{x}_i^2\ \mathbf{x}_i\ \mathbf{1}\ \varepsilon_1\ \varepsilon_2\ \varepsilon_3]$ where $\varepsilon_1, \varepsilon_2, \varepsilon_3$ are random values drawn by a zero-mean unit-variance Gaussian distribution. Least-squares yields non-zero weights even for the last 3 dimensions of the data that contain noise. As a result, the predicted values are also very noisy, far from the ground truth model.

Therefore, reducing the complexity of the learned model is very important for scenarios where we want to discover the most relevant features of the input data to the target property. Discovering simpler models also helps decreasing the storage requirements as well as the computation needed to evaluate the mapping during the runtime.

A common technique to treat this problem in the machine learning literature is to introduce a prior probability distribution over the parameters $\mathbf{w}$ of the model. Using the Bayes theorem, the posterior distribution over for the model parameters is proportional to the product of the likelihood function of the target property values and this prior distribution over the parameters:

$$p(\mathbf{w}|\mathbf{x}, \mathbf{t}, \sigma^2, \alpha) \propto p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \sigma^2) \cdot p(\mathbf{w}|\alpha) \tag{2.15}$$

where α are called the hyperparameters of the prior distribution over the model parameters. Introducing priors over the parameters and finding the maximum of the posterior distribution is called Maximum a Posteriori (MAP) estimation of the parameters. It is common and conve-

nient to use a prior distribution which belongs to the same family with the likelihood distribution. Since we assumed a Gaussian distribution for the likelihood (as defined in Equation 2.3), we also assume a Gaussian distribution over the model parameters:

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{m},\mathbf{S}) \tag{2.16}$$

where $\mathbf{m}$ is the mean and $\mathbf{S}$ covariance matrix for the prior. Then, the posterior is also Gaussian. Then, maximizing the posterior corresponds to maximizing its logarithm, which is:

$$lnp(\mathbf{w}|\mathbf{x},\mathbf{t},\sigma^2,\alpha) = -\frac{1}{2\sigma^2}\sum_{i=1}^{N}(f(\mathbf{x}_i;\mathbf{w})-\mathbf{t}_i)^2 - \frac{1}{2}(\mathbf{w}-\mathbf{m})^T\cdot\mathbf{S}^{-1}(\mathbf{w}-\mathbf{m}) + const. \tag{2.17}$$

where "const." includes terms that do not depend on the model parameters. Thus, the MAP estimation requires the minimization of the term corresponding to the least-squares error plus a term that "pushes" the parameters to a range of values around the mean $\mathbf{m}$ of the prior distribution. If we aim at finding sparse models, then the model parameters should be "pushed" to 0, thus, reasonable choice is to set $\mathbf{m} = 0$. For simplicity, assume that the covariance matrix is diagonal and has all its diagonal values equal to the same variance s. In this case:

$$lnp(\mathbf{w}|\mathbf{x},\mathbf{t},\sigma^2,\alpha) = -\frac{1}{2\sigma^2}\sum_{i=1}^{N}(f(\mathbf{x}_i;\mathbf{w})-\mathbf{t}_i)^2 - \frac{1}{2s}\mathbf{w}^T\cdot\mathbf{w} + const. \tag{2.18}$$

Thus, in this case, maximizing the posterior essentially reduces to the minimization of two terms: $E_D(\mathbf{w}) = (f(\mathbf{x}_i;\mathbf{w})-\mathbf{t}_i)^2$, which is the sum of squared residuals and $E_W(\mathbf{w}) = \lambda\mathbf{w}^T\mathbf{w} = \sum_{d=1}^{D} w_d^2$, which is called regularization term ($d = 1,2,...D$ is an index for each model parameter in the vector $\mathbf{w}$):

$$E(\mathbf{w}) = E_D(\mathbf{w}) + \lambda E_W(\mathbf{w}) \tag{2.19}$$

The parameter $\lambda = \sigma^2/2\mathbf{s}$ is called regularization parameter and determines the influence of the regularization term. The larger the λ is, the sparser the feature vector $\mathbf{w}$ will be. Minimizing 2.19 with this regularization term is also known as ridge regression (or Tikhonov regularization).

There are other regularization terms that can be used to acquire sparse models. A popular choice is $E_W = \lambda\sum_{d=1}^{D}|w_d|$. Minimizing 2.19 with this choice of regularization term is also

known as Lasso [140]. The Lasso formulation is useful because of its tendency to prefer solutions with fewer nonzero parameter values than ridge regression. An example of applying the Lasso formulation is shown in Figure 2.2(bottom). In the context of geometry processing, ridge regression has been used to smoothly fit meshes to a set of control point (called Least-Squares Meshes [133]) or fit sparse models of surface visibility with respect to the animation parameters in [103].

There are limitations to regularization. Regularized models yield biased estimators e.g, training the same model with the same regularization parameters to different samples of the training dataset can yield low variance on the predicted hypotheses (i.e., the solutions to the individual datasets vary a little around their average), but unfortunately can also yield high bias (i.e., the average predicted hypothesis deviates a lot than the ground truth hypothesis). For example, using the zero-mean Gaussian distribution for the prior distribution on the model parameters (Equation 2.16) might pull several parameters to be small instead of generating a few non-zero parameters that would better approximate the ground truth hypothesis. This issue becomes increasingly important when we have very high-dimensional input feature spaces. In this case, it might be better to use an ”aggressive” feature selection technique that attempts to combines models of the selected features. This can be achieved with the boosting techniques that we will discuss in Section 2.4.

Setting the parameter λ for either the ridge regression or the Lasso formulation is also not trivial. On the other hand, setting λ plays a crucial role for determining the model complexity i.e., the larger the λ is, the sparse the model will be. The parameter λ can be set empirically, however, it would be much more preferable to estimate it from the data. One way to do this is to perform cross-validation. Cross-validation involves splitting the training dataset into subsets. For each subset, we fit our model using various values for λ and measure the error on the other subsets (called validation error). We select the value for λ that minimizes the average of validation errors. We can perform the splitting randomly into one training and one validation

dataset and repeat (this process is called repeated random sub-sampling validation) or split the dataset into many subsets and use each of the subsets exactly once as the validation data (this process is called K-fold cross-validation where K corresponds the number of subsets). Cross-validation is very expensive and also has the disadvantage that we cannot use the whole training dataset for fitting all the parameters. Cross-validation is reduced to holdout-validation if we split our exemplar dataset into one training and one validation dataset only once, which can be useful if cross-validation is computationally very expensive.

An alternative to cross-validation is to perform Bayesian regression that also automatically selects the complexity of the model to be fitted. In this case, we introduce prior distributions on the hyperparameters (i.e., the σ^2 and s used in 2.18), called hyperpriors. Then we set the hyperparameters to specific values determined by maximizing the marginal likelihood function $p(\mathbf{t}|\sigma^2, s)$ which is obtained by integrating over the model parameters:

$$p(\mathbf{t}|\sigma^2, s) = \int_{\mathbf{w}} p(\mathbf{t}|\mathbf{w}, \sigma^2) p(\mathbf{w}|s) d\mathbf{w} \tag{2.20}$$

Maximizing the logarithm of above marginal likelihood is called evidence approximation or type 2 maximum likelihood and can be achieved with the Expectation-Maximization algorithm. Finally, we can make predictions $\tilde{\mathbf{t}}$ for new values of $\mathbf{x}$, by evaluating the predictive distribution, given by:

$$p(\tilde{t}|\mathbf{t}) = \int_{\sigma^2} \int_{\mathbf{s}} \int_{\mathbf{w}} p(t|\mathbf{w}, \sigma^2) p(\mathbf{w}|\mathbf{t}, \sigma^2, s) p(\sigma^2, s|\mathbf{t}) \, d\mathbf{w} \, ds \, d\sigma^2 \tag{2.21}$$

in which $\mathbf{t}$ represents the target values from the training set. More details about Bayesian regression can be found in [11]. Bayesian Regression has also some difficulties. Although its main advantage is that the inclusion of prior knowledge arises naturally in many cases, a general criticism is that the hyperpriors are selected on the basis of mathematical convenience rather than representing beliefs about true facts regarding the model parameters. Another disadvantage of Bayesian Regression is that it is computationally expensive especially in high-dimensional feature spaces, since it involves computations that depend on all the dimensions of the input data.

2.2.4 Mixture of Regression Models

In some cases, it turns out to be that the target property is related to the input features with multiple different regression models. The observed data can come from all the models at once, thus it is not possible to fit one model at a time (Figure 2.3(top)). Instead, we need to find a way to use a mixture of regression models for fitting the observed data.

The probabilistic interpretation of regression described in Section 2.2 can be used as a component to formulate a solution to this problem. Assuming a Gaussian noise model, the output value of the target property t_i follows a mixture of Gaussian distributions in this case:

$$p(t_i|\mathbf{x}_i,\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(t_i|f(\mathbf{x}_i;\mathbf{w}_k),\sigma^2) \tag{2.22}$$

where θ denotes the set of all adaptive parameters in the model, namely the parameters of each regression model $\{\mathbf{w}_k\}$, the variance of the Gaussian distributions $\{\sigma^2\}$ and the mixing coefficients $\{\pi_k\}$ that adjust the "weight" of each Gaussian component. The mixing coefficient π_k can be also seen as a prior of picking the $k-$th component to generate a sample from the above mixture distribution. Given N training pairs $\{\mathbf{x}_i,\mathbf{t}_i\}$, the log-likelihood of all the training target property values given the unknown parameters θ can be expressed as:

$$\ln p(\mathbf{t}|\mathbf{x},\theta) = \sum_{i=1}^{N} \ln\Big(\sum_{k=1}^{K} \pi_k \mathcal{N}(t_i|f(\mathbf{x}_i;\mathbf{w}_k),\sigma^2)\Big) \tag{2.23}$$

Maximizing the above log-likelihood cannot be done with a closed-form solution, since the problem is that we do not know which of the components generated each sample. In order to maximize this log-likelihood, we introduce binary latent variables $z_{ik} \in \{0,1\}$ which indicates which component of the mixture is responsible for generating each data point. Hence, for each data point i, all z_{ik}'s are 0 expect to one which is equal to 1 and indicates the component of the mixture that generated this sample. The probability of choosing a component for a sample point is equal to the corresponding mixing coefficient i.e: $p(z_{ik}=1)=\pi_k$ and $p(z_{ik}=0)=1-\pi_k$ and also $\sum_{k=1}^{K}\pi_k = 1$ holds.

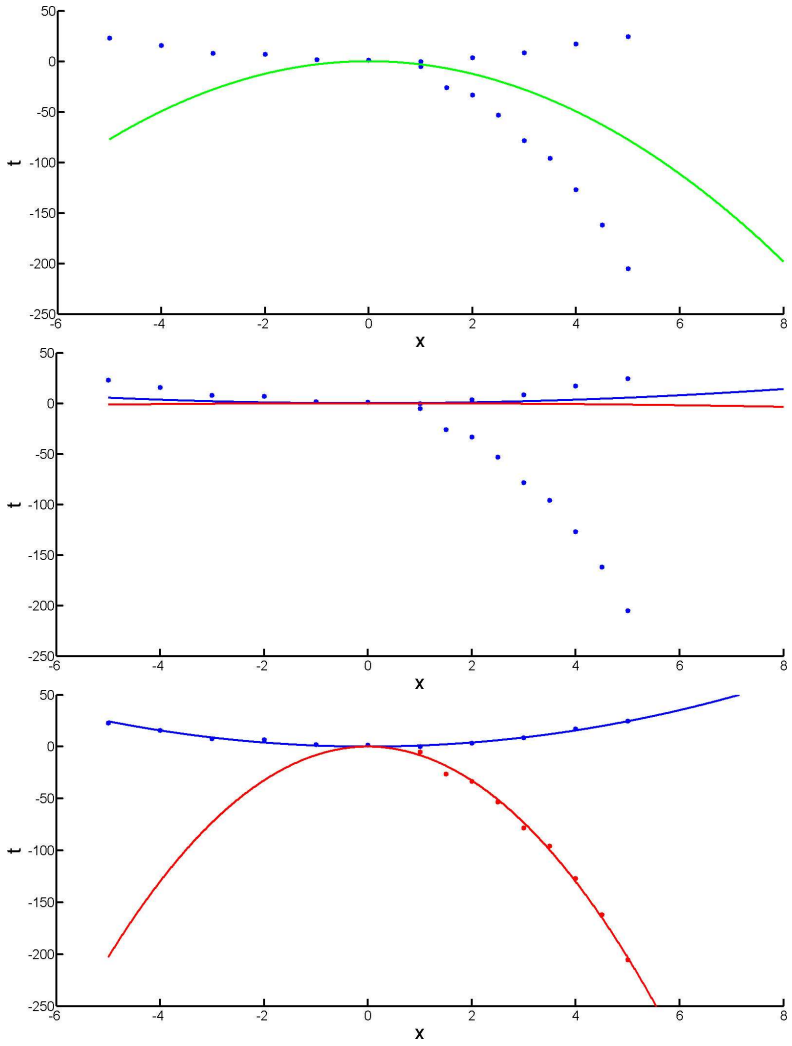


Figure 2.3: *Example of a synthetic dataset where a mixture of regression models is applicable.* **Left:** *Fitting one model* $f(\mathbf{x}) = ax^2$ *(green line) to the data points is rather inappropriate for capturing the underlying mixed components.* **Middle:** *Random initialization for a mixture of two regression models.* **Right:** *Optimizing the mixture model with Expectation-Maximization after* 5 *iterations successfully converges close to the two ground-truth models used to generate the data points*

The complete-data log-likelihood can now be rewritten as follows:

$$\ln p(\mathbf{t}, \mathbf{z}|\mathbf{x}, \theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} z_{ik} \ln \left(\pi_k \mathcal{N}(t_i | f(\mathbf{x}_i; \mathbf{w}_k), \sigma^2)\right) \tag{2.24}$$

or:

$$\ln p(\mathbf{t},\mathbf{z}|\mathbf{x},\theta) = \sum_{i=1}^{N}\sum_{k=1}^{K} z_{ik}(\ln \pi_k + \ln \mathcal{N}(t_i|f(\mathbf{x}_i;\mathbf{w}_k),\sigma^2)) \tag{2.25}$$

This log-likelihood can be maximized using the Expectation-Maximization (EM) algorithm. In general, the EM algorithm can be applied in problems where the goal is to maximize the likelihood $p(\mathbf{t}|\theta)$ with respect to θ, given a joint distribution $p(\mathbf{t},\mathbf{z}|\theta)$ over the observed variables $\mathbf{t}$ and latent variables $\mathbf{z}$, governed by these parameters θ. The EM algorithm iteratively alternates between performing an expectation (E) step, which computes the expectation of the posterior distribution of the latent variables evaluated using the current estimate of the parameters, and a maximization (M) step, which computes the parameters maximizing the log-likelihood. The iterations keep running until a convergence criterion is satisfied. It should be noted, that this iterative scheme might converge to a local maximum. Below, we describe the steps of the EM algorithm and how they can specifically be applied to the mixture model problem:

Step 1 - Initialize the parameters θ: Let θ^{old} the initial setting of the parameters. The parameters θ^{old} can be initialized randomly in general, but this might not be a good strategy, since the EM algorithm can get stuck to a local maximum. Any strategy to find a good initial guess (e.g., with random restarts, a greedy initial search on the parameter space etc) can be helpful for pushing the algorithm to converge to a better solution.

Step 2 (E step) - Evaluate $p(\mathbf{z}|\mathbf{t},\mathbf{x},\theta^{old})$: In the case of the mixture model, using Bayes' theorem, we have:

$$p(z_{ik}|\mathbf{t},\mathbf{x},\theta) = \frac{p(z_{ik})p(t_i|\mathbf{x}_i,\theta_k,z_{ik})}{\sum_{j=1}^{K} p(z_{ij})p(t_i|\mathbf{x}_i,\theta_j,z_{ij})} \tag{2.26}$$

The above equation can be rewritten as:

$$p(z_{ik}|\mathbf{t},\mathbf{x},\theta) = \frac{\pi_k \mathcal{N}(t_i|f(\mathbf{x}_i;\mathbf{w}_k),\sigma^2)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(t_i|f(\mathbf{x}_i;\mathbf{w}_k),\sigma^2)} \tag{2.27}$$

The above posterior probability corresponds to how likely it is for each point i to belong to each component k. It is also known as responsibility of each component k for generating every

data point i. For notation compactness, it is common to represent the responsibilities with $\gamma_{ik} = p(z_{ik}|\mathbf{t}, \mathbf{x}, \theta)$.

Step 3 (M step) - Evaluate $\theta^{new} = \arg\max_{\theta} Q(\theta, \theta^{old})$ where:

$$Q(\theta, \theta^{old}) = \sum_{\mathbf{z}} p(\mathbf{z}|\mathbf{t}, \mathbf{x}, \theta^{old}) \ln p(\mathbf{t}, \mathbf{z}|\mathbf{x}, \theta) \tag{2.28}$$

For the case of the mixture model, we have:

$$Q(\theta, \theta^{old}) = \sum_{i=1}^{N} \sum_{k=1}^{K} \gamma_{ik} (\ln \pi_k + \ln \mathcal{N}(t_i | f(\mathbf{x}_i; \mathbf{w}_k), \sigma^2)) \tag{2.29}$$

Setting the derivative of Q to 0 with respect to π_k and taking into account the constraint $\sum_{k=1}^{K} \pi_k = 1$, we can find that:

$$\pi_k = \frac{1}{N} \sum_{i=1}^{N} \gamma_{ik} \tag{2.30}$$

Setting the derivative of Q to 0 with respect to w_k results in least-squares fitting the K functions $f(\mathbf{x}; \mathbf{w}_k)$ to the data points, using the responsibilities γ_{ik} as weights. For example, if we use linear models of the form $f(\mathbf{x}; \mathbf{w}_k) = \mathbf{w}_k \cdot \phi(\mathbf{x})$ (including kernel transformations as described in the above section), then we solve the following weighted least-squares problem:

$$\mathbf{w}_{ML} = (\Phi^T \Gamma_k \Phi)^{-1} \cdot \Phi^T \Gamma_k \mathbf{t} \tag{2.31}$$

where $\Gamma_k = diag(\gamma_{ik})$ is a diagonal NxN matrix.

Finally, setting the derivative of Q to 0 with respect to σ^2 results in obtaining the following solution:

$$\sigma^2 = \frac{\sum_{i=1}^{N} \sum_{k=1}^{K} \gamma_{ik} (t_i - f(\mathbf{x}; \mathbf{w}_k))^2}{N} \tag{2.32}$$

Step 4 Check for convergence: If the log-likelihood or the parameter values are the same or changed very little with respect to the previous iteration, then terminate. If there is no convergence, go to Step 2 and repeat with $\theta^{old} \leftarrow \theta^{new}$.

An example of applying a mixture of regression models is shown in Figure 2.3. Although the initial guesses for the models are far from the data points (Figure 2.3(middle)), the models are correctly found after a few EM iterations (Figure 2.3(bottom)).

2.2.5 Mixture of Experts

The mixture of regression models that we described above can be still very limited. In many cases, the target property is related to the input features with multiple different regression models that vary according to the values of the input features or even some other features.

We can extend the mixture of regression model described above, by allowing the mixing coefficients to be functions of the input variables:

$$p(t_i|\mathbf{x}_i,\theta) = \sum_{k=1}^{K} \pi_k(\mathbf{x}_i) p_k(t_i|\mathbf{x}_i,\theta) \tag{2.33}$$

The above model is known as mixtures-of-experts model [60], in which the mixing coefficients $\pi_k(\mathbf{x}_i)$ are known as gating functions and the probability densities $p_k(t_i|\mathbf{x}_i,\theta)$ are called experts. This formulation results in having different components responsible for different regions of the input space (i.e, the probability densities $p_k(t_i|\mathbf{x}_i,\theta)$ are experts at making predictions in their own regions and the gating functions determine which components are more important in which region).

The whole model can be fitted again with the Expectation-Maximization algorithm [65], following the general steps described in the above section. The mixture-of-experts of model can be extended even more, by having a mixture distribution for each $p_k(t_i|\mathbf{x}_i,\theta)$ i.e., each component is mixture itself in a mixture distribution. This model is called Hierarchical Mixture-of-Experts [65] and can be fitted with the EM algorithm, starting from the lowest level and then sequentially proceeding with the upper levels of this hierarchical model.

2.3 Classification

When the target property is categorical, then the goal of the learning step is to map the input feature vector $\mathbf{x}$ to the discrete values that the target property can take. These discrete val-

ues are also known as class labels. In the most common scenario, the classes are taken to be disjoint, so that each feature vector is assigned to only one class. In this case, what classification does is to split the input space of $\mathbf{x}$ into a set of decision regions. Each decision region is assigned to one class. The boundaries of these decision regions are called decision boundaries or decision surfaces in higher than 2-dimensional spaces. Linear classification techniques yield linear decision surfaces, while non-linear techniques yield non-linear boundaries. Classification techniques have been widely applied to 3D object segmentation and recognition (e.g., [3, 88, 98, 42]). In non-photorealistic rendering, a few classification methods haven been applied to learn locations of feature curves based on 3D geometry data. Lum and Ma [92] use neural networks and support vector machines to learn locations of feature curves and Cole *et al.*. [18] study feature curve locations using decision trees and linear regression. Fu *et al.* use a combination of Random Forests and Support Vector Machines to predict the upward orientation of objects [36].

We show an example of a categorical property in Figure 2.4. In this example, the target properties we wish to predict are the part labels for each face in a $3D$ mesh. For humans, the part labels take values from the set $\{$head, torso, upper arm, lower arm, upper leg, lower leg, hand, foot$\}$ and for animals they take values from the set $\{$head, torso, neck, leg, tail, ear$\}$. Our goal is to learn a classification function that maps from shape features $\mathbf{x}$ to the segment labels so that we reliably predict the labels on novel unlabeled input meshes. For this reason, we are provided training labeled meshes, whose mesh faces are already labeled (Figure 2.4(left)).

Least-squares for classification: One naive approach to classification is to least-squares fit a linear function of the form $f(\mathbf{x}) = \mathbf{w}\mathbf{x} + w_0$ to the the training data $\{\mathbf{x}_i, t_i\}$, where t_i represents the indices of the labels ($\{1,2,3,4...\}$). The values predicted by this function would be continuous and in fact, they could also be negative or much larger than the number of class labels. However, this does not make sense for a classification problem, where we want to predict class labels, rather than continuous values. A more reasonable approach is to least-squares fit a lin-

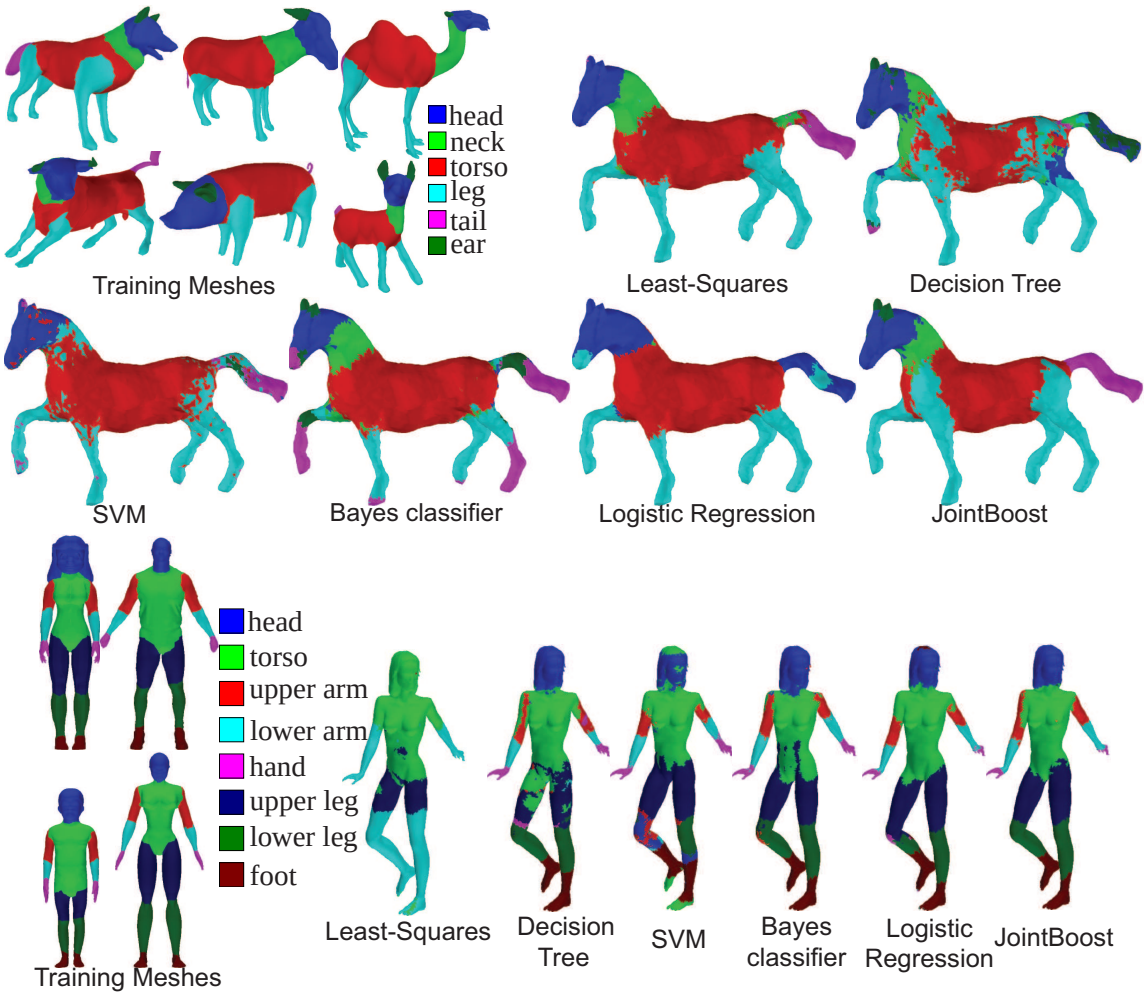


Figure 2.4: *Results of applying various classifiers for labeling animal and human meshes. The goal is to learn a classification function that maps from shape features* $\mathbf{x}$ *to part labels given labeled training meshes. We show results of least-squares, decision trees, SVMs (linear kernel), Gaussian Bayes, Logistic Regression and JointBoost classifiers. For all classifiers, the same input features* $\mathbf{x}$ *are used. These include curvature, PCA, shape diameter, medial surface, geodesic distance, shape context, and spin image features described in Appendix A.1. The regularization parameters of SVMs, Gaussian Bayes, Logistic Regression are estimated by hold-out validation; the validation meshes are selected to be the bottom-right training mesh for humans and the two rightmost training meshes (camel and small goat) for animals. The validation meshes are also used to terminate the boosting iterations for Jointboost. The rest of the training meshes are used for learning the main parameters for each of the above methods (see text for details). Least-squares and decision trees use all the training meshes.*

ear model of the form $f_c(\mathbf{x}) = \mathbf{w}_c\mathbf{x} + w_{c0}$ for each class c separately. In this case, the training pairs are transformed to $\{\mathbf{x}_i, 1\}$ if $t_i = c$ (the data point has training label c) and $\{\mathbf{x}_i, 0\}$ if $t_i \neq c$ (the data point does not have training label c). A new input $\mathbf{x}$ is assigned to the class label $\tilde{c}$ for which the output of the fitted model $f_c(\mathbf{x})$ gives highest value. However, the model outputs do not have any probabilistic interpretation, and also do not lie in the interval $[0,1]$. Also, least-squares assumes a gaussian model of noise that changes the values of the target properties continuously (e.g., from 2 to 2.1), while for classification, noise essentially alters the label of a data point (e.g., from ”head” to ”torso”). A few outliers can easily cause unpredictable effects to the classification, as the sum of squares error heavily penalize predictions that lie far away from the decision boundary, but they are still in the correct decision region. Figure 2.4 shows the rather unpredictable behavior of least-squares for classification: in the animals example, least-squares provides reasonable results, however, it largely fails in the humans example.

Instead, a more correct approach to classification is to predict discrete class labels or to compute posterior probabilities for each class label that lie in the interval $[0,1]$. There are three approaches for achieving either of these two:

- construct a discriminant function that directly assigns the input vectors $\mathbf{x}$ to classes by minimizing some loss function corresponding to the classification error $\sum_i^N \tilde{c}_i \neq c_i$.
- model the class-conditional distributions given by $p(\mathbf{x}|t = c)$, together with prior distributions $p(t = c)$ and then compute posterior probabilities using the Bayes' theorem: $p(t = c|\mathbf{x}) = \frac{p(\mathbf{x}|t=c)p(t=c)}{p(\mathbf{x})}$ (this is also known as probabilistic generative models)
- directly model the conditional probability distribution $p(t = c|\mathbf{x})$ (this is also known as probabilistic discriminative models)

Below, we show characteristic techniques for each of the above approaches.

2.3.1 Discriminant Functions

The classification techniques that are based on discriminant functions aim at directly mapping the input vector **x** to classes. They do not explicitly provide probabilistic output, but they can provide very good results, when they minimize a quantity that is related to the classification error. Unfortunately, minimizing the classification error in terms of the total number of misclassified points in the training data, does not lead to a simple learning algorithm. Such error function is piecewise constant function with respect to the model parameters and gradient-based optimization methods would fail to minimize it, since its gradient is zero or undefined.

Perceptron: Let us focus for now for binary classification problems with two classes (i.e., $t_i \in \{-1, 1\}$). An alternative error function for classification, called perceptron criterion, is the following:

$$E(\mathbf{w}) = -\sum_{i \in M} (\mathbf{w}^T \cdot x_i) t_i \tag{2.34}$$

where M is the set of misclassified training pairs $\{\mathbf{x}_i, t_i\}$. The above error function penalizes misclassified examples and associates zero error for correctly classified examples. The error function is piecewise linear, but can be minimized with stochastic gradient descent. At each step, we update the weights as follows:

$$\mathbf{w} = \mathbf{w} + \eta x_i t_i \tag{2.35}$$

where η is the descent step size (it is called learning rate). Note that we can replace x_i with $\phi(x_i)$ where ϕ is a basis function, as explained in Section 2.2.2.

This algorithm is called Perceptron and is simple but powerful: it can be proved that it can converge to the exact solution in a finite number of steps, if there is such exact solution [112]. However, if there is no such exact solution (i.e., the dataset is not linearly separable or in other words there is no linear decision boundary that splits the dataset into two regions where all points are correctly classified), then the perceptron will never converge. However, it can

find an approximate solution. Also, the perceptron does not generalize readily to multi-class classification problems (i.e., when the number of classes is more than 2).

Decision Trees: Another method to construct a discriminant function is to directly split the input feature space so that each region represents each class. Let $p_{\tau c}$ the current proportion of training points in region R_τ that is assigned to class c. Then, the following error functions are commonly used to determine how to split the input feature space:

$$Q(R_\tau) = \sum_c p_{\tau c} \ln p_{\tau c} \tag{2.36}$$

which is known as cross-entropy, and:

$$Q(R_\tau) = \sum_c p_{\tau c}(1 - p_{\tau c}) \tag{2.37}$$

which is known as Gini index. The above error functions can be minimized iteratively. At each iteration, a threshold on one of the features is selected to split the corresponding region of the feature space into two regions so that the above error function is minimized. The goal is to partition the input feature space with each region having a high proportion of points assigned to a specific class. This partitioning leads to a formation of a ”decision tree”. For a new input data point, we can find the region it belongs to by traversing the tree on a depth-first style according to the decision criteria that are associated to each node.

There have been many versions of decision trees for classification that define alternative error functions along with some pruning criteria for the trees to avoid overfitting. For a survey on decision trees and pruning criteria, see [99]. Decision trees have the advantage that their learned structure is easily interpretable by humans. However, this structure may not correspond to meaningful classification rules. In practive, it has been found that small changes to the training data may result in a different series of splits [47].

In figure 2.4, I show the result of applying a decision tree for classification in the humans and animals example. I experimented with all the types of error functions provided by Matlab: Gini

index, twoing rule and deviance (automatic pruning is used in all cases). I show results for the error function that had the least training error. As it can been seen, the classification results are not that satisfactory compared to other methods.

A more stable approach is to to use random forests. Random forests are ensemble classifiers that consists of many decision trees and outputs probabilities per class averaged over the leaf nodes of all the decision trees [13].

Support Vector Machines: Another method for classification is to attempt to maximize the distance between the decision boundaries and any of the training samples. This distance is also called margin. The main assumption is that if there are many decision boundaries offering exact solutions, it is better to select the boundary that maximizes the margin to achieve the lowest generalization error. This types of classifiers are also known as support vector machines.

Let us assume for now that decision boundary is linear, hence, the boundary is given by a hyperplane defined by the implicit equation $f(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} + w_0 = 0$. Assume that we have a binary classification problem with training data points $\{\mathbf{x}_i, t_i\}$, where $t_i = \{-1, 1\}$. The distance from a training data point $\mathbf{x}_i$ to the decision hyperplane is $|f(x)|/||\mathbf{w}||$. Each point is on the right side of the decision boundary, when $t_i f(\mathbf{x}_i) > 0$. The maximum margin solution is given by solving:

$$\arg\max_{\mathbf{w},w_0} \frac{1}{||\mathbf{w}||} \min_i [t_i f(\mathbf{x}_i)] \tag{2.38}$$

This problem can be posed as the following constrained optimization problem:

$$\arg\min_{\mathbf{w},w_0} ||w||^2 \tag{2.39}$$

subject to the constraints:

$$\forall i,\ t_i f(\mathbf{x}_i) \geq \kappa \tag{2.40}$$

The above constraint ensures that the margin of each data point is at least κ. The above equation can be simplified by rescaling it with division with κ without loss of generality:

$$\forall i,\ t_i f(\mathbf{x}_i) \geq 1 \tag{2.41}$$

The solution to the above constrained optimization problem is infeasible when the data points are not linearly separable. Thus, we can loose the above constraints by introducing a slack variable ξ which is 0 for correctly classified training data points and $|t_i - f(\mathbf{x}_i)|$ for the rest. In this case, we solve for the following constrained optimization problem:

$$\arg\min_{\mathbf{w},w_0} \lambda \sum_i \xi_i + ||w||^2 \tag{2.42}$$

which is equivalent to:

$$\arg\min_{\mathbf{w},w_0} \lambda \sum_i |1 - f(\mathbf{x}_i)t_i| + ||w||^2 \tag{2.43}$$

subject to the constraints:

$$\forall i,\ t_i(\mathbf{w}^T \cdot \mathbf{x}_i + w_0) >= 1 - \xi_i \tag{2.44}$$

where λ is a regularization parameter.

The above formulation can be extended by performing the kernel trick and replacing x_i with $\phi(x_i)$ where ϕ is a basis function, as explained in Section 2.2.2.

A limitation of support vector machines is that they do not provide posterior probabilities for their outputs. In addition, the parameter λ need to be cross-validated for each case. These limitations can be overcome with the Relevance Vector Machine framework [141].

The support and relevance vector machines can be also applied to multiclass classification problems. This is usually done by constructing C vector machines (where C is the number of classes), where for each of them, we attempt to discriminate one class from the rest. This strategy is called one-versus-the-rest and has the disadvantage that discriminating each class from the others is posed as different optimization problems that do not share any common terms or features. I show results for the mesh labeling problem in Figure 2.4, where SVMs are trained using the one-versus-the-rest strategy. The validation meshes are used to estimate the λ regularization parameter using the L-BFGS numerical optimization technique provided by Matlab.

2.3.2 Probabilistic Generative Models

The generative models aim at modeling the class-conditional distribution $p(\mathbf{x}|t=c)$, as well as the class priors $p(t=c)$, then compute posterior probabilities using the Bayes' theorem: $p(t=c|\mathbf{x}) = \frac{p(\mathbf{x}|t=c)p(t=c)}{p(\mathbf{x})}$. This can be rewritten as follows:

$$p(t=c|\mathbf{x}) = \frac{p(\mathbf{x}|t=c)p(t=c)}{\sum_j p(\mathbf{x}|t=j)p(t=j)} \tag{2.45}$$

By setting $a_j = ln(p(\mathbf{x}|t=j)p(t=j))$ (where $j = 1,2,...C$), the above distribution can be rewritten as:

$$p(t=c|\mathbf{x}) = \frac{exp(a_c)}{\sum_j \exp(a_j)} \tag{2.46}$$

The term $\frac{exp(a_c)}{\sum_j \exp(a_j)}$ is known as normalized exponential and is the generalization of the sigmoid function. It is also known as softmax function, as it represents a smoothed version of the 'max' function. Intuitively, what it means is that for large positive values of a_c, the posterior is saturated close to 1 while for large negative values, it is saturated to 0. There is a sharp transition for values of a_c close to 0. This "squashing" form of the sigmoid function makes it very useful for classification tasks.

In order to find the posterior probabilities for some input $\mathbf{x}$, we have to assume some distribution for the class-conditional densities $p(\mathbf{x}|t=c)$ and set some value for the priors $p(t=c) = r_c$. It is popular to use assume a Gaussian for the class-conditional densities. In this case, the classifier is known as Gaussian Bayes classifier:

$$p(\mathbf{x}|t=c) = \frac{1}{(2\pi)^{D/2}|\mathbf{S}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}_i - \mathbf{m}_c)^T \cdot \mathbf{S}^{-1} \cdot (\mathbf{x}_i - \mathbf{m}_c)\right) \tag{2.47}$$

where D is the dimension of the feature vector $\mathbf{x}$. Let us transform the training pairs into $\{\mathbf{x}_i, t_{ic}\}$, where $t_{ic} = 1$ if $t_i = c$ and $t_{ic} = 0$ if $t_i \neq c$. In this case the likelihood function is:

$$p(\mathbf{t}|\mathbf{r},\mathbf{m},S) = \prod_{i=1}^{N}\prod_{c=1}^{C} [\mathbf{r}_j \mathcal{N}(\mathbf{x}_i|\mathbf{m}_c,S)]^{t_{ic}} \tag{2.48}$$

Using maximum-likelihood (i.e., setting the derivative of the log likelihood with respect to each of the parameters), we find that:

$$\begin{aligned} r_c &= \tfrac{N_c}{N} \\ m_c &= \tfrac{1}{N_c} \sum_{i=1}^{N} t_{ic}\mathbf{x}_i \\ S_c &= \tfrac{1}{N} \sum_{c=1}^{C} \sum_{i:\{t_{ic}=1\}} (\mathbf{x}_i - \mathbf{m}_c)(\mathbf{x}_i - \mathbf{m}_c)^T \end{aligned} \tag{2.49}$$

where N_c is the number of samples that have $t_{ic} = 1$.

A problem using maximum-likelihood is that when the number of samples N_c is small for a class, the prior $r_c = \frac{N_c}{N}$ will be very small for this class. This results in very biased estimates of the parameters. In this case, we need to incorporate a prior belief to modulate the priors:

$$r'_c = \frac{N_c + \mu}{N + C\mu} \tag{2.50}$$

where μ is the number of "pseudo-counts" used to smooth the priors. Also, the covariance matrices S_c can be smoothed:

$$S'_c = (1-\lambda)S_c + \lambda S \tag{2.51}$$

where λ is a regularization parameter and S is the covariance matrix for all samples.

A problem with the Gaussian Bayes classifier is that the assumption that we have a Gaussian distribution assigned to each of the classes is not appropriate when there are outliers. Also, the distribution of the samples per class might not be Gaussian or in general can be poorly approximated with any analytically defined distribution. I show the results of using the Gaussian Bayes classifier for the mesh labeling problem in Figure 2.4. The validation meshes are used for estimating μ and λ in this case. The classification results are relatively reasonable for the humans, but not as good for the animals case.

2.3.3 Probabilistic Discriminative Models

The discriminative models attempt to directly maximize the likelihood of the conditional distribution $p(t = c|\mathbf{x})$. The advantage of such approach is that it does not need to explicitly model the distribution of the samples per class, and this can improve the generalization performance. On the other hand, discriminative approaches typically require more training examples per class; for scarce training data, generative models are more appropriate since they model the input instead. For the same reason, generative models also exhibit lower variance of parameter estimation, at the expense of possibly introducing biased estimators.

As we saw in generative models (Equation eq. 2.46, the posterior probabilities can be expressed using a softmax transformation involving functions of the input features $\mathbf{x}$:

$$p(t = c|\mathbf{x}) = \frac{exp(a_c)}{\sum_j \exp(a_j)} \tag{2.52}$$

The goal of discriminative models is to directly maximize the likelihood of the conditional distribution $p(t = c|\mathbf{x})$. Let us assume that a_k are given by linear functions on the features: $a_c = \mathbf{w}_c\mathbf{x}$. By performing the kernel trick, we can also replace $\mathbf{x}$ with basis functions, so that $a_c = \mathbf{w}_c\phi$.

Our goal now is to determine the parameters $\mathbf{w}$ using directly maximum likelihood on this model. This approach is known as "logistic regression", as we essentially attempt to fit a sigmoid function for each class. The likelihood function is given by:

$$p(\mathbf{t}|\mathbf{w}) = \prod_{i=1}^{N}\prod_{c=1}^{C} p(t = c|\phi)^{t_{ic}} \tag{2.53}$$

where t_{ic} is defined as above. The negative log-likelihood is given by:

$$-lnp(\mathbf{t}|\mathbf{w}) = -\sum_{i=1}^{N}\sum_{c=1}^{C} \mathbf{t}_{ic} \ln p(t_i = c|\phi_i) \tag{2.54}$$

Due to the nonlinearity of the sigmoid function, we cannot find a closed-form solution, as we did in the case of maximum likelihood for regression. Thus, the negative log-likelihood can

be minimized using an optimization technique, for which we should also provide its analytic gradient.

Similar to the case of regression 2.2.3, we can also add a regularization term to avoid overfitting the parameters $\mathbf{w}$. Based on the ridge regression approach, we can minimize the following cost function:

$$L(\mathbf{w}) = -\sum_{i=1}^{N}\sum_{c=1}^{C} \mathbf{t}_{ic} \ln p(t_i = c|\phi_i) + \lambda \sum_{d=1}^{D} w_d^2 \tag{2.55}$$

The Lasso formulation can provide an even sparser model. In this case, we minimize:

$$L(\mathbf{w}) = -\sum_{i=1}^{N}\sum_{c=1}^{C} \mathbf{t}_{ic} \ln p(t_i = c|\phi_i) + \lambda \sum_{d=1}^{D} |w_d| \tag{2.56}$$

The parameter λ needs to be cross-validated. An alternative approach is to define a prior distribution over the weights, as in the case of Bayesian regression (section 2.2.3) and perform Bayesian inference [11].

Logistic regression is a pretty powerful approach for classification, however, estimating the parameters $\mathbf{w}$ relies on the solution of a non-linear optimization problem that could also be computationally expensive especially in very-high dimensional spaces. As in the case of regression, regularization might pull several parameters to be small instead of generating a few non-zero parameters that would better approximate the ground truth hypothesis. In this case, it might be better to proceed with an ”aggressive” feature selection technique that attempts to combines classification models of selected features. This can be achieved with the boosting techniques that we will discuss in section 2.4.

I show the results of using the Logistic Regression classifier including the Lasso regularization technique for the mesh labeling problem in Figure 2.4. The classification results are good for humans, but for animals, several parts are mislabeled (neck and tail).

2.3.4 Conditional Random Fields for Classification

In some cases, we want to classify multiple target properties that have interdependencies on their discrete values. For example, this can happen in image or mesh labeling problems when we want to classify image or mesh elements (e.g., pixels, vertices, triangles) respectively into a set of categories according to their features. One option would be to classify each element separately based on its features. However, in such structured data, assigning a label to an element strongly depends on the labels assigned to its neighbors. Even if the features are locally continuous, classifying each element independently from the others can easily yield discontinuous and noisy results especially for elements whose features lie close to decision surfaces.

A naive solution to this problem would be to find all the labels assigned to the neighbors of each element and then assign the most common label to it. However, such solution would be strongly dependent on the parameter that determines the size of the neighborhood, would not have any probabilistic interpretation and would still yield noisy boundaries.

A much better way to treat this problem is to find a probabilistic formulation such that the decision for the label assigned to an element takes into account the assignment of labels of its neighbors. Since the labels of its neighbors also depend on the labels of their neighbors and so on, this labeling problem should be solved in a global fashion, such that the labels of all the elements are jointly optimized.

When there are such probabilistic dependencies between the random variables in our problem, structured probabilistic models are more appropriate to use. In the machine learning literature, these models are usually represented by graphs; the graphs comprise of nodes representing the random variables and links which express the probabilistic relationships between the variables. As a result, the graph encodes the joint distribution over the random variables as well the factorized representation of the set of independences that hold in the joint distribution.

There are two main classes of graphical representations: Belief networks and Markov Random Fields. The Belief Networks use directed acyclic graphs to represent a factorization of the joint probability distribution of the random variables into a product of local conditional distributions. They are more useful for expressing causal relationships between random variables. The Markov Random Fields use undirected graphs that specify both a factorization and induced dependencies between the random variables. These are better suited to express constraints between random variables. For more information on Belief Networks and Markov Random Fields, see [11].

Here, we focus on a variant of Markov Random Fields (MRFs), called Conditional Random Fields (CRFs) [82] which are undirected graphical model especially suited for classification. CRFs are often used for the labeling or parsing of sequential data, such as natural language text or biological sequences and have been also used for segmentation and labeling of images in computer vision [127, 88].

Each node in a CRF graph corresponds to a random variable c_i which represents an element to be labeled and whose distribution is to be inferred, and links represent label dependencies between the random variables. Each random variable may also be conditioned upon a set of observations $\mathbf{x}$. The links form cliques in the graph that are defined as a subset of nodes such that there is a link between all pairs of nodes in the subset. A maximal clique is a clique such that it is impossible to include any other nodes in the set without it ceasing to be a clique. For example, in Figure 2.5, we show a portion of a CRF graph defined over a mesh whose faces we want to label according to some underlying features $\mathbf{x}$. The nodes c_i represent mesh faces that are connected for adjacent faces. The labels of mesh faces are conditioned upon a set of observed features bx_i on the faces. The choice of labels for adjacent faces is additionally conditioned upon a set of some other observed features bx_{ij} related to these adjacent faces. The maximal cliques are $\{c_i, c_{j1}, \mathbf{x}_{ij1}\}$, $\{c_i, c_{j2}, \mathbf{x}_{ij2}\}$, $\{c_i, c_{j3}, \mathbf{x}_{ij3}\}$, $\{c_i, \mathbf{x}_i\}$, $\{c_{j1}, \mathbf{x}_{j1}\}$, $\{c_{j2}, \mathbf{x}_{j2}\}$, $\{c_{j3}, \mathbf{x}_{j3}\}$ etc (not all nodes for the mesh faces are shown in the graph for clarity reasons).

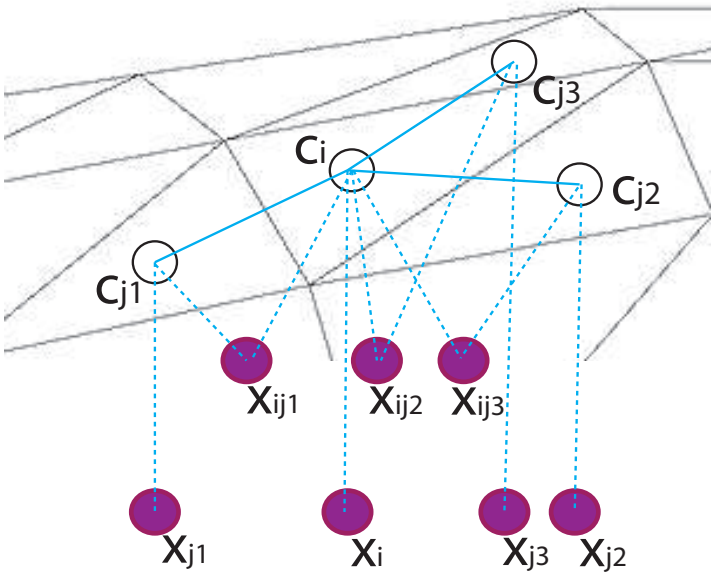


Figure 2.5: *A CRF graph defined mesh faces we wish to label according to some underlying unary features* $\mathbf{x}_i$ *and pairwise features* $\mathbf{x}_{ij}$. *The open circles represent the nodes* c_i *to be labeled. There are links for nodes that have their corresponding mesh faces adjacent. The shaded circles represent the observed features and are not generated by the CRF model*

The conditional probability of the set of labels $\mathbf{c} = \{c_1, c_2, ..., c_V\}$ is written as a product of potential functions $\psi_q(\mathbf{c}_q, \mathbf{x}_q)$ over the maximal cliques q in the graph:

$$p(\mathbf{c}|\mathbf{x}, \theta) = \frac{1}{Z(\mathbf{x}, \theta)} \prod_{q \in \mathscr{C}} \psi_q(\mathbf{c}_q, \mathbf{x}_q) \tag{2.57}$$

where θ are the parameters governing the conditional distribution $p(\mathbf{c}|\mathbf{x}, \theta)$ and $Z(\mathbf{x}, \theta)$ is called the partition function, which ensures that the conditional distribution is correctly normalized:

$$Z(\mathbf{x}, \theta) = \sum_{\mathbf{c}} \prod_{q \in \mathscr{C}} \psi_q(\mathbf{c}_q, \mathbf{x}_q) \tag{2.58}$$

Note that the potential functions are not restricted to have a specific probabilistic interpretation. Thus, they are not necessarily probability functions. In order to ensure that $p(\mathbf{c}|\mathbf{x}, \theta) > 0$, we are restricted to potential functions that are strictly positive, thus it is convenient to express them as exponentials $\psi_q(\mathbf{c}_q, \mathbf{x}_q) = exp(-E(\mathbf{c}_q, \mathbf{x}_q)$, where $E(\mathbf{c}_q, \mathbf{x}_q)$ is called an energy function. The energy functions play the role of expressing which labeling settings of the random

variables in a clique are preferred to others. In the example of Figure 2.5, the conditional probability can be rewritten as:

$$p(\mathbf{c}|\mathbf{x},\theta) = \frac{1}{Z(\mathbf{x},\theta)} \exp\left(\sum_{i=1}^{v} E_1(c_i,\mathbf{x}_i) + \sum_{i=1}^{v}\sum_{j=1}^{N(i)} E_2(c_i,c_j,\mathbf{x}_{ij})\right) \tag{2.59}$$

where $N(i)$ is the set of faces that are adjacent to the face with index i. The term E_1 measures consistency between the features $\mathbf{x}_i$ of mesh face i and its label c_i. Such terms involving one random variable and the related observations are usually called unary terms. The term E_2 measures consistency between adjacent face labels c_i and c_j, given features $\mathbf{x}_{ij}$. These terms involving two linked random variables are usually called pairwise terms. In this thesis, we will show applications of CRFs of the above form. Examples of applying CRFs for labeling problems are shown later in the thesis (Figures 3.2 and 4.3).

The main limitation of CRFs (and similarly MRFs in general) is that if we have V nodes for labeling, and each of them have C possible labels, then the evaluation of the normalization term requires summing over all C^V possible assignments, as it can been seen in Equation 2.57. Thus, inferring the most probable assignment of labels for general graphs (including the example graph of Figure 2.5) is a $\#P$-complete problem, and thus computationally intractable in the general case. If the graph has a simple structure as e.g. in linear-chain CRFs, where each node is linked to exactly one other node, the forward-backward algorithm can be used for inference which has polynomial time complexity. For general graphs, approximate inference techniques are used, such as variational inference, loopy belief propagation and Monte Carlo sampling methods and graph-cuts. Detailed discussion of these are beyond the scope of this thesis.

Learning the parameters θ in CRFs based on training data $\mathbf{x}_i, \mathbf{c}_i$ involves maximizing the log-likelihood for the CRF which is expressed as follows:

$$L(\theta) = \sum_{i=1}^{N} \ln \frac{1}{Z(\mathbf{x}_i,\theta)} + \sum_{q\in\mathscr{C}} E(\mathbf{c}_{iq},\mathbf{x}_{iq})) \tag{2.60}$$

For general graphs, it is also not possible to analytically determine the parameter values that maximize the log-likelihood. Setting the gradient to zero does not yield a closed form solution.

Parameter estimation can be performed approximately by using approximate techniques such as pseudo-likelihood, piecewise training [136] or contrastive divergence [48].

2.4 Boosting techniques

Boosting is a class of techniques for combining multiple ”base” classifiers to produce predictions that are better than any of the ”base” classifiers. Even if the ”base” classifiers perform slightly better than random guesses, boosting can still combine them appropriately to yield much better classification performance. The ”base” classifiers are called ”weak learners” and their resulting combination is usually called ”strong learner”. The weak learners can be any classifiers, however, most commonly, it is sufficient to use simple ones, such as decision stumps or decision trees. Most commonly, these simple classifiers may also use only one of the features (dimensions) of the input vector $\mathbf{x}$. As a result, boosting combines weak learners that select specific features, that are more relevant for the classification task. This kind of feature selection makes boosting ideal for handling very high-dimensional input spaces, when only a few dimensions are relevant for each task.

The key idea of boosting is to train the ”base” classifiers in sequence, where each ”base” classifier is trained using a weighted form of the training dataset. At each boosting iteration, one ”base” classifier is selected that yields the best classification performance among all other ”base” classifiers based on the current weighted form of the training dataset. Then, the training data points are re-weighted so that each weight associated with each data point depends on the performance of the previously selected classifier. Misclassified data points are associated with higher weight, so that subsequent classifiers have the chance to classify it correctly.

Boosting was originally developed for classification tasks by Freund and Schapire [34]. Later, it was also extended to solve regression problems, where in this case, the ”weak learners” are

simple regressors that are combined to yield a much better approximating regression function. Here, I will briefly refer to the most popular version of boosting, called AdaBoost (Adaptive Boosting), and then I will focus on multiclass classification with JointBoost [142]. Then, I will refer to a boosting algorithm for regression [155].

2.4.1 Adaboost

The original version of Adaboost deals with binary classification problems. Given N training samples $\{\mathbf{x}_i, t_i\}$, where $t_i = \{-1, 1\}$, we associate a weight w_i for each sample, which is initially set to $1/N$. Then, at each boosting iteration $m = 1, 2, ..., M$, we perform the following steps:

- select a base classifier $h_m(x)$ which best minimizes the weighted classification error function: $J_m = \sum_{i=1}^{N} w_i I(h_m(\mathbf{x}_i) \neq t_i)$ where $I(h_m(\mathbf{x}_i) \neq t_i)$ is an indicator function that equals 1 if $h_m(\mathbf{x}_i) \neq t_i$ and 0 otherwise.
- evaluate the normalized error of the weak learner: $\varepsilon_m = \frac{\sum_{i=1}^{N} w_i I(h_m(\mathbf{x}_i) \neq t_i)}{\sum_{i=1}^{N} w_i}$ and then $\alpha_m = \ln(1 - \varepsilon_m)/\varepsilon_m$
- update the weights: $w_i = w_i \exp(\alpha_m I(h_m(\mathbf{x}_i) \neq t_i)$ and normalize them so that they sum to 1.

Finally, the predictions are done by linearly combining the predictions of the weak classifiers:

$$H(\mathbf{x}_i) = sign(\sum_{m=1}^{M} \alpha_m h_m(\mathbf{x})) \tag{2.61}$$

The choice of the above weights on the samples w_i and weak learners α_m were motivated by findings in the statistical learning theory. It can be proved that if there is a bounded probability that the error of the weak learners is less than 50%, then there is a bound on the training and generalization error. More specifically, suppose that $Pr(\varepsilon_m < 0.5 - \gamma_m) < \delta$, where γ_m is a constant that measures how much better than random the predictions of the weak learner m are, and δ is a constant the bounds this probability. Then, it can be proved that training error

$E \leq exp(-2\sum_{m=1}^{M}\gamma_m^2)$. Practically, this means that if the weak learner is slightly better than random, then the training error drops exponentially fast. There is also an upper bound for the generalization error $\mathscr{E} \leq E + O(\sqrt{(M \cdot V/N)}$, where M is the number of boosting iterations, N is the number of samples and V is the VC-dimension of the weak learner. The VC-dimension corresponds to the largest set of data points in the feature space that can be split by the weak classifier in any possible labels assignment and arrangement of them. This bound is rather loose and practically may not be that useful.

A different interpretation of Adaboost was given by Friedman *et al.* [35]. It can be proved that based on the above formulation, Adaboost minimizes an exponential error function corresponding to the classification error:

$$E = \sum_{n=1}^{N} \exp\left(-t_i h_m(\mathbf{x}_i)\right) \tag{2.62}$$

The exponential error function heavily penalizes data points that are misclassified. There are other variants of Adaboost that can be used to minimize the above exponential error function. A particular variant that is more numerically stable is GentleBoost [35]. Gentleboost performs each boosting iteration, as follows:

- select a base classifier $h_m(x)$ to the training data which best minimizes the weighted classification error function.
- update the weights: $w_i = w_i \exp\left(-h_m(\mathbf{x}_i)t_i\right)$ and normalize them so that they sum to 1.

The output of the strong learner is given by:

$$H(\mathbf{x}_i) = sign(\sum_{m=1}^{M} h_m(\mathbf{x})) \tag{2.63}$$

Both Adaboost and GentleBoost have been extended for multiclass classification problems. Below, we focus on perhaps one of the most powerful versions for multiclass classification, called JointBoost.

2.4.2 JointBoost

Jointboost was introduced by Torralba *et al.* [142] and its main characteristic is its ability to deal with multiple overlapping subsets of classes in the feature space. For example, in the case of part labeling of a hand, imagine that a feature (e.g., shape diameter) is excellent for distinguishing the index, middle, and ring fingers from the palm and the thumb, but not for distinguishing the index from the ring fingers, since they may have approximately the same values for this feature (e.g., shape diameter). Many multiclass classifiers (e.g., SVMs) adopt a one-against-the-rest strategy, which attempts to separate each class from all the rest. Such classifiers would not be able to benefit from this feature in this case, since its values are not clearly discriminating all classes. In fact, there might be no class-specific features at all. JointBoost can exploit this feature to distinguish these fingers first, and then in the next rounds of boosting, other features can be selected to further discriminate them. This process of finding commonalities between classes substantially improves the generalization error as shown by Torralba *et al.* [142].

The classifier is composed of *decision stumps*. A decision stump is a very simple classifier that scores each possible class label c, given the feature vector $\mathbf{x}$, based only on thresholding its f-th entry x_f. A JointBoost decision stump can be written as:

$$h(\mathbf{x}, c; \phi) = \begin{cases} a & x_f > \tau \text{ and } c \in \mathscr{C}_S \\ b & x_f \leq \tau \text{ and } c \in \mathscr{C}_S \\ k_c & c \notin \mathscr{C}_S \end{cases} \tag{2.64}$$

In other words, each decision stump stores a set of classes $\mathscr{C}_S$. If $c \in \mathscr{C}_S$, then the stump compares x_f against a threshold τ, and returns a constant a if $x_f > \tau$, and another constant b otherwise. If $c \notin \mathscr{C}_S$, then the comparison is ignored; instead, a constant k_c is returned instead. There is one k_c for each $c \notin \mathscr{C}_S$. The parameters ϕ of a single decision stump are f, a, b, τ, the set $\mathscr{C}_S$, and k_c for each $c \notin \mathscr{C}_S$.

The probability of a given class c is then computed by summing the decision stumps and then performing the softmax transformation:

$$H(\mathbf{x}, c) = \sum_m h_m(\mathbf{x}, c; \phi_m) \tag{2.65}$$

$$P(c|\mathbf{x}) = \frac{\exp(H(\mathbf{x}, c))}{\sum_{j=1}^{C} \exp(H(\mathbf{x}, j))} \tag{2.66}$$

Given N training pairs $(\mathbf{x}_i, t_i)$, JointBoost minimizes the weighted multiclass exponential loss over the training set:

$$J = \sum_{i=1}^{N} \sum_{c \in \mathscr{C}} w_{i,c} \exp\left(-I(t_i, c)\, H(\mathbf{x}_i, c)\right) \tag{2.67}$$

where each training pair is assigned a per-class weight $w_{i,c}$, $H(\mathbf{z}, l)$ is defined in Equation 2.65, $\mathscr{C}$ is the set of possible class labels, and $I(t_i, c)$ is an indicator function that is 1 when $t_i = c$ and -1 otherwise.

The algorithm proceeds iteratively as Adaboost. The algorithm stores a set of weights $\tilde{w}_{i,c}$ that are initialized to the weights $w_{i,c}$, representing the confidence for each sample. Then, at each iteration, one decision stump (Equation 2.64) is added to the classifier. The parameters ϕ_m of the stump at iteration m are computed to optimize the following weighted least-squares objective:

$$J_{wse}(\phi_m) = \sum_{c \in \mathscr{C}} \sum_{i=1}^{N} \tilde{w}_{i,c} (I(t_i, c) - h_m(\mathbf{x}_i, c; \phi_m))^2 \tag{2.68}$$

where $\mathscr{C}$ are the possible class labels. The optimal a, b, k_c are computed in closed-form, and $f, \tau, \mathscr{C}_S$ are computed by brute-force. When the number of labels $|\mathscr{C}|$ is large, then a greedy heuristic search can be used for $\mathscr{C}_S$ [142]. Once the parameters ϕ_m are determined, the weights are updated as:

$$\tilde{w}_{i,c} \leftarrow \tilde{w}_{i,c} \exp(-I(t_i, c)\, h(\mathbf{x}_i, c; \phi_m)) \tag{2.69}$$

and the algorithm continues with the next decision stump.

The complexity of JointBoost is $O(|\mathscr{C}|^2 \cdot N \cdot D \cdot T)$, if greedy search is used for finding $\mathscr{C}_S$, and $O((2^{|\mathscr{C}|}) \cdot N \cdot D \cdot T)$ if brute force search is used, where $|\mathscr{C}|$ is the number of labels, N is the

number of training samples, D is the number of dimensions of $\mathbf{x}$, T is the number of boosting rounds.

We show results of applying the JointBoost algorithm for the mesh labeling problem in Figure 2.4. The validation meshes are used to determine when to stop the boosting iterations; at each iteration, the classification error is measured on the validation meshes. We stop at the iteration where the validation error is minimized. Jointboost has reasonable results in both the humans and animals example. However, I have to emphasize that by no means Jointboost or boosting are the best classification techniques for any task in general. In the problems presented in this thesis, boosting was an appropriate choice, because we are dealing with very high-dimensional input feature spaces, where only a few different features might be relevant for each task. Boosting also offers fast sequential learning algorithms that was also important in our problems, since we were dealing with large training datasets. Jointboost also produced output probabilities suitable for combination with other terms in the Conditional Random Fields models that we used in our problems.

However, boosting also has a number of limitations. The greedy weak learner selection strategy may not always yield the optimal set of features that are relevant for a task. The exponential cost function that boosting mainly deals with, is not robust to outliers. The number of boosting iterations is a parameter that is usually user-adjusted. Running too many boosting iterations might result in overfitting, although Adaboost has been shown to have very good generalization performance in many applications. Alternatively, the boosting iterations can be terminated when the error measured in a validation set reaches to a minimum. However, using a validation set prevents us from using the whole training dataset and the generalization performance might strongly depend on the specific selection of the validation set.

2.4.3 Boosting for regression

Adaboost has been extended for regression as well. If we consider a sum-of-squares error function for regression, then the iterative minimization of an additive model of the form 2.61 simply involves fitting each weak learner to the residual errors $t_i - f_{m-1}(\mathbf{x})$ of the previous model [35]. Here, the weak learner can be a simple regression function involving one of the input features, as also in the case of boosting for classification.

Here we will focus on a technique, that aims at approximating the target property itself rather than the residuals at each boosting iteration, since it is better for learning complex target properties and is less prone to overfitting. This technique was introduced by Zemel and Pitassi [155] and is known as gradient-based boosting for regression. the gradient-based boosting technique aims at learning an additive model of the following form to approximate a target property:

$$F(\mathbf{x}) = \sum_{m} \alpha_m \phi_m(\mathbf{x}) \tag{2.70}$$

where the functions $\phi_m(\mathbf{x})$ are the weak learners and α_m are their corresponding weights. The functions $\phi_m(\mathbf{x})$ can be selected to be linear functions of single features, as in the case of boosting for classification with decision stumps.

Given N training pairs $\{\mathbf{x}_i, t_i\}, i = \{1, 2, ..., N\}$, where t_i are exemplar values of the target property, the gradient-based boosting algorithm attempts to minimize the average error of the weak learners with respect to the weight vector $\mathbf{m}$:

$$L(\mathbf{r}) = \sum_{i=1}^{N} \Big(\prod_{m=1}^{M} {r_m}^{-0.5} \Big) \exp\Big(\sum_{m=1}^{M} r_m \cdot (t_i - \phi_m(\mathbf{x}_i))^2 \Big) \tag{2.71}$$

This objective function is minimized iteratively by updating a set of weights $\{w_i\}$ on the training samples. The weights are initialized to be uniform i.e. $w_i = 1/N$, unless there is a prior confidence on each sample. In this case, the weights can be initialized according to this confidence. Then, we initiate the boosting iterations that have the following steps:

- for each weak learner, we minimize the following function:

$$L_m = \sum_{i=1}^{N} w_i (r_m^{-0.5} \exp\left(r_m (t_i - \phi_m(\mathbf{x}_i))\right)^2) \tag{2.72}$$

  with respect to r_m as well as the parameters of the weak learner functions. The parameter r_m are optimized with line search. For the first boosting iteration $m = 1$, we set $r_m = 1$ always.

- we select the weak learner that yields the lowest value for L_m.
- we update the weights on the training pairs:

$$w_i = w_i \cdot r_m^{-0.5} \exp\left(r_m (t_i - \phi_m(\mathbf{x}_i)\right)^2 \tag{2.73}$$

- we normalize $w_i = w_i / \sum_i w_i$ so that they sum to 1.

Finally, we normalize the weights $r_m = r_m / \sum_k r_m$ so that they sum to 1. The final prediction is given by Eq. 2.70. The number of boosting iterations can be given as a parameter or we can measure the hold-out validation error at each round and terminate boosting when it reaches a minimum.

As in the case of boosting for classification, boosting for regression has the same limitations. The greedy weak learner strategy might be suboptimal and the exponential cost function (Equation 2.71) is not robust to outliers. Terminating the boosting iterations using hold-out validation error may also be suboptimal and is not using the whole training dataset.

2.5 Dimensionality Reduction

For many problems, the input data **x** or target property data **t** may be very high-dimensional. On the other hand, our data points may lie close to a manifold of much lower dimensionality than our original data space. For example, consider the case of a vertex attribute on an animated mesh. The attribute data are NxD matrices, where N is the number of animation frames

and D is the number of vertices on the mesh. The vertex attribute may have significant spatial correlations on the mesh during the animation e.g., the position, the normal or the high-order derivatives of a vertex of nearby vertices are significantly correlated. Thus, we can apply a transformation to project the vertex attribute data to a lower subspace of much lower dimensionality $M << D$. As a result, we can now perform processing of this reduced data in this subspace. For example, if we want to predict surface curvature from animation parameters, instead of learning a function for each vertex, we can learn much fewer functions for the components of this subspace. This results in more compact models that can also be evaluated more efficiently during runtime. We can re-project the reduced data back to the original space by applying the inverse transformation. There are many other scenarios, where dimensionality reduction can be useful and has been used extensively. For skeletal-based mesh animations, the vertices positions are heavily correlated spatially when they are displaced according to the rotations of joints, thus dimensionality reduction can be used to avoid computing coordinates of each single vertex [149]. Dimensionality reduction can be also applied to the space of displacement fields that move the vertices from a reference pose due to their redundancy e.g., skin bulging in similar directions [2, 80, 121]. Similarly, dimensionality reduction is employed to reduce the space the state space parameterization of deformable shapes and also find low-rank approximations of diffuse radiance transfer for low-frequency lighting [61, 104, 102].

The most common technique for dimensionality reduction in computer graphics is Principal Component Analysis. PCA can be defined as the orthogonal projection of the data onto a linear subspace, known as principal subspace, such that the variance of the projected data is maximized. Alternatively, PCA can be defined as the linear projection that minimizes the squared error between the original data points and their projections. In both cases, the solution is the same. Given a training data set $\mathbf{Y}$, first, we subtract its mean $\overline{\mathbf{Y}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{Y}_i$ from it. Then we compute the eigenvectors of the covariance matrix $\mathbf{S}_{YY} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{Y}_i - \overline{\mathbf{Y}})(\mathbf{Y}_i - \overline{\mathbf{Y}})^T$ and retain a subset $\mathbf{V}$ of them corresponding to the largest eigenvalues. The columns of this subset are orthonormal basis vectors. The data $\mathbf{Y}$ are projected to their linear subspace using:

$\mathbf{Y}_S = \mathbf{V}^T\mathbf{Y}$.

Here, we will present PCA from a probabilistic point of view. We will show that PCA gives the maximum likelihood solution to a particular form of linear Gaussian latent variable model (Section 2.5.1). Then, we will see another dimensionality reduction technique, called Independent Component Analysis, which is based on the completely different assumption that the latent variable model is non-Gaussian. Finally, we will briefly refer to non-linear dimensionality reduction techniques.

2.5.1 Principal Component Analysis

Let $\mathbf{y}$ be the $D-$dimensional variable whose dimensionality we wish to reduce. Let $\mathbf{z}$ be the unknown $M-$dimensional latent variable with $M << D$ onto which $\mathbf{y}$ is projected. We assume that the variable $\mathbf{z}$ is Gaussian-distributed, has zero mean and unit covariance: $p(\mathbf{z}) = \mathcal{N}(0, \mathbf{I})$. Based on the PCA model, we wish to find a linear transformation that maps from $\mathbf{z}$ to $\mathbf{y}$ plus noise:

$$\mathbf{y} = W\mathbf{z} + \mu + \varepsilon \tag{2.74}$$

where W is the unknown DxM re-projection matrix, ε is a $D-$dimensional zero-mean Gaussian distributed noise variable with covariance $\sigma^2\mathbf{I}$. This corresponds to a linear-Gaussian model, thus the distribution of $\mathbf{y}$ is also Gaussian with the same mean:

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\mu, \mathbf{C}) \tag{2.75}$$

The covariance matrix C can be expressed in terms of $\mathbf{W}$ as follows:

$$C = E[(\mathbf{W}\mathbf{z} + \varepsilon)(\mathbf{W}\mathbf{z} + \varepsilon)^T] = E[\mathbf{W}\mathbf{z}\mathbf{z}^T\mathbf{W}^T] + E[\varepsilon\varepsilon^T] = \mathbf{W}\mathbf{W}^T + \sigma^2\mathbf{I} \tag{2.76}$$

(E here represents expectation). The posterior distribution $p(\mathbf{z}|\mathbf{y})$ will have the following form [11]:

$$p(\mathbf{z}|\mathbf{y}) = \mathcal{N}(\mathbf{z}|(\mathbf{W}^T\mathbf{W} + \sigma^2\mathbf{I})^{-1}\mathbf{W}^T(\mathbf{y} - \mu), \sigma^{-2}\mathbf{W}^T\mathbf{W} + \mathbf{I}) \tag{2.77}$$

We wish now to estimate $\mathbf{W}, \mu, \sigma$ with maximum-likelihood. Given a training data set $\mathbf{Y}$, the corresponding log-likelihood is given as:

$$lnp(\mathbf{Y}|\mu,\mathbf{W},\sigma^2) = \sum_{i=1}^{N} lnp(\mathbf{y}_i|\mathbf{W},\mu,\sigma^2) = -\frac{ND}{2}ln(2\pi) - \frac{N}{2}\ln det(\mathbf{C}) - \frac{1}{2}\sum_{i=1}^{N}(\mathbf{Y}_i-\mu)^T C^{-1}(\mathbf{Y}_i-\mu) \tag{2.78}$$

Setting the derivative of log-likelihood, the following solutions for μ, σ^2, and $\mathbf{W}$ can be found:

$$\begin{aligned} \mu_{ML} &= \bar{\mathbf{Y}} = \tfrac{1}{N}\textstyle\sum_{i=1}^{N}\mathbf{Y}_i \\ \sigma^2_{ML} &= \tfrac{1}{D-M}\displaystyle\sum_{i=M+1}^{D}\lambda_i \\ \mathbf{W}_{ML} &= \mathbf{U}\left(\mathbf{L}-\sigma^2\mathbf{I}\right)^{1/2}\mathbf{R} \end{aligned} \tag{2.79}$$

where U is a DxM matrix which is any subset (of size M) of the eigenvectors of the covariance matrix $\mathbf{S}_{YY}$ of $\mathbf{Y}$, λ_i are its corresponding eigenvalues, L is a MxM diagonal matrix that has the eigenvalues λ_i as diagonal elements, and R is an arbitrary rotation matrix. This means that the maximum-likelihood estimation of W is uniquely defined up to a rotation i.e., the distribution of $\mathbf{y}$ is left unchanged if we apply rotations to the latent space, since the distribution of the latent variable $\mathbf{z}$ is isotropic Gaussian. In the classical PCA, it is assumed that $\mathbf{R} = \mathbf{I}$. In this case, the columns of $\mathbf{W}_{ML}$ are scaled versions of the eigenvectors of the covariance matrix S (known as principal components). The maximum of the likelihood is obtain by selecting the M eigenvectors corresponding to the M largest λ_i eigenvalues of the covariance matrix S.

From the above, in order to project the $\mathbf{y}$ to the corresponding linear subspace based on PCA, we find the mean of the posterior of Eq. 2.77:

$$E[\mathbf{z}|\mathbf{y}] = (\mathbf{W}_{ML}^T\mathbf{W}_{ML} + \sigma^2\mathbf{I})^{-1}W_{ML}^T(\mathbf{y}-\bar{\mathbf{Y}}) \tag{2.80}$$

In the special case where we assume $\sigma^2 \to 0$ i.e., there is no noise in Equation 2.74, then the posterior mean is simplified as follows:

$$E[\mathbf{z}|\mathbf{y}] = (\mathbf{W}_{ML}^T\mathbf{W}_{ML})^{-1}W_{ML}^T(\mathbf{y}-\bar{\mathbf{Y}}) \tag{2.81}$$

If we compute W based on the eigendecomposition of S which results in orthogonal columns for $\mathbf{W}$, then $E[\mathbf{z}|\mathbf{y}] = W_{ML}^T(\mathbf{y} - \bar{\mathbf{Y}})$, which is the result of the classical PCA (assuming an orthogonal W also results in much faster projections since there is much less computation involved). Thus, classical PCA is a special case of this probabilistic formulation. The probabilistic formulation of PCA is also called Probabilistic PCA (PPCA) and allows to handle missing values in the training data. It also allows to compute the principal components using other algorithms (such as Expectation-Maximization), that are more efficient than performing eigendecomposition in the covariance matrix (or alternatively, Singular Value Decomposition on $\mathbf{Y}$), especially when D is very large.

An issue with PCA and its probabilistic formulation is how to select M i.e., assuming that we perform eigendecomposition on the data covariance matrix, how many eigenvectors we should keep. A typical solution is to retain the first eigenvectors that correspond to a prescribed variance of the data. This threshold however assumes that we know how much percentage of the variance corresponds to noise. This might not be always possible. Another technique is to use a hold-out validation set, and check for which values of M the log-likelihood of Equation 2.78 is maximized for this set. This can be however costly and also enforces us not to use the whole training dataset. A different approach is to use a Bayesian approach for PCA, where we marginalize out the model parameters $\mathbf{W}, \mu, \sigma^2$ with respect to prior distributions. Specifically, it is common to use a Gaussian prior over each column of $\mathbf{W}$, that could also lead to a sparse solution for it. More details for performing Bayesian PCA can be found here [11].

This version of PCA has also several limitations. First, it finds the principal components of the data under the strict assumption that we have a linear Gaussian model of the form 2.74. However, there might be no appropriate subspace that is linearly related to our data and even worse, the latent variables might not follow Gaussian distributions. PCA is also based on the assumption that the high-variance principal components correspond to the interesting dynamics of the data while the low-variance ones correspond to noise. This might not be also true, especially in

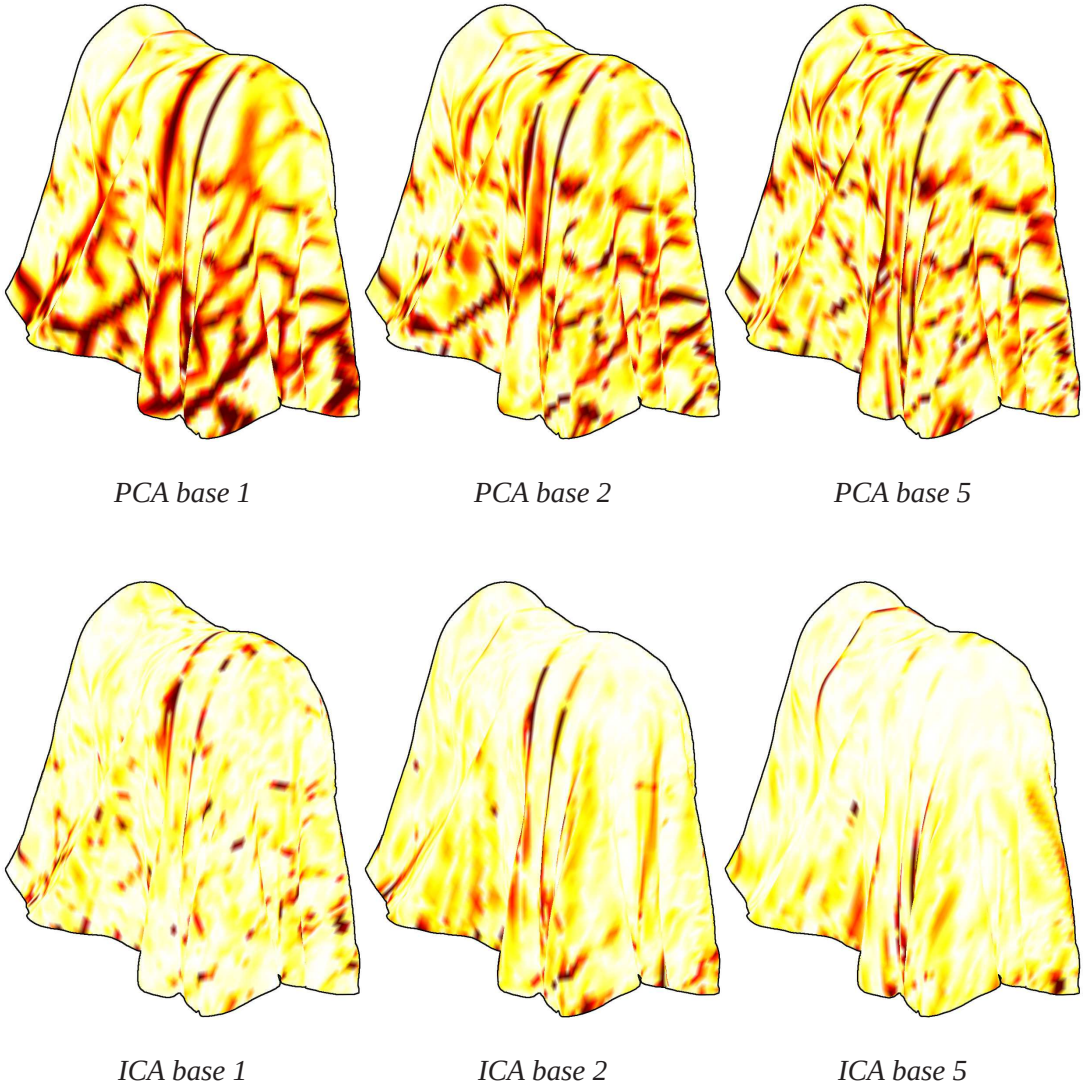


Figure 2.6: *Cloth curvature-bases found by PCA (**top**) and ICA (**bottom**). The ICA bases exhibit much greater sparsity and locality, capturing fold and wrinkle structures. Colors correspond to magnitude, with white for zero and red for the largest magnitude.*

the case where we have geometry data that have some structure e.g., they are correlated locally. For example, in the case of data-driven curvature, in Figure 2.6 we show that PCA yields basis vectors that are rather global, and does not correspond to the localized structure of curvature on a mesh. ”The main reason for this is that it chooses orthogonal basis vectors that mainly correspond to high-variance in the data. These are optimal in terms of minimizing the mean

squared error between the original data points and their projections, however this version of PCA does not impose any requirements for them to be sparse or localized.”.

2.5.2 Independent Component Analysis

As we saw above, PCA assumes models with latent variables based on linear-Gaussian distributions. This limitation lead us to search for other formulations of dimensionality reduction where the distributions of the latent variables are non-Gaussian. A particular class of such models attempts to find models of the form $\mathbf{z} = \mathbf{Qy}$ by assuming that the distributions over the latent variables factorizes so that:

$$p(\mathbf{z}) = \prod_{j=1}^{M} p(\mathbf{z}_j) \tag{2.82}$$

In this case, the latent space consists of statistically independent latent variables represented by $\mathbf{z}_j$. This class of models is also known as Independent Component Analysis. Finding such independent components is not possible with PCA that assumes a Gaussian distribution on the latent variables i.e., the PCA model cannot distinguish between two different choices of latent components that differ by a rotation in latent space. Instead, in order to estimate the independent latent components, we need to assume that they are non-gaussian. One popular way to find independent components is to find the transformation $\mathbf{z} = \mathbf{Qy}$ that maximizes non-gaussianity. There are many different ways to measure non-gaussianity. One way to measure it is by estimating the kurtosis of $\mathbf{Qy}$. The kurtosis is a statistical measure of the ”peakedness” of the probability distribution of a real-valued random variable. The kurtosis is zero for a gaussian random variable. For most (but not quite all) non-gaussian random variables, kurtosis is nonzero. Thus, the goal in this formulation would be to maximize the absolute kurtosis.

$$\arg\max_{\mathbf{Q}} \{E[\mathbf{z}^4] - 3(E[\mathbf{z}^2])^2\} \tag{2.83}$$

where E represents again expectation. In practice we would start from an initialization of $\mathbf{Q}$, compute the direction in which the kurtosis is growing most strongly (if kurtosis is positive)

or decreasing most strongly (if kurtosis is negative) based on an optimization technique. A problem with using kurtosis is that it is very sensitive to outlier samples of $\mathbf{y}$.

Another way to measure non-gaussianity is to maximize the negentropy of the distribution associated with $\mathbf{z}$, which is defined as follows:

$$J(z) = H(\mathbf{z}_{gauss}) - H(\mathbf{z}) \tag{2.84}$$

where $\mathbf{z}_{gauss}$ is the Gaussian random variable of the same covariance matrix as $\mathbf{z}$ and $H(\mathbf{z})$ is the entropy of the distribution of $\mathbf{z}$:

$$H(\mathbf{z}) = -\sum_i p(\mathbf{z}) \log p(\mathbf{z}) \tag{2.85}$$

Negentropy has the property to be always non-negative, and is zero if and only if $\mathbf{z}$ has a Gaussian distribution.

Another option is to use maximum-likelihood by assuming a specific non-Gaussian distribution on the latent components. In practice, a common choice for the latent-variable distribution is the following:

$$p(\mathbf{z}_j) = \frac{1}{\pi \cosh(\mathbf{z}_j)} = \frac{1}{\pi(e^{\mathbf{z}_j} + e^{-\mathbf{z}_j})} \tag{2.86}$$

There are many more techniques to estimate the independent components in the latent space; a tutorial can be found here [57]. In general, ICA is more appropriate to use than PCA in geometry processing applications for which we expect that the geometry signal is a linear superposition of other signals, possibly localized. It has been often noted in the literature of image processing that ICA applied to image data yields localized basis, e.g., [7, 9]. In the case of data-driven curvature, in Figure 2.6 we show that ICA yields more localized and sparse basis vectors corresponding to structure in the data, such as folds, wrinkles, and other similar structures.

ICA also has its limitations. First, depending on the specific formulation of ICA, different sets of independent components can be found. Finding is also the independent components is a

non-linear problem in contract to PCA. In practice, different techniques should be tested. In addition, in contrast to PCA, we cannot also determine the order of the independent components or their associated variance.

2.5.3 Non-linear dimensionality reduction techniques

PCA and ICA assume linear subspaces for dimensionality reduction i.e., the projected data are linearly related to the original data. However, there could be cases where the data points may lie close to a non-linear manifold of much lower dimensionality. The PCA and ICA models can be extended to learn non-linear manifolds, by applying the kernel trick on the input data points, using various kernels, such as the ones described in Section 2.2.2. This gave rise to the kernel PCA and kernel ICA models. Kernel-based techniques require much more training data to reliably learn a non-linear manifold and require significant more computation. Fortunately, in many cases, much of the computation may not need to be performed in feature space $\phi(\mathbf{y})$, since the kernel trick can factor away much of the computation [11].

Other techniques attempt to compute a non-linear embedding so that relationships in local neighborhoods of data-points are preserved, such as geodesic distances (Isomap [138], Curvilinear Distance Analysis ([24]). Alternatively, a graph is constructed so that neighborhood information of the data points is incorporated and then the Laplacian of the graph is used to compute a low-dimensional representation of the data (Laplacian Eigenmap [8]). The method of Locally-Linear Embedding similarly expresses each point as a linear combination of its neighbors and then an eigenvector-based optimization technique is used to find the low-dimensional embedding of points [114]. A different approach to nonlinear dimensionality reduction is through the use of autoencoders, a special kind of feed-forward neural networks [54].

A complete analysis of non-linear dimensionality techniques is beyond the scope of this thesis. More information on non-linear dimensionality reduction techniques can be found in [11].

2.6 Other learning topics

The literature in machine learning is vast and it is worth exploring many different topics and techniques for applications in geometry processing. This chapter focused on several important concepts in machine learning in regression, classification, boosting, and dimensionality reduction that are common in problems that we want to learn a mapping from a input feature space to a set of continuous or categorical properties. There are several other areas in machine learning that are beyond the scope of this thesis, such as Bayesian inference, neural networks, Bayesian networks, sampling techniques from probability distributions (e.g., Monte Carlo sampling techniques), dynamic graphical models (e.g., Hidden Markov Models), modeling of probability distributions (e.g., density estimation) to name a few. On the other hand, possible applications of techniques from these areas for geometry processing and computer graphics in general can be worth exploring in the future.

Chapter 3

Learning mesh segmentation and labeling

In this chapter, I present a machine learning approach for 3D mesh segmentation and labeling of parts [1]. Segmentation and labeling of shapes into meaningful parts is fundamental to shape understanding and processing. Numerous tasks in geometric modeling, manufacturing, animation and texturing of 3D meshes rely on their segmentation into parts. Many of these problems further require labeled segmentations, where the parts are also recognized as instances of known part types. For most of these applications, the segmentation and labeling of the input shape is manually specified. For example, to synthesize texture for a humanoid mesh, one must identify which parts should have "arm" texture, which should have "leg" texture, and so on. Even tasks such as 3D shape matching or retrieval, which do not directly require labeled-segmentations, could benefit from knowledge of constituent parts and labels. However, there has been very little research in part labeling for 3D meshes, and 3D object segmentation likewise remains an open research problem [16].

[1] The work presented in this chapter is also published in ACM Transactions on Graphics, Vol. 29, No. 3, 2010 [68]. Project web page: http://www.dgp.toronto.edu/~kalo/papers/LabelMeshes/,

Labeling of mesh parts is expressed as a problem of learning a mapping from shape-based features to segment labels for each face. The segment labels are categorical target properties, thus, this mapping is a classification problem (Section 2.3). The labels of neighboring faces are strongly correlated, thus, the labeling is expressed as a problem of optimizing a Conditional Random Field 2.3.4. The CRF objective function includes unary terms that assess the consistency of faces with labels, and pairwise terms between labels of adjacent faces. The objective function is learned from a collection of labeled training meshes. The basic terms of the CRF are learned using JointBoost classifiers 2.4.2, which automatically select from among hundreds of possible geometric features to choose those that are relevant for a particular segmentation task. Holdout validation is used to learn additional CRF parameters.

We evaluate our method on the Princeton Segmentation Benchmark, with manually-added labels. Our method yields 94% labeling accuracy, and is the first labeling method applicable to such a broad range of meshes. In segmentation, our method yields 9.5% Rand Index error, significantly better than the current state-of-the-art, with results similar to human-provided segmentations for most classes. No manual parameter tuning is required. The main limitation of our approach is that it requires a consistently-labeled training set; however, we find that, for many cases, just a few training meshes suffice to obtain high-quality results. Different segmentation tasks can be specified by providing examples of the new task, without requiring any manual parameter adjustments. Once learned, the algorithm can be applied to databases of the same type of objects to automatically segment and label them.

To date, nearly all existing mesh segmentation methods attempt segmentation without recognition. When the goal of segmentation can be formulated mathematically (e.g., partitioning into developable patches), low-level geometric cues may be sufficient. However, many tasks require some understanding of the functions or relationships of parts, which are not readily available from low-level geometric cues. It is unknown whether human-level 3D mesh segmentation is possible without the benefit of higher-level cues. It is worth noting that, in computer vision,

after decades of research on performing image segmentation alone, most work has turned to the joint segmentation and recognition of images. Furthermore, current models are learned from training data, allowing them to employ much more sophisticated models than are possible with manually-tuned models. These methods produce state-of-the-art results on several benchmark tests. Hence, it is worth asking: is part recognition useful for 3D mesh segmentation? Furthermore, can segmentation algorithms benefit from models learned from human-labeled meshes? Our work provides positive evidence for both questions.

3.1 Related work

Mesh segmentation has been a very active area of research in computer graphics. Most effort has focused on finding simple geometric criteria for segmentation of a single input mesh [93, 125, 74, 90, 73, 128, 6, 89, 40, 87, 83, 84, 55]; see [5, 123, 16] for surveys. Such approaches employ simple, interpretable geometric algorithms, but are limited to a single generic rule (e.g., concavity, skeleton topology, fitting shape primitives) or a single feature (e.g., shape diameter, curvature tensor, geodesic distances) to partition an input mesh. Our method employs many of the geometric features proposed by these methods. For many problems, different types of surfaces and different surface parts may require different features for segmentation. Because our model is learned, it can employ many different geometric features to partition the input mesh. Our algorithm learns problem-specific parameters from training examples, rather than requiring manually-tuned parameters. Furthermore, our method jointly segments and labels meshes. Simari *et al.* [129] perform segmentation and labeling jointly. However, this method requires manual definition and tuning of objective functions for each type of part, and is sensitive to local minima.

A few approaches make use of part matching for segmentation, and can transfer part labels based on the matches. Kraevoy *et al.* [79] and Shapira *et al.* [124] perform an initial segmenta-

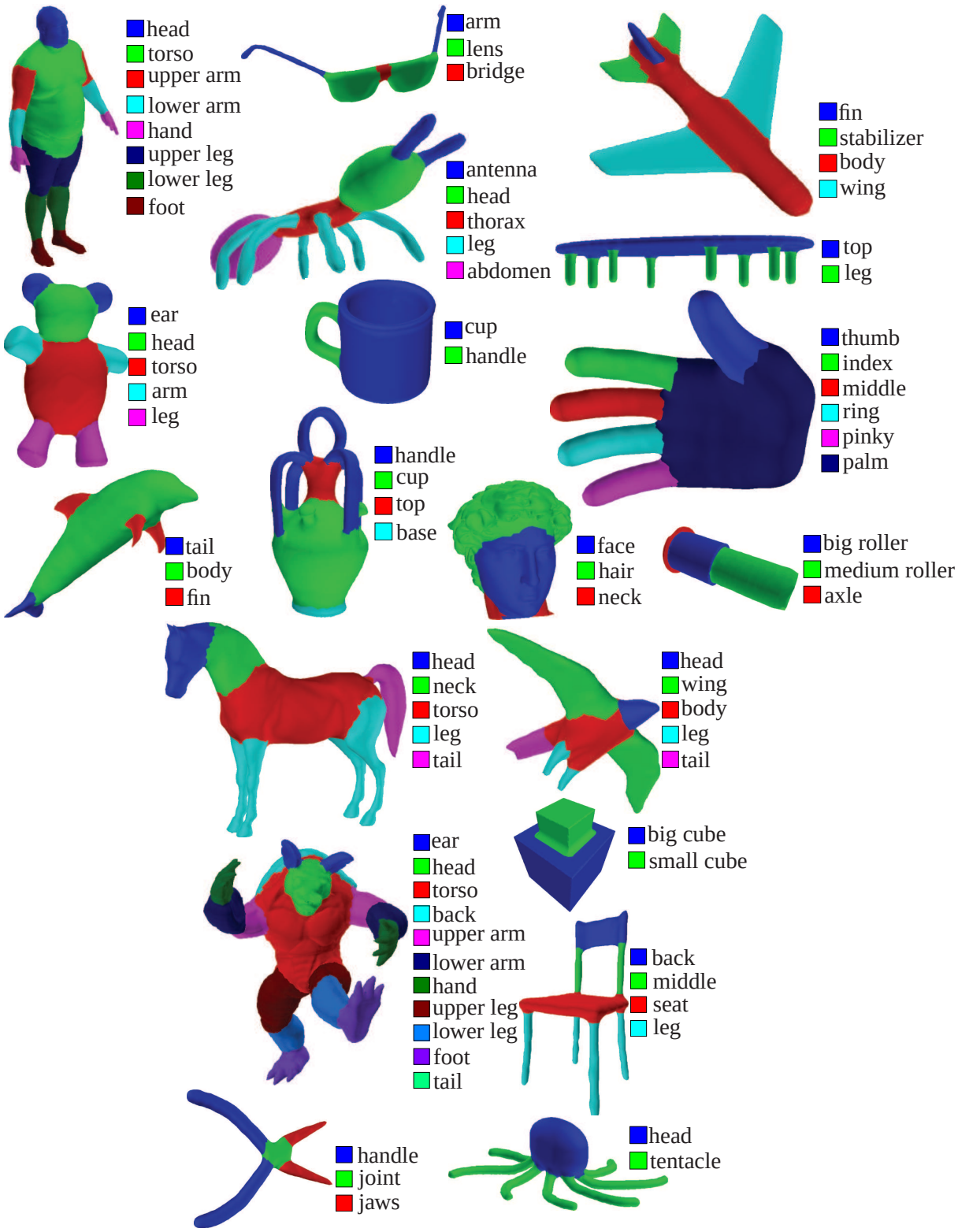
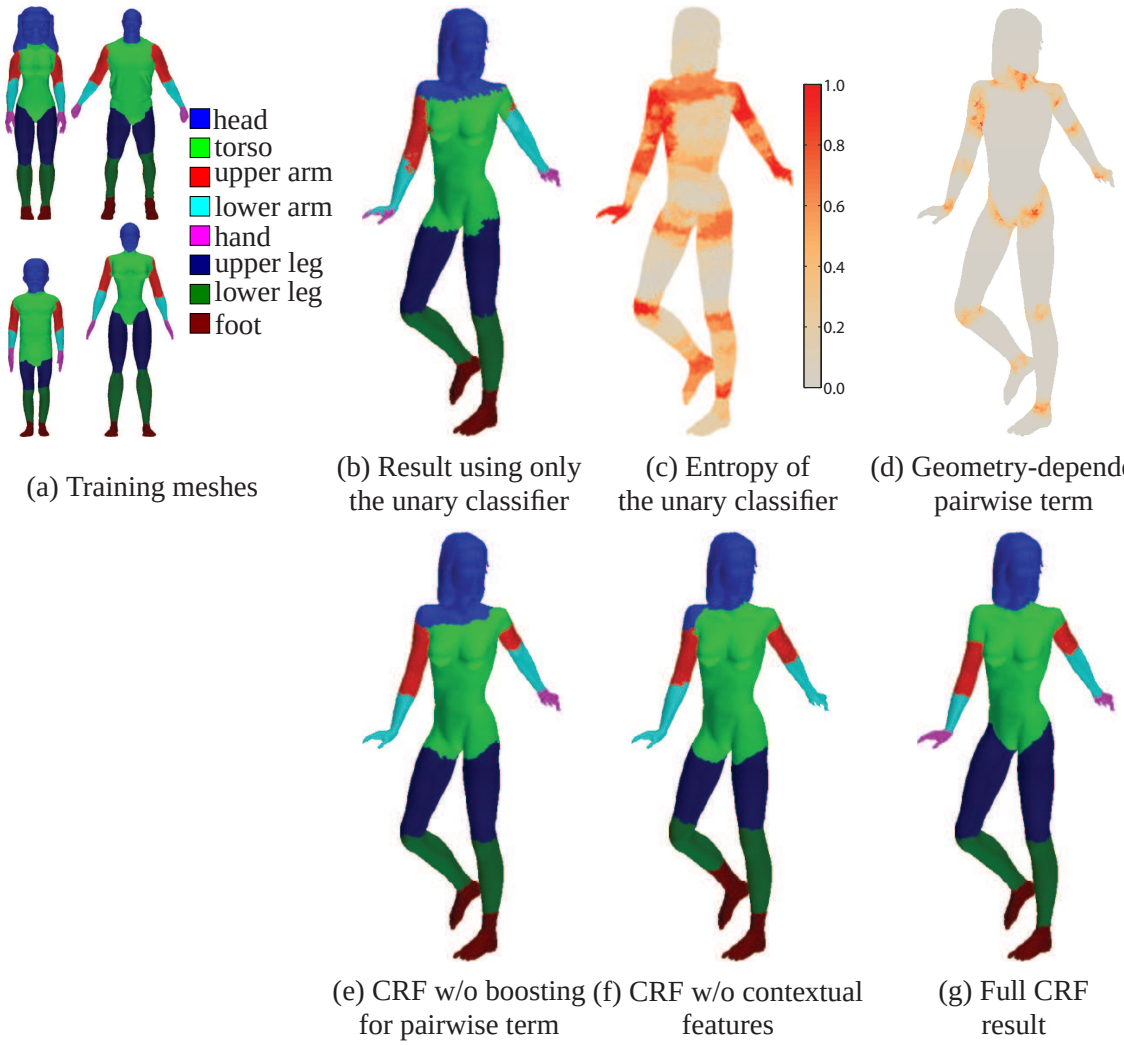


Figure 3.1: *Labeling and segmentation results from applying our algorithm to one mesh each from every category in the Princeton Segmentation Benchmark [16]. For each result, the algorithm was trained on the other meshes in the same class.*

tion, and then match segments and transfer labels based on this segmentation. These methods require the initial segmentation to be sufficiently reliable. Pekelny and Gotsman [108] track and label rigid components in sequences of 3D range data through the Iterated Closest Point registration algorithm, given an initial user segmentation. Similarly, Golovinskiy and Funkhouser [41] simultaneously partition collections of 3D models by matching points between meshes based on rigid mesh alignment. A user may provide example segmentations to be included in the matching. These methods are limited to cases where an accurate rigid correspondence exists.

Joint image segmentation and recognition has recently been an active topic in computer vision research. Early works in this area include [26, 48, 78, 81, 144, 122]. Our method is most directly inspired by TextonBoost [127], which performs joint image segmentation and recognition, using a model learned from a training database. As in Textonboost, we also make use of JointBoost and Conditional Random Fields. We add new components to the model, including 3D geometric feature vectors, 3D contextual features, cascades of classifiers, and a learned pairwise classifier term, all of which we find to be essential to obtaining good results.

Our work is also related to segmentation and recognition of 3D range data [3, 88, 98]. These methods employ small sets of features, such as local point density or height from ground, which are specialized to discriminate a few object categories in outdoor scenes, or to separate foreground from background. Golovinskiy et al. [42] segment urban range data using a graph cut method, and then apply a learned classifier, based on geometric and contextual shape cues. Range data methods aim to identify large-scale structures from point clouds, such as separating cars from roads, whereas we aim to distinguish smaller parts in 3D meshes. Hence, unlike these methods, we employ a large variety of shape-based mesh features along with appropriate contextual features, and also use sophisticated classifiers for the unary and pairwise terms.



(a) Training meshes

(b) Result using only the unary classifier

(c) Entropy of the unary classifier

(d) Geometry-dependent pairwise term

(e) CRF w/o boosting for pairwise term

(f) CRF w/o contextual features

(g) Full CRF result

Figure 3.2: *Components of our algorithm.* ***(a)*** *The entire training set for this example consists of four meshes. The bottom-right mesh is used as the validation set.* ***(b)*** *Labeling result using only the unary classifier.* ***(c)*** *Visualization of classifier uncertainty, computed as the entropy of the probabilities output by the unary classifier. Red values indicate greater uncertainty. The classifier is uncertain mainly near object boundaries, and where corresponding parts in the training meshes have inconsistent boundaries.* ***(d)*** *Geometry-dependent pairwise term (exponentiated and normalized). This term prefers boundaries to occur at specific locations.* ***(e)*** *Result of applying a CRF model without JointBoost for the pairwise term.* ***(f)*** *CRF result, but omitting contextual label features.* ***(g)*** *Result of applying our complete CRF model. Note the accuracy of the result, despite the mesh having different pose and body shape from the training meshes.*

3.2 CRF model for segmentation and labeling

We now describe our algorithm for segmenting and recognizing parts of a mesh; the procedure for learning this model is described in Section 3.3. Our goal is to label each mesh face i with a label $l \in \mathscr{C}$, where $\mathscr{C}$ is a predefined set of possible labels, such as "arm," "leg," or "torso." Each face has a vector of *unary features* $\mathbf{x}_i$, which includes descriptors of local surface geometry and context, such as curvatures, shape diameter, and shape context. These features provide cues for face labeling. In addition, for each adjacent pair of faces, we define a vector of *pairwise features* $\mathbf{y}_{ij}$, such as dihedral angles, which provide cues to whether adjacent faces should have the same label. Then, computing all mesh labels involves minimizing the following objective function:

$$E(\mathbf{c};\theta) = \sum_i a_i E_1(c_i;\mathbf{x}_i,\theta_1) + \sum_{i,j} \ell_{ij} E_2(c_i,c_j;\mathbf{y}_{ij},\theta_2) \tag{3.1}$$

where the unary term E_1 measures consistency between the features $\mathbf{x}_i$ of mesh face i and its label c_i, the pairwise term E_2 measures consistency between adjacent face labels c_i and c_j, given pairwise features $\mathbf{y}_{ij}$. The model parameters are $\theta = \{\theta_1, \theta_2\}$. The terms are weighted by the area a_i of face i, and the length of the edge ℓ_{ij} between faces i and j. In order to make energies comparable across meshes, the areas a_i are normalized by the median face area in the mesh, and the edge lengths ℓ_{ij} are normalized by the median edge length. Details of the energy terms and feature vectors are given later in this section.

As mentioned in Section 2.3.4, this type of model is referred to as a Conditional Random Field. The conditional probability of a labeling given the mesh is shown in Equation 2.57. The CRF model is more appropriate for segmentation and labeling than a Markov Random Field model that would define a joint probability over the mesh and the labels, from which the conditional may then be derived. CRFs have two advantages over MRFs for segmentation and labeling. Since the CRF allows the labels to be conditioned upon a set of observed features, the pairwise term E_2 in a CRF can depend on the input data, which is not true in an MRF. This allows us, for

example, to express that segment boundaries are more likely to occur between a pair of faces with a small exterior dihedral angle. Second, CRF learning algorithms optimize for labeling performance, whereas MRF learning algorithms attempt to model both the input features and the labels, and thus may have worse labeling performance.

The objective $E(\mathbf{c};\theta)$ is optimized using alpha-expansion graph-cuts [12]. The resulting labeling $\mathbf{c}$ implicitly defines a segmentation of the mesh, with segment boundaries lying between each pair of faces with differing labels. Note that this means that our method cannot separate adjacent parts that share the same label. Furthermore, our method is only suitable for learning segmentations that have attached labels. However, we do not require the number of segments to be specified in advance.

3.2.1 Unary Energy Term

The unary energy term evaluates a classifier. The classifier takes the feature vector $\mathbf{x}$ for a face as input, and returns a probability distribution of labels for that face: $P(c|\mathbf{x},\theta_1)$. Specifically, we use a JointBoost classifier [127, 142], summarized in Section 2.4.2. Then, the unary energy of a label c is equal to its negative log-probability:

$$E_1(c;\mathbf{x},\theta_1) = -\log P(c|\mathbf{x},\theta_1) \tag{3.2}$$

The unary classifier is the most important component of our system. As illustrated in Figure 3.2(b), labeling using just this term alone gives good results in part interiors, but not near boundaries. This is accurately reflected by the uncertainty of the classifier (Figure 3.2(c)). Next, we add a pairwise term to refine these boundaries.

3.2.2 Pairwise Energy Term

The pairwise energy term penalizes neighboring faces being assigned different labels:

$$E_2(c,c';\mathbf{y},\theta_2) = L(c,c')\ G(\mathbf{y}) \tag{3.3}$$

This term consists of a label-compatibility term L, weighted by a geometry-dependent term G. The main role of the pairwise term is to improve boundaries between segments and to prevent incompatible segments from being adjacent. The pairwise energy term is always zero when c and c' have the same label. Hence, the pairwise term cannot be used on its own, since it assigns zero energy when all faces have the same label. The geometry-dependent term is visualized in Figure 3.2(d).

The label-compatibility term $L(c,c')$ measures the consistency between two adjacent labels. This term is represented as a matrix of penalties for each possible pair of labels, which allows different pairs of labels to incur different penalties. For example, head-ear boundary edges may need to be penalized less than head-torso boundary edges (since ears might be much smaller parts and less common in the training examples) while head-foot boundaries might never occur. The costs are non-negative $(0 \leq L(k,l))$ and symmetric $(L(k,l) = L(l,k))$, for labels $k,l \in \mathscr{C}$. Furthermore, we constrain there to be no penalty when there is no discontinuity: $L(k,k) = 0$ for all k.

The geometry-dependent term $G(\mathbf{y})$ measures the likelihood of there being a difference in labels, as a function of the geometry alone. This term has the following form:

$$\begin{aligned} G(\mathbf{y}) = &- \kappa \log P(c \neq c'|\mathbf{y},\xi) \\ &- \lambda \log\left(1 - \min(\omega/\pi, 1) + \varepsilon\right) + \mu \end{aligned} \tag{3.4}$$

The first term is the output of a JointBoost classifier that computes $P(c \neq c'|\mathbf{y},\xi)$, the probability of two adjacent faces having distinct labels, as a function of pairwise geometric features $\mathbf{y}$. This classifier helps detect boundaries better than using only dihedral angles (Figure 3.2e). The

second term penalizes boundaries between faces with high exterior dihedral angle ω, following Shapira et al. [124]. The μ term penalizes boundary length and is helpful for preventing jaggy boundaries and for removing small, isolated segments [40, 124]. A small constant ε is added to avoid computing $\log 0$.

3.2.3 Feature vectors

We do not know in advance which features will be useful for segmentation. Furthermore, it may be that different features are informative for different mesh parts and for different styles of segmentation. As a result, we construct our feature vectors out of as many informative features as possible. Since the JointBoost algorithm performs automatic feature selection, each classifier only uses a subset of the provided features. In our experiments, we have not found a case where adding informative features led to worse results. Hence, one may add other features besides the ones listed here. We find that the precise form of the features is important: careful selection of details, such as binning strategy and normalization, can improve results. Adding features does increase computation time, especially for preprocessing and learning. Hence, we have attempted to design features that are as informative as possible.

Unary features. We use multi-scale surface curvature, singular values extracted from Principal Component Analysis of local shape, shape diameter [124], distances from medial surface points [91], average geodesic distances [53, 157], shape contexts [10], and spin images [64] to form a basic 651-dimensional feature vector $\tilde{\mathbf{x}}_i$ per face i. Full details of our implementation for these features are given in Appendix A.1.

Contextual label features. Training a classifier using only the above features is often sufficient for labeling. However, in many cases, better results can be achieved by re-training an

additional classifier that uses information about the global distribution of labels around each mesh face. Since the labels are not known in advance, they are approximated by an initial application of the classifier with the above features. We introduce *contextual label features* based on these initial labels.

We first train an initial JointBoost classifier using the initial feature vector $\tilde{\mathbf{x}}$. This classifier can be applied to each training mesh to produce per-face class probabilities $P(\mathbf{c}|\tilde{\mathbf{x}})$. Then, for each face i, we compute a histogram of these probabilities, which captures the global distribution of part labels relative to the face, in a manner inspired by shape contexts [10], and similar to image auto-contexts [143] and bags of semantic textons [126]. The histogram bins are determined as a function of geodesic and euclidean distances. These features allow the algorithm to make use of estimates of labels from the global context of each face. Details of the histograms are given in the Appendix.

The values of these histogram bins form a set $\bar{\mathbf{x}}_1$ of contextual label features that are concatenated with $\tilde{\mathbf{x}}$ to produce the full feature vector $\mathbf{x}_1 = [\tilde{\mathbf{x}}^T, \bar{\mathbf{x}}_1^T]^T$. The new feature vector has $651 + 35 \cdot |\mathscr{C}|$ features, where $|\mathscr{C}|$ is the number of labels. Then, we train a new JointBoost classifier from $\bar{\mathbf{x}}_1$ to class probabilities. The new classifier will now take into account the generated contextual features to further discriminate parts, as shown in Figure 3.2g, compared to the result of Figure 3.2f, where only the initial JointBoost classifier is used (without the contextual label features).

After training the second classifier, we bin the newly produced class probabilities $P(\mathbf{c}|\mathbf{x}_1)$ to produce new contextual label features $\bar{\mathbf{x}}_2$. These are concatenated with $\tilde{\mathbf{x}}$ to produce a third feature vector $\mathbf{x}_2$. Then, we can train a third classifier based on the feature vector $\mathbf{x}_2$. This process can be iterated to further refine the discrimination of classes, similar to cascade generalization [39]. This approach may be iterated N times to produce N feature vectors, until the error on the validation set does not increase. In our experiments, the algorithm usually selects $N = 3$. Computing the feature vectors for a new surface entails repeating the same process as above: $\tilde{\mathbf{x}}$

is computed for each face, then the first JointBoost produces the first set of contextual features $\bar{\mathbf{x}}$, and the process repeats until getting $\mathbf{x}_N$, which is used as the complete feature vector $\mathbf{x}$. We find that using these contextual features produces a significant improvement in performance, about $3-10\%$, depending on the mesh category.

Pairwise features. The pairwise feature vector $\mathbf{y}_{ij}$ between faces i and j consists of the dihedral angles between the faces, and differences of the following features between the faces: curvatures and surface third-derivatives, shape diameter differences, and differences of distances from medial surface points. We note that dihedral angles are included in both the pairwise features and in Equation 3.4. The reason is that the probability of two adjacent faces having distinct labels, as outputted by the pairwise classifier, might not be very well localized around a potential boundary. This especially happens when the boundaries of the training human segmentations are noisy. Including the dihedral angles in Equation 3.4 may further ”move” the segmentation boundaries towards faces with high exterior dihedral angle more accurately.

We also use contextual label features, similar to the features above; however, we found in our experiments that these contextual features have little impact on the results (about 0.5% improvement). The complete feature vector $\mathbf{y}$ is 191-dimensional. Details of the pairwise features are given in Appendix A.2.

3.3 Learning CRF parameters

We now describe a procedure for learning the parameters of the CRF model, given a set of labeled training meshes. The natural approach to CRF learning is Maximum Likelihood or MAP, e.g., maximizing Equation 2.57 over all training meshes. Unfortunately, as mentioned in 2.3.4, computing the normalization Z is intractable. While contrastive divergence can be used for this optimization [48], this method is computationally expensive, and would not be feasible

at the scale of mesh processing.

Instead, we perform the following steps, based on the approach of Shotton et al. [127]. First, we randomly split the training meshes into an *exemplar set* and a *validation set*, in a proportion of approximately $4:1$. We then learn the JointBoost classifiers for the unary term and the pairwise term from the exemplar set. Finally, the remaining CRF parameters are learned by iteratively optimizing segmentation performance on the validation set. These steps are described below.

3.3.1 Learning JointBoost classifiers

Here, we summarize the JointBoost learning algorithm, for learning classifiers of the form described in Section 2.4.2. See also [142] for an excellent explanation and derivation of the algorithm.

The input to the algorithm is a collection of M training pairs $(\mathbf{z}_i, c_i)$, where $\mathbf{z}_i$ is a feature vector and c_i is the corresponding class label for that feature. Furthermore, each training pair is assigned a per-class weight $w_{i,c}$. JointBoost minimizes the weighted multiclass exponential loss over the exemplar set:

$$J = \sum_{i=1}^{M} \sum_{l \in \mathscr{C}} w_{i,c} \exp\left(-I(c_i, l)\, H(\mathbf{z}_i, l)\right) \tag{3.5}$$

where $H(\mathbf{z}, l)$ is defined in Equation 2.65, $\mathscr{C}$ is the set of possible class labels, and $I(c, c')$ is an indicator function that is 1 when $c = c'$ and -1 otherwise.

For the unary terms, the training pairs are the per-face feature vectors and their labels $(\mathbf{x}_i, c_i)$ for all mesh faces in the exemplar set. For the pairwise terms, the training pairs are the pairwise feature vectors and their binary labels $(\mathbf{y}_{ij}, c_i \neq c_j)$. For the unary term, the $w_{i,c}$ is the area of face i. For the pairwise term, $w_{i,c}$ is used to re-weight the boundary edges, since the training data contains many more non-boundary edges. Let N_B and N_{NB} be the number of each type of

edge, then $w_{i,c} = \ell N_B$ for non-boundary edges and $w_{i,c} = \ell N_{NB}$ for boundary edges, where ℓ is the corresponding edge length.

The algorithm proceeds iteratively. The algorithm stores a set of weights $\tilde{w}_{i,c}$ that are initialized to the weights $w_{i,c}$. Then, at each iteration, one decision stump (Equation 2.64) is added to the classifier. The parameters ϕ_j of the stump at iteration j are computed to optimize the following weighted least-squares objective:

$$J_{wse}(\phi_j) = \sum_{l \in \mathscr{C}} \sum_{i=1}^{M} \tilde{w}_{i,l} \big(I(c_i, l) - h(\mathbf{z}_i, l; \phi_j) \big)^2 \tag{3.6}$$

where $\mathscr{C}$ are the possible class labels. Following Torralba et al. [142], the optimal a, b, k_l are computed in closed-form, and $f, \tau, \mathscr{C}_S$ are computed by brute-force. When the number of labels $|\mathscr{C}|$ is greater than 6, the greedy heuristic search is used for $\mathscr{C}_S$. Once the parameters ϕ_j are determined, the weights are updated as:

$$\tilde{w}_{i,c} \leftarrow \tilde{w}_{i,c} \exp(-I(c_i, l)\ h(\mathbf{z}_i, l; \phi_j)) \tag{3.7}$$

and the algorithm continues with the next decision stump.

We run the algorithm for at most 300 iterations. To avoid overfitting, we also monitor the classifier's performance on the validation set by computing the cost function of Eq. 3.5 after each iteration, and keep track of which iteration j^* gave the best score. At the end of the process, we return the classifier from step j^* (i.e., discarding decision stumps from after step j^*). We also terminate early if the classifier's performance in the validation set has not improved over the last 50 iterations.

3.3.2 Learning the remaining parameters

Once the JointBoost classifiers have been learned, we learn the remaining parameters of the pairwise term $(\kappa, \lambda, \mu, L)$ by hold-out validation. Specifically, for any particular setting of these

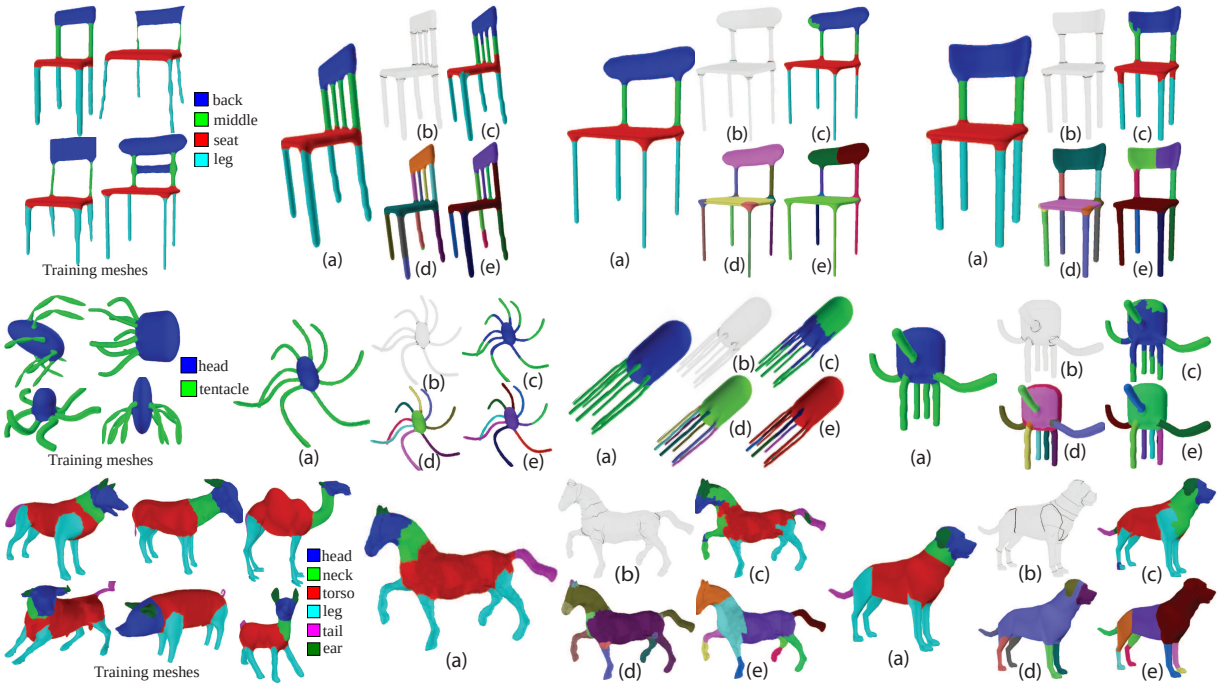


Figure 3.3: *Comparisons to previous segmentation methods, for chairs, octopuses, and quadrupeds. For each test, the entire training set is shown on the left. In each figure, the methods compared are:* ***(a)*** *our method,* ***(b)*** *average human segmentation from the Princeton Segmentation Benchmark,* ***(c)*** *Consistent Segmentation [Golovinskiy 2009],* ***(d)*** *Shape Diameter [Shapira In Press],* ***(e)*** *Randomized Cuts [Golovinskiy 2008], with number of segments defined as the average number of segments in the category. The Consistent Segmentation method provides labels in addition to segmentation based on the same training set. The other methods only perform segmentation, and do not make use of training data.*

parameters, we can apply the CRF to all of the validation meshes, and evaluate the classification results. We seek the values of these parameters that give the best score on the validation meshes.

We need to define an error function by which to evaluate classification results. A obvious choice would be to measure what percentage of the mesh's surface area is correctly labeled. We refer to this as the Classification Error:

$$E = \left(\sum_i a_i (I(c_i, c_i^*) + 1)/2 \right) / \left(\sum_i a_i \right) \tag{3.8}$$

where a_i is the area of face i, c_i is the ground-truth label for face i, $c_i^* = \arg\max P(c|\mathbf{x}_i)$ is the output of the classifier for face i, and $I(c,c')$ defined as in Section 3.3.1. However, when training against this error, the algorithm tends to mostly refine boundaries between larger parts but skip cuts that generate small parts, producing noticeable errors in the results without incurring much penalty.

Instead, we optimize with respect to the Segment-Weighted Error which weighs each segment equally:

$$E_S = \sum_i \frac{a_i}{A_{c_i}} (I(c_i, c_i^*) + 1)/2 \tag{3.9}$$

where A_{c_i} is the total area of all faces within the segment that has ground-truth label c_i.

These parameters are optimized in two steps. First, the Segment-Weighted Error is minimized over a coarse grid in parameter space by brute-force search. Second, starting from the minimal point in the grid, optimization continues using MATLAB's implementation of Preconditioned Conjugate Gradient with numerically-estimated gradients.

3.4 Results

We now describe experimental validation and analysis of our approach.

Data set. We employed data from the Princeton Segmentation Benchmark [16] for all of our tests. The dataset provides 19 categories of meshes, segmentations provided by human users, source code for computing evaluation scores, and the results of applying many previous segmentation methods.

We performed a few initial steps to process the data. Since segment labels are not provided with the data, we manually assigned a set of labels to each class (Figure 3.1), according to

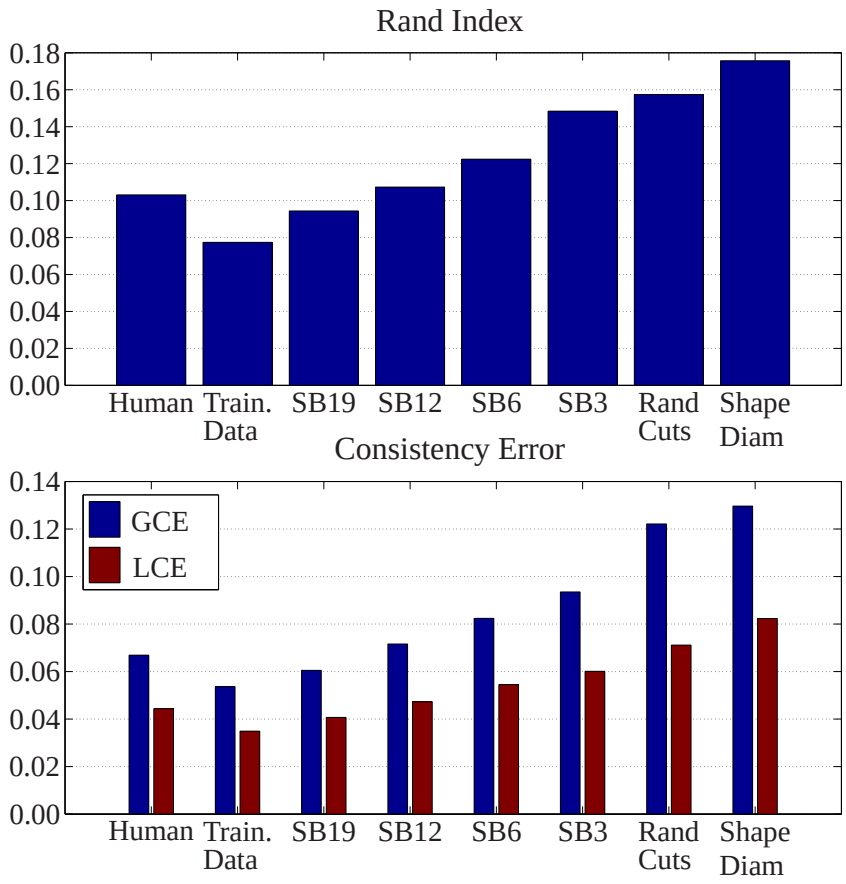


Figure 3.4: *Evaluation of segmentation. For all methods, evaluations are performed according to the protocols of [Chen et al. 2009], using all human segmentations in the Princeton Segmentation Benchmark. 'SB19' represents leave-one-out-error of our technique averaged over all the categories of the benchmark. 'SB12', 'SB6', 'SB3' represents the average error using training sets of size 12,7,6, and 3 (see text for details). SB19 performs almost* 50% *better than the best existing methods. Performance drops with less training data, but, even with only* 3 *examples, our method still out-performs previous methods by a small margin.*

the average human segmentation for each category. For example, almost all users partition the elements of the chair class (Fig. 3.3) into legs, seats, back, and middle.

For each mesh, the Princeton benchmark provides multiple segmentations. The dataset contains significant variations in types of segmentations: while many segmentations are consistent with each other, one user might segment a human into just 4 segments, whereas another

might use 50 segments. We select one of these segmentations to be labeled and used as the training/test data for that mesh, in order to reduce the size of the dataset and remove outlier segmentations. For most meshes, the exemplar segmentation was selected as the segmentation with the lowest average Rand Index to all other segmentations for that mesh. However, in a few cases, the mesh with the best score had a very atypical segmentation to the rest of the category (e.g., the best segmentation for one octopus mesh had tentacles subdivided into many parts, whereas the tentacles in the rest of the category were not), in which case, we manually merged segments or chose the second-best segmentation.

Labeling results. We now evaluate the quality of the labels produced by our method. Because each category in the database has only 20 meshes, we evaluate our method using leave-one-out cross-validation. For each mesh i in each category, we train a CRF model on the other 19 meshes in that class, and then apply it to mesh i, and compute the Classification Error (Eq. 3.8) according to the ground-truth data. We report Recognition Rate, which is one minus Classification Error, reported as a percentage. Averaging over all categories, our method obtains approximately 94% accuracy.

In order to determine the effect of training set size, we repeated the experiment with smaller training sets. When testing on mesh i, the CRF is trained on a subset of M of the remaining 19 meshes. These are averaged over 5 randomly-selected subsets. We tested with $M = 3, 6, 12$. Table 3.1(left) shows scores of our method for different mesh categories and for different values of M. When reducing the training set size, we find that our method still gives excellent results for categories with little geometric variation (such as the Octopus), whereas other categories, such as Bust and Bearing, have very different geometric parts; a subset of 3 meshes often lacks some of the labels used elsewhere in the category.

The Segment-Weighted Error (Eq. 3.9) scores of our algorithm are: 89% (leave-one-out error), 85% ($M = 12$ training examples), 81% ($M = 6$ training examples) and drops to 75% ($M = 3$

training examples). The scores using this metric are lower than those of Classification Error, because tiny missing segments cause disproportionally large penalties.

To our knowledge, the only previous method that can label meshes by example is that of Golovinskiy and Funkhouser [41]. This method assumes that rigid alignment can be performed between a mesh and the training data, which does not hold for most of the benchmark data; comparisons are shown in Figure 3.3. To our knowledge, our method is the first to be able to accurately label such a broad class of meshes.

Segmentation results. In contrast to labeling, we test our segmentation algorithm using all original human segmentations for all meshes, according to the protocol from Chen et al. [16]. Results are shown in Figure 3.4, and comparisons to previous methods are shown in Figure 3.3. Our method gives a significant improvement over the previous state-of-the-art, according to all measures proposed by Chen et al. [16]. Even when training on just three meshes, our method obtains better scores than other methods in nearly all cases. Table 3.1(right) provides Rand Index scores for each category and for different choices of training set size as above.

There are a few details to note about these experiments. First, Rand Cuts requires as input the number of segments for the mesh. For this, we used the average number of segments for each category. Second, the Human Score is worse than the score for our training data (computed as in [16]) because we reduce the training set as described above. Third, when disconnected parts on the same mesh have the same label (e.g., the two hands on a human), they are scored as separate segments.

Feature selection. Figure 3.5 visualizes which features were selected by JointBoost in the unary term, for various subsets of the data. For example, the top row shows, for each type of feature, the percentage of this feature that was used, across the entire benchmark dataset. The most features came from Shape Contexts [10] and the Contextual Label. The third row

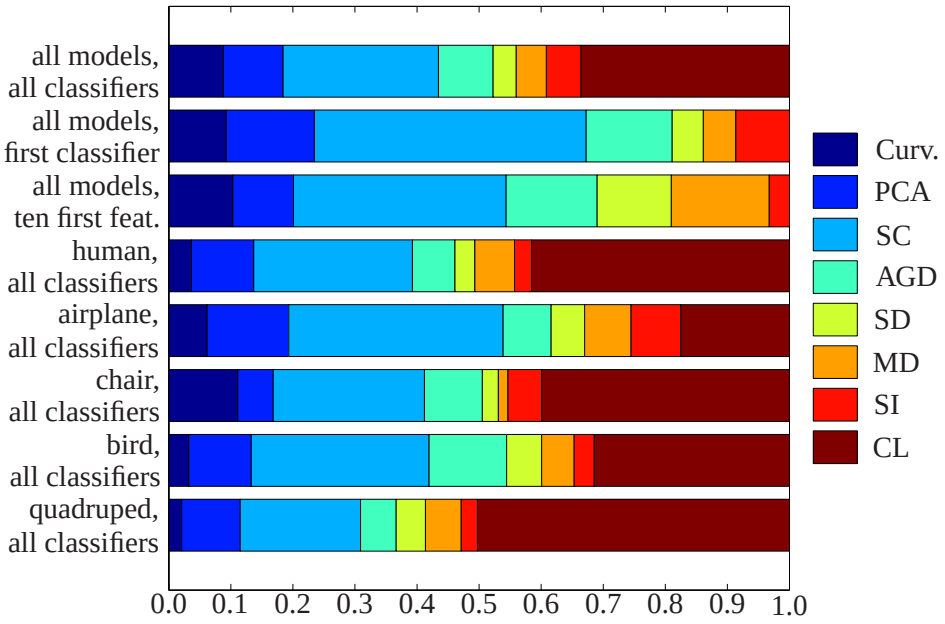


Figure 3.5: *Percentages of features used by JointBoost for different cases. See text for details. Legend: Curv.=curvature, PCA=PCA singular values, SC=shape contexts, AGD=average geodesic distances, SD=shape diameter, MD=distance from medial surface, SI = Spin Images, CL = contextual label features.*

shows the ten features that were selected by the first ten rounds of JointBoost (i.e., the features used by the first ten decision stumps). The remaining rows show features used for individual categories. These results indicate that the Shape Context features were the most important among the basic features. However, each type of feature was used multiple times. This is a common theme among boosting algorithms: adding more features that provide independent sources of information typically improves results.

Generalizing to different categories. Fig. 3.6 shows results in which we train on one category and test on another. The algorithm yields reasonable results when labeling airplanes like birds, tables like chairs, and people like quadrupeds.

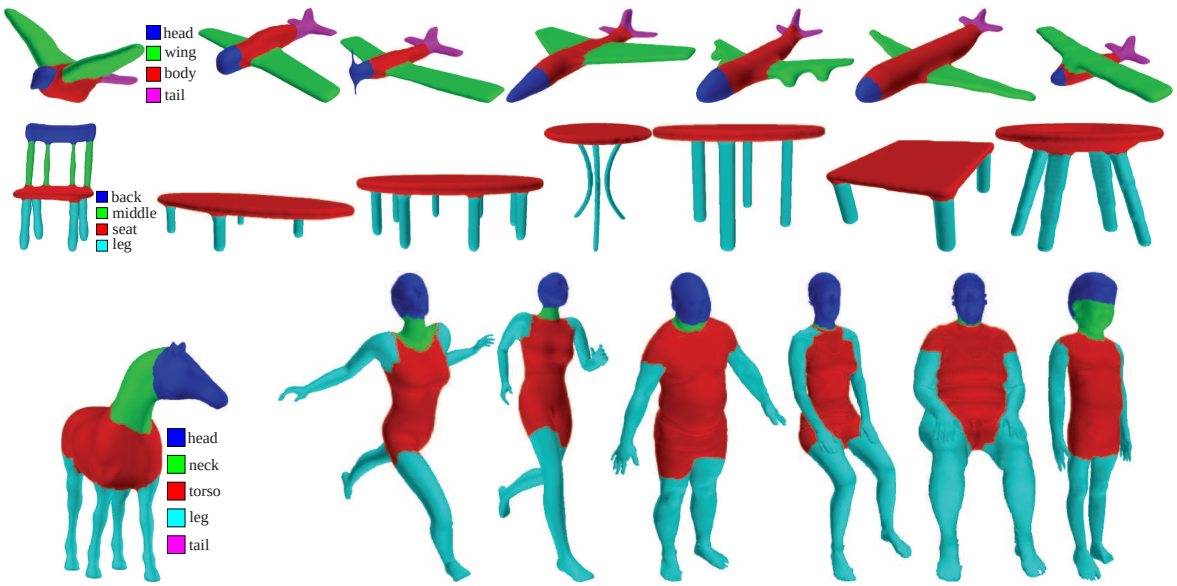


Figure 3.6: *Experiments where training and test categories are different. Sample training data shown on the left (complete training sets not shown).* ***Top row:*** *Training on birds, applying to planes.* ***Middle row:*** *Labeling tables as chairs.* ***Bottom row:*** *Labeling humans as quadrupeds. A failure case here is in the lower-right, where much of the child's face is confused for a neck. The seated humans illustrate a limitation of our method, i.e., that connected segments with the same label are not separated; here, a left arm is not separated from a leg when they connect.*

Different styles of segmentations. Our algorithm can be used to learn different styles of segmentation for different tasks. We demonstrate this capability with a set of animal segmentations from the benchmark data that separate the torso into three segments (Fig. 3.7), unlike the dataset used for quantitative evaluation. Our algorithm correctly applies these labels to several test meshes, except the giraffe.

Merging categories. Figure 3.8 shows an example in which a CRF was learned on a training set consisting of both humans and teddy bears, and applied to new humans and teddy bears. The algorithm successfully learns a model given the non-homogeneous data.

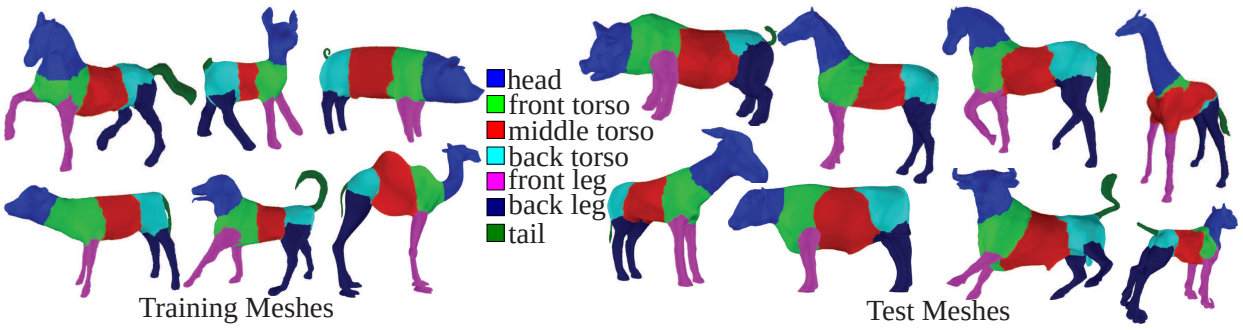


Figure 3.7: *Using an alternative segmentation style. Our main quantitative experiments used a segmentation style from the Princeton Benchmark in which each animal has a single torso (e.g., see Fig. 3.3). Here we train on examples from the Benchmark in which the torso is split into three segments. The six training meshes are shown on the left. Changing the style does not require any manual parameter tuning. Good results are obtained for several test meshes, except the giraffe, where the torso is not labeled accurately.*

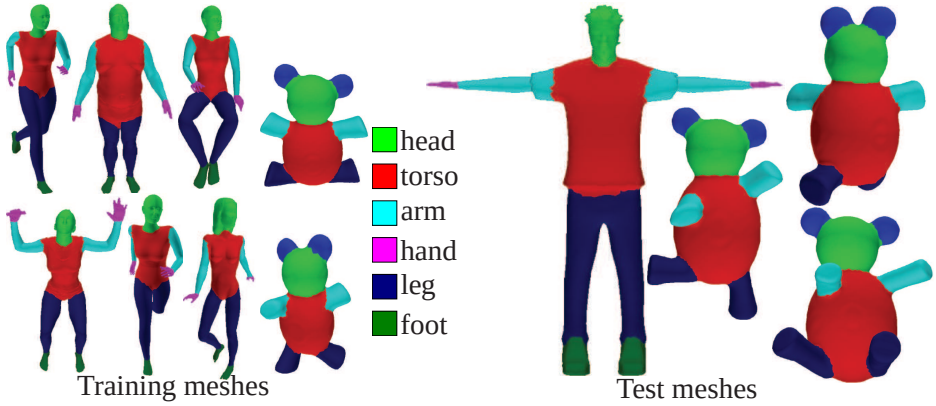


Figure 3.8: *Merging categories. A CRF was learned from the training meshes on the left, which include both humans and teddy bears. Results on a test human and bears shown on right.*

3.5 Applications

We now briefly describe a few procedures that illustrate how our approach could be used to automate workflows that would otherwise involve laborious manual effort. For each application, we implemented an automatic pipeline that takes a mesh of an object category as input,

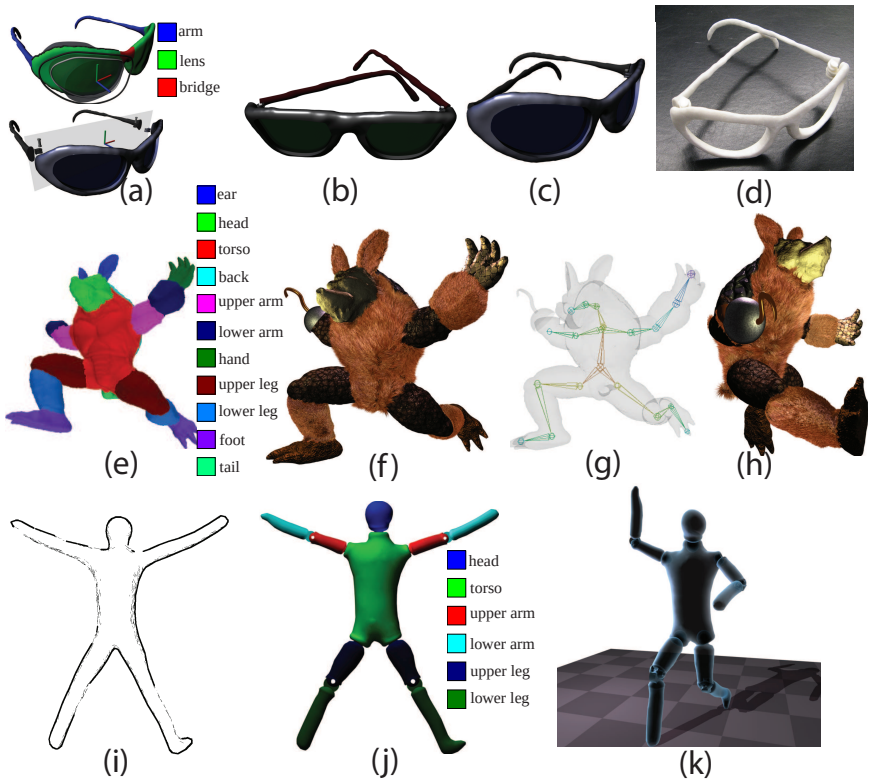


Figure 3.9: ***Top row:*** *Automatic procedure for converting Glasses meshes into manufacturable 3D objects with lenses and hinges.* ***(a)*** *The mesh is broken at the segment boundaries between the frame and arms with our labeling technique, and corresponding hinges are placed. Lenses are procedurally offset and subtracted from the frame.* ***(b-c)*** *Two example glasses created with this procedure.* ***(d)*** *A functional prototype, with working hinges, printed on a 3D printer.* ***Middle row:*** *Automatic shader assignment and rigging based on segment labels.* ***(e)*** *Labeled armadillo.* ***(f)*** *Procedural shaders assigned based on part labels, e.g., fur for the torso.* ***(g)*** *An animation skeleton is fitted to Armadillo automatically by placing the joints at the centroids of corresponding segment boundaries.* ***(h)*** *Posed armadillo.* ***Bottom row:*** *Automatic conversion of a 3D model drawn with the Teddy sketching package into an articulated mannequin.* ***(i)*** *Sketched 3D model.* ***(j)*** *Labeled model with mechanical joints placed at segment boundaries.* ***(k)*** *Articulated model.*

computes segmentation and labeling, and then processes the extracted parts. Such procedures could automate processing of large databases of objects of the same category.

Functional prototyping. Functional prototyping entails creating a real and working 3D object from a mesh, such as created by a designer. Our eyeglass pipeline (Figure 3.9(a-d)) takes a mesh as input and computes segmentation and labeling. The frontal silhouettes of the *lens* parts are offset, extruded, and subtracted from the object to create a frame. A frontal plane passing through the combined centroid of the two arm-lens segment boundaries is used to cut the mesh, separating the arms from the frame. Hinges and pins are created at the cut, resulting in a wearable pair of glasses. We have also implemented a procedure that, using a modeling tool like Teddy [58], converts a single sketched stroke into an articulated 3D mannequin, with joint-types based on extracted part labels (Figure 3.9i-k).

Rigging and texturing. Given an automatically computed segmentation and labeling, a skeleton may be created by placing joints at centroids of part boundaries. We further create texture for armadillo meshes (Figure 3.9(e-h)), using textures and accessories assigned to different labels, such as leathery skin for the feet, fur for the torso, and a hook in place of a missing hand.

3.6 Discussion

We have described the first learning algorithm for both labeling and segmentation of 3D meshes. The model is learned from a training set without requiring any manual parameter tuning, and it obtains state-of-the-art results on all categories in the Princeton Segmentation Benchmark. Our method is the first to demonstrate effective labeling on a broad class of meshes. As our method represents an early attempt in this area, there are several limitations to our method (Figure 3.10), and many exciting directions for future work.

While considerable effort has rightly been put into devising geometric criteria for shape classification, it remains an open question as to whether simple geometric criteria are sufficient

for segmenting the way humans do. Our work suggests that learning models from data—using carefully-chosen geometric features—can significantly improve results. While this method is not easily interpretable in terms of geometric intuitions, this kind of approach may nonetheless be of great practical value.

A major limitation of our approach is the need for labeled training data. The dataset must have consistent labels, although some variation can be tolerated. For example, in Figure 3.3, the pig does not have a neck segment, unlike the other meshes in the training data.

Generalization performance typically drops with fewer training meshes. Classes with larger variability across the data require larger training sets for good results. For example, the Ant and Octopus classes give good results with very few training examples, whereas the Bust and Vase categories give very poor results with small training sets (Table 3.1). For all classes, increasing the training set size improves performance.

Our method cannot learn "generic" segmentations, that is, segmentation without class-specific labels. The method cannot model segmentations where connected parts share labels (Figure 3.10(a-b)). We also assume that the target mesh is consistent with the training data; e.g., there are no outlier segments. However, we believe that elements of our approach could be useful for these or related problems. For example, our pairwise term could be used with a different unary term, such as one based on interactive labeling or mesh alignment.

Adding additional informative geometric features should improve results. At present, our algorithm cannot distinguish left/right/up/down (e.g., left arm vs. right arm); features informative of orientation [36] may help. Symmetry-based features and constraints could also be useful. Because many of our features depend on geodesic distances, they may not be very accurate when a test mesh exhibits significantly different topology than the training. Developing new part-aware and topology-insensitive shape descriptor features may help our method.

Our choice of features assumes that each shape is described by a watertight 3D mesh with a

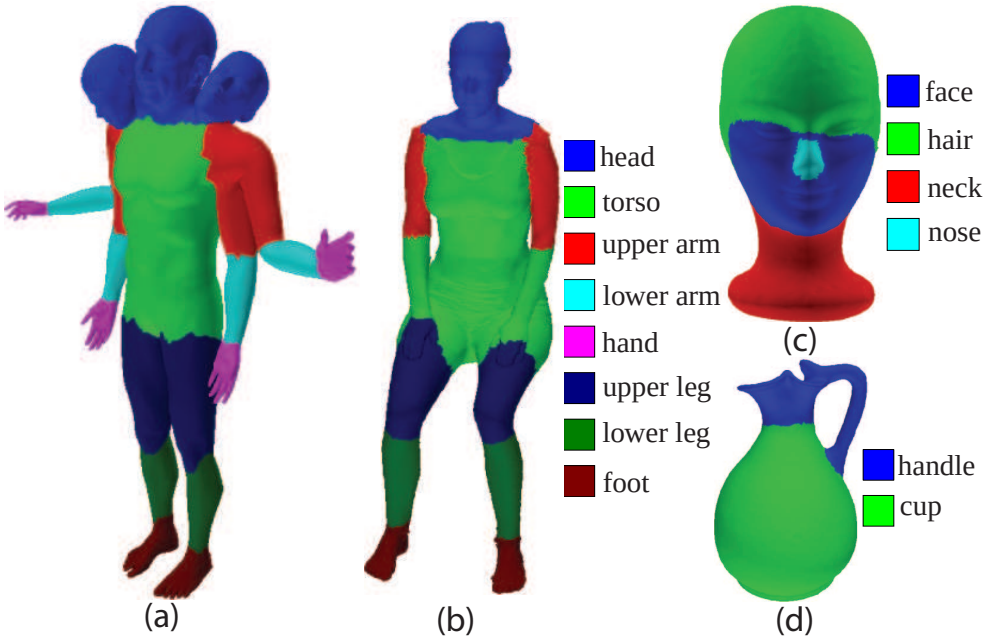


Figure 3.10: *Examples of limitations of our algorithm:* ***(a)*** *Shiva statue (not included in the benchmark), classified with a CRF learned from the Human category. The algorithm correctly labels the multiple heads and arms, but cannot separate connected segments with the same label.* ***(b)*** *Example of a test human mesh that has significantly different topology than the other training meshes of the Human category; its arms are connected to the legs, causing the algorithm to mislabel the lower arms, hands and upper torso.* ***(c)*** *Our lowest scores in the benchmark were on the Bust category; even when all the other busts are used as training meshes, our algorithm can still have significant errors.* ***(d)*** *Example of a vase, classified with a model learned from* 3 *other training meshes from the Vase category; the performance drops significantly in some categories with large variability, when few training meshes are used.*

single connected component. Applying our technique for point clouds or polygon soups would require several modifications in our feature set. This should allow our method to be applied to data such as found in 3D scanning and architectural applications.

The size of our training set is limited by training time, which is several days for our largest datasets. Some of the features are expensive to compute e.g., computing geodesic distances between all faces on a mesh is $O(N^2 logN)$ in our implementation or computing shape diameter, shape context and spin images has complexity of $O(N^2)$. For example, our implementation for SDF computation takes several minutes per mesh, since we perform exhaustive ray-triangle intersections. However, it is important to note that our implementation for feature extraction is far from optimal and can be accelerated with several ways (i.e, use spatial search structure to accelerate computations of intersections).

Training the JointBoost classifier is also a computationally expensive procedure i.e., it has complexity of $O(|\mathscr{C}|^2 \cdot N \cdot D \cdot T)$, if greedy search is used for finding $\mathscr{C}_S$, where $|\mathscr{C}|$ is the number of labels, N is the number of training samples, D is the number of features used, T is the number of boosting rounds. JointBoost can be accelerated by using less number of samples (i.e., instead of using every single mesh face as a training sample) and by using randomized feature selection at each boosting round [127]. The CRF inference with graph-cut alpha-expansion relies on maximum flow on graphs to compute the minimum cut; empirically, it seems to scale near-linearly with the mesh size and the number of labels [12].

For example, training on the quadrupeds category with 6 training meshes of about 10K-30K faces and 6 labels, takes about 8 hours on a single Xeon E5355 2.66GHz processor. Approximately 30% of the time is consumed by feature extraction, 50% is consumed by JointBoost, 20% of the time is used by hold-out validation. Testing on another animal mesh of, e.g. 20K faces takes about 15 minutes and the vast majority of the time is consumed by feature extraction (CRF inference and evaluating decision stumps take a few seconds at most). If feature extraction is improved (e.g. SDF computations could be reduced to a few seconds), if JointBoost is

implemented more optimally along with subsampling and randomized feature selection, and if hold-out validation is handled by a more efficient optimizer (in C/C++ rather than Matlab), a rough estimate could be much less than an hour for training on such dataset and far less than 5 minutes for testing per mesh. Such improvements could permit the method to train on much larger datasets with more classes still within reasonable amounts of time in the future.

	Labeling: Recognition Rate				**Segmentation: Rand Index**					
	SB19	*SB12*	*SB6*	*SB3*	*Bench.*	*Train.*	*SB19*	*SB12*	*SB6*	*SB3*
Human	93.6	93.2	89.4	83.2	13.5	11.2	11.9	12.9	14.3	14.7
Cup	99.6	99.6	99.1	96.3	13.6	9.8	9.9	9.9	10.0	10.0
Glasses	97.4	97.2	96.1	94.4	10.1	8.4	13.6	14.1	14.1	14.2
Airplane	96.3	96.1	95.5	91.2	9.2	7.4	7.9	8.2	8.0	10.2
Ant	98.8	98.8	98.7	97.4	3.0	1.7	1.9	2.2	2.3	2.6
Chair	98.5	98.4	97.8	97.1	8.9	5.2	5.4	5.6	6.1	6.6
Octopus	98.4	98.4	98.6	98.3	2.4	1.8	1.8	1.8	2.2	2.2
Table	99.4	99.3	99.1	99.0	9.3	5.9	6.2	6.6	6.4	11.1
Teddy	98.1	98.1	93.3	93.1	4.9	3.1	3.1	3.2	5.3	5.6
Hand	90.5	88.7	82.4	74.9	9.1	9.1	10.4	11.2	13.9	15.8
Plier	97.0	96.2	94.3	92.2	7.1	5.1	5.4	9.0	10.0	10.5
Fish	96.7	95.6	95.6	94.1	15.5	11.8	12.9	13.2	14.2	13.5
Bird	92.5	87.9	84.2	76.3	6.2	4.4	10.4	14.8	14.8	18.6
Armadillo	91.9	90.1	84.0	83.7	8.3	6.3	8.0	8.4	8.4	8.6
Bust	67.2	62.1	53.9	52.2	22.0	18.8	21.4	22.2	33.4	39.3
Mech	94.6	90.5	88.9	82.4	13.1	8.5	10.0	11.8	12.7	24.0
Bearing	95.2	86.6	84.8	61.3	10.4	6.8	9.7	17.6	21.7	32.7
Vase	87.2	85.8	77.0	74.3	14.4	10.5	16.0	17.1	19.9	25.3
FourLeg	88.7	86.2	85.0	82.0	14.9	11.6	13.3	13.9	14.7	16.3
Average	**93.8**	**92.0**	**89.4**	**85.4**	**10.3**	**7.7**	**9.4**	**10.7**	**12.2**	**14.8**

Table 3.1: ***Left:*** *Recognition rate scores for our method across all categories in the benchmark, and for various training set sizes* $M = 3, 6, 12, 19$. ***Right:*** *Rand Index scores for human segmentations, training segmentations, and our method. Recognition rate is measured against our labeling dataset (see text for details), whereas the Rand Index is measured against all human segmentations in the Princeton benchmark.*

Chapter 4

Learning hatching for pen-and-ink illustration of surfaces

In this chapter, I present a machine learning method for creating pen-and-ink illustrations of surfaces by example [1]. In contrast to previous hatching algorithms that are manually designed from insight and intuition, this example-based method provide a largely automated and potentially more natural workflow for an artist.

Given a single illustration of a 3D object, drawn by an artist, the method learns a model of the artist's hatching style, and can apply this style to rendering new views or new objects. Hatching and cross-hatching illustrations use many finely-placed strokes to convey tone, shading, texture, and other qualities. Rather than trying to model individual strokes, the algorithm focuses on the *hatching properties* across an illustration: hatching level (hatching, cross-hatching, or no hatching), stroke orientation, spacing, intensity, length, and thickness. Whereas the strokes themselves may be loosely and randomly placed, hatching properties are more stable and pre-

[1] The work presented in this chapter is conditionally accepted to ACM Transactions on Graphics. Future project web page: http://www.dgp.toronto.edu/~kalo/papers/MLHatching/

dictable. The learning is based on piecewise-smooth mappings from geometric, contextual, and shading features to these hatching properties. Because we do not know in advance which input features are the most important for different styles, we use the boosting techniques described in Section 2.4. Because we found that artists appear to apply different types of stroke direction fields and tone for different surface parts, we also use a mixture-of-experts model (Section 2.2.5) to decompose the drawing into parts; a different model of stroke orientations is extracted depending on each part. The identified parts produced by the mixture-of-experts model are categorical properties that are also learned using Conditional Random Fields and JointBoost, as in the case of part labeling in meshes.

To generate a drawing for a novel view and/or object, a Lambertian-shaded rendering of the view is first generated, along with the selected per-pixel features. The learned mappings are applied, in order to compute the desired per-pixel hatching properties. A stroke placement algorithm then places hatching strokes to match these target properties. We demonstrate results where the algorithm generalizes to different views of the training shape or and different shapes.

This work focuses on learning hatching properties; we use existing techniques to render feature curves, such as contours, and an existing stroke synthesis procedure. We do not learn properties like randomness, waviness, pentimenti, or stroke texture. Each style is learned from a single example, without performing analysis across a broader corpus of examples. Nonetheless, our method is still able to successfully reproduce many aspects of a specific hatching style even with a single training drawing.

4.1 Related Work

Previous work has explored various formulas for hatching properties. Saito and Takahashi [119] introduced hatching based on isoparametric and planar curves. Winkenbach and Salesin

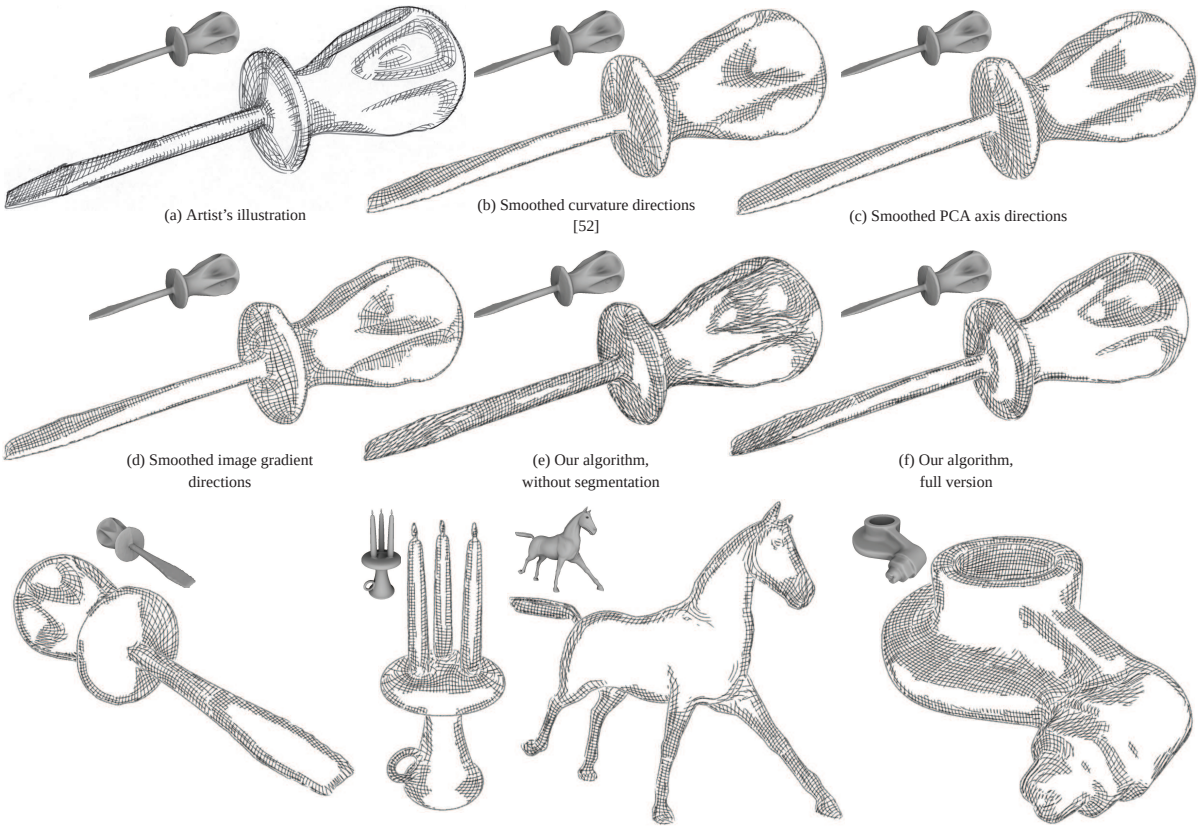


Figure 4.1: ***Data-driven line art illustrations generated with our algorithm and comparisons with alternative approaches.*** ***(a)*** *Artist's illustration of a screwdriver.* ***(b)*** *Illustration produced by the algorithm of Hertzmann and Zorin [52]. Manual thresholding of $\vec{N} \cdot \vec{V}$ is used to match the tone of the hand-drawn illustration and globally-smoothed principal curvature directions are used for the stroke orientations.* ***(c)*** *Illustration produced with the same algorithm, but using local PCA axes for stroke orientations before smoothing.* ***(d)*** *Illustration produced with the same algorithm, but using the gradient of image intensity for stroke orientations.* ***(e)*** *Illustration whose properties are learned by our algorithm for the screwdriver, but without using segmentation (i.e., orientations are learned by fitting a single model on the whole drawing and no contextual features are used for learning the stroke properties).* ***(f)*** *Illustration learnt by applying all steps of our algorithm. This result more faithfully matches the style of the input than the other approaches.* ***(g)*** *Results on new views and new objects.*

[151, 152] identify many principles of hand-drawn illustration, and describe methods for rendering polyhedral and smooth objects. Many other analytic formulae for hatching directions have been proposed, including principal curvature directions [28, 52, 110, 77], isophotes [76], shading gradients [130], other parametric curves [28] and user-defined direction fields (e.g., [107]). Stroke tone and density are normally proportional to depth, shading, or texture, or else based on user-defined prioritized stroke textures [110, 151, 152]. In these methods, each hatching property is computed by a hand-picked function of a single feature of shape, shading, or texture (e.g., proportional to depth or curvature). As a result, it is very hard for such approaches to capture the variations evident in artistic hatching styles (Figure 4.1). We propose the first method to learn hatching of 3D objects from examples.

There have been a few previous methods for transferring properties of artistic rendering by example. Hamel and Strothotte [45] automatically transfer user-tuned rendering parameters from one 3D object to another. Hertzmann et al. [50] transfer drawing and painting styles by example using non-parametric synthesis, given image data as input. This method maps directly from the input to stroke pixels. In general, the precise locations of strokes may be highly random—and thus hard to learn—and non-parametric pixel synthesis can make strokes become broken or blurred. Mertens et al. [95] transfer spatially-varying textures from source to target geometry using non-parametric synthesis. Jodoin et al. [63] model relative locations of strokes, but not conditioned on a target image or object. Kim et al. [135] employ texture similarity metrics to transfer stipple features between images. In contrast to the above techniques, our method maps to hatching properties, such as desired tone. Hence, though our method models a narrower range of artistic styles, it can model these styles much more accurately.

A few 2D methods have also been proposed for transferring styles of individual curves [33, 51, 67], a problem which is complementary to ours; such methods could be useful for the rendering step of our method.

A few previous methods learn synthesis of feature curves, such as contours and silhouettes.

Lum and Ma [92] use neural networks and Support Vector Machines to learn locations of feature curves. Cole et al. [18] study feature curve locations in hand-drawn artwork. They also fit a model of feature curve locations to a large training set of hand-drawn images. These methods focus on learning locations of feature curves, whereas we focus on hatching. Hatching exhibits substantially greater complexity and randomness than feature curves, since hatches form a network of overlapping curves of varying orientation, thickness, density, and cross-hatching level. Hatching also exhibits significant variation in artistic style.

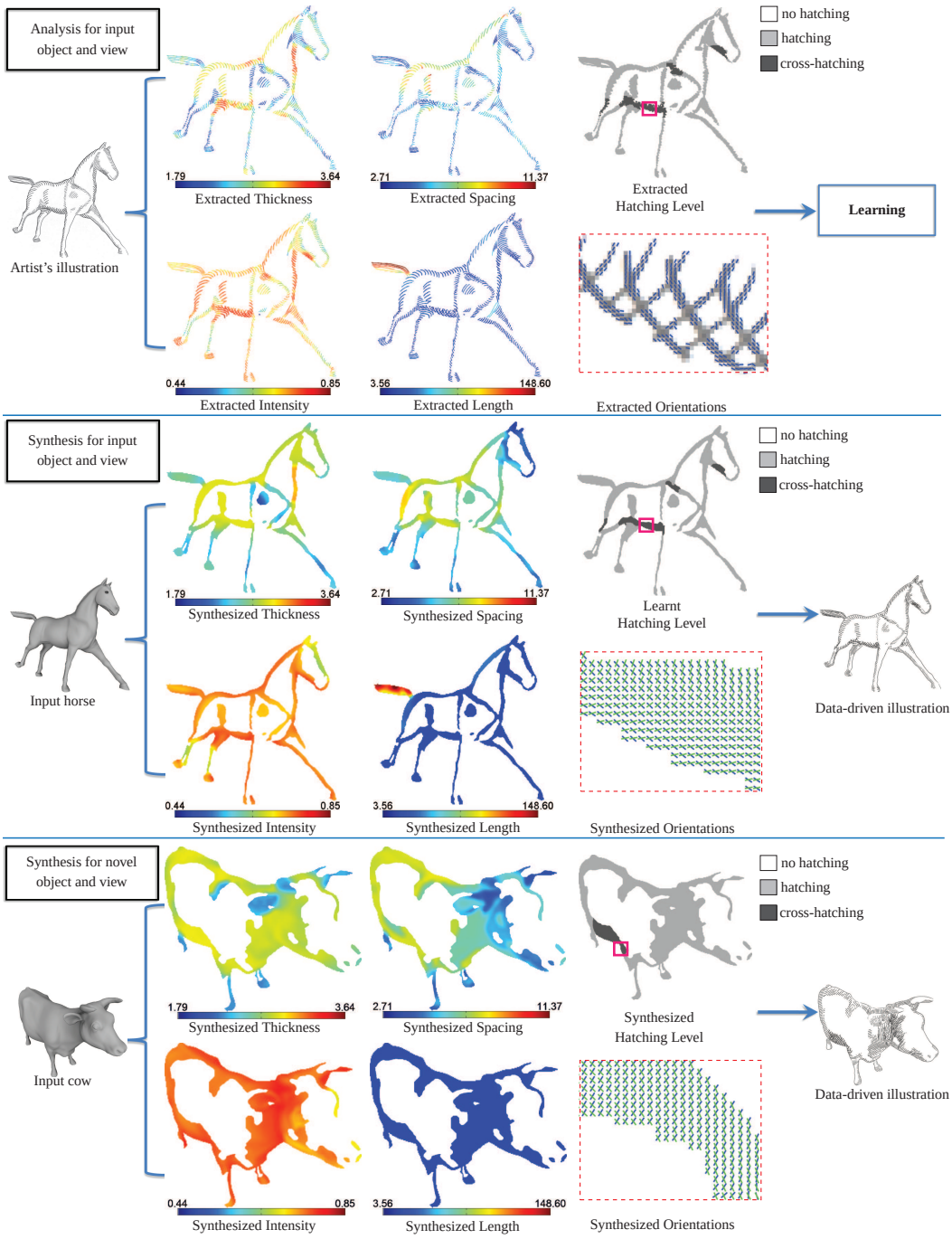


Figure 4.2: ***Extraction of hatching properties from a drawing, and synthesis for new drawings.*** ***Top:*** *The algorithm decomposes a given artist's illustration into a set of hatching properties: stroke thickness, spacing, hatching level, intensity, length, orientations. A mapping from input geometry is learned for each of these properties.* ***Middle:*** *Synthesis of the hatching properties for the input object and view. Our algorithm automatically separates and learns the hatching (blue-colored field) and cross-hatching fields (green-colored fields).* ***Bottom:*** *Synthesis of the hatching properties for a novel object and view.*

4.2 Overview

Our approach has two main phases. First, we analyze a hand-drawn pen-and-ink illustration of a 3D object, and learn a model of the artist's style. This model can then be applied to synthesize renderings of new views and new 3D objects.

Hatching properties Our goal is to model the way artists draw hatching strokes in line drawings of 3D objects. The actual placements of individual strokes exhibit much variation and apparent randomness, and so attempting to accurately predict individual strokes would be very difficult. However, we observe that the individual strokes themselves are less important than the overall appearance that they create together. Indeed, art instruction texts often focus on achieving particular qualities such as tone or shading (e.g., [44]). Hence, similar to previous work [151, 52], we model the rendering process in terms of a set of intermediate *hatching properties* related to tone and orientation. Each pixel containing a stroke in a given illustration is labeled with the following properties:

- **Hatching level ($h \in \{0,1,2\}$)** indicates whether a region contains no hatching, single hatching, or cross-hatching.
- **Orientation ($\phi_1 \in [0...\pi]$)** is the stroke direction in image space, with 180-degree symmetry.
- **Cross-hatching orientation ($\phi_2 \in [0..\pi]$)** is the cross-hatch direction, when present. Hatches and cross-hatches are not constrained to be perpendicular.
- **Thickness ($t \in \Re^+$)** is the stroke width.
- **Intensity ($I \in [0..1]$)** is how light or dark the stroke is.
- **Spacing ($d \in \Re^+$)** is the distance between parallel strokes.
- **Length ($l \in \Re^+$)** is the length of the stroke.

The decomposition of an illustration into hatching properties is illustrated in Figure 4.2 (top). In the analysis process, these properties are estimated from hand-drawn images, and models are learned. During synthesis, the learned model generates these properties as targets for stroke synthesis.

Modeling artists' orientation fields presents special challenges. Previous work has used local geometric rules for determining stroke orientations, such as curvature [52] or gradient of shading intensity [130]. We find that, in many hand-drawn illustrations, no local geometric rule can explain all stroke orientations. For example, in Figure 4.3, the strokes on the cylindrical part of the screwdriver's shaft can be explained as following the gradient of the shaded rendering, whereas the strokes on the flat end of the handle can be explained by the gradient of ambient occlusion ∇a. Hence, we segment the drawing into regions with distinct rules for stroke orientation. We represent this segmentation by an additional per-pixel variable:

- **Segment label ($c \in \mathscr{C}$)** is a discrete assignment of the pixel to one of a fixed set of possible segment labels $\mathscr{C}$.

Each set of pixels with a given label will use a single rule to compute stroke orientations. For example, pixels with label c_1 might use principal curvature orientations, and those with c_2 might use a linear combination of isophote directions and local PCA axes. Our algorithm also uses the labels to create contextual features (Section 4.4.2), which are also taken into account for computing the rest of the hatching properties. For example, pixels with label c_1 may have thicker strokes.

Features For a given 3D object and view, we define a set of features containing geometric, shading, and contextual information for each pixel, as described in Appendices B.2 and B.3. There are two types of features: "scalar" features $\mathbf{x}$ (Appendix B.2) and "orientation" features θ (Appendix B.3). The features include many object-space and image-space properties which

may be relevant for hatching, including features that have been used by previous authors for feature curve extraction, shading, and surface part labeling. The features are also computed at multiple scales, in order to capture varying surface and image detail. These features are inputs to the learning algorithms, which map from features to hatching properties.

Data acquisition and preprocessing The first step of our process is to gather training data and to preprocess it into features and hatching properties. The training data is based on a single drawing of a 3D model. An artist first chooses an image from our collection of rendered images of 3D objects. The images are rendered with Lambertian reflectance, distant point lighting, and spherical harmonic self-occlusion [131]. Then, the artist creates a line illustration, either by tracing over the illustration on paper with a light table, or in a software drawing package with a tablet. If the illustration is drawn on paper, we scan the illustration and align it to the rendering automatically by matching borders with brute force search. The artist is asked not to draw silhouette and feature curves, or to draw them only in pencil, so that they can be erased. The hatching properties (h, ϕ, t, I, d, l) for each pixel are estimated by the preprocessing procedure described in Appendix B.1.

Learning The training data comprise a single illustration with features $\mathbf{x}, \theta$ and hatching properties given for each pixel. The algorithm learns mappings from features to hatching properties (Section 4.4). The segmentation c and orientation properties ϕ are the most challenging to learn, because neither the segmentation c nor the orientation rules are immediately evident in the data; this represents a form of "chicken-and-egg" problem. We address this using a learning and clustering algorithm based on Mixtures-of-Experts (Section 4.4.1).

Once the input pixels are classified, a pixel classifier is learned using Conditional Random Fields with unary terms based on JointBoost (Section 4.4.2). Finally, each real-valued property is learned using boosting for regression (Section 4.4.3). We use boosting techniques for clas-

sification and regression since we do not know in advance which input features are the most important for different styles. Boosting can handle large number of features, can select the most relevant features, and has a fast sequential learning algorithm.

Synthesis A hatching style is transferred to a target novel view and/or object by first computing the features for each pixel, and then applying the learned mappings to compute the above hatching properties. A streamline synthesis algorithm [52] then places hatching strokes to match the synthesized properties. Examples of this process are shown in Figure 4.2.

(a) Estimated clusters using our mixture-of-experts model

(b) Learnt labeling with Joint Boosting

(c) Learnt labeling with Joint Boosting+CRF

(d) Synthesized labeling for another object

$\vec{f}_1 = \nabla a_2$
$\vec{f}_2 = .54(\vec{k}_{max,1}) + .46(\vec{r}_\perp)$

$\vec{f}_1 = .73(\nabla I_3) + .27(\vec{r})$
$\vec{f}_2 = .69(\vec{k}_{max,2}) + .31(\nabla I_{\perp,3})$

$\vec{f}_1 = .77(\vec{e}_{b,3}) + .23(\nabla I_3)$
$\vec{f}_2 = \vec{v}$

$\vec{f}_1 = .59(\vec{e}_{b,3}) + .41(\nabla(\vec{L}\cdot\vec{N})_3)$
$\vec{f}_2 = .63(\vec{e}_{a,3}) + .37(\nabla(\vec{L}\cdot\vec{N})_{\perp,3})$

$\vec{f}_1 = .88(\nabla a_3) + .12(\nabla(\vec{L}\cdot\vec{N})_3)$
$\vec{f}_2 = .45(\vec{k}_{max,2}) + .31(\nabla a_{\perp,3}) + .24(\vec{e}_{a,3})$

Figure 4.3: ***Clustering orientations.*** *The algorithm clusters stroke orientations according to different orientation rules. Each cluster specifies rules for hatching ($\vec{f}_1$) and cross-hatching ($\vec{f}_2$) directions. Cluster labels are color-coded in the figure, with rules shown below. The cluster labels and the orientation rules are estimated simultaneously during learning.* ***(a)*** *Inferred cluster labels for an artist's illustration of a screwdriver.* ***(b)*** *Output of the labeling step using the most likely labels returned by the Joint Boosting classifier alone.* ***(c)*** *Output of the labeling step using our full CRF model.* ***(d)*** *Synthesis of part labels for a novel object.* ***Rules:*** *In the legend, we show the corresponding orientation functions for each region. In all cases, the learned models use one to three features. Subscripts $\{1,2,3\}$ indicates the scale used to compute the field. The $\perp$ operator rotates the field by 90 degrees in image-space. The orientation features used here are: maximum and minimum principal curvature directions ($\vec{k}_{max}$, $\vec{k}_{min}$), PCA directions corresponding to first and second largest eigenvalue ($\vec{e}_a$, $\vec{e}_b$), fields aligned with ridges and valleys respectively ($\vec{r}$, $\vec{v}$), Lambertian image gradient (∇I), gradient of ambient occlusion (∇a), and gradient of $\vec{L}\cdot\vec{N}$ ($\nabla(\vec{L}\cdot\vec{N})$). Features that arise as 3D vectors are projected to the image plane. See Appendix B.3 for details.*

4.3 Synthesis Algorithm

The algorithm for computing a pen-and-ink illustration of a view of a 3D object algorithm is as follows. For each pixel of the target image, the features $\mathbf{x}$ and θ are first computed (Appendices B.2 and B.3). The segment label and hatching level are each computed as a function of the scalar features $\mathbf{x}$, using image segmentation and recognition techniques. Given these segments, orientation fields for the target image are computed by interpolation of the orientation features θ. Then, the remaining hatching properties are computed by learning functions of the scalar features. Finally, a streamline synthesis algorithm [52] renders strokes to match these synthesized properties. A streamline is terminated when it crosses an occlusion boundary, or the length grows past the value of the per-pixel target stroke length l, or violates the target stroke spacing d.

We now describe these steps in more detail. In Section 4.4, we will describe how the algorithm's parameters are learned.

4.3.1 Segmentation and labeling

For a given view of a 3D model, the algorithm first segments the image into regions with different orientation rules and levels of hatching. More precisely, given the feature set $\mathbf{x}$ for each pixel, the algorithm computes the per-pixel segment labels $c \in \mathscr{C}$ and hatching level $h \in \{0,1,2\}$. There are a few important considerations when choosing an appropriate segmentation and labeling algorithm. First, we do not know in advance which features in $\mathbf{x}$ are important, and so we must use a method that can perform feature selection. Second, neighboring labels are highly correlated, and performing classification on each pixel independently yields noisy results (Figure 4.3). Hence, we use a Conditional Random Field (CRF) recognition algorithm (Section 2.3.4), with JointBoost unary terms (Section 2.4.2). [69, 127, 142]. One such model

is learned for segment labels c, and a second for hatching level h. Learning these models is described in Section 4.4.2.

The CRF objective function includes unary terms that assess the consistency of pixels with labels, and pairwise terms that assess the consistency between labels of neighboring pixels. Inferring segment labels based on the CRF model corresponds to minimizing the following objective function:

$$E(\mathbf{c}) = \sum_i E_1(c_i; \mathbf{x}_i) + \sum_{i,j} E_2(c_i, c_j; \mathbf{x}_i, \mathbf{x}_j) \tag{4.1}$$

where E_1 is the unary term defined for each pixel i, and E_2 is the pairwise term defined for each pair of neighboring pixels $\{i, j\}$, where $j \in N(i)$ and $N(i)$ is defined using the 8-neighborhood of pixel i.

The unary term evaluates a JointBoost classifier that, given the feature set $\mathbf{x}_i$ for pixel i, determines the probability $P(c_i|\mathbf{x}_i)$ for each possible label c_i. The unary term is then:

$$E_1(c_i; \mathbf{x}) = -\log P(c_i|\mathbf{x}_i). \tag{4.2}$$

The mapping from features to probabilities $P(c_i|\mathbf{x}_i)$ is learned from the training data using the JointBoost algorithm [142].

The pairwise energy term scores the compatibility of adjacent pixel labels c_i and c_j, given their features $\mathbf{x}_i$ and $\mathbf{x}_j$. Let e_i be a binary random variable representing if the pixel i belongs to a boundary of hatching region or not. We define a binary JointBoost classifier that outputs the probability of boundaries of hatching regions $P(e|\mathbf{x})$ and compute the pairwise term as:

$$E_2(c_i, c_j; \mathbf{x}_i, \mathbf{x}_j) = -\ell \cdot I(c_i, c_j) \cdot (\log((P(e_i|\mathbf{x}_i) + P(e_j|\mathbf{x}_j))) + \mu) \tag{4.3}$$

where ℓ, μ are the model parameters and $I(c_i, c_j)$ is an indicator function that is 1 when $c_i \neq c_j$ and 0 when $c_i = c_j$. The parameter ℓ controls the importance of the pairwise term while μ contributes to eliminating tiny segments and smoothing boundaries.

Similarly, inferring hatching levels based on the CRF model corresponds to minimizing the following objective function:

$$E(\mathbf{h}) = \sum_i E_1(h_i; \mathbf{x}_i) + \sum_{i,j} E_2(h_i, h_j; \mathbf{x}_i, \mathbf{x}_j) \tag{4.4}$$

As above, the unary term evaluates another JointBoost classifier that, given the feature set $\mathbf{x}_i$ for pixel i, determines the probability $P(h_i|\mathbf{x}_i)$ for each hatching level $h \in \{0,1,2\}$. The pairwise term is also defined as:

$$E_2(h_i, h_j; \mathbf{x}_i, \mathbf{x}_j) = -\ell \cdot I(h_i, h_j) \cdot (\log((P(e_i|\mathbf{x}_i) + P(e_j|\mathbf{x}_j))) + \mu) \tag{4.5}$$

with the same values for the parameters of ℓ, μ as above.

The most probable labeling is the one that minimizes the CRF objective function $E(\mathbf{c})$ and $E(\mathbf{h})$, given their learned parameters. The CRFs are optimized using alpha-expansion graph-cuts [12]. Details of learning the JointBoost classifiers and ℓ, μ are given in Section 4.4.2.

4.3.2 Computing orientations

Once the per-pixel segment labels c and hatching levels h are computed, the per-pixel orientations ϕ_1 and ϕ_2 are computed. The number of orientations to be synthesized is determined by h. When $h = 0$ (no hatching), no orientations are produced. When $h = 1$ (single hatching), only ϕ_1 is computed, and, when $h = 2$ (cross-hatching), ϕ_2 is computed as well.

Orientations are computed by regression on a subset of the orientation features θ for each pixel. Each cluster c may use a different subset of features. The features used by a segment are indexed by a vector σ, i.e., the features indices are $\sigma(1)$, $\sigma(2)$, ..., $\sigma(k)$. Each orientation feature represents an orientation field in image space, such as the image projection of principal curvature directions. In order to respect 2-symmetries in orientation, a single orientation θ is transformed to a vector as

$$\mathbf{v} = [\cos(2\theta), \sin(2\theta)]^T \tag{4.6}$$

The output orientation function is expressed as a weighted sum of selected orientation features.

$$f(\theta;\mathbf{w}) = \sum_k w_{\sigma(k)} \mathbf{v}_{\sigma(k)} \tag{4.7}$$

where $\sigma(k)$ represents the index to the k-th orientation feature in the subset of selected orientation features, $v_{\sigma(k)}$ is its vector representation, and $\mathbf{w}$ is a vector of weight parameters. There is an orientation function $f(\theta;\mathbf{w}_{c,1})$ for each label $c \in \mathscr{C}$ and, if the class contains cross-hatching regions, it has an additional orientation function $f(\theta;\mathbf{w}_{c,2})$ for determining the cross-hatching directions. The resulting vector is computed to an image-space angle as $\phi = \text{atan2}(y,x)/2$.

The weights $\mathbf{w}$ and feature selection σ is learned by the gradient-based boosting for regression algorithm of Zemel and Pitassi [155]. The learning of the parameters and the feature selection is described in Section 4.4.1.

4.3.3 Computing real-valued properties

The remaining hatching properties are real-valued quantities. Let y be a feature to be synthesized on a pixel with feature set $\mathbf{x}$. We use multiplicative models of the form:

$$y = \prod_k (a_k x_{\sigma(k)} + b_k)^{\alpha_k} \tag{4.8}$$

where $x_{\sigma(k)}$ is the index to the k-th scalar feature from $\mathbf{x}$. The use of a multiplicative model is inspired by Goodwin et al. [43], who propose a model for stroke thickness that can be approximated by a product of radial curvature and inverse depth. The model is learned in logarithmic domain, which reduces the problem to learning the weighted sum:

$$\log(y) = \sum_k \alpha_k \log(a_k x_{\sigma(k)} + b_k) \tag{4.9}$$

Learning the parameters $\alpha_k, a_k, b_k, \sigma(k)$ is again performed using gradient-based boosting [155], as described in Section 4.4.3.

4.4 Learning

We now describe how to learn the parameters of the functions used in the synthesis algorithm described in the previous section.

4.4.1 Learning Segmentation and Orientation Functions

In our model, the hatching orientation for a single-hatching pixel is computed by first assigning the pixel to a cluster c, and then applying the orientation function $f(\theta;\mathbf{w}_c)$ for that cluster. If we knew the clustering in advance, then it would be straightforward to learn the parameters $\mathbf{w}_c$ for each pixel. However, neither the cluster labels nor the parameters $\mathbf{w}_c$ are present in the training data. In order to solve this problem, we develop a technique inspired by Expectation-Maximization for Mixtures-of-Experts (described in Section 2.2.5), but specialized to handle the particular issues of hatching.

The input to this step is a set of pixels from the source illustration with their corresponding orientation feature set θ_i, training orientations ϕ_i, and training hatching levels h_i. Pixels containing intersections of strokes or no strokes are not used. Each cluster c may contain either single-hatching or cross-hatching. Single-hatch clusters have a single orientation function (Equation 4.7), with unknown parameters $\mathbf{w}_{c1}$. Clusters with cross-hatches have two subclusters, each with an orientation function with unknown parameters $\mathbf{w}_{c1}$ and $\mathbf{w}_{c2}$. The two orientation functions are not constrained to produce directions orthogonal to each other. Every source pixel must belong to one of the top-level clusters, and every pixel belonging to a cross-hatching cluster must belong to one of its subclusters.

For each training pixel i, we define a labeling probability γ_{ic} indicating the probability that pixel i lies in top-level cluster c, such that $\sum_c \gamma_{ic} = 1$. Also, for each top-level cluster, we define a subcluster probability β_{icj}, where $j \in \{1,2\}$, such that $\beta_{ic1} + \beta_{ic2} = 1$. The probability β_{icj}

measures how likely the stroke orientation at pixel i corresponds to a hatching or cross-hatching direction. Single-hatching clusters have $\beta_{ic2} = 0$. The probability that pixel i belongs to the subcluster indexed by $\{c, j\}$ is $\gamma_{ic}\beta_{icj}$.

The labeling probabilities are modeled based on a mixture-of-Gaussians distribution [11]:

$$\gamma_{ic} = \frac{\pi_c \exp(-r_{ic}/2s)}{\sum_c \pi_c \exp(-r_{ic}/2s)} \tag{4.10}$$

$$\beta_{icj} = \frac{\pi_{cj} \exp(-r_{icj}/2s_c)}{\pi_{c1} \exp(-r_{ic1}/2s_c) + \pi_{c2} \exp(-r_{ic2}/2s_c)} \tag{4.11}$$

where π_c, π_{cj} are the mixture coefficients, s, s_c are the variances of the corresponding Gaussians, r_{icj} is the residual for pixel i with respect to the orientation function j in cluster c, and r_{ic} is defined as follows:

$$r_{ic} = \min_{j \in \{1,2\}} ||\mathbf{u}_i - f(\theta_i; \mathbf{w}_{cj})||^2 \tag{4.12}$$

where $\mathbf{u}_i = [\cos(2\phi_i), \sin(2\phi_i)]^T$.

The process begins with an initial set of labels γ, β, and $\mathbf{w}$, and then alternates between updating two steps: the *model update step* where the orientation functions, the mixture coefficients and variances are updated, and the *label update step* where the labeling probabilities are updated.

Model update Given the labeling, orientation functions for each cluster are updated by minimizing the boosting error function, described in Section 2.4.3, using the initial per-pixel weights $\alpha_i = \gamma_{ic}\beta_{icj}$.

In order to avoid overfitting, a set of holdout-validation pixels are kept for each cluster. This set is found by selecting rectangles of random size and marking their containing pixels as holdout-validation pixels. Our algorithm stops when 25% of the cluster pixels are marked as holdout-validation pixels. The holdout-validation pixels are not considered for fitting the weight vector $\mathbf{w}_{cj}$. At each boosting iteration, our algorithm measures the holdout-validation error measured

on these pixels. It terminates the boosting iterations when the holdout-validation error reaches a minimum. This helps avoid overfitting the training orientation data.

During this step, we also update the mixture coefficients and variances of the gaussians in the mixture model, so that the data likelihood is maximized in this step [11]:

$$\pi_c = \sum_i \gamma_{ic}/N, \quad s = \sum_{ic} \gamma_{ic} r_{ic}/N \tag{4.13}$$

$$\pi_{cj} = \sum_i \beta_{icj}/N, \quad s_c = \sum_{ij} \beta_{icj} r_{icj}/N \tag{4.14}$$

where N is the total number of pixels with training orientations.

Label update Given the estimated orientation functions from the above step, the algorithm computes the residual for each model and each orientation function. Median filtering is applied to the residuals, in order to enforce spatial smoothness: r_{ic} is replaced with the value of the median of r_{*c} in the local image neighborhood of pixel i (with radius equal to the local spacing S_i). Then the pixel labeling probabilities are updated according to Equations 4.10 and 4.11.

Initialization The clustering is initialized by a constrained mean-shift clustering process with a flat kernel, similar to constrained K-means [148]. The constraints arise from a region-growing strategy to enforce spatial continuity of the initial clusters. Each cluster grows by considering randomly-selected seed pixels in their neighborhood and adding them only if the difference between their orientation angle and the cluster's current mean orientation is below a threshold. In the case of cross-hatching clusters, the minimum difference between the two mean orientations is used. The threshold is automatically selected once during pre-processing by taking the median of each pixel's local neighborhood orientation angle differences. The process is repeated for new pixels and the cluster's mean orientation(s) are updated at each iteration. Clusters composed of more than 10% cross-hatch pixels are marked as cross-hatching clusters; the rest are marked as single-hatching clusters. The initial assignment of pixels to clusters gives a binary-valued initialization for γ. For cross-hatch pixels, if more than half the pixels in the cluster are

assigned to orientation function $\mathbf{w}_{k2}$, our algorithm swaps $\mathbf{w}_{k1}$ and $\mathbf{w}_{k2}$. This ensures that the first hatching direction will correspond to the dominant orientation. This aids in maintaining orientation field consistency between neighboring regions.

An example of the resulting clustering for an artist's illustration of screwdriver is shown in Figure 4.3 (a). We also include the functions learned for the hatching and cross-hatching orientation fields used in each resulting cluster.

4.4.2 Learning Labeling with CRFs

Once the training labels are estimated, we learn a procedure to transfer them to new views and objects. Here we describe the procedure to learn the Conditional Random Field model of Equation 4.1 for assigning segment labels to pixels as well as the Conditional Random Field of Equation 4.4 for assigning hatching levels to pixels.

Learning to segment and label Our goal here is to learn the parameters of the CRF energy terms (Equation 4.1). The input is the scalar feature set $\tilde{\mathbf{x}}_i$ for each stroke pixel i (described in Appendix B.2) and their associated labels c_i, as extracted in the previous step. Following [143, 126, 69], the parameters of the unary term are learned by running a cascade of JointBoost classifiers. The cascade is used to obtain contextual features which capture information about the relative distribution of cluster labels around each pixel. The cascade of classifiers is trained as follows.

The method begins with an initial JointBoost classifier using an initial feature set $\tilde{\mathbf{x}}$, containing the geometric and shading features, described in Appendix B.2. The classifier is applied to produce the probability $P(c_i|\tilde{\mathbf{x}}_i)$ for each possible label c_i given the feature set $\tilde{\mathbf{x}}_i$ of each pixel i. These probabilities are then binned in order to produce contextual features. In particular, for each pixel, the algorithm computes a histogram of these probabilities as a function of geodesic

distances from it:

$$p_i^c = \sum_{j: d_b \leq \text{dist}(i,j) < d_{b+1}} P(c_j)/N_b \tag{4.15}$$

where the histogram bin b contains all pixels j with geodesic distance range $[d_b, d_{b+1}]$ from pixel i, and N_b is the total number of pixels in the histogram bin b. The geodesic distances are computed on the mesh and projected to image space. 4 bins are used, chosen in logarithmic space. The bin values p_i^c are normalized to sum to 1 per pixel. The total number of bins are $4|C|$. The values of these bins are used as contextual features, which are concatenated into $\tilde{\mathbf{x}}_i$ to form a new scalar feature set $\mathbf{x}_i$. Then, a second JointBoost classifier is learned, using the new feature set $\mathbf{x}$ as input and outputting updated probabilities $P(c_i|\mathbf{x}_i)$. These are used in turn to update the contextual features. The next classifier uses the contextual features generated by the previous one, and so on. Each JointBoost classifier is initialized with uniform weights and terminates when the holdout-validation error reaches to a minimum. The holdout-validation error is measured on pixels that are contained in random rectangles on the drawing, selected as above. The cascade terminates when the holdout-validation error of a JointBoost classifier is increased with respect to the holdout-validation error of the previous one. The unary term is defined based on the probabilities returned by the latter classifier.

To learn the pairwise term of Equation 4.3, the algorithm needs to estimate the probability of boundaries of hatching regions $P(e|\mathbf{x})$, which also serve as evidence for label boundaries. First, we observe that segment boundaries are likely to occur at particular parts of an image; for example, pixels separate by an occluding and suggestive contour are much less likely to be in the same segment as two pixels that are adjacent on the surface. For this reason, we define a binary JointBoost classifier, which maps to probabilities of boundaries of hatching regions for each pixel, given the subset of its features $\mathbf{x}$ computed from the feature curves of the mesh (see Appendix B.2). In this binary case, JointBoost reduces to an earlier algorithm called GentleBoost [35]. The training data for this pairwise classifier are supplied by the marked boundaries of hatching regions of the source illustration (see Appendix B.1); pixels that are

marked as boundaries have $e = 1$, otherwise $e = 0$. The classifier is initialized with more weight give to the pixels that contain boundaries of hatching level regions, since the training data contains many more non-boundary pixels. More specifically, if N_B are the total number of boundary pixels, and N_{NB} is the number of non-boundary pixels, then the weight is N_{NB}/N_B for boundary pixels and 1 for the rest. The boosting iterations terminate when the hold-out validation error measured on validation pixels (selected as above) is minimum.

Finally, the parameters ℓ and μ are optimized by maximizing the following energy term:

$$E_S = \sum_{i:c_i \neq c_j, j \in N(i)} P(e_i|\mathbf{x}) \tag{4.16}$$

where $N(i)$ is the 8-neighborhood of pixel i, and c_i, c_j are the labels for each pair of neighboring pixels i, j inferred using the CRF model of Equation 4.1 based on the learned parameters of its unary and pairwise classifier and using different values for ℓ, μ. This optimization attempts to "push" the segment label boundaries to be aligned with pixels that have higher probability to be boundaries. The energy is maximized using Matlab's implementation of Preconditioned Conjugate Gradient with numerically-estimated gradients.

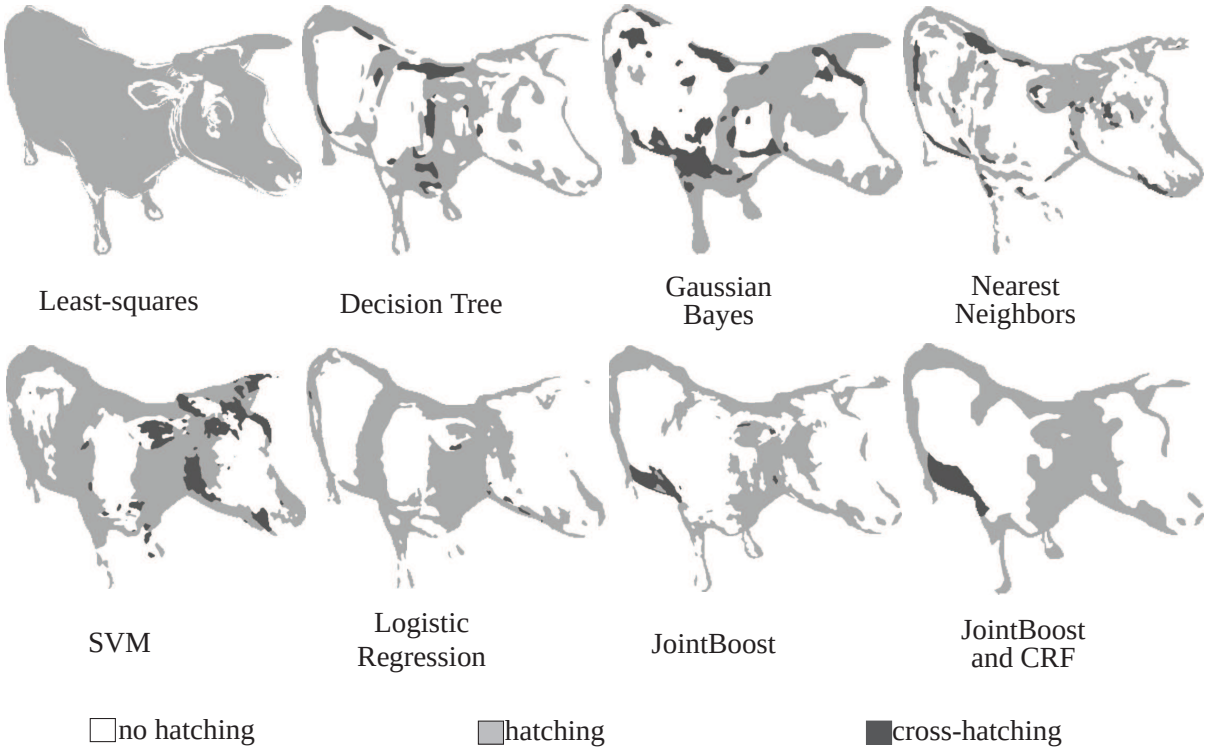


Figure 4.4: ***Comparisons of various classifiers for learning the hatching level.*** *The training data is the extracted hatching level on the horse of Figure 4.2 and feature set* **x**. ***Left to right:*** *least-squares for classification, decision tree (Matlab's implementation based on Gini's diversity index splitting criterion), Gaussian Naive Bayes, Nearest Neighbors, Support Vector Machine, Logistic Regression, Joint Boosting, Joint Boosting and Conditional Random Field (full version of our algorithm). The regularization parameters of SVMs, Gaussian Bayes, Logistic Regression are estimated by hold-out validation with the same procedure as in our algorithm.*

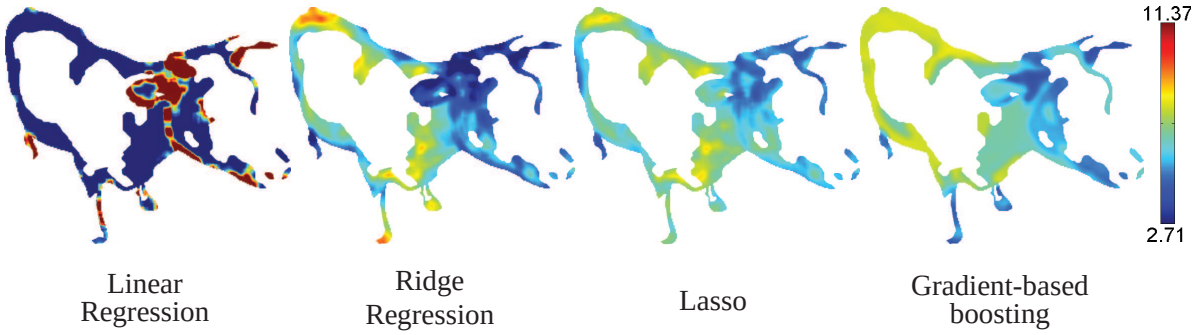


Figure 4.5: *Comparisons of the generalization performance of various techniques for regression for the stroke spacing. The same training data are provided to the techniques based on the extracted spacing on the horse of Figure 4.2 and feature set* **x**. ***Left to right:*** *Linear regression (least-squares without regularization), ridge regression, Lasso, gradient-based boosting. Fitting a model on such very high-dimensional space without any sparsity prior yields very poor generalization performance. Gradient-based boosting gives more reasonable results than ridge regression or Lasso, especially on the legs of the cow, where the predicted spacing values seem to be more consistent with the training values on the legs of the horse (see Figure 4.2). The regularization parameters of Ridge Regression and Lasso are estimated by hold-out validation with the same procedure as in our algorithm.*

Learning to generate hatching levels The next step is to learn the hatching levels $h \in \{0,1,2\}$. The input here is the hatching level h_i per pixel contained inside the rendered area (as extracted during the pre-processing step (Appendix B.1) together with their full feature set $\mathbf{x}_i$ (including the contextual features as extracted above).

Our goal is to compute the parameters of the second CRF model used for inferring the hatching levels (Equation 4.4). Our algorithm first uses a JointBoost classifier that maps from the feature set $\mathbf{x}$ to the training hatching levels h. The classifier is initialized with uniform weights and terminates the boosting rounds when the hold-out validation error is minimized (the hold-out validation pixels are selected as above). The classifier outputs the probability $P(h_i|\mathbf{x}_i)$, which is used in the unary term of the CRF model. Finally, our algorithm uses the same pairwise term parameters trained with the CRF model of the segment labels to rectify the boundaries of the hatching levels.

Examples comparing our learned hatching algorithm to several alternatives are shown in Figure 4.4.

4.4.3 Learning Real-Valued Stroke Properties

Thickness, intensity, length, and spacing are all positive, real-valued quantities, and so the same learning procedure is used for each one in turn. The input to the algorithm are the values of the corresponding stroke properties, as extracted in the preprocessing step (Section B.1) and the full feature set $\mathbf{x}_i$ per pixel.

The multiplicative model of Equation 4.8 is used to map the features to the stroke properties. The model is learned in the log-domain, so that it can be learned as a linear sum of log functions. The model is learned with gradient-based boosting for regression (Section 2.4). The weights for the training pixels are initialized as uniform. As above, the boosting iterations stop

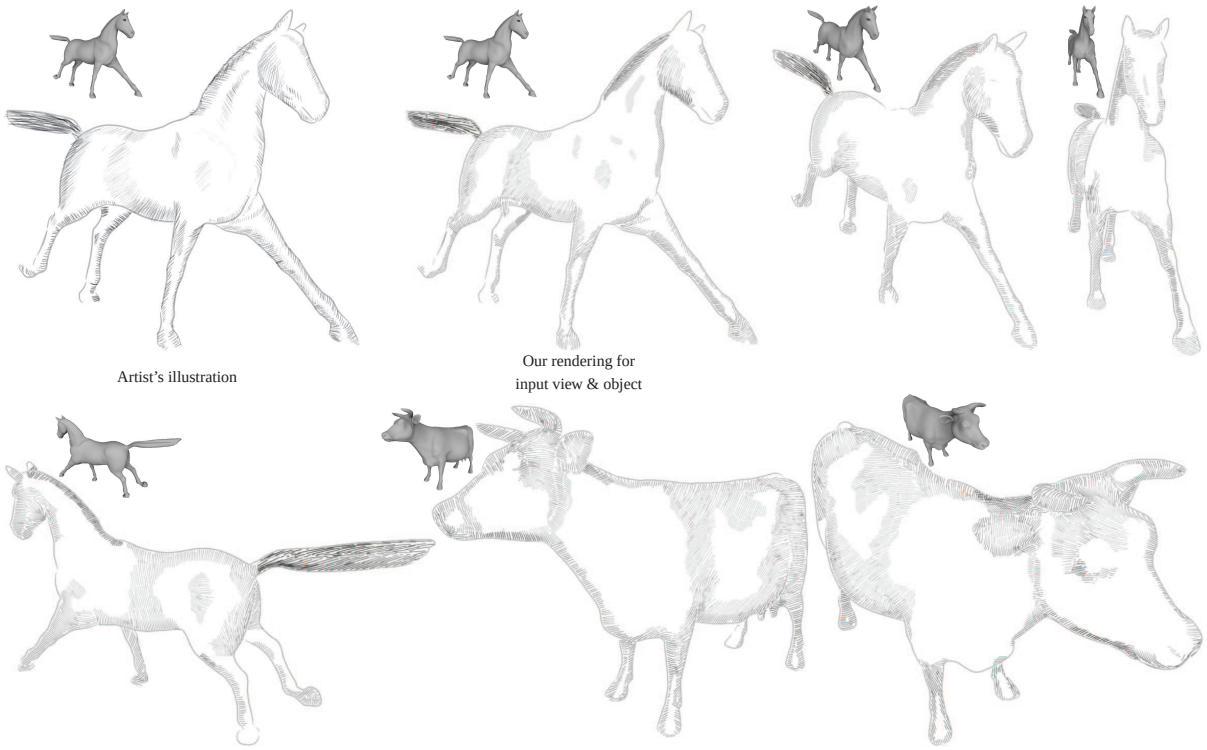


Figure 4.6: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a horse. Rendering of the model with our learnt style. Renderings of new views and new objects.*

when the holdout-validation measured on randomly selected validation pixels is minimum.

Examples comparing our method to several alternatives are shown in Figure 4.5.

4.5 Results

The figures throughout our paper show synthesized line drawings of novel objects and views with our learning technique (Figures 4.1, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14). As can be seen in the examples, our method captures several aspects of the artist's drawing style, better than alternative previous approaches (Figure 4.1). Our algorithm adapts to different styles of drawing and successfully synthesizes them for different objects and views. For example,

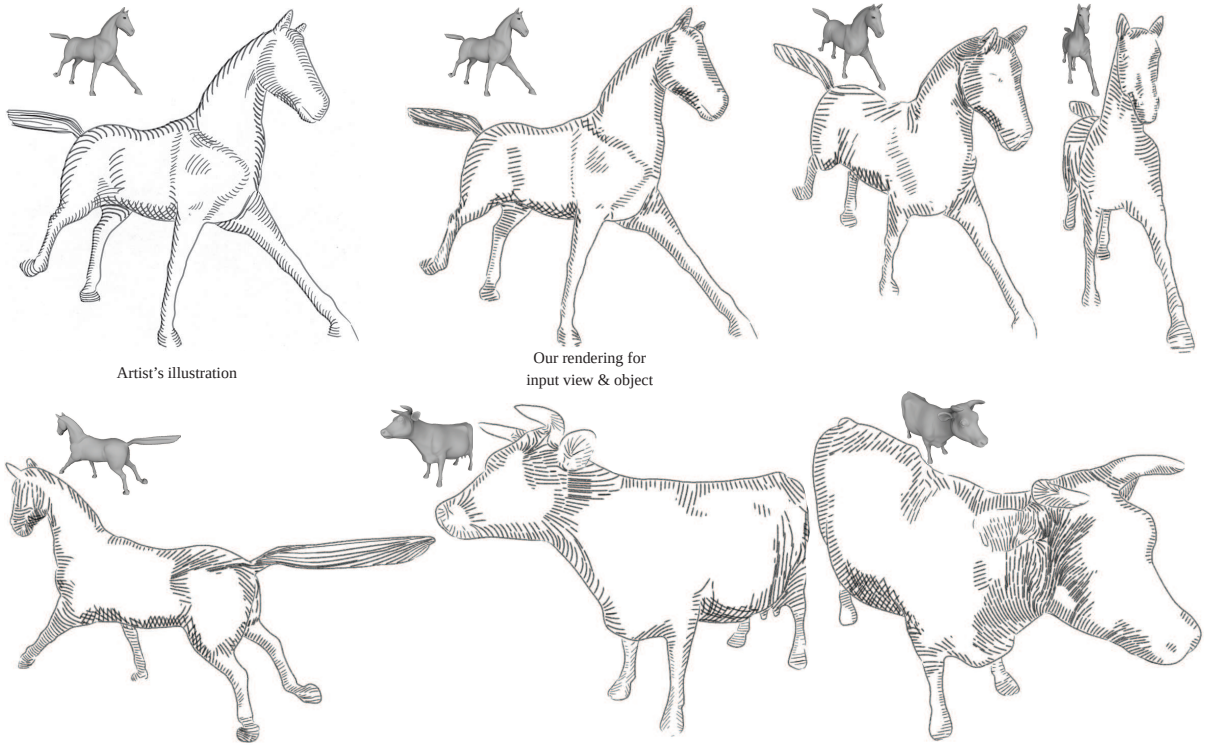


Figure 4.7: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a horse with a different style than 4.6. Rendering of the model with our learnt style. Renderings of new views and new objects.*

Figures 4.6 and 4.7 show different styles of illustrations for the same horse, applied to new views and objects. Figure 4.14 shows more examples of synthesis with various styles and objects.

However, subtleties are sometimes lost. For example, in Figure 4.12, the face is depicted with finer-scale detail than the clothing, which cannot be captured in our model. In Figure 4.13, our method loses variation in the character of the lines, and depiction of important details such as the eye. One reason for this is that the stroke placement algorithm attempts to match the target hatching properties, but does not optimize to match a target tone. These variations may also depend on types of parts (e.g., eyes versus torsos), and could be addressed given part labels [69]. Figure 4.11 exhibits randomness in stroke spacing and width that is not modeled by our technique.

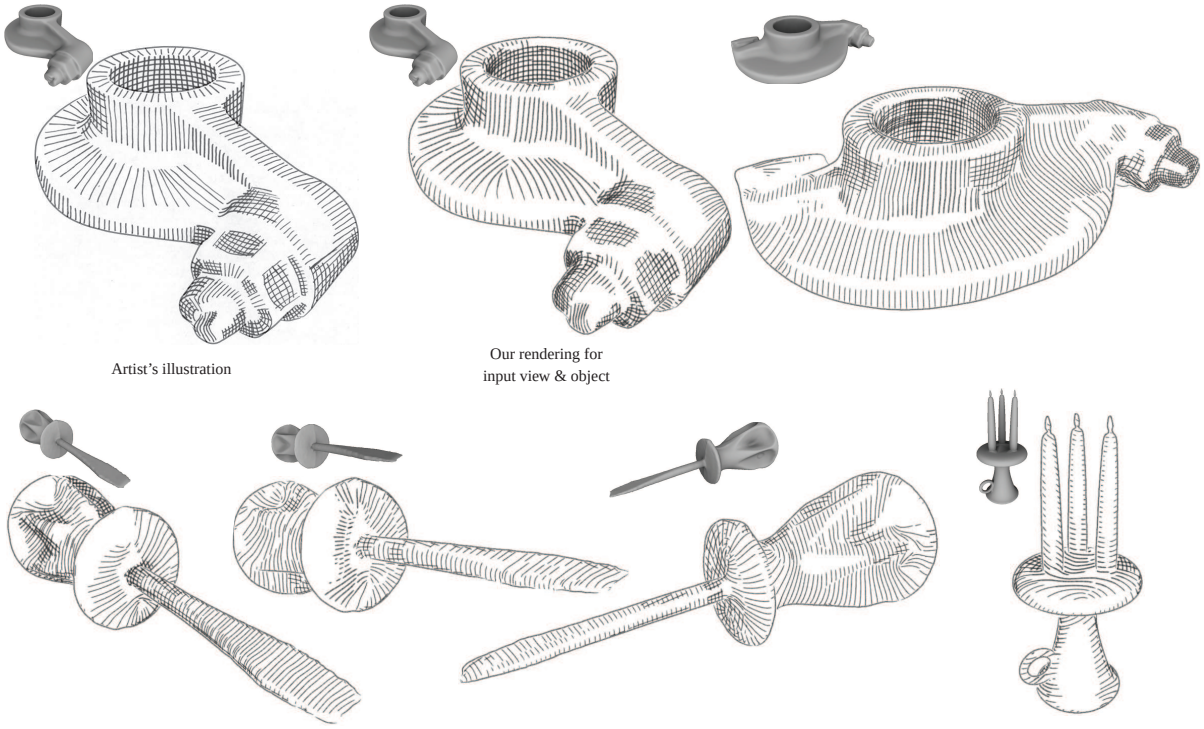


Figure 4.8: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a rocker arm. Rendering of the model with our learnt style. Renderings of new views and new objects.*

Figure 4.9: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a pitcher. Rendering of the model with our learnt style. Renderings of new views and new objects.*

Selected features We show the frequency of orientation features selected by gradient-based boosting and averaged over all our nine drawings in Figure 4.15. Fields aligned with principal curvature directions as well as local principal axes (corresponding to candidate local planar

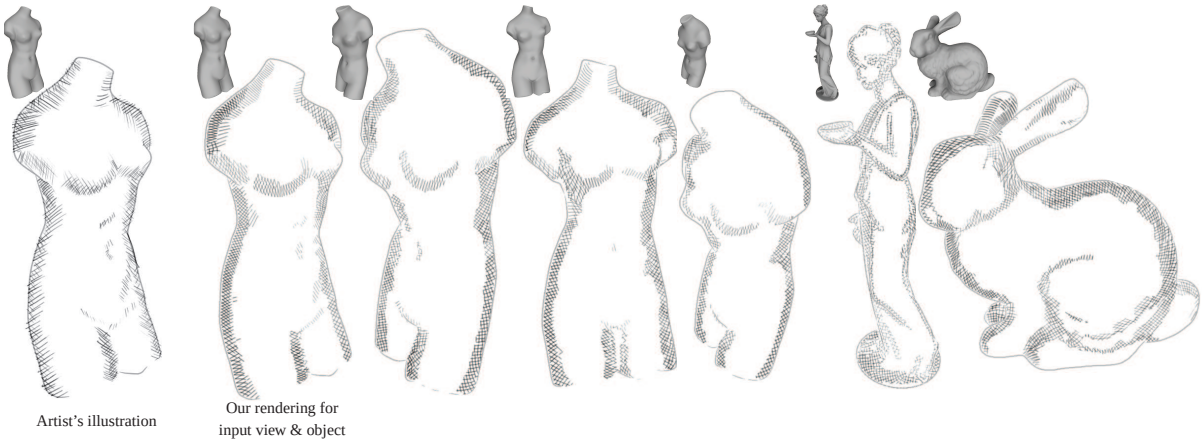


Figure 4.10: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a Venus statue. Rendering of the model with our learnt style. Renderings of new views and new objects.*

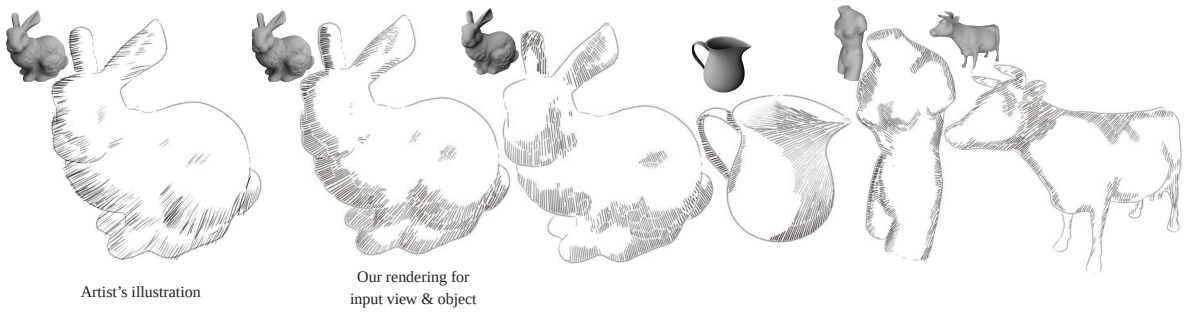


Figure 4.11: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a bunny using a particular style; hatching orientations are mostly aligned with point light directions. Rendering of the model with our learnt style. Renderings of new views and new objects.*

symmetry axes) play very important roles for synthesizing the hatching orientations. Fields aligned with suggestive contours, ridges and valleys are also significant for determining orientations. Fields based on shading attributes have moderate influence.

We show the frequency of scalar features averaged selected by boosting and averaged over all our nine drawings in Figure 4.16 for learning the hatching level, thickness, spacing, intensity, length, and segment label. Shape descriptor features (based on PCA, shape contexts, shape

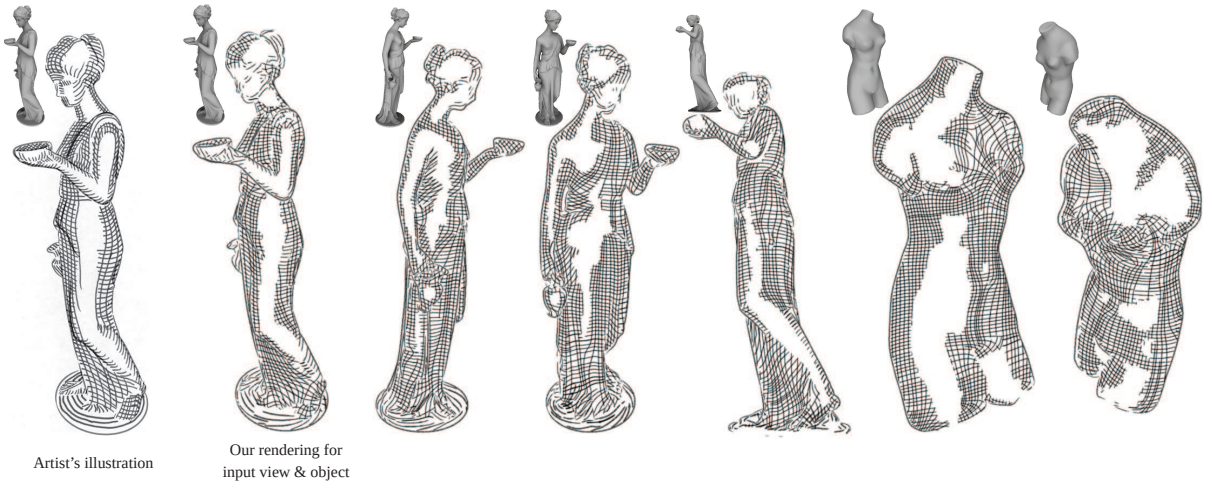


Figure 4.12: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a statue. Rendering of the model with our learnt style. Renderings of new views and new objects.*

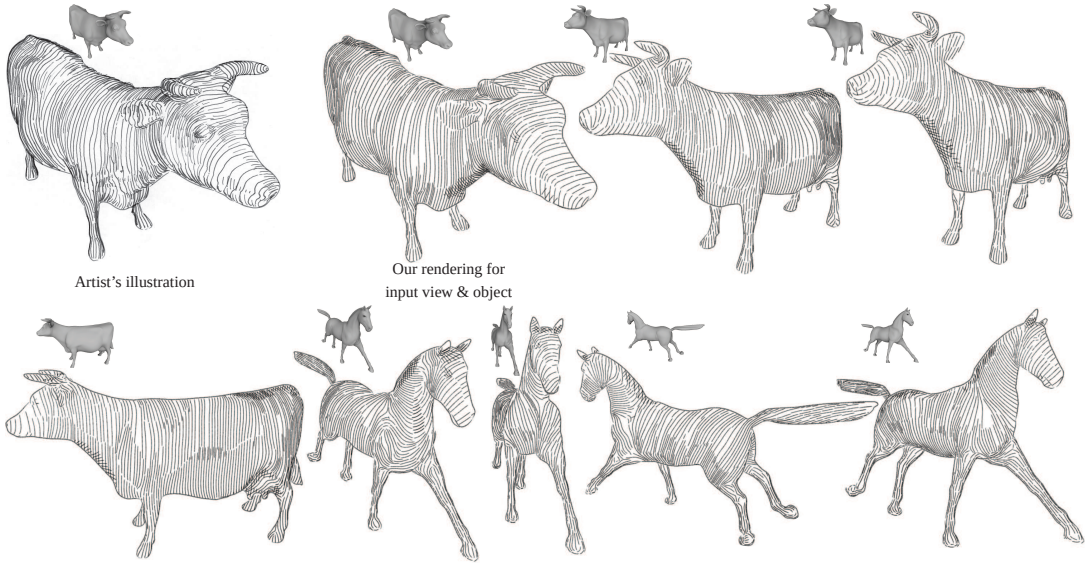


Figure 4.13: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a cow. Rendering of the model with our learnt style. Renderings of new views and new objects.*

diameter, average geodesic distance, distance from medial surface, contextual features) seem to have large influence on all the hatching properties. This means that the choice of tone is probably influenced by the type of shape part the artist draws. The segment label is mostly de-

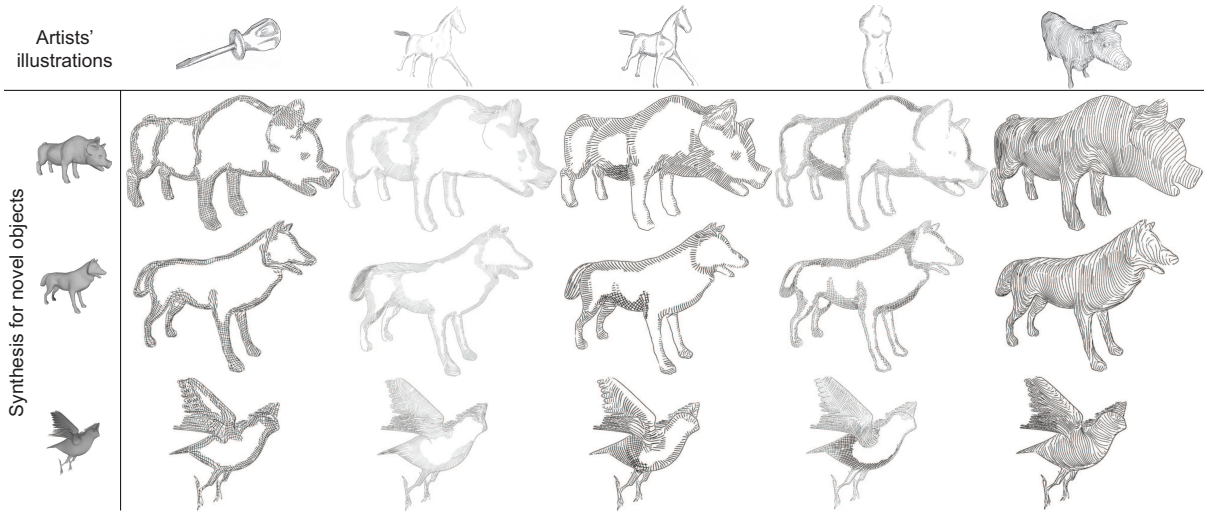


Figure 4.14: *Data-driven line art illustrations generated with our algorithm based on the learned styles from the artists' drawings in Figures 4.1, 4.6, 4.7, 4.10, 4.13.*

termined by the shape descriptor features, which is consistent with the previous work on shape segmentation and labeling [69]. The hatching level is mostly influenced by image intensity, $\vec{V} \cdot \vec{N}$, $\vec{L} \cdot \vec{N}$. The stroke thickness is mostly affected by shape descriptor features, curvature, $\vec{L} \cdot \vec{N}$, gradient of image intensity, the location of feature lines, and, finally, depth. Spacing is mostly influenced by shape descriptor features, curvature, derivatives of curvature, $\vec{L} \cdot \vec{N}$, and $\vec{V} \cdot \vec{N}$. The intensity is influenced by shape descriptor features, image intensity, $\vec{V} \cdot \vec{N}$, $\vec{L} \cdot \vec{N}$, depth, and the location of feature lines. The length is mostly determined by shape descriptor features, curvature, radial curvature, $\vec{L} \cdot \vec{N}$, image intensity and its gradient, and location of feature lines (mostly suggestive contours).

However, it is important to note that different features are learned for different input illustrations. For example, in Figure 4.11, the light directions mostly determine the orientations, which is not the case for the rest of the drawings. We include histograms of the frequency of orientation and scalar features used for each of the drawing in the supplementary material.

Computation time In each case, learning a style from a source illustration takes 5 to 10 hours on a laptop with Intel i7 processor. Most of the time is consumed by the orientation and

clustering step (4.4.1) (about 50% of the time for the horse), which is implemented in Matlab. Learning segment labels and hatching levels (4.4.2) represents about 25% of the training time (implemented in C++) and learning stroke properties 4.4.3 takes about 10% of the training time (implemented in Matlab). The rest of the time is consumed for extracting the features (implemented in C++) and training hatching properties (implemented in Matlab). We note that our implementation is currently far from optimal, hence, running times could be improved. Once the model of the style is learned, it can be applied to different novel data. Given the predicted hatching and cross-hatching orientations, hatching level, thickness, intensity, spacing and stroke length at each pixel, our algorithm traces streamlines over the image to generate the final pen-and-ink illustration. Synthesis takes 30 to 60 minutes. Most of the time (about 60%) is consumed here for extracting the features. The implementation for feature extraction and tracing streamlines are also far from optimal.

4.6 Summary and Future Work

Ours is the first method to generate predictive models for synthesizing detailed line illustrations from examples. We model line illustrations with a machine learning approach using a set of features suspected to play a role in the human artistic process. The complexity of man-made illustrations is very difficult to reproduce; however, we believe our work takes a step towards replicating certain key aspects of the human artistic process. Our algorithm generalizes to novel views as well as objects of similar morphological class.

There are many aspects of hatching styles that we do not capture, including: stroke textures, stroke tapering, randomness in strokes (such as wavy or jittered lines), cross-hatching with more than two hatching directions, style of individual strokes, and continuous transitions in hatching level. Interactive edits to the hatching properties could be used to improve our results, similarly to [120].

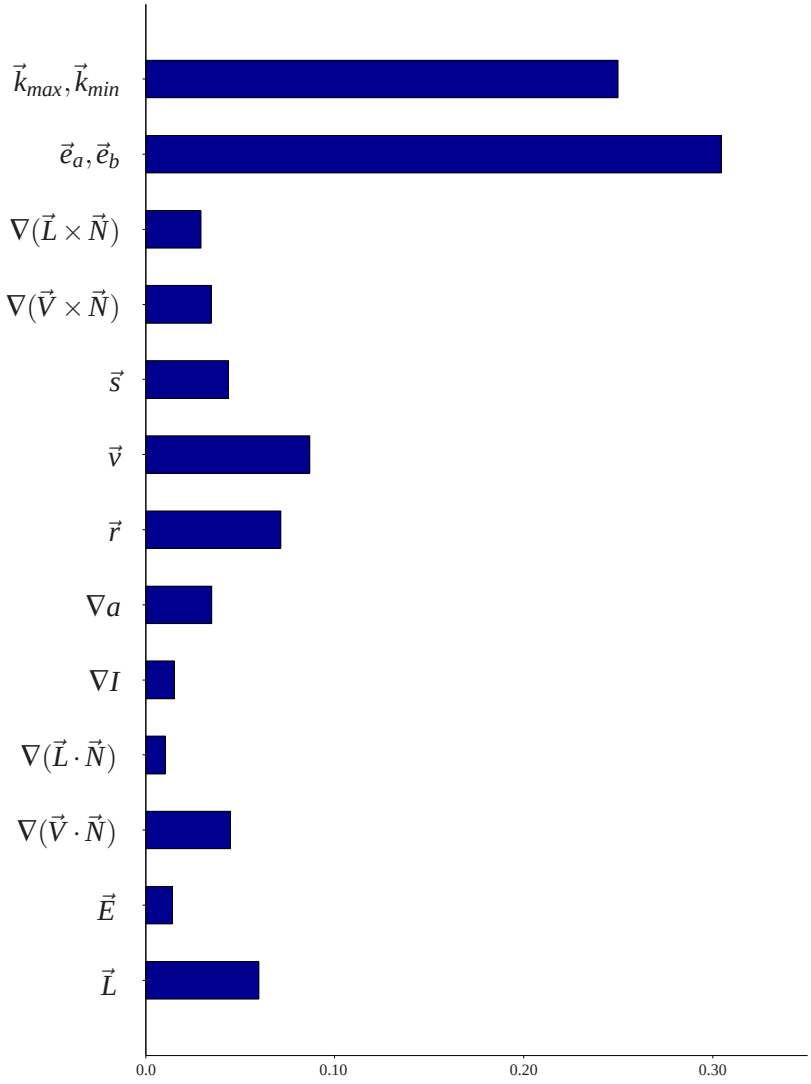


Figure 4.15: ***Frequency of the first three orientation features selected by gradient-based boosting for learning the hatching orientation fields.*** *The frequency is averaged over all our nine training drawings (Figures 4.1, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13). The contribution of each feature is also weighted by the total segment area where it is used. The orientation features are grouped based on their type: principal curvature directions ($\vec{k}_{max}, \vec{k}_{min}$), local principal axes directions ($\vec{e}_a, \vec{e}_b$), $\nabla(\vec{L} \times \vec{N})$, $\nabla(\vec{V} \times \vec{N})$, directions aligned with suggestive contours ($\vec{s}$), valleys ($\vec{v}$), ridges ($\vec{r}$), gradient of ambient occlusion (∇a), gradient of image intensity (∇I), gradient of $(\vec{L} \cdot \vec{N})$, gradient of $(\vec{V} \cdot \vec{N})$, vector irradiance ($\vec{E}$), projected light direction ($\vec{L}$).*

Since we learn from a single training drawing, the generalization capabilities of our method to novel views and objects are limited. For example, if the relevant features differ significantly

between the test views and objects, then our method will not generalize to them. Our method relies on holdout validation on randomly selected regions to avoid overfitting; this ignores the hatching information existing in these regions that might be valuable. Re-training the model is sometimes useful to improve results, since these regions are selected randomly. Learning from a broader corpus of examples could help with these issues, although this would require drawings where the hatching properties change consistently across different object and views. In addition, if none of the features or a combination of them cannot be mapped to a hatching property, then our method will also fail.

Finding what and how other features are relevant to artists' pen-and-ink illustrations is an open problem. Our method does not represent the dependence of style on part labels (e.g., eyes versus torsos), as previously done for painterly rendering of images [156]. Given such labels, it could be possible to generalize the algorithm to take this information into account.

The quality of our results depend on how well the hatching properties were extracted from the training drawing during the preprocessing step. This step gives only coarse estimates, and depends on various thresholds. This preprocessing cannot handle highly-stylized strokes such as wavy lines or highly-textured strokes.

Example-based stroke synthesis [33, 51, 67] may be combined with our approach to generate styles with similar stroke texture. An optimization technique [145] might be used to place streamlines appropriately in order to match a target tone. Our method focus only on hatching, and render feature curves separately. Learning the feature curves is an interesting future direction. We also believe that our learning techniques could be used for analyzing data from surveys with larger datasets than ours. Another direction for future work is hatching for animated scenes, possibly based on a data-driven model similar to [70]. Finally, we believe that aspects of our approach may be applicable to other applications in geometry processing and artistic rendering, especially for vector field design.

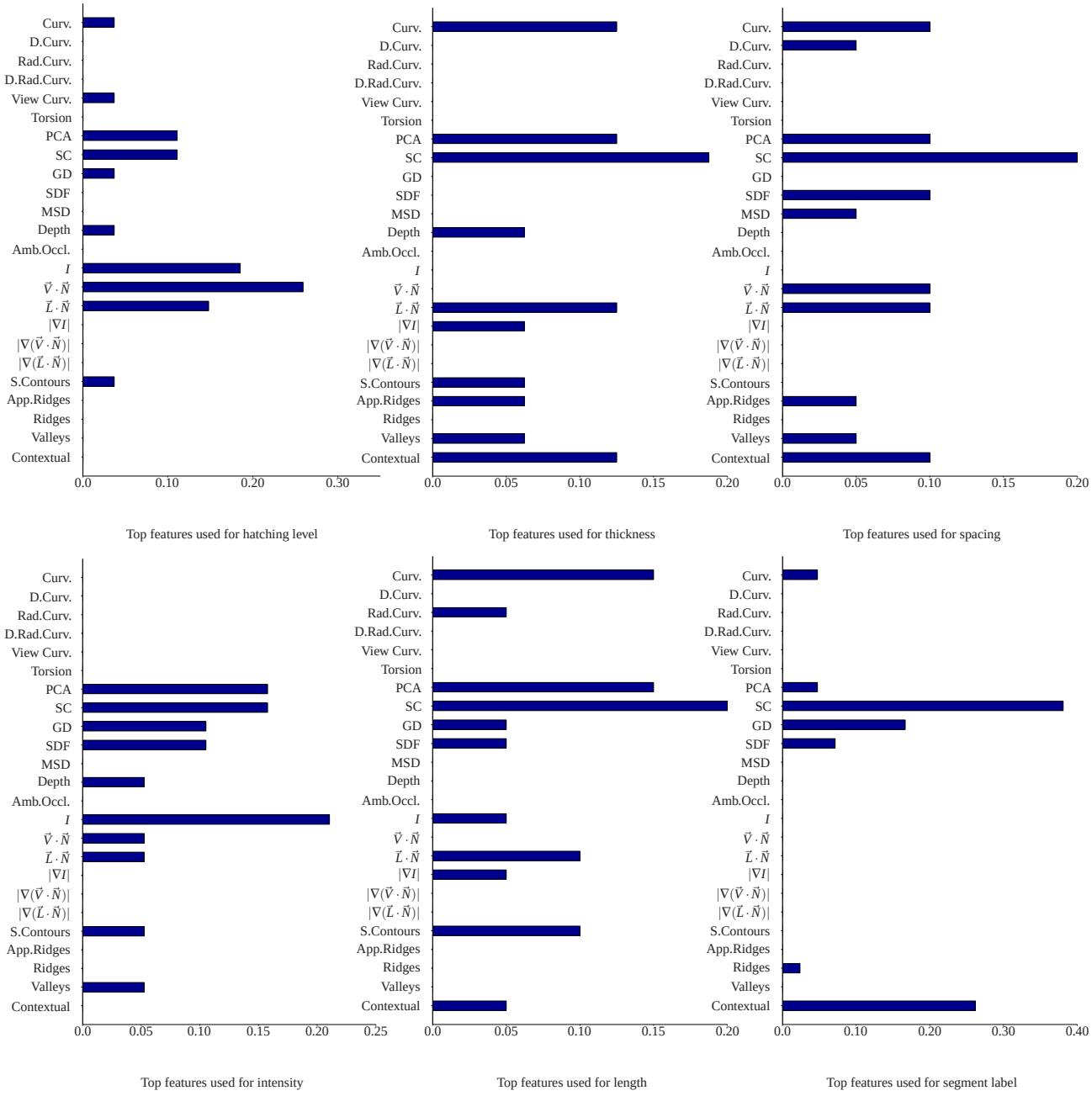


Figure 4.16: *Frequency of the first three scalar features selected by the boosting techniques used in our algorithm for learning the scalar hatching properties. The frequency is averaged over all nine training drawings. The scalar features are grouped based on their type: Curvature (Curv.), Derivatives of Curvature (D. Curv.), Radial Curvature (Rad. Curv.),Derivative of Radial Curvature (D. Rad. Curv.), Torsion, features based on PCA analysis on local shape neighborhoods, features based Shape Context histograms [10], features based on geodesic distance descriptor [53], shape diameter function features [124], distance from medial surface features [91], depth, ambient occlusion, image intensity (I),* $\vec{V}\cdot\vec{N}$, $\vec{L}\cdot\vec{N}$, *gradient magnitudes of the last three, strength of suggestive contours, strength of apparent ridges, strength of ridges and values, contextual label features.*

Chapter 5

Data-driven computation of surface attributes for animated scenes

In this chapter, I investigate the second type of geometry processing problems for which learning algorithms can be useful. These problems involve the efficient, possibly real-time computation of shape attributes for dynamic, animated scenes. When the shape attributes exhibit strong correlation to the animation parameters of the shape, learning algorithms can be used to learn a function from the shape representation to the attributes; in this case, the shape is represented by a low-dimensional state vector describing its animation.

The approach to learning such mappings begins with computing a set of training pairs $\{(\mathbf{x}_i, \mathbf{y}_i)\}$, where $\mathbf{x}_i$ are the animation parameters for a mesh and $\mathbf{y}_i$ are a set of target attributes. Then, a mapping is learned from the low-dimensional parameterization to target attributes:

$$\mathbf{y} = f(\mathbf{x}) \tag{5.1}$$

This mapping is very high-dimensional, and can also be highly nonlinear. Furthermore, it is crucial that the mapping can be evaluated fast enough to allow real-time computations of the attributes during runtime. In some cases, the locality of the mapping can be exploited, e.g., the

target attribute may only be affected by a few nearby joints.

I developed a method that combines dimensionality reduction and regression, while also taking advantage of locality when possible. Here, I describe the details for the method in the case of surface curvature attributes (Section 5.1). In [102], we discuss a similar methodology for the case of surface visibility represented by its spherical harmonics coefficients. Both surface curvature and visibility are attributes that need to be computed in real-time for many applications in computer graphics. For example, surface curvature estimation is an important component of object-space line drawing for many types of curves, such as suggestive contours [22]. Computing surface visibility is fundamental to photo-realistic rendering. Real-time evaluation of the learned mappings for these attributes also enable real-time execution of these applications for deforming shapes.

5.1 Data-driven curvature for real-time line drawing of dynamic scenes

Here, I will describe our approach for learning mappings from animation parameters to a set of *curvature attributes*—namely, curvature tensors and derivatives [1]. We apply the learned mappings for real-time line drawing. Line drawing is based on rendering a variety of curves defined on 3D surfaces, such as suggestive contours [22], ridges and valleys [59, 105, 139], apparent ridges [66], highlight lines [23] (Figure 5.1). These curves are essential components of high-quality line drawing of smooth surfaces and require surface curvature and curvature derivatives to be computed everywhere on the surface.

[1]The work presented in the following sections is also published in ACM Transactions on Graphics, Vol. 28, No. 1, 2009 [70]. Project web page: http://www.dgp.toronto.edu/~kalo/papers/MLcurvature/, ©ACM, (2009). This is the author's version of the work. It is posted here by permission of ACM for your personal use. The definitive version was published in ACM Transactions on Graphics, Vol. 28, No. 1, 2009, http://doi.acm.org/10.1145/1477926.1477937

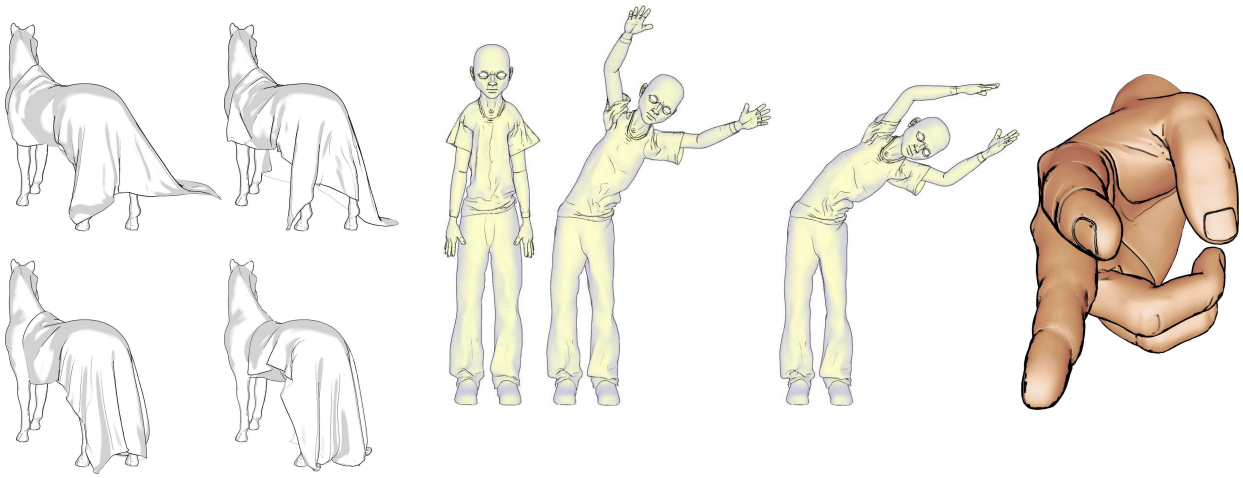


Figure 5.1: *Line drawings of deforming 3D objects, generated in real-time (24 to 80 FPS) by our system.*

Interactive line drawings of static geometry can be rendered in real-time, because curvatures can be precomputed [21]. However, for dynamic geometry, curvatures must be recomputed for each frame, since storing all the curvature and derivatives of curvature values per frame or storing key poses and then interpolating would require prohibitive amounts of storage (Section 5.8). There are several curvature estimation algorithms that rely on simple differential geometry formulas that can be evaluated on meshes very efficiently (e.g., [137, 96, 17]), however they often suffer from degenerate cases and noisy estimates, and do not compute third-order surface derivatives [115]. Other fast methods based on focal surface approximations [153] are also affected by degeneracies and do not apply in parabolic regions (unless refined by slow non-linear optimization techniques [154]). In general, in order to maintain robustness to noise, irregular tessellation, and also to fully compute third-order derivatives, more expensive computations are necessary. Typically, multiple steps of curvature smoothing or feature-preserving optimization of the curvature tensors are required [115, 72], therefore computing surface curvatures and their derivatives reliably enough is generally too slow to be used in real-time (even for moderate-sized meshes).

An alternative solution for interactive curvature-based line drawing would be to use using GPU-

based image processing operations [85]. Image-space methods are appealing in that they are generally simple and easy to implement. However, there are a number of drawbacks as well: accuracy is limited by pixel resolution (often resulting in jagged or irregular lines), stylization options are limited (e.g., curves cannot be textured), speed is limited by hardware image processing performance, and careful setting of user-defined thresholds is required.

Thus, it would be highly desirable to have an object-space algorithm for reliably computing surface curvature and its derivatives in real-time for animated surfaces that would provide improvements in speed, visual quality and stylization options.

In the following sections, I will describe the learning method we followed to achieve this. Our algorithm can produce animated 3D line drawings at real-time rates for meshes of 100K triangles in a single processor. We employ a series of learning techniques during precomputation so that only a few megabytes of storage are required per dataset, curvature synthesis is performed very efficiently and accurately during runtime and generalization capabilities are offered for novel, unseen animation sequences. With our algorithm, it is now possible to generate accurate and stylizable curvature-based line drawings of 3D animated surfaces in real-time (Figures 5.1 and 5.2).

The results of our method are nearly indistinguishable from the per-triangle tensor fitting method of Rusinkiewicz [115], with similar temporal coherence, but require an order-of-magnitude less computation during runtime. We apply our approach to three types of surfaces: skeleton-based characters, cloth simulation and blend-shape facial animation. We show the ability of our system to generalize to novel animation sequences that are not included in the training set. We demonstrate stroke stylization with real-time chaining (Figures 5.15 and 5.16). In addition, stroke thickness can be determined as a function of surface curvature.

Figure 5.2: *Real-time renderings generated with our method (principal highlights and suggestive contours for the horse and apparent ridges and valleys for the other figures). For the hand, we apply textured chained-strokes for stylization.*

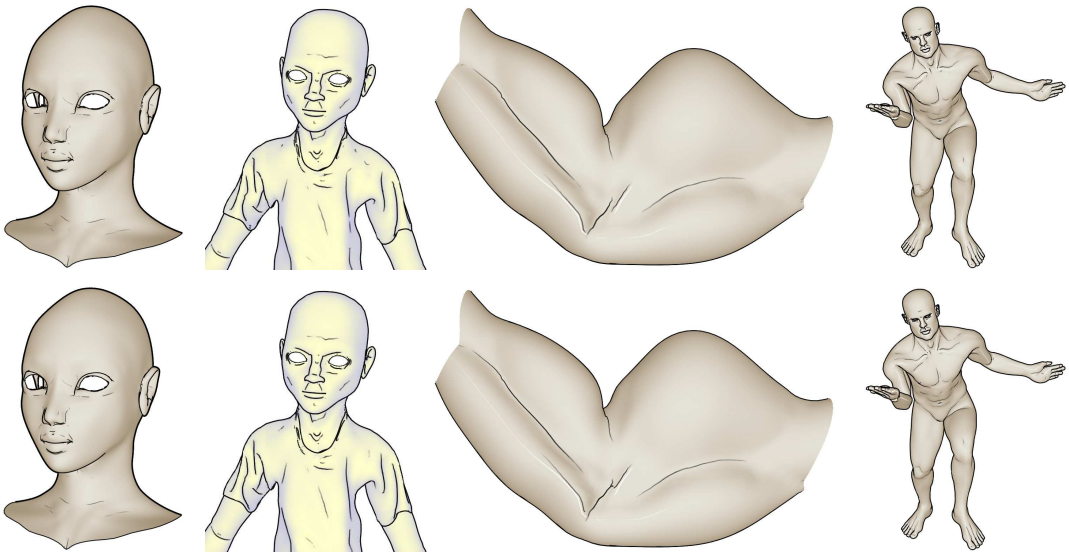


Figure 5.3: *Results generated in real-time using our method (**top**) compared to those generated with explicit curvature re-calculation (**bottom**).*

5.2 Related work

Our work is inspired by methods for precomputing deformation and radiance transfer. Example-based skinning algorithms [86, 97, 149] learn mappings from skeleton parameters to 3D shapes; our method for skeleton-based characters learns mappings to surface curvature. For cloth simulation, our method is in the same spirit with the photorealistic rendering algorithms of James and Fatahalian [61] and Nowrouzezahrai *et al.* [104, 103, 102], in which dimensionality reduction is applied to relate simulation and animation to rendering. To the best of our knowledge, this thesis presents the first data-driven method for curvature estimation.

5.3 Overview

Our approach to computing surface curvatures has two stages: the preprocessing stage, which is performed offline, and the runtime synthesis stage, which is performed in real-time.

Preprocessing. In a preprocessing stage, we begin with an animation sequence, from which we can compute a set of M training pairs $\{(\mathbf{x}_i, \mathbf{y}_i)\}$, where $\mathbf{x}_i$ are the parameters for a mesh and $\mathbf{y}_i$ are a set of curvature attributes. The curvatures for these meshes are computed with the algorithm of Rusinkiewicz [115]. Additional curvature smoothing and optimization steps are required in order to obtain high-quality results [22, 72].

Then, we learn a mapping from the low-dimensional parameterization to surface curvatures. For a skeleton-based character, linear regression with a low-order polynomial model is used. A quadratic model is used for surface curvatures and principal directions and a cubic model is used for the derivatives of curvature. For cloth simulation, first, a low-dimensional representation of geometry is discovered and then linear regression is used. For facial animation, neural network regression maps from the blending parameters to the curvature space.

Run-time rendering. During an interactive session, the parameters **x** are determined for each frame. The curvatures **y** are computed by $\mathbf{y} = f(\mathbf{x})$. Then, various rendering options are supported. The mesh can be rendered with contours, suggestive contours, and any other lines that requires curvature. Real-time chaining can be performed to provide more stylization options, such as texturing strokes. Stroke thickness can also be determined as a function of surface curvature.

Our method for skeleton-based surfaces is described in Section 5.4. Simulated cloth surfaces are described in Section 5.5, followed by our method for blend-shape facial animation in Section 5.6.

5.3.1 Curvature attributes

The surface curvature data **y** can be represented in different ways. For our application, there are three primary considerations in choosing a representation. First, we want a spatially smooth representation that exploits the local correlations in the curvature field in order to reduce the size of the model through dimensionality reduction (Section 5.3.2). Second, we want the representation to smoothly vary as a function of animation parameters in order to achieve accurate regression and better temporal coherence. Lastly, we want a representation that stores as few values as possible for each vertex, in order to reduce storage costs. In order to fulfill these goals, we represent the curvature attributes as follows:

1. **The principal curvatures** k_1 and k_2. We use the standard definition where $k_1 > k_2$, rather than $|k_1| > |k_2|$ [116], since the latter definition introduces temporal discontinuities in the curvature field (swapping of principal directions), which adversely affects the learning procedure.

2. **The principal direction of maximum curvature** $\vec{e}_1$. This direction is represented by

using its first two components in a local coordinate system. This particular representation of the curvature attributes is chosen for invariance to rigid transformations of parts of the surface. While a 1D angular representation (i.e., the angle in the tangent plane of each vertex) would be more compact, this parametrization would have singularities at 2π.

The local coordinate systems are determined by first segmenting the surface into rigid segments and then performing PCA on the vertices of each segment. For skeleton-based characters, the segmentation is computed by applying mean-shift clustering [19] to the skinning weights. For cloth simulation, rigid components are found using the method of James and Twigg [62]. At run-time, the third component of $\vec{e}_1$ and the other principal direction $\vec{e}_2$ can be computed from this representation and the per-vertex normals. Per-vertex normals are computed in a standard manner per-frame, i.e., as the weighted average of incident face normals. In order to improve spatial smoothness, we also adjust the principal directions to match the segment's coordinate system orientation. The local rigid coordinate frame is aligned to a reference mesh edge and normal vector (which are selected to match closely the PCA directions) for each segment. Then, in subsequent frames, we orient the principal directions to match their previous orientation in order to achieve temporal coherence.

3. **The derivatives of curvatures.** These derivatives form a $2 \times 2 \times 2$ tensor, which, due to symmetry, can be represented by four values.

We will learn a separate mapping ($\mathbf{y} = f(\mathbf{x})$) from the animation parameters $\mathbf{x}$ to each of the eight curvature attributes listed above.

5.3.2 Dimensionality reduction

The attribute vector **y** for a mesh is very high-dimensional. However, as there are significant spatial correlations in the curvatures, dimensionality reduction can be employed to significantly compress these vectors, based on the discussion of Section 2.5. In this case, dimensionality reduction also helps to denoise unstable attributes (such as principal directions near umbilical points), since noisy data are not captured by the first few principal components, since they correspond to larger variance in the data. Thus, noisy data are not represented in the low-dimensional subspace.

We use the ICA technique [9, 20, 14] for dimensionality reduction (Section 2.5.2). More specifically, we use the FastICA variant [56], which maximizes non-gaussianity by using the approximation of negentropy. An alternative option would be to use PCA, but as we discussed in Section 2.5.2, while PCA has the property that it is least-squares optimal for compressing the training data, this does not guarantee that it will generalize to new shapes not included in the training data. In fact, we find that ICA does generalize better because it prefers sparse bases, yielding localized basis functions corresponding to structure in the data, such as folds, wrinkles, and other similar structures (Figure 2.6). Similarly, it has been often noted in the literature that ICA applied to image data yields localized features, e.g., [7, 9]. In contrast, the PCA bases are global: the first components contain a mixture of many distinct folds and wrinkles that are less likely to co-occur for novel poses.

5.4 Skeleton-based deformations

Our method for skeleton-based curvature prediction exploits the special structure of skinned geometry. Specifically, we note that the skeleton's joint angle values provide a natural parameterization, and so we will use them as the inputs **x** to the regression. Furthermore, the curvature

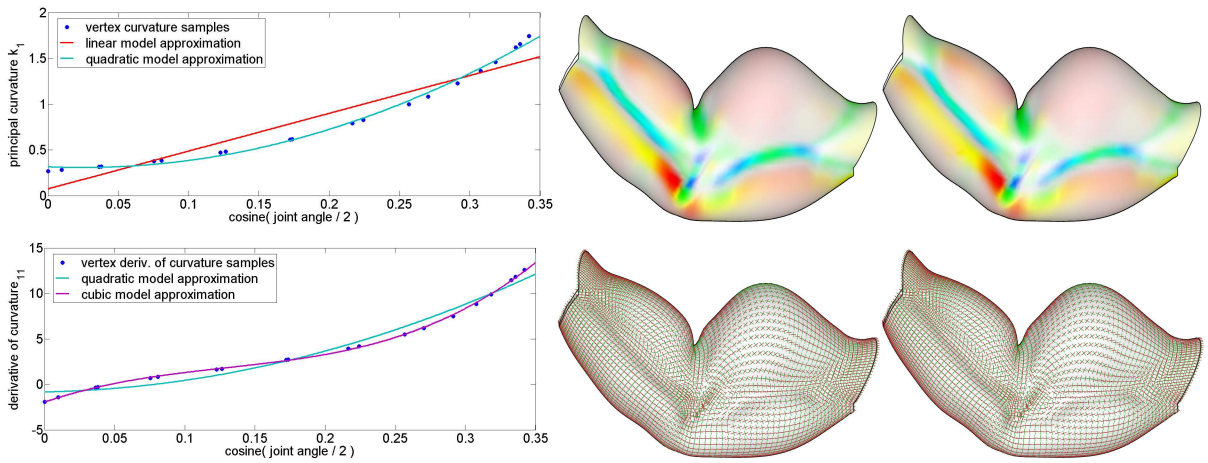


Figure 5.4: Left: *Typical plots of curvature and derivatives of curvature at a vertex as a function of one joint angle, for the muscle mesh. The vertex shown is the one with highest variance in principal curvature k_1 during training. The quadratic model is more suitable than a simple linear model, while a cubic model is more appropriate for the derivatives of curvature components.* Top right: *Comparison of principal curvatures produced by the method of Rusinkiewicz [2004], as compared to those produced by our method.* Bottom right: *Comparison of principal directions. The most significant differences in principal direction estimates occur at umbilical points where the directions are unstable. We also report the running times for Rusinkiewicz's and our method.*

attributes we wish to predict depend only locally on joint values. For example, the angle of an elbow affects the skin only within its nearby support area, and not the rest of the body. This is similar to the locality of weights used in example-based skinning algorithms (e.g., [97, 149]).

Our method for skeleton-based characters works as follows. First, we gather the training data, and represent it as described in the next section. We predict curvature as a function of joint angles, using a polynomial regression model described in Section 5.4.2. For each vertex, we determine which joints have a significant influence on the curvature at the vertex by applying a statistical test (Section 5.4.3). To simplify the regression, we perform dimensionality reduction

on the curvature attributes of the influenced vertices per joint (Section 5.4.4). Finally, we apply regression to build the mapping from animation parameters to curvature (Section 5.4.5).

5.4.1 Training

We begin with a set of training poses. These poses may, for example, correspond to a typical animated sequence for this character. Following Wang et al.'s scale/shear regression [149], we represent a pose $\mathbf{x}$ as a vector of *bones*, where each bone is parameterized by the associated joint and its parent joint angles. Each joint is represented as three Euler rotation angles with respect to the corresponding axis of rotation in its local coordinate frame. Therefore, each bone has 6 degrees of freedom. We represent each element of $\mathbf{x}$ as $\cos(\theta/2)$, where θ is a joint angle. This representation is motivated by the fact that the discrete mean curvature at an edge depends on the cosine of half of the dihedral angle [109], and thus these values were found to be better for predicting curvature.

For each training pose i, we compute the corresponding surface attributes $\mathbf{y}_i$. Note that some vertices can be treated as rigid, such as vertices with neighborhoods influenced only by one bone. We detect vertices with curvature variation less than 0.5% of the maximum curvature variation in the data. These vertices are treated as having constant curvature and removed from the learning process. In the Mr. Fit model (Figure 5.7), about 25% of the vertices are treated as rigid.

5.4.2 Regression model

In order to select an appropriate regression model, we first consider the case of a character with only a single joint. As shown in Figure 5.4, we find that the curvature at the vertices around a joint can be approximated very well by a quadratic function of the joint angle, while a cubic

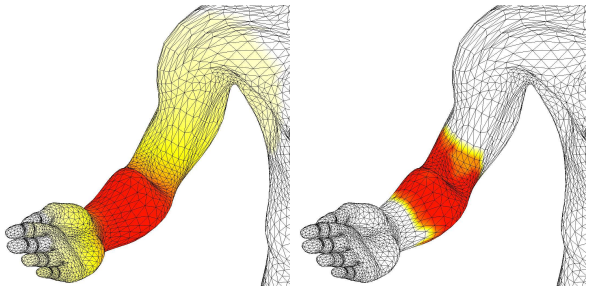


Figure 5.5: Left: *Typical smooth skinning weights for an elbow joint.* Right: *Curvature attribute weights ($w_{j,v}$) from the elbow joint as determined by our method. Note that fewer curvatures require input from this joint; additionally, the distribution of weights is noticeably different from the skinning weights. We observed that blending the curvatures with skinning weights between different joints resulted in significant errors and lower runtime speed. With our weighting scheme, the weights of the joints on the curvature of vertices were distributed more appropriately.*

is sufficient for derivative of curvature. We found that higher-order models (such as B-splines) are more powerful than necessary for articulated data, thus requiring more storage and running time for the same-quality results while also exhibiting poorer generalization.

Hence, we will perform regression with the model

$$\mathbf{y} = \mathbf{V}\phi(\mathbf{x}) \tag{5.2}$$

where $\mathbf{V}$ is a matrix of regression weights. For surface curvatures, we use quadratic features:

$$\phi(\mathbf{x}) = [1, x_1, ..., x_K, x_1^2, ..., x_K^2]^T \tag{5.3}$$

while, for derivatives of curvature, we use cubic features:

$$\phi(\mathbf{x}) = [1, x_1, ..., x_K, x_1^2, ..., x_K^2, x_1^3, ..., x_K^3]^T \tag{5.4}$$

where K is the total number of joint angles. We omit the bilinear terms $x_i x_j$ for $i \neq j$ and other higher-order terms, as we have found that these lead to worse generalization, due to overfitting.

5.4.3 Determining which joints influence curvature at each vertex

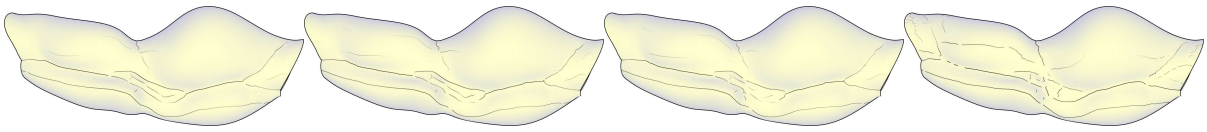
In general, the curvature attributes at a vertex can be affected by more than one joint, namely, all joints with nonzero skinning weight at that vertex. Joints with nonzero weights at neighboring vertices can also affect curvature. However, the curvature can often be predicted using only a subset of these joints. In order to reduce the model size, we need to determine a subset of joints to be used for regression at each vertex. That is, we find the joints which have a significant effect on the curvature at vertex v. This is a feature selection problem that can be treated with the boosting for regression technique that was discussed in Section 2.4.3. However, we used a much simpler technique, because the number of joints to process is very small (compared to the high-dimensional spaces we had in the case of learning stroke properties for line illustrations in the previous chapter), and the regression model to be learned is not that complex. The technique we used is a statistical test that is applied at each vertex. This statistical test is performed based on prediction of mean curvature $\kappa = (k_1 + k_2)/2$. The joints selected to influence each vertex based on mean curvature will be used for all other curvature attributes.

Specifically, for each vertex, we fit the mean curvature values for each training pose i by least-squares regression, minimizing:

$$E_{\text{FULL}} = \sum_i ||\kappa_i - \mathbf{a}^T \phi(\mathbf{x}_i^{[v]})||^2 \tag{5.5}$$

where $\mathbf{x}_i^{[v]}$ are the K elements (joint angles) of $\mathbf{x}_i$ that influence vertex v (as determined by the skinning weights), ϕ is a quadratic feature vector (Equation 5.3), and $\mathbf{a}$ are the regression weights. Then this regression is repeated using only individual joints as inputs (as in Section 5.4.2). Regression on joint j (i.e., using the six elements of its angles and its parent joint angles as the inputs $\mathbf{x}_i^{[v]}$) gives another residual E_j. An F-test [150] is then applied to determine whether to keep the joint's influence: this test simply determines whether including a joint makes a significant improvement to the residual. Specifically, the F statistic is:

$$F = \frac{(E_j - E_{\text{FULL}})/(9J - 6)}{E_{\text{FULL}}/(N - 9J)} \tag{5.6}$$



Ground Truth	95% of variance	90% of variance	zero-order
	0.8MB, 1.4ms	0.5MB, 1.1ms	0.2MB, 0.01ms

Figure 5.6: *Ridges and Valleys for muscle dataset with respect to decreasing variance captured by the basis. Ridges and Valleys based on ground truth curvature data is on the left. A zero-order prediction based only on the mean of the curvature data is also depicted on the right for comparison. A reasonable choice that balances the trade-off between speed and accuracy is selecting the number of components based on 95% of the variance. The size of the model and running times per frame are also shown.*

where J is the number of joints for this vertex with non-zero skinning weights and N is the number of training poses. The corresponding joint will then be kept for the regression for v if F is greater than the critical value for the F distribution for $p > 0.05$.

This test is repeated for all joints with nonzero skinning weights at this vertex; those that pass the test are deemed as influencing this vertex. If all the joints fail the F-test, then the one with smallest residual E_j is kept. In practice, we observe that two joints are sufficient for most vertices in most cases. Although it is possible that a joint of a bone will affect the curvature at a vertex for which it has zero skinning weight, smoothness of the skinning weights implies that the effect of the bone is negligible. In our experiments, this statistical test typically halves the size of the learned model and speeds up run-time curvature prediction by 150-200%. We also experimented with boosting for regression to select the subset of joints that influence curvature at each vertex; it had similar performance in both results and execution time.

5.4.4 Dimensionality reduction

Due to the large number of vertices, directly learning the mapping to all the curvature attributes per vertex would require estimating and storing an impractical number of weights. Instead, we exploit the spatial coherence of the curvature attributes and perform regression on a reduced-dimensional model, as described below. The following process is performed eight times, once for each curvature attribute.

For each joint j, we define a vector $\mathbf{y}^{(j)}$ consisting of the values of the curvature attribute to be predicted from this joint. One such vector $\mathbf{y}_i^{(j)}$ is computed for each pose i in the training set and contains the attribute to be predicted (e.g., k_1). Because this vector $\mathbf{y}^{(j)}$ is high-dimensional (its dimensionality is equal to the number of vertices influenced by the joint), we apply ICA to the training data to obtain a reduced representation:

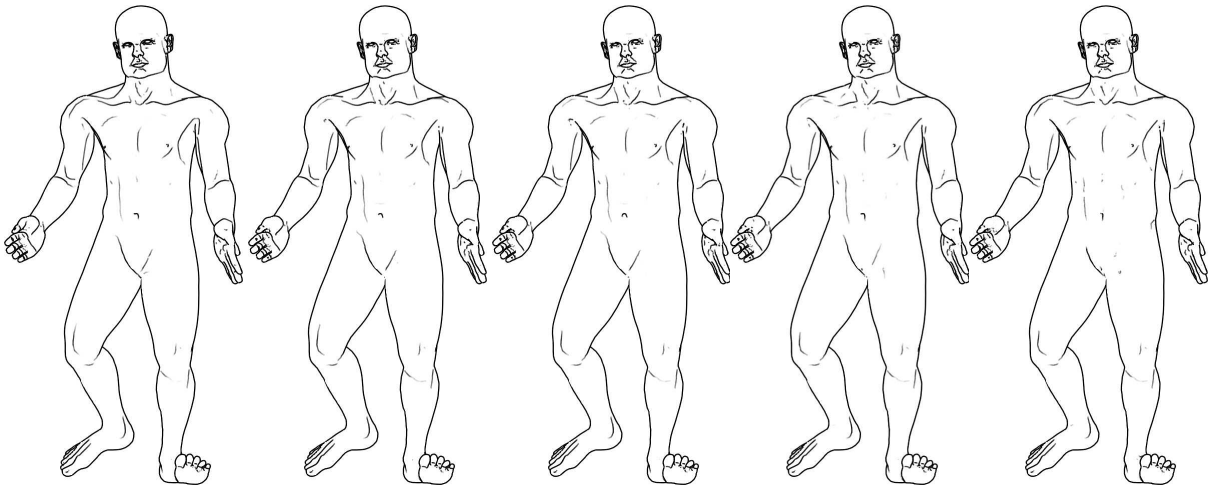
$$\mathbf{y}^{(j)} = \mathbf{W}_j \mathbf{z} + \bar{\mathbf{y}} \tag{5.7}$$

All terms on the right-hand side are determined by the FastICA algorithm [56]. We keep the first D independent bases, where D is set to the number of the eigenvalues required to capture 95% of the variance of W. This threshold is selected empirically to balance the trade-off between speed and accuracy (Figure 5.6).

5.4.5 Regression

We use least-squares regression to map from the animation parameters $\mathbf{x}$ to their corresponding values $\mathbf{z}$ in the low-dimensional space of curvature attributes. Specifically, we solve for the weights $\mathbf{V}$ that minimize

$$\sum_{i=1}^{M} ||\mathbf{z}_i - \mathbf{V}_j \phi(\mathbf{x}_i^{(j)})||^2 \tag{5.8}$$



Ground Truth 100 training frames 50 training frames 25 training frames 20 training frames

Figure 5.7: *Suggestive contours for Mr. Fit dataset with respect to the number of training examples. Suggestive contours based on ground truth curvature data are on the left. Our system can accurately synthesize surface curvatures using a few training examples.*

where $\mathbf{x}^{(j)}$ are the six elements of $\mathbf{x}$ that depend on joint j. Each joint now provides a separate predictor of the curvature at a particular vertex v, i.e.,

$$\tilde{y}_{v,j}(\mathbf{x}) = \mathbf{W}_j \mathbf{V}_j \phi(\mathbf{x}^{(j)}) + \bar{\mathbf{y}}_v \tag{5.9}$$

where the subscript v indexes rows specific to that vertex. The predictor $\tilde{y}_{v,j}(\mathbf{x})$ can be viewed as an estimate of y_v. (Note that each joint j will have its own $\mathbf{W}$ and $\mathbf{V}$ matrices).

We create the final predictor of y_v by linearly combining these predictors in a manner similar to boosting [11].

$$y_v^* = f_v(\mathbf{x}) = \sum_{j=1}^{\hat{J}} w_{j,v} \tilde{y}_{v,j}(\mathbf{x}) \tag{5.10}$$

where $\hat{J}$ is the number of joints with nonzero influence on this vertex (as determined in Section 5.4.3). One option for determining the weights w is by least-squares fitting. However, we have obtained better results by weighting the predictors according to their fit to the training data. Specifically, let $r_j = \sum_i (y_{i,v} - \tilde{y}_{v,j}(\mathbf{x}))^2$ be the residual of the j-th predictor. Then, we

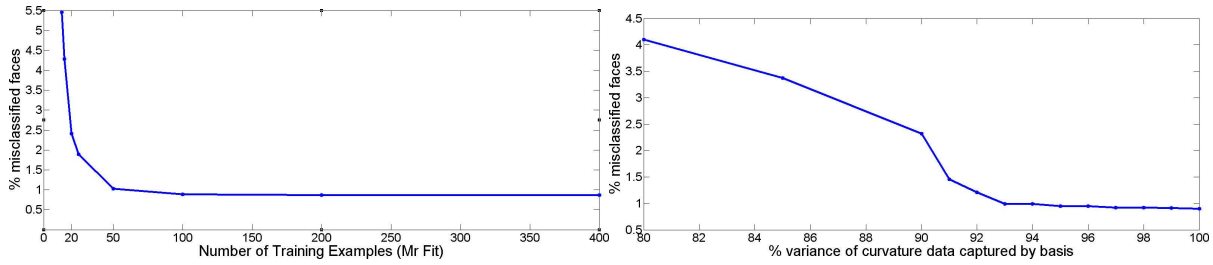


Figure 5.8: Left: *Plot of % misclassified faces for the suggestive contour drawings for the Mr. Fit test sequences versus number of training examples (the number of ICA components is chosen to correspond with the 95% of the variance of the curvature data). More precisely, we compute the percentage of mesh faces that are not identified as having or not having a suggestive contour. Note that the test error is smoothly decreasing and is relatively small even for a small number of training examples. The minimal amount of training data is 19 training poses for character animation sequences since there are at most 6 DOFs and the feature vector is cubic for derivatives of curvature.* Right: *Plot of % misclassified faces for ridge and valley drawings for the muscle dataset versus the variance of the curvature data captured by our basis. The zero-order prediction had an error of 6.25%.*

set the weight for predictor j proportional to the sum of the residuals for all other predictors, normalized to sum to 1:

$$w_{j,v} = \frac{\sum_{k \neq j} r_k}{(\hat{J}-1)\sum_{k=1}^{\hat{J}} r_k} \tag{5.11}$$

where $\hat{J}$ is the number of predictors. This can be thought of as similar to the linear blend skinning process, but averaging target curvatures rather than target poses. We visualize our resulting weights in Figure 5.5.

5.4.6 Run-time evaluation

During run-time, given a new pose **x**, the curvature attributes for each vertex are computed by applying Equation 5.10. Curvature prediction is visualized in Figure 5.4. We also provide error

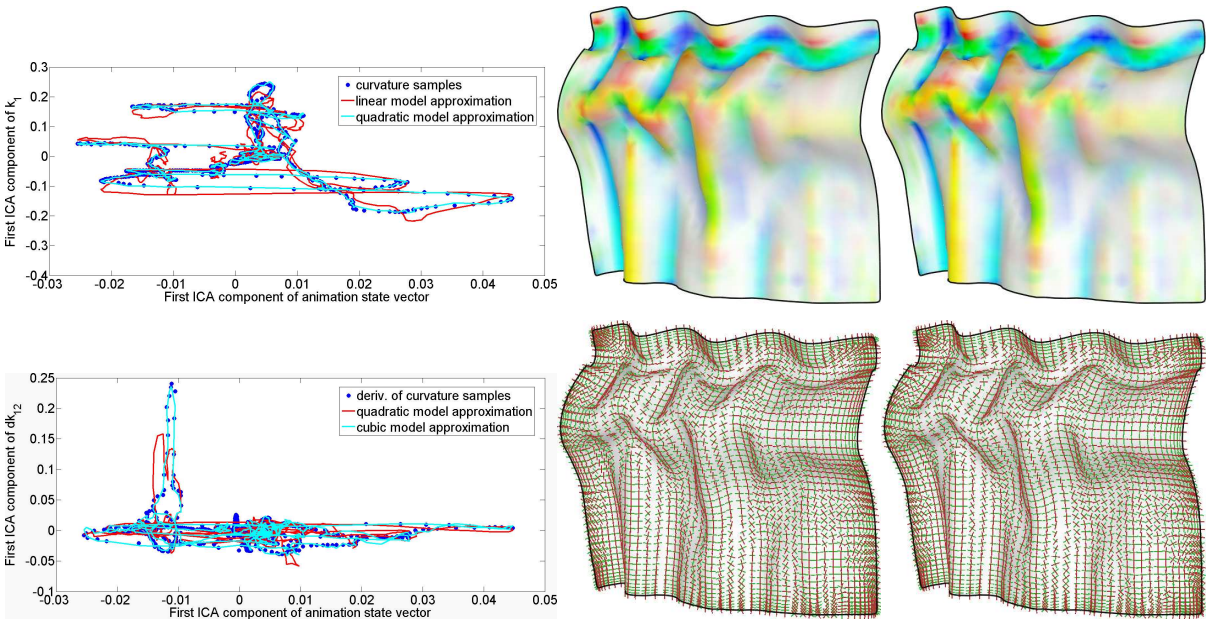


Figure 5.9: Left: *Typical plots of the first ICA component of curvature and derivatives of curvature data for a cloth simulation with respect to the first ICA component of the animation state vector. A quadratic and a cubic model are more appropriate for fitting curvatures and derivatives of curvatures respectively.* Top right: *Comparison of principal curvatures produced by the method of Rusinkiewicz [2004] and smoothed, as compared to those produced.* Bottom right: *Comparison of principal directions. We also report running times for both methods.*

analysis with respect to the number of ICA components and number of training examples used in Figure 5.7. Example skeleton-based renderings are shown in Figures 3.1, 5.2, 5.3, 5.6, 5.7, 5.16 and in the accompanying video. We also show examples of generalization of our method to novel animation sequences in the accompanying video.

5.5 Cloth simulation

To learn curvatures for cloth simulation, we begin with an animated cloth sequence $(\mathbf{s}_1, .., \mathbf{s}_M)$ as training data. Our goal is to be able to compute curvatures $\mathbf{y}$ for a new cloth shape $\mathbf{s}$. Because no low-dimensional state vector is provided for the cloth, we apply dimensionality reduction to the animation state to obtain one. We will learn a mapping from this low-dimensional space derived from the current cloth shape $\mathbf{s}$ (Section 5.5.1) to the low-dimensional space of surface curvatures (Section 5.5.2).

5.5.1 Dimensionality reduction for cloth state

We apply ICA to the 3D cloth shapes $\{\mathbf{s}_i\}$ to obtain animation parameters $\{\mathbf{x}_i\}$ such that $\mathbf{s} = \mathbf{A}\mathbf{x} + \bar{\mathbf{s}}$ [11, 61]. For this step, we represent the cloth state $\mathbf{s}$ in terms of dihedral angles. For example, we typically find that 50 basis vectors are sufficient to represent 95% of the variation for the horse cloth with 10K vertices (and thus 20K dihedral angles) providing a good trade-off between speed and prediction accuracy.

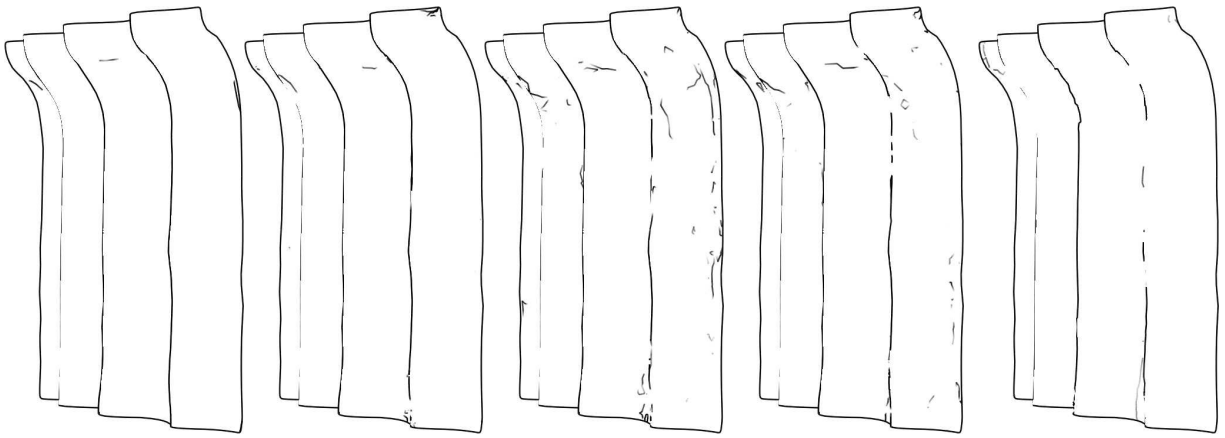
In addition, ICA is applied to curvature data to obtain a reduced representation as well:

$$\mathbf{y} = \mathbf{W}\mathbf{z} + \bar{\mathbf{y}} \tag{5.12}$$

5.5.2 Regression

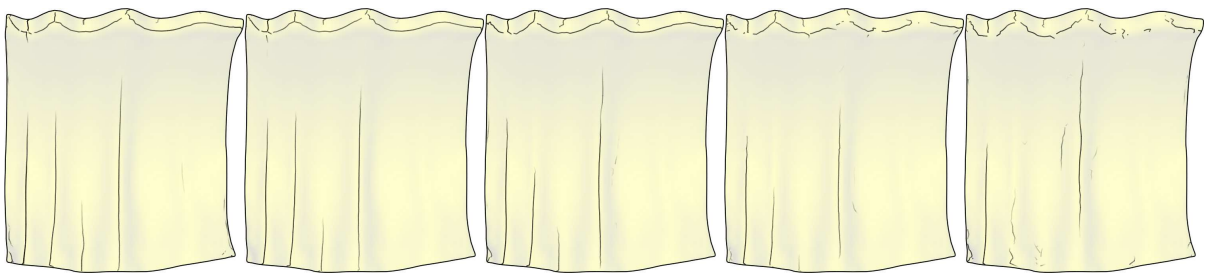
As for articulated characters, we use least-squares regression with quadratic features to map from the low-dimensional animation state $\mathbf{x}$ to the corresponding low-dimensional surface curvatures $\mathbf{z}$. More specifically, we estimate weights $\mathbf{V}$ to minimize:

$$\sum_{i=1}^{M} ||\mathbf{z}_i - \mathbf{V}\phi(\mathbf{x}_i)||^2 \tag{5.13}$$



Ground Truth | ICA 95% 4.0MB,1.5ms | PCA 95% 4.0MB,1.5ms | PCA 99% 7.1MB,2.9ms | zero-order prediction 0.1MB,0.01ms

Figure 5.10: *Suggestive contours for a novel frame of cloth with respect to the basis used corresponding to the given variance.* From left to right: *We show results for ground truth, ICA with number of base vectors corresponding to 95% of the variance of the curvature data, PCA capturing 95% of the variance and zero-order prediction. The sparsity and locality of ICA, as depicted in Figure 2.6, offers better line drawing results. Even if the number of basis is increased for PCA (99% correspond to three times more coefficients), the result does not improve much.*



Ground Truth | 500 training frames | 250 training frames | 175 training frames | 100 training frames

Figure 5.11: *Apparent ridges for a novel frame of cloth with respect to the number of training examples.*

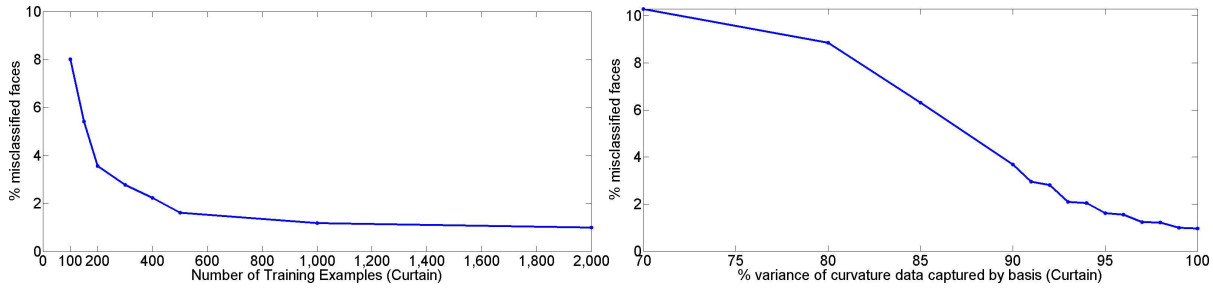


Figure 5.12: Left: *Plot of % misclassified faces for apparent ridges drawing for the curtain test sequence versus number of training examples (the number of ICA components is chosen to correspond the 95% of the variance of the curvature data). The minimal amount of training data is 97 training poses for character animation sequences since the dimensionality of the animation state vector is 32 and the feature vector is cubic for derivatives of curvature (the minimal amount of training data depends on the dimensionality of the reduced animation state vector deduced in the first step. Typically, for keeping 95% of the animated geometry, this varies from 30 to 100 in our examples).* Right: *Plot of % misclassified faces for apparent ridges drawing for the same dataset versus variance of curvature data captured by the basis for curvature. The zero-order prediction had error 20.58%.*

5.5.3 Run-time evaluation

Given a new cloth shape **s**, generating curvatures requires the following steps. First, the dihedral angles are projected to the ICA subspace to obtain the low-dimensional state. Then, the new curvatures $\mathbf{y}^*$ are predicted for the vertices of the cloth as:

$$\mathbf{y}^* = f(\mathbf{x}) = \mathbf{W}\mathbf{V}\phi(\mathbf{x}) + \overline{\mathbf{y}} \tag{5.14}$$

Example cloth renderings using our method are shown in Figures 3.1, 5.2, 5.3 and in the accompanying video. In Figure 5.10 and 5.11, we also provide error analysis as a function of the number of independent bases and the number of training examples used respectively. Given training data covering a range of motions, our model can still predict the curvature when the pa-

rameters of the dynamics (e.g., an air field or a turbulence field) controlling the cloth animation change. We show the generalization of our method in the accompanying video.

5.6 Blend-shape facial animation

In the case of blend-shape facial animation, we assume we are given M low-dimensional weight vectors $\mathbf{x}$, each of which can be used to generate a 3D face shape $\mathbf{s}$ by blending. For each training pose, we compute the surface curvature attributes $\mathbf{y}$. Unlike with skeleton-based characters and cloth, in the case of facial animation, we did not find a simple linear relationship between the blending parameters and the curvature attributes (Figure 5.13). We employ Artificial Neural Network (ANN) regression to fit this nonlinear map.

5.6.1 Neural Network Regression

As before, the learning process starts by reducing the curvature data with ICA, $\mathbf{y} = \mathbf{W}\mathbf{z} + \bar{\mathbf{y}}$, once for each of the eight curvature attributes. We then perform ANN regression [11] to learn a nonlinear mapping from the dimensionality-reduced shape $\mathbf{x}$ to the dimensionality-reduced curvature $\mathbf{z}$; one such regression is performed for each of the ICA coefficients of all the 8 curvature attributes. The ANN for each attribute has the form:

$$g(\mathbf{x}) = \sum_{\ell=1}^{L} \mathbf{w}_\ell \tanh\left(\mathbf{b}_\ell^T \mathbf{x} + b_0\right) + w_0 \tag{5.15}$$

where L is the number of neurons, $\mathbf{w}_\ell$ and $\mathbf{b}_\ell$ are L pairs of weight vectors, and w_0 and b_0 are bias terms. The weights are obtained by optimizing the following regularized least-squares objective:

$$E(w,b) = \sum_i ||\mathbf{z}_i - g(\mathbf{x}_i)||^2 + \lambda \sum_{\ell=1}^{L} \left(||\mathbf{w}_\ell||^2 + ||\mathbf{b}_\ell||^2\right) \tag{5.16}$$

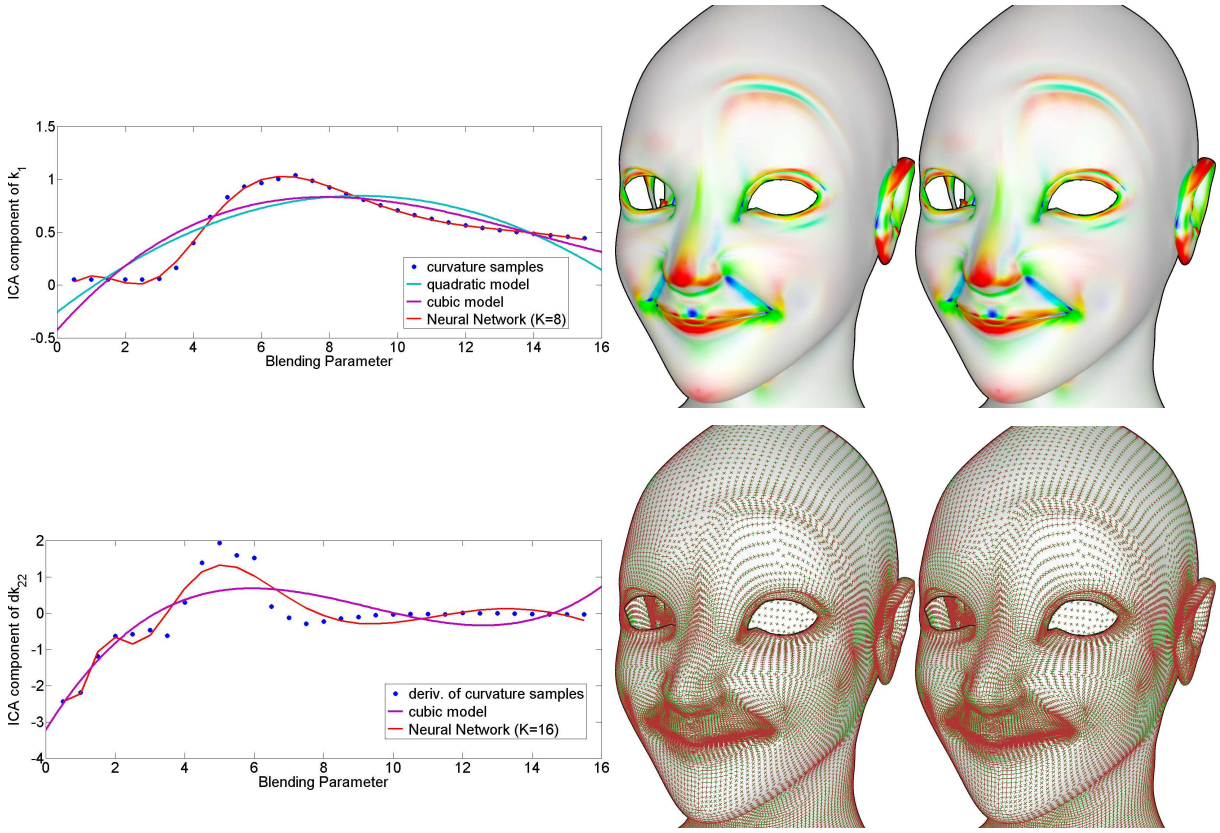


Figure 5.13: Left: *Typical plots of the first ICA component of curvature and derivatives of curvature for a face animation with respect to one of the blending parameters. In this case, a quadratic or a cubic model cannot approximate the data well. On the other hand, non-linear regression with ANNs is more appropriate in this case. The number of neurons is selected with cross-validation.* Middle: *Comparison of principal curvatures produced by the method of Rusinkiewicz [2004] and smoothed, as compared to those produced by our ANN.* Right: *Comparison of principal directions.*

where λ is a smoothing parameter and L is the number of neurons. Optimization is performed by 5000 iterations of the BFGS algorithm with cubic line search [100]. The weights $\mathbf{w}$ and $\mathbf{b}$ are initialized by sampling from a uniform distribution over $-1/K$ to $1/K$ for the elements of $\mathbf{w}_\ell$ (where K is the number of blending parameters) and over $-1/L$ to $1/L$ for $\mathbf{b}_\ell$. The

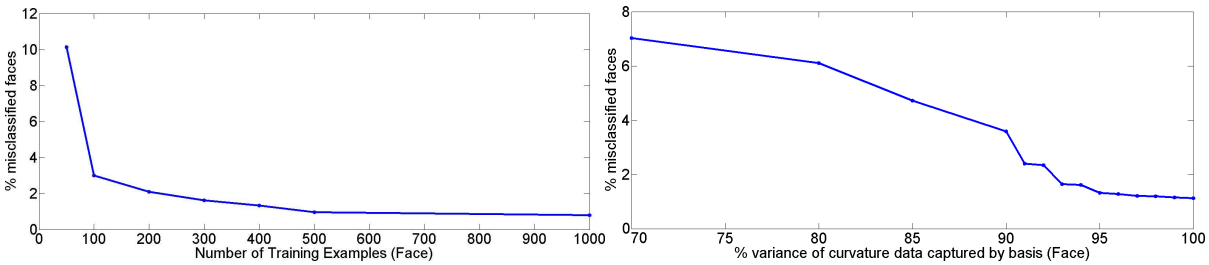


Figure 5.14: Left: *Plot of % misclassified faces for suggestive contours for the face test sequence versus the number of training examples (the number of ICA components is chosen to correspond the 95% of the variance of the curvature data).* Right: *Plot of % misclassified faces for suggestive contour drawings for the same dataset versus the variance of curvature data captured by the basis. The zero-order prediction had an error of 14.85%.*

smoothing parameter λ and the number of neurons L is chosen by cross-validation [11] in a preprocessing step.

5.6.2 Run-time evaluation

Given a new face with blending parameters $\mathbf{x}$, we compute the surface curvatures as follows:

$$\mathbf{y}^* = f(\mathbf{x}) = \mathbf{W}g(\mathbf{x}) + \bar{\mathbf{y}} \tag{5.17}$$

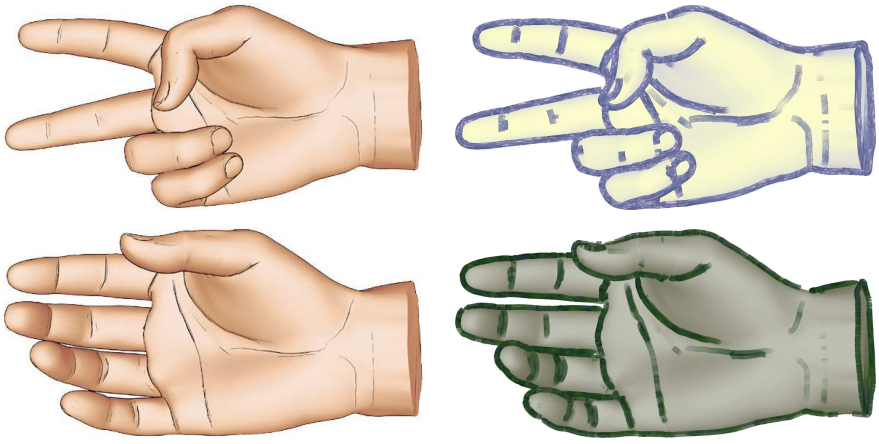
We show our curvature synthesis results in Figure 5.3 and in the accompanying video.

5.7 Stylization

Our default rendering style entails detecting surface curves (such as contours and suggestive contours) defined as zero-sets [22, 52]. Each mesh face yields a line segment, which is rendered in OpenGL. The curvatures generated by our method can also be used for stroke stylization: following Goodwin *et al.* [43], we make line thickness T a function of depth z and radial

curvature κ_r: $T = \text{clamp}(c/(z(\kappa_r + \varepsilon)))$, where c and ε are user-defined constants, and clamp$(\cdot)$ clamps the thickness to a user-defined range.

Additional stylization effects are possible by chaining curves on the surface; we modify the method of randomized contour detection of Markosian *et al.* [94] for zero-set contours and suggestive contours. For each frame, the algorithm iterates over every face in the mesh. When a face is detected that contains a contour or suggestive contour (represented as a line segment), the algorithm "walks" along the mesh, following the contour or suggestive contour until it ends or loops. This walking is performed in two directions from the starting face. This produces a chain of line segments (one for each face). Visibility for each point on the chain is computed using a reference ID image, and visible portions of chains are rendered with textured triangle strips [101].



Plain Stylization *Stroke Texturing*

Figure 5.15: *Regular curvature-modulated stylization (**left**) and textured chained-strokes (**right**), using apparent ridges and valleys.*

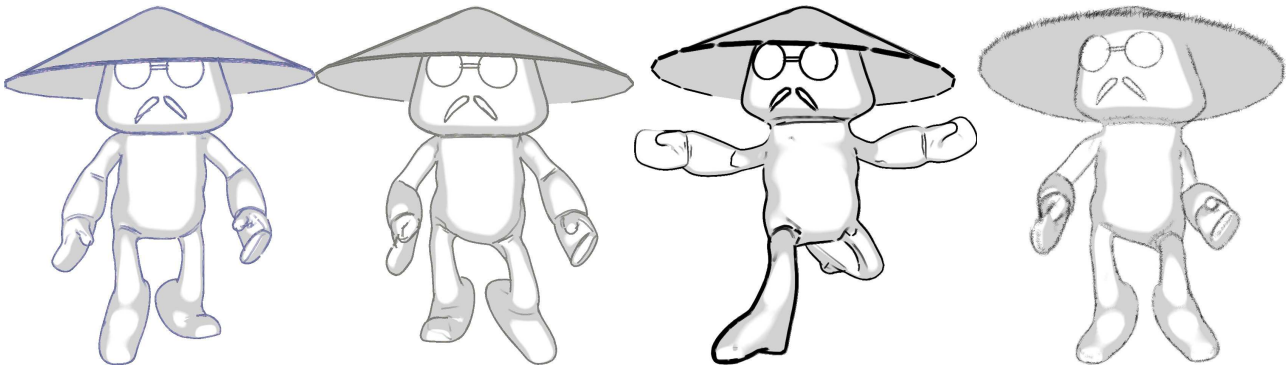


Figure 5.16: *More textured chained-strokes for Master Pai dataset, using apparent ridges and suggestive contours.*

5.8 Results

We test our method on ten datasets, including skeleton-based characters, cloth and facial animation (Figures 3.1, 5.2 and 5.3 and the accompanying video). Curvatures computed with our method have very low error (Figure 5.3, 5.8, 5.12, 5.14). Visual differences between our curvatures and ground truth are negligible (Figures 5.4, 5.9 and 5.13); differences in final line drawings are also negligible. As ground truth, we used Rusinkiewicz's method plus curvature smoothing when necessary [115] and Kalogerakis *et al.*'s method [72].

As a baseline comparison, we compare with the performance of Rusinkiewicz's method that is efficient and can fully compute both curvatures and derivatives-of-curvature for line drawings. Our curvature calculation at runtime is about 10 times faster than this method. However, this comparison is somewhat misleading: in order to generate smooth and more temporally coherent line drawings for many datasets, a few rounds of curvature and derivatives of curvature smoothing are required based on vector field diffusion [25] (also implemented in the trimesh2 library [116]) or robust statistical estimates [72]. These operations add significantly to run-time computation. Simple mesh smoothing can be done in advance but eliminates surface detail and alters the mesh.

Thus, our method is approximately 10 times faster than Rusinkiewicz's method (e.g., for smooth and regularly sampled meshes), but in most cases, it is about 20-50 times faster than performing all the necessary smoothing or optimization steps for high-quality smooth and temporally coherent line drawings. More specifically, in our experiments, we smoothed the deriva-

Dataset name	Number of Vertices	Rusinkiwicz's method (ms)	plus smoothing /optimization	Our Method (ms)	Model size (MB)
Mr. Fit	20536	81	240	7.9	10.72
Master Pai	11850	29	87	3.2	5.21
Muscle	5256	20	105	1.4	0.8
Hand	9284	25	227	2.6	4.05
Angela	25462	119	930	14	26.07
Curtain	2401	16	91	1.7	4.2
Flag	3285	19	101	2.5	5.0
Horse cloth	7921	41	529	5.0	11.9
Draping cloth	3969	26	124	2.8	4.8
Face	40767	207	750	23	32.66

Table 5.1: *Running times (in sec) for curvature estimation with our method (fifth column) compared to an explicit re-estimation with Rusinkiewicz's method (third column) and explicit re-estimation with Rusinkiewicz's method plus the necessary curvature smoothing or Kalogerakis* et al.*'s optimization technique (fourth column). Note that smooth and plausible line drawings require curvature smoothing in many cases that cannot be performed in advance. In both cases, we exclude the vertices whose curvatures do not change significantly (less than 1% of maximum variance). Timings are captured on a 2GHz Intel Core Duo Processor (no parallelization is used for any of the above methods). We also report the size of our learned model (last column).*

tives of curvature for Mr. Fit, Master Pai and face using vector field diffusion. We smoothed the curvatures for the muscle, draping, curtain, and flag datasets. We used Kalogerakis *et al.*'s method to robustly compute the curvatures and their derivatives for the Angela, hand and horse cloth datasets that seemed to be more noisy. We present running times for our method versus Rusinkiewicz's method and the total curvature re-estimation time including the necessary curvature smoothing in Table 5.1.

An alternative is to precompute curvatures for all frames and store them, for cases where generalization to new frames is not necessary. However, this would be prohibitively expensive; e.g., storing all curvatures for the Mr. Fit dataset (50K faces and 2000 frames) would require about 1 Gb of storage, whereas our method requires 10.7 Mb at run-time. Nearest-neighbor interpolation of curvature values based e.g., on a regularly-sampled grid of examples would also need orders-of-magnitude larger storage (at least 300 Mb) than our technique and with no generalization capability to novel poses. Note that such interpolation requires an exponential amount of storage with respect to the number of DOFs and would quickly result in huge model representations when many DOFs are present.

For the case of cloth, approximately 30% of the time is spent on the projection to the ICA basis for the cloth shape. Then, 65% of the time is spent on the ICA re-projection of curvatures. The remainder is used for the model regression and the re-projection of principal directions to the global coordinate system. For face and skeleton-based characters, about 90% of the time is spent on the ICA re-projection of curvatures and the remainder is used by the rest of the operations.

5.9 Summary, Limitations and Future work

We have presented a method for learning mappings from animation parameters to target attributes of deformable shapes, when these attributes are correlated to the animation of the shape. Here, we focused in the case of surface curvature, which has several applications to NPR. Curvature is a fundamental component of digital geometry processing, hence, we believe many previously off-line techniques—such as real-time hatching with smoothed directions [52], exaggerated shading [117], apparent relief [147], curvature-domain shape processing [27], and dynamic model simplification [49]—can be made real-time for dynamic geometry.

The major limitation of our approach is the need for training data and a preprocessing step, along with storage space for the learned mappings. This is typical with many real-time rendering applications that are based on offline precomputation steps [131, 61]. The most crucial goals in such approaches are efficiency during runtime and compactness of the model, which are fully achieved by our method.

Another important limitation of our approach is that the abovementioned learning method can be applied when the target attributes are spatially and temporally coherent with respect to the animation parameters. The generalization capabilities of our method to novel animation sequences also rely on the training data; i.e., the training data should be sufficient to cover a range of motions based on the analysis and examples we provided in this chapter. If the testing data cover completely different ranges of motion, then our method will not generalize. For example, if an elbow joint is not active during the training sequence, our method will not predict the curvatures around this joint for animation sequences where this joint is active; our method will not generalize from a cloth falling onto a table to a flag animation. This dependence on the training data is typical of data-driven methods [61, 149].

Chapter 6

Conclusion and Future Work

This thesis introduced machine learning algorithms for geometry processing by example. Using machine learning, I tackled two types of problems. First, I proposed algorithms for learning complex functions of shape involving several different geometric or/and appearance-based features. These functions map to target properties on which hard geometry processing tasks depend, such as shape segmentation and line illustrations. The parameters of the functions are learned automatically using boosting learning techniques and Conditional Random Fields based on the provided training examples. The estimated parameters incorporate several aspects of the user's style and preferences for the task. The learned models can apply to large databases of shapes repetitively, without needing to perform any manual re-tuning. As a result, several significant improvements are achieved over the state-of-the-art techniques; in many cases, the algorithms produce results of comparable quality of those produced of humans.

Second, I proposed algorithms for learning functions of shape from animation parameters. These function map to target shape attributes, that need to be computed efficiently for real-time geometry processing and rendering tasks. When these attributes are temporally and spatially coherent with respect to the animation parameters, these functions can be learned using

compact models. Then, they can be evaluated very efficiently during runtime. For the case of surface curvature, our proposed technique is at least an order-of-magnitude faster than state-of-the-art techniques that re-compute it at each frame geometrically.

The proposed learning methods have also their limitations. First, they require from the user to specify a consistent set of training examples. Although this represents some workload for the user, example-based techniques offer a more automated and potentially natural workflow which does not requires manual parameter tuning. On the other hand, the user has to provide all the necessary training examples; this may not be always technically feasible. In addition, some design effort is required to come up with a good learning algorithm in the first place. In addition, there are no theoretical guarantees on the generalization performance of the algorithms in a deterministic sense. The learning step also usually requires lots of computing power and time. However, once the models are learned, they can be applied to novel data very efficiently. There are also specific limitations to each application presented in this thesis, as mentioned in Sections 3.6, 4.6, 5.9.

On the other hand, the limitations are not unreasonable and can be easily tolerated by choosing the appropriate strategy for formulating the learning problem together with the appropriate learning techniques. For this reason, I presented the general steps and considerations for developing learning techniques for geometry processing tasks in Chapter 2.

There are lots of exciting future work directions for applying machine learning to geometry processing problems. First of all, this research could be extended to support learning of functions that map to many other interesting target shape properties. For example, automatically inferring an animation skeleton including its joints and their location as well as the skinning weights from examples could be an interesting future direction. In addition, inferring the texture parameters and placement constraints, given an exemplar database of textured meshes could be an interesting possibility.

Another important extension would be to also infer missing parts in a mesh or in an entire scene. This could be particularly useful for 3D modeling, where a learning algorithm could automatically suggest to the modeler a list of potential parts to augment a shape together with their potential locations. Such approach would further automate the 'Modeling by Example' framework presented in [37].

The vision behind all these techniques for learning functions of shape by example is to develop automated pipelines where the user creates, textures, edits and animates a shape, based on pre-existing training databases of shapes that serve as examples for various styles and preferences. Such pipelines would considerably decrease the user's workload and facilitate the consistent processing of large numbers of shapes.

The applications of machine learning may not only be limited to cases of learning functions to target shape properties. Perhaps one of the most challenging questions in geometry processing is how to find maps between shapes that best demonstrate their similar structure and semantics. This is very useful for shape categorization and retrieval, morphing, rigging, segmentation, modeling to name a few applications. For example, in the case of our approach to mesh segmentation and labeling, we assumed that we know the category of the test mesh. Based on this assumption, we simply applied the CRF model trained from meshes of the same category. This can become a severe limitation, especially if we want to label a large database of shapes of many different categories. Learning a low-dimensional representation of shape descriptors that would characterize semantically and structurally similar shapes could be an interesting future work direction.

Hopefully, the ideas of this thesis will help other researchers for developing machine learning techniques for many other geometry processing problems.

Appendix A

Features used For Learning Mesh Segmentation and Part Labeling

A.1 Unary Features

For each face i in a mesh, we compute a $651+35|\mathscr{C}|$-dimensional feature vector $\mathbf{x}_i$ to be used in the Unary Energy Term (Equation 3.2). Before computing any features, we translate the mesh so that its mass center lies at the origin and we normalize the scale of the mesh according to the 30th percentile of geodesic distances between all pairs of vertices. The features are as follows:

a) Curvature features: Curvatures have been used for partial matching (e.g., [38]). Around each face, we fit cubic patches of various geodesic radii ($1\%, 2\%, 5\%, 10\%$ relative to the median of all-pairs geodesic distances). The patches are fitted using the face centers and normals and every sample is weighted with its face area. Let k_1 and k_2 be the principal curvatures of a patch. We include the following features: k_1, $|k_1|$, k_2, $|k_2|$, k_1k_2, $|k_1k_2|$, $(k_1+k_2)/2$, $|(k_1+k_2)/2|$, k_1-k_2, yielding 36 features total.

b) PCA features: We compute the singular values s_1, s_2, s_3 of the covariance of local face centers (weighted by face area), for various geodesic radii (5%, 10%, 20%, 30%, 50%), and add the following features for each patch: $s_1/(s_1+s_2+s_3)$, $s_2/(s_1+s_2+s_3)$, $s_3/(s_1+s_2+s_3)$, $(s_1+s_2)/(s_1+s_2+s_3)$, $(s_1+s_3)/(s_1+s_2+s_3)$, $(s_2+s_3)/(s_1+s_2+s_3)$, s_1/s_2, s_1/s_3, s_2/s_3, $s_1/s_2+s_1/s_3$, $s_1/s_2+s_2/s_3$, $s_1/s_3+s_2/s_3$, yielding 75 features total.

c) Shape diameter: The Shape Diameter Function (SDF) [124] is computed using cones of angles 30, 60, 90, 120. For each cone, we get the weighted average, median, and squared mean of the samples. We include these shape diameters and their logarithmized versions with different normalizing parameters $\alpha = 1$, $\alpha = 2$, $\alpha = 4$, $\alpha = 8$. This yields 60 features representing different moments and approximations of the local shape diameter.

d) Distance from medial surface: For each of the cones above, we compute the diameter of the maximal inscribed sphere touching each face center and the corresponding medial surface point is roughly its center [91]. Then we send rays from this point uniformly sampled on a Gaussian sphere, gather the intersection points and measure the ray lengths. As with the shape diameter features, we use the weighted average, median and squared mean of the samples, we normalize and logarithmize them with the same above normalizing parameters. This yields 60 features.

e) Average Geodesic Distance: The Average Geodesic Distance (AGD) function has been used for shape matching [53, 157]. The function measures how "isolated" each face is from the rest of the surface e.g., limbs have usually higher AGD than other parts in humanoid models. The AGD for each face is computed by averaging the geodesic distance from its face center to all the other face centers. In our case, we also consider the squared mean and the $10th, 20th, ..., 90th$ percentile. Then, we normalize each of these 11 statistical measures by subtracting its minimum over all faces.

f) Shape contexts: Shape contexts have been used for 2D shape matching [10]. For each face,

we measure the distribution of other faces (weighted by their area) in 5 logarithmic geodesic distance and 6 uniform angle bins, where angles are measured relative to the normal of each face. The geodesic distance bins cover a distance range from 0 to the $95th$ percentile of all-pairs geodesic distances on the mesh and the angle bins cover a angle range from 0 to 180 degrees.

g) Spin images: Spin images [64] are created with a fixed 10×10 bin resolution (bin size 0.3), yielding 100 features.

h) Orientation features: We also include the x, y, z coordinates of each face center in the case that the training dataset is oriented.

i) Contextual label features: The above features provide a feature vector $\tilde{\mathbf{x}}$, which are used to learn contextual features, as described in Section 3.2.3. The output of a JointBoost classifier provides per-face probabilities $P(c|\tilde{\mathbf{x}})$. The contextual features are histograms of these probabilities around each face:

$$p_i^l = \sum_{j: d_b \leq \text{dist}(i,j) < d_{b+1}} a_j \cdot P(c_j = l) \tag{A.1}$$

where the bin b contains all faces j with distance range $[d_b, d_{b+1}]$ from face i. The a_j is the area of face j, normalized by the sum of face areas in the mesh. The distances between faces are measured from shortest parts (thus, approximating geodesic distances), as well as the Principal Component Axes and dominant symmetry axes of the mesh (measured in absolute values, since the principal axes are uniquely defined up to their sign). We use $B = 5$ ranges of distances $[d_b, d_{b+1})$ where d_b are chosen in the logarithmic space of $[0, \max_i(\max_j(\text{dist}(i,j)))]$, yielding $35|\mathscr{C}|$ contextual features.

A.2 Pairwise Features

For each pair of adjacent faces i and j, the following 191-dimensional feature vector $\mathbf{y}_{ij}$ is computed, for use in the Pairwise Energy Term (Section 3.2.2). We chose features that are potentially indicative of boundaries between parts.

a) Dihedral angles: Let ω_{ij} be the exterior dihedral angle between faces i and j. The scalar feature is given as $\min(\omega_{ij}/\pi, 1)$. We also compute the average of the dihedral angles around each edge at geodesic radii of 0.5%, 1%, 2%, 4% of the median of all-pairs geodesic distances in the mesh. We then exponentiate each of the above features with each exponent in the range 1 to 10. This yields 50 dihedral angle features in total.

b) Curvature and third-order surface derivatives: We first compute the curvature and the derivative-of-curvature tensor per mesh vertex at geodesic radii of 0.5%, 1%, 2%, 4% of the median of all-pairs geodesic distances. For each scale, we include the principal curvatures and the curvature derivatives along the principal directions (in order to assign curvature to each edge, we average the corresponding curvature values of its vertices). This yields 16 features.

b) Shape diameter differences: For each pair of adjacent faces, we include the absolute values of the differences between their corresponding 60 shape diameter features (as described above).

d) Distance from medial surface differences: Similarly, we include the absolute difference of the 60 distance-from-medial-surface features between adjacent faces (as described above).

e) Contextual label features: We also use pairwise contextual features, as described in Section 3.2.3. The above features form an initial feature vector $\tilde{\mathbf{y}}_{ij}$. We learn a JointBoost classifier $p(c_i \neq c_j | \tilde{\mathbf{y}}_{ij})$, and then bin them, as with the unary contextual features. Here, we bin them based only on geodesic distances in logarithmic space up to 5% of the median of all-pairs geodesic distances in the mesh. This yields 5 pairwise contextual features in total.

Appendix B

Properties and Features used For Learning Pen-And-Ink Illustrations

B.1 Image Preprocessing

Given an input illustration drawn by an artist, we apply the following steps to determine the hatching properties for each stroke pixel. First, we scan the illustration and align it to the rendering automatically by matching borders with brute force search. The following steps are sufficiently accurate to provide training data for our algorithms.

Intensity: The intensity I_i is set to the grayscale intensity of the pixel i of the drawing. It is normalized within the range $[0,1]$.

Thickness: Thinning is first applied to identify a single-pixel-wide skeleton for the drawing. Then, from each skeletal pixel, a Breadth-First Search (BFS) is performed to find the nearest pixel in the source image with intensity less than half of the start pixel. The distance to this pixel is the stroke thickness.

Orientation: The structure tensor of the local image neighborhood is computed at the scale

of the previously-computed thickness of the stroke. The dominant orientation in this neighborhood is given by the eigenvector corresponding to the smallest eigenvalue of the structure tensor. Intersection points are also detected, so that they can be omitted from orientation learning. Our algorithm marks as intersection points those points detected by a Harris corner detector in both the original drawing and the skeleton image. Finally, in order to remove spurious intersection points, pairs of intersection points are found with distance less than the local stroke thickness, and their centroid is marked as an intersection instead.

Spacing: For each skeletal pixel, a circular region is grown around the pixel. At each radius size, the connected components of the region are computed. If at least 3 pixels in the region are not connected to the center pixel, with orientation within $\pi/6$ of the center pixel's orientation, then the process halts. The spacing at the center pixel is set to the final radius.

Length: A BFS is executed on the skeletal pixels to count the number of pixels per stroke. In order to follow a single stroke (excluding pixels from overlapping cross-hatching strokes), at each BFS expansion, pixels are considered inside the current neighborhood with similar orientation (at most $\pi/12$ angular difference from the current pixel's orientation).

Hatching Level: For each stroke pixel, an ellipsoidal mask is created with its semi-minor axis aligned to the extracted orientation, and major radius equal to its spacing. All pixels belonging to any of these masks are given label $H_i = 1$. For each intersection pixel, a circular mask is also created around it with radius equal to its spacing. All connected components are computed from the union of these masks. If any connected component contains more than 4 intersection pixels, the pixels of the component are assigned with label $H_i = 2$. Two horizontal and vertical strokes give rise to a minimum cross-hatching region (with 4 intersections).

Hatching region boundaries: Pixels are marked as boundaries if they belong to boundaries of the hatching regions or if they are endpoints of the skeleton of the drawing.

We perform a final smoothing step (with a Gaussian kernel of width equal to the median of the spacing values) to denoise the properties.

B.2 Scalar features

There are 1204 scalar features ($\tilde{\mathbf{x}} \in \Re^{760}$) for learning the scalar properties of the drawing. The first 90 are mean curvature, gaussian curvature, maximum and minimum principal curvatures by sign and absolute value, derivatives of curvature, radial curvature and its derivative, view-dependent minimum and maximum curvatures [66], geodesic torsion in the projected viewing direction [23]. These are measured in three scales (1%, 2%, 5% relative to the median of all-pairs geodesic distances in the mesh) for each vertex. We also include their absolute values, since some hatching properties may be insensitive to sign. The above features are first computed in object-space and then, projected to image-space.

The next 110 features are based on local shape descriptors, also used in [69] for labeling parts. We compute the singular values s_1, s_2, s_3 of the covariance of vertices inside patches of various geodesic radii (5%, 10%, 20%) around each vertex, and also add the following features for each patch: $s_1/(s_1+s_2+s_3)$, $s_2/(s_1+s_2+s_3)$, $s_3/(s_1+s_2+s_3)$, $(s_1+s_2)/(s_1+s_2+s_3)$, $(s_1+s_3)/(s_1+s_2+s_3)$, $(s_2+s_3)/(s_1+s_2+s_3)$, s_1/s_2, s_1/s_3, s_2/s_3, $s_1/s_2+s_1/s_3$, $s_1/s_2+s_2/s_3$, $s_1/s_3+s_2/s_3$, yielding 45 features total. We also include 24 features based on the Shape Diameter Function (SDF) [124] and distance from medial surface [91]. The SDF features are computed using cones of angles 60, 90, and 120 per vertex. For each cone, we get the weighted average of the samples and their logarithmized versions with different normalizing parameters $\alpha = 1$, $\alpha = 2$, $\alpha = 4$. For each of the cones above, we also compute the distance of medial surface from each vertex. We measure the diameter of the maximal inscribed sphere touching each vertex. The corresponding medial surface point will be roughly its center. Then we send rays from this point uniformly sampled on a Gaussian sphere, gather the intersection points and measure the ray lengths. As with the shape diameter features, we use the weighted average of the samples, we normalize and logarithmize them with the same above normalizing parameters. In addition, we use the average, squared mean, $10th, 20th, ..., 90th$ percentile of the geodesic distances of each vertex to all the other mesh vertices, yielding 11 features. Finally, we use 30

shape context features [10], based on the implementation of [69]. All the above features are first computed in object-space per vertex and then, projected to image-space.

The next 53 features are based on functions of the rendered 3D object in image space. We use maximum and minimum image curvature, image intensity, and image gradient magnitude features, computed with derivative-of-gaussian kernels with $\sigma = 1, 2, 3, 5$, yielding 16 features. The next 12 features are based on shading under different models: $\vec{V} \cdot \vec{N}$, $\vec{L} \cdot \vec{N}$ (both clamped at zero), ambient occlusion, where $\vec{V}$, $\vec{L}$, and $\vec{N}$ are the view, light, and normal vectors at a point. These are also smoothed with gaussian kernels of $\sigma = 1, 2, 3, 5$. We also include the corresponding gradient magnitude, the maximum and minimum curvature of $\vec{V} \cdot \vec{N}$ and $\vec{L} \cdot \vec{N}$ features, yielding 24 more features. We finally include the depth value for each pixel.

We finally include the per pixel intensity of occluding and suggestive contours, ridges, valleys and apparent ridges extracted by the rtsc software package [118]. We use 4 different thresholds for extracting each feature line (the rtsc thresholds are chosen from the logarithmic space $[0.001, 0.1]$ for suggestive contours and valleys and $[0.01, 0.1]$ for ridges and apparent ridges). We also produce dilated versions of these features lines by convolving their image with gaussian kernels with $\sigma = 5, 10, 20$, yielding in total 48 features.

Finally, we also include all the above 301 features with their powers of 2 (quadratic features), -1 (inverse features), -2 (inverse quadratic features), yielding 1204 features in total. For the inverse features, we prevent divisions by zero, by truncating near-zero values to $1e-6$ (or $-1e-6$ if they are negative). Using these transformations on the features yielded slightly better results for our predictions.

B.3 Orientation features

There are 70 orientation features (θ) for learning the hatching and cross-hatching orientations. Each orientation feature is a direction in image-space; orientation features that begin as 3D vectors are projected to 2D. The first six features are based on surface principal curvature directions computed at 3 scales as above. Then, the next six features are based on surface local PCA axes projected on the tangent plane of each vertex corresponding to the two larger singular values of the covariance of multi-scale surfaces patches computed as above. Note that the local PCA axes correspond to candidate local planar symmetry axes [128]. The next features are: $\vec{L} \times \vec{N}$ and $\vec{V} \times \vec{N}$. The above orientation fields are undefined at some points (near umbilic points for curvature directions, near planar and spherical patches for the PCA axes, and near $\vec{L} \cdot \vec{N} = 0$ and $\vec{V} \cdot \vec{N} = 0$ for the rest). Hence, we use globally-smoothed direction based on the technique of [52]. Next, we include $\vec{L}$, and vector irradiance $\vec{E}$ [4]. The next 3 features are vector fields aligned with the occluding and suggestive contours (given the view direction), ridges and valleys of the mesh. The next 16 features are image-space gradients of the following scalar features: $\nabla(\vec{V} \cdot \vec{N})$, $\nabla(\vec{L} \cdot \vec{N})$, ambient occlusion and image intensity ∇I computed at 4 scales as above. The remaining orientation features are the directions of the first 35 features rotated by 90 degrees in the image-space.

Bibliography

[1] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Point set surfaces. In *VIS '01: Proceedings of the conference on Visualization '01*, 2001.

[2] Marc Alexa and Wolfgang Muller. Representing animations by principal components. In *Eurographics*, 2000.

[3] Dragomir Anguelov, Ben Taskar, Vassil Chatalbashev, Daphne Koller, Dinkar Gupta, Geremy Heitz, and Andrew Ng. Discriminative Learning of Markov Random Fields for Segmentation of 3D Scan Data. In *CVPR*, 2005.

[4] James Arvo. Applications of irradiance tensors to the simulation of non-lambertian phenomena. In *Proc. SIGGRAPH*, pages 335–342, New York, NY, USA, 1995. ACM.

[5] M. Attene, S. Katz, M. Mortara, G. Patane, M. Spagnuolo, and A. Tal. Mesh Segmentation - A Comparative Study. In *Proc. SMI*, 2006.

[6] Marco Attene, Bianca Falcidieno, and Michela Spagnuolo. Hierarchical Mesh Segmentation Based on Fitting Primitives. *Vis. Comput.*, 22(3), 2006.

[7] M. Bartlett, J. Movellan, and T. Sejnowski. Face recognition by independent component analysis. *IEEE Transations on Neural Networks*, 13(6):1450–1464, 2002.

[8] M. Belkin and P. Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in Neural Information Processing Systems*, volume 14, pages 585–591, 2002.

[9] Anthony J. Bell and Terrence J. Sejnowski. The independent components of natural scenes are edge filters. *Vision Research*, 37:3327–3338, 1997.

[10] S. Belongie, J. Malik, and J. Puzicha. Shape Matching and Object Recognition Using Shape Contexts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(4), 2002.

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[12] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast Approximate Energy Minimization via Graph Cuts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(11), 2001.

[13] Leo Breiman. Random forests. *Mach. Learn.*, 45(1), 2001.

[14] Yong Cao, Petros Faloutsos, and Frédéric Pighin. Unsupervised Learning for Speech Motion Editing. In *Proceedings of the Symposium on Computer Animation 2003*, pages 225–231, 2003.

[15] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. In *SIGGRAPH '01*, pages 67–76, 2001.

[16] Xiaobai Chen, Aleksey Golovinskiy, and Thomas Funkhouser. A Benchmark for 3D Mesh Segmentation. *ACM Trans. Graphics*, 28(3), 2009.

[17] David Cohen-Steiner and Jean-Marie Morvan. Restricted delaunay triangulations and normal cycle. In *Proceedings of the Symposium on Computational Geometry 2003*, pages 312–321, 2003.

[18] Forrester Cole, Aleksey Golovinskiy, Alex Limpaecher, Heather Stoddart Barros, Adam Finkelstein, Thomas Funkhouser, and Szymon Rusinkiewicz. Where Do People Draw Lines? *ACM Trans. Graph.*, 27(3), 2008.

[19] Dorin Comaniciu and Peter Meer. Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, 2002.

[20] Pierre Comon. Independent component analysis, a new concept? *Signal Processing*, 36(3):287–314, 1994.

[21] Doug DeCarlo, Adam Finkelstein, and Szymon Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. In *Proceedings of the International symposium on Non-photorealistic animation and rendering 2004*, pages 15–24, 2004.

[22] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. Suggestive Contours For Conveying Shape. *ACM Trans. Graph.*, 22(3), 2003.

[23] Doug DeCarlo and Szymon Rusinkiewicz. Highlight lines for conveying shape. In *NPAR*, 2007.

[24] Pierre Demartines and Jeanny Herault. Curvilinear component analysis: A self-organizing neural network for nonlinear mapping of data sets. *IEEE Trans Neural Netw*, 1(8), 1997.

[25] Udo Diewald, Tobias Preusser, and Martin Rumpf. Anisotropic diffusion in vector field visualization on euclidean domains and surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):139–149, 2000.

[26] Pinar Duygulu, Kobus Barnard, Nando de Freitas, and David Forsyth. Object Recognition as Machine Translation: Learning a Lexicon for a Fixed Image Vocabulary. In

Proc. ECCV, 2002.

[27] Michael Eigensatz, Robert W. Sumner, and Mark Pauly. Curvature-domain shape processing. *Computer Graphics Forum (Eurographics Proceedings)*, 27(2):241–250, 2008.

[28] Gershon Elber. Line Art Illustrations of Parametric and Implicit Forms. *IEEE TVCG*, 4(1):71–81, 1998.

[29] Wei-Wen Feng, Byung-Uck Kim, and Yizhou Yu. Real-time data driven deformation using kernel canonical correlation analysis. *ACM Trans. Graph.*, 27(3), 2008.

[30] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. pages 726–740, 1987.

[31] Shachar Fleishman, Daniel Cohen-Or, and Cláudio T. Silva. Robust moving least-squares fitting with sharp features. *ACM Trans. Graph.*, 24(3), 2005.

[32] Arthur Flexer. Statistical evaluation of neural network experiments: Minimum requirements and current practice. In *Cybernetics and Systems*, pages 1005–1008, 1996.

[33] William T. Freeman, Joshua Tenenbaum, and Egon Pasztor. Learning style translation for the lines of a drawing. *ACM Trans. Graph.*, 22(1):33–46, 2003.

[34] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Thirteenth International Conference on Machine Learning*, 1996.

[35] J. Friedman, T. Hastie, and R. Tibshirani. Additive Logistic Regression: a Statistical View of Boosting. *The Annals of Statistics*, 38(2), 2000.

[36] Hongbo Fu, Daniel Cohen-Or, Gideon Dror, and Alla Sheffer. Upright Orientation of Man-made Objects. *ACM Trans. Graph.*, 27(3), 2008.

[37] Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. Modeling by example. *ACM Trans. Graph.*, 23(3), 2004.

[38] Ran Gal and Daniel Cohen-Or. Salient Geometric Features for Partial Shape Matching and Similarity. *ACM Trans. Graph.*, 25(1), 2006.

[39] Joao Gama and Pavel Brazdil. Cascade Generalization. *Mach. Learn.*, 41(3), 2000.

[40] Aleksey Golovinskiy and Thomas Funkhouser. Randomized Cuts for 3D Mesh Analysis. *ACM Trans. on Graph.*, 27(5), 2008.

[41] Aleksey Golovinskiy and Thomas Funkhouser. Consistent Segmentation of 3D Models. *Proc. SMI*, 33(3), 2009.

[42] Aleksey Golovinskiy, Vladimir G. Kim, and Thomas Funkhouser. Shape-based Recognition of 3D Point Clouds in Urban Environments. In *Proc. ICCV*, 2009.

[43] Todd Goodwin, Ian Vollick, and Aaron Hertzmann. Isophote Distance: A Shading Approach to Artistic Stroke Thickness. In *Proc. NPAR*, pages 53–62, 2007.

[44] Arthur L. Guptill. *Rendering in Pen and Ink*. Watson-Guptill, edited by Susan E. Meyer, 1997.

[45] J. Hamel and T. Strothotte. Capturing and Re-Using Rendition Styles for Non-Photorealistic Rendering. *Computer Graphics Forum*, 18(3):173–182, 1999.

[46] F. R. Hampel, E. M. Ronchetti, P. J. Rousseeuw, and W. A. Stahel. *Robust Statistics: The Approach Based on Influence Functions*. Wiley-Interscience, 1986.

[47] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning:*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2009.

[48] Xuming He, R.S. Zemel, and M. A. Carreira-Perpiñán. Multiscale Conditional Random Fields for Image Labeling. In *Proc. CVPR*, volume 2, 2004.

[49] Paul S. Heckbert and Michael Garland. Optimal triangulation and quadric-based surface simplification. *Computational Geometry Theory and Applications*, 14:49–65, 1999.

[50] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image Analogies. *Proc. SIGGRAPH*, 2001.

[51] Aaron Hertzmann, Nuria Oliver, Brian Curless, and Steven M. Seitz. Curve Analogies. In *Proc. EGWR*, 2002.

[52] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *Proc. SIGGRAPH*, pages 517–526, 2000.

[53] Masaki Hilaga, Yoshihisa Shinagawa, Taku Kohmura, and Tosiyasu L. Kunii. Topology Matching for Fully Automatic Similarity Estimation of 3d Shapes. In *SIGGRAPH*, 2001.

[54] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006.

[55] Qixing Huang, Martin Wicke, Bart Adams, and Leonidas Guibas. Shape Decomposition Using Modal Analysis. *J. Computer Graphics Forum*, 28, 2009.

[56] A. Hyvärinen. Fast and Robust Fixed-Point Algorithms for Independent Component Analysis. *IEEE Transations on Neural Network*, 10(3):626–634, 1999.

[57] A. Hyvärinen and E. Oja. Independent component analysis: algorithms and applications. *Neural Networks*, 13(4-5), 2000.

[58] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A Sketching Interface for 3d Freeform Design. In *SIGGRAPH*, 2007.

[59] Victoria Interrante, Henry Fuchs, and Stephen Pizer. Enhancing Transparent Skin Surfaces with Ridge and Valley Lines. In *Proceedings of the 6th conference on Visualization 1995*, pages 52–59, 1995.

[60] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Comput.*, 3(1), 1991.

[61] Doug L. James and Kayvon Fatahalian. Precomputing interactive dynamic deformable scenes. *ACM Transactions on Graphics*, 22(3):879–887, 2003.

[62] Doug L. James and Christopher D. Twigg. Skinning mesh animations. *ACM Transactions on Graphics*, 24(3):399–407, 2005.

[63] Pierre-Marc Jodoin, Emric Epstein, Martin Granger-Piché, and Victor Ostromoukhov. Hatching by Example: a Statistical Approach. In *Proc. NPAR*, pages 29–36, 2002.

[64] Andrew Johnson and Martial Hebert. Using Spin Images for Efficient Object Recognition in Cluttered 3D Scenes. *IEEE Trans. PAMI*, 21(5):433–449, 1999.

[65] Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural Comput.*, 6(2), 1994.

[66] Tilke Judd, Frédo Durand, and Edward Adelson. Apparent ridges for line drawing. *ACM Trans. Graph.*, 26(3), 2007.

[67] Robert Kalnins, Lee Markosian, Barbara Meier, Michael Kowalski, Joseph Lee, Philip Davidson, Matthew Webb, John Hughes, and Adam Finkelstein. WYSIWYG NPR: drawing strokes directly on 3D models. In *Proc. SIGGRAPH*, pages 755–762, 2002.

[68] Evangelos Kalogerakis, Aaron Hertzmann, and Karan Singh. Learning 3D Mesh Segmentation and Labeling. *ACM Transactions on Graphics*, 29(3), 2010.

[69] Evangelos Kalogerakis, Aaron Hertzmann, and Karan Singh. Learning 3d mesh segmentation and labeling. *ACM Trans. Graph.*, 29(3), 2010.

[70] Evangelos Kalogerakis, Derek Nowrouzezahrai, Patricio Simari, James McCrae, Aaron Hertzmann, and Karan Singh. Data-driven curvature for real-time line drawing of dynamic scene. *ACM Transactions on Graphics*, 28(1), 2009.

[71] Evangelos Kalogerakis, Derek Nowrouzezahrai, Patricio Simari, and Karan Singh. Extracting lines of curvature from noisy point clouds. *Special Issue of the Elsevier Computer-Aided Design journal on Point-Based Computational Techniques*, 41(4):282–292, 2009.

[72] Evangelos Kalogerakis, Patricio Simari, Derek Nowrouzezahrai, and Karan Singh. Robust statistical estimation of curvature on discretized surfaces. In *Proceedings of the Eurographics/ACM Siggraph Symposium on Geometry Processing*, 2007.

[73] S. Katz, G. Leifman, and A. Tal. Mesh segmentation using feature point and core extraction. *Visual Computer*, 21(8), 2005.

[74] Sagi Katz and Ayellet Tal. Hierarchical Mesh Decomposition Using Fuzzy Clustering and Cuts. *ACM Trans. Graphics*, 2003.

[75] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proc. SGP*, pages 61–70, 2006.

[76] SungYe Kim, Insoo Woo, Ross Maciejewski, , and David S. Ebert. Automated Hedcut Illustration using Isophotes. In *Proc. Smart Graphics*, 2010.

[77] Yongjin Kim, Jingyi Yu, Xuan Yu, and Seungyong Lee. Line-art Illustration of Dynamic and Specular Surfaces. *ACM Trans. Graphics*, 2008.

[78] Scott Konishi and A.L. Yuille. Statistical Cues for Domain Specific Image Segmentation With Performance Analysis. *Proc. CVPR*, 2000.

[79] Vladislav Kraevoy, Dan Julius, and Alla Sheffer. Model Composition From Interchangeable Components. In *Proc. PG*, 2007.

[80] Paul Kry, Doug James, and Dinesh Pai. Eigenskin: real time large deformation character skinning in hardware. In *Proc. SCA*, pages 153–159, 2002.

[81] Sanjiv Kumar and Martial Hebert. Discriminative Random Fields: A Discriminative Framework for Contextual Interaction in Classification. In *Proc. ICCV*, 2003.

[82] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *ICML*, 2001.

[83] Yu-Kun Lai, Shi-Min Hu, Ralph R. Martin, and Paul L. Rosin. Fast Mesh Segmentation Using Random Walks. In *ACM symposium on Solid and Physical Modeling*, 2008.

[84] Guillaume Lavoué and Christian Wolf. Markov Random Fields for Improving 3D Mesh Analysis and segmentation. In *Eurographics workshop on 3D object retrieval*, 2008.

[85] Yunjin Lee, Lee Markosian, Seungyong Lee, and John F. Hughes. Line drawings via abstracted shading. *ACM Transactions on Graphics*, 26(3):18, 2007.

[86] J. P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *SIGGRAPH 2000 Proceedings*, pages 165–172, 2000.

[87] Xin Li, Xianfeng Gu, and Hong Qin. Surface matching using consistent pants decomposition. In *ACM Symposium on Solid and Physical Modeling*, 2008.

[88] E. Lim and David Suter. Conditional Random Field for 3D Point Clouds With Adaptive Data Reduction. In *Cyberworlds*, 2007.

[89] Hsueh-Yi Sean Lin, Hong-Yuan Mark Liao, and Ja-Chen Lin. Visual Salience-Guided Mesh Decomposition. *IEEE Transactions on Multimedia*, 9(1), 2007.

[90] Rong Liu and Hao Zhang. Segmentation of 3D Meshes Through Spectral Clustering. In *Proc. PG*, 2004.

[91] Rong F. Liu, Hao Zhang, Ariel Shamir, and Daniel Cohen-Or. A Part-Aware Surface Metric for Shape Analysis. *Computer Graphics Forum, (Eurographics 2009)*, 28(2), 2009.

[92] Eric B. Lum and Kwan-Liu Ma. Expressive line selection by example. *The Visual Computer*, 21(8):811–820, 2005.

[93] Alan P. Mangan and Ross T. Whitaker. Partitioning 3D Surface Meshes Using Watershed Segmentation. *IEEE Trans. on Vis. and Comp. Graph.*, 5(4), 1999.

[94] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering. In *SIGGRAPH 1997 Proceedings*, pages 415–420, 1997.

[95] T. Mertens, J. Kautz, J. Chen, P. Bekaert, and F. Durand. Texture transfer using geometry correlation. In *Proceedings of Eurographics Symposium on Rendering*, 2006.

[96] Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H. Barr. Discrete differential-geometry operators for triangulated 2-manifolds. In *Visualization and Mathematics III*, pages 35–57. 2002.

[97] Alex Mohr and Michael Gleicher. Building efficient, accurate character skins from examples. *ACM Transactions on Graphics*, 22(3):562–568, 2003.

[98] Daniel Munoz, Nicolas Vandapel, and Martial Hebert. Directional Associative Markov Network for 3-D Point Cloud Classification. In *Proc. 3DPVT*, 2008.

[99] Sreerama K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Min. Knowl. Discov.*, 2(4), 1998.

[100] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer-Verlag, 1999.

[101] J. D. Northrup and Lee Markosian. Artistic Silhouettes: A Hybrid Approach. In *Proceedings of the International symposium on Non-photorealistic animation and rendering 2000*, pages 31–38, 2000.

[102] Derek Nowrouzezahrai, Evangelos Kalogerakis, and Eugene Fiume. Shadowing dynamic scenes with arbitrary BRDFs. In *Eurographics 2009 (To Appear)*, 2009.

[103] Derek Nowrouzezahrai, Evangelos Kalogerakis, Patricio Simari, and Eugene Fiume. Shadowed relighting of dynamic geometry with 1d BRDFs. In *Eurographics 2008 Proceedings*, 2008.

[104] Derek Nowrouzezahrai, Patricio Simari, Evangelos Kalogerakis, Karan Singh, and Eugene Fiume. Compact and efficient generation of radiance transfer for dynamically articulated characters. In *Proceedings of the GRAPHITE 2007*, pages 147–154, 2007.

[105] Yutaka Ohtake, Alexander Belyaev, and Hans-Peter Seidel. Ridge-valley lines on meshes via implicit surface fitting. *ACM Transactions on Graphics*, 23(3):609–612, 2004.

[106] Cengiz Oztireli, Gael Guennebaud, and Markus Gross. Feature preserving point set surfaces based on non-linear kernel regression. In *Eurographics 2009*, pages 493–501, 2009.

[107] Jonathan Palacios and Eugene Zhang. Rotational Symmetry Field Design on Surfaces. *ACM Trans. Graph.*, 2007.

[108] Yuri Pekelny and Craig Gotsman. Articulated Object Reconstruction and Markerless Motion Capture from Depth Video. *J. Computer Graphics Forum*, 27:399–408, 2008.

[109] K. Polthier. *Polyhedral surfaces of constant mean curvature*. PhD thesis, TU-Berlin, 2002.

[110] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-Time Hatching. In *Proc. SIGGRAPH*, 2001.

[111] Lutz Prechelt. A quantitative study of experimental evaluations of neural network learning algorithms: Current research practice. In *4th Intl. Conf. on Artificial Neural Networks*, pages 223–227, 1995.

[112] Frank Rosenblatt. *The Perceptron: A Perceiving and Recognizing Automaton*. Report No. 85-460-1, New York: Cornell Aeronautical Laboratory, First edition, first issue, 1957.

[113] Peter J. Rousseeuw. Least median of squares regression. *Journal of the American Statistical Association*, 79(388):871–880, 1984.

[114] Sam T. Roweis and Lawrence K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323 – 2326, 2000.

[115] Szymon Rusinkiewicz. Estimating Curvatures and Their Derivatives on Triangle Meshes. In *Proceedings of the International Symposium on 3D Data Processing, Visualization and Transmission 2004*, pages 486–493, 2004.

[116] Szymon Rusinkiewicz. Trimesh2 library. http://www.cs.princeton.edu/gfx/proj/trimesh2/, 2007.

[117] Szymon Rusinkiewicz, Michael Burns, and Doug DeCarlo. Exaggerated shading for depicting shape and detail. *Proc. SIGGRAPH*, 25(3):1199–1205, 2006.

[118] Szymon Rusinkiewicz and Doug DeCarlo. rtsc library. http://www.cs.princeton.edu/gfx/proj/sugcon/, 2007.

[119] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In *Proc. SIGGRAPH*, pages 197–206, 1990.

[120] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen-and-ink illustration. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 101–108, 1994.

[121] Mirko Sattler, Ralf Sarlette, and Reinhard Klein. Simple and efficient compression of animation sequences. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 209–217, 2005.

[122] Yaar Schnitman, Yaron Caspi, Daniel Cohen-or, and Dani Lischinski. Inducing Semantic Segmentation From an Example. In *Proc. ACCV*, 2006.

[123] Ariel Shamir. A Survey on Mesh Segmentation Techniques. *Computer Graphics Forum*, 26(6), 2008.

[124] Lior Shapira, Shy Shalom, Ariel Shamir, Richard H. Zhang, and Daniel Cohen-Or. Contextual Part Analogies in 3D Objects. *International Journal of Computer Vision*, In Press.

[125] S. Shlafman, A. Tal, and S. Katz. Metamorphosis of Polyhedral Surfaces Using Decomposition. In *Eurographics*, 2002.

[126] J. Shotton, M. Johnson, and R. Cipolla. Semantic Texton Forests for Image Categorization and Segmentation. In *Proc. CVPR*, 2008.

[127] Jamie Shotton, John Winn, Carsten Rother, and Antonio Criminisi. TextonBoost for Image Understanding: Multi-Class Object Recognition and Segmentation by Jointly Modeling Texture, Layout, and Context. *Int. J. Comput. Vision*, 81(1), 2009.

[128] Patricio Simari, Evangelos Kalogerakis, and Karan Singh. Folding Meshes: Hierarchical Mesh Segmentation Based on Planar Symmetry. In *SGP*, 2006.

[129] Patricio Simari, Derek Nowrouzezahrai, Evangelos Kalogerakis, and Karan Singh. Multi-objective shape segmentation and labeling. *Computer Graphics Forum*, 28(5), 2009.

[130] Mayank Singh and Scott Schaefer. Suggestive Hatching. In *Proc. Computational Aesthetics*, 2010.

[131] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. In *Proc. SIGGRAPH*, pages 527–536, 2002.

[132] Peter-Pike J. Sloan, Rose Charles F., and Michael F. Cohen. Shape by example. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 135–143, 2001.

[133] Olga Sorkine and Daniel Cohen-Or. Least-squares meshes.

[134] Charles V. Stewart. Robust parameter estimation in computer vision. *SIAM Rev.*, 41(3), 1999.

[135] Tobias Isenberg William M. Andrews Wei Chen Mario Costa Sousa David S. Ebert SungYe Kim, Ross Maciejewski. Stippling by example. In *Proceedings of the 7th international symposium on Non-photorealistic animation and rendering (NPAR)*, 2009.

[136] Charles Sutton and Andrew McCallum. Piecewise pseudolikelihood for efficient training of conditional random fields. In *Proceedings of the 24th international conference on*

Machine learning, pages 863–870, 2007.

[137] G. Taubin. Estimating the tensor of curvature of a surface from a polyhedral approximation. In *Proceedings of the Fifth International Conference on Computer Vision 1995*, 1995.

[138] Joshua B. Tenenbaum, Vin Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[139] Jean-Philippe Thirion and Alexis Gourdon. The 3D marching lines algorithm. *Graphical Models and Image Processing*, 58(6):503–509, 1996.

[140] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society*, 58:267–288, 1994.

[141] M. E. Tipping. Sparse Bayesian Learning and the Relevance Vector Machine. *J. Machine Learning Res.*, (1):211–244, 2001.

[142] Antonio Torralba, Kevin P. Murphy, and William T. Freeman. Sharing Visual Features for Multiclass and Multiview Object Detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(5), 2007.

[143] Zhuowen Tu. Auto-context and its Application to High-level Vision Tasks. In *Proc. CVPR*, 2008.

[144] Zhuowen Tu, Xiangrong Chen, Alan Yuille, and Song-Chun Zhu. Image Parsing: Unifying Segmentation, Detection, and Recognition. *International Journal of Computer Vision*, 63(2), 2005.

[145] Greg Turk and David Banks. Image-guided streamline placement. In *SIGGRAPH*, 1996.

[146] Greg Turk and James F. O'brien. Modelling with implicit surfaces that interpolate. *ACM Trans. Graph.*, 21(4), 2002.

[147] Romain Vergne, Pascal Barla, Xavier Granier, and Christophe Schlick. Apparent relief: a shape descriptor for stylized shading. In *Proceedings of the International symposium on Non-photorealistic animation and rendering 2008*, 2008.

[148] Kiri Wagstaff, Claire Cardie, Seth Rogers, and Stefan Schrödl. Constrained k-means clustering with background knowledge. In *ICML*, 2001.

[149] Robert Y. Wang, Kari Pulli, and Jovan Popović. Real-time enveloping with rotational regression. *ACM Transactions on Graphics*, 26(3):73, 2007.

[150] Sanford Weisberg. *Applied Linear Regression*. Wiley/Interscience, 3rd edition edition, 2003.

[151] Georges Winkenbach and David Salesin. Computer-generated pen-and-ink illustration. In *Proc. SIGGRAPH*, pages 91–100, 1994.

[152] Georges Winkenbach and David Salesin. Rendering parametric surfaces in pen and ink. In *Proc. SIGGRAPH*, pages 469–476, 1996.

[153] Shin Yoshizawa, Alexander Belyaev, Hideo Yokota, and Hans-Peter Seidel. Fast and faithful geometric algorithm for detecting crest lines on meshes. In *Pacific Graphics 2007 Proceedings*, pages 231–237, 2007.

[154] Jingyi Yu, Xiaotian Yin, Xianfeng Gu, Leonard McMillan, and Steven Gortler. Focal surfaces of discrete geometry. In *Proceedings of the Symposium on Geometry Processing 2007*, pages 23–32, 2007.

[155] Richard Zemel and Toniann Pitassi. A gradient-based boosting algorithm for regression problems. In *Neural Information Processing Systems*, 2001.

[156] Kun Zeng, Mingtian Zhao, Caiming Xiong, and Song-Chun Zhu. From image parsing to painterly rendering. *ACM Trans. Graph.*, 29, 2009.

[157] Eugene Zhang, Konstantin Mischaikow, and Greg Turk. Feature-based Surface Parameterization and Texture Mapping. *ACM Trans. Graph.*, 24(1), 2005.