INFERNING NETWORKS FOR GRAPH PARTITIONING

A Thesis

by

QUAN HOANG NGUYEN

B.S., Texas A&M University-Corpus Christi, 2017

Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in

MATHEMATICS

Texas A&M University-Corpus Christi
Corpus Christi, Texas

May 2019

INFERNING NETWORKS FOR GRAPH PARTITIONING

A Thesis

by

QUAN HOANG NGUYEN

This thesis meets the standards for scope and quality of
Texas A&M University-Corpus Christi and is hereby approved.

Beate Zimmer, Ph.D.
Chair

Reid Porter, Ph.D.
Co-Chair

Lei Jin, Ph.D.
Committee Member

May 2019

ABSTRACT

Image analysis, pattern recognition, and computer vision pose very interesting and challenging problems, especially in this time when billions of images are generated and uploaded every single day. In many computer vision systems, segmentation often is the first step in which it localizes the objects presented in the image by partitioning the image into meaningful segments. More precisely, image segmentation is the task of assigning discrete labels to pixels, where the label represents the segment (or cluster) to which a pixel belongs. We express an image as a graphical model, also known as an affinity graph, whose vertices denote pixels or regions and whose edges denote adjacency. Each edge in the affinity graph has a weight that quantifies the similarity between the adjoining vertices. To partition the graph, we select a threshold and discard all edges with weights below the threshold and then form segments as path-connected regions. We can define an energy function of a segmentation by adding up the weights of all edges between different segments, which is referred to as the correlation clustering energy. Partitioning the graph to obtain a segmentation is seen as inference in a graphical model and can be formulated as the minimization of the energy function over all possible segmentations, an NP-hard problem. We train a deep neural network to produce the affinity graph, whose segmentation minimizes a natural learning objective called the Rand error. For a graph with a ground truth segmentation, the Rand error measures the pairwise misclassification error between a predicted segmentation and the ground truth segmentation over all possible vertex pairs. We describe an efficient mini-batch learning algorithm based on Kruskal's algorithm and discuss two formulations of the loss function and two graph encoding schemes used in training the neural net. We present a novel concept, namely that during the training process we select the optimal threshold that minimizes the correlation clustering energy function over a restricted set of segmentations given by different thresholds. We present experiments on a synthetic dataset that illustrate that adding this extra inference step to the training of the neural net causes it to learn different affinities that lead to a 5% reduction in the Rand error on a validation set of similar synthetic images.

ACKNOWLEDGEMENTS

I would like to acknowledge the financial support of the Department of Mathematics & Statistics and the College of Graduate Studies at Texas A&M University-Corpus Christi. Without their generous help, it would have been extremely difficult for me to pursue my study and research.

The completion of my thesis would not have been possible without the help and nurturing of every member of my committee. I would like to express my deepest appreciation to Dr. Beate Zimmer, the chair of my committee. She taught me a great deal about both scientific career and life in general as well as pushed me with constant support to where I am now. I am extremely grateful to Dr. Reid Porter whom I have had the pleasure to work with during this thesis project. He has shown me how to be a positive and inspiring scientist and how research is about exploring a story to tell. I also want to thank Dr. Lei Jin for his extensive professional advice and guidance.

I am very lucky to be a member of a wonderful family. I am forever in indebted to my parents, whose love and support I never forget. I am grateful to my brother, Tu, for his technical advice. Finally, I wish to thank my loving wife, Ngan, who has always been very supportive during the pursuit of my career goals.

TABLE OF CONTENTS

CONTENTS PAGE

ABSTRACT v

ACKNOWLEDGEMENTS vi

TABLE OF CONTENTS vii

LIST OF FIGURES ix

INTRODUCTION 1

LITERATURE REVIEW 5

2.1 The Image Segmentation Problem 5

2.2 Inference and Graphical Models 7

2.3 Graph Based Segmentation 9

2.4 Correlation Clustering 11

2.5 Inference with Connected Components and Correlation Clustering 13

2.6 Learning and Segmentation 16

2.7 A Learning Model: Neural Networks, Convolutional Layers, and the U-net 17

2.8 A Learning Objective: Rand Error for Segmentation Evaluation 25

2.9 A Learning Algorithm: Maximin Affinity Learning 27

2.10 Inferning 29

METHODOLOGY 32

3.1 Learning with Rand Error and Kruskal's Algorithm 32

3.2 Loss functions for Rand Error and Kruskal's algorithm 35

3.3 Inferning Connected Components for Segmentation 42

IMPLEMENTATION 47

4.1 Introduction 47

4.2 The dataset 48

4.3 Auto differentiation 50

4.4 1D encoding 51

4.5 2D encoding 53

RESULTS 56
5.1 Using a linear classifier for initial results 56
5.2 Using a mini U-net with “diff” loss 61
5.3 Using a mini U-net with “hit” loss 66
5.4 Comparison of inferning and threshold zero models for a mini U-net with the “diff” loss 70
DISCUSSION 74
REFERENCES 78

LIST OF FIGURES

FIGURES PAGE

2.1 A sample from the ADE20K dataset with three segmentation levels: (top-right) object level, (bottom-left) first parts level, and (bottom-left) second parts level. 6

2.2 Samples from the Alpert et al. Segmentation evaluation database: (left) color images, (middle) grayscale images, and (right) object segmentations. 7

2.3 An illustration of a perceptron. Adapted from [20]. 17

2.4 An illustration of a multi layers perceptron neural network. Adapted from [20]. 18

2.5 The logistic activation function, a smooth version of the binary step function that outputs $\{0,1\}$. Adapted from [20]. 19

2.6 An example of a single channel 2D convolution in neural network with kernel of size 2×2, stride 1, where only the "valid" part of the input is used, (i.e. no padding to the input). The input is a 3×4 matrix and the output is a 2×2.. Adapted from [12]. 22

2.7 U-net architecture with input image of shape (572x572x1). Adapted from [25]. 23

3.8 The hinge loss vs. affinity when $l_{uv} = 1$ and $\theta = 0$. 43

3.9 The hinge loss vs. affinity when $l_{uv} = -1$ and $\theta = 0$. 43

3.10 The hinge loss vs. affinity when $l_{uv} = 1$ and $\theta = 0.2$. 43

3.11 The hinge loss vs. affinity when $l_{uv} = -1$ and $\theta = 0.2$. 43

3.12 An example of the effect of the choice of the parameter threshold θ on the segmentation and its correlation clustering energy. 45

4.13 Top row: Original sample images. Bottom row: The same images with added noise. . . . 49

4.14 An image sample from the synthetic dataset: (from left to right) the original image with noise, the compressed image, and the smoothed compressed image. 49

4.15 Flowchart for the linear classifier acting on a 1D vector of edge information. SGD stands for stochastic gradient descent. 53

4.16 Flowchart for a classifier acting on a 2D representation of the affinity graph. SGD stands for stochastic gradient descent. 54

5.17 Training statistics (training and validation loss/Rand error) of a typical training trial of the linear model with “diff” loss. 58
5.18 Training statistics (training and validation loss/Rand error) of a typical training trial of the linear model with “hit” loss. 59
5.19 Some ground truths (left) and segmentation predictions (right) of the linear classifier with the “diff” loss. 59
5.20 Some ground truths (left) and segmentation predictions (right) of the linear classifier with the “hit” loss. 60
5.21 Average hinge losses on the training set (green) and the validation set (orange) of 5 independent trials of the mini U-net and “diff” loss model. 63
5.22 Corresponding average Rand errors on the training set (green) and the validation set (orange) of 5 independent trials of the mini U-net and “diff” loss model. 63
5.23 Training statistics (training and validation loss/Rand error) of a typical training trial of the mini U-net with the “diff” loss. 64
5.24 Some ground truths (left) and segmentation predictions (right) of the mini U-net and “diff” loss model. Each color specifies a class. 65
5.25 Average hinge losses on the training set (green) and the validation set (orange) of 5 independent trials of the mini U-net and “hit” loss model. 67
5.26 Corresponding average Rand errors on the training set (green) and the validation set (orange) of 5 independent trials of the mini U-net and “hit” loss model. 68
5.27 Training statistics (training and validation loss/Rand error) of a typical training trial of the mini U-net with the “hit” loss. 68
5.28 Some ground truths (left) and segmentation predictions (right) of the mini U-net and “hit” loss model. Each color specifies a class. 69
5.29 Corresponding average Rand errors on the validation images of models trained with and without using the inferning threshold. There are 5 trials of each model and the mean curve of each is highlighted. 70

5.30 The box-plots of affinities predicted by models trained with (red) and without (blue) using the inferning threshold. 71

5.31 The receiver operating characteristic curves of pixel-pair connectivity classification of the inferning and non-inferning model. The blue dots are threshold zero, red + signs indicate the inferning thresholds. 73

INTRODUCTION

Image segmentation is the process of dividing an image into multiple parts. Typically, this is applied to identify objects and regions or other meaningful information in the image. Many computer vision systems in various applications employ image segmentation as the first step in understanding and processing the image. For example, in medical imaging, image segmentation is used to locate tumors and other organs. In autonomous driving, image segmentation involves in identifying pedestrian detection, brake light detection, traffic lights detection, etc.

In this thesis, we study a learning framework in which a real-valued affinity score for each neighboring pixel pair in an image is produced by a neural network. We regard an image as a graph where neighboring pixels are joined by an edge and the affinities provide weights for those edges. The affinity graph is then cut by the connected components procedure with a threshold parameter θ, resulting in a segmentation of the image ([23], [28]). To infer a segmentation from the graphical model is to cut the graph in a way that best agrees with the affinities between adjoining vertices in the graph. If the affinity (a correlation score) between two vertices is positive or high then they should be in the same segment and oppositely, if the affinity between two vertices is negative or low then they should be in different segments. The connected components procedure is a simple method to get a segmentation from the affinity graph. The procedure discards edges whose weights are below a threshold parameter θ and then grows segments from the vertices that still have paths between them after this procedure. While the parameter θ is usually chosen to be some measure of the center of the affinities (e.g. in this thesis, the affinities are zero-centered and hence the baseline threshold is zero), we do vary the threshold to find an optimal threshold for some assessment of quality of a segmentation. The correlation clustering energy function for example can be used for such an assessment. Then the "best" segmentation is one that minimizes the correlation clustering energy. It has been proven to be NP-hard to find this optimal segmentation solution in general. However, if only the connected components procedure is considered to segment an affinity graph, we can find the best connected components segmentation in polynomial time by going through all possible thresholds for the affinity graph. While the affinities

could for example be calculated as a function of the difference in intensities of neighboring pixels, they can also be generated by a neural networks. If trained properly, a neural network opens doors to produce affinity graphs that are superior to ones from fixed designs. We investigate the online stochastic gradient descent maximin affinity learning [28] and the mini-batch gradient descent learning based on Kruskal's algorithm [23], which are two frameworks that learn to produce affinity graphs for segmentations by the connected components procedure. We explore whether we can add the correlation clustering energy idea to the existing frameworks and see whether the extra computations in finding the best threshold for the connected components procedure can be justified by a performance gain.

The main contributions of this thesis are:

- We formulate an $\mathcal{O}(e \log n)$ mini batch affinity learning algorithm based on Kruskal's algorithm, where n is the number of vertices and e is the number of edges in a connected graph. Previous work [28] had used an $\mathcal{O}(e \log n)$ algorithm to estimate the gradient of each of the n^2 terms in the Rand error. Our modification computes the gradient of all n^2 terms in a single $\mathcal{O}(e \log n)$ pass.

- We introduce an inference step into connected component segmentation that, in our experiments, leads to lower Rand errors and a more robust distribution of the affinities, supporting our hypothesis that more time spent in inference improves the overall performance of the connected component segmentation.

- We discuss how to write almost everywhere differentiable loss functions that approximate the Rand error and can be used with the automatic differentiation feature in deep learning libraries such as Tensorflow. We discuss two different options to map the pixel based training data to edge affinities in a graphical model and demonstrate their performance empirically.

- Correlation clustering is NP-hard. Our algorithm uses the correlation clustering energy function but only minimizes it over a linearly ordered set given by maximum spanning tree (MST) graph cuts. Instead of trying to minimize the energy function over all possible segmentations, we use the learning process to get better affinities and a different chain of segmentations in each step.

The thesis is organized as follows:

In Chapter 2 - "Literature review", we describe relevant concepts that motivate our research. We introduce the problem of image segmentation by describing some of the benchmark datasets. Then we discuss how we consider the affinity graph as a graphical model and describe some existing inference methods which cut the affinity graph and produce segmentations in sections 2.2 and 2.3. We consider the correlation clustering energy function as a measurement of the quality of a segmentation (section 2.4). In section 2.5, we discuss the connected component algorithm, which plays a crucial role in our learning framework. Since we will train a neural network to learn affinities, we first need some background on neural networks and need to decide on the neural network architecture that we want to use. There are many available options which we discuss in section 2.7. We then need a way to evaluate our segmentation and some measures to serve as the learning objective for the neural network. For this we choose a variation of the Rand index [24], called Rand error, which will be discussed in section 2.8. We discuss an existing learning algorithms that is closely related to our work, that is, in section 2.9 we describe the online stochastic gradient descent maximin affinity learning [28]. We then introduce the idea of "inferning" in section 2.10.

Chapter 3 - "Methodology" lays out the original contributions that are part of our algorithm. We describe the modification of Kruskal's algorithm that not only constructs a maximum spanning tree for a graph but also keeps track of how many correct and incorrect edges each MST edge adds to a segmentation. Since we use gradient descent to optimize the parameters of the neural network, we also need to study specifically how we define and calculate the training loss function, which is used to make the gradient updates to the parameters of the network. The Rand error itself is not continuous or differentiable, hence it is not suitable as a loss function. To remedy this issue, we discuss how to design continuous loss functions that allow us to calculate gradients and allow the neural net to learn to minimize the Rand error of the segmentations from cutting the produced affinity graphs. We provide two suggestions that improve upon the mini-batch gradient learning algorithm [23]. At last, we discuss the idea of "inferning", in which we try to link correlation clustering energy and to the existing framework.

Chapter 4 - "Implementation" is devoted to technical details of coding the learning framework.

First we describe how our training and test images were constructed. Then we discuss how we preprocess the images before feeding them into the classifier. We encode a graph structure into a multidimensional array (also known as a tensor), which is the main medium in machine learning libraries such as Tensorflow [1]. The automatic differentiation in Tensorflow is also introduced in this chapter. We rely on this automatic differentiation heavily in our implementation and for it to work correctly as intended, we have to make sure all of our formulations are well defined.

Chapter 5 - "Results" presents our experiments and their results. We train the same convolutional neural network architecture with two different formulations of the hinge loss function, which we call "diff" and "hit" models. Details of theses two formulation can be found in section 3.2. We train and validate these networks on a set of synthetic images with added Gaussian noise. We also present results of applying the "inferning" idea to the same setup and show that this indeed reduces the Rand error on the test images.

Chapter 6 - "Discussion" summarizes our results and describes areas where we would have liked to have had more time for further investigation and discuss further research directions.

LITERATURE REVIEW

2.1 The Image Segmentation Problem

In computer vision, image segmentation is the task in which we partition an image into multiple connected regions (i.e. set of pixels). Image segmentation is considered a low-level problem in the sense that its goal is to decompose an image into something that is easier to analyze. Image segmentation is usually the first module of a vision system. There are many practical applications that take advantage of image segmentation, including medical imaging, object detection, and object recognition. There is a large field of literature and resources available for this problem.

The learning framework in this thesis is formulated as a supervised learning problem. That is, for each training or validation image that our framework processes, there is a corresponding ground truth segmentation that is usually annotated by a human subject. Ideally, the learning framework can learn from the examples of input images and target segmentation to the point where it can generalize its "knowledge" to unknown input instances. In our context, we train a neural network to learn from a dataset of images and their segmentations to produce a structure called the affinity graph, which will be ultimately partitioned and give a segmentation. We describe two datasets for image segmentation:

- The ADE20K dataset [31] is free and available through the Computer Vision Group at MIT's website. The dataset contains more than $22,000$ color images of various scenes and places and the corresponding annotations including object segmentation masks, part segmentation masks, and descriptions of the contents. This is rather a large dataset than can be use to train learning framework of multiple tasks (e.g. object segmentation, segmentation as in this thesis, image classification, image understanding, etc..). For our framework, we could use the part segmentation masks as pixel ground truth labels. Each image has multiple parts segmentation masks (figure 2.1) that indicate different level in the part hierarchy. We can be combine these into one single mask to train our learning framework.

- The Alpert et al. Segmentation evaluation database [2] is another available choice. The database

Figure 2.1
A sample from the ADE20K dataset with three segmentation levels: (top-right) object level, (bottom-left) first parts level, and (bottom-left) second parts level.

contains 200 grayscale images and their color source with hand-drawn object segmentation ground truths. The database tries to avoid ambiguities as it only considers images that clearly depict one or two objects (figures 2.2). Also, each each segmentation ground truth is agreed by three different human subjects to increase the consistency of the ground truth. To use this database with our learning framework, we can consider the background and the object(s) as segments, resulting in two or three segments per image. Being a relatively small database and having only few segments per image, this database is worth looking into because of its simplicity and consistency.

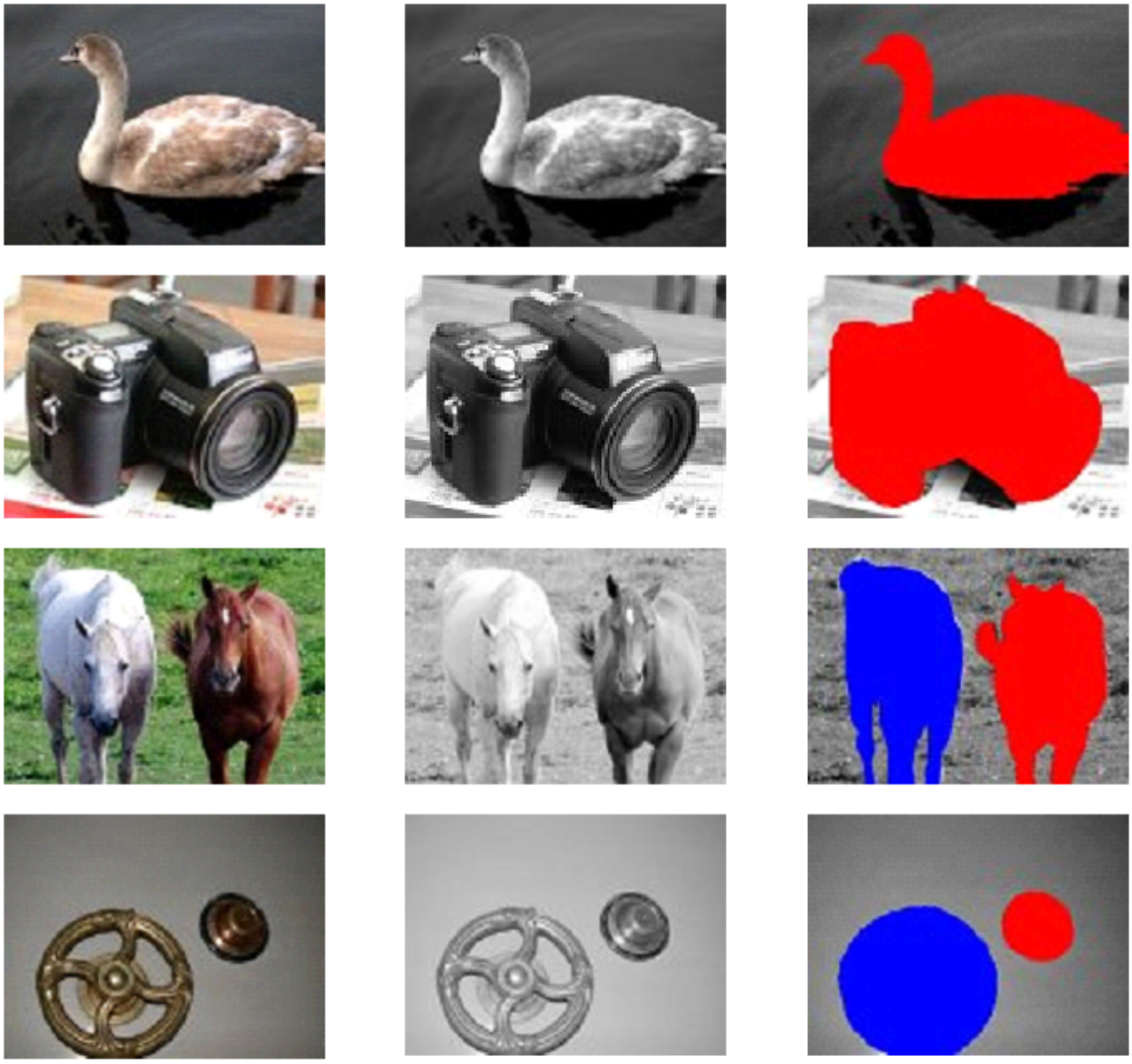

Figure 2.2
Samples from the Alpert et al. Segmentation evaluation database: (left) color images, (middle) grayscale images, and (right) object segmentations.

2.2 Inference and Graphical Models

A graphical model is a very powerful tool to represent an image segmentation problem. The use of graphical models allows us to exploit theorems and algorithms for graphs that have been developed over the years, especially since graph theory can be traced back to as early as 1736 [7]. Graphical models have appeared as a popular method to model relations and processes in many physical, biolog-

ical, social and information systems. The flexibility in using graphs can be utilized in many instances of structured data, and imagery is among those.

Definition 2.2.1. A graphical model is a probabilistic model for which a graph expresses the conditional dependence structure between random variables.

In a graphical model, the vertices represent random variables (e.g. labels of pixels) and the edges indicate the relationship between these variables (e.g. correlations between pixels). There is more than one type of graphical models, for example:

- Bayesian networks, also known as directed graphical models, have directed edges (i.e. each edge points from one vertex to another). Each edge indicates which variable's distribution is dependent on the other's.
- Markov random fields, also known as undirected graphical models, have undirected edges. The interaction between two adjoining vertices has no intrinsic direction.

In graphical models for image segmentation applications, each pixel has a label $l_i \in \{0, 1, 2, ...\}$, which is considered a random variable. For any two neighboring vertices, there is a weighted edge that indicates their relationship, typically their similarity. For example, we could have a 4-connected or a 8-connected lattice graph represent an image. It is natural to use an undirected model in our context since we want to operate in both direction of the adjacency relation (i.e. there is no necessity to specify a directional relationship between labels of a pixels). To quantify the similarity between the adjoining vertices, a weight, which we call an affinity, is assigned to each edge.

We refer to the weighted graph as an affinity graph. In our affinity graph, the affinities (or edge weights) are defined by a function

$$A : X_{ij} \to \mathbb{R}, \tag{2.1}$$

where X_{ij} is a subset of the image that will be used by the function to determine the weight between vertices i and j. Generally each affinity between a vertex pair is based on the neighborhood of the pair, which could be the whole image. The affinity function (equation 2.1) might be designed as a fixed function for specific applications or might be as flexible as a neural network.

Once an affinity graph is defined (i.e. the graph structure and the edge weights are defined), we can turn to the problem of inference in graphical models, in which we wish to compute the distribution of the random variables. In our context, we want to infer the label of each vertex of the affinity graph, resulting in a segmentation of the image. A segmentation can thus be seen as an inference in the affinity graph. This allows the formulation of the optimal segmentation as a minimization of an energy function (also known as a cost function), which is defined as

$$\widehat{Y} = \arg\min_{Y} \mathscr{E}(Y, E) \tag{2.2}$$

where $\widehat{Y}$ is the prediction for the best segmentation, Y is a segmentaion and E is the set of all edges in the affinity graph. One choice of the energy function $\mathscr{E}$ is the correlation clustering energy function, which will be introduced in section 2.4. We note that different choices of the energy function produce different outcomes. We now discuss some inference methods in graphical models in the context of image segmentation.

2.3 Graph Based Segmentation

In recent years, graph-based algorithms have been getting more attention in the research community in image segmentation and data clustering [8]. Graph-based approaches have a discrete and mathematically simple representation. They are suitable for many efficient and provably correct methods and algorithms for various tasks. A lot of work has been done on graph theory and we can utilize existing ideas, algorithms, and theorems for specific tasks. In recent publications on graph-based segmentation algorithms such as graph cuts, random walker, and shortest paths appear frequently as the baseline.

A graph cut algorithm, formulated on a weighted graph, does exactly what its name suggests. Given a connected graph $G(V,E)$, the algorithm provides a set of edges whose removal partitions the graph into sub-graphs. Here, the edge weights indicate the similarity between vertices. That is, the pair of vertices whose edge weight is high should be in the same segment and oppositely, the pair of vertices whose edge weight is low should be in different segments. A cut has a cost that is dependent on the weights of edges and how the graph is cut. A basic form of such a graph cut algorithm cost

function is defined in the min-cut algorithm, which cuts the graph into two segments A and B. The cost for the cut is the sum of the weights of cut edges:

$$\text{cut}(A,B) = \sum_{u \in A, v \in B} A_{uv}, \tag{2.3}$$

where A_{uv} is the affinity (i.e. the edge weight) between vertices u and v and A and B are the two disjoint segments. The best cut is one that minimizes the cost function. A min-cut only partitions the graph into two sub-graphs A and B, however one can perform multiple min-cuts to get more segments. A segmentation (i.e. labeling) is easily achieved once a graph is cut. However, the min-cut tends to produce small isolated components [8], which is undesirable.

The normalized-cut algorithm [27], which also requires a connected graph, addresses this problem by normalizing the cost:

$$\text{Ncut}(A,B) = \frac{\text{cut}(A,B)}{\sum_{u \in A, t \in V} A_{ut}} + \frac{\text{cut}(A,B)}{\sum_{u \in B, t \in V} A_{ut}}, \tag{2.4}$$

where V is the set of all vertices of the graph and other terms are defined in equation 2.3. With this definition, if a cut tries to partition a graph into too small segments, the cost will go up even though some of the cut edges may have small weights. In general, the normalized-cut problem is proven to be NP-complete [27]. There have been many proposed techniques that deal with the disadvantages of normalized-cut. These techniques addresses the metrication error (i.e. "blockiness") and the preference for balanced partitions (i.e. sizes of segments) [8].

The random walker segmentation algorithm [13] is another graph cut method. It requires a weighted undirected graph with labeling seeds. The seeds indicate different segments, that is, if we expect to have n segments, there should be one labeling seed located in each segment. A random walker is placed at each unlabeled vertex. A random walk is started at each vertex with the edge weights as transition probabilities (i.e. the edge weight indicates the probability that the random walk will move from one vertex to the other). A diffusion process is initialized with the labeling seeds as tracers. Each vertex is assigned to the label of the seed which the random walk most likely reaches first. Here, the edge weights affect the decisions of a random walk. In image segmentation, these edge weights should tailor the random walk to avoid crossing region boundaries, hence edges at

region boundaries should have lower probabilities to be crossed than edges inside each region. In general, the method provide a stable solution that is robust to perturbations. However, since the method depends heavily on the prior (i.e. the labeling seeds), it is a disadvantage that this method needs an expert to assign the seeds in applications where it is difficult to algorithmically assign those or when the number of segments is not predetermined.

The shortest path algorithm uses foreground seeds and background seeds. A vertex is labeled as foreground if there is a shorter path from that vertex to one of the foreground seeds than to any of the background seeds. This algorithm also requires a weighted graph, although it could be directed or undirected. The weights can be interpreted similarly to ones in graph cut and random walker. This approach is fast and relatively easy to understand and implemented. It also does not suffer from the problem of small isolated regions. However, it also has a strong dependence on the seed assignment as the random walker algorithm and is more likely to cross weak object boundaries because only a single shorter path is required for label assignment [8].

2.4 Correlation Clustering

Correlation clustering seems a natural and intuitive concept, but surprisingly only goes back to a 2002 paper [4] by Bansal et al. or [5]. It was driven by wanting to sort documents into groups. Their motivation was to overcome two major shortcomings of other clustering methods. First, the number of clusters is not predetermined as it would be, for example, in k-means clustering. Secondly, it is not based on Euclidean distances between data points, which in high dimensions can suffer from the "curse of dimensionality" – a term coined by Bellman in 1961 - where the indegree distribution of the k-nearest neighbor directed graph becomes skewed due to the emergence of hub points that appear in many k-nearest neighbor lists. This affects clustering if the feature vectors for each vertex are high-dimensional. Redundant information in the feature vector can also negatively affect clustering methods. Instead of the Euclidean distance, correlation clustering uses the correlation between two feature vectors or vertices. Such affinities A_{ij} between vertices v_i and v_j can be decomposed into their sign and their magnitude. A positive sign indicates that the vertices should be in the same segment and the magnitude can be interpreted as a probability that the sign is chosen correctly. This allows

the interpretation of noise in the affinity function. The clustering has to try to remove this noise. A comprehensive but basic survey of correlation clustering methods is given in [21].

The goal in correlation clustering is to maximize the number of positive edges inside clusters and the number of negative edges between clusters, which is called "maximizing agreements". Equivalently one could try to minimize disagreements. Originally correlation clustering was designed for a fully connected graph. One important basic fact is that if there is a perfect clustering, it is easy to find by deleting all negative edges. Another important basic fact is that if the graph contains a triangle with two positive and one negative edge, there is no perfect clustering. Finding the optimal solution of the correlation clustering problem has been shown to be NP-hard, meaning it is equivalent to a number of other problems for which no polynomial time solution is known. In [5], the authors prove NP hardness of reducing disagreements for the unweighted case by reducing the problem of partition into triangles – which is known to be NP-hard - to the problem of minimizing disagreements. Instead of looking for the exact solution, approximate solutions are used for correlation clustering. Various approximations have been proposed. In 2004, Bansal et al. proposed a constant factor approximation for minimizing number of disagreements. It produces a 3-approximation. When a 3-approximation is applied to two clusters (the best case), if the graph representing the vertices of the two clusters contains k edges which are treated opposite to the sign of their affinity, then the approach used for minimizing disagreements will produce at most $3k$ such mistakes. For an n-approximation, the algorithm will produce not more than nk mistakes.

Another approach in [5] is a polynomial time approximation for maximizing agreements. The goal is to obtain a clustering having at most εn^2 fewer agreements than the optimal clustering, provided that the optimal clustering has at least $n(n-1)/4$ agreements and has non-singleton clusters having size greater than εn.

For a fully connected graph one can regard minimizing disagreements as a linear programming problem, also in [5]. Let $X_{ij} \in \{0,1\}$ be the variable defining which edges are cut: if the edge e_{ij} is within a cluster, $X_{ij} = 0$ and if the edge e joins two clusters $X_e = 1$. We write the set of positive edges

as E^+ and the set of negative edges as E^-. The linear program can then be written as

$$\text{Minimize} \sum_{e \in E^+} A_e X_e + \sum_{e \in E^-} A_e (1 - X_e)$$

subject to the constraints

$$X_{ij} \in [0,1]$$

$$X_{ik} + X_{kj} \geq X_{ij}$$

$$X_{ij} = X_{ji}$$

Then they use a process called rounding to construct a segmentation from the X_{ij} in which regions are grown in a way that avoids disagreement within clusters. A termination condition of a $\mathcal{O}(\log(n))$ approximation is placed on the region growing procedure.

Another way to define correlation clustering is in terms of inference in graphical models.

$$\mathcal{E}(Y,E) = \sum_{e_{ij}} I(y_i \neq y_j) A(X_{ij}), \tag{2.5}$$

where Y denotes the set of labels, each $y_i \in Y$ denotes the label of vertex i, and $A(X_{ij})$ denotes the affinity function (equation 2.1). Here, $I(y_i \neq y_j)$ is the indicator function that gives 1 if $y_i \neq y_j$ and 0 otherwise. In the context of correlation clustering, each affinity is in $[-1,1]$ and its sign denotes whether the edge should be cut and its magnitude indicates the confidence of the decision. In the later discussions, we interpret our affinity graph as in correlation clustering but do not require the affinities to be bounded.

2.5 Inference with Connected Components and Correlation Clustering

The connected components segmentation (algorithm 1), which was mentioned earlier, is a two-step procedure that partitions a weighted graph based on a fixed threshold θ: (1) discard the edges with affinity weights less than θ, and (2) iterate through the resulting graph to locate connected components. The algorithm is good at identifying blobs of vertices that are connected by edges whose affinities are higher than the parameter threshold θ. While it is a relatively easy to understand and to implement algorithm, the connected components segmentation does has weaknesses. The algorithm

tends to produce small isolated segments at the boundaries where the similarity between vertices is lower. Also, it can under-segment (i.e. make too few segments) badly in a noisy affinity graph because only one edge whose affinity is higher that the threshold is enough to connects two segments.

Data: A weighted undirected graph $G(V,E)$ whose A_e is the weight of edge e and a threshold θ
Result: A set of vertex-label pairs for all vertices

```
L is initialized to an empty collection of (key, value) pairs, or a dictionary;
i = 0;
for each e in E do
    if A_e < θ then
        delete e;
    end
end
for each v in V do
    if v is not in L then
        C = {traversed vertices in G from v} ;
        for each u in C do
            add (u, i) to L;
        end
        i ← i + 1
    end
end
return L
```

Algorithm 1: The connected components procedure.

Lemma 2.5.1. *Applying the connected components segmentation with threshold θ to a weighted graph produces the same segmentation that would be obtained by applying the connected components segmentation with threshold θ to a graph with the same vertices, but only the edges that form a maximum spanning tree for the graph.*

Proof. Assuming the procedure is applied to a maximum spanning tree (MST) of a weighted undirected connected graph (assuming the edge weights are not necessary unique, there might be more than one MST), consider any two vertices u and v afterward in the cut MST, two cases are possible:

- Case 1: u and v are in the same segment of the cut MST. On the cut MST graph, the path that connects u and v must consist of edges whose weights exceed the threshold θ. Hence, if the procedure is applied to the original graph directly, there exits at least one path that connects u

and v must consist of edges whose weights are all greater or equal to the threshold θ so u and v are in the same segment.

- Case 2: u and v are in different segments of the cut MST. On the cut MST, there is no path between u and v. If the connected components procedure is applied to the original graph, and u and v are in the same segment, then there is at least one path on the cut graph that connects the two vertices. Edges on the path have weights greater than or equal to θ. Adding all edges of this path (ignoring ones that are already on the spanning tree) to the uncut MST will create some cycles in the MST graph. For each cycle, we can delete one edge whose weight is smaller than the added edge's (there must be such an edge otherwise we would have a path between u and v on the cut MST). In this fashion we have constructed a new spanning tree whose sum of edge weights is larger than sum of edge weights of the original MST, which is a contradiction. Therefore, u and v must be in different segment in the cut graph.

□

We propose to use the connected components segmentation method to simplify the NP-hard correlation clustering problem: we try to learn an affinity function that is geared towards simplifying the optimization by reducing the noise in the affinities. In particular, we want to use the connected components algorithm, which is based on a simple thresholding of the affinity function instead of a more elaborate optimization process to minimize disagreements in the clustering. We hope to offset the restriction in the model space by learning a better affinity function.

By Lemma 2.5.1, the connected components procedure for a given set of edge affinities can only produce a limited subset of all possible segmentations as the threshold parameter θ varies. Specifically, the maximum spanning tree of a graph has $(\|V\| - 1)$ edges so there at most $\|V\|$ different ways (including keeping all edges and not cutting any edge) to segment the spanning tree, or equivalently the graph. Here, V is the set of all vertices in the graph. For each of the possible connected components segmentations we can calculate the value of the correlation clustering energy function. In the end, we can pick the threshold that gives the lowest value of this function as the segmentation

threshold varies. In this manner, we are minimizing the correlation clustering energy function, but only over an increasing family of at most $\|V\|$ segmentations, not over all possible segmentations.

Calculating the correlation energy given a segmentation using a brute-force algorithm takes $O(\|E\|)$, where E are the set of edges in the affinity graph. If the graph is lattice, this is equivalent with $O(\|V\|^2)$, where V is the set of vertices in the affinity graph. Hence for fixed affinities, we can find the best segmentation produced by the connected components procedure in polynomial time while the original correlation clustering problem, which considers all possible segmentations, is NP-hard. The method outlined above is an approximation to the actual minimization of the correlation clustering energy function, constrained by the connected components procedure.

The connected components procedure has some interesting properties. If there is no confusion in the affinity graph, that is, if there is no conflicting affinities, the connected components procedure with $\theta = 0$ gives the optimal segmentation that minimizes the correlation clustering energy. On the other hand, the set of segmentations produced by the connected components procedure is invariant to the shifting of all the affinities. That is, suppose we get a segmentation Y by applying the connected components procedure with a threshold θ to a graph, if all affinities of this graph are shifted by a fixed amount, we can retrieve Y by shifting the threshold by the same amount. This is directly observed from the definition of correlation clustering energy (equation 2.5).

2.6 Learning and Segmentation

In machine learning, supervised learning is a task where a function is learned from examples of pairs of input and output. A supervised learning algorithm can be arbitrarily complex, but overly complex models tend to overfit the data. This means the model generates the correct outputs for the training inputs, but does not generalize to unseen test inputs in a reasonable fashion. At its best, the learned function will be able to generate an output based on an input correctly, even with unseen instances. This requires the learning model to have the ability to generalize its "knowledge" from the examples reasonably. One choice of the learning model is a deep neural network. For image processing, deep neural networks have been shown to be very effective in specific tasks (e.g. image segmentation, image classification, image context understanding, etc...) given enough training examples and a proper

training process [14].

In sections 2.3, 2.4 and 2.5, we discussed ways to cut an affinity graph and achieve a segmentation, assuming that the affinity graph is defined and each affinity reflects how similar the adjoining vertices are. We now turn to the problem of generating an affinity graph for a given image. As mentioned at the end of section 2.2, the affinity between vertices i and j is a function of a neighborhood around the two vertices (equation 2.1). The function can be predefined for specific applications (e.g. affinities based on the ℓ_1-difference in the intensities, or affinities based on an edge detector [11], etc...). However, these predefined affinities often suffer from noise and hence the quality of segmentations obtained by cutting the affinity graph is not guaranteed. We are interested in using a deep neural network, especially ones that utilize convolutions, to produce affinities for images which result in good segmentations when the connected components procedure is applied to the resulting affinity graph. This has been previously shown to be effective in [28] and [29]. To do this, we need to decide on a neural network architecture, a learning objective, and a learning algorithm (i.e. the loss function). We continue to discuss these in some existing works on these subjects in the next sections.

2.7 A Learning Model: Neural Networks, Convolutional Layers, and the U-net

Much of this introduction to neural networks is based on [20]. In machine learning, a neural network is one among many classification algorithms. A basic unit in a neural network is a perceptron. A perceptron takes in several inputs $x_1, x_2, x_3, ...$, and produces a single output (figure 2.3). Just like a

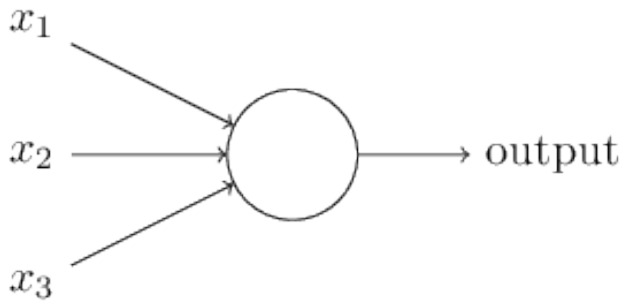

Figure 2.3
An illustration of a perceptron. Adapted from [20].

biological neuron, a perceptron can decide whether to "fire" or not to, given the inputs. A simple way

to model this behavior was proposed in [26]. The neuron's output is dependent on the weighted sum of the inputs $\sum_i w_i x_i$. The neural will "fire" if this sum exceeds a threshold. If we consider the output to be a binary variable, a mathematical model of a perceptron is:

$$\text{output} = \begin{cases} 0 & \sum_i w_i x_i \leq \text{threshold} \\ 1 & \sum_i w_i x_i > \text{threshold} \end{cases} \tag{2.6}$$

In practice, a single perceptron can be viewed as a binary classifier. Obviously, it is not complex enough to do anything other than binary classification. However, we can stack up many perceptron together and form a layer of perceptrons as well as stack up these layers a form a more complex network. Figure 2.4 is an illustration of a three layer neural network.

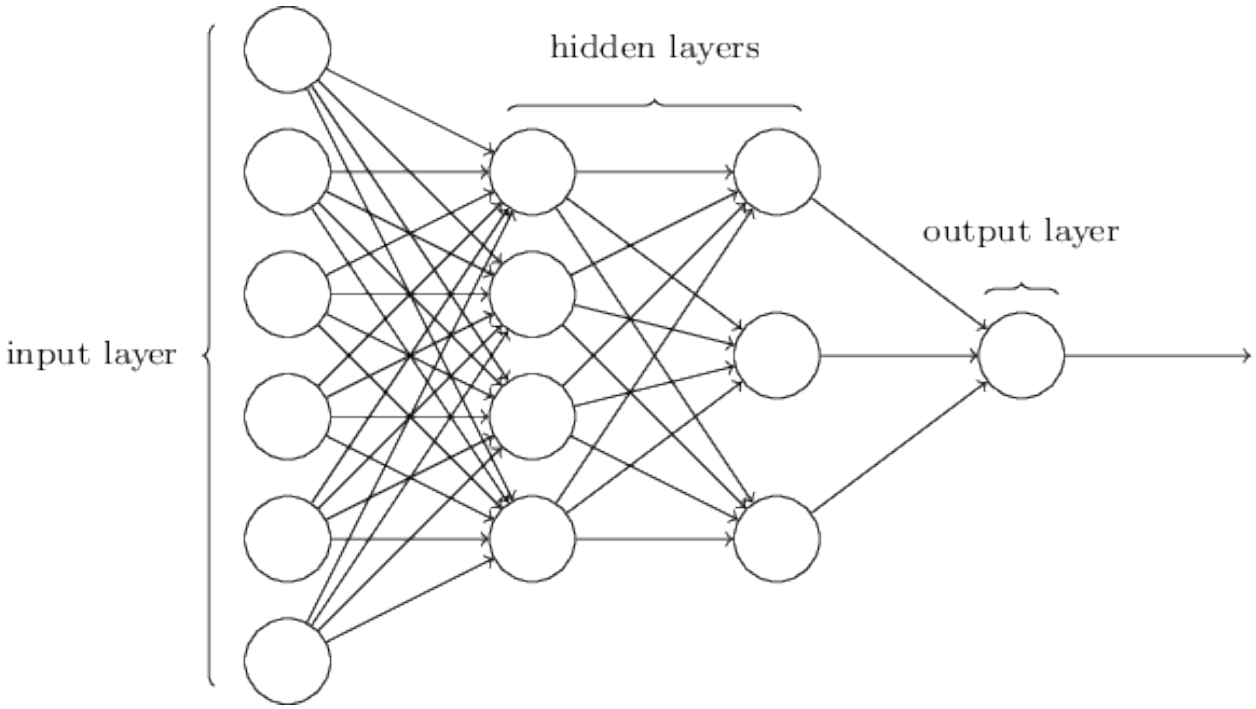

Figure 2.4
An illustration of a multi layers perceptron neural network. Adapted from [20].

Often, the output of a perceptron is paired with a smooth activation function - a differentiable version of the step function (2.6). This combination is usually referred to as sigmoid neurons due to the fact that the activation function is taken from the sigmoid family of functions. A sigmoid function is a bounded, differentiable, real function that is defined for all real input values and has a non-negative

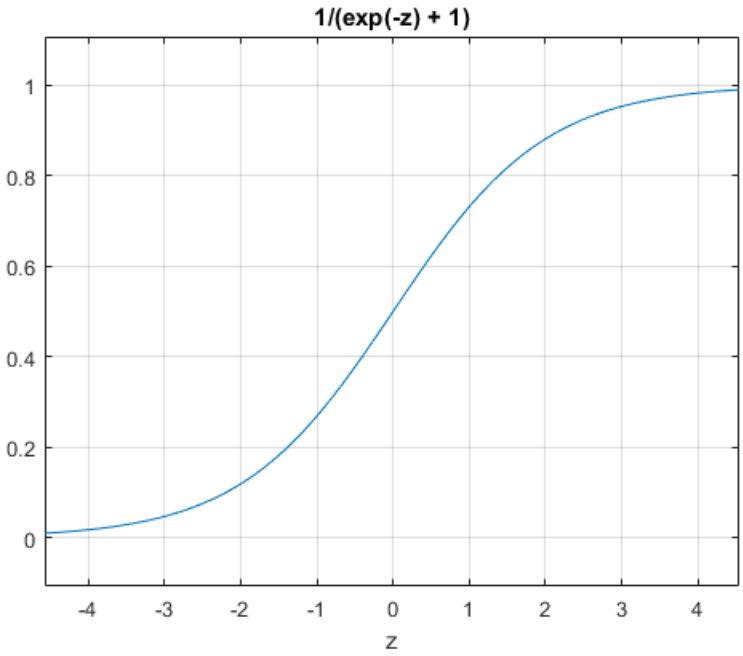

Figure 2.5
The logistic activation function, a smooth version of the binary step function that outputs {0,1}. Adapted from [20].

derivative at each point ([15]). An example of an activation function is the logistic function:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.7}$$

Then, with the adaptation to vector forms and the use of a bias instead of a threshold, the formulation of a sigmoid neuron is:

$$\text{output} = \sigma(\mathbf{w}^T\mathbf{x} + b) \tag{2.8}$$

The output of a sigmoid neuron is not binary, but any real number between the bounds (they are 0 and 1 in the logistic neuron example). One way to interpret this output is to look at it as the chance for the neuron to "fire". For example, an output of 0.851 means that there is 85.1% chance that this neuron will "fire". Roughly speaking, this activation function allows the algorithm to track what will happen to the output when a small change is made in any parameters w_i. Then, given a target and the output of the current state of the neural network with a smooth activation function, we can use a back-propagation algorithm to let the network adjust itself to produce "more" correct output when it takes some "similar" inputs again. Note that we use the word "similar" with the meaning that there

might be a small change in the inputs that the network has seen. In statistical learning, this process is called fitting a model, or learning a neural network in our context. In designing a neural net, it is common to test and compare as many combinations of number of layers, number of neurons per layer, and the activation functions as possible to determine the best network architecture for a problem. As the number of layers gets larger and the interaction between layers gets more complex, people usually call the network a "deep" neural net.

When it comes to image processing tasks such as image segmentation, image classification, or image filtering, a fully connected neural network (FCN), which describes an architecture in which only sigmoid neurons are used, is not recommended [17]. For example, if we feed an image of 100 pixels into a fully connected neural network that contains 100 neurons, each neuron will have 101 parameters (1 bias) and the total number of parameters is 10100. The point is that a model with a large number of parameters might be too powerful for a dataset, in the sense that it may have the ability to learn and express all the the data that it has been trained on while losing the ability to generalize to new instances [20]. This is called overfitting.

Convolutional neural networks (CNN) [17], which use the matrix convolution operation, have additional interesting properties, namely shift-invariance and space-invariance, along with a reduced number of parameters as compared to FCN. One of the very first convolutional networks was proposed in [17]. It was successfully applied to recognize hand-written ZIP codes from a U.S. Postal Service dataset. The most distinguishing feature of a convolutional layer as compared to an FCN layer is that the parameters are shared across the inputs.

For example, given an 1D input that contains 10 elements $x_1, ...x_{10}$, a 1D convolution neuron with filter of size 2 will have 3 parameters w_1, w_2, b. The output of this neuron will be

$$\text{output} = \sigma\left(\begin{bmatrix} w_1 \cdot x_1 + w_2 \cdot x_2 + b \\ w_1 \cdot x_2 + w_2 \cdot x_3 + b \\ \cdots \\ w_1 \cdot x_9 + w_2 \cdot x_{10} + b \end{bmatrix}\right), \tag{2.9}$$

where the transfer function σ is applied componentwise. In comparison, a fully-connected layer would have 11×10 parameters (i.e. 10 neurons with 11 parameters each).

An example of a single channel 2D convolution operation is illustrated in figure 2.6. In convolutional neural networks, the kernels of the convolution layers are variables that the network has to learn. Such a single-channel convolution produces a matrix of the same size or of a size reduced by the distance of the convolution map center to its edge. For a 2D convolution on a multichannel image (i.e. a 3D volume), each channel is individually convolved with a 2D kernel and then all the coefficients are added up with a bias and a transfer function is optinally applied to form a pixel value in the output of the operation. The result of such a multichannel convolution is a one-channel image. These multi-channel convolutions are used in deep neural networks. In practice, given a $H \times W \times C$ volume, to apply a 2D convolution on this, one must decide on the number of filters K, their spatial extent F (i.e. the size of each kernel), the stride S, and the amount of padding P. The convolution then produces a volume of size $H^* \times W^* \times C^*$, where $H^* = (H - F + 2P)/S + 1$, $W^* = (W - F + 2P)/S + 1$, and $C^* = K$. In a multilayer neural network, we call this a 2D convolution layer with K filters of size F by F, stride S, and P padding.

The transpose-convolution, also known as inverse-convolution or up-convolution ([9]), as its name suggest, it is defined as the inversion of a convolution. It maintains a one-to-many operation rather than a many-to-one operation. In [9], the authors describe the details and provide examples of this operation. In practice, the transpose-convolution is usually used in an upsampling layers of deep CNNs, especially ones that process images.

In CNNs, it is also common to have some pooling layers, which non-linearly down-sample the inputs. An example is the max pooling operation ([32]), which outputs the maximum element in a neighborhood. The most common form of the max pooling layer in modern deep network have kernel of size 2×2 and stride 2 (i.e. the window moves by 2 units in each step/direction), which reduces the size of each axis of the input by half, i.e. to dismisses 75% of the information of the input in a 2-dimensional image. There are several other types of the pooling operations, including average pooling, ℓ_2-norm pooling, and weighted average pooling. A typical setup for a CNN consists of at least one convolution layer, a non-linear activation function, and finally a pooling layer.

The U-net is a deep CNN that was designed for biomedical image segmentation, proposed in [25]. It learns to do segmentation in an end-to-end setting: it takes an input image in and produces the

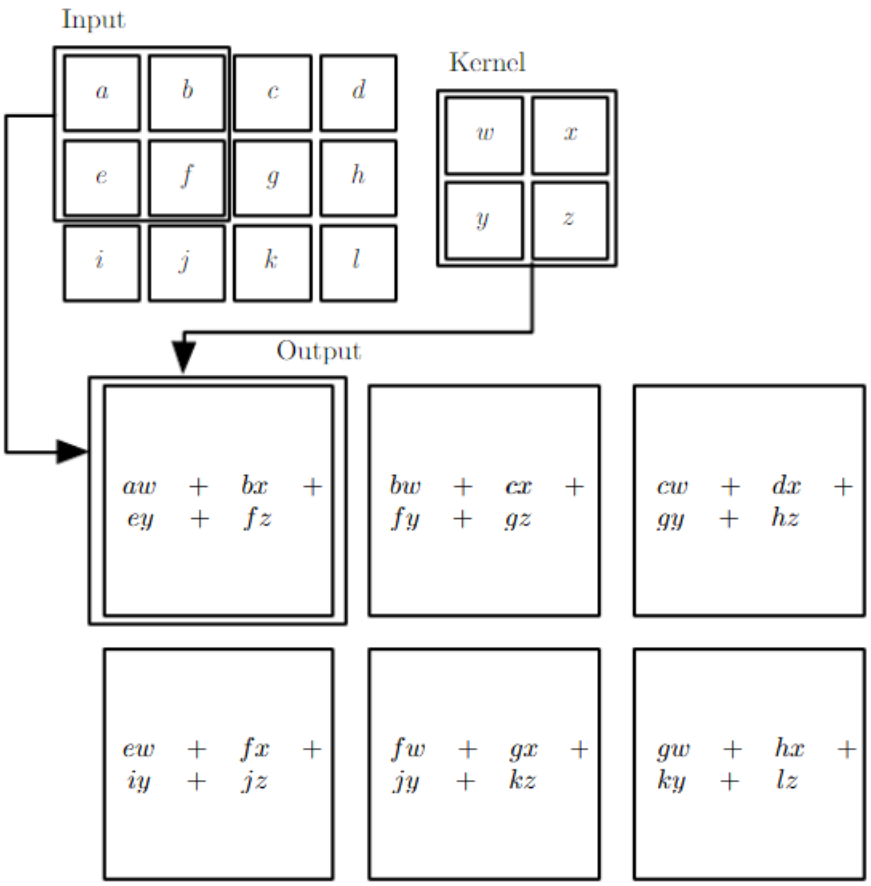

Figure 2.6
An example of a single channel 2D convolution in neural network with kernel of size 2×2, stride 1, where only the "valid" part of the input is used, (i.e. no padding to the input). The input is a 3×4 matrix and the output is a 2×2.. Adapted from [12].

corresponding segmentation output. This is quite different to what to what we are doing. We want the network to produce an affinity graph that will be cut by the connected components procedure (or other graph-cut techniques). The architecture of U-net is illustrated in Figure 2.7.

Similar to other deep CNNs, the U-net consists of several different operations, namely 2D-convolution, max pooling, and upsampling, cropping, and concatenation. Each operation is presented as a colored arrow. At the start, an image is fed into the network and propagated through all the arrows and at the end, an output segmentation map is formed. Most operations are convolutions with a 3 by 3 kernel followed by a non-linear activation function. One of the important design choice that the authors mention is that only the valid part of the input to the convolution is used, which means a 1-pixel border is

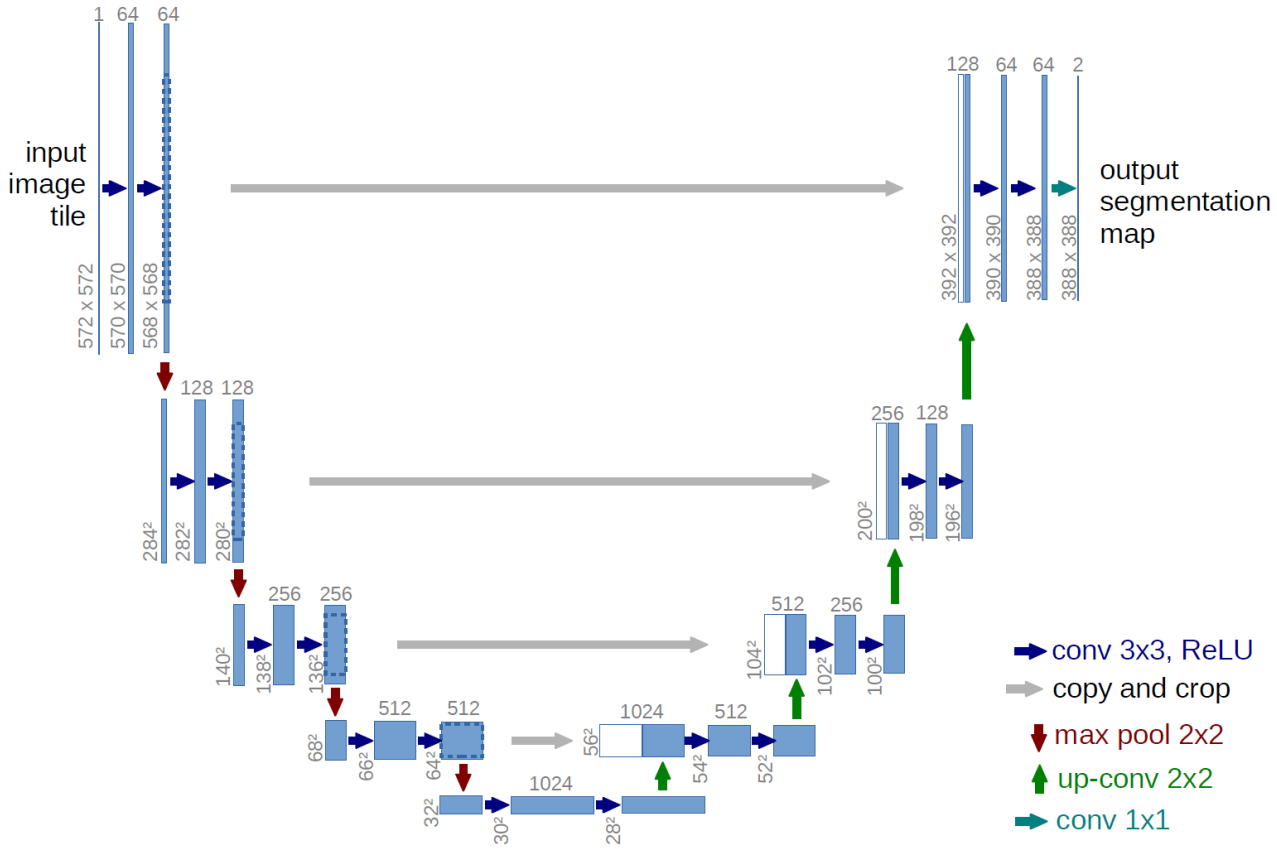

Figure 2.7
U-net architecture with input image of shape (572x572x1). Adapted from [25].

lost after each convolution operation (see Figure 2.6).

At a high level, the U-net consists of a "contractive" path, which consists of four "encoding" blocks, and an "expansive" path, which consists of four "decoding" blocks [25]. The "contractive" path is similar to the common set-up of a CNN. There are several 3×3 convolutions with the reLU activation function, all followed with max pooling layers. The reLU activation function is defined as

$$\text{reLU}(x) = x^+ = \max(0, x) \tag{2.10}$$

where x is a tensor, also known as a multidimensional array (e.g. a vector, a matrix, etc...) and the max operation is applied to each element of x. Along the "contractive" path, while the size of the meta-images (the tensors that are being propagated) is designed to decrease gradually due to the effects of valid convolution and max pooling layers, the number of channel increases with more and more number of kernels. In a symmetrical fashion, the "expansive" path does the exact opposite thing

as compared to the "contractive" path. The "expansive" path uses upsampling layers to increase the size and uses 1×1 convolution layers to reduce the number of channels of the meta-images from the results of the "contractive" path. Intuitively, along the "contractive" path, the network are learning more the "what" and less the "where" of the features of the input images, while along the "expansive" path, it learns how to infer from a low-resolution to a higher-resolution meta-images. Overall, the U-net takes in images and produces output segmentation masks of slightly smaller size than the input images.

In our implementation, we use a smaller scale variation of the U-net architecture described above. For our mini U-net model, we use one "encoding" block, one "decoding" block, and the center block. We use "same" padding option provided by Tensorflow so that the results of the convolutions have the same shape as the inputs. At the end, we add a 2D-convolution with two 3 by 3 filters and no activation function. Ultimately, the mini U-net we use takes a (meta) image and produces a set of affinity matrices whose shape is the same as one channel of the input image and whose elements are horizontal and vertical affinity predictions that represent a 4-connected lattice graph.

Assume the input tensors have the shape $H \times W \times C$:

- The encoding block starts with the size $H \times W \times C$ input tensors. It consists of two consecutive multichannel 2D convolution layers on with reLU activations (equation 2.10). Each convolution layer uses 16 filters of size 3 by 3 (i.e the shape of the kernels are $3 \times 3 \times C$), a stride of 1, and "same" padding. One copy of the output of these convolutions is passed to the decoder block. For the other copy, we apply a max pooling layer with a 2 by 2 filter and a stride of 2 and pass it to the center block. The encoding block passes two tensors to the latter blocks: one of shape $\frac{H}{2} \times \frac{W}{2} \times 16$ that is passed to the center block and one of shape $H \times W \times 16$ that is passed to the decoding block.

- The input to the center block is a size $\frac{H}{2} \times \frac{W}{2} \times 16$ tensor from the encoding block. The center block has two consecutive convolution layers but uses 32 filters and no max pooling layer follows. The center block produces an output of shape $\frac{H}{2} \times \frac{W}{2} \times 32$. This is passed to the decoding block.

- The decoding block has two inputs: one is the output from the convolutions of the encoding block (a $H \times W \times 16$ tensor) and one is the output from the center block (a $\frac{H}{2} \times \frac{W}{2} \times 32$ tensor). It first uses an upsampling layer (i.e. a deconvolution or transpose convolution layer) with 16 filters of size 2 by 2, stride of 2, and "same" padding on the input from the center block, which produces a tensor of shape $H \times W \times 16$. This is then concatenated with the input from the encoder block, producing a $H \times W \times 32$ tensor which goes through a reLU activation. This is then followed by two consecutive convolution layers as in the encoder block with 16 filters. Finally, the decoder block produces an output of shape $H \times W \times 16$ that is passed to the last layer.

- The last layer is a multichannel 2D convolution layer with 2 filters of size 1 by 1 (i.e. the shape of the kernels is $1 \times 1 \times 16$) with a linear activation function. This takes the input from the decoding block and produces the final output of shape $H \times W \times 2$. This last layer is equivalent to two independent weighted sums of all the channels of the input tensors, hence reduces the input tensor from 16 channels to 2 channels.

2.8 A Learning Objective: Rand Error for Segmentation Evaluation

The Rand Index was proposed in [24] as an objective criterion to evaluate clustering methods. It measures the similarity between two clusterings. For image segmentation, we use a variation of the Rand index, namely the Rand error (RE), which counts all differences between two segmentation over all possible pairs of vertices. It is defined as

$$RE(\widehat{Y},Y) = \binom{\|V\|}{2}^{-1} \sum_{i<j} I(\widehat{y_i} \neq \widehat{y_j}) I(y_i = y_j) + I(\widehat{y_i} = \widehat{y_j}) I(y_i \neq y_j), \tag{2.11}$$

where $\widehat{Y}$ and Y respectively, are the predicted and ground truth segmentation labels and V is the set of all vertices in the graph. Here, we use the same indicator function I as in equation 2.5.

Pairs of vertices with $I(\widehat{y_i} \neq \widehat{y_j}) I(y_i = y_j) = 1$ are false negatives, (i.e. vertices are predicted to be separated but in the ground truth are in the same segment). For false negatives, the prediction oversegments, resulting in too many segments. Pairs with $I(\widehat{y_i} = \widehat{y_j}) I(y_i \neq y_j) = 1$ are false positives (i.e. vertices are in different segments in the ground truth but are in the same segment in the

prediction). For a false positive, the prediction undersegments, resulting in too few segments. The Rand error adds the number of false positives and false negatives and is normalized by the number of vertex pairs, hence it takes a value between 0 and 1. $RE = 0$ means that the two segmentations are in total agreement; and, $RE = 1$ indicates that the two segmentations are in total disagreement, in other words, the two do not agree on any pair of vertices.

The Rand error is very dependent on the number of segments. If the ground truth has many segments, false positives are likely to happen, whereas if the ground truth has few segments, false negatives are the main source of error. Given that the graph is 4-connected, below are some extreme cases that highlight how the Rand error behaves:

- If the ground truth has one segment and the prediction also has one segment, meaning every vertex is in the same segment then there is no error and $RE = 0$.
- If the ground truth has one segment and the prediction has two segments with an equal numbers of vertices in each segment, then there are $\frac{\|V\|^2}{4}$ false negatives and no false positives. Note that this is the worst possible scenario for a ground truth with just one segment.
- If the ground truth has one segment and the prediction has $\|V\|$ different segments, meaning each vertex is in its own segment. Then the total number of false negatives is $\binom{\|V\|}{2}$, there is no false positive, and RE = 1.
- If the ground truth has $\|V\|$ segments and the prediction has one segment: there are $\binom{\|V\|}{2}$ false positives and no false negatives and RE = 1. This is equivalent to the previous case, since the Rand error is symmetric between ground truth and predicted segmentation.
- If the ground truth has $\|V\|$ segments and the prediction has two segments with equal numbers of vertices in each segment then there are $2 \times \binom{\frac{\|V\|}{2}}{2}$ false positives and no false negative.
- If the ground truth has $\|V\|$ segments and the prediction also has $\|V\|$ segments, then there is no error and RE = 0.
- If one vertex inside a ground truth segment of size k becomes its own segment in the prediction, then there are $(k-1)$ additional false negatives.

- If the label of one vertex at the boundary of two ground truth segments changes from one to the other in the prediction (from the label of segment of sizes k and to the label of segment of size m), then there are $k-1$ additional false negative (incorrectly split from segment k) and m additional false positive (incorrectly merge with segment m).

We can now move on to discuss an online stochastic gradient descent learning algorithm that allows a neural network to learn to produce affinity graphs whose segmentations by the connected component procedure (2.5 have a low Rand error.

2.9 A Learning Algorithm: Maximin Affinity Learning

It is very hard to optimize the Rand error directly for any assignment of affinities to edges due to the fact that we will have to consider all possible pairs of pixels in our image plus the issues arising in the transformation from an affinity map to a segmentation. In [28], the authors choose to relax the problem by considering maximin affinities of pairs of any two vertices. A maximin affinity between vertices i and j is defined as

$$A^*_{ij} = \max_{P \in P_{ij}} \min_{(k,l) \in P} A_{kl}, \tag{2.12}$$

where P_{ij} denotes the set of all the paths between vertices i and j. The connected components algorithm uses these maximin affinities to settle contradictions in the affinity graph, where one path may suggest to connect two vertices and another may suggest to separate the same pair. If the maximin affinity A^*_{ij} between two vertices exceeds a given threshold θ, then these two vertices are connected in the connected components segmentation, since there exists at least one path connecting vertices i and j whose minimum edge weight exceeds θ. This does allow for other paths between the two vertices that have a negative minimum weight. One can view that as favoring connecting over cutting, since no matter how strong the evidence along some paths for separating the vertices may be, one path with a large enough minimum weight will ensure that the vertices are connected. If we start with correlations as affinities, then the natural threshold would be zero, but through the maximin process small noise in the affinities can lead to mergers of a large number of vertices. Hence when using the connected

component procedure, zero might not be the best threshold to use. This observation brought up the idea of using inference in the training process.

If the connected components algorithm with threshold θ is used on a graph then the Rand error can then be written as

$$RE(\widehat{Y},Y;W) = \binom{\|V\|}{2}^{-1} \sum_{i<j} I(\widehat{y_i} \neq \widehat{y_j}) I(A^*_{ij} \geq \theta) + I(\widehat{y_i} = \widehat{y_j}) I(A^*_{ij} < \theta). \tag{2.13}$$

We note that we use the same indicator function I as seen earlier, which takes values in $\{0,1\}$ and $\widehat{y_i}$ denotes the predicted label of pixel i. We need to convert this definition to a continuous loss function. First, we observe that the difference between the characteristic functions $I(\widehat{y_i} = \widehat{y_j}) - I(A^*_{ij} \geq \theta)$ can take values in $\{-1,0,1\}$. It takes the value zero for a pair of vertices for which ground truth and connected component segmentation agree. For a false positive it takes the value -1 and for a false negative it takes the value $+1$. Hence the function $|I(\widehat{y_i} = \widehat{y_j}) - I(A^*_{ij} \geq \theta)|$ takes the value 1 for a segmentation error on pair (i,j) and for a correct segmentation this function takes the value 0. This allows us to rewrite equation 2.13 as

$$RE(\widehat{Y},Y;W,\theta) = \binom{\|V\|}{2}^{-1} \sum_{i<j} |I(\widehat{y_i} = \widehat{y_j}) - I(A^*_{ij} \geq \theta)| \tag{2.14}$$

This loss function has jump discontinuities which makes optimization by gradient descent impossible. To make the loss continuous and suitable for gradient learning, Equation 2.14 can be relaxed by using a continuous loss function such as the mean square loss or the hinge loss in place of $|I(\widehat{y_i} = \widehat{y_j}) - I(A^*_{ij} \geq \theta)|$ ([28]). The hinge loss for example uses the fact that the affinities A_{ij} lie in the interval $[-1,1]$ and the threshold θ is in the interval $[0,1]$. If we rescale the characteristic function for the ground truth from $\{0,1\}$ as the range to $\{-1,1\}$ by using $2I(\widehat{y_i} = \widehat{y_j}) - 1$ and use the actual difference between maximin affinity A^*_{ij} and threshold θ instead of the characteristic function $I(A^*_{ij} \geq \theta)$, then a positive product $\left(2I(\widehat{y_i} = \widehat{y_j}) - 1\right)(A^*_{ij} - \theta)$ would indicate a correct segmentation and a negative product would indicate an incorrect segmentation. The further the maximin affinity is from the threshold, the more negative this quantity gets.

$$1 - \left(2I(\widehat{y_i} = \widehat{y_j}) - 1\right)(A^*_{ij} - \theta) = \begin{cases} 1 - (A^*_{ij} - \theta) & \text{if } \widehat{y_i} = \widehat{y_j} \\ 1 - (\theta - A^*_{ij}) & \text{if } \widehat{y_i} \neq \widehat{y_j}. \end{cases}$$

This quantity equals 1 if the maximin affinity equals the threshold and is zero if the maximin affinity is one unit away from the threshold in the correct direction (i.e. above for $\widehat{y_i} = \widehat{y_j}$ and below for $\widehat{y_i} \neq \widehat{y_j}$). If the maximin affinity is on the wrong side of the threshold, then this quantity exceeds one. To avoid negative losses, we define the hinge loss below:

$$l(I(\widehat{y_i} = \widehat{y_j}), A^*_{ij}) = \max\left(0, 1 - \left(2I(\widehat{y_i} = \widehat{y_j}) - 1\right)\left(A^*_{ij} - \theta\right)\right) \tag{2.15}$$

Denote by $l(I(\widehat{y_i} = \widehat{y_j}), A^*_{ij})$ be such a continuous loss function that approximates our loss function. With this hinge loss, we obtain a continuous approximation to the Rand error:

$$REA(\widehat{Y}, Y; W, \theta) = \binom{\|V\|}{2}^{-1} \sum_{i<j} l(I(\widehat{y_i} = \widehat{y_j}), A^*_{ij}) \tag{2.16}$$

This Rand error loss function is the loss function we want to minimize by learning affinities A^*_{ij}.

In Turaga's paper [28], online stochastic gradient learning is used to learn the affinities. In each iteration of the learning process, a random pair of pixels is chosen, the maximin affinity between them is computed as a function of the parameters in the CNN. The gradient of the loss function $l(I(\widehat{y_i} = \widehat{y_j}), A^*_{ij})$ with respect to all the parameters is then computed in order to minimize the loss function. If W denotes all the parameters, then for each iteration, the gradient descent updates them in the following way, which involves finding the maximin edge between vertices i and j:

$$W \leftarrow W - \eta \frac{\partial}{\partial W} l(I(\widehat{y_i} = \widehat{y_j}), A^*_{ij}). \tag{2.17}$$

This online learning method is presented by Turaga as a trade-off between speed and performance of the classifier. It takes $\mathcal{O}(\|V\| \log \|E\|)$ to find the maximin affinity between a pair of vertices and estimate their gradient. However, the number of pairs is quadratic, which would require multiple iteration of this learning process to cover all possible pairs of vertices.

2.10 Inferning

"Inferning" is a name used in a number of machine learning workshops for research that focuses on the intersection of inference (sections 2.2 - 2.5) and learning (sections 2.6 - 2.9). Inference algorithms are those that use a model to make some predictions about the input data and learning algorithms are

those that estimate the parameters of a model of the data. Understanding the interactions between the two is of theoretical and practical interest to advance the state of the art on various tasks such as computer vision, natural language processing, etc. In general, there are two directions: how the choice of an inference method affects the training process of a model and how the learning objective (that motivates inference algorithms) impacts the quality of the final prediction.

In our context, inference focuses on the optimization of an objective function (e.g. an energy function) of affinities between adjoining vertices. Graph inference methods are well defined and offer nice theories and analysis (including running time and error analysis). However, a noisy affinity graph (where lots of confusions exist) would decrease the performance of a graph-based inference methods badly.

Recent developments in deep learning have greatly pushed the performance of computer vision systems and learning frameworks in various tasks, including segmentation. The U-net mentioned in section 2.7 is an example of an end-to-end model which achieves very good performance on biomedical image segmentation tasks. Even though end-to-end models such as U-net can work very well for some specific tasks, it is still a challenge to develop theories and uncertainty analysis for them. In our context, we refer to using deep neural networks to learn a specific task (e.g. producing affinities) as a learning approach.

Inference and learning are complementary tasks, and can be used in combination to improve performance. Learning is typically an offline process, using large labeled training sets. Inference is typically online and can help models adapt to the data at hand. Inferning refers to the situation where we have learning with inference in the loop. As we have described, when considered independently, learning and inference typically involve NP-hard optimization problems. In combination, we have two NP-hard problems to contend with (motivating the title inferning), but this also opens up opportunities for new methods that trade off performance between the two problems in particular applications, which motivated this thesis.

Both inference and learning can be efficiently done and provide good results on various computer vision problems, under their favorable conditions. Inference in graphs requires accurate affinities in order to perform well. The corresponding affinity graph of a given image is designed for specific

applications, depending on the to-be-used inference method, the type of the input image, preferences for probable solutions (e.g. do we prefer to over-segment or under-segment?), etc... Traditionally, the functions that map images to affinity graphs are simple. That is, they only compute an affinity by looking at the two adjoining vertices (e.g. absolute difference in intensities, Gaussian weighting functions, etc...) or a very small neighborhood around the two vertices (e.g. an edge filter uses a 3 by 3 neighborhood). Hence, these functions take little or no advantage of spatial information, which is crucial in image analysis and processing. They also lack the ability to handle noise. Unlike vision systems with inference approaches, systems with end-to-end learning approaches can handle almost everything (e.g. noisy complicated images), granted enough training examples and well-defined learning objectives. The biggest reason that makes people hesitate to use these systems is that there are not enough theorems and error estimates to guarantee their performance or to bound their errors.

METHODOLOGY

3.1 Learning with Rand Error and Kruskal's Algorithm

While the original maximin procedure for the connected component segmentation as described in [28] is done on the whole graph (generally a 4-connected graph of the image), an equivalent result could be achieved with a maximum spanning tree (MST) of the original weighted graph. This is based on the fact that the maximin affinity between two vertices is an MST edge. This fact suggests that we can leverage Kruskal's algorithm to efficiently compute the Rand Error for a connected components segmentation.

Kruskal's algorithm [16] is a greedy algorithm that finds a maximum spanning tree (MST) for a connected weighted graph. It first sorts the weighted edges in decreasing order. In each step, the edge of highest weight that does not create a cycle in the tree is added to the spanning tree. Kruskal's algorithm can be efficiently implemented using the Disjoint Set, or Union-Find, data structure. This data structure keeps track of a set of elements partitioned into a number of disjoint subsets. It maintains a collection $\mathbf{Q}$ of disjoint sets Q_i. A unique element, called the canonical element, represents a set Q_i it belongs to. For example, in our case, each subset Q_i represents a segment and has vertices as its elements; the union of all subsets Q_i makes up the whole set of vertices. There are three important operations:

- MakeSet(v_i): make a new set Q_j that contains v_i and add it to the collection $\mathbf{Q}$, given that v_i is not already an element of $\mathbf{Q}$.
- FindCanonical(v_i): find and return the canonical element of the set containing v_i.
- Union(v_i, v_j): given that v_i and v_j are in different subsets, Q_i contains v_i and Q_j contains v_j, and they are not necessary the canonical elements, then in this operation the two sets Q_i and Q_j will be removed from the collection $\mathbf{Q}$, the union $Q_k = Q_i \cup Q_j$ will be added to the collection $\mathbf{Q}$ and a new canonical element is selected in Q_k.

In Kruskal's algorithm, if vertices u and v are already in the same component (i.e. in the same subset Q_i of $\mathbf{Q}$), then connecting vertices u and v would create a cycle. To avoid cycles, we initialize the collection of component sets $\mathbf{Q}$ as a collection of one-element sets for each vertex in the graph and merge components Q_i when an edge is added. This is the MakeSet(v_i) operation. Each component has one representative vertex, found by FindCanonical(v_i). When two components merge, one of the representatives is chosen as the representative for the new component through the Union(v_i, v_j) operation. Hence, to check whether two vertices u and v are in the same segment, we check if the representative vertices of the components that u and v belong to are the same. Comparing FindCanonical(u) and FindCanonical(v) will fulfill this.

Algorithm 2, proposed in [23], computes the Rand error of a connected components segmentation through building an MST. The algorithm takes a weighted graph $G(V,E,A)$ and outputs the list of the MST edges and the counts #same and #diff at each time an edge is added to the MST. Here, growing an MST can be viewed as growing connected components. At the start each vertex is its own component and with each added edge (connecting two vertices), two components are merged until all vertices are in the same component.

Definition 3.1.1. To each edge that is added to the MST, we record two counts #same and #diff: #same is the number of additional true positives (i.e. correctly merged vertex pairs) when an edge is added to the MST (i.e. two components get connected); #diff is the number of additional false positives (i.e. falsely merged vertex pairs) when an edge is added to the MST.

After running algorithm 2, we obtain a matrix with rows for each edge added into the MST and two columns containing #same and #diff for those edges. These #same and #diff counts are computed from a membership array for each segment. This membership array is a vector of length equal to the total number of segments in the ground truth. At each iteration in the Kruskal's algorithm, a membership array is assigned to each Union-Find set (labels[i] in 2). These arrays count how many vertices in a connected component Q_i are in each ground truth segment. For example, for a ground truth that has 3 segments, the array $[2,0,3]$ associated with a set Q_i of five vertices let us know that there are two of them in the first ground truth segment, none in the second, and three in the third.

Assume the ground truth has S segments and there are N vertices. Initially, when each vertex is in its own set Q_i, there are N different arrays of size S, one for each set Q_i. Such an initial array has a value of one at the i-th index if its associated vertex is in the i-th ground truth segment and zeros at all other indices. Following Kruskal's Algorithm, when an edge is added to the MST, two sets Q_i and Q_j are merged and a new membership array will represent this merged set.

Lemma 3.1.1. *When two sets Q_i and Q_j are merged, the new membership array is formed by an element-wise addition of the two arrays associated with the two sets Q_i and Q_j.*

Proof. This follows immediately from the definition of the membership array as a counter of how many elements of each segment the set contains. □

For example, in the case of three ground truth segments, let $[1,0,0]$ and $[1,1,0]$ be the arrays associated with two merging sets $\{i\}$ and $\{j,k\}$. The new array associated with the merged set $\{i,j,k\}$ is $[2,1,0]$. Observe that by connecting the two sets $\{i\}$ and $\{j,k\}$, vertex i is connected with one vertex that is in the same ground truth segment and another vertex from a different ground truth segment.

Lemma 3.1.2. *When connecting two segments Q_i and Q_j with associated membership arrays m_i and m_j are merged, we get*

$$\#same = m_i \bullet m_j,$$

where $\bullet$ is the dot product, and

$$\#diff = |m_i| \cdot |m_j| - \#same,$$

where $|m_i|$ is the sum of all elements of m_i.

Proof. When merging two component sets to form a larger component, the element-wise product of the two associated arrays gives the numbers of label agreements for each ground truth segment at each index, hence, the dot product of the two arrays gives the total number of label agreements at each iteration of the Kruskal's algorithm. As well, since the total number of new connections made at each iteration can be computed as the product of the ℓ_1-norms of the two associated arrays and would be $1 \times 2 = 2$ in the example above. Hence, the number of label disagreements can be easily computed as the different between total new connections and the number of label agreements. □

Now we have defined all the terms that are used in the modified Kruskal's algorithm (algorithm 2) and can state the pseudo-code for the algorithm. The algorithm plays a crucial role in our formulation of the training loss functions that minimize the Rand error. This will be described in the next section, section 3.2.

Data: A weighted graph $G(V,E,A)$ and the ground truth label for each vertex (i.e. a set of vertex-label pairs of all vertices

Result: An ordered list of MST edges, the counts #same and #diff label pairs at each threshold level, corresponding to an MST edge

Collection **Q** is initialized to Ø;
Set E is sorted in decreasing order by weights;
$idx \leftarrow 0$;
for *each* $x_i \in V$ **do**
    MakeSet(x_i);
    labels[i] is initialized to a zero array of length that equals the number of different labels;
    labels[i][y_i] $\leftarrow 1$
end
for *each edge* $(u,v) \in E$ **do**
    $c_u \leftarrow$ FindCanonical(u);
    $c_v \leftarrow$ FindCanonical(v);
    if $c_u \neq c_v$ **then**
        Union(c_u, c_v);
        $c_{new} \leftarrow$ FindCanonical(c_u);
        MST[idx] $\leftarrow (u,v)$;
        #same[idx] $\leftarrow$ labels[c_u]$^T \cdot$ labels[c_v];
        #diff[idx] $\leftarrow \|$labels[c_u]$\|_1 \cdot \|$labels[c_v]$\|_1$ - #same[idx];
        labels[c_{new}] $\leftarrow$ labels[c_u] + labels[c_v];
        $idx \leftarrow idx + 1$;
    end
end

Algorithm 2: Playing with Kruskal's Algorithm. Adapted from [23].

3.2 Loss functions for Rand Error and Kruskal's algorithm

If we have a list of MST edges found by Kruskal's algorithm, we get a hierarchy of segmentations. It starts with the segmentation in which each vertex is its own component and then, as edges are added to the MST, vertices are joined to form bigger and bigger path-connected segments until all vertices are in the same component (or segment). The threshold for the MST edges determines which segmentation is chosen: all MST edges with weights above the threshold are joined and all edges

with weights below the threshold are cut. The Rand error will vary with each chosen threshold. If each edge in the MST has a unique weight, then the total number of different threshold level equals the number of vertices, $\|V\|$. In other words, for a given affinity graph and the connected component procedure as described in section 2.5, the MST can be segmented in at most $\|V\|$ different ways, which leads to up to $\|V\|$ different Rand errors. For each threshold except the highest one, there is a a corresponding pair of #same and #diff counts. The matrix with columns #same and #diff generated in Kruskal's algorithm is used to calculate the Rand error. Algorithm 3 describes this computation in pseudo-code. We first describe the main ideas with some examples:

- If a threshold level greater than the largest edge weight (the largest affinity) is picked for the connected component procedure, then all the MST edges will be cut and each vertex forms its own segment. In this case, every pair of vertices in the same ground truth segment is incorrectly split and hence the largest possible number of false negatives for the Rand error occurred. On the other hand, every pair of vertices that are in different ground truth segments is correctly split and hence no false positives occurred. The #same array contains the counts of all pairs whose ground truth labels are the same. In this case when no pairs of vertices is connected, the sum of all elements in the #same array is the number of false negatives. Note that a false positive is an error made by incorrectly joining vertices that have different ground truth labels and a false negative describes an error made by incorrectly splitting edges between vertices that have the same ground truth labels.

- Consider the threshold level that is between the two largest affinity weights. When thresholding at that value, all the edges in the MST are cut except the edge that has the largest weight. In other words, it is predicted that the two vertices connected by the largest MST edge, are in the same segment while each of the other vertices is in its own segment. There is two cases, whether or not the two connected vertices are in the same ground truth segment:

  - In the case when the two vertices are in the same ground truth segment, we have one less false negative. No false positives are added. In this case, #same$[0] = 1$ and #diff$[0] = 0$. Here, index 0 indicates the first MST edge in algorithm 2.

- If the two vertices are in different ground truth segments, we add a false positive. The number of false negatives doesn't change. In this case, #same$[0] = 0$ and #diff$[0] = 1$.

The idea is to use the #same$[i]$ and #diff$[i]$ counts of the i-th MST edge to iteratively compute the number of false positives and the number of false negatives (i.e. numbers of incorrectly merged and incorrectly split pairs) for each segmentation on the chain of segmentations given by the MST. We can accumulate the new error counts to the total false positives and false negatives from the first threshold level to get the total errors at this second threshold level. For the first case above, we get the new positive error by subtracting the #same$[0]$ count from the previous total false positives while we get the new negative error by adding #diff$[0]$ to the previous total false negatives. This process of accumulation gets repeated and efficiently provides the total numbers of pairwise misclassifications at each threshold level.

- At the last threshold level that is smaller than all the MST edge affinity weights, all the edges in the MST are kept and it is predicted that every vertex is in the same segment. All the pair of vertices that are in the same ground truth segment are correctly merged (i.e. we have zero false negatives) while all the pair of vertices that is in different ground truth segment are incorrectly merged (i.e. largest false positive, which equals the sum of all elements in the #diff array). At this point, the process described above has iteratively subtracted all the #same counts and added all the #diff counts for each new MST edge.

Lemma 3.2.1. *The largest possible number of false negatives is given by* $\|\#same\|_1$ *and the largest possible number of false positives is given by* $\|\#diff\|_1$.

Proof. By definition, each #same count equals the number of additional true positives when an MST edge is connected, hence, the sum of all these counts gives the largest number of true positives when all the MST edges are kept in the connected components procedure. Oppositely, if no MST edge is kept and each vertex is in its own segment, this total is interpreted as the largest number of false negatives. Likewise, by definition each #diff count equals the number of additional false positives when an MST edge is connected, hence, the sum of all these counts gives the largest number of false positives when all the MST edges are kept in the connected components. □

Data: A given threshold T; the ordered list of MST edges, the #same, and #diff arrays given by algorithm 2

Result: The Rand error of the segmentation using connected components procedure with the given threshold

totalPos $\leftarrow \|\#\text{same}\|_1$;
totalNeg $\leftarrow \|\#\text{diff}\|_1$;
FN $\leftarrow$ totalPos;
FP $\leftarrow 0.0$;
$RE \leftarrow (\text{FN} + \text{FP})/(\text{totalPos} + \text{totalNeg})$;
for $idx \leftarrow 0$ ***to*** $length(\#same)$ ***step*** 1 **do**
    if $T > MST[idx]$ **then**
        return RE
    else
        FN $\leftarrow$ FN - #same[idx];
        FP $\leftarrow$ FP + #diff[idx];
        $RE \leftarrow (\text{FN} + \text{FP})/(\text{totalPos} + \text{totalNeg})$;
    end
end
return RE

Algorithm 3: The Rand error calculation for a fixed threshold T. Adapted from [23].

In short, the #same and #diff contain counts of connections and separations associated with each MST edge. Algorithm 2 provides a mapping between the MST edges and the false positives and false negatives counts for the associated segmentations. It also computes how much each MST edge contributes to the Rand error. Specifically, for each MST edge, if the corresponding #same count is greater than #diff count, want to keep the affinity weight of this edge above the threshold so that we get a lower Rand error.

This suggests using the whole MST as a natural batch for learning affinity. The MST, the #same and #diff counts can be computed very efficiently through the Kruskal's algorithm, namely, with complexity $\mathcal{O}(\|E\| \log \|E\|) = \mathcal{O}(\|E\| \log \|V\|)$ when the affinity graph is lattice. Hence we are training a classifier by looking at all the MST edges in each iteration instead of just one MST edge that was the maximin edge for a random pair of pixels as Turaga et. al. did. Training with Kruskal's modified algorithm suggests advantages in training speed when scaling to larger data. Our minibatch consists of all MST edges and we use a gradient descent that considers all summands in equation 2.16.

The actual implementation of the mini-batch learning has a few more details. An important observation is that we need the whole MST to calculate the largest number of false positives and false

negatives as described in Lemma 3.2.1 to calculate the Rand error for thersholding the maximin affinities at zero. Once we have done this calculation, we can just as easily calculate use algorithm 3 to compute the Rand error for all thresholds, not just zero. This was the key idea to introduce inferning, the ability to then chose the best of all thresholds for a test image based on the threshold that minimizes the correlation clustering energy function.

In [23], a meta label for each MST edge (u,v) is defined by setting

$$l_{uv} = sign(\#\text{same}[idx] - \#\text{diff}[idx]), \tag{3.18}$$

where each index idx is associated with an MST edge (u,v). This label indicates whether an MST edge increases or decreases the Rand error. The #same and #diff counts are also used to calculate a weight vector that represents the relative contribution of each MST edge to the total Rand error:

$$W_{uv} = \frac{1}{\sum_{(i,j)\in MST} W_{ij}}(\#\text{same}[idx] - \#\text{diff}[idx]) \tag{3.19}$$

The weighted hinge loss with meta label l_{uv} and weights W_{uv} can then be written as:

$$l(A,Y) = \sum_{(u,v)\in MST} W_{uv} \cdot max(0, 1 - l_{uv}(A_{uv} - \theta)) \tag{3.20}$$

To compare with the loss function in section 2.9, the meta label is used in place of the term $(2I\left(\widehat{y}_i = \widehat{y}_j\right) - 1)$ in the hinge loss function in equation 2.15. This new label does not just consider whether the two vertices adjoining the MST edge under consideration belong to the same segment, but considers the effect of that edge on the whole graph. The other difference is that this loss function for a given ground truth and affinity does not consider all edges but just the MST edges.

This meta label allows us to quickly predict if an MST edge should have its weight above the threshold θ based on its contribution to the Rand error, with respect to the connected components procedure:

- An MST edge is better kept (i.e. its meta label is 1) if its #same count is larger than its #diff count because keeping this edge will decrease the final Rand error.
- An MST edge should ideally be cut (i.e. its meta label is -1) if its #same count is smaller than its #diff count, because discarding this edge will decrease the final Rand error.

- When the #same and #diff counts of an MST edge are the same, the corresponding weight is zero by definition (equation 3.19) so the edge does not contribute to the final loss.

The way the meta labels l_{uv} and the weights W_{uv} are assigned determines the nature of the model. A change in either of these two defines a different loss function, hence a different optimization process and a different optimal solution that minimizes the loss function. The formulation of the meta label and the weight above might encounter some problems:

- First, since the loss in equation 3.20 is defined solely on the difference between the #same and #diff counts, it does not consider the proportional relation of the counts contributed by an MST edge over the total counts. The MST edges of that are first added to the MST and have the largest affinities will have small #same and #diff counts since they are connecting small segments together and hence only small numbers of new connected pairs are formed. Consequently, the difference between the #same and #diff counts of each of the top MST edges has smaller range as compared to those MST edges that get added at the end of algorithm 2. For example, if the #same and #diff counts of one of the top MST edges and one of the bottom MST edges is 1,0 and 5001, 5000 respectively, then the two edges contribute the same in the loss function (equation 3.20). This is undesirable, since the former MST edge should weigh more than the latter, because if one of the top MST edge is misclassified, the attribution of following MST will have to share the consequences.

- Second, the formulation opens the door for solutions that minimize the loss function by arranging the MST edges in a way where each associated pair of #same and #diff counts cancel out, leading to a weight of zero and a meta label of zero for each MST edge. This creates situations where the loss is decreasing while the Rand error, which is the main objective, is not and might even be increasing.

To combat these issues, we attempt two approaches: (1) modifying the the loss functions presented above, and (2) going for a complete different formulation. Specifically,

1. To address the above problems by modifying the presented loss function, we change the weight of each MST edge to the proportion of the absolute value of difference #same $-$ #diff for that particular edge to the total counts of pairs connected by that edge:

$$\text{diff-}W_{uv} = \left| \frac{\#\text{same}[idx] - \#\text{diff}[idx]}{\#\text{same}[idx] + \#\text{diff}[idx]} \right| \tag{3.21}$$

Here, we still use the meta label defined in equation 3.18. When using this meta label with the weight defined in equation 3.21, we denote it as diff-l_{uv}. This directly fixes the first problem by introducing the total counts of each MST edge. The weights are better distributed between the largest and smallest MST edges. From here, to prevent the canceling effect, we can optionally add a tolerance parameter δ to equation 3.21, where δ can be any function of the MST edge index idx. With equation 3.21, we enhance the exisiting framework proposed in [23].

2. The other way of approaching this is to always force the classifier to consider flipping the affinity of an MST edge about the threshold θ. That is, if an MST edge is predicted to have its affinity greater than the threshold θ, we assign to it a meta label of -1; oppositely, if an MST edge is predicted to have its affinity smaller than the threshold θ, we assign to it a meta label of 1:

$$\text{hit-}l_{uv} = \begin{cases} 1 & \text{if } \widehat{A}_{uv} = \theta \\ -sign(\widehat{A}_{uv} - \theta) & \text{otherwise} \end{cases} \tag{3.22}$$

The weight associate with each MST edge is defined as:

$$\text{hit-}W_{uv} = \begin{cases} \frac{\#\text{diff}[idx]}{\#\text{same}[idx]+\#\text{diff}[idx]} & \text{if } \widehat{A}_{uv} \geq \theta \\ \frac{\#\text{same}[idx]}{\#\text{same}[idx]+\#\text{diff}[idx]} & \text{if } \widehat{A}_{uv} < \theta \end{cases} \tag{3.23}$$

In the perfect scenario, all the MST edges whose weights are larger than θ correctly connect vertices inside ground truth segments together and do not create any false positive when joining vertices from different segments. That is, in this case, if we discard all others MST edges whose weights are smaller than θ, we have a correct segmentation of the image (i.e. zero Rand error). At the same time, when applying algorithm 2 to such an MST, all the MST edges whose affinities exceed θ will have zero #diff counts because they only contribute to the number of

true positives (i.e. they only have #same counts); conversely, all the MST edges whose affinities are below θ will have zero #same counts because they only contribute to the number of false positives (i.e. they only have #diff counts). The weights defined for each MST edge (equation. 3.23) will then be all zeros as well, leading to a zero loss. We think gradient descent would struggle to optimize this loss, but eventually, the loss should train our model to produce the perfect scenario described above.

The loss function in 3.20 should tailor the affinities for the graph to be centered at θ. When using the hinge loss with $\theta = 0$, an edge whose weight lies outside the interval centered at θ with radius one and have the same sign as the label does not get penalized (see figures 3.8 to 3.11). For example, in equation 3.18, if $A_{uv} - \theta \geq 1$ and $l_{uv} = 1$, then the edge (u,v) has zero contribution to the loss function; likewise, if $A_{uv} - \theta \leq 1$ and $l_{uv} = -1$, then the edge (u,v) also has zero contribution.

With the revised definitions of the loss function discussed above, we want the classifier to learn to raise the affinities of all the MST edges that should be kept, as well as reduce the affinities of those edges that should be discarded in order to decrease the Rand error. The value of the parameter θ plays an important role in guiding the classifier. In practice, to make a connection to the correlation clustering in the literature, the value of θ is can be set at zero while the values of affinity weights can optionally be limited in [-1,1]. The output affinity weights of the classifier can then ideally be interpreted as correlations between neighboring pixels.

3.3 Inferning Connected Components for Segmentation

The frameworks presented in [28] and [23] focus on learning to produce ideal affinity graphs such that if we segment these graphs with the connected component procedure (using a fixed threshold θ, which is zero in our setup), we will have a good segmentation (measured by Rand error with respect to a ground truth). Note that there are learning (to produce affinities) and inference (to use the produced affinities) acting in these frameworks. Increasing the complexity of the learning architecture (i.e. making deeper neural networks/classifiers) will certainly give more flexibility to the frameworks and hence a performance gain is possible but no extra insight is acquired. This is not as fascinating

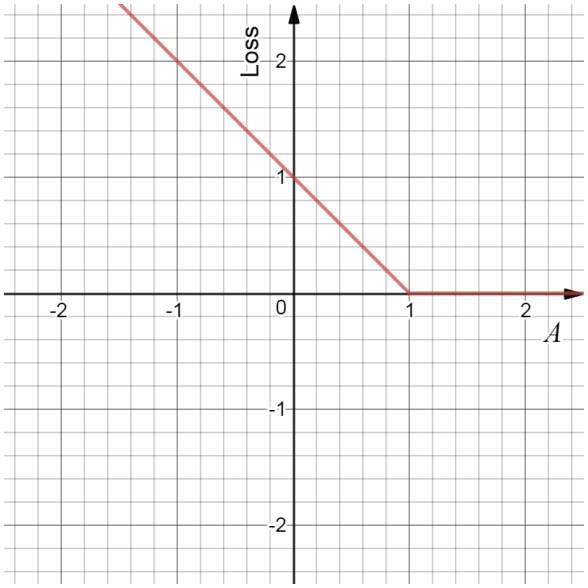

Figure 3.8
The hinge loss vs. affinity when $l_{uv} = 1$ and $\theta = 0$.

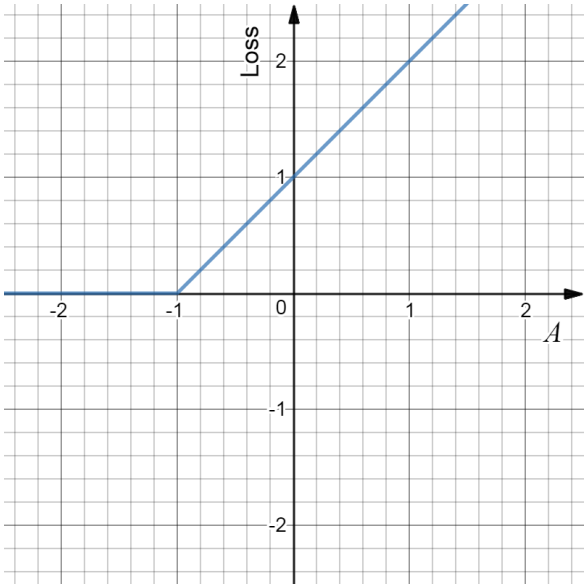

Figure 3.9
The hinge loss vs. affinity when $l_{uv} = -1$ and $\theta = 0$.

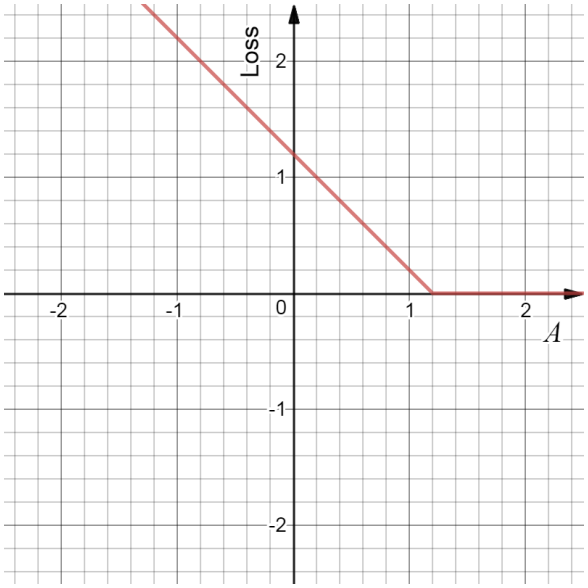

Figure 3.10
The hinge loss vs. affinity when $l_{uv} = 1$ and $\theta = 0.2$.

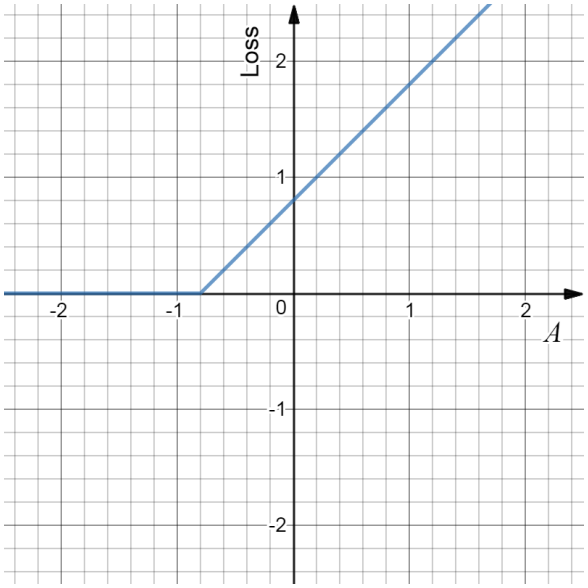

Figure 3.11
The hinge loss vs. affinity when $l_{uv} = -1$ and $\theta = 0.2$.

as to make the inference step better. Specifically, we want to instead vary the threshold θ used in the connected component procedure. Instead of fixing it at zero, we want to use the best threshold that gives the lowest correlation clustering energy defined in equation 2.5 for each given affinity graph. We hope that by doing this, we not only see a performance gain, but also see that the learned affinities can be related to correlations between adjoining vertices. Note that given a segmentation of a graph, the correlation clustering energy is calculated as the sum of affinities of all adjacent pairs of vertices that have different labels. The optimal segmentation is the minimization of the correlation clustering energy. In other words, the best segmentation by correlation clustering energy is the one that minimizes the "disagreements" between adjoining vertices.

A by-product of the modified Kruskal's algorithm (algorithm 2) is the information of vertices that get connected as the MST is built (or equivalently, as the threshold decreases, when segments are grown by the connected component procedure with decreasing thresholds). Let S be the sum of all affinities of the graph. This is the correlation clustering energy of a completely disjoint graph in which every vertex is its own segment. Then, the correlation clustering energy of the connected component segmenation at each threshold level of the MST can then be computed by subtracting from S the affinities between vertices that are in the same component. Note that the edges of theses affinities are not necessary on the MST. We can add a subroutine to algorithm 2 to make use of the by-product mentioned above and efficiently compute the correlation clustering energy for all possible threshold levels. Picking the best threshold for the connected component procedure is then a line search over all correlation clustering energies for different thresholds.

Each time an edge is added to the MST, two segments are connected. The modified algorithm incrementally builds a pointer cycle around all the vertices in a segment. As the MST is built, when two segments are merged, this pointer efficiently iterates through all the vertices of one segment and checks for the ones that are adjoining vertices of the other segment. The affinities between these pairs of vertices are subtracted from the previous correlation clustering energy S to compute the value of the correlation clustering energy for the lower threshold.

The threshold that gives the minimum correlation clustering energy, here denoted by T is used to compute the hinge loss (equation 3.20). Examples of how this threshold affects the loss function are

listed below.

- Assume, for example, $T = 0.2$ gives the minimum correlation clustering energy. Equation 3.20 then penalizes all MST edges that have $+1$ meta-labels since their affinities are below 1.2 and forces them to increase, while it also penalizes all MST edges that have -1 meta-labels and affinities above -0.8, forcing them to decrease. As compared with using a threshold at zero, this loss function does not penalize the MST edges that are weighted between -1 and -0.8 even if these edges have $+1$ meta-labels.

- If $T = -0.2$ gives the minimum correlation clustering energy, then Equation 3.20 does exactly the opposite. The MST edges whose affinities are between 0.8 and 1 do not get penalized even if their meta-labels are -1.

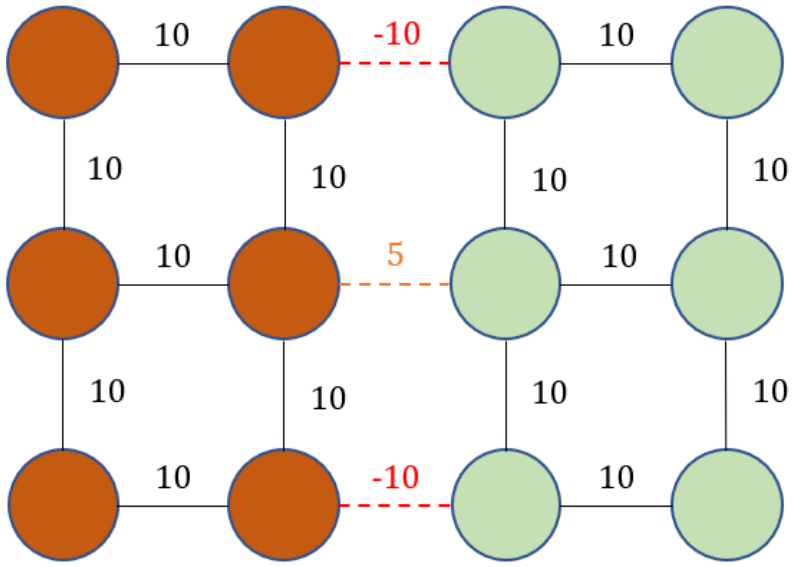

Figure 3.12
An example of the effect of the choice of the parameter threshold θ on the segmentation and its correlation clustering energy.

In figure 3.12, to identify the two objects (the orange one and the green one), the three highlighted dashed edges should be discarded. If $\theta = 0$, the orange dashed edge is still intact and hence all vertices are still connected, resulting only one segment. The correlation clustering energy in this case

is 0. If the threshold $\theta = 6$, all the three dashed edges are discarded and there are two separated segments. The correlation clustering energy decreases to $-10+5-10=-15$, which is in fact the lowest correlation clustering energy that can be achieved in this affinity graph. For this example, we see how considering the threshold that gives the minimum correlation clustering energy (restricted by the connected components) encourages a better segmentation.

Regarding the complexity of the energy computation in the case of a 4 connected-graph, at each iteration, we have to check at most $4 \times N$ edges, where N is the number of vertices that the pointer cycle is used for. The sizes of the segments that get connected at each iteration vary, but on average they increase linearly. That is, as we build the MST, we connect larger and larger segments and have to check more and more vertices for adjoining pairs in order to calculate the correlation clustering energy at each level.

In the simplest scenario when one of the two segments always contains a single vertex, we have to check the adjoining edges of an additional vertex after each iteration as our pointer cycle grows. Assume the graph is 4-connected, the running time complexity is then:

$$\mathcal{O}(4(1+2+\cdots+(\|V\|-1))) = \mathcal{O}(2\|V\|^2) = \mathcal{O}(\|V\|^2)$$

While it is difficult to estimate the complexity of this process, in our experiments, the inference step does take a noticeable additional time to complete. We note that a brute-force calculation of the correlation energy takes $O(\|E\|)$ for a segmentation, hence the calculation of all possible segmentations produced by the connected component procedure takes at most $O(\|V\|\|E\|)$.

IMPLEMENTATION

4.1 Introduction

Implementing any work on inferning requires an implementation of affinity learning with Kruskal's algorithm first. There are several steps to a successful implementation. First of all we need to settle on a dataset of images and segmentation ground truths. Then we need to investigate which of the loss functions discussed above performs better based on the training statistics. Next we need to decide on the learning model. In [23] a linear classifier was trained with Kruskal's algorithm. This can be used for a proof of concept and has the benefit of being applicable to any graph and relatively easy to implement, but it is not a very expressive classifier. We want to use a convolutional neural network similar to the U-net employed in [25] due to its proven ability in the segmentation learning task. The convolutions need a consistent graph structure, or in particular a matrix, hence are only applicable to a lattice graph with consistent connectivity (e.g. 4-connected or 8-connected graph). We pay for the more expressive classifier by a loss in flexibility for the structure of the graph. Another issue that surfaced was how the affinities are being presented by the classifier. Turaga's affinities for a 3D graph are the three layers of an output image, one layer for horizontal affinities, one for vertical ones and one for front to back ones. While this seems a nice idea, the automatic differentiation in Tensorflow that is used in the gradient descent algorithm has trouble with this type of data structure. Porter et al. presented an output image from which the affinities were then computed. We discuss different graph encoding methods for the affinities.

We consider various experimental setups to test the presented frameworks. The main considerations are how to represent the input data, which classifier to use, and how to represent the output graph structure. We discuss the advantages and disadvantages of the choice of the decisions on these aspects in our image segmentation application. For all setups, we use tensor structures (i.e. multi-dimensional arrays) for the numerical computation in Tensorflow and Numpy and for mappings between a graph and a tensor. Specifically, the output of a classifier is in the format of a tensor. We map this output

tensor to a graph structure so we can use algorithm 2. We then need to create the weight tensor and the meta label tensor to compute the training loss (equations 3.20).

4.2 The dataset

We considered several choices of available datasets. One option was the BSDS500 dataset [3] which is a benchmark set of labeled training images consisting of photos and crowd-sourced hand-drawn segmentations. Several of those images have different ground truth segmentations that differ quite a bit. One question that arises for such images is "should only the animal be segmented or the patterns in its fur"? These ambiguities made us move away from this dataset for our basic experiments. Another dataset we considered were grey scale microscope images of materials available internally through Los Alamos National Laboratory. These images have fairly obvious segmentations and a expert-labeled set of images created by only one expert is available, but restrictions on publishing these images made us hesitate.

While there are some other options on the table, we decided to experiment with a synthetic image dataset that was introduced in [6]. The dataset contains 100 synthetic color images, each of which has 3 channels of size 100 by 100, in bmp format. Each image has several segments of different colors and shapes. These synthetic images are easy to be segmented in order to find the ground truth segmentations. We then modify the dataset by adding Gaussian noise to the images not to have too simple a task for our learning frameworks. Specifically, each image is scaled to $[0,1]$ and three independent Gaussian-distributed masks (i.e. each pixel values of the masks is independently drawn from a normal distribution) that have the same shape as the image are created with mean 0.0 and standard deviation 0.1. We add these masks to the original image while clipping to keep image's intensities in $[0,1]$. With this, about 70 percent of pixels' intensities of a channel of the original image might get increased or decreased by less than 0.1 (before clipping). Some examples of the original and the corresponding noised images are shown in Figure 4.13.

We further preprocess the data by first compressing the image by a factor n, where n is a positive integer. We do this by subsampling pixels at every n-th location on each axis. Due to the design of the U-net, we need our input images' heights and widths to be multiples of 2. We do this by compressing

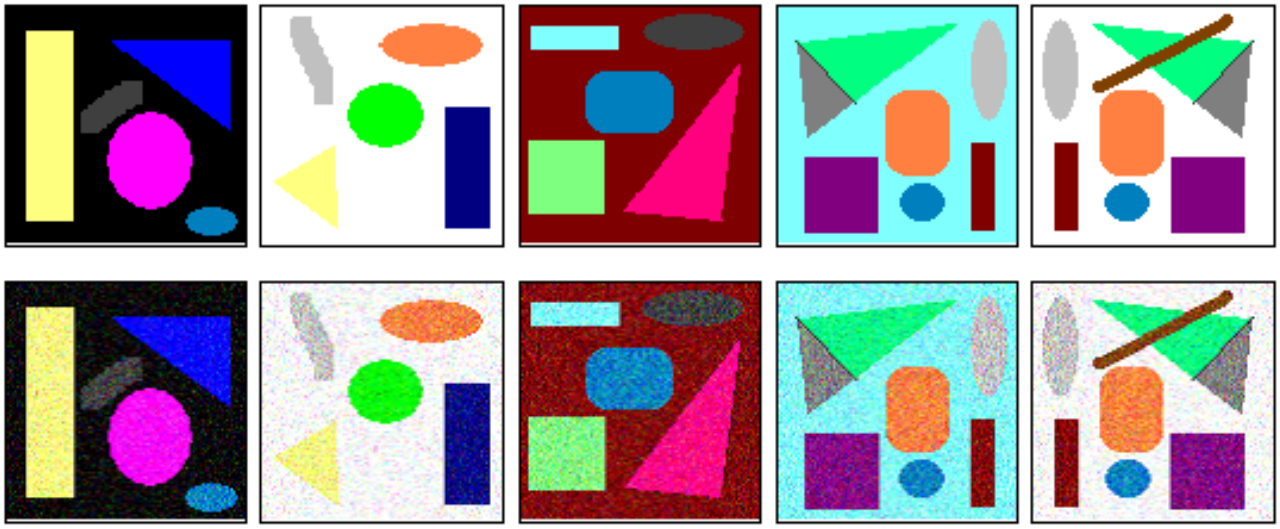

Figure 4.13
Top row: Original sample images. Bottom row: The same images with added noise.

the top left 96 by 96 corner of any of the synthetic 100 by 100 pixel images with added Gaussian noise described in section 4.2 by a factor $n = 4$. This results in images of size 24 by 24 for the actual training and test set. This compression speeds up our experiments considerably and makes the segmentation task harder.

Figure 4.14
An image sample from the synthetic dataset: (from left to right) the original image with noise, the compressed image, and the smoothed compressed image.

We use an averaging filter of size 3 by 3 to reduce the noise. While this filter is more likely to give extra little segments at the edge of the original segments, it effectively prevents sub-segments forming inside bigger segments. We do a convolution with this filter for each channel of the image with "same" symmetric padding. That is, we use mirror padding at the boundaries and the convolution produces a smoothed image that has the same size as the input image. For the 1D encoding setup, we need to

further convert this image to a tensor that encodes the edge information (described in section 4.4). For the 2D encoding setup, we can directly feed the smoothed image into the classifier. Here, we consider the output of the preprocessing step as the input to the classifier. Some additional implementation details that are characteristic of each experiment will be discussed later in chapter 5.

In our experiments, we sample from these preprocessed images and their ground truth segmentations for the training set and the validation set. The training set is used to update the parameters of the classifier. In training, we use a batch size of one, meaning that the gradient of the loss function is calculated and used to update the classifier's parameters after a training image has been processed. The classifier is trained on the first image and the parameters are updated, then the classifier is trained on the second image and so on until it returns to the first image. We count one pass through all the training images as one training epoch. While the classifier is being trained, it is also evaluated using the validation set to minimize overfitting. The parameters of the classifier are not affected by the results provided by the validation set. We periodically record all the statistics (e.g. the loss and Rand error of the training set and the validation set) and the parameters of the classifier. We use the statistics to determine which instance of the classifier is the best.

4.3 Auto differentiation

In gradient-based learning algorithms, an important part is to compute the gradient update for all the parameters of a classifier. Auto differentiation works based on the fact that most common functions can be decomposed into a sequence of primitive operations (e.g. addition, subtraction, multiplication, division, etc.) and applications of basic functions (e.g. exponentials, logarithms, sines, cosines, etc.). The derivative of an arbitrary function composed of these primitives can be automatically computed by applying the chain rule recursively.

In our implementation, we rely on the automatic differentiation ability of available libraries, namely of Tensorflow [1]. Tensorflow has a large number of mathematical operations with well-defined gradients. Tensorflow also works seamlessly with Numpy [30] - a Python library used for numerical computations.

Tensorflow and Numpy are open source libraries (in Python) and available through the Python

package installer. Both use a similar core data structure called a multidimensional array (in Numpy) or a tensor (in Tensorflow). A tensor in Tensorflow is a generalized version of a multidimensional array. The libraries have been developed to the point where both of them can work interchangeably. Numpy arrays and Tensorflow tensors can be easily converted to the other. Tensorflow allows the exploitation of GPU computing power and eases the implementation of end-to-end deep neural networks, from data preparation to model building, training, and evaluation. While we are aware of many other machine learning libraries and frameworks, we choose Tensorflow because of the flexibility (working nicely with a number of other libraries) and the abstraction (ability to connect building blocks) it provides. On the other hand, Numpy is simpler and is highly optimized in manipulating multidimensional matrices. We specifically use Numpy in some subroutines that require value assignment and iteration for matrices (e.g. computing the weights and meta labels).

4.4 1D encoding

In the initial setup, we use a vector (i.e. a one dimensional tensor) to encode the edge affinities, hence the name "1D encoding". For example, a 4-connected lattice graph representation of an H by W image contains $H \times (W-1) + (H-1) \times W$ edges and the output of the classifier will be a tensor of shape $\|E\| \times 1$, where $\|E\| = H \times (W-1) + (H-1) \times W$. We use a linear classifier for this setup. The feed forward step is defined by applying a linear function to the feature vector of each edge

$$\widehat{A_i} = X_i \cdot \mathbf{w} + b \tag{4.24}$$

In equation 4.24, $\widehat{A_i}$ is the affinity prediction for edge i; X_i is the feature vector of edge i and is a row vector, which contains n features; $\mathbf{w}$ and b are the weight vector and the bias of the linear classifier, where $\mathbf{w}$ has n elements corresponding to n features of X_i. In tensor form, equation 4.24 can be written as

$$\widehat{A} = X \cdot \mathbf{w} + b \cdot \mathbf{1}, \tag{4.25}$$

where $\widehat{A}$ contains all affinity predictions and is a tensor of shape $\|E\| \times 1$, X is a tensor of shape $\|E\| \times n$ and can be seen as a matrix whose each row X_i is a feature vector as in the equation 4.24, $\mathbf{w}$ is a tensor of shape $n \times 1$, and b is a scalar and $\mathbf{1}$ is a tensor of shape $\|E\| \times 1$. Using a linear

classifier is likely to hide all spatial information of the data. However, spatial information for pixels is crucial in image segmentation applications. One way to combat the problem is to employ a feature extractor filter that collects a window of information about the neighborhood of each edge as a vector associated with each edge. That is, we use a vector that contains all the information around an edge as the feature for each edge. This technique is similar to a convolution applied to the corresponding 2D matrices, without adding up the terms. For example, if we use a feature extractor filter of size 5×5, we have a feature vector of 25 elements for each edge in each channel of the input.

We note that this feature extraction does not have to be applied to the input image, but could be used on a feature map of the image such as one resulting from a convolution with Sobel filters, which will be described in section 5.1, or one resulting from the Gaussian weighting function [13]). Given an input tensor of shape $10 \times 10 \times 3$, whose 4-connected lattice graph representation has $9 \times 10 + 10 \times 9 = 180$ edges, using the method above with a filter of size 5×5 gives a feature tensor X of shape 75×180 (in Figure 4.15, each X_i would be a vector of length $25 \times 3 = 75$). The corresponding weight tensor and bias tensor of such input tensor would have shape 75×1 and 180×1 respectively. The output of the classifier is then a vector of length 180, or said say differently, a tensor of shape 180×1.

To train a classifier, we also need a ground truth label for each pixel, which is required by algorithm 2. Note that we do not use the ground truth label to calculate the hinge loss, instead, we use the meta labels defined in section 3.2. The training process is straightforward. In the feed-forward step, our classifier gives the output of all edge affinities in the graph. However, only the MST edge weights contribute to the loss (section 3.2). To compute the training loss, we need to map the output affinity tensor back to a graph structure to be able to apply algorithm 2. This mapping is required to be matched with how we encode the edge weights for the input.

This basic setup is good to start with to collect preliminary results but is not the most expressive classifier. The difficulty we encountered in implementing it was keeping track of the edge vector elements and their positions in the graph. Tensorflow posed some challenges in setting up the correspondence correctly. Reconstructing the weighted graph of edge affinities from this edge vector has to be done very carefully.

Overall, this 1D setup struggles to scale up to larger images and more expressive classifiers. In

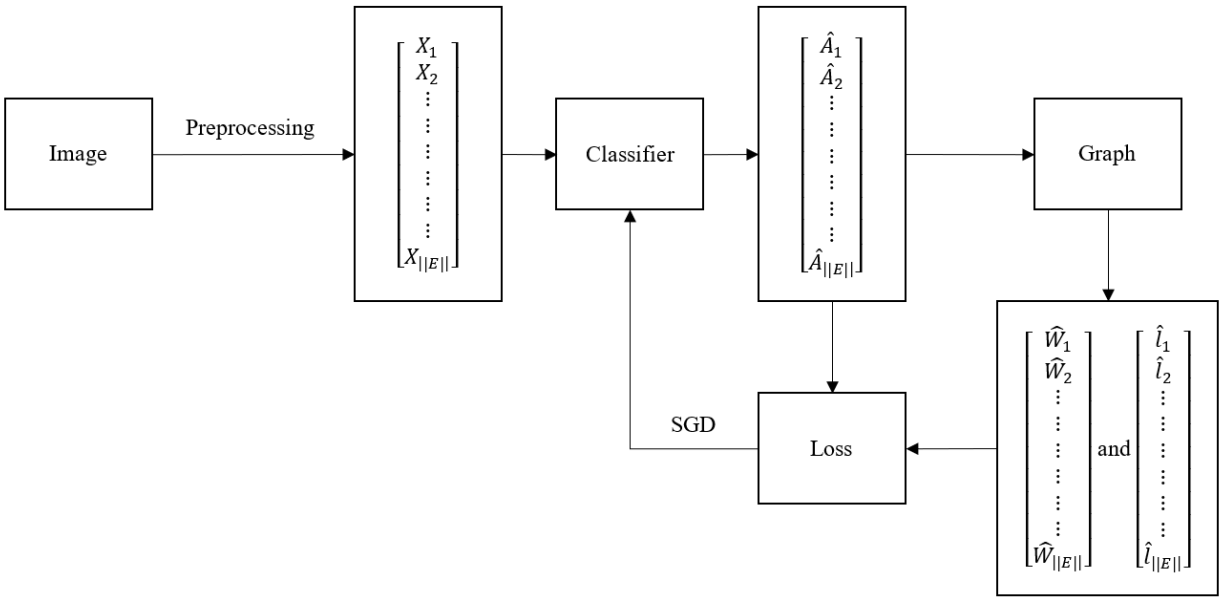

Figure 4.15
Flowchart for the linear classifier acting on a 1D vector of edge information. SGD stands for stochastic gradient descent.

the context of neural networks, a fully connected layer (as described in section 2.7) of N nodes is required to take a 1D-tensor of size N and produce a tensor of the same shape. Since each node has $N+1$ parameters, there are $N \times (N+1)$ parameters in total. To produce edge affinities, we can only use fully connected layers with this setup, which leads to a large number of parameters. A fully connected neural network is extremely expensive and is usually prone to overfitting due to the large number of parameters it has, which is not desirable.

4.5 2D encoding

As a different approach, similar to that employed by Turaga et. al. in [28], we encode the edges of an $H \times W$ image as a set of matrices of the size $H \times W$, one matrix for each possible edge direction. A first layer could be the affinities for the horizontal edges. To keep the same size, we need to pad one column for horizontal edges or one row for vertical edges. If we only consider horizontal and vertical edges, we need two edge matrices. Being careful not to shift things, we can construct the affinity graph from those edge matrices. It is also easy to map these edge matrices back to a graph

structure. For a 4-connected lattice graph of size H by W, the affinity prediction is a tensor of shape $H \times W \times 2$. For a 3-dimensional microscope image where each pixel has six neighbors, we would obtain 3 channels. With this mapping, an 8-connected lattice graph can be easily represented by 6 matrices. Note that when mapping a graph to a tensor, there is an extra row and an extra column that are padded with zeros in each matrix channel; when mapping the tensor back to a graph, these extra row and column are ignored.

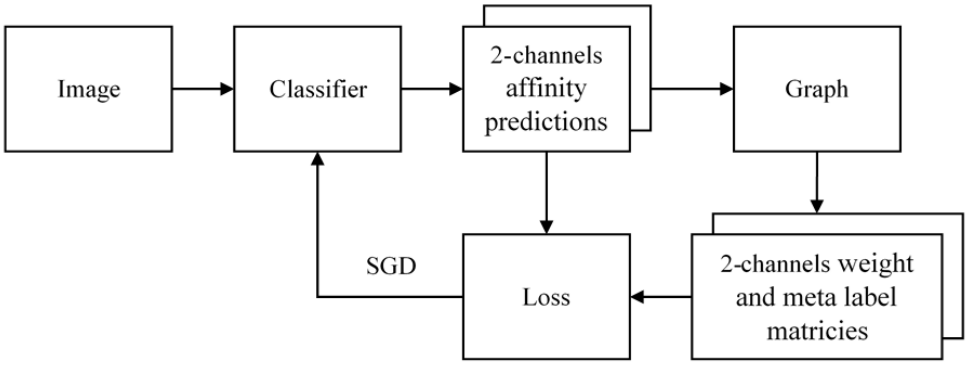

Figure 4.16
Flowchart for a classifier acting on a 2D representation of the affinity graph. SGD stands for stochastic gradient descent.

For this setup, we can directly feed images into our model without the need to first compute the edge information as we have to for the 1D encoding and we ask the classifier for a set of affinity matrices as an output. The setup allows us to leverage convolutional neural networks, which have proven to be effective in image processing tasks ([18], [19], [22], [25], [29]), especially in learning spatial features of images. It is optional to add edge information (e.g. edges detected by Sobel filters) in into the input as additional channels added to the original image.

It is important to correctly define the mappings between the two channel representation with edge information and the affinity graph. If there is a shift in the affinities, the gradient updates will not be applied correctly to the weights and biases. Apart from technical issues in matching up affinity matrices and the affinity graph, this setup is superior to the 1D setup in section 4.4. This is because it is better in scaling up to larger images due to the fact that we can directly use images as inputs and

use convolutions in our classifier to exploit spatial relationships of the vertices.

RESULTS

5.1 Using a linear classifier for initial results

We begin our experiments with a linear classifier to obtain initial results on our loss functions described in section 3.2. We want to make sure that by optimizing the loss functions, we actually optimize the Rand error (i.e our model learns to correctly classify the relationship between each pair of vertices). We use the 1D encoding and a linear classifier as described in section 4.4. We train this linear classifier using two formulations of the meta labels and the weights in the hinge loss function defined by equation 3.20. We note that the threshold parameter θ is fixed at zero.

- In the first model, the meta labels and the weights are defined by equations 3.18 and 3.21:

$$\text{diff-}l_{uv} = sign(\#\text{same}_{uv} - \#\text{diff}_{uv})$$

and

$$\text{diff-}W_{uv} = \left| \frac{\#\text{same}_{uv} - \#\text{diff}_{uv}}{\#\text{same}_{uv} + \#\text{diff}_{uv}} \right|.$$

Since the meta labels and the weights are defined by the differences of the counts from algorithm 2, we name the model with this loss function the "diff" model.

- In the second model, we use the second formulation described in equations 3.22 and 3.23:

$$\text{hit-}l_{uv} = \begin{cases} 1 & \text{if } \widehat{A}_{uv} = \theta \\ -sign(\widehat{A}_{uv} - \theta) & \text{otherwise} \end{cases}$$

and

$$\text{hit-}W_{uv} = \begin{cases} \frac{\#\text{diff}_{uv}}{\#\text{same}_{uv}+\#\text{diff}_{uv}} & \text{if } \widehat{A}_{uv} \geq \theta \\ \frac{\#\text{same}_{uv}}{\#\text{same}_{uv}+\#\text{diff}_{uv}} & \text{if } \widehat{A}_{uv} < \theta \end{cases}.$$

With these definitions, each MST edge always takes a hit from the classifier (i.e. the classifier considers flipping an MST edge about the threshold θ every time). We call this the "hit" model.

The input to the linear classifier is prepared as described in section 4.4 using the sub-sampled synthetic images described in section 4.2. Given an input image, we first calculate its gradient image by applying two convolutions with Sobel filters for each channel of the image. These convolutions are defined as

$$G_{xij} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * N_{ij} \quad \text{and} \quad G_{yij} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * N_{ij}$$

where N_{ij} is a 3 by 3 neighborhood around pixel (i, j) and $*$ is the convolution operation. The convolutions result in two gradient maps for each channel (i.e. one in horizontal and the other in vertical direction). The gradient magnitude estimation at each pixel (i, j) is defined as

$$I_{ij} = \sqrt{G_{xij}^2 + G_{yij}^2}.$$

This calculation reduces the gradient maps to a three channels gradient image whose shape is the same as the shape of the image. This gradient image is used to make the feature vector for each edge. To take advantage of some spatial features, we make a feature vector for each edge (u, v) using a neighborhood around pixels u and pixel j of the gradient image. Specifically, for an edge (u, v), in each channel, we gather a neighborhood of size 5 by 5 around pixels u and another neighborhood around pixel v and calculate the average (element-wise) of the two neighborhoods, and finally reshape all of them to a vector of length $5 \times 5 \times 3 = 75$. This vector gives information that helps in identifying edges. If the average is high, it is more likely that the two vertices are in different segments. From the gradient image, we form a feature matrix in which each row is the feature vector of an edge.

Recall that the linear model is defined in equation 4.25 as

$$\widehat{A} = X \cdot \mathbf{w} + b \cdot \mathbf{1}$$

In our implementation, with an input image of size 24×24, the affinity prediction $\widehat{A}$ is a tensor of shape 1104×1, the feature tensor X is calculated as described above and has shape 1104×75 whose each row is a feature vector, the weight $\mathbf{w}$ is a tensor of shape 75×1, the bias b is a scalar and $\mathbf{1}$ has shape 1104×1. Here, 1104 is the total number of edges in the 4-connected graph representation of a $24 \times 24 \times 3$ synthetic image. Each value in the affinity prediction $\widehat{A}$ is mapped to the correct edge

in the graph. Algorithm 2 follows and the loss is computed. Back-propagation by stochastic gradient descent is handled by Tensorflow.

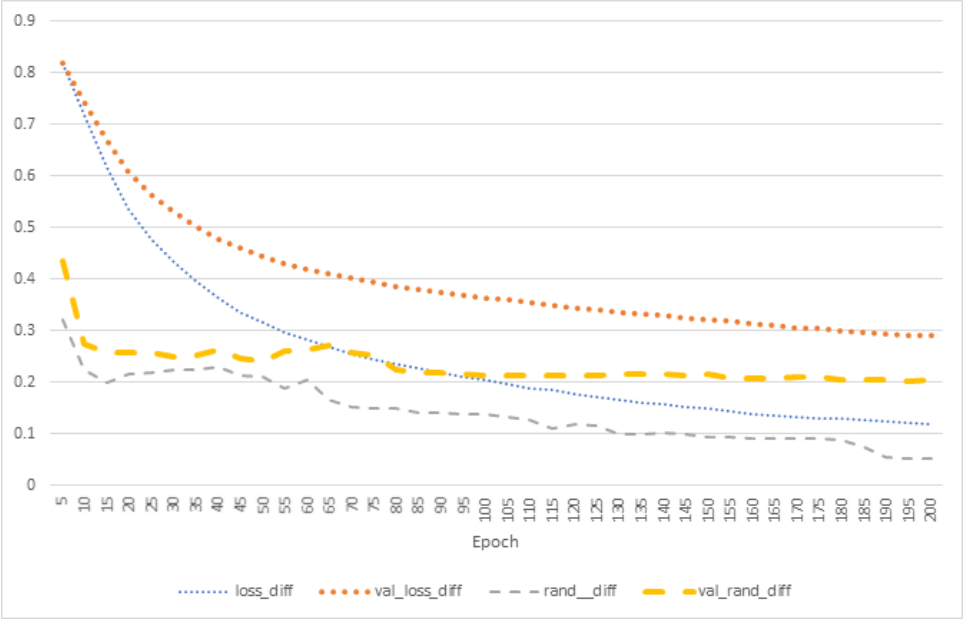

Figure 5.17
Training statistics (training and validation loss/Rand error) of a typical training trial of the linear model with "diff" loss.

The "diff" and "hit" linear models are trained on a training set of 5 images and validated on another set of 5 compressed synthetic images. Each classifier is trained for 200 epochs using gradient descent optimizer with learning rate 0.05. The learning rate is selected among $\{0.01, 0.05, 0.1, 0.5\}$ because it gives a relatively steady training statistics. We plot the averages of the hinge loss and the Rand error of the predicted segmentation for both the training and validation set for each models in figures 5.17 and 5.18.

In both figures, we see that the average of the Rand error over all images in the training set is generally decreasing, which signals that our modified learning objectives are actually guiding the linear model to produce affinities that yield better connected components segmentations. We observe that the "diff" model looks smoother (i.e. having little variation) compared to the "hit" model. By definition, the "hit" loss always considers raising the negative affinities and lowering the positive affinities. Recall that only when all the MST edges are correctly classified (i.e all MST edges connecting ver-

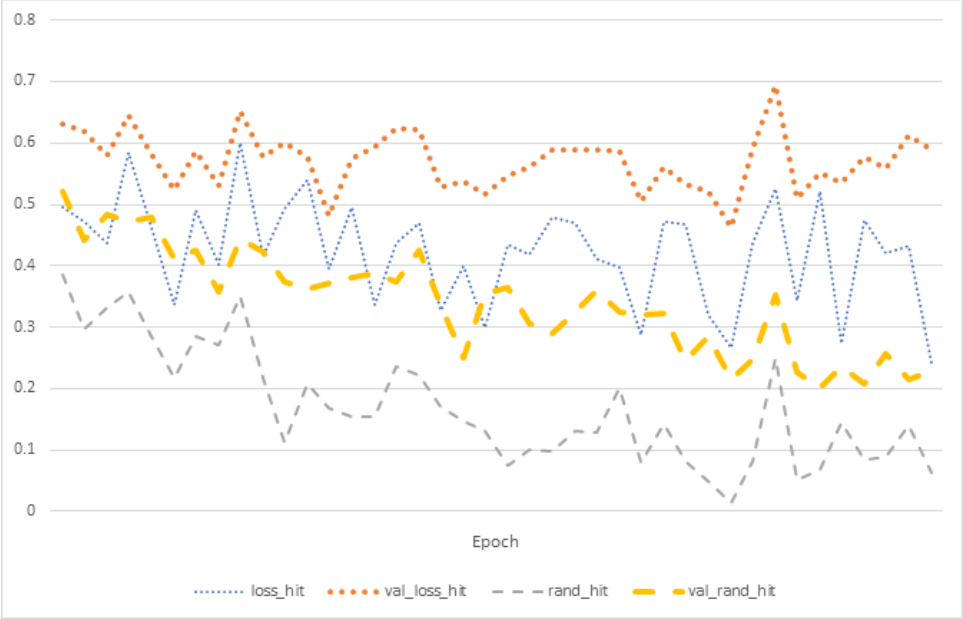

Figure 5.18
Training statistics (training and validation loss/Rand error) of a typical training trial of the linear model with “hit” loss.

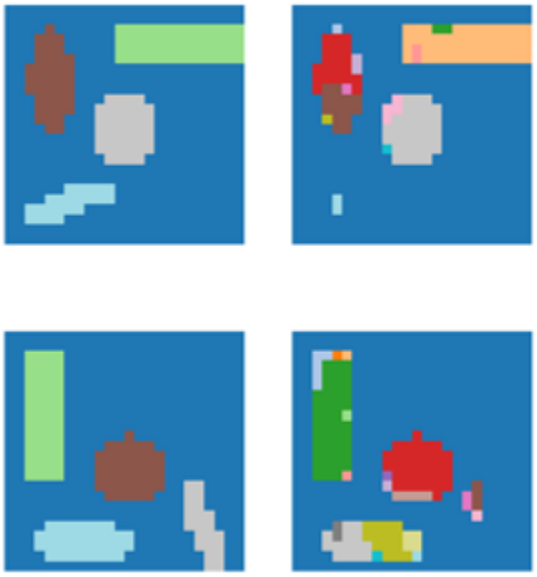

Figure 5.19
Some ground truths (left) and segmentation predictions (right) of the linear classifier with the “diff” loss.

tices of the same ground truth segment are positive and the other MST edges are all negative), the contribution weights W_{uv} for all the MST edge are zero. Even if there is only a few misclassified MST

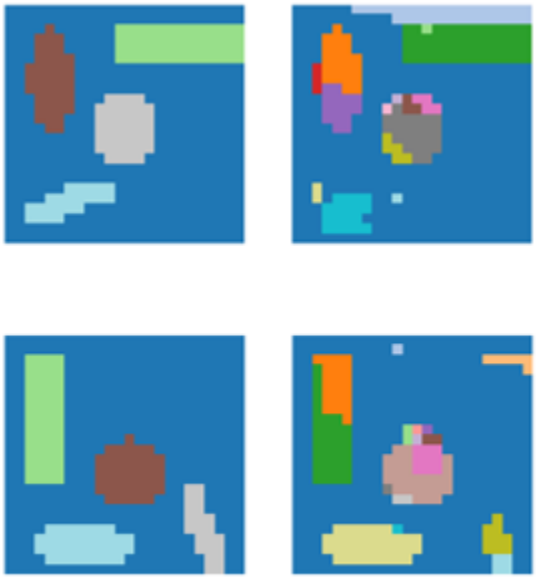

Figure 5.20
Some ground truths (left) and segmentation predictions (right) of the linear classifier with the "hit" loss.

edges, they could still badly affect the counts of the latter added MST edges in algorithm 2, and hence the loss might still be high.

Regardless, we observe that the two models are learning to generate correct affinities. In figures 5.19 and 5.20, we see that the two models are able to locate the objects but still oversegment inside the objects badly. If we look at the smoothed input images (in section 4.2), it is difficult to visually see the edges, however, the two classifiers are pretty good in identifying the boundaries. This suggests that our learning algorithm can minimize the Rand error with inputs that have no clear boundaries between objects. Even though we use only a few images to train and validate the two linear models, the results are encouraging because it shows promising results that signal our implementation of the learning algorithm and the 1D graph encoding is working as intended. We continue experimenting the two loss formulations with a more expressive classifier, namely the mini U-net, that employs convolutions and 2D graph encoding.

5.2 Using a mini U-net with "diff" loss

For this experiment, we use 2D encoding with a mini version of the U-net as described in section 2.7. Since we train and validate our model on a relatively small and simple dataset, the full original U-net architecture would be too excessive. We note that the original U-net was trained and validated on 572×572 grayscale images [25].

For training and validation images, we use the preprocessed synthetic images described in section 4.2. We train on 30 synthetic compressed images of size 24 by 24, and validate on a different set of 30 images. We pick the learning rate for gradient descent to be 0.05. This learning rate is established through trial and error, where we look for a balance between the training speed, the model's ability to reach to a good minimum (i.e. to not converge to a local minimum) of the loss function, and the model's ability to converge.

In this experiment, we use the "diff" loss that is defined using

$$\text{diff-}l_{uv} = sign(\#\text{same}_{uv} - \#\text{diff}_{uv})$$

and

$$\text{diff-}W_{uv} = \left| \frac{\#\text{same}_{uv} - \#\text{diff}_{uv}}{\#\text{same}_{uv} + \#\text{diff}_{uv}} \right|.$$

The linear model with this loss formulation in section 5.1 had produced the smoother curves, while attaining about the same minimum Rand error compared to the linear model with the "diff" formulation.

The mini U-net is trained for 500 epochs to the point where overfitting is clearly observed on the validation set (i.e. when the training loss and Rand error are decreasing but the validation loss and validation Rand error are increasing). We record the averaged training statistics (e.g. training/validation losses and Rand errors) every 10 epochs. That is, after every 10 epochs, we compute all statistics for each sample (the 30 samples in the training set, and the 30 samples in the validation set) and average those statistics over the the two sets. The experiment is repeated 5 times to confirm that the result is reproducible (figures 5.21 and 5.22) and the best instance of the model is chosen by the best validation Rand error. We show a typical plot of these statistics and some predicted segmentations on the validation images in figures 5.23 and 5.24 respectively.

In figure 5.21, we see that the hinge loss defined using the meta-labels and weights defined in equations 3.18 and 3.21 works quite well in training the mini U-net with gradient descent. The training loss decreases steadily, which signals that we have a suitable learning rate. The best instances based on the validation loss occur roughly between epoch 90 and epoch 150. Figure 5.22 shows the corresponding Rand errors on training set and validation set. While they are more fluctuating, the general trend is consistent with the corresponding losses. The best instances based on validation Rand error also occur roughly between training epoch 90 and epoch 150. While the training and validation losses are pretty smooth, the variation of the corresponding training and validation Rand error curves suggests that there might be some extreme cases in which even though the loss is small but the Rand error is still large.

Looking at the segmentations that result from the learned affinity graphs when thresholded at $\theta = 0$ (figure 5.24), we find the overall shapes but oversegment at their edges. This might be due to the use of the average filter on the input image, which adversely smooth out the edges. In the third validation shown in figure 5.24, the prediction failed to identify the two triangles objects at the bottom-left and falsely identify a square object at the top-left. This might suggest overfitting in the model, however, it seems the model does not overfit much because it does quite well on other validations.

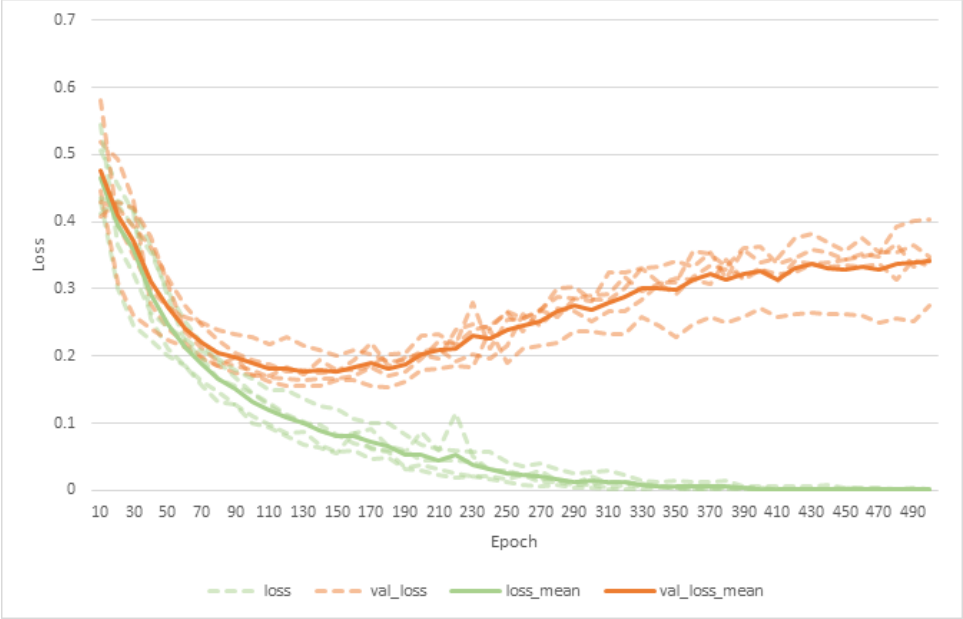

Figure 5.21
Average hinge losses on the training set (green) and the validation set (orange) of 5 independent trials of the mini U-net and "diff" loss model.

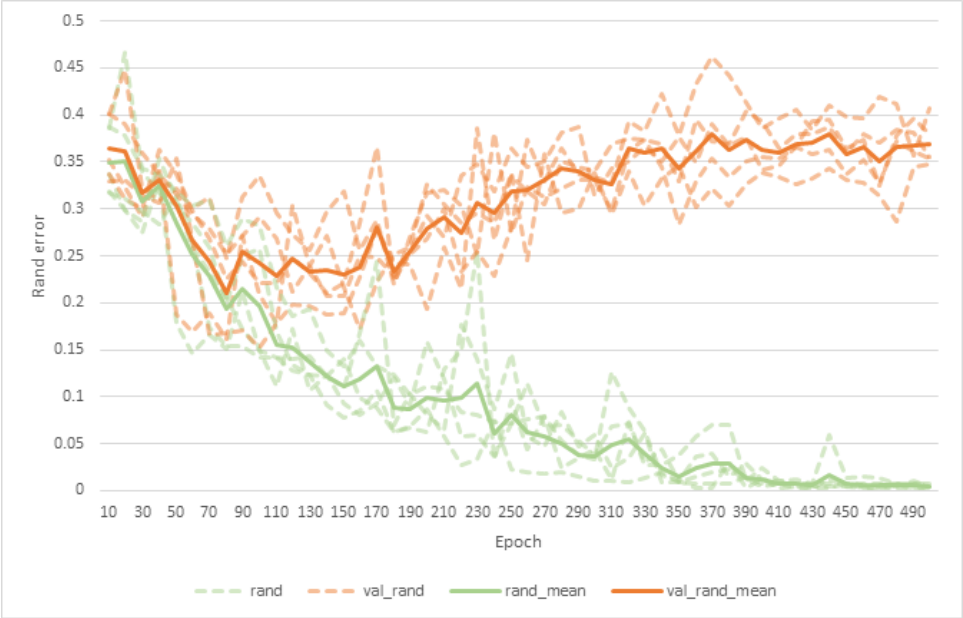

Figure 5.22
Corresponding average Rand errors on the training set (green) and the validation set (orange) of 5 independent trials of the mini U-net and "diff" loss model.

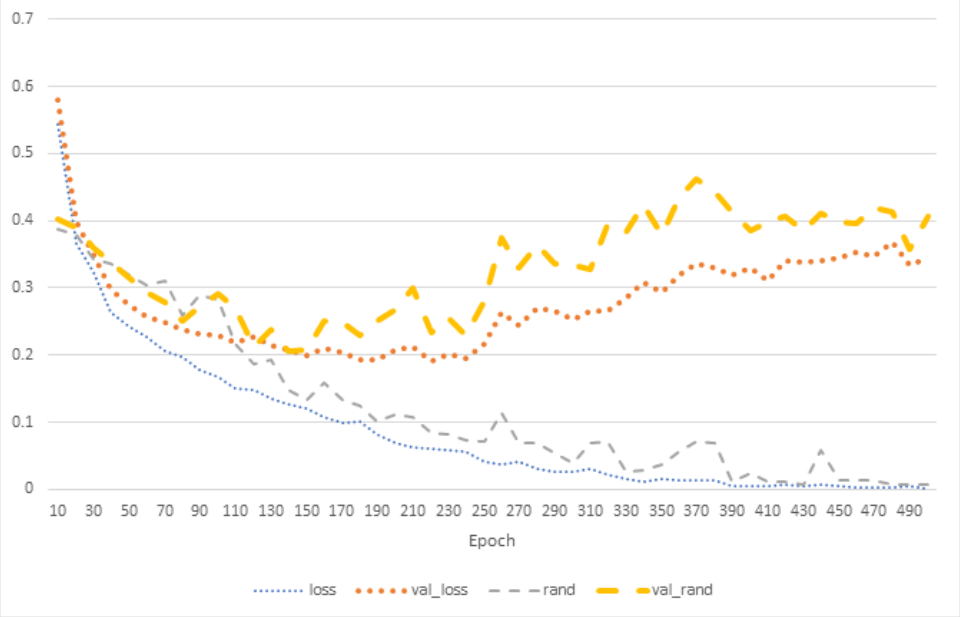

Figure 5.23
Training statistics (training and validation loss/Rand error) of a typical training trial of the mini U-net with the “diff” loss.

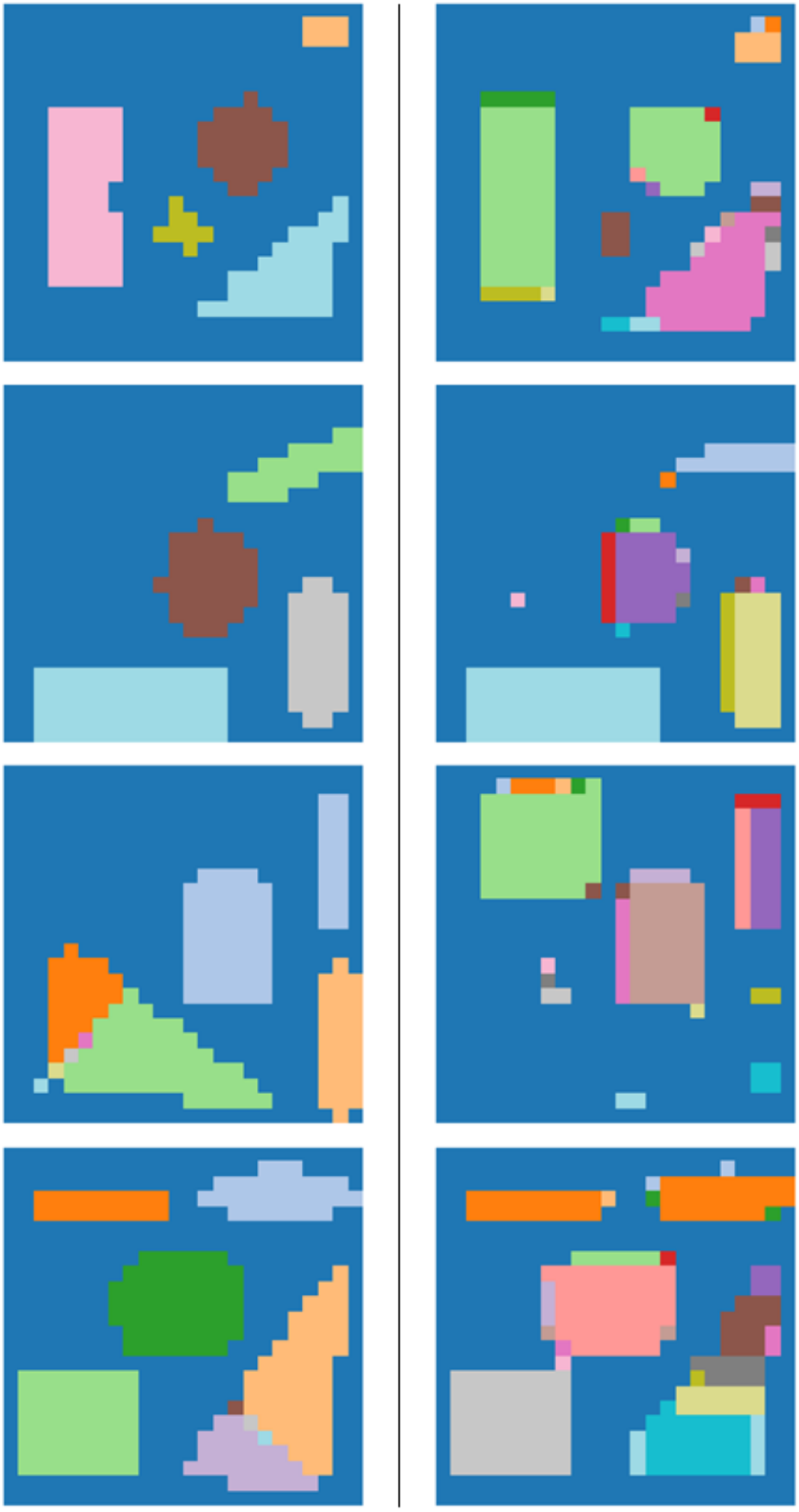

Figure 5.24
Some ground truths (left) and segmentation predictions (right) of the mini U-net and “diff” loss model. Each color specifies a class.

5.3 Using a mini U-net with "hit" loss

We train the same mini U-net architecture as described in the previous section, using the same weighted hinge loss function (equation 3.20) but with different weights and meta-labels. This time we use the "hit" formulation with the weights and the meta-labels defined as

$$\text{hit-}l_{uv} = \begin{cases} 1 & \text{if } \widehat{A}_{uv} = \theta \\ -sign(\widehat{A}_{uv} - \theta) & \text{otherwise} \end{cases}$$

and

$$\text{hit-}W_{uv} = \begin{cases} \frac{\#\text{diff}_{uv}}{\#\text{same}_{uv}+\#\text{diff}_{uv}} & \text{if } \widehat{A}_{uv} \geq \theta \\ \frac{\#\text{same}_{uv}}{\#\text{same}_{uv}+\#\text{diff}_{uv}} & \text{if } \widehat{A}_{uv} < \theta \end{cases}.$$

We use the same training and validation sets as described in section 5.2. We still use a batch size of 1 and the learning rate is picked to be 0.05, which balances the training speed and variations of the training statistics.

The network is trained for 500 epochs to collect results to compare with the model in section 5.2. We notice that this formulation is not as stable as the previous formulation (section 5.2) in terms of the variance of the training statistics.

In figure 5.25, we see that the values of the "hit" training loss do not have a clear trend. They vary a lot and are almost always above 0.4. This does not quite match with figure 5.26 in which the training Rand error seems to going down on average. In this regard, the model with "diff" loss (e.g. equations 3.18 and 3.21) is better due to its stability (in terms of variations of training statistics) and reproducibility (i.e there is not much difference between different trials).

In figure 5.26, we observe many high spikes in the Rand error, which are not desirable. These spikes do not stop appearing even after 500 training epochs. However, we note that these spikes seem to gradually become less intense. This might be because the minimization of the loss function has too much impact on the Rand error. However, we have tried smaller learning rates which did not seem to fix the problem, this suggests that the weights of the MST edges in the "hit" formulation are not optimally distributed.

In figure 5.28, we observe that the best “hit” model performs not as good as the best “diff” model in section 5.2. The “hit” model struggles to identify some objects, however, the objects that it successfully detects have quite good outlined edges. Overall, the “hit” model does seem to learn to minimize Rand error but there is room for improvement.

Figure 5.25
Average hinge losses on the training set (green) and the validation set (orange) of 5 independent trials of the mini U-net and “hit” loss model.

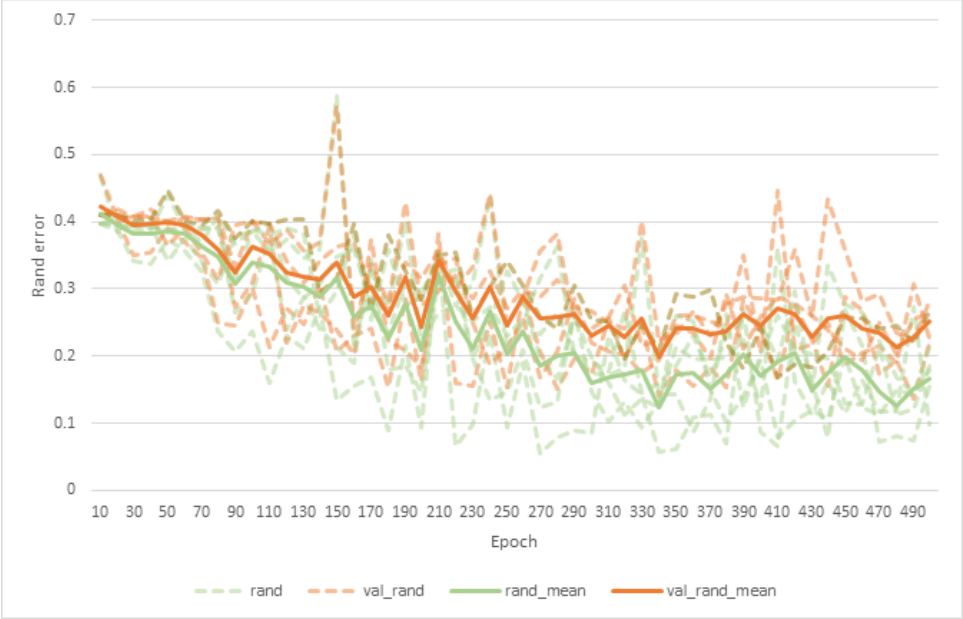

Figure 5.26
Corresponding average Rand errors on the training set (green) and the validation set (orange) of 5 independent trials of the mini U-net and "hit" loss model.

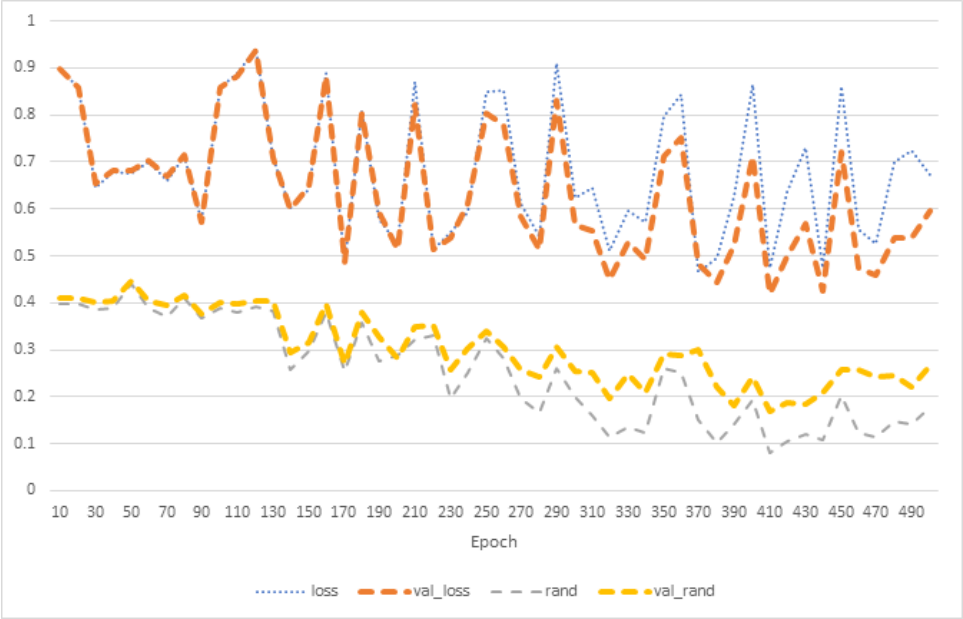

Figure 5.27
Training statistics (training and validation loss/Rand error) of a typical training trial of the mini U-net with the "hit" loss.

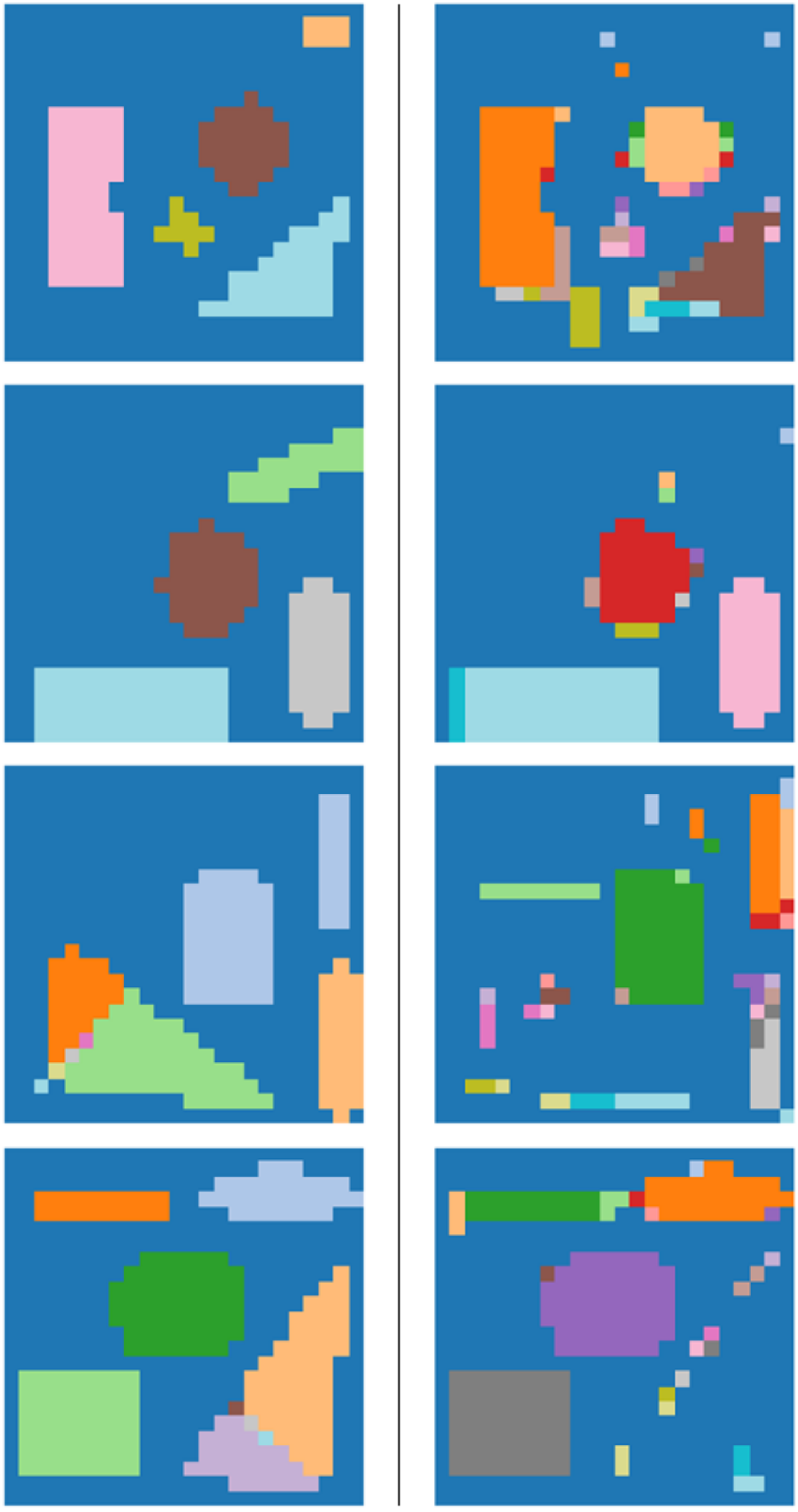

Figure 5.28
Some ground truths (left) and segmentation predictions (right) of the mini U-net and “hit” loss model. Each color specifies a class.

5.4 Comparison of inferning and threshold zero models for a mini U-net with the "diff" loss

Based on the results described in the two previous sections, we choose the "diff" loss formulation as the model to test "inferning" on our dataset of synthetic images. We use the same setup as described in section 5.2, but for the inferning model we have an extra threshold parameter θ in the hinge loss function (equation 3.20). This time instead of using zero as a threshold, we consider all possible thresholds for segmenting the MST and select the value that minimizes the correlation clustering energy function as the threshold used in the segmentation and loss function. We call this threshold the "inferning threshold". The values of the correlation clustering energy function are calculated for each threshold from the highest to the lowest as described in section 3.3. We collect statistics of the results from training this "inferning model" and compare them side by side with what we achieved in section 5.2 without using the inferning threshold (the "non-inferning model").

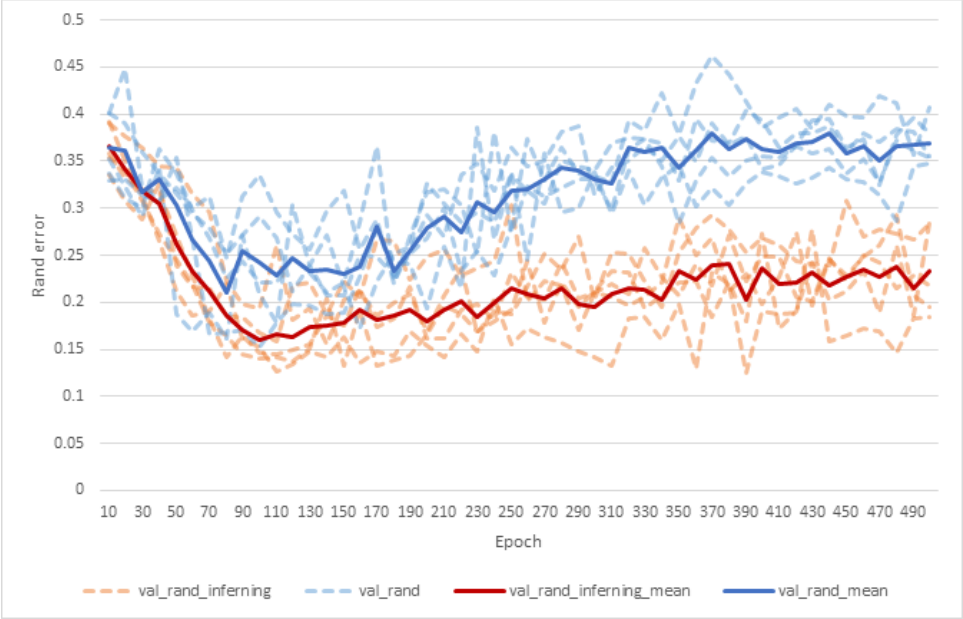

Figure 5.29
Corresponding average Rand errors on the validation images of models trained with and without using the inferning threshold. There are 5 trials of each model and the mean curve of each is highlighted.

Figure 5.29 illustrates that inferning does indeed reduce the Rand error on the validation set. The difference in the minimum Rand error of the two mean curves is 0.0506, an improvement of five

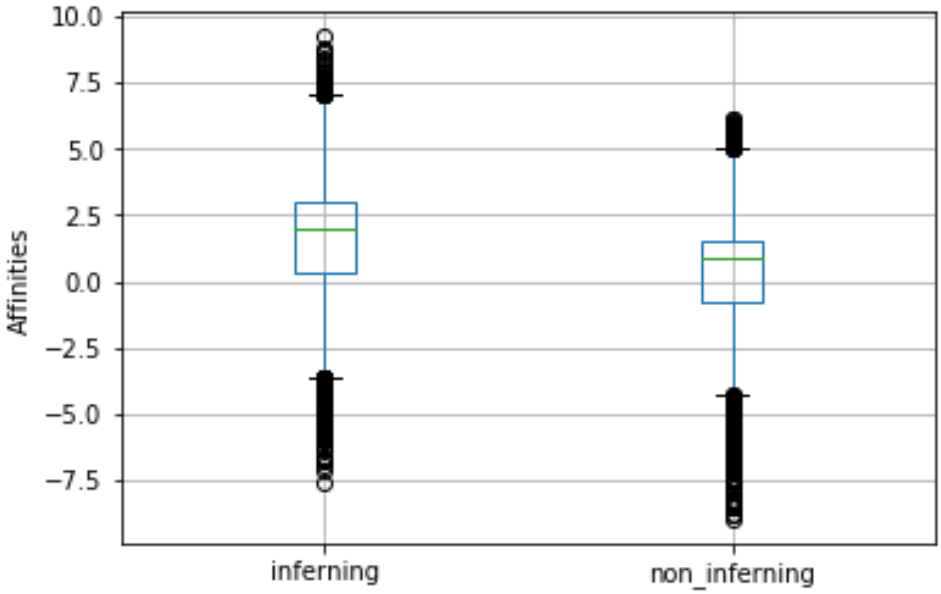

Figure 5.30
The box-plots of affinities predicted by models trained with (red) and without (blue) using the inferning threshold.

percent. As in the previous sections, this was based on training on 30 synthetic images size 24 by 24 and validation on 30 different images from the same synthetic data set. We note that the optimal instances of the inferning model are achieved after about 100 training epochs, which is very close to the number of training epochs needed for the optimal non-inferning model, which was around 80. This means the extra inference step in finding the inferning threshold does not require a large number of extra training epochs in the optimization process. This result shows that inferning is indeed a very promising approach to generate better segmentation models. In figure 5.30, we plot the distributions of the predicted affinities of all 30 validation images and a clear difference is observed. This confirms that adding the extra inference step in the training loop does have an effect on the output of the neural network.

As a next step we compare the two models by plotting their receiver operating characteristic curves. A receiver operating characteristic (ROC) curve is a common way to assess classification models, see for example [10]. It plots the true positive rate (TPR) as a function of the false positive rate (FPR) of a classification model over various threshold settings. The diagonal on that graph would

be a neutral model which has the same rates of true positives and false positives. The further a ROC curve is above this diagonal, the better the model. We define a true positive as a correct classification of the relationship between two vertices in the affinity graph, that is, the two vertices are correctly assigned the same label in the segmentation. Likewise, we define a false positive as an incorrect classification (i.e. incorrectly assigning two vertices the same label). In our formulation, the learning algorithm only suggests a single threshold for the connected component procedure. In practice, by varying the threshold parameter, one can generate all possible segmentaions from over-segmentations into individual pixels or small segments (high thresholds) to under-segmentations (low thresholds) down to just one segment for the affinity graph and its MST. The ROC curve can be easily computed from the counts produced by algorithm 2 due to the fact that the counts represent the number of true positives and false positives.

In figure 5.31, we show the receiver operating characteristic curves of pixel-pair connectivity classification of the inferening and non-inferning model. We compile a ROC curve for each of the 30 validation images. Each curve is linearly interpolated from a set of $\|V\|$ pairs of (TPR, FPR) because there are at most $\|V\|$ possible threshold levels and corresponding pairs of (TPR, FPR). The red crosses and the blue dots specify the corresponding (TPR, FPR)'s that are resulted from the choice of the thresholds made by the inferning model and the non-inferning model. We observe that the inferning model chooses significantly better thresholds and effectively segments the graph (i.e. for a ROC curve, the further the corresponding (TPR, FPR) is above the diagonal line, the better segmentation choice is made). This suggests that the inferning model has learnt to produced better affinities in the sense that it allows the interpretation and application of the correlation clustering energy. This also substantiates our hypothesis that more time spent in inference improves the overall performance of the connected component segmentation.

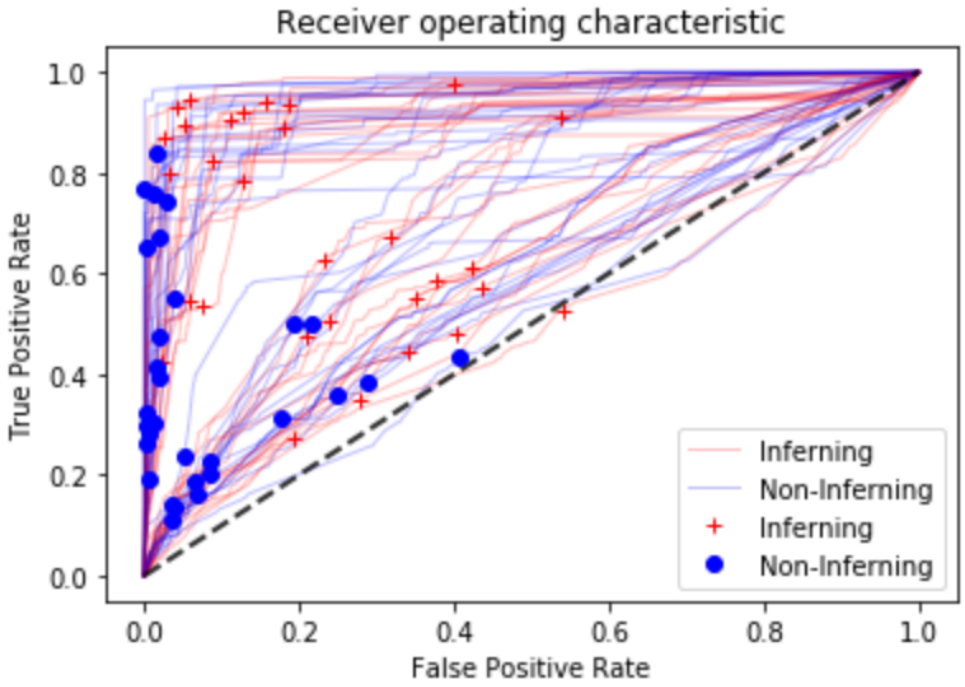

Figure 5.31
The receiver operating characteristic curves of pixel-pair connectivity classification of the inferning and non-inferning model. The blue dots are threshold zero, red + signs indicate the inferning thresholds.

DISCUSSION

For this thesis, we have used the idea of representing an image as a graphical model in order to segment it into meaningful regions. We devised and implemented a novel way of producing affinities or edge weights for this graph. To do so, we have built on Turaga's work [28] that used deep learning with a convolutional neural network. In Turaga's work, the threshold for the connected component procedure was fixed at zero. Our method treats the threshold as a parameter to be optimized. For the implementation of the stochastic gradient descent, we need a continuous loss function that approximates the Rand error. For the learning algorithm, we use a mini-batch learning algorithm of MST edges instead of the online learning algorithm that only uses the maximin affinity edge between two randomly chosen vertices. That required a modification of Kruskal's algorithm to keep track of the number of true positives and false positives as the MST is built (i.e. as segments are merged). The counts generated by this algorithm are used in the training and in the loss function. We experimented with two different versions of a new loss function, as described in chapter 5. We adapted the learning framework to Tensorflow and Numpy, changed the loss function and changed from online learning to mini-batch learning. We have done experiments on a synthetic dataset in which we found a 5% decrease in average of the Rand error on the validation set when using the optimal threshold rather than thresholding the affinities at zero.

In our learning framework, computing the meta labels and the MST weights is the major bottleneck. This computation has to be done sequentially on a single a core in a central processing unit (CPU). Hence, training a large number of large images requires significant time and computing power. Kruskal's algorithm (algorithm 2) has a running complexity of $\mathcal{O}(\|E\| \log(\|E\|))$ per image. Parallel execution of the algorithm (i.e. processing multiple inputs and each input is processed by a different computing core) is one way to speed up the training process. Current development of Tensorflow only supports parallel training on a GPU, which stands for graphic processing unit [1]. While the GPU is originally used for 3D game rendering, it has been exploited by machine learning libraries for general computing tasks (e.g. matrix multiplications). Parallel execution of our learning algorithm is

not applicable since algorithm 2 works with a graph data structure that is not supported by the GPU.

Another way to speed up the training process is to train on patches of large images, which is the direction that most frameworks that handle large images would choose. For a lattice graph, the number of edges is of $\mathcal{O}(\|V\|^2)$. Hence, processing n image patches requires n executions of algorithm 2 but each takes a significantly shorter amount of time compared to processing a large image. We have not yet experimented with this approach.

We have introduced different graph encodings as 1D and 2D. While the 1D encoding can be applied to an arbitrary graph, it hides the spatial relationship between vertices and requires feature engineering to recover information about the neighbors of a pixel in order to be able to perform relatively well. On the other hand, the 2D encoding can only be applied to a lattice graph (4 or 8-connected graph), but it is able to keep the spatial relationships between vertices intact and hence more expressive classifiers that employ convolution operations can be used. Future work could consider developing a version of convolutions for arbitrary graphs, where the number of edges from a vertex determines which filter will be used. This would allow us to still use a deep neural net for later steps in a hierarchical clustering process when we no longer have a 4-connected structure.

In sections 5.2 and 5.3, we present experimental results with 2D encoding and two different formulations of the hinge loss function namely the "diff" and "hit" loss function. The "diff" loss for which results are outlined in section 5.2 shows less variation in the training process and the model with "diff" loss has better reproducibility, while the minimum average Rand errors evaluated on the validation set of both models are very close to each other with the "diff" model slightly better on some trials. Regardless, both formulations lead to a lower validation Rand error, which substantiates the effectiveness of using the #same and #diff counts resulting from algorithm 2. However, we believe some adjustments on the weights that quantify the contributions of the MST edges to the Rand error along with some tuning of the hyper parameters (e.g. learning rate, number of training epochs, use of a regularization term in the loss function) can push our learning framework even further.

We have tried to implement the online gradient descent maximin learning framework outlined in [28], but we could not replicate the results of the paper for our dataset. There are many implementation details that are not presented in the paper, which hinders us from successfully reproducing a good

enough result for a comparison with the mini batch learning algorithm in [23]. Specifically, we struggle to get our Rand error on the training set anywhere near zero (which signals if our learning model is working properly). The optimizer, the learning rate, and the use of regularization are among those hidden implementation details that we would like to have had more time to explore and test on.

Our work provides initial results on the use of inference in the training loop. This idea was based on the hope that the learned affinities can be interpreted as correlations between adjoining pixels. The initial results show an average improvement of 5 percent on the validation Rand error when inference is added to the training process. Figure 5.30 shows that the distribution of the affinities predicted by the inferning and the non-inferning models is different. In this example, the median of the affinities has increased and the range between the median and the third quartile was spread out. This substantiates the hypothesis that the inference step has a positive effect in the training loop. There are many more datasets to explore and further investigation might provide more insights into how the learning objective of the inferning model impacts the final outcome.

With the 1D encoding, our learning algorithm has the advantage of applicability to any arbitrary graph structure. We can potentially train several modules that handle various levels of affinity graphs (e.g. the first level handles affinity graphs where each vertex is a pixel and the next level handles affinity graphs where each vertex is a segment created by the first step and edges denote adjacency). The overarching idea here is that undersegmenting is a smaller issue than oversegmenting. Merging segments is easier than splitting them. Hence a first step could do a cautious segmentation into small segments and later steps then merge these small segments. In an implementation, this would require us to dynamically specify the graph structure and dynamically choose the correct parameters for each specific input. While at the pixel level, the graphs handled by the first module are uniformly 4-connected (i.e. the graphs have the same structure for any input image), the graphs formed by segmentation of the first module are not uniform (i.e. the number of edges incident with a vertex is not fixed). The graphs that the second module would have to handle might have any number of edges for a vertex, as the number of adjacent segments is not fixed. This idea also needs some consideration on whether each module should be trained independently or not. Training several modules simultaneously will require a modified loss function which reflects the contribution of each module. Research

in this direction would be of practical and theoretical interest, especially in the area of dynamic neural networks, which are deep neural nets that can handle inputs whose lengths or shapes are different.

In this thesis, we have shown that considering an inference objective (e.g. the minimization of the correlation clustering energy function) in the optimization process of a training objective (e.g. the Rand error) can lead to a significantly better outcome than just considering the training objective. This thesis has provided initial results on a fairly basic data set. There is plenty of room for further research in the this area. For example, we can explore how the use of different inference methods such as the watershed cut or the normalized cut affect the performance of the training loss functions. We can also investigate whether the misclassification error is the ideal quantity to optimize, or whether some entropy function as used in decision tree design might be a better choice. All in all, this thesis has found a positive interaction between inference and learning, which encourages further exciting research in this direction.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Sharon Alpert, Meirav Galun, Ronen Basri, and Achi Brandt. Image segmentation by probabilistic bottom-up aggregation and cue integration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, June 2007.

[3] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour detection and hierarchical image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(5):898–916, May 2011.

[4] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, page 238, Washington, DC, USA, 2002. IEEE Computer Society.

[5] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56(1):89–113, Jul 2004.

[6] Kishore Bhoyar and Omprakash Kakde. Color image segmentation based on jnd color histogram. *International Journal of Image Processing (IJIP)*, 3(6):283.

[7] Norman Biggs, E Keith Lloyd, and Robin J Wilson. *Graph Theory, 1736-1936*. Oxford University Press, 1986.

[8] Camille Couprie, Leo Grady, Laurent Najman, and Hugues Talbot. Power watershed: A unifying graph-based optimization framework. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(7):1384–1399, July 2011.

[9] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[10] Tom Fawcett. An introduction to ROC analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[11] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, 2018.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] Leo Grady. Random walks for image segmentation. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (11):1768–1783, 2006.

[14] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48, 2016.

[15] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International Workshop on Artificial Neural Networks*, pages 195–201. Springer, 1995.

[16] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[17] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[18] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[19] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 Fourth International Conference on 3D Vision (3DV)*, pages 565–571. IEEE, 2016.

[20] Michael A. Nielsen. Neural networks and deep learning, 2018.

[21] Divya Pandove, Shivani Goel, and Rinkle Rani. Correlation clustering methodologies and their fundamental results. *Expert Systems*, 35(1):e12229. e12229 10.1111/exsy.12229.

[22] Sérgio Pereira, Adriano Pinto, Victor Alves, and Carlos A. Silva. Brain tumor segmentation using convolutional neural networks in MRI images. *IEEE transactions on medical imaging*, 35(5):1240–1251, 2016.

[23] Reid Porter, Diane Oyen, and Beate G. Zimmer. Learning watershed cuts energy functions. In Jón Atli Benediktsson, Jocelyn Chanussot, Laurent Najman, and Hugues Talbot, editors, *Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 497–508, Cham, 2015. Springer International Publishing.

[24] William M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.

[25] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[26] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[27] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *Departmental Papers (CIS)*, page 107, 2000.

[28] Srinivas C. Turaga, Kevin L. Briggman, Moritz Helmstaedter, Winfried Denk, and H. Sebastian Seung. Maximin affinity learning of image segmentation. In *Proceedings of the 22Nd International Conference on Neural Information Processing Systems*, NIPS'09, pages 1865–1873, USA, 2009. Curran Associates Inc.

[29] Srinivas C. Turaga, Joseph F. Murray, Viren Jain, Fabian Roth, Moritz Helmstaedter, Kevin Briggman, Winfried Denk, and H. Sebastian Seung. Convolutional networks can learn to generate affinity graphs for image segmentation. *Neural Computation*, 22(2):511–538, 2010. PMID: 19922289.

[30] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[31] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Scene parsing through ade20k dataset. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[32] Yi-Tong Zhou and Rama Chellappa. Computation of optical flow using a neural network. In *IEEE International Conference on Neural Networks*, volume 1998, pages 71–78, 1988.