Instance-level Semantic Segmentation by Deep Normalized Cuts

Alexander Davidson

Master's thesis
2017:E55

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

Centrum Scientiarum Mathematicarum

Abstract

Instance-level semantic segmentation refers to the task of assigning each pixel in an image an object class and an instance identity label. Due to its complexity, this problem has received little attention by the research community in the past. However, owing to the recent successes of deep learning technology, solving it suddenly seems possible.

Starting from an initial pixel-level semantic segmentation, we combine ideas from spectral clustering and deep learning to investigate the suitability of using normalized cuts for additional instance-level segmentation. Specifically, the similarity matrix on which cuts are based is learned from examples using a convolutional network. We use the Cityscapes dataset for urban street scene understanding to train our deep normalized cuts model to segment individual cars. When using the ground truth number of instances to help determine the correct number of cuts, our method attains almost twice the performance of our connected components baseline.

Acknowledgements

The author would like to thank Prof. Dr. Cristian Sminchisescu for suggesting the topic of this thesis, for providing access to a GPU, and for offering some guidance along the way.

Contents

Chapter 1

Introduction

1.1 Objective

Object recognition comes in many forms, two of the most popular being **object detection** and **pixel-level semantic segmentation** (Hariharan et al., 2014). In object detection, the task is to provide a tight, rectangular bounding box around each object of a particular class in an image. Typically, a predicted bounding box is deemed correct if it overlaps with a ground truth box by 50% or more, and the performance of an algorithm is evaluated based on its **precision** (the fraction of correctly predicted boxes) and **recall** (the fraction of correct boxes reported). While an object detection system attempts to find and estimate the spatial location of each instance of a class, the bounding boxes allow the detected objects to be localized only very coarsely.

Pixel-level semantic segmentation requires each pixel in an image to be assigned to one of several semantic classes. The standard measure used to assess the performance of an algorithm concerned with pixel-level segmentation is the **intersection over union** metric, which for each class computes the fraction of intersecting pixels between the predicted and ground truth segments. The pixel-level segmentation task lacks the notion of object instances. While an algorithm may successfully report all "car" pixels in an image, it does not indicate how many cars there are, nor does it explicitly predict the precise spatial extent of any of those cars.

Intersecting these two forms of object recognition naturally leads to the problem of **instance-level semantic segmentation**. Here, the task is to detect all object instances of a class in an image and, for every instance, indicate what pixels belong to it. The performance measure used for this task is analogous to the one used for object detection. Compared to object detection and pixel-level semantic segmentation, the problem of instance-level semantic segmentation requires a more complex, arguably more useful output.

The objective of this thesis is to investigate a particular approach for solving the problem of instance-level semantic segmentation. Starting with an algorithm that performs pixel-level semantic segmentation, the goal is to further partition the pixels of a certain class into individual object instances using the method of **normalized cuts** (Shi and Malik, 2000). Following the work of Bach and Jordan (2006) and Ionescu et al. (2015), **features** on which those cuts are based are to be learned through **deep learning**, using a **convolutional neural network**. We refer to the resulting method as **deep normalized cuts** (DNC).

1.2 State of the Art

The recent resurrection of deep learning, in part due to today's availability of large-scale datasets and the advent of general purpose GPU computing, has stimulated great progress in many fields, including that of computer vision. These days, methods that exploit neural network technology in one way or another dominate leaderboards of many computer vision benchmarks. Deep learning research is very active, and advances are made at a fast pace. As a matter of fact, several of the methods mentioned below have been proposed during the writing of this thesis.

Many recent systems that perform instance-level semantic segmentation make use of the **region-based convolutional neural network** (R-CNN) object detection framework (Girshick et al., 2014). R-CNN consists of three modules. First, class-agnostic **region proposals** that define the set of candidate detections are generated through **selective search** (Uijlings et al., 2013). Secondly, a convolutional neural network extracts a fixed-length feature vector for each (warped) region. Lastly, a set of class-specific, linear **support vector machines** (SVMs) are used for classification. During inference, some 2,000 region proposals are generated. **Non-maximum suppression** is used to discard sub-optimal bounding boxes, and the SVMs are used to predict the semantic class of each box.

Fast R-CNN (Girshick, 2015) makes some modifications to R-CNN that lead to improvements in terms of both speed and accuracy. **DeepMask** (Pinheiro et al., 2015) learn to propose candidate object segments, which are then classified by Fast R-CNN.

Faster R-CNN (Ren et al., 2015) generates region proposals using a **region proposal network** (RPN), and is currently the leading framework in several object detection benchmarks. **Mask R-CNN** (He et al., 2017) achieve state-of-the-art performance by adapting Faster R-CNN for instance-level semantic segmentation. In parallel to performing bounding box regression and classification, Mask R-CNN additionally predicts a binary object mask. This is different from many other recent systems, in which classification is dependent on mask predictions.

Another successful method for instance-level semantic segmentation is proposed by Li et al. (2016). It combines the segment proposal system of Dai, He, Li, Ren and Sun (2016) with the object detection system of Dai, Li, He and Sun (2016). The common idea in these systems is to detect instances from **position-sensitive** feature maps that jointly consider object classes, bounding boxes, and segmentation masks, making their implementations fast.

Some methods based on object detectors may assign multiple instances to the same pixel. A conceptually different way of performing instance-level semantic segmentation, that resolves this problem, is to instead begin with a pixel-level semantic segmentation, then attempt to further partition the segments of a given class into individual object instances. The **pyramid scene parsing network** (PSPNet) of Zhao et al. (2016) demonstrates state-of-the-art performance on several pixel-level segmentation benchmarks. Starting from an image's pixel-level semantic segmentation, as produced by PSPNet, Bai and Urtasun (2016) combine intuitions from the classical watershed transform and modern deep learning to produce an energy map of the image, where object instances are represented as energy basins. A cut at a single energy level yields connected components that correspond to object instances. Arnab and Torr (2017) use an initial pixel-level semantic segmentation module and cues from an object detector to feed an instance-aware subnetwork that uses an end-to-end conditional random field to predict instances. Both of these methods prove successful at performing instance-level semantic segmentation.

The mechanism used by human beings to infer object segments is arguably more similar to the above methods that start off with object detection. Be that as it may, this thesis follows the latter approach and attempts to carve out individual object instances from pixel-level segmentations. (It does, however, not attempt to compete with current state of the art methods.)

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 begins with a brief introduction to the field of machine learning, then moves on to discuss aspects of deep learning that are important for understanding deep normalized cuts. The primary reference for this chapter is the recent textbook of Goodfellow et al. (2016). Most claims not explicitly cited here can be traced back to this textbook. Chapter 3 is concerned with image partitioning through normalized cuts, and how features suitable for normalized cuts can be learned from data. Chapter 4 first describes the dataset and implementation used for experimentation, then presents the result of applying the latter on the former. Chapter 5 concludes.

Chapter 2

Deep Learning

2.1 Machine Learning

2.1.1 Learning Algorithms

A **machine learning algorithm** is an algorithm that can learn from data. The following definition, which is due to Mitchell (1997), is often used.

> *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

Given this definition, learning is not the task. Instead, learning refers to the process of acquiring the ability to perform the task. Tasks are commonly described in terms of how the algorithm should process an **example**, which is a collection of **features** associated with some object or event. In the case of an image, the features could be the intensity values of the image's pixels.

The performance measure offers a means to quantitatively measure the abilities of a machine learning algorithm. It is specific to the task being solved. In order to assess how aptly an algorithm will work once deployed in the real world, we are typically interested in how well it performs on previously unseen data. To this end, the performance is not evaluated on the data used to **train** the algorithm, but on a separate **test set**.

Machine learning algorithms can be coarsely grouped as **unsupervised** or **supervised** by the kind of experience they are provided during training. In essence, unsupervised learning involves observing examples $\boldsymbol{x}$ of a random vector $\mathbf{x}$ and trying to learn the probability distribution $p_{\mathbf{x}}(\boldsymbol{x})$, or some useful properties thereof, whereas the supervised learning case additionally involves observing instances $\boldsymbol{y}$ of an associated target $\mathbf{y}$, and learning to predict $\mathbf{y}$ from $\mathbf{x}$, typically by estimating $p_{\mathbf{y}}(\boldsymbol{y} \mid \mathbf{x} = \boldsymbol{x})$.

2.1.2 Generalization

The central challenge in machine learning lies in designing algorithms that perform well on unobserved data, an ability called **generalization**. During training, some measure of **training error** can typically be computed. Minimizing this error is simply an optimization problem. What sets machine learning apart from optimization is that we also want the **generalization error** to be small. The generalization error is defined as the expected value of the error on a new input, taken over possible inputs drawn from the distribution of inputs that we expect to encounter in practice. It is usually approximated by the average error on the test set examples.

If the training and test sets are gathered arbitrarily, there is no reason to expect the test error to be affected when only the training set is observed during training. Some assumptions about how the data is collected are needed. The training and test data are generated by a probability distribution over datasets called the **data generating process**. If we assume that the examples in each dataset are independent from each other, and that the two datasets are identically distributed, drawn from the same probability distribution, then we may describe the data generating process with a probability distribution over a single example. This shared data generating distribution is then used to generate every training and test example. It follows that the expected training error of a randomly selected model equals the expected test error of that model, and, in turn, that machine learning algorithms can generalize well from a finite set of training examples.

In machine learning, we want both the training error and the gap between the training and test errors to be small. These concerns correspond to two principal machine learning challenges, called **underfitting** and **overfitting**. Underfitting refers to the situation in which the model fails to obtain a small enough training error. If the training error is small but the test error is not, overfitting has occurred.

By adjusting the **capacity** of the model, we can control how likely it is to underfit or overfit the training data. A model's capacity may be viewed as its ability to fit a wide range of functions. A model with low capacity may not be able to fit the training set. A model with high capacity is prone to overfit the training set by capturing specific properties of the training data that do not fully generalize to the test data.

The capacity of a machine learning algorithm can be controlled via the choice of its **hypothesis space**, which is the set of functions from which it selects the solution. Adding more complex functions to the hypothesis space increases the model capacity. The model describes which family of functions the learning algorithm may chose from when adjusting its parameters in order to lower the training error. This is referred to as the **representational** capacity of the model. To find the optimal function within this family can be very difficult.

Typically, the learning algorithm simply settles on a function that results in a small enough training error. As a consequence, the **effective** capacity of the model may be less than the representational capacity of the model family.

2.1.3 The No Free Lunch Theorem

The **no free lunch theorem** for machine learning (Wolpert, 1996) states that every classification algorithm will attain the same error rate on previously unseen data, when averaged across every possible data generating distribution. The meaning of this is that no machine learning algorithm is universally any better than any other. Luckily, if taking into consideration only the types of probability distributions we expect to confront in practice, then we can build algorithms that perform well on those distributions. The ambition of machine learning research is to understand what machine learning algorithms work well on data drawn from data generating distributions that are relevant in the real world, and not to find a universal machine learning algorithm.

2.1.4 Regularization

A machine learning algorithm is designed to work well on a particular task by incorporating certain preferences into it. Apart from altering the representational capacity of the model, we can also give a learning algorithm a preference for some solutions in its hypothesis space to others. While all solutions are allowed, an unpreferred solution must fit the training data significantly better than a preferred solution in order to be selected.

Preferences can be expressed in different ways. These different ways are collectively known as **regularization**, which is any modification made to a learning algorithm that is intended to lower its generalization error but not its training error. Regularization is an important topic in machine learning, and just as with the learning algorithm itself, what kind of regularization to use depends on the nature of the particular task being solved.

2.1.5 Hyperparameters

In addition to the model parameters being learned, a machine learning algorithm usually has **hyperparameters** that influence its behavior. Sometimes a parameter is chosen to be a hyperparameter because it is difficult to optimize, but often a parameter must be a hyperparameter because it is inappropriate to learn it from the training data. This includes all hyperparameters that control model capacity, because learning such parameters inevitably results in the highest possible capacity being selected, which might cause overfitting.

To help us choose appropriate values for the hyperparameters, we construct a **validation set** of examples that are not used during training. It is important that the examples in the test set are in no way used to make decisions about a model, including the associated hyperparameters. To this end, the training data is split into two disjoint subsets. One of these, also referred to as the training set, is used to learn the parameters of the model, whereas the other subset, the validation set, is used to approximate the generalization error during or after training, which allows for the hyperparameters to be tuned. The validation set error is likely to underestimate the generalization error, since the validation set is used to adjust the hyperparameters. However, it usually does so by a smaller amount than does the training set error. Upon the completion of hyperparameter optimization, the generalization error may be estimated using the test set.

2.1.6 Designing a Machine Learning Algorithm

A typical machine learning algorithm can be described as a combination of a dataset, a cost function to minimize, an optimization scheme, and a model. These building blocks can be interchanged mostly independently of each other, which results in a wide range of possible algorithms.

The cost function usually includes one or more terms that cause the learning process to perform statistical estimation. It may also include additional terms, such as a regularizer. If the model is nonlinear, most cost functions cannot be optimized in closed form, but demand the employment of an iterative, numerical optimization procedure, such as gradient descent.

2.2 Deep Feedforward Networks

Deep feedforward networks, or **feedforward neural networks**, are the prototypical deep learning models. The goal of a deep feedforward network is to approximate some function mapping $\boldsymbol{y} = f(\boldsymbol{x})$. It does so by defining an approximate mapping $\boldsymbol{y} \approx \hat{f}(\boldsymbol{x}; \boldsymbol{\pi})$ and learning the values of the parameter vector $\boldsymbol{\pi}$ that result in the best approximation of f.

Deep feedforward networks are called "feedforward" because information flows from the input $\boldsymbol{x}$, through the computational operations that define $\hat{f}$, before it reaches the output $\boldsymbol{y}$. There is no feedback loop that connects the output back to the model.

Deep feedforward networks are called "networks" because these models are typically constructed by composing together many different functions. For instance, $\hat{f}$ may be composed of N chained functions to form $\hat{f}(\boldsymbol{x}) = \left(\hat{f}^{(N)} \circ \hat{f}^{(N-1)} \circ \cdots \circ \hat{f}^{(1)}\right)(\boldsymbol{x})$. Then, $\hat{f}^{(1)}$ is called the first **layer**, $\hat{f}^{(2)}$ the second layer, and so on. The final layer of a feedforward network is called the **output** layer, and the length of the chain corresponds to the

depth of the network, hence the term **deep learning**.

Each training example $(\boldsymbol{x}, \boldsymbol{y})$ provides a noisy, approximate example of how f acts. A training example specifies that the output layer should produce a quantity close to $\boldsymbol{y}$ when $\boldsymbol{x}$ is fed into the model. However, the behavior of the intermediate layers is left unspecified by the training data, and the learning algorithm is left to self decide how to use those layers to produce the prescribed output. Since the training data does not show the desired output for any of these layers, these layers are called **hidden**.

Finally, these models are called "neural" because they draw some inspiration from neuroscience. Each hidden layer of the network may be seen as a set of **units** that operate in parallel, with each unit playing a role similar to that of a biological neuron, in the sense that it receives input from many other units and then computes its own activation value. However, the goal of neural network research is not to perfectly model the brain. It is perhaps best to view feedforward networks as function approximators built to achieve statistical generalization, that draw some insights from what is known about the brain.

2.2.1 Interpretation

Linear models are simple and can be fit efficiently, either in closed form or through convex optimization. However, their capacity is restricted to linear functions, which means that they cannot capture the interaction between any two input variables.

We can extend linear models to represent nonlinear functions of the input $\boldsymbol{x}$ by applying the linear model not to $\boldsymbol{x}$ itself, but to a transformed version $\phi(\boldsymbol{x})$ of $\boldsymbol{x}$. The nonlinear mapping ϕ may be thought of as providing a set of features describing $\boldsymbol{x}$, or giving a new **representation** of $\boldsymbol{x}$. There are different ways to choose ϕ. The strategy of deep learning is to learn it from examples. Using this strategy gives a model $\boldsymbol{y} = \hat{f}(\boldsymbol{x}; \boldsymbol{\pi}, \boldsymbol{w}) = \boldsymbol{w}^T \phi(\boldsymbol{x}; \boldsymbol{\pi})$, where the parameter vector $\boldsymbol{\pi}$ is used to learn ϕ, and the weights $\boldsymbol{w}$ map $\phi(\boldsymbol{x})$ to the desired output. This is an example of a feedforward network, with ϕ defining a hidden layer.

The nonlinear transformation used to describe the features is commonly formed as an affine transformation controlled by learned parameters, followed by a fixed, nonlinear **activation function**. That is to say, $\phi(\boldsymbol{x}) = a(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$, with weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$. The activation function a is usually applied element-wise. Today, the standard activation function is the **rectified linear unit**, or ReLU, defined as $a(z) = \max(z, 0)$. Although nonlinear, the rectified linear unit is close to linear in the sense that it consists of two linear pieces, and thus keeps some of the properties that make linear models easy to optimize with gradient-based methods, as well as some of the properties that make linear models generalize well. The fact that the rectified linear unit is non-differentiable at the origin may seem like a problem in the context of gradient-based learning. However, in practice, we do

not expect training to reach a point where the gradient is exactly zero, which reduces this fact to a technical detail.

What type of output unit to use depends on the choice of cost function. Any kind of unit used in the output layer can also be used in a hidden layer. An example of an output unit is the so-called **softmax** unit, which is commonly used as the output unit of a classification network, to represent the probability distribution over K different classes. The softmax function maps an input vector $\boldsymbol{z} \in \mathbb{R}^K$ into a probability vector that can be used to represent a categorical distribution. It is defined as

$$\operatorname{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}, \tag{2.1}$$

for $i \in \{1, 2, \ldots, K\}$. The softmax function can be seen as a generalization of the logistic function used to represent the probability distribution over a binary variable.

2.2.2 The Back-propagation Algorithm

In contrast to the linear equation solvers used to train linear regression models, or the convex optimization algorithms with global convergence guarantees used to train models such as logistic regression and support vector machines, neural networks are typically trained using iterative, gradient-based optimization schemes that merely drive the cost function to a low value. It is, however, perfectly viable to train also these other models with gradient descent. This is commonly done when the training set is very large. In this sense, training a neural network is not much different from training any other model. Computing the gradient is somewhat more involved, but can still be done efficiently and exactly.

The stage during which information flows through the network from input to output is called **forward propagation**. During training, forward propagation yields a scalar cost $C(\boldsymbol{\pi})$, or typically, an approximation thereof. The **back-propagation** algorithm (Rumelhart et al., 1986) enables information from the cost to then flow backwards through the network in order to compute the gradient.

The back-propagation algorithm refers only to the method of computing the gradient, whereas a separate optimization algorithm is responsible for using the gradient to perform learning. It is not restricted to neural networks, but can in principle be used to compute derivatives of any function for which these are defined. In the context of machine learning, the gradient we most commonly require is the gradient of the cost function with respect to the parameters, $\nabla_{\boldsymbol{\pi}} C(\boldsymbol{\pi})$.

In calculus, we compute derivatives of functions formed by composing other functions whose derivatives are known using the chain rule. Back-propagation is an algorithm that unfolds the chain rule in an efficient order.

Let $\boldsymbol{x} \in \mathbb{R}^N$, $\boldsymbol{y} \in \mathbb{R}^M$, $g : \mathbb{R}^N \to \mathbb{R}^M$, and $h : \mathbb{R}^M \to \mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = h(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{M} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}, \tag{2.2}$$

for $i \in \{1, 2, \ldots, N\}$. This is equivalent to writing

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z, \tag{2.3}$$

where $\partial \boldsymbol{y} / \partial \boldsymbol{x}$ is the $M \times N$ Jacobian matrix of g. Thus, the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\partial \boldsymbol{y} / \partial \boldsymbol{x}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propagation algorithm carries out such a Jacobian-gradient multiplication for each operation in the network. It is not restricted to work only on vectors, but is applicable to tensors of arbitrary dimensionality. Conceptually, this is the same thing, the only difference being how the numbers are organized.

It is straightforward to derive an algebraic expression for the gradient of a scalar with respect to any computational node in the graph representation of a network using the chain rule. Actually evaluating that expression on a computer requires some extra care. Specifically, many subexpressions may appear several times within the overall expression of the gradient, and so decisions about whether to store or recompute these subexpressions need be made. For complicated networks, the same subexpression can appear so many times that a naive implementation of the chain rule becomes infeasible. The back-propagation algorithm reduces the number of common subexpressions without regard to memory by storing intermediate result, a strategy called dynamic programming.

The back-propagation algorithm is just one approach to performing automatic differentiation. Other approaches evaluate the subexpressions of the chain rule in different orders. Therefore, the back-propagation algorithm is not the only way, let alone the optimal way of computing the gradient, but is a practical method that currently serves the deep learning community well.

2.2.3 Architecture Design

An important aspect when designing a deep learning system is the choice of **architecture**, which refers to the structure of the network. This includes how many units different layers should have, and how these layers should be connected to each other. The key architectural considerations for chain-based networks are the depth of the network and the width of each layer. Deep networks can be numerically difficult to optimize, but are typically able to generalize well to the test set using relatively few units per layer. There are few guidelines

when it comes to architecture design, and the ideal network architecture for a task must be found through experimentation.

2.3 Optimization

2.3.1 Empirical Risk Minimization

In the typical machine learning setting, we are interested in building a system that performs well according to some performance measure P. Sometimes, the very nature of P makes it impossible to optimize. In addition, P is evaluated with respect to the test set, which means that we can only target it indirectly. We do this by minimizing some other cost function C, and hope that doing so will improve performance with respect to P as well.

The cost function can usually be written as an average over the training set, such as

$$C(\boldsymbol{\pi}) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\hat{p}_{\text{data}}} L\Big(\hat{f}(\boldsymbol{x};\boldsymbol{\pi}), \boldsymbol{y}\Big) \tag{2.4}$$

$$= \frac{1}{N}\sum_{i=1}^{N} L\Big(\hat{f}\Big(\boldsymbol{x}^{(i)};\boldsymbol{\pi}\Big), \boldsymbol{y}^{(i)}\Big)\,, \tag{2.5}$$

where L is a per-example loss function, $\hat{f}(\boldsymbol{x};\boldsymbol{\pi})$ is the output predicted by the model when the input is $\boldsymbol{x}$, and $\hat{p}_{\text{data}}$ is the **empirical distribution**, which puts equal probability mass on each of the N training examples. In the supervised case, $\boldsymbol{y}$ is the target output. Ideally, we would prefer to minimize the expectation in (2.4) with respect to the true data generating distribution p_{data},

$$R(\boldsymbol{\pi}) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p_{\text{data}}} L\Big(\hat{f}(\boldsymbol{x};\boldsymbol{\pi}), \boldsymbol{y}\Big)\,, \tag{2.6}$$

which is nothing but the expected generalization error, also known as the **risk**. If we knew p_{data}, risk minimization would be a regular optimization problem. However, since we do not know p_{data}, but merely have a limited set of training examples, we face a machine learning problem. To convert this problem back into an optimization problem, we replace the true distribution p_{data} with the empirical distribution $\hat{p}_{\text{data}}$ and instead minimize the **empirical risk** defined by (2.4). Hopefully, minimizing the average training error will lead to a significant decrease in the true risk as well.

2.3.2 Mini-batch Algorithms

Gradient-based optimization algorithms used in machine learning compute parameter updates based on the gradient of the empirical risk with respect to the model parameters. Provided that the cost function decomposes as a sum over the training examples, this gra-

dient is given by

$$\nabla_{\boldsymbol{\pi}} C(\boldsymbol{\pi}) = \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim \hat{p}_{\text{data}}} \nabla_{\boldsymbol{\pi}} L\Big(\hat{f}(\boldsymbol{x}; \boldsymbol{\pi}), \boldsymbol{y}\Big) \tag{2.7}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\pi}} L\Big(\hat{f}\Big(\boldsymbol{x}^{(i)}; \boldsymbol{\pi}\Big), \boldsymbol{y}^{(i)}\Big) \,. \tag{2.8}$$

Computing this expectation exactly is likely to be very expensive, since it requires evaluating the model on each of the N training examples. Fortunately, we can approximate the expectation by using only a small, randomly sampled subset of the training examples.

Optimization algorithms that in each iteration compute the gradient based on the entire training set are traditionally called **batch** algorithms, whereas algorithms that do so by using only a single example at a time are called **stochastic**. Many optimization algorithms used for deep learning use more than one but less than all training examples to perform parameter updates. Such algorithms are referred to as **mini-batch** algorithms. An example of a commonly used mini-batch algorithm is **mini-batch gradient descent**.

What mini-batch size to use for a mini-batch algorithm depends on several factors. Larger mini-batches can estimate the gradient more accurately, but do so with less than linear returns. Smaller mini-batches can have a regularizing effect (Wilson and Martinez, 2003), maybe because of the noise they add to the learning process. The purely stochastic case often results in the lowest generalization error, but training with such a small mini-batch size may require taking very small steps in the direction of the negative gradient to maintain stability due to the high variance in the gradient estimate. This might result in slow convergence, not only due to the small steps themselves, but also because it takes more updates to observe the entire training set. First-order methods that base parameter updates only on the gradient are usually fairly robust and can handle small mini-batch sizes. Second-order methods that, in addition to the Jacobian also use the Hessian matrix, often require much larger mini-batch sizes to be stable.

Computing an unbiased estimate of the expected gradient from a subset of training examples requires that those examples be selected randomly. Moreover, since we want subsequent estimates of the gradient to be independent of each other, subsequent mini-batches should be independent of each other.

One motivation for mini-batch gradient descent is that is follows the gradient of the true risk in (2.6) if no examples are repeated. If both $\boldsymbol{x}$ and $\boldsymbol{y}$ are discrete, (2.6) can be expressed as a sum,

$$R(\boldsymbol{\pi}) = \sum_{\boldsymbol{x}} \sum_{\boldsymbol{y}} p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})\, L\Big(\hat{f}(\boldsymbol{x}; \boldsymbol{\pi}), \boldsymbol{y}\Big) \,, \tag{2.9}$$

having the exact gradient

$$\boldsymbol{g} = \nabla_{\boldsymbol{\pi}} R(\boldsymbol{\pi}) = \sum_{\boldsymbol{x}} \sum_{\boldsymbol{y}} p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y}) \nabla_{\boldsymbol{\pi}} L\Big(\hat{f}(\boldsymbol{x}; \boldsymbol{\pi}), \boldsymbol{y}\Big) . \tag{2.10}$$

A similar result holds for the continuous case. As a consequence of this, an unbiased estimator for the gradient of the risk can be obtained by randomly sampling a mini-batch of M examples $\big(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\big), \ldots, \big(\boldsymbol{x}^{(M)}, \boldsymbol{y}^{(M)}\big)$ from the data generating distribution p_{data} and computing the gradient of the average per-example loss with respect to the parameters given this mini-batch,

$$\hat{\boldsymbol{g}} = \frac{1}{M} \nabla_{\boldsymbol{\pi}} \sum_{i=1}^{M} L\Big(\hat{f}\Big(\boldsymbol{x}^{(i)}; \boldsymbol{\pi}\Big), \boldsymbol{y}^{(i)}\Big) . \tag{2.11}$$

Updating the parameter vector $\boldsymbol{\pi}$ in the negative direction of $\hat{\boldsymbol{g}}$ performs mini-batch gradient descent on the true risk.

This result only holds if no examples are reused. Unless the training set is very large, we usually shuffle it once and then make several passes through it. When making multiple such passes, or **epochs**, the estimator for the gradient of the risk becomes biased from the second pass and onwards. However, the reduced training error attained from using multiple epochs typically outweighs the harm they cause by widening the gap between training error and test error.

An important property of mini-batch gradient descent is that the amount of computation needed per update does not depend on the size of the training set, allowing for convergence no matter how many training examples there are.

Having computed an estimate of the gradient using (2.11), mini-batch gradient descent updates the parameter vector as $\boldsymbol{\pi} \leftarrow \boldsymbol{\pi} - \lambda \hat{\boldsymbol{g}}$. The hyperparameter λ is called the **learning rate**, and controls the size of the update step. The learning rate is crucial. Taking too large steps may cause the optimization to diverge, whereas taking too small steps could get us stuck in a poor local minimum. If we use the true gradient of the total cost function, then this becomes exactly $\mathbf{0}$ when we reach a local minimum, which means that we in this scenario can use a fixed learning rate. However, when using mini-batch gradient descent, the gradient estimate is noisy and will not reach exactly $\mathbf{0}$ even when we arrive at a local minimum. To this end, it is important to gradually decrease the learning rate over the course of the optimization process. There exist various policies for doing this. Examples include decreasing the learning rate linearly for some time and then leaving it constant, letting it decrease in a polynomial fashion, or reducing it by some factor whenever the cost seems to stop improving. However, picking the learning rate can be tricky, and most guidance on this subject should be taken with some skepticism.

An optimization algorithm used in a pure optimization scenario will proceed until a local minimum is reached. However, when used to train a machine learning algorithm, optimization usually halts when overfitting starts to occur. This means that training might stop while the cost function still has large derivatives, which is very different from the pure optimization setting, in which optimization ends when the derivatives become very small.

2.3.3 Momentum

Mini-batch gradient descent is a simple, yet popular optimization algorithm. However, learning with it is sometimes slow. The method of **momentum** (Polyak, 1964) is designed to speed up learning. It does so by accumulating previous gradients in an exponentially decaying moving average, and continuing to partially move in their direction.

The name momentum is due to a physical analogy, in which the negative gradient is a force that moves a particle through parameter space as governed by Newton's laws of motion. In physics, momentum is defined as mass times velocity. The momentum algorithm assumes unit mass, in which case momentum equals velocity. Formally, the algorithm introduces a momentum parameter $\boldsymbol{p}$ which acts as a force in the direction that the parameters currently move through parameter space. A decay rate parameter $\delta \in [0, 1)$ controls the decay of previous gradients. The larger the quotient δ/λ, the more impact earlier gradients have on the current direction. Algorithm 2.1 describes mini-batch gradient descent with momentum in more detail.

Algorithm 2.1: Mini-batch gradient descent with momentum. Note that the learning rate is typically set to decrease over the course of iterations.

Require: An initial parameter vector $\boldsymbol{\pi}$.
Require: A learning rate λ.
Require: An initial momentum $\boldsymbol{p}$.
Require: A decay rate parameter $\delta \in [0, 1)$.
while stopping criterion not met **do**
  Sample a mini-batch $\left(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\right), \ldots, \left(\boldsymbol{x}^{(M)}, \boldsymbol{y}^{(M)}\right)$ of M training examples.
  Estimate the gradient: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{M} \nabla_{\boldsymbol{\pi}} \sum_{i=1}^{M} L\Big(\hat{f}\left(\boldsymbol{x}^{(i)}; \boldsymbol{\pi}\right), \boldsymbol{y}^{(i)}\Big)$.
  Update the momentum: $\boldsymbol{p} \leftarrow \delta \boldsymbol{p} - \lambda \hat{\boldsymbol{g}}$.
  Update the parameter vector: $\boldsymbol{\pi} \leftarrow \boldsymbol{\pi} + \boldsymbol{p}$.
end while

2.3.4 Adaptive Learning Rates

Many times, the cost is highly sensitive to changes in some directions in parameters space but insensitive to changes in others. Momentum can help alleviate this problem to some extent, but does so at the expense of adding another hyperparameter. However, if the directions of sensitivity are somewhat axis-aligned, it may be a good idea to use a separate learning rate for each parameter, and automatically adapt these during training.

Adam (Kingma and Ba, 2014) is one of several optimization algorithms that adapt the learning rate of each model parameter individually. The name "Adam" is a portmanteau of the words "adaptive moments." Parameter updates are based on estimates of the first and second moments of the gradient, which are its mean and uncentered variance, respectively. The estimates are formed from exponentially decaying averages of past gradients and their squares. The moment vectors are initialized at the origin, which causes their entries to be biased toward zero. To account for these biases, the estimates are biases-corrected. The update step for each parameter is proportional to the first moment, scaled by the inverse of the second moment's square root. This causes parameters with a large partial derivative of the cost to have a quick decrease in their learning rate, whereas parameters with a small partial derivative of the cost have a comparatively slow decrease in their learning rate. The result of this is greater progress in the directions of parameter space that have low curvature. The Adam algorithm is described more formally in Algorithm 2.2.

2.3.5 Parameter Initialization

Iterative training algorithms for deep models require us to specify an initial point in parameter space from which to begin the optimization. Training deep models is a difficult task, and the initial point can determine whether the learning algorithm converges at all. When it converges, the initial point has an immediate impact on how quickly it does, and if the final solution corresponds to a point with low or high cost. Moreover, points with similar cost can have drastically different generalization error.

Because optimization of deep models is not well understood, initialization strategies are simple and heuristic. Possibly the only fact known with certainty is that units with the same activation function and inputs should have their parameters initialized differently in order to "break symmetry." Provided that the optimization algorithm, the cost, and the model are deterministic, these units would otherwise always be updated in the exact same way. A cheap way to circumvent this issue is to initialize the parameters randomly from a high-entropy distribution. This is unlikely to assign any two units the exact same parameters. However, biases are usually not initialized randomly, but rather set to some heuristically chosen constant, such as 0. Sometimes we may want to choose the bias to avoid causing

Algorithm 2.2: Adam. The symbol $\odot$ denotes the Hadamard product, and the formula for updating the parameter vector $\boldsymbol{\pi}$ should be read element-wise.

Require: An initial parameter vector $\boldsymbol{\pi}$.
Require: A learning rate λ.
Require: Exponential decay rates $\delta_1 \in [0, 1)$ and $\delta_2 \in [0, 1)$ for the moment estimates.
Require: A small constant ϵ for numerical stability.
Initialize the first-order and second-order moment variables: $\boldsymbol{m}_1 = \mathbf{0}$ and $\boldsymbol{m}_2 = \mathbf{0}$.
Initialize the time step: $t = 0$.
while stopping criterion not met **do**
Sample a mini-batch $\big(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\big), \ldots, \big(\boldsymbol{x}^{(M)}, \boldsymbol{y}^{(M)}\big)$ of M training examples.
Estimate the gradient: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{M} \nabla_{\boldsymbol{\pi}} \sum_{i=1}^{M} L\Big(\hat{f}\big(\boldsymbol{x}^{(i)}; \boldsymbol{\pi}\big), \boldsymbol{y}^{(i)}\Big)$.
Increment the time step: $t \leftarrow t + 1$.
Update the estimate of the first moment: $\boldsymbol{m}_1 \leftarrow \delta_1 \boldsymbol{m}_1 + (1 - \delta_1)\boldsymbol{g}$.
Update the estimate of the second moment: $\boldsymbol{m}_2 \leftarrow \delta_2 \boldsymbol{m}_1 + (1 - \delta_2)\boldsymbol{g} \odot \boldsymbol{g}$.
Correct for the bias in the first moment: $\boldsymbol{m}_1 \leftarrow \boldsymbol{m}_1 / \big(1 - \delta_1^t\big)$.
Correct for the bias in the second moment: $\boldsymbol{m}_2 \leftarrow \boldsymbol{m}_2 / \big(1 - \delta_2^t\big)$.
Update the parameter vector: $\boldsymbol{\pi} \leftarrow \boldsymbol{\pi} - \lambda \boldsymbol{m}_1 / \big(\sqrt{\boldsymbol{m}_2} + \epsilon\big)$.
end while

too much saturation at initialization. For example, we may set the bias of a hidden unit that uses a rectifier to 0.1 rather than 0 to avoid saturating the ReLU at initialization.

2.4 Convolutional Networks

Convolutional networks, or **convolutional neural networks**, are a specialized kind of neural network that make use of the convolution operator. This makes them suitable for processing data that has a grid-like topology, such as images. Convolutional networks have been highly successful in practical applications. Any network that uses convolution instead of general matrix multiplication in one or more of its layers is a convolutional network.

2.4.1 The Convolution Operator

The convolution between two functions g and k is defined as

$$(g * k)(t) \;=\; \int_{\mathbb{R}} g(\tau)\, k(t - \tau)\, \mathrm{d}\tau. \tag{2.12}$$

In the terminology of convolutional networks, g is called the **input**, k the **kernel**, and the output is commonly referred to as the **feature map**. The discrete counterpart of (2.12) is

defined as

$$(g * k)(t) = \sum_{i=-\infty}^{\infty} g(\tau)\, k(t-i)\,. \tag{2.13}$$

In deep learning applications, the input and kernel are typically multidimensional arrays of data and learnable parameters, respectively. To be able to evaluate an infinite summation on a computer, we assume that g and k are zero everywhere but the finite number of points for which we store their values.

If the input is a two-dimensional image $\boldsymbol{I}$, we typically also want a two-dimensional kernel $\boldsymbol{K}$, in which case

$$(\boldsymbol{I} * \boldsymbol{K})(h, w) = \sum_i \sum_j \boldsymbol{I}(i, j)\, \boldsymbol{K}(h-i, w-j)\,. \tag{2.14}$$

Convolution is commutative, which means that

$$(\boldsymbol{I} * \boldsymbol{K})(h, w) = (\boldsymbol{K} * \boldsymbol{I})(h, w) = \sum_i \sum_j \boldsymbol{I}(h-i, w-j)\, \boldsymbol{K}(i, j)\,. \tag{2.15}$$

The commutative property arises because the kernel has been flipped relative to the input. It is usually not an important property of a neural network implementation. Many software libraries instead implement the related cross-correlation function, which is the same as convolution but without kernel flipping, and call it convolution:

$$(\boldsymbol{I} * \boldsymbol{K})(h, w) = \sum_i \sum_j \boldsymbol{I}(h+i, w+j)\, \boldsymbol{K}(i, j)\,. \tag{2.16}$$

This will cause a learning algorithm to learn kernels that are flipped relative to the kernels it would learn if true convolution was used.

Convolution is a linear operation. This implies that discrete convolution may be viewed as multiplication with a matrix that is constrained to conform to a certain structure. For instance, a one-dimensional kernel can be represented as a Toeplitz matrix, whereas a two-dimensional kernel can be represented as a doubly block circulant matrix. The kernel is typically much smaller than the input, which makes these matrices very sparse.

2.4.2 Motivation

A traditional neural network layer uses matrix multiplication with each element describing the interaction between one input unit and one output unit, which means that every input unit interacts with every output unit. In contrast, convolutional network kernels that are smaller than the input have **sparse interactions**. For example, a kernel used to detect some feature in an image, such as an edge, can do so using only tens or hundreds of pixels,

even if the image itself consists of thousands or millions of pixels. With multiple convolutional layers, units in the deeper layers may then indirectly interact with a larger portion of the input image, allowing the network to capture complicated interactions between many variables using simple building blocks, each of which interacts only sparsely.

Furthermore, features that can appear anywhere in the image allow us to learn only one set of parameters instead of a separate set of parameters for every input location. This means that each member of a kernel is used at every position of the input, except possibly at some boundary pixels, depending on how we decide to treat the boundary cases. Such **parameter sharing** does not apply to traditional, **fully connected** layers in which each parameter is used only once. As a result, convolutional layers allow us to construct deep networks using relatively few parameters. This reduces the risk of overfitting. The particular form of parameter sharing employed by a convolutional layer makes it invariant to translation. In the case of images, if an object in an image is translated, the representation of that object in the feature map will move accordingly.

2.4.3 Pooling

It is common to arrange a convolutional layer to consist of two or three stages. The first stage performs convolution with many different kernels in parallel to output a linear set of activations. These linear activations are then passed through a nonlinear activation function such as the rectified linear unit. This second stage is sometimes referred to as the **detector** stage. An optional third stage that applies a **pooling function** may be used to modify the layer output even further.

A pooling function replaces the output of a layer at a particular position with a summary statistic computed from nearby outputs. Examples of popular pooling functions include **max pooling** and **average pooling**, which compute the maximum and average output value within some rectangular neighborhood, respectively.

Pooling helps make the representation become roughly invariant to small translations of the input. Such invariance can be useful in cases where we care more about the potential presence of some feature, and not so much about its exact location.

2.4.4 Practicalities

A convolutional layer typically performs convolution with many different kernels in parallel. This is because a single kernel can only extract one specific feature, and we usually want each layer to extract many different features. When working with images, the first convolutional layer of a convolutional network might detect edges of various orientation,

while layers of greater depth learn to combine features detected by preceding layers into features of increasingly sophisticated nature.

Moreover, each location of the input is in general vector-valued. For example, an image might have a red, a green, and a blue intensity at each pixel location. In this setting, we usually think of the input and output of the convolution as being three-dimensional tensors, with one axis indexing into the different **channels**, and the remaining two axes indexing into each channel's spatial coordinates. What is more, software implementations commonly operate in mini-batch mode, and thus add a fourth axis that corresponds to the different examples in the mini-batch.

Unless the spatial dimensions of the input to a convolutional layer are padded, the height and width of the output will shrink by one pixel less than the kernel height and width, respectively. The ability to implicitly pad the input is an essential feature of any implementation. Zero-padding is the most common way of expanding the input, but several other padding strategies, such as reflection padding, are sensible. If we actually want to reduce the spatial extent of the feature maps, then a more controlled way of doing so is to modify the so-called **stride** used by convolutional and pooling layers. The stride defines how much a kernel or pooling window is shifted in between adjacent computations, and thus offers a means to perform sub-sampling of the input. For example, using a stride of two along each spatial coordinate will halve the spatial extent of the output in each direction. This can be an effective way to reduce the amount of computation needed to propagate data through a network.

As a final note, convolutional networks can be used to output a high-dimensional, structured object instead of just a class label for a classification task or a real value for a regression task. This object is usually just a tensor T emitted by a convolutional layer, where entry $\mathsf{T}_{k,i,j}$ could, for example, indicate the likelihood of pixel (i, j) belonging to class k. Such a structured output allows us to classify each pixel in the image and draw detailed boundaries around individual objects.

Chapter 3

Deep Normalized Cuts

3.1 Normalized Cuts

Clustering refers to the task of grouping a set of objects in such a way that objects that are grouped together to form a cluster are more similar to each other than to objects that are grouped together to form other clusters, as measured by some notion of similarity. **Spectral clustering** is a collective term for clustering techniques that are based on the eigenstructure of a **similarity matrix**, comprised of pairwise similarity relations between data points. In the case of image segmentation, data points correspond to features of image pixels.

Assume we have an image consisting of N pixels. Let $\boldsymbol{S}$ be a symmetric $N \times N$ similarity matrix such that elements $S_{ij} = S_{ji} \geq 0$ are a measure of the similarity between pixels i and j, with a large value suggesting that these pixels should belong to the same cluster, and a small value suggesting that they should not. The goal of spectral clustering is to organize the set of pixels into disjoint subsets, such that the similarity within subsets is high, while the similarity across subsets is low.

Let $\mathcal{P} = \{1, 2, \ldots, N\}$ be an index set of the set of pixels in the image. We wish to partition $\mathcal{P}$ into K disjoint, non-empty subsets $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_K$ according to a particular loss function. The loss function we will consider is the K-way **normalized cut** (Shi and Malik, 2000; Gu et al., 2001), defined as

$$\mathrm{ncut}(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_K) = \sum_{k=1}^{K} \frac{\mathrm{cut}(\mathcal{P}_k, \mathcal{P} \backslash \mathcal{P}_k)}{\mathrm{cut}(\mathcal{P}_k, \mathcal{P})}, \tag{3.1}$$

with $\mathrm{cut}(\mathcal{P}_k, \mathcal{P} \backslash \mathcal{P}_k) = \sum_{i \in \mathcal{P}_k} \sum_{j \in \mathcal{P} \backslash \mathcal{P}_k} S_{ij}$, and similarly for $\mathrm{cut}(\mathcal{P}_k, \mathcal{P})$. Because of the normalizing denominators in (3.1), the K-way normalized cut criterion penalizes unbalanced solutions. This is not the case in the non-normalized setting, which often leads to some clusters containing only a single pixel. Minimizing the inter-cluster similarity

through (3.1) is equivalent to maximizing the intra-cluster similarity (Shi and Malik, 2000). Hence, both of these criteria can be achieved simultaneously.

Let $\boldsymbol{E} \in \{0,1\}^{N\times K}$ be an indicator matrix such that E_{ij} equals 1 if pixel i belongs to part j, and 0 otherwise. Since full knowledge of $\boldsymbol{E}$ entails full knowledge of $(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_K)$, and vice versa, we will refer to $\boldsymbol{E}$ as a partition. Let $\boldsymbol{D} = \mathrm{diag}(\boldsymbol{S}\boldsymbol{1})$ be the diagonal matrix for which entry D_{ii} is the sum of the i^{th} row of $\boldsymbol{S}$. A simple calculation (Gu et al., 2001) reveals that

$$\mathrm{ncut}(\boldsymbol{E}) = \sum_{k=1}^{K} \frac{\boldsymbol{E}_k^T(\boldsymbol{D}-\boldsymbol{S})\boldsymbol{E}_k}{\boldsymbol{E}_k^T\boldsymbol{D}\boldsymbol{E}_k}, \tag{3.2}$$

where $\boldsymbol{E}_k$ denotes the k^{th} column of $\boldsymbol{E}$.

To find a matrix $\boldsymbol{E}$ that minimizes the normalized cut is an NP-hard problem (Shi and Malik, 2000; Meila and Xu, 2003). However, if we embed the problem in the domain of real values, an approximate, discrete solution based on eigenvalue decomposition can be found efficiently.

Bach and Jordan (2006) show that the K-way normalized cut loss $\mathrm{ncut}(\boldsymbol{E})$ is equal to $K - \mathrm{trace}\big(\boldsymbol{Q}^T\boldsymbol{D}^{-1/2}\boldsymbol{S}\boldsymbol{D}^{-1/2}\boldsymbol{Q}\big)$, for any matrix $\boldsymbol{Q} \in \mathbb{R}^{N\times K}$ that (i) has orthonormal columns and (ii) is such that the columns of $\boldsymbol{D}^{-1/2}\boldsymbol{Q}$ are proportional to the corresponding columns of $\boldsymbol{E}$. Removing the latter constraint yields a tractable relaxation whose solutions involve the eigenstructure of $\tilde{\boldsymbol{S}} = \boldsymbol{D}^{-1/2}\boldsymbol{S}\boldsymbol{D}^{-1/2}$. Consider the eigendecomposition $\tilde{\boldsymbol{S}} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T$, where the eigenvalues on the diagonal of $\boldsymbol{\Lambda}$ are ordered in descending order. It follows from Ky-Fan's theorem (Overton and Womersley, 1993) that the maximum of $\mathrm{trace}\big(\boldsymbol{Q}^T\tilde{\boldsymbol{S}}\boldsymbol{Q}\big)$ is equal to the sum of the K largest eigenvalues of $\tilde{\boldsymbol{S}}$, and that the maximum is attained for all $\boldsymbol{Q}$ of the form $\boldsymbol{Q} = \boldsymbol{U}_{1:K}\boldsymbol{A}$, where $\boldsymbol{U}_{1:K} \in \mathbb{R}^{N\times K}$ is any orthonormal basis of the K-dimensional principal subspace of $\tilde{\boldsymbol{S}}$, or, in other words, the columns of $\boldsymbol{U}_{1:K}$ are the eigenvectors associated with the K largest eigenvalues of $\tilde{\boldsymbol{S}}$, and $\boldsymbol{A}$ is any orthogonal matrix in $\mathbb{R}^{K\times K}$. Naturally, the solution found via this relaxation will in general not fulfill criterion (ii), which means that it must be projected back to the set of allowed solutions.

The set of matrices $\boldsymbol{R}$ that correspond to a partition defined by $\boldsymbol{E}$ and satisfy constraints (i) and (ii) are of the form $\boldsymbol{R} = \boldsymbol{D}^{1/2}\boldsymbol{E}\big(\boldsymbol{E}^T\boldsymbol{D}\boldsymbol{E}\big)^{-1/2}\boldsymbol{B}$, where $\boldsymbol{B} \in \mathbb{R}^{K\times K}$ is an arbitrary orthogonal matrix (Bach and Jordan, 2006).

Since both $\boldsymbol{Q}$ and $\boldsymbol{R}$ are defined up to an orthogonal matrix, it makes sense to try to align the subspace spanned by the columns of $\boldsymbol{Q}$ with the subspace spanned by the columns of $\boldsymbol{R}$. A common way to compute the distance between subspaces is to compare the Frobenius norm between the orthogonal projection operators onto those subspaces (Golub and Van Loan, 2012). Note that such a metric is invariant to permutations of the cluster indices.

The orthogonal projection operators onto the subspaces spanned by the columns of $\boldsymbol{Q}$ and $\boldsymbol{R}$ are

$$\boldsymbol{\Pi}_{\boldsymbol{Q}} = \boldsymbol{Q}\boldsymbol{Q}^T = \boldsymbol{U}_{1:K}\boldsymbol{U}_{1:K}^T \tag{3.3}$$

and

$$\boldsymbol{\Pi}_{\boldsymbol{R}} = \boldsymbol{R}\boldsymbol{R}^T = \boldsymbol{D}^{1/2}\boldsymbol{E}\big(\boldsymbol{E}^T\boldsymbol{D}\boldsymbol{E}\big)^{-1}\boldsymbol{E}^T\boldsymbol{D}^{1/2}, \tag{3.4}$$

respectively. Measuring the Frobenius norm between these projectors gives rise to the loss function

$$L_1(\boldsymbol{S}, \boldsymbol{E}) = \frac{1}{2}\left\|\boldsymbol{\Pi}_{\boldsymbol{Q}} - \boldsymbol{\Pi}_{\boldsymbol{R}}\right\|_F^2 \tag{3.5}$$

$$= \frac{1}{2}\left\|\boldsymbol{U}_{1:K}\boldsymbol{U}_{1:K}^T - \boldsymbol{D}^{1/2}\boldsymbol{E}\big(\boldsymbol{E}^T\boldsymbol{D}\boldsymbol{E}\big)^{-1}\boldsymbol{E}^T\boldsymbol{D}^{1/2}\right\|_F^2. \tag{3.6}$$

Bach and Jordan (2006) note that if the rank of the similarity matrix $\boldsymbol{S}$ is equal to K, then L_1 is equal to the K-way normalized cut, $\mathrm{ncut}(\boldsymbol{E})$.

By construction, $\boldsymbol{U}_{1:K}^T\boldsymbol{U}_{1:K} = \boldsymbol{I}$. Additional normalization of the rows of $\boldsymbol{U}_{1:K}$ has proved beneficial in clustering applications (Ng et al., 2001). This can be seen by considering the ideal setting, in which the similarity is strictly positive for pixels that should belong to the same cluster, and zero otherwise. Then, the eigenvalue 1 of the matrix $\tilde{\boldsymbol{S}}$ has multiplicity K, and $\boldsymbol{D}^{1/2}\boldsymbol{E}$ is an orthonormal basis of the K-dimensional principal subspace of $\tilde{\boldsymbol{S}}$ (Bach and Jordan, 2006). Consequently, any basis $\boldsymbol{U}_{1:K}$ of this subspace has rows that are located on rays in $\mathbb{R}^K$, with the distance from the i^{th} row of $\boldsymbol{U}_{1:K}$ to the origin being $\sqrt{D_{ii}}$. By dividing each row of $\boldsymbol{U}_{1:K}$ with its distance to the origin, the rays collapse to points in $\mathbb{R}^K$, and thus become trivial to cluster. Whereas this holds only in the ideal setting, Ng et al. (2001) show that when the similarity matrix is close to ideal, the properly normalized rows tightly cluster around an orthonormal basis.

Motivated by these observations, Bach and Jordan (2006) also introduce an alternative loss function. By multiplying $\boldsymbol{U}_{1:K}$ by $\boldsymbol{D}^{-1/2}$ and re-orthogonalizing, they obtain the matrix $\boldsymbol{V} = \boldsymbol{D}^{1/2}\boldsymbol{U}_{1:K}\big(\boldsymbol{U}_{1:K}^T\boldsymbol{D}\boldsymbol{U}_{1:K}\big)^{-1/2}$, where any matrix square root of $\boldsymbol{U}_{1:K}^T\boldsymbol{D}\boldsymbol{U}_{1:K}$ is valid. The modified loss function then reads

$$L_2(\boldsymbol{S}, \boldsymbol{E}) = \frac{1}{2}\left\|\boldsymbol{\Pi}_{\boldsymbol{V}} - \boldsymbol{\Pi}_{\boldsymbol{E}}\right\|_F^2 = \frac{1}{2}\left\|\boldsymbol{V}\boldsymbol{V}^T - \boldsymbol{E}\big(\boldsymbol{E}^T\boldsymbol{E}\big)^{-1}\boldsymbol{E}^T\right\|_F^2. \tag{3.7}$$

Both L_1 and L_2 measure the ability of the similarity matrix $\boldsymbol{S}$ to produce the partition $\boldsymbol{E}$ when using the eigenvectors $\boldsymbol{U}_{1:K}$.

3.2 Learning the Similarity Matrix

Bach and Jordan (2006) assume each data point is described by a set of features, and that the similarity matrix is a function of these features and a parameter vector $\boldsymbol{\sigma}$. For instance,

they consider diagonally-scaled Gaussian kernel matrices, for which each element of $\boldsymbol{\sigma}$ is the scale of the corresponding dimension. However, they assume the features are given on beforehand, and only seek to learn $\boldsymbol{\sigma}$.

Ionescu et al. (2015) instead assume $\boldsymbol{S} = \boldsymbol{F}\boldsymbol{\Sigma}\boldsymbol{F}^T$, where $\boldsymbol{F}$ is a feature matrix and $\boldsymbol{\Sigma}$ is a parameter matrix, and aim to learn $\boldsymbol{F}$ and $\boldsymbol{\Sigma}$ using convolutional networks. They do so by modifying the loss functions L_1 and L_2 and deriving the matrix derivatives needed to enable learning via back-propagation.

Importantly, Ionescu et al. (2015) do not truncate the spectrum during training, but through the use of projectors implicitly consider the entire eigenspace of $\tilde{\boldsymbol{S}}$. For example, the term $\boldsymbol{V}\boldsymbol{V}^T = \boldsymbol{D}^{1/2}\boldsymbol{U}_{1:K}\left(\boldsymbol{U}_{1:K}^T\boldsymbol{D}\boldsymbol{U}_{1:K}\right)^{-1}\boldsymbol{U}_{1:K}^T\boldsymbol{D}^{1/2}$ in (3.7) is replaced with $\boldsymbol{D}^{1/2}\boldsymbol{U}\left(\boldsymbol{U}^T\boldsymbol{D}\boldsymbol{U}\right)^{-1}\boldsymbol{U}^T\boldsymbol{D}^{1/2}$. Recalling that a projector $\boldsymbol{\Pi}_{\boldsymbol{M}}$ of a matrix $\boldsymbol{M}$ is idempotent and leaves $\boldsymbol{M}$ unchanged, they note that

$$\left(\boldsymbol{D}^{1/2}\boldsymbol{U}\left(\boldsymbol{U}^T\boldsymbol{D}\boldsymbol{U}\right)^{-1}\boldsymbol{U}^T\boldsymbol{D}^{1/2}\right)^2 = \boldsymbol{D}^{1/2}\boldsymbol{U}\left(\boldsymbol{U}^T\boldsymbol{D}\boldsymbol{U}\right)^{-1}\boldsymbol{U}^T\boldsymbol{D}^{1/2} \tag{3.8}$$

and

$$\begin{aligned}
\boldsymbol{D}^{1/2}\boldsymbol{U}\left(\boldsymbol{U}^T\boldsymbol{D}\boldsymbol{U}\right)^{-1}\boldsymbol{U}^T\boldsymbol{D}^{1/2}\boldsymbol{S} &= \boldsymbol{D}^{1/2}\boldsymbol{U}\left(\boldsymbol{U}^T\boldsymbol{D}\boldsymbol{U}\right)^{-1}\boldsymbol{U}^T\boldsymbol{D}^{1/2}\boldsymbol{D}^{1/2}\tilde{\boldsymbol{S}}\boldsymbol{D}^{1/2} && (3.9)\\
&= \boldsymbol{D}^{1/2}\boldsymbol{U}\left(\boldsymbol{U}^T\boldsymbol{D}\boldsymbol{U}\right)^{-1}\boldsymbol{U}^T\boldsymbol{D}\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T\boldsymbol{D}^{1/2} && (3.10)\\
&= \boldsymbol{D}^{1/2}\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T\boldsymbol{D}^{1/2} && (3.11)\\
&= \boldsymbol{D}^{1/2}\tilde{\boldsymbol{S}}\boldsymbol{D}^{1/2} && (3.12)\\
&= \boldsymbol{S}, && (3.13)
\end{aligned}$$

which implies that $\boldsymbol{D}^{1/2}\boldsymbol{U}\left(\boldsymbol{U}^T\boldsymbol{D}\boldsymbol{U}\right)^{-1}\boldsymbol{U}^T\boldsymbol{D}^{1/2}$ is a projector for $\boldsymbol{S}$. This leads to the following, modified version of L_2:

$$L_3(\boldsymbol{S}, \boldsymbol{E}) = \frac{1}{2}\left\|\boldsymbol{\Pi}_{\boldsymbol{S}} - \boldsymbol{\Pi}_{\boldsymbol{E}}\right\|_F^2 = \frac{1}{2}\left\|\boldsymbol{S}\boldsymbol{S}^+ - \boldsymbol{E}\left(\boldsymbol{E}^T\boldsymbol{E}\right)^{-1}\boldsymbol{E}^T\right\|_F^2, \tag{3.14}$$

where $\boldsymbol{S}^+$ denotes the Moore-Penrose pseudoinverse of $\boldsymbol{S}$. Also L_1 is modified, but as we only use L_3 in our learning implementation, we omit any further contemplation of L_1 here.

Ionescu et al. (2015) put considerable effort into the derivation of the matrix derivatives needed to perform learning. This requires introducing some theoretical machinery. Here, we only present the derivatives needed to back-propagate errors from L_3. Unfortunately, $\partial L_3/\partial\boldsymbol{\Sigma} = \boldsymbol{0}$, so we cannot learn $\boldsymbol{\Sigma}$ using the above projector trick. We instead let $\boldsymbol{\Sigma} = \boldsymbol{I}$, which gives $\boldsymbol{S} = \boldsymbol{F}\boldsymbol{F}^T$. Assume the final layers are composed as $\boldsymbol{F} \to \boldsymbol{S} \to L_3 \leftarrow \boldsymbol{E}$. Since $\boldsymbol{E}$ contains no parameters, $\partial L_3/\partial\boldsymbol{E} = \boldsymbol{0}$. Ionescu et al. (2015) show that

$$\frac{\partial L_3}{\partial \boldsymbol{S}} = 2(\boldsymbol{\Pi}_{\boldsymbol{S}} - \boldsymbol{I})\,\boldsymbol{\Pi}_{\boldsymbol{E}}\boldsymbol{S}^+ \tag{3.15}$$

and

$$\frac{\partial L_3}{\partial \boldsymbol{F}} = \left(\frac{\partial L_3}{\partial \boldsymbol{S}} + \left(\frac{\partial L_3}{\partial \boldsymbol{S}} \right)^T \right) \boldsymbol{F}. \tag{3.16}$$

Using these expressions, the gradients needed for optimization can be propagated backwards to $\boldsymbol{F}$ as $\partial L_3 / \partial \boldsymbol{F} \leftarrow \partial L_3 / \partial \boldsymbol{S} \leftarrow L_3$.

3.3 Inference

Upon completion of the training stage, we want to compute actual partitions of images based on the learned features. Given an image, $\boldsymbol{S}$ is now fixed, and we should minimize L_2 with respect to $\boldsymbol{E}$ (note that L_3 is only used during training). Bach and Jordan (2006) show that for any partition $\boldsymbol{E}$, the minimum of

$$\sum_{i=1}^{N} \sum_{j=1}^{K} E_{ij} \left\| \boldsymbol{V}_{i,:}^T - \boldsymbol{\mu}^{(j)} \right\|_2^2 \tag{3.17}$$

with respect to centroids $\left(\boldsymbol{\mu}^{(1)}, \boldsymbol{\mu}^{(2)}, \ldots, \boldsymbol{\mu}^{(K)}\right) \in \mathbb{R}^{K \times K}$ is equal to L_2. This naturally leads to using a K-means algorithm for inferring a partition based on the computed eigenvectors, presented in Algorithm 3.1.

Algorithm 3.1: A spectral clustering algorithm for minimizing L_2 with respect to $\boldsymbol{E}$.

Require: A similarity matrix $\boldsymbol{S} \in \mathbb{R}^{N \times N}$.
Require: The number of clusters K.
Let $\tilde{\boldsymbol{S}} = \boldsymbol{D}^{-1/2} \boldsymbol{S} \boldsymbol{D}^{-1/2}$, where $\boldsymbol{D} = \operatorname{diag}(\boldsymbol{S}\boldsymbol{1})$.
Compute the eigenvectors $\boldsymbol{U}_{1:K}$ corresponding to the K largest eigenvalues of $\tilde{\boldsymbol{S}}$.
Compute $\boldsymbol{V} = \boldsymbol{D}^{1/2} \boldsymbol{U}_{1:K} \left(\boldsymbol{U}_{1:K}^T \boldsymbol{D} \boldsymbol{U}_{1:K} \right)^{-1/2}$ using any square root of $\boldsymbol{U}_{1:K}^T \boldsymbol{D} \boldsymbol{U}_{1:K}$.
Initialize $\boldsymbol{E}$.
while $\boldsymbol{E}$ is not stationary **do**
  for $j = 1, 2, \ldots, K$ **do**
    Form centroid j: $\boldsymbol{\mu}^{(j)} \leftarrow \frac{1}{\boldsymbol{1}^T \boldsymbol{E}_j} \sum_{i=1}^{N} E_{ij} \boldsymbol{V}_{i,:}^T$.
  end for
  for $i = 1, 2, \ldots, N$ **do**
    Determine which centroid is closest to pixel i: $k \leftarrow \arg\min_j \left\| \boldsymbol{V}_{i,:}^T - \boldsymbol{\mu}^{(j)} \right\|_2$.
    If $E_{ik} = 0$, update row $\boldsymbol{E}_{i,:}$ so that pixel i belongs to part k.
  end for
end while

Chapter 4

Experimental Evaluation

4.1 Cityscapes

Cityscapes (Cordts et al., 2016) is a large-scale dataset and benchmark suite for training and testing methods for pixel-level and instance-level semantic segmentation, specifically tailored for autonomous driving in an urban environment. The dataset contains images of highly complex inner-city street scenes obtained from 50 different European cities, most of which are located in Germany.

The dataset defines 30 visual classes for pixel-level segmentation, grouped into eight categories: *vehicle*, *human*, *construction*, *object*, *nature*, *sky*, *flat*, and *void*. Some of the classes are excluded from the benchmark, leaving 19 classes for pixel-level evaluation. Out of these, the classes *car*, *truck*, *bus*, *on rails*, *motorcycle*, *bicycle*, *human*, and *rider* are included also in the instance-level benchmark.

A total of $5{,}000$ images from 27 of the 50 cities have been finely annotated. These images are split into $2{,}975$ images for training, 500 images for validation, and $1{,}525$ images for testing. No two splits contain images from the same city. Ground truth annotations for the training and validation images are publicly available, but segmentations predicted for the test images need be submitted to a test server for evaluation. In addition, $20{,}000$ images from the remaining 23 cities have been provided with coarse, polygonal annotations. These images are primarily intended for methods that leverage large volumes of weakly annotated data. All images are of resolution 1024×2048 pixels (height $\times$ width) and have a color depth of 8 bits. Figure 4.1 includes a raw image and its (fine) pixel-level annotation.

To assess pixel-level segmentation performance, two metrics are used. One of these is the standard **Jaccard index**, also known as the **intersection over union** metric, which is defined as $\mathrm{IoU} = \mathrm{TP}/(\mathrm{TP} + \mathrm{FP} + \mathrm{FN})$. Here, TP, FP, and FN denote the number of **true positive**, **false positive**, and **false negative** pixels, respectively. As an example, given

Figure 4.1: An image from the Cityscapes dataset (left), its ground truth pixel-level annotation (center), and the pixel-level annotation predicted by the Dilation10 network (right). Pixels in the ground truth annotation that are colored in black correspond to classes that are excluded from evaluation by the pixel-level benchmark. The Dilation10 network only considers classes that are evaluated. Due to restrictions in terms of hardware, the prediction above was obtained by splitting the input image into 16 tiles of dimension 256×512 pixels, computing a segmentation for each tile separately, and then stitching the individual predictions back together. This compromise might introduce some boundary artifacts.

an image's ground truth annotation, a predicted annotation, and a semantic class, say *car*, TP is the number of pixels for which both annotations report *car*, FP is the number of pixels for which the predicted annotation reports *car* while the ground truth annotation reports some other class, and FN is the number of pixels for which the ground truth annotation reports *car* but the predicted annotation does not. The final score is obtained by averaging across all test images and all 19 semantic classes included in the benchmark. Unfortunately, this global IoU measure is biased towards object instances that cover a large image area. This is problematic for street scenes that often exhibit large variation in terms of object scale. The second metric being used is designed to better capture how well individual instances are represented in predicted pixel-level segmentations.

More important to us is the evaluation metric used for instance-level segmentation. Algorithms that attempt to solve this task are required to provide a binary segmentation mask, a class label, and a confidence score for each predicted instance. A predicted instance of a certain class is considered to match a ground truth instance of the same class if the percentage of overlapping pixels between the pair of segments is greater than some threshold value θ. If two or more predictions match the same ground truth instance, the one with the highest confidence score is kept while the others are treated as false positives. However, a single instance prediction cannot match more than one ground truth instance (in fact, this is not possible if $\theta \geq 50\%$). For a given class and a fixed threshold θ, each test image is associated with two values: **precision**, defined as $\mathrm{P} = \mathrm{TP}/(\mathrm{TP}+\mathrm{FP})$, and **recall**, defined as $\mathrm{R} = \mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$. Note that here, TP, FP, and FN are computed on the level of instances, not pixels. Plotting each such pair of values against each other yields a **precision-recall curve**. Computing the area under this curve gives the **average precision**,

$\mathrm{AP}^{\theta}_{class}$. Following Lin et al. (2014), this is done for threshold values θ ranging from 50% to 95%, in increments of 5%, to avoid a bias towards a specific value. Taking the average over these ten values gives the final score for the class in question, AP_{class}. Additionally computing the mean across the different classes gives the **mean average precision**, mAP, which is used by the instance-level benchmark as the ultimate measure of performance. In addition, $\mathrm{mAP}^{50\%}$ serves as a minor score.

4.2 Implementation

We chose to use a pixel-level semantic segmentation network called **Dilation10** (Yu and Koltun, 2016) as backbone for our method. This network had already been trained on the Cityscapes dataset, with a global IoU score of 67.1%. Figure 4.1 includes an example of a segmentation produced by the Dilation10 network.

The Dilation10 network consists of two convolutional modules. First, a "front-end" module made up from 16 convolutional layers computes 19 feature maps from an input image. The front-end module is an adaptation of the VGG16 classification network (Simonyan and Zisserman, 2014) for dense prediction. Then, a "context" module further processes the output of the front-end module to produce 19 semantic heat maps that form the basis for pixel-level segmentation. The context module comprises ten layers of **dilated** convolutions. To see what these are, rewrite (2.14) as

$$(\boldsymbol{I} * \boldsymbol{K})(h, w) = \sum_{i+k=h} \sum_{j+l=w} \boldsymbol{I}(i, j)\, \boldsymbol{K}(k, l)\,. \tag{4.1}$$

Convolution with a dilation factor d generalizes (4.1) to

$$(\boldsymbol{I} *_d \boldsymbol{K})(h, w) = \sum_{i+dk=h} \sum_{j+dl=w} \boldsymbol{I}(i, j)\, \boldsymbol{K}(k, l)\,. \tag{4.2}$$

The context module uses dilated convolutions to systematically aggregate multi-scale contextual information, without losing resolution. The architecture is based on the fact that dilated convolutions allow for an exponential expansion of the **receptive field** (the area of a layer's input that a kernel is applied to) without loss in resolution or coverage. The dilation factors used by the different layers of the context module range from 1 to 64. Reflection padding ensures that the resolution of the incoming feature maps is preserved.

A **deconvolutional** layer is used to upsample the heat maps output by the context module to the resolution of the original image, and a softmax layer maps the 19 values associated with each pixel into class probabilities, which is the final output of the network during inference. The actual pixel-level segmentation is obtained by for each pixel mapping the index of the largest probability to the class being represented by that index.

In our implementation, we removed the deconvolutional and softmax layers, and instead used the output of the context module as input to a third, "normalized cuts" module. Diagrams illustrating the conceptual structure of this module, during training as well as inference, are shown in Figure 4.2.

The data layer of the training network outputs three arrays: a preprocessed image, a downsampled version of the ground truth instance-level annotation of that image, and a binary foreground mask indicating which pixels of the downsampled annotation belong to the class we wish to partition. Image preprocessing consists merely of channel-wise mean subtraction, where the mean values were computed from the training data, as well as mirroring the image about its vertical axis with a probability of 0.5. Downsampling of the ground truth annotation is done by means of majority vote among neighboring pixels.

The preprocessed image is passed through the front-end and context modules of the Dilation10 network, resulting in the 19 semantic heat maps mentioned earlier. Due to restrictions in terms of hardware, as well as to speed up forward propagation, the heat maps of each image were computed in advance. Consequently, the data layer of our actual implementation does not output a preprocessed image, but instead outputs the pre-computed heat maps and passes them directly to the normalized cuts module. What is more, the resolution of the Cityscapes images is too large for the Dilation10 network to handle, and to compute the 128×256 pixel heat maps corresponding to a whole image required stitching together smaller heat maps computed from 256×512 pixel image tiles. Hence, precomputing the heat maps was rather an attractive way to go.

The normalized cuts module contains a chain of eight convolutional layers. The purpose of these layers is to learn features that will result in similarity matrices suitable for object partitioning through normalized cuts. The number of convolutional layers and kernels to use, as well as the size of those kernels, was determined primarily through experimentation. The number of kernels in the final convolutional layer was set to 40 to restrict the rank of the similarity matrix. To limit the size of the similarity matrix, a 2×2 max pooling layer with a stride of 2 reduces the spatial extent of the feature maps to 64×128 pixels. The final ReLU guarantees that each entry of the similarity matrix is non-negative.

The binary foreground mask output by the data layer is used to extract the pixels in the final feature maps and the downsampled ground truth annotation that correspond to the semantic class being subject to partitioning. The pixels extracted from the ground truth annotation are used to construct the target partition matrix $\boldsymbol{E}$. Reshaping the tensor obtained by extracting the foreground pixels of the feature maps yields the feature matrix $\boldsymbol{F}$, which in turn gives the similarity matrix through $\boldsymbol{S} = \boldsymbol{F}\boldsymbol{F}^T$. Finally, the loss layer evaluates $L_3(\boldsymbol{S}, \boldsymbol{E})$ as given by (3.14). Forward propagation is identical during training

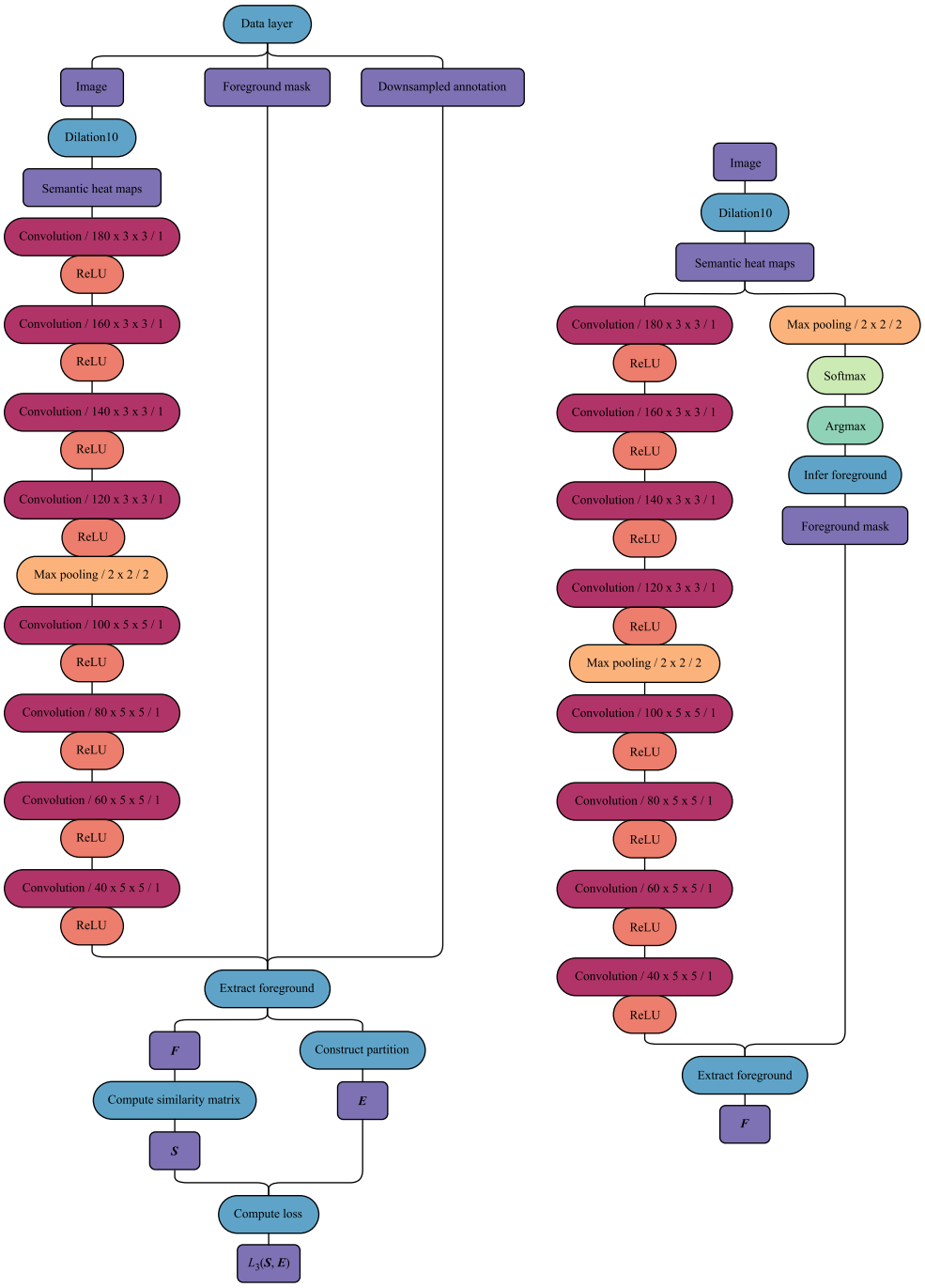

Figure 4.2: Networks used for training and validation (left), and for inference (right). The numerical codes that are associated with the convolutional layers and the pooling layers should be decoded as *number of kernels* $\times$ *kernel height* $\times$ *kernel width* / *stride* and *pooling height* $\times$ *pooling width* / *stride*, respectively. All convolutional layers except the final one have biases.

and validation. However, training additionally employs back-propagation to compute the gradients required for learning the parameters of the convolutional layers in the normalized cuts module.

During inference, the feature matrix $\boldsymbol{F}$ is computed in the same way, except that the binary foreground mask used to extract the pixels corresponding to the class we wish to partition is obtained from the semantic heat maps output by the context module, and not from the ground truth annotation, as is the case during training and validation. To have the resolution of these heat maps match the resolution of the feature maps output by the final convolutional layer of the normalized cuts module, a 2×2 max pooling layer with a stride of 2 is used. Max pooling seemed to work marginally better than average pooling. A softmax layer then transforms the 19 values associated with each spatial location in the downsampled heat maps into class probabilities. Lastly, an "argmax" layer extracts the index corresponding to the most probable class of each pixel, which results in a pixel-level semantic segmentation. This segmentation is then used to produce the binary foreground mask needed to construct the feature matrix $\boldsymbol{F}$, which is the final output of the network used for inference. The similarity matrix $\boldsymbol{S} = \boldsymbol{F}\boldsymbol{F}^T$ is then passed to Algorithm 3.1 for computing an instance-level segmentation of the foreground pixels. Upsampling the inferred segmentation to the original resolution forms the final prediction. No post-processing is applied. Since our method cannot produce overlapping mask predictions, the prediction confidence value required for each mask by the evaluation script serves no purpose.

A major drawback of our approach is that the number of instances K need be specified manually. Picking this number wrongly may drastically reduce the accuracy of the predicted segmentation. Spectral clustering methods sometimes base the number of clusters on the spectrum of the similarity matrix. We analyze our method using both the rank of the similarity matrix and the ground truth number of instances. Since no ground truth annotations are available for the test set, we replaced the true test with the validation set, and split the original training set into new training and validation sets. The thus formed validation set consists of the images from the cities of Aachen and Jena. This modification of the arrangement of the dataset results in a training set of $2{,}682$ images, a validation set of 293 images, and a test set of 500 images. Hereafter, whenever we mention any of the data splits, we refer to this modified organization of the data. It is possible that the heavy downsampling applied to the ground truth annotations turns the additional $20{,}000$ coarsely annotated training images rather useful. In spite of this, we refrain from using these images for training our model.

When using a ground truth annotation to help determine the number of instances to use during inference, K is set to the number of downsampled ground truth instance segments

that are, partially or fully, contained within the predicted foreground mask. This way, instance segments that are completely missed by the Dilation10 network do not contribute to the number K.

The networks were implemented using the Caffe deep learning framework (Jia et al., 2014). Non-built-in layers, such as the loss layer used to evaluate $L_3(\boldsymbol{S}, \boldsymbol{E})$, were implemented through Caffe's Python interface.

4.3 Results

For simplicity, we decided to consider instances of a single semantic class only. The choice fell on *car*, for which the Dilation10 network attains an IoU score of 93.3% (in terms of accuracy, *person* comes in second place with an IoU of 78.9%). Most ground truth annotations contain several cars, and do so also after downsampling.

Training was carried out on one of two 6GB GPUs of an Nvidia® GeForce® GTX Titan Z dual-chip graphics card for the built-in layers (Caffe did not support the use of multiple GPUs when some layers were written in Python), and an 8-core Intel® Xeon® E5-2640 CPU for the custom layers. A total of $650{,}000$ iterations were performed using the Adam algorithm. Adam seemed to work better than standard mini-batch gradient descent with momentum, possibly due to the difficulty of appropriately setting the learning rate for the latter algorithm. The hyperparameters of the Adam algorithm were set to $\lambda = 4 \times 10^{-6}$, $\delta_1 = 0.95$, $\delta_2 = 0.999$, and $\epsilon = 1 \times 10^{-8}$. Following He et al. (2015), each kernel parameter of a convolutional layer was initialized randomly from $\mathcal{N}\big(0, \sqrt{2/chw}\big)$, where c denotes the number of incoming channels (feature maps), and h and w are the kernel height and width, respectively. This initialization heuristic is designed to give all layers the same activation variance, taking rectifiers into account. All biases were initialized to 0.05. A **weigth decay** term of $(\alpha/2)\|\boldsymbol{\pi}\|_2^2$ was added to L_3 to offer some regularization, with $\alpha = 1 \times 10^{-4}$. The bias parameters were excluded from this norm penalizer. Since the number of foreground pixels varies between images, the mini-batch size needed be set to 1, resulting in purely stochastic optimization.

After some $650{,}000$ iterations, overfitting to the training set began to occur. As can be seen in Figure 4.3, the average training and validation losses reached a minimum after some $50{,}000$ iterations, then increased to and stabilized around a value of roughly $L_3 \approx 4.5$. Interestingly, the performance measures AP_{car} and $\mathrm{AP}_{car}^{50\%}$, as evaluated on the validation set, improved throughout the course of training. We can only speculate about the underlying reason for this behavior. Perhaps early optimization was mainly influenced by large instance segments, leading to moderate performance since the average precision metric

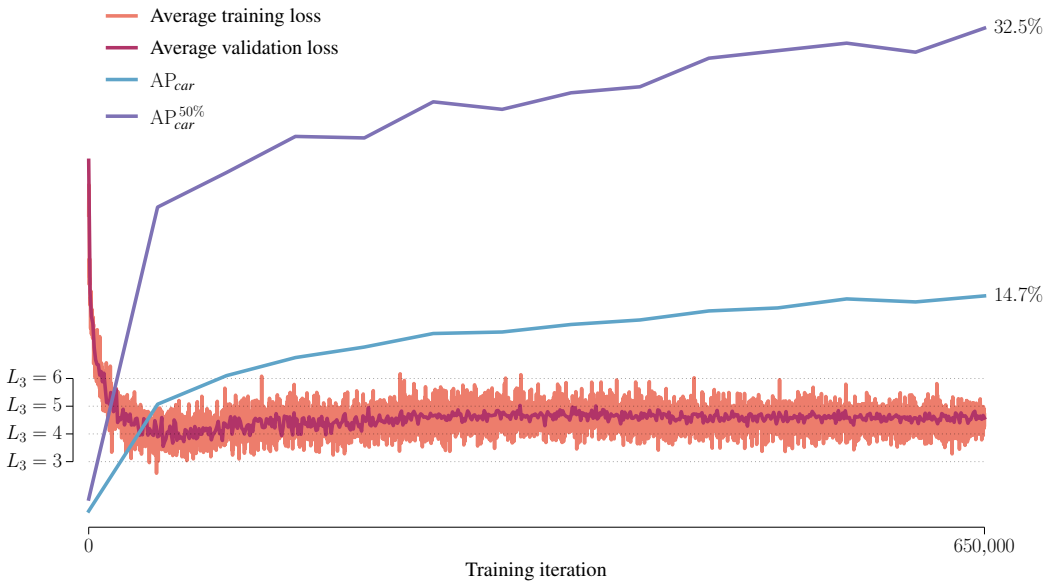

Figure 4.3: Average training loss and validation loss, and AP_{car} and $\mathrm{AP}_{car}^{50\%}$ on the validation set. The average training loss was computed using a moving average that considers the 25 most recent iterations. Each point on the validation loss curve is the average loss across the entire validation set.

is blind to the size of individual instances, but as training progressed adjusted the model parameters to better represent also small instances.

While the size of the similarity matrix depends on the number of foreground pixels, training processed an average of about 1.3 images per second, implying that the 650,000 iterations required a bit over a week to complete. The primary reason for training being so slow is the inherent complexity of computing $\boldsymbol{S}^{+}$ for large similarity matrices $\boldsymbol{S}$, which is done through eigendecomposition. Some effort was put into investigating whether this computation could be done faster on GPU than CPU, but without much success.

As baseline, we use a predictor that simply maps each connected component of the pixel-level segmentation foreground to a single object instance. We refer to this rather crude predictor as "CC". While it effectively infers instances that correspond to isolated segments, it fails for any isolated segment that contains two or more instances, as well as for instances that, due to occlusion, appear across multiple isolated segments.

To obtain an upper bound on the performance at a fixed resolution of prediction (no post-processing allowed), the performance of using downsampled ground truth annotations for prediction was evaluated. Downsampling was performed by means of majority vote among neighboring pixels. Upsampling the instances of the downsampled *car* segments to the original resolution formed the final predictions. We refer to this predictor as "GT".

Predictor	Downsampling factor	Foreground mask	K	AP_{car}	$\mathrm{AP}_{car}^{50\%}$
GT	2	Ground truth		86.2%	100.0%
GT	4	Ground truth		59.7%	94.8%
GT	8	Ground truth		32.6%	65.3%
GT	16	Ground truth		15.4%	35.6%
CC	1	Dilation10		6.9%	13.7%
CC	16	Dilation10		5.9%	13.3%
DNC	16	Dilation10	Ground truth	12.8%	29.1%
DNC	16	Dilation10	$\mathrm{rank}(\boldsymbol{S})$	5.9%	15.9%

Table 4.1: Test set performance of all predictors. The foreground mask used by the CC predictor at full resolution was obtained using the deconvolutional layer of the original Dilation10 network. For reference, although capable of segmenting all instance-level classes, Mask R-CNN (He et al., 2017) attains AP_{car} and $\mathrm{AP}_{car}^{50\%}$ scores of 46.9% and 68.3%, respectively, on the official test set.

The test set performances of all predictors are presented in Table 4.1. Evaluation of the GT predictor for different downsampling factors suggests that downsampling limits the maximum attainable performance rather drastically, especially for large values of the threshold variable θ. This is most likely an effect primarily caused by instance boundaries becoming coarser, which affects instance segments of small to moderate size, and less so due to some small instances going lost in the downsampling process. The baseline based on the connected components algorithm is, however, insensitive to downsampling. Our deep normalized cuts predictor performs significantly better than the baseline when using the ground truth annotations to help specify the number of instances K. It is, in fact, not too far from the performance of the GT predictor at the same resolution of prediction. Having said that, instead setting K to the rank of the similarity matrix severely reduces performance. The rank is typically larger than the true number of instances, which thus leads to an over-segmentation.

Figure 4.4 shows predicted instance-level annotations for a few test images in the case when K has been specified with the help of the ground truth annotations. Even though the normalized cuts loss function favors balanced solutions, our method partitions the pixels into parts of various size. Also note that our method is capable of assigning multiple isolated segments to the same instance (for good and for bad). It is clear that smoothing the mask boundaries could improve performance somewhat. However, such post-processing adds little value in highlighting the potential of the method, and is omitted.

Figure 4.4: Test images (left), ground truth instance-level annotations for the class *car* (center), and predicted instance-level annotations for the class *car* when using the ground truth number of instances (right). Note that mask colors carry no particular meaning.

Chapter 5

Conclusion

5.1 Recapitulation

Following work by Bach and Jordan (2006) and Ionescu et al. (2015), the objective of this thesis was to investigate the suitability of using normalized cuts for instance-level semantic segmentation, in a setting where the similarity matrix is learned from data using a convolutional network. Our architecture adds a convolutional module on top of the pixel-level semantic segmentation network of Yu and Koltun (2016), whose layers are able to learn representations for additionally segmenting object instances via normalized cuts. We trained our deep normalized cuts model to segment individual cars in the Cityscapes dataset (Cordts et al., 2016). While our method lacks a way to accurately predict the correct number of instances, using the ground truth number of instances to help specify the number of cuts resulted in almost twice the performance of our connected components baseline.

5.2 Future Work

We restricted the implementation to consider a single semantic class only. A generalization to multiple classes could be achieved in one of two ways. In principle, one could use a separate convolutional branch for each class. However, as the number of classes grows, this quickly turns impractical. Moreover, as the input to each such branch would be the same, and the target partition $\boldsymbol{E}$ knows nothing about semantics, features thus learned by different branches could turn out much similar. A better, certainly more practical way, would be to use a single branch for all classes. This could also prove to be more robust.

On a related note, the normalized cuts module developed here takes only the final feature maps produced by the Dilation10 network as input. Additionally taking as input also feature maps produced by some preceding layers could be useful.

Due to a lack of GPU memory, it was not possible to fine-tune the parameters of the Dilation10 backbone. After having trained the normalized cuts module, it would have been desirable to jointly fine-tune all convolutional layers, which could have improved performance somewhat. The Dilation10 parameters would then have been updated based on gradients back-propagated from both our loss function L_3, as well as the cross-entropy loss function used for pixel-level semantic segmentation.

Lastly, the rank of the similarity matrix did not work well as an indicator of the number of instances. An investigation of alternative ways to specify the number of instances would be needed if developing the method any further. It might be possible to explicitly learn the network to predict the number of instances, as was done by Liang et al. (2015).

Bibliography

Arnab, A. and Torr, P. H. (2017). Pixelwise Instance Segmentation With a Dynamically Instantiated Network, *arXiv:1704.02386*.

Bach, F. R. and Jordan, M. I. (2006). Learning Spectral Clustering, With Application to Speech Separation, *Journal of Machine Learning Research* **7**(Oct): 1963–2001.

Bai, M. and Urtasun, R. (2016). Deep Watershed Transform for Instance Segmentation, *arXiv:1611.08303*.

Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S. and Schiele, B. (2016). The Cityscapes Dataset for Semantic Urban Scene Understanding, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3213–3223.

Dai, J., He, K., Li, Y., Ren, S. and Sun, J. (2016). Instance-sensitive Fully Convolutional Networks, *European Conference on Computer Vision*, pp. 534–549.

Dai, J., Li, Y., He, K. and Sun, J. (2016). R-FCN: Object Detection via Region-based Fully Convolutional Networks, *Neural Information Processing Systems*, pp. 379–387.

Girshick, R. (2015). Fast R-CNN, *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1440–1448.

Girshick, R., Donahue, J., Darrell, T. and Malik, J. (2014). Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 580–587.

Golub, G. H. and Van Loan, C. F. (2012). *Matrix Computations*, JHU Press.

Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*, MIT Press. http://www.deeplearningbook.org.

Gu, M., Zha, H., Ding, C., He, X., Simon, H. and Xia, J. (2001). Spectral Relaxation Models and Structure Analysis for K-way Graph Clustering and Bi-clustering, *Technical report*.

Hariharan, B., Arbeláez, P., Girshick, R. and Malik, J. (2014). Simultaneous Detection and Segmentation, *European Conference on Computer Vision*, pp. 297–312.

He, K., Gkioxari, G., Dollár, P. and Girshick, R. (2017). Mask R-CNN, *arXiv:1703.06870.*

He, K., Zhang, X., Ren, S. and Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-level Performance on Imagenet Classification, *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034.

Ionescu, C., Vantzos, O. and Sminchisescu, C. (2015). Training Deep Networks With Structured Layers by Matrix Backpropagation, *arXiv:1509.07838.*

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T. (2014). Caffe: Convolutional Architecture for Fast Feature Embedding, *Proceedings of the 22nd ACM International Conference on Multimedia*, pp. 675–678.

Kingma, D. and Ba, J. (2014). Adam: A Method for Stochastic Optimization, *arXiv:1412.6980.*

Li, Y., Qi, H., Dai, J., Ji, X. and Wei, Y. (2016). Fully Convolutional Instance-aware Semantic Segmentation, *arXiv:1611.07709.*

Liang, X., Wei, Y., Shen, X., Yang, J., Lin, L. and Yan, S. (2015). Proposal-free Network for Instance-level Object Segmentation, *arXiv:1509.02636.*

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P. and Zitnick, C. L. (2014). Microsoft COCO: Common Objects in Context, *European Conference on Computer Vision*, pp. 740–755.

Meila, M. and Xu, L. (2003). Multiway Cuts and Spectral Clustering, *Technical report*.

Mitchell, T. M. (1997). *Machine Learning*, McGraw-Hill.

Ng, A. Y., Jordan, M. I., Weiss, Y. et al. (2001). On Spectral Clustering: Analysis and an Algorithm, *Advances in Neural Information Processing Systems*, pp. 849–856.

Overton, M. L. and Womersley, R. S. (1993). Optimality Conditions and Duality Theory for Minimizing Sums of the Largest Eigenvalues of Symmetric Matrices, *Mathematical Programming* **62**(1-3): 321–357.

Pinheiro, P. O., Collobert, R. and Dollár, P. (2015). Learning to Segment Object Candidates, *Advances in Neural Information Processing Systems*, pp. 1990–1998.

Polyak, B. T. (1964). Some Methods of Speeding up the Convergence of Iteration Methods, *USSR Computational Mathematics and Mathematical Physics* **4**(5): 1–17.

Ren, S., He, K., Girshick, R. and Sun, J. (2015). Faster R-CNN: Towards Real-time Object Detection With Region Proposal Networks, *Advances in Neural Information Processing Systems*, pp. 91–99.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). Learning Representations by Back-propagating Errors, *Nature* **323**: 533–536.

Shi, J. and Malik, J. (2000). Normalized Cuts and Image Segmentation, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **22**(8): 888–905.

Simonyan, K. and Zisserman, A. (2014). Very Deep Convolutional Networks for Large-scale Image Recognition, *arXiv:1409.1556.*

Uijlings, J. R., Van De Sande, K. E., Gevers, T. and Smeulders, A. W. (2013). Selective Search for Object Recognition, *International Journal of Computer Vision* **104**(2): 154–171.

Wilson, D. R. and Martinez, T. R. (2003). The General Inefficiency of Batch Training for Gradient Descent Learning, *Neural Networks* **16**(10): 1429–1451.

Wolpert, D. H. (1996). The Lack of A Priori Distinctions Between Learning Algorithms, *Neural Computation* **8**(7): 1341–1390.

Yu, F. and Koltun, V. (2016). Multi-scale Context Aggregation by Dilated Convolutions, *International Conference on Learning Representations*.

Zhao, H., Shi, J., Qi, X., Wang, X. and Jia, J. (2016). Pyramid Scene Parsing Network, *arXiv:1612.01105.*

Master's Theses in Mathematical Sciences 2017:E55
ISSN 1404-6342

LUTFMA-3329-2017

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/