Optimized U-Net for Left Ventricle Segmentation

by

Sadegh Charmchi

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

The left ventricle segmentation is an important medical imaging task necessary to measure a patient's heart pumping efficiency. Recently, convolutional neural networks (CNN) have shown great potential in achieving state-of-the-art segmentation for such applications. However, most of the research is focusing on building complicated variations of these networks with modest changes to its performance. There is little to no insights on how these CNNs work and most of them are unfortunately treated the neural network as a black box. In this thesis, the famous U-Net architecture is used to segment the left ventricle from cardiac magnetic resonance (MR) images because of its simplicity and ability to analyze images at multiple scales. Posterior analysis of the network functionality demonstrates that by replacing the first set of layers of the U-Net with fixed filters, there is little change in performance compared to its fully connected version. This optimization was achieved by performing a Fourier analysis and visualization of the convolution layers after the completion of the network training phase. This analysis allows us to discover that some early layers approximate uniform filters which can then be replaced by fixed uniform kernel weights. Furthermore, in a separate experiment by removing the middle layers of the U-Net one can reduce the number of U-Net parameters from 31 million to 0.5 million to achieve faster prediction time without compromising the performance. Experimental results and analysis are presented.

Acknowledgements

I would first like to thank my thesis advisors Dr. Pierre Boulanger of the Department of Computing Science and Dr. Kumar Punithakumar of the Department of Radiology at the University of Alberta. The door to their office was always open whenever I ran into a trouble spot or had a question about my research or writing. They consistently allowed this thesis to be my own work, but steered me in the right direction whenever they thought I needed it.

I would also like to thank the experts who were involved in SERVIER Virtual Cardiac Center for this research project: Dr. Michelle Noga and Dr. Abhilash Hareendranathan. Without their passionate participation and input, this could not have been successfully conducted.

Finally, I must express my very profound gratitude to my parents and my sister for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Task

In cardiology, segmenting the left ventricle (LV) from cardiac magnetic resonance imaging (MRI) is of prime importance in order to measure properties such as: ejection fraction, LV volume, and wall thickness to evaluate the patients' cardiovascular health and diagnose their disease. Most segmentation tasks are performed manually which is very inefficient, error-prone and consuming a large portion of the radiologists' time. The automation of segmentation is one of those essential tasks that radiologists would benefit from. Machine learning is starting to play a major role in automating the segmentation task and is the main topic of this thesis.

The process of segmentation consists of labeling each pixel in an input image as either background or left ventricle regions. In order to train a neural network to perform automatic segmentation, one must first train the network by providing examples where a set of images with their associated labels (performed by the radiologist) are used. Once the network is trained, a new unclassified image can be segmented by the neural net.

1.2 Challenges

There are numerous challenges to the left ventricle segmentation from cardiac MRI. One of them is the significant overlap of the intensity distributions of cardiac structures and the surrounding background regions. Also, the shape of the left ventricle varies significantly (big to small and back to big) during the cardiac cycle and across different slices.

The number of pixels in the background greatly exceeds the number of pixels corresponding to the left ventricle, hence making the segmentation task even more difficult. In other words, in most cases, predicting all pixels as background would yield a 90% accuracy which could be misleading. As such, the type of metrics used for determining the difference between the neural net segmentation and the one provided by the radiologist is critical.

Other challenges related to the segmentation task involve the inherent intensity variability in cine MRI which happens across different institutions, scanner, and populations. There is also an inherent noise associated with cine MRI, besides the fact that boundary and edge information could be fuzzy specifically in basal and apical slices.

1.3 Access to Training Data

Since neural network training requires a large number of examples in order to obtain accurate results, one needs to to get access to large data sets of MR images with its corresponding label maps. One of those data sets is provided by the 2009 Medical Image Computing and Computer Assisted Invention (MICCAI) challenge and another one by the 2011 Statistical Atlases and Computational Modeling of the Heart (STACOM) challenge. These challenges provide a training set with the expert ground truth contours for the left ventricle, a test set and evaluation metrics to assess the performance.

1.4 Traditional Segmentation Algorithms

Traditional segmentation algorithms for the left ventricle segmentation are reviewed in an excellent paper by Petitjean and Dacher (2011) [19]. Their review includes an analysis of: deformable models, level sets, graph cuts, knowledge-based models, atlas-based models, and thresholding algorithms. Many of those algorithms suffer from limitations such as the lack of generalization, issues with their robustness, and lower accuracy.

More generally, one can divide the recent work into semi-automated and fully-automated segmentation algorithms. An example of a semi-automated approach is from Ngo and Carneiro (2013) [26] which combines a restricted Boltzmann machines (RBMs) algorithm with a level set method. Another example of a fully-automated algorithm is the work by Queiros (2014) [21] which combines the 2D and 3D segmentation with contour propagation.

Methods that combine deep learning with deformable models have also been proposed. Unfortunately, they require multi-stage and offline training which make their application difficult and not robust. Algorithms such as the one presented in Avendi et al. (2015) [1] combines stacked auto-encoders/convolutional neural nets with deformable models.

1.5 Convolutional Neural Networks (CNNs) for Segmentation

Deep learning neural networks (DNNs) have become very popular in the past few years thanks to their great success in computer vision, natural language processing, and speech recognition tasks. One architecture of DNN, the convolutional neural networks (CNNs) have played a huge role in this success. There are a few advantages for CNNs compared to more traditional methods as they can be trained end to end if one has access to a large number of examples. These examples must combine input images with ground truth labels in order for the network to learn how to map these inputs to the outputs. Once trained, a new image can be presented to the CNN and the predicted output will produce a successful segmentation.

Training a CNN using this procedure means that there is almost no need for prepossessing, post processing and hand-crafted features allowing us to save a lot of time that can be spent on other tasks.

Some of the famous CNN architectures include AlexNet (Krizhevsky et al., 2012) [15], VGGNet (Simonyan and Zisserman, 2015) [24], GoogLeNet (Ioffe and Szegedy, 2015) [25], and ResNet (He et al., 2015) [10] which all have been used to obtain state-of-the-art image classification results.

1.5.1 CNNs for Segmentation

The common architecture for a CNN consists of multiple layers of convolution, pooling and, fully connected layers. U-Net [22] is one of those CNN architectures that have been successfully applied to medical image segmentation (the paper has been cited 2000 times so far). The U-Net architecture consists of numerous encoding and decoding layers. The encoding part of the network consists of numerous convolution and MaxPool layers and the decoding part consists of de-convolution and up-sampling layers. More details about the architecture will be described later.

1.6 Thesis Contributions

Our contribution in this thesis is two folds:

1. First, we proposed a shallow version of the U-Net architecture which works as well as the original deep version for segmenting the left ventricles from cardiac MRIs.

2. Second, we show some of the parameters in the first few convolution layers do not need to be trained and can be replaced by uniform filters without losing accuracy. This is shown by visualizing the Fourier transform of the convolution filters layers by layers.

1.7 Outline

Chapter 2 will discuss some background information about neural networks and deep learning and ends with a brief introduction to the U-Net.

Chapter 3 proposes the main ideas in this thesis on how to make the U-Net more optimized, and the two variations of the U-Net is discussed here.

Chapter 4 presents our results and experiments backing up our evidence on making the U-Net more efficient and simpler while maintaining the accuracy.

Finally, Chapter 5 concludes the thesis with a summary and a direction for the future work.

Chapter 2

Literature Review

In this chapter, we provide the details on a few concepts in CNN that will help the reader understand the materials presented in the rest of the thesis. First we describe the building blocks that construct a CNN, then we will explain some challenges on how to make this network to work and finally we will briefly explain the U-Net applied for segmentation.[1] [2]

2.1 Neural Networks and Deep Learning

The simplest form of a neural network is the perceptron. In a perceptron, each input values $x_1, \ldots, x_D$ has an associated weight defined as $w_1, \ldots, w_D$. The output of a perceptron is calculated as $y = f(z)$ where z is the weighted sum of the inputs transformed by a function f called the activation function.

One can visualize neural networks as a directed graph $G = (V, E)$ where V is a set of nodes and E a set of edges connecting the nodes. The mapping $(u, v) \in E$ means that a directed edge from node u to v exists within the graph. In a network graph, given two units u and v, a directed edge from u to v means that the output of unit u is used by unit v as input. A simple perceptron is illustrated in Figure 2.1.

[1]The background materials and figures in this section has been mostly adopted from `https://github.com/davidstutz/seminar-convolutional-neural-networks` under Creative Commons Licence.

[2]Materials in the optimization section has been mostly adopted from `http://ruder.io/optimizing-gradient-descent/`

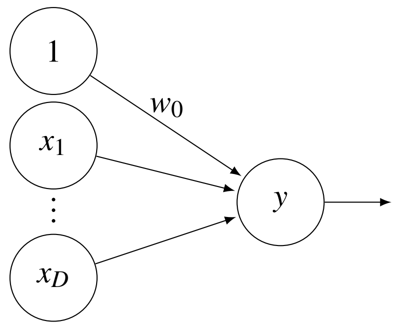


Figure 2.1: A perceptron maps all inputs $w_0, x_1 \dots, x_D$ to the actual input z, and an activation function f which is applied on the actual input to form the output $y = f(z)$. Here, w_0 represents an external input called bias and $x_1, \dots, x_D$ are inputs from other units of the network. In a network graph, each unit is labeled according to its output. Therefore, to include the bias w_0 as well, a dummy unit (see section 2.1.1) with value 1 is included.

2.1.1 Multilayer Perceptron

A multi-layer perceptron, as shown in Figure 2.2, consists of D input units, C output units, and multiple hidden layers. It is primarily a multilayer perceptron that comprises an input layer, an output layer and L hidden layers. Normally the input layer is not counted in the number of layers as illustrated in Figure 2.2 where the number of layers is $L+1$.

The i^{th} unit in layer l computes the output

$$y_i^{(l)} = f\left(z_i^{(l)}\right) \quad \text{where} \quad z_i^{(l)} = \sum_{k=1}^{m^{(l-1)}} w_{i,k}^{(l)} y_k^{(l-1)} + w_{i,0}^{(l)} \tag{2.1}$$

where $w_{i,k}^{(l)}$ denotes the weighted connection from the k^{th} unit in layer $(l-1)$ to the i^{th} unit in layer l, and $w_{i,0}^{(l)}$ can be regarded as external input to the unit and is referred to as bias. Here, $m^{(l)}$ denotes the number of units in layer l, such that $D = m^{(0)}$ and $C = m^{(L+1)}$. For simplicity, the bias can be regarded as weight when introducing a dummy unit $y_0^{(l)} := 1$ in each layer and the previous equation can be simplified as follows:

$$z_i^{(l)} = \sum_{k=0}^{m^{(l-1)}} w_{i,k}^{(l)} y_k^{(l-1)} \quad \text{or} \quad z^{(l)} = w^{(l)} y^{(l-1)} \tag{2.2}$$

where $z^{(l)}$, $w^{(l)}$ and $y^{(l-1)}$ denote the corresponding vector and matrix representations of the actual inputs $z_i^{(l)}$, the weights $w_{i,k}^{(l)}$ and the outputs $y_k^{(l-1)}$, respectively. This vectorized form will turn out to be very useful both in implementing these networks efficiently and also understanding the more abstract blocks as we will see

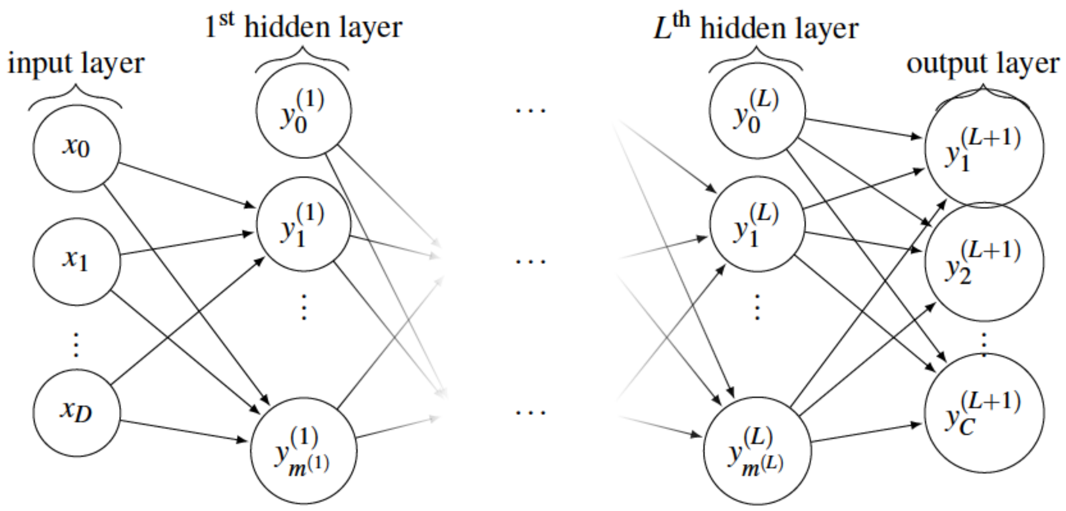


Figure 2.2: Network graph of a $(L+1)$-layer perceptron with D input units and C output units. The lth hidden layer contains $m^{(l)}$ hidden units.

later.

Overall, a multilayer perceptron represents a function

$$y(\cdot, w) : \mathbb{R}^D \to \mathbb{R}^C, x \mapsto y(x, w) \tag{2.3}$$

where the output vector $y(x, w)$ comprises the output values $y_i(x, w) := y_i^{(L+1)}$ and w is the vector of all weights within the network.

If there are more than three hidden layers, then we call that network a deep neural network [2]. Training of these deep neural networks is challenging [2].

2.1.2 Activation Functions

When it comes to activation functions, there are a few options that one can pick from. The most common activation function these days is called Rectified Linear Unit (ReLU):

$$ReLU(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}. \tag{2.4}$$

More traditional activation functions include threshold functions and piece-wise linear which both have some drawbacks. First, for network training, we may need the activation function to be differentiable. Second, nonlinear activation functions are preferable due to the additional non-linearity they introduce which helps with function approximation.

The most common type of activation function is the sigmoid function which is defined as:

$$\sigma(z) = \frac{1}{1+\exp(-z)}. \tag{2.5}$$

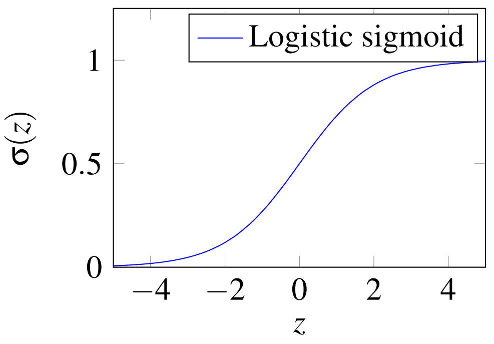
This s-shaped function is differentiable as well as monotonic. The hyperbolic tangent $\tanh(z)$ is also used as a linear transformation of the logistic sigmoid onto the interval $[-1,1]$. Note that both activation functions are saturating as input approaches large positive and negative number. As we shall see later that this signal saturation is responsible to the well known vanishing gradient problem when training the deep neural networks.

In classification tasks where one have an input vector x of D dimensions and the goal is to assign x to one of C discrete classes, a softmax activation function for output units is used to interpret the output values as probabilities. In other words, the outputs $y_i^{(L+1)}$, $1 \leq i \leq C$, can be interpreted as probabilities as they lie in the interval $[0,1]$ and sum to 1 [5]. Then, the output of the i^{th} unit in the output layer is given by

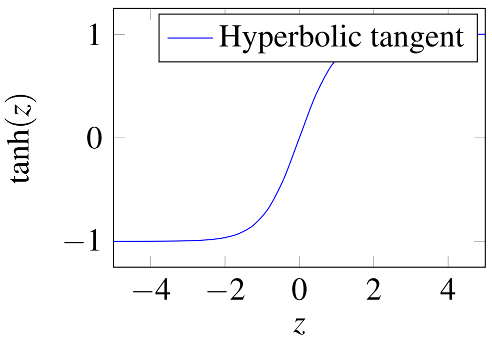
$$\sigma(z^{(L+1)}, i) = \frac{\exp(z_i^{(L+1)})}{\sum_{k=1}^{C} \exp(z_k^{(L+1)})}. \tag{2.6}$$

Experiments in [8] show that the logistic sigmoid as well as the hyperbolic tangent perform rather poorly in deep learning. Better performance is reported using the softsign activation function:

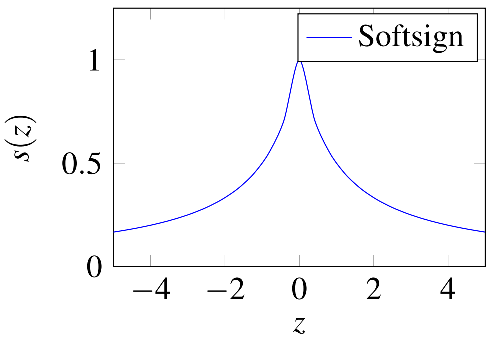
$$s(z) = \frac{1}{1+|z|}. \tag{2.7}$$



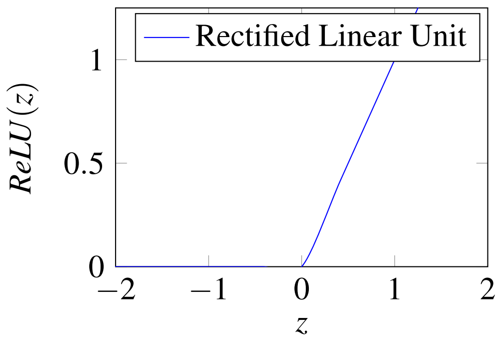
(a) Logistic sigmoid



(b) Hyperbolic tangent



(c) Softsign



(d) Rectified Linear Unit

Figure 2.3: Commonly used activation functions include the logistic sigmoid $\sigma(z)$ defined in equation (2.5) and the hyperbolic tangent $tanh(z)$. More recently used activation functions are the softsign of equation (2.7) and the rectified linear unit.

In [15], a non-saturating activation function is used:

$$r(z) = \max(0, z). \tag{2.8}$$

Hidden units using the activation function in equation (2.8) are called rectified linear units. Some of the well-known activation functions are shown in Figure 2.3

2.1.3 Training Neural Networks

Training a neural network is the process of finding the best possible set of weights that maps the training data to the output. It is essentially finding a function g that approximates this mapping using an optimization algorithm. The training set defined as:

$$T_S := \{(x_n, t_n) : 1 \leq n \leq N\} \tag{2.9}$$

which comprises both input values x_n and corresponding desired output values such that $t_n \approx g(x_n)$.

Objective Functions

During the training, the weights w of the neural network are adjusted in order to minimize a chosen objective function which can be described as the difference between network output $y(x_n)$ and desired target output t_n. This difference between the actual and predicted outputs can be defined in a different way. Popular choices for classification include the sum-of-squared error defined as:

$$E(w) = \sum_{n=1}^{N} E_n(w) = \sum_{n=1}^{N} \sum_{k=1}^{C} (y_k(x_n, w) - t_{n,k})^2, \tag{2.10}$$

or the cross-entropy error measure defined as:

$$E(w) = \sum_{n=1}^{N} E_n(w) = \sum_{n=1}^{N} \sum_{k=1}^{C} t_{n,k} \log(y_k(x_n, w)), \tag{2.11}$$

where $t_{n,k}$ is the k^{th} entry of the target value t_n and C is the number of classes.

For the left ventricle segmentation algorithm, we will not use any of these objective functions as we shall see they do not perform well. A Dice score will be used instead of the above functions and we will define the Dice score in chapter 4 where we discuss our experiments and results.

Training Procedures

There are three different ways to perform the training of a neural network using the training data:

Stochastic training An input value is chosen at random and the network weights are updated based on the error $E_n(w)$.

Full-Batch training All input values are processed and the weights are updated based on the overall error $E(w) = \sum_{n=1}^{N} E_n(w)$.

Online training Every input value is processed only once and the weights are updated using the error $E_n(w)$.

A common practice is to combine stochastic training and batch training:

Mini-batch training A random subset $M \subseteq \{1, \ldots, N\}$ (mini-batches) of the training set is processed and the weights are updated based on the cumulative error $E_M(w) := \sum_{n \in M} E_n(w)$.

Parameter Optimization

Considering stochastic training, one needs to minimize E_n with respect to the network weights w. In most cases, there are numerous of local minima where the network can converge into but our real goal is to find the global minimum. The necessary criterion can be written as

$$\frac{\partial E_n}{\partial w} = \nabla E_n(w) \overset{!}{=} 0 \tag{2.12}$$

where ∇E_n is the gradient of the error E_n.

Due to the complexity of behaviour of the error E_n, a closed-form solution is not possible and one must rely on an iterative approach. Let $w[t]$ denote the weight vector in the t^{th} iteration. In each iteration, one can compute a weight update $\Delta w[t]$ and update the weights accordingly [5, p. 236-237]:

$$w[t+1] = w[t] + \Delta w[t]. \tag{2.13}$$

From the field of unconstrained optimization, we have several optimization techniques at our disposal. Gradient descent is a first-order method. Gradient descent uses only information of the first derivative of E_n and can be used in combination with error back-propagation as described in section 2.1.3, whereas Newton's method is a second-order method and needs to evaluate the Hessian matrix H_n of

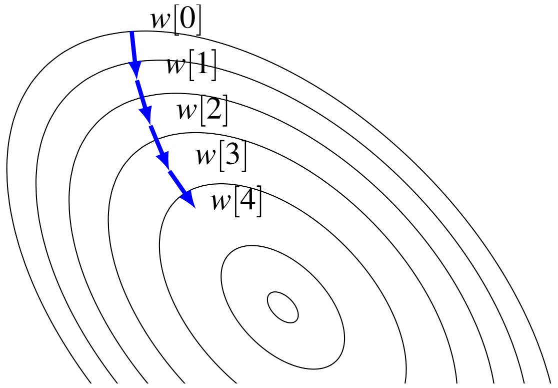


Figure 2.4: Illustrated using a quadratic function to minimize, the idea of gradient descent is to follow the negative gradient at the current position as it describes the direction of the steepest descent. The learning rate γ describes the step size taken in each iteration step. Gradient descent is a first-order optimization technique.

E_n[3] (or an appropriate approximation of the Hessian matrix) in each iteration step. In this section, we only describe the gradient descent algorithm since this is the preferred method to perform the optimization in most neural networks. Newton method is rarely used due to its computational complexity in evaluating the Hessian matrix.

Gradient descent Gradient descent is motivated by the idea to take a step in the direction of the steepest descent, that is the direction of the negative gradient, to reach a local minimum [4, p. 263-267]. This concept is illustrated by Figure 2.4. Using gradient decent, the weight update is given by:

$$\Delta w[t] = -\gamma \frac{\partial E_n}{\partial w[t]} = -\gamma \nabla E_n(w[t]) \tag{2.14}$$

where γ is the learning rate. As discussed in [5, p.263-272], this approach has several difficulties. For example, how to choose the learning rate to get fast convergence but at the same time avoid oscillation? Oscillation occurs when the chosen learning rate is too large resulting in successive overstepping over the local minimum. This is called overshooting of the gradient. In short, too small of a step will slow down the convergence and too big of a step will overshoot the gradient and will not lead to the global optimum.

[3]The Hessian matrix H_n of a the error E_n is the matrix of second-order partial derivatives: $(H_n)_{r,s} = \frac{\partial^2 E_n}{\partial w_r \partial w_s}$

Weight Initialization

As we use an iterative optimization technique, the initialization of the weights w is crucial. In [8], an initialization scheme called normalized initialization is introduced. We choose the weights randomly in the range of:

$$-\frac{\sqrt{6}}{\sqrt{fan\_in + fan\_out}} < w_{i,j}^{(l)} < \frac{\sqrt{6}}{\sqrt{fan\_in + fan\_out}}. \tag{2.15}$$

The derivation of this initialization scheme can be found in [8]. Experimental results in [8] demonstrate an improved learning rate when using normalized initialization.

An alternative to these weight initialization schemes is given by He et al [9]:

$$-\frac{\sqrt{6}}{\sqrt{fan\_in}} < w_{i,j}^{(l)} < \frac{\sqrt{6}}{\sqrt{fan\_in}}. \tag{2.16}$$

Where fan_in is the the number of input units in the weight tensor and fan_out is the number of output units in the weight tensor (i.e. number of units in the previous layer and the next layer respectively).

Back-propagation Error

Algorithm 2.1.1, proposed in [23], is used to evaluate the gradient $\nabla E_n(w[t])$ of the error function E_n in each iteration step. More details as well as a thorough derivation of the algorithm can be found in [4] or [23].

Algorithm 2.1.1 (Back-propagation Error)

1. *Propagate the input value x_n through the network to get the actual input and output of each unit.*

2. *Calculate the so called errors $\delta_i^{(L+1)}$ [5, p. 241-245] for the output units:*

$$\delta_i^{(L+1)} := \frac{\partial E_n}{\partial y_i^{(L+1)}} f'(z_i^{(L+1)}). \tag{2.17}$$

3. Determine $\delta_i^{(l)}$ for all hidden layers l by using error backpropagation:

$$\delta_i^{(l)} := f'(z_i^{(l)}) \sum_{k=1}^{m^{(l+1)}} w_{i,k}^{(l+1)} \delta_k^{(l+1)}. \tag{2.18}$$

4. Calculate the required derivatives:

$$\frac{\partial E_n}{\partial w_{j,i}^{(l)}} = \delta_j^{(l)} y_i^{(l-1)}. \tag{2.19}$$

2.1.4 Regularization

It has been shown that multilayer perceptrons with at least one hidden layer can approximate any target mapping up to an arbitrary accuracy [12] which is also known as the universal approximation theorem. Thus, the training data may be over-fitted, that is the training error may be very low on the training set but high on unseen data [2]. Regularization describes the task to avoid over-fitting to give better generalization performance, meaning that the trained network should also perform well on unseen data. Therefore, the training set is usually split up into an actual training set and a validation set. The neural network is then trained using the new training set, and its generalization performance is evaluated on the validation set. Finally, once all the hyper-parameters are determined using the validation set, the final performance is evaluated on the test set.

There are different methods to perform regularization. Often, the training set is augmented by introducing certain invariances that the network is expected to learn [15]. Other methods add a regularization term to the error measure aiming to control the complexity and form of the solution [4]:

$$\hat{E}_n(w) = E_n(w) + \eta P(w) \tag{2.20}$$

where $P(w)$ influences the form of the solution and η is a balancing parameter on how much we want to regularize the optimization solution.

L_p-Regularization

A popular example of L_p-regularization is the L_2-regularization which is often referred to as weight decay:

$$P(w) = \|w\|_2^2 = w^T w. \tag{2.21}$$

The idea is to penalize large weights as they tend to result in over-fitting [4]. In general, arbitrary p can be used to perform L_p-regularization. Another example sets $p = 1$ where the norm $\|\cdot\|_1$ is defined by $\|w\|_1 = \sum_{k=1}^{W} |w_k|$ and W is the dimension of the weight vector w. This is to enforce sparsity of the weights, *i.e.,* many of the weights should vanish:

$$P(w) = \|w\|_1. \tag{2.22}$$

Stopping Criteria

While the error on the training set tends to decrease with the number of iterations, the error on the validation set usually starts to rise again once the network starts to over-fit the training set. To avoid over-fitting, training can be stopped as soon as the error on the validation set reaches a minimum, *i.e.,* before the error on the validation set rises again [4]. This method is called early stopping.

Dropout

In [11], another regularization technique based on observation of the human brain is proposed. Whenever the neural network is given a training sample, each hidden unit is skipped with probability $\frac{1}{2}$. This method can be interpreted in different ways [11]. First, units cannot rely on the presence of other units. Second, this method leads to the training of multiple different networks simultaneously. Thus, dropout can be interpreted as model averaging which essentially tries to reduce the error by averaging the prediction of different models [11].

Weight Sharing

The idea of weight sharing was introduced in [23] in the context of the T-C problem[4]. Weight sharing describes the idea of different units within the same layer to use identical weights. This can be interpreted as a regularization method as the complexity of the network is reduced and prior knowledge may be incorporated into the network architecture.

When using weight sharing, error back-propagation can be applied as usual, however, equation (2.19) changes to

$$\frac{\partial E_n}{\partial w_{j,i}^{(l)}} = \sum_{k=1}^{m^{(l)}} \delta_k^{(l)} y_i^{(l-1)} \tag{2.23}$$

when assuming that all units in layer l share the same set of weights, that is $w_{j,i}^{(l)} = w_{k,i}^{(l)}$ for $1 \leq j,k \leq m^{(l)}$. Nevertheless, equation (2.19) still needs to be applied in the case that the errors need to be propagated to preceding layers [5].

2.2 Convolutional Neural Networks

Although traditional neural networks can be applied to computer vision tasks, to get good generalization performance, convolutional neural networks aim to use spatial information between the pixels of an image to get state-of-the-art results in vision tasks. Therefore, they are based on discrete convolution. After introducing discrete convolution, we discuss the basic components of convolutional neural networks.

2.2.1 Convolution

For simplicity we assume a gray scale image to be defined by a function:

$$I : \{1,\ldots,n_1\} \times \{1,\ldots,n_2\} \to W \subseteq \mathbb{R}, (i,j) \mapsto I_{i,j} \tag{2.24}$$

[4]The T-C problem describes the task of classifying images into those containing a "T" and those containing a "C" independent of position and rotation [23].

such that the image I can be represented by an array of size $n_1 \times n_2$ and often, W will be the set $\{0, \ldots, 255\}$ representing an 8-bit channel. Then, a color image can be represented by an array of size $n_1 \times n_2 \times 3$ assuming three color channels, for example RGB. Given the filter $K \in \mathbb{R}^{2h_1+1 \times 2h_2+1}$, the discrete convolution of the image I with filter K is given by

$$(I * K)_{r,s} := \sum_{u=-h_1}^{h_1} \sum_{v=-h_2}^{h_2} K_{u,v} I_{r+u,s+v} \tag{2.25}$$

where the filter K is given by:

$$K = \begin{pmatrix} K_{-h_1,-h_2} & \cdots & K_{-h_1,h_2} \\ \vdots & K_{0,0} & \vdots \\ K_{h_1,-h_2} & \cdots & K_{h_1,h_2} \end{pmatrix}. \tag{2.26}$$

Note that the behavior of this operation towards the borders of the image needs to be defined properly. For example, consider a gray scale image of size $n_1 \times n_2$. When applying an arbitrary filter of size $(2h_1 + 1) \times (2h_2 + 1)$ to the pixel at location $(1, 1)$ the sum of equation (2.25) includes pixel locations with negative indices. To solve this problem, several approaches can be considered, one could be padding the image in some way or applying the filter only for locations where the operation is defined properly resulting in the output array being smaller than the image.

A commonly used filter for smoothing is the discrete Gaussian filter $K_{G(\sigma)}$ [7] which is defined by:

$$\left(K_{G(\sigma)}\right)_{r,s} = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{r^2 + s^2}{2\sigma^2}\right) \tag{2.27}$$

where σ is the standard deviation of the Gaussian distribution [7]. Although it may sound unusual but the Gaussian filter that has been traditionally used in image processing for a long time is basically a convolution with the above filter (kernel). This is to say that this convolution idea is not a new deep learning concept and has been around for a few years.

2.2.2 Layers

We follow [13] and introduce the different types of layers used in convolutional neural networks. Based on these layers, complex architectures as used for classification in [15] can be built by stacking multiple layers.

Convolutional Layers

Let layer l be a convolutional layer. Then, the input of layer l comprises $m_1^{(l-1)}$ feature maps from the previous layer, each of size $m_2^{(l-1)} \times m_3^{(l-1)}$. In the case where $l = 1$, the input is a single image I consisting of one or more channels. This way, a convolutional neural network directly accepts raw images as input. The output of layer l consists of $m_1^{(l)}$ feature maps of size $m_2^{(l)} \times m_3^{(l)}$. The i^{th} feature map in layer l, denoted $Y_i^{(l)}$, is computed as

$$Y_i^{(l)} = B_i^{(l)} + \sum_{j=1}^{m_1^{(l-1)}} K_{i,j}^{(l)} * Y_j^{(l-1)} \tag{2.28}$$

where $B_i^{(l)}$ is a bias matrix and $K_{i,j}^{(l)}$ is the filter of size $(2h_1^{(l)} + 1) \times (2h_2^{(l)} + 1)$ connecting the j^{th} feature map in layer $(l - 1)$ with the i^{th} feature map in layer l. Note the difference between a feature map $Y_i^{(l)}$ comprising $m_2^{(l)} \cdot m_3^{(l)}$ units arranged in a two-dimensional array and a single unit $y_i^{(l)}$ is used in the multilayer perceptron. As mentioned above, $m_2^{(l)}$ and $m_3^{(l)}$ are influenced by border effects. When applying the discrete convolution, only in the so called valid region of the input feature maps, *i.e.,* only for pixels where the sum of equation (2.25) is defined properly, the output feature maps have size

$$m_2^{(l)} = m_2^{(l-1)} - 2h_1^{(l)} \quad \text{and} \quad m_3^{(l)} = m_3^{(l-1)} - 2h_2^{(l)}. \tag{2.29}$$

Often, the filters used for computing a fixed feature map $Y_i^{(l)}$ are the same, *i.e.,* $K_{i,j}^{(l)} = K_{i,k}^{(l)}$ for $j \neq k$. In addition, the sum in equation (2.28) may also run over a subset of the input feature maps.

To relate the convolutional layer and its operation as defined by equation (2.28) to the multilayer perceptron, we rewrite the above equation. Each feature map $Y_i^{(l)}$

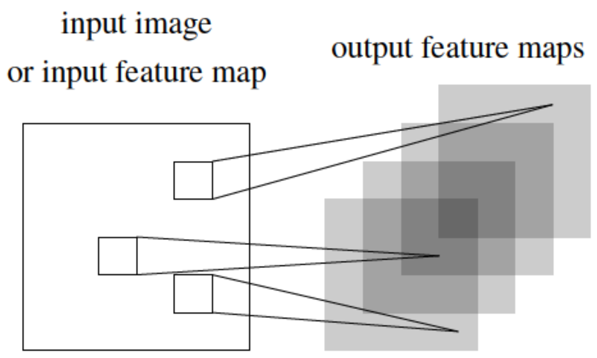


Figure 2.5: Illustration of a single convolutional layer. If layer l is a convolutional layer, the input image (if $l = 1$) or a feature map of the previous layer is convolved by different filters to yield the output feature maps of layer l.

in layer l consists of $m_2^{(l)} \cdot m_3^{(l)}$ units arranged in a two-dimensional array. The unit at position (r,s) computes the output:

$$\left(Y_i^{(l)}\right)_{r,s} = \left(B_i^{(l)}\right)_{r,s} + \sum_{j=1}^{m_1^{(l-1)}} \left(K_{i,j}^{(l)} * Y_j^{(l-1)}\right)_{r,s} \tag{2.30}$$

$$= \left(B_i^{(l)}\right)_{r,s} + \sum_{j=1}^{m_1^{(l-1)}} \sum_{u=-h_1^{(l)}}^{h_1^{(l)}} \sum_{v=-h_2^{(l)}}^{h_2^{(l)}} \left(K_{i,j}^{(l)}\right)_{u,v} \left(Y_j^{(l-1)}\right)_{r+u,s+v}. \tag{2.31}$$

The trainable weights of the network can be found in the filters $K_{i,j}^{(l)}$ and the bias matrices $B_i^{(l)}$.

As we will see in section 2.2.2, subsampling is used to decrease the effect of noise and distortions. Subsampling can be done using so called skipping factors $s_1^{(l)}$ and $s_2^{(l)}$. The basic idea is to skip a fixed number of pixels, both in horizontal and in vertical direction, before applying the filter again. With skipping factors as above, the size of the output feature maps is given by:

$$m_2^{(l)} = \frac{m_2^{(l-1)} - 2h_1^{(l)}}{s_1^{(l)} + 1} \quad \text{and} \quad m_3^{(l)} = \frac{m_3^{(l-1)} - 2h_2^{(l)}}{s_2^{(l)} + 1}. \tag{2.32}$$

Non-Linear Layers

If layer l is a non-linear layer, its input is given by $m_1^{(l)}$ feature maps and its output comprises again $m_1^{(l)} = m_1^{(l-1)}$ feature maps, each of size $m_2^{(l-1)} \times m_3^{(l-1)}$ such that

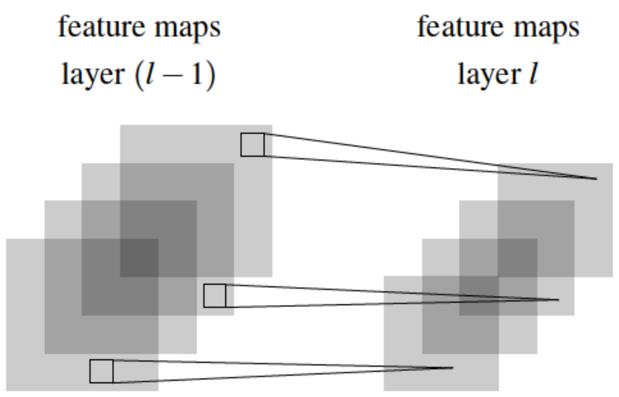


Figure 2.6: Illustration of a pooling and subsampling layer. If layer l is a pooling and subsampling layer and given $m_1^{(l-1)} = 4$ feature maps of the previous layer, all feature maps are pooled and subsampled individually. Each unit in one of the $m_1^{(l)} = 4$ output feature maps represents the average or the maximum within a fixed window of the corresponding feature map in layer $(l-1)$.

$m_2^{(l)} = m_2^{(l-1)}$ and $m_3^{(l)} = m_3^{(l-1)}$, given by

$$Y_i^{(l)} = f\left(Y_i^{(l-1)}\right). \tag{2.33}$$

where f is the activation function used in layer l and operates point wise. In [13] additional gain coefficients are added:

$$Y_i^{(l)} = g_i f\left(Y_i^{(l-1)}\right). \tag{2.34}$$

Almost always a convolution layer is followed by a non-linear layer, and most of the time, this non-linear layer is a ReLU activation function as discussed previously. Note that in [13], this constitutes a single layer whereas we separate the convolutional layer and the non-linearity layer.

Feature Pooling and Sub-sampling Layers

The motivation of subsampling the feature maps obtained by previous layers is to increase the robustness to noise and distortions [13]. Reducing the resolution can be accomplished in different ways. In [13], this approach is combined with pooling and performed in a separate layer, while in the traditional convolutional neural networks, the subsampling is performed by applying skipping factors.

Let l be a pooling layer. Its output comprises $m_1^{(l)} = m_1^{(l-1)}$ feature maps of reduced size. In general, the pooling operates by placing windows at non-overlapping positions in each feature map and keeping one value per window such that the feature maps are subsampled. We distinguish two types of pooling:

Average pooling When using this method, we take the average of all the pixels that fall under the window. This operation is called average pooling and the layer is denoted by P_{A}.

Max pooling For max pooling, the maximum value of each window is taken. The layer is denoted by P_{M}.

Max pooling is usually used to get faster convergence during training and also to increase the robustness of the network. Both average and max pooling can also be applied using overlapping windows of size $2p \times 2p$ which are placed q units apart. Then, the windows overlap if $q < p$. This is found to reduce the chance of overfitting the training set [15].

Fully Connected Layer

Let layer l be a fully connected layer. If layer $(l-1)$ is a fully connected layer, as well, we may apply equation (2.2). Otherwise, layer l expects $m_1^{(l-1)}$ feature maps of size $m_2^{(l-1)} \times m_3^{(l-1)}$ as input and the ith unit in layer l computes:

$$y_i^{(l)} = f\left(z_i^{(l)}\right) \quad \text{with} \quad z_i^{(l)} = \sum_{j=1}^{m_1^{(l-1)}} \sum_{r=1}^{m_2^{(l-1)}} \sum_{s=1}^{m_3^{(l-1)}} w_{i,j,r,s}^{(l)} \left(Y_j^{(l-1)}\right)_{r,s}. \tag{2.35}$$

where $w_{i,j,r,s}^{(l)}$ denotes the weight connecting the unit at position (r,s) in the jth feature map of layer $(l-1)$ and the ith unit in layer l. In practice, convolutional layers are used to learn a feature hierarchy and one or more fully connected layers are used for classification purposes based on the computed features, so basically the fully connected layer usually only appears once as the last layer in the network [16]. Note that a fully-connected layer already includes the non-linearities while for a convolutional layer the non-linearities are separated in their own layers.

Up-Convolution Layer

There are different terminologies used for Up-Convolution layer such as Up-Sampling, Deconvolution and Transpose-Convolution which all have the same meaning.

In U-Net, the transpose convolution simply reduces the depth of the input cube and adds to the width and height of the cube. In other words, say we have a $16 \times 16 \times 128$ cube as input where 128 is the depth and 16 is the number of rows and columns for each image stacked on top of each other. After applying transpose convolution, the size of the output cube becomes $32 \times 32 \times 64$. More explanations on this concept will be discussed later.

2.2.3 Modern ConvNet Architectures

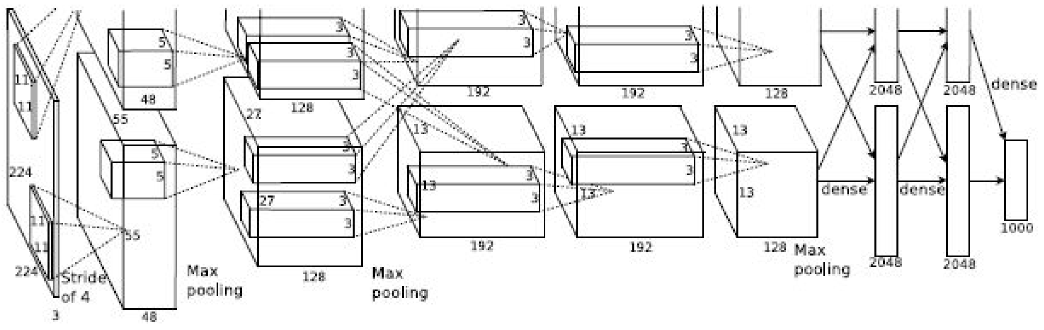


Figure 2.7: AlexNet Architecture

As example of a modern convolutional neural network, we explore the architecture used in [15] which gives excellent performance on the ImageNet Dataset [27]. This architecture is illustrated in Figure 2.7. The architecture is organized into five convolutional layers each followed by a rectified linear unit non-linearity layer, brightness normalization and overlapping pooling. Classification is done using three additional fully-connected layers. To avoid over-fitting, [15] uses dropout as regularization technique. Details can be found in [15].

2.3 Optimization Methods and Parameter Tuning

Deep neural networks are very sensitive to hyper-parameters, *e.g.,* learning rate, initialization, number of layers, number of neurons in each layer, size of the filter (in the case of a CNN) and many more. Hence, a good optimization approach and tuning the hyper-parameters is of utmost importance.

In this section, we give a brief explanation of a few recent optimization algorithms and we also introduce methods for searching the hyper-parameter space that could possibly lead to a set of parameters of choice for our final model.

2.3.1 Modern Optimization Algorithms

Almost all modern neural network optimization algorithms are based on the gradient descent algorithm. In this section, we review a few of those gradient descent optimization algorithms.

Batch Gradient Descent

Batch gradient decent computes the gradient of the cost function with respect to parameters θ. In this case, the parameters are updated only after the algorithm goes over all the examples in the training set.

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta) \tag{2.36}$$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Stochastic Gradient Descent (SGD)

In contrast to batch gradient descent, SGD performs an update after each training example. One disadvantage of stochastic gradient descent is its frequent updates which brings in high variance when updating the parameters, hence causing the objective function to fluctuate. This could be both an advantage and a disadvantage as the fluctuation can help jump over the hills and find a better local minimum and ultimately the global optimum, but at the same time the overshooting can cause the optimization to fail. However, it has been shown that if we keep decreasing the learning rate, the SGD's performance is similar to batch gradient descent.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{2.37}$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function , example , params)
    params = params - learning_rate * params_grad
```

Mini-batch Gradient Descent

In mini-batch gradient descent, the algorithm performs an update after each mini-batch. This has two benefits: first, it reduces the variance in the parameter update. Second, most of the deep learning libraries have efficient computation of gradient with respect to a mini-batch. Mini-batch size can vary anywhere from 8 to 256 depending on the memory constraints and applications. In our experiments the mini-batch size is set to 16. The term SGD can also sometimes refer to mini-batch gradient descent as this is the algorithm of choice when training the neural networks most of the time. The iterative equation is defined as:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.38}$$

An example code of a mini-batches of size 50 is given below:

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data , batch_size=50):
    params_grad = evaluate_gradient(loss_function , batch , params)
    params = params - learning_rate * params_grad
```

However, there are a few challenges to vanilla mini-batch gradient descent as explained below:

- choosing the learning rate is important. Too small of a learning rate can lead to a very slow convergence. Too big of a learning rate can lead to fluctuation around the minimum or even divergence.
- All parameter updates are done using the same learning rate. The problem arises when we have parameters that have very different frequencies and our data is sparse.
- SGD has trouble escaping from the saddle points. Saddle points happen where we have an uphill in one dimension and downhill in another dimension. In this case, the gradient is zero in all directions because of the plateau.

Momentum

SGD has trouble navigating through a space where the surface curve is much steeper in one dimension than in another (aka ravines). Momentum [20] helps with accelerating the gradient in one direction and damping the oscillation in the other direction as shown in Figure 2.8.

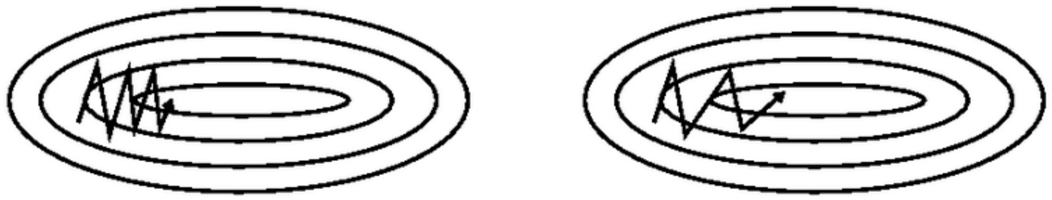


Figure 2.8: Left: SGD without mometum. Right: SGD with momentum.

This is done by adding a fraction γ of the previous update vector to the current update vector:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{2.39}$$

Imagine a ball going downhill and gaining momentum as it rolls down and rolling faster and faster. So the momentum term increases if a gradient points to the same direction, and the momentum term decreases if the gradient keeps changing directions.

Nesterov Accelerated Gradient

Nesterov Accelerated Gradient (NAG) [18] provides a notion of look-ahead to the ball. This means that the ball will roll down smarter; instead of blindly following the slope rolling downhill, it will look ahead and slow down before the hill slopes up again. Before we used the term γv_{t-1} to move the parameters θ. Hence by computing $\theta - \gamma v_{t-1}$, we can get an approximation of where our parameters are going to be next. Therefore, the lookahead is done by calculating the gradient w.r.t. the approximate future position of our parameters instead of the current position of parameters.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \\ \theta &= \theta - v_t \end{aligned} \tag{2.40}$$

Adagrad

One disadvantage of the algorithms discussed previously is that they update all the parameters with the same importance. We would rather have an algorithm that updates individual parameters based on their importance, some parameters will get larger updates while some will get smaller updates. Adagrad [6] does exactly the above as it adapts the learning rate to parameters such that infrequent features get larger learning rate for larger updates and frequently occurring features get smaller learning rate for smaller updates. Therefore it works very well when one is dealing with sparse data.

Adagrad uses a different learning rate for each parameter θ_i at time step t. In our notation, g_t is the gradient at time step t and the partial derivative of the objective function with respect to parameter θ_i is denoted by $g_{t,i}$ and is equal to:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}). \tag{2.41}$$

Hence, the SGD update rule for each parameter θ_i at time step t goes as follows:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}. \tag{2.42}$$

In Adagrad, the past gradients that have been computed for θ_i affects the learning rate in which we update parameter θ_i at time step t:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} \cdot g_{t,i}. \tag{2.43}$$

In the above equation, $G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix where each element i,i on the diagonal of the matrix represents the sum of square of the gradients for each parameter θ_i up to time step t.

Vectorizing the equation above with the matrix-vector operation $\odot$ yields the following:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t \tag{2.44}$$

The main weakness of the Adagrad is the increasing term in the denominator of the learning rate. Since that term is the sum of squared of the gradients, it is always positive and increasing, it can get bigger and bigger to a point where learning rate approaches zero and the learning stops as the training goes on . Following algorithms aim to address this issue.

Adadelta

In order to address the problem of ever decreasing learning rate that we had with Adagrad, Adadelta [28] limits the window of sum of squared gradients computed to w. However instead of inefficiently storing the w past squared gradients, sum of gradients is defined recursively as a decaying average of all past squared gradients. So the running average $E[g^2]_t$ at time step t is calculated based on previous average $E[g^2]_{t-1}$ and the current gradient g_t^2. γ could be set to 0.9.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{2.45}$$

Rewriting the vanilla SGD update in terms of the parameter update vector $\Delta\theta_t$ gives the following:

$$\begin{aligned} \Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned}. \tag{2.46}$$

So for Adagrad:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t, \tag{2.47}$$

and for Adadelta:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t. \tag{2.48}$$

The denominator is basically the root-mean-squared error of the gradient, which can be re-written as:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t. \tag{2.49}$$

The problem with the previous update rule is that the unit on the rhs does not match the unit on the lhs. To show this they define another decaying average as below but this is based on squared parameter update, not the squared gradient:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2, \tag{2.50}$$

and hence the root-mean-square of the parameter update gives the following:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \varepsilon}. \tag{2.51}$$

But since $RMS[\Delta\theta]_t$ is not known, one can approximate its value with $RMS[\Delta\theta]_{t-1}$ which is the RMS of parameter updates until the previous step. Finally, the Adadelta update rule is derived by replacing the learning rate η with $RMS[\Delta\theta]_{t-1}$:

$$\begin{aligned} \Delta\theta_t &= -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{2.52}$$

As one can see, one do not even need to set a default learning rate here since it is not present in the equation above.

ADAM Algorithm

ADAM [14] stands for Adaptive Moment Estimation and is similar to Adadelta where it computes adaptive learning rate for each parameter. Adam keeps the exponentially decaying average of past gradients m_t (similar to momentum) in addition to keeping the exponentially decaying average of past squared gradients v_t as it was the case for Adadelta. One can think of momentum as a ball rolling down a slope and Adam as a heavy ball with friction which prefers flat minima. Decaying average of past gradients and past squared gradients (m_t and v_t respectively) are computed as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2. \end{aligned} \tag{2.53}$$

One can think of m_t and v_t as estimates of first moment (mean) and the second moment (uncentered variance) of the gradients respectively. Authors who proposed the Adam algorithm noticed that m_t and v_t are biased towards zero after the initialization, so in order to counteract this effect they compute the bias-corrected first and second moments as follows:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned}. \tag{2.54}$$

So the Adam update rule becomes:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\varepsilon}\hat{m}_t. \tag{2.55}$$

The default values of 0.9 for β_1, 0.999 for β_2 and 10^{-8} for ε is proposed. They show Adam performs better compared to other adaptive learning methods.

2.3.2 Hyper-parameter Tuning

Tuning the hyper-parameters of a deep neural net plays a major role in the optimization and achieving state-of-the-art results. The two well-known approaches to tackle this are Grid Search and Random Search. Both of them are very easy to parallelize since training the model on different hyper-parameters is independent from each other.

Grid Search Algorithm

In gird search, a range of values is defined for hyper-parameters that we are trying to estimate. Then, an exhaustive search is applied to those hyper-parameters to find the ones where model could perform the best on cross-validation set.

Random Search Algorithm

In random search, instead of exhaustively search through all the combinations of hyper-parameters, we select them randomly. Bengio et al. [3] has shown that random search can outperform grid search when only a small number of hyper-parameters are affecting the performance of the deep net.

2.4 U-Net for Medical Image Segmentation

The U-Net paper [22] published in 2015 was a breakthough in deep learning for medical image segmentation. The idea of U-Net is in fact not new as it has been previously discussed in [17]. As of writing this thesis, the U-Net paper has been cited over 2000 times, which makes it the best baseline for most of the medical image segmentation tasks.

Since U-Net is the basic architecture of this thesis, let us discuss this architecture in details here and understand its building blocks. In the next chapter, we will explain our version of U-Net which is modified compared to what is explained here. The difference is that the convolutions that we use for U-Net adds padding to the input in a way that after applying the convolution the image keeps its dimension, but the architecture explained in the original U-Net does not account for the padding hence after applying the 3×3 convolution to the input, size of the output image is 2 pixels short in width and height (as can be seen in Figure 2.9).

2.4.1 U-Net Architecture

Figure 2.9 illustrates the U-Net architecture. As mentioned in [22], there are two main advantages of U-Net compared with other methods: First, U-Net learns segmentation in an end-to-end setting which means input image is given in one end and the output segmentation is produced in the other end. Second, U-Net needs very few annotated images (approximately 30 per application).

The size of the convolution kernels in U-Net is 3×3 and each convolution is followed by a ReLU activation function. Size of the Max-Pool layer is 2×2 with a stride of 2, so the resulting feature map has factor 2 lower spatial resolution.

We can see the U-Net as being divided into two separate paths, an encoder path and a decoder path as shown in Figure 2.10. The encoder path reduces the spatial resolution of each feature map but increases the number of feature maps in each layer. In contrast the decoder path increases the spatial resolution for each feature map but decreases the number of feature maps in each layer in total.

The expansion path in the U-Net consists of convolution, transpose convolution and concatenate layers.

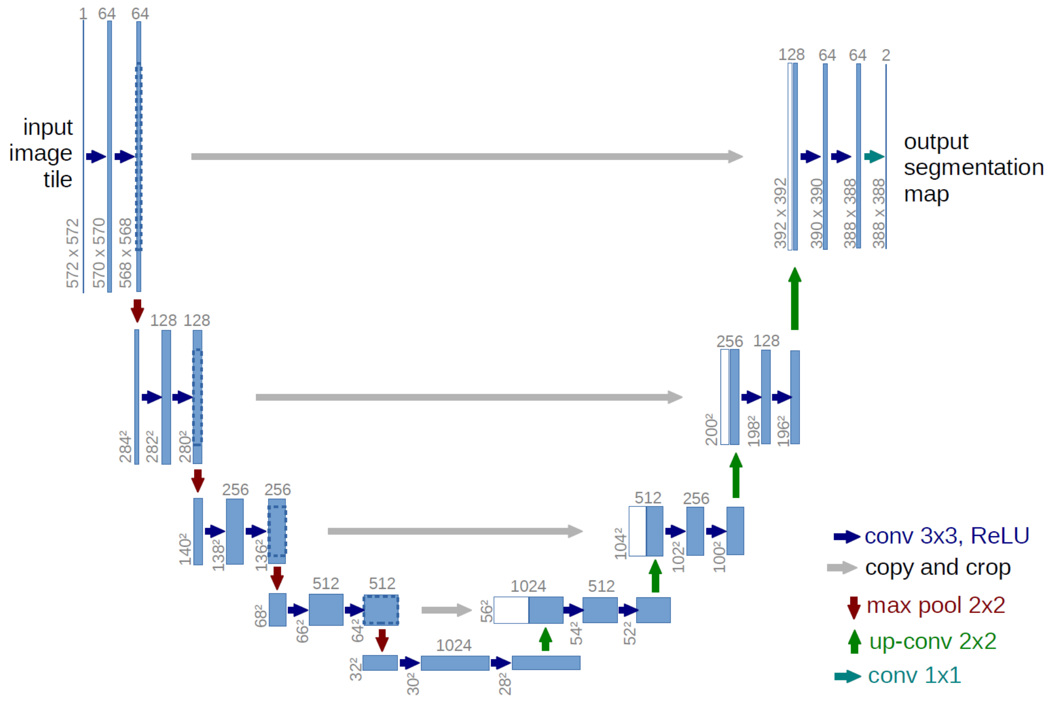


Figure 2.9: U-Net Architecture

The convolution layer in the expansion path acts very similar to the convolution in the encoder/contraction path. It is a 3×3 convolution with the SAME padding which means it keeps the dimensionality after applying the convolution.

The transpose convolution layer is very similar to a convolution operation. Some people call this de-convolution but the term transpose convolution fits better since the operation being done is a normal convolution with a very slight change in the input. For example, if we want to go from a tensor that has the shape of the output of some convolution to a tensor that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution, we use transpose convolution.

Finally, the concatenate layer just combines the corresponding layer from the encoder path to the decoder path to provide the information from the earlier layers to the later layers and matches the dimensionality for the next layer.

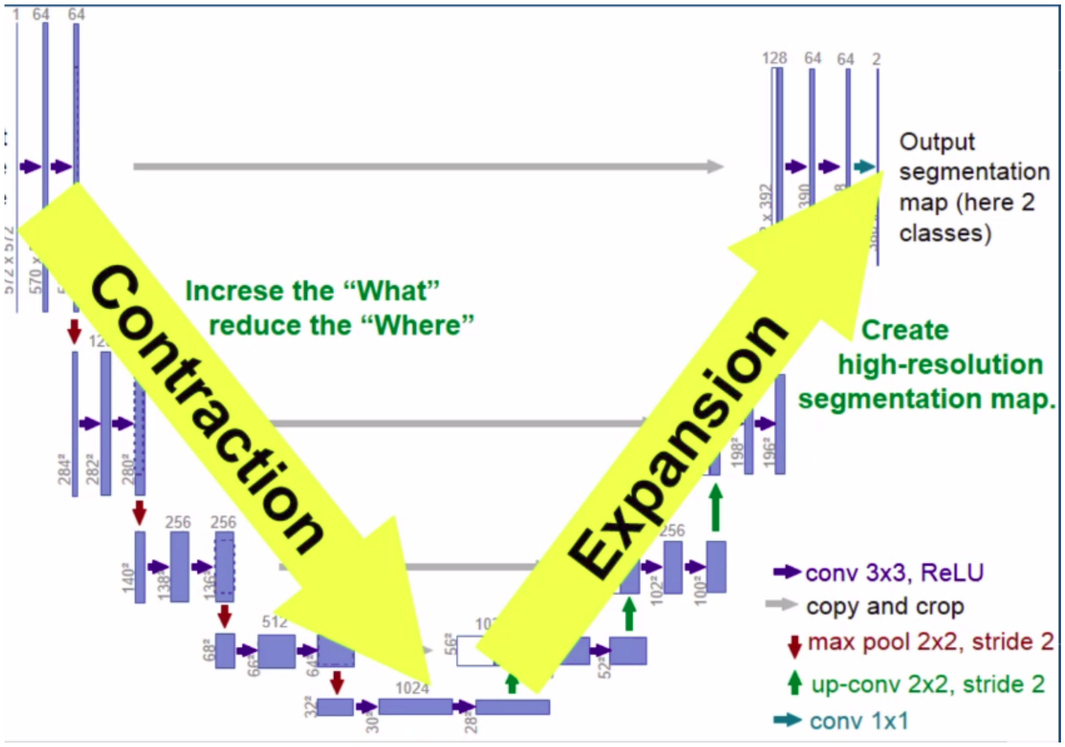

Contraction
Increse the "What" reduce the "Where"
Expansion
Output segmentation map (here 2 classes)
Create high-resolution segmentation map.
conv 3x3, ReLU
copy and crop
max pool 2x2, stride 2
up-conv 2x2, stride 2
conv 1x1
64
128
256
512
1024
2

Figure 2.10: U-Net Encoder Decoder path

2.4.2 U-Net Applications

In the original U-Net paper, the application for the network consisted at segmenting neuronal structures in electron microscopic stacks, cell segmentation task in light microscopic images, and segmentation of HeLa cells on a flat glass recorded by differential interference contrast (DIC) microscopy.

Note that the architecture of the U-Net has been modified and extended to work with fewer training images and to yield more precise segmentations. There has been a lot of research studies since the introduction of U-Net and different papers have applied U-Net to different medical image segmentation tasks.

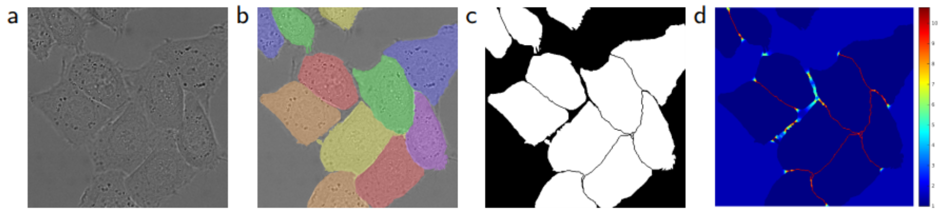


Figure 2.11: Segmentation example in U-Net paper

Chapter 3

Modified U-Net Architecture

In this section, we will discuss modifications that were applied to U-Net in order to show that U-Net does not need to be as deep or even does not need to learn all the convolutional filters proposed in the original paper.

3.1 Shallow U-Net

By analyzing the gradient flow in the bottom layers of the U-Net, one can see that the gradients are very close to zero. This leads us to the idea that maybe U-Net does not need to be as deep as it is architectured in the original paper.

We start by removing the layers from the bottom and work our way up to both ends of the network, and we can show that after removing all the unnecessary layers one can reduce the number of parameters of the network from 31 million to about 0.5 million which also gives us a boost in training and inference time.

3.1.1 Shallow U-Net Architecture

Figure 3.2 depicts the architecture of the shallow U-Net which was used in our experiments. The original U-Net architecture is shown in Figure 3.1 for comparison.

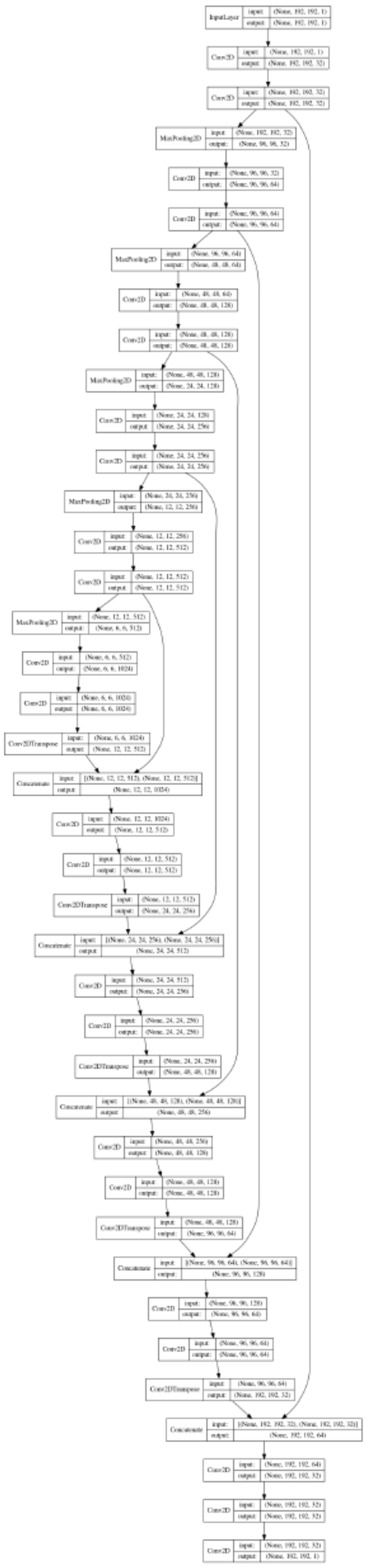


Figure 3.1: Original U-Net architecture

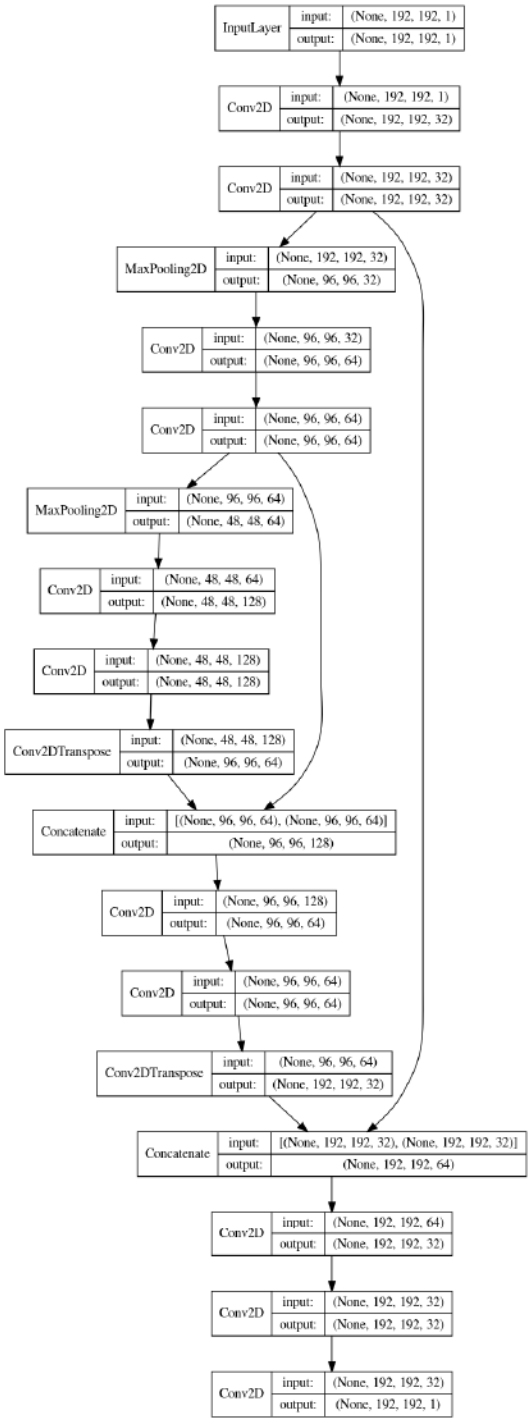


Figure 3.2: Shallow U-Net architecture

3.2 Uniform Filter U-Net

Further, experiment on early layers of the U-Net can provide evidence that convolution filters in early layers are learning weight matrices that resemble a uniform filter. In other words, replacing those convolution kernels with uniform filters should not affect the network performance.

3.2.1 Uniform Filter U-Net Architecture

The architecture of the uniform filter U-Net is very similar to the original U-Net because there are no layers that are removed here. The only difference here is that all the convolution kernel weights in the first four convolution layers are replaced with a uniform filter of the same size and with the value of $1/9$ for each cell in the filter.

3.3 Understanding U-Net Convolution Layers Using Fourier Analysis

In order to understand the Fourier transform of an image, let us explain the Fourier transform of a 1-dimensional (1D) signal first. Then, we will explain the Fourier of a 2D signal. Finally, we will apply those ideas to images and see what a Fourier transform of a uniform filter looks like. [5]

3.3.1 Continuous Fourier Transform

The Fourier transform of $x(t)$ is defined as:

$$X(\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt, \tag{3.1}$$

where t is the time variable in seconds across the time domain, and ω is the frequency variable in radian per second across frequency domain.

[5]The materials in this section has been mostly adopted from `https://github.com/jeremyfix/FFTConvolution/tree/master/Convolution/Doc`

Applying the similar transform to $X(\omega)$ yields the inverse Fourier transform:

$$x(t) = \int_{-\infty}^{\infty} X(\omega) e^{j2\pi\omega t} d\omega, \tag{3.2}$$

where we write $x(t)$ as a weighted sum of complex exponentials.

The Fourier transform pair above can be denoted as:

$$x(t) \overset{\mathcal{F}}{\leftrightarrow} X(\omega), \tag{3.3}$$

where the left hand side of the symbol $\overset{\mathcal{F}}{\leftrightarrow}$ is before Fourier transform, while the right hand side of the symbol $\overset{\mathcal{F}}{\leftrightarrow}$ is after Fourier transform.

3.3.2 Discrete Fourier Transform

One can define the discrete Fourier transform of signal $f[n]$ which has length N as follows:

$$FT[f][k] = \hat{f}[k] = \sum_{n=0}^{N-1} f[n] e^{-\frac{2i\pi kn}{N}}. \tag{3.4}$$

The inverse discrete Fourier transform of a discrete frequency signal $\hat{f}[k]$ is defined as :

$$IFT[\hat{f}][n] = f[n] = \frac{1}{N} \sum_{k=0}^{N-1} \hat{f}[k] e^{\frac{2i\pi kn}{N}}.$$

In terms of complexity, one needs an order of N^2 operations to compute the DFT of a signal of length N (N products for each of the N components). In the following section, we show how one can compute a n-dimensional DFT from only 1D DFT.

Computing a 2D Fourier transform from 1D Fourier transforms

A 2D Fourier transform can be computed from 1D Fourier transforms. For sake of simplicity, let us write $w_P = e^{\frac{-2i\pi}{P}}$. The 2D Fourier transform of a signal $s[n,m]$ of size (N,M) is defined as :

$$
\begin{aligned}
S[k,j] &= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} s[n,m] e^{\frac{-2i\pi jm}{M}} e^{\frac{-2i\pi kn}{N}} && (3.5)\\
&= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} s[n,m] w_M^{jm} w_N^{kn} && (3.6)\\
&= \sum_{n=0}^{N-1}\Big(\sum_{m=0}^{M-1} s[n,m] w_M^{jm}\Big) w_N^{kn} && (3.7)
\end{aligned}
$$

Because of orthogonality between axis a 2D, DFT can be computed by performing a 1D DFT on the rows of s (the inner sum is on n fixed) followed by a 1D DFT on the columns of the result. Let us denote $\hat{S}[n,j]$ the 2D signal for which the row n holds the 1D DFT of the line n of the original signal $s[n,m]$:

$$\hat{S}[n,j] = \sum_{m=0}^{M-1} s[n,m] w_M^{jm}. \tag{3.8}$$

One can write :

$$S[k,j] = \sum_{n=0}^{N-1} \hat{S}[n,j] w_N^{kn}. \tag{3.9}$$

This equation corresponds to a 1D DFT performed on the columns of $\hat{S}$. Therefore, to compute a 2D DFT of s you need to :

1. Perform a 1D DFT on the rows of s which leads to the 2D signal $\hat{S}$

2. Perform a 1D DFT on the columns of $\hat{S}$

3.3.3 Fourier Transform of a Uniform Filter

In order to recognize the patterns that are created in the Fourier transform of the early layers of the U-Net, let us provide the Fourier transform of a few signals and images (2D signals) here.

Let us look at the signal for the Rect function (Figure 3.3), its definition (Figure 3.5) and its Fourier transform which is calculated in Figure 3.5 [6].

[6]From `https://en.wikipedia.org/wiki/Rectangular_function`

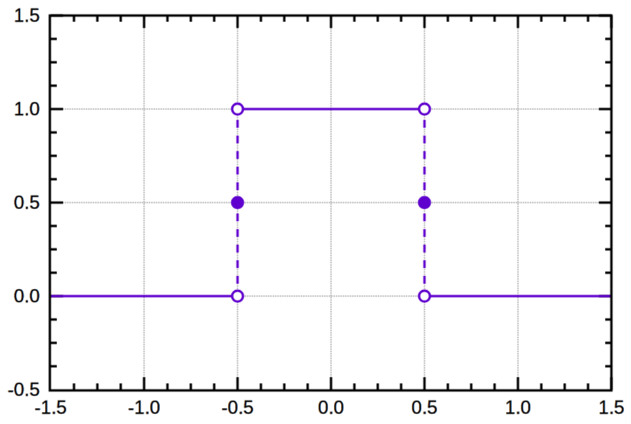


Figure 3.3: Illustration of the Rect function

$$\mathrm{rect}(t) = \Pi(t) = \begin{cases} 0 & \text{if } |t| > \frac{1}{2} \\ \frac{1}{2} & \text{if } |t| = \frac{1}{2} \\ 1 & \text{if } |t| < \frac{1}{2}. \end{cases}$$

Figure 3.4: Rect function definition

Now that we know the Fourier of the Rect function is the Sinc function, let us look at the plot for the Sinc function in Figure 3.6 [7].

Now consider the images in Figure 3.7 through Figure 3.9 and their corresponding Fourier transforms in the bottom row. The top image in Figure 3.7 is a 5×5 square which all values are 1. The rest of the image consists of black pixels with value of zero. The Fourier transform of this image is shown in the bottom row. As one can see before, the Fourier of the Rect function is the Sinc function and this is visible by the horizontal and vertical bars in the image.

The bigger the square in the center, the more bars one can see in the corresponding Fourier transform and those bars are more squashed towards the center.

[7]From `https://en.wikipedia.org/wiki/Sinc_function`

$$\int_{-\infty}^{\infty} \mathrm{rect}(t) \cdot e^{-i2\pi f t}\, dt = \frac{\sin(\pi f)}{\pi f} = \mathrm{sinc}(f)$$

Figure 3.5: Fourier transform of the Rect function

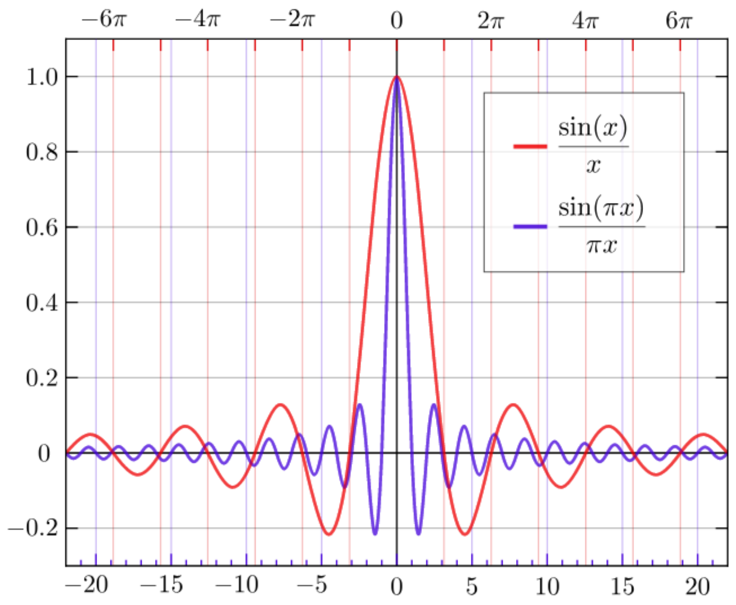


Figure 3.6: Sinc function plot

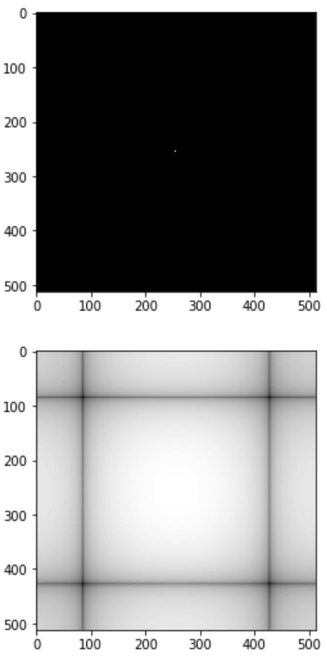


Figure 3.7: 5×5 square

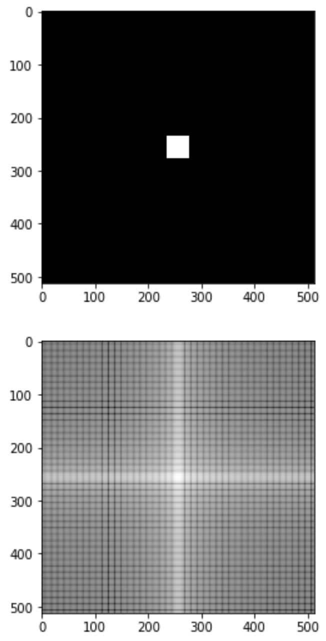


Figure 3.8: 45×45 square

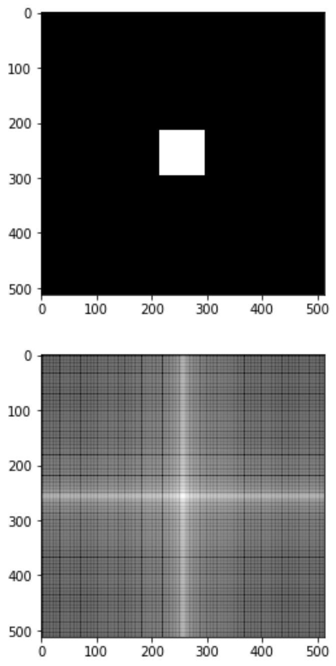


Figure 3.9: 85×85 square

Chapter 4

Experimental Results

In this chapter, we provide the results of our experiments using the U-Net architecture described in the previous chapter. First, let us introduce the data set that we are going to perform our experiments on.

4.1 Data Set

4.1.1 Data access

The ACDC data set[8] was originally collected from real clinical exams. The data is anonymized and includes several well-defined pathologies with enough cases for us to properly train our neural networks. The dataset is composed of 150 exams (all from different patients) divided into 5 evenly distributed subgroups (4 pathological plus 1 healthy subject groups) as described below. Furthermore, each data set comes with the following additional patient information: weight, height, as well as the diastolic and systolic phase instants.

The training dataset consists of 100 patients with their corresponding manual references based on the analysis of one clinical expert. The testing dataset composed of 50 new patients, without manual annotations and hence not used in our experiments (only used for participating in the competition). The raw input images

[8]Information regarding the data access, study population and imaging modality have all been adopted from `https://www.creatis.insa-lyon.fr/Challenge/acdc/databases.html`

are provided through the Nifti format [9].

4.1.2 Study Population

The target population for the study is composed of 150 patients divided into 5 subgroups as follows:

- 30 normal subjects
- 30 patients with previous myocardial infarction
- 30 patients with dilated cardiomyopathy
- 30 patients with hypertrophic cardiomyopathy
- 30 patients with abnormal right ventricle

4.1.3 Imaging Modality

"The acquisitions were obtained over a 6 year period using two MRI scanners of different magnetic strengths (1.5 T (Siemens Area, Siemens Medical Solutions, Germany) and 3.0 T (Siemens Trio Tim, Siemens Medical Solutions, Germany)). Cine MR images were acquired in breath hold with a retrospective or prospective gating and with a SSFP sequence in short axis orientation. Particularly, a series of short axis slices cover the LV from the base to the apex, with a thickness of 5 mm (or sometimes 8 mm) and sometimes an interslice gap of 5 mm (then one image every 5 or 10 mm, according to the examination). The spatial resolution goes from 1.37 to 1.68 mm2/pixel and 28 to 40 images cover completely or partially the cardiac cycle (in the second case, with prospective gating, only 5 to 10% of the end of the cardiac cycle was omitted), all depending on the patient."

4.2 Image Preprocessing

Image preprocessing step consists of cropping the image to size 192×192. The images with number of rows or columns less than 192 will be padded to match 192,

[9]NIfTI stands for Neuroimaging Informatics Technology Initiative. Details can be found here: `https://nifti.nimh.nih.gov/`

such that all images have the same size for the convolutional neural net to work properly.

There is no data augmentation used in our pipeline. Although the data augmentation would likely improve the performance of the network, it is outside the scope of this thesis since our purpose is to show how U-Net can be optimized.

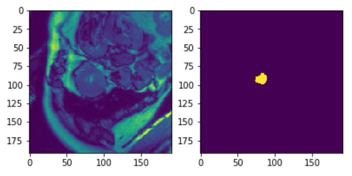


Figure 4.1: Train Example 1

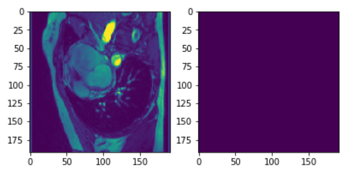


Figure 4.2: Train Example 2

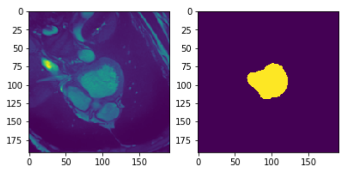


Figure 4.3: Train Example 3

4.3 Distance Metrics

We use Dice score and Hausdorff distance as our evaluation metrics. Pixel classification accuracy is not a very useful measure in our task since most of the pixels in the input image are background pixels. In other words, if we have an algorithm that classifies all pixels as background, we can achieve up to 95% accuracy for some of the images. The reason being the left ventricle consisting a very small region of the whole input image (in this specific case less than 5%).

4.3.1 Dice Score Segmentation Metric

Dice score tries to make up for this issue by computing the intersection of the regions for prediction and the gold standard and divide that by the summation of those regions as calculated in the formula below:

$$DSC = \frac{2|X \cap Y|}{|X| + |Y|}.$$

Figure 4.4: Dice Score Formula

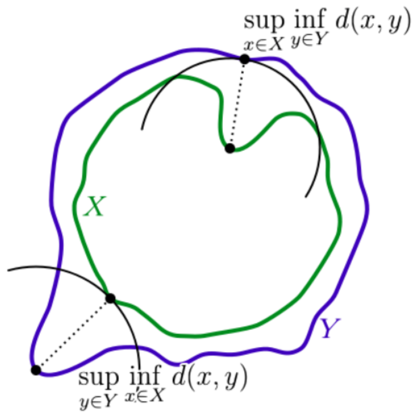
4.3.2 Hausdorff Distance Metric

Hausdorff distance measures how far two subsets of a metric spaced from each other and is defined as:

$$d_{\mathrm{H}}(X,Y) = \max\{\sup_{x\in X}\inf_{y\in Y} d(x,y),\ \sup_{y\in Y}\inf_{x\in X} d(x,y)\}.$$

Figure 4.5: Hausdorff Distance

A visual explanation on how Hausdorff distance is provided in Figure 4.6[10].



$\sup_{x\in X}\inf_{y\in Y} d(x,y)$
X
Y
$\sup_{y\in Y}\inf_{x\in X} d(x,y)$

Figure 4.6: Explanation of Hausdorff Distance

4.4 Original U-Net Architecture Configuration

The architecture of the U-Net used in our implementation was briefly described in Figure 3.1 previously. However in this section, we provide more details of this architecture layer by layer explaining the purpose of each layer and the number of parameters involved between layers. All the convolution operations are 3×3 except the very last convolution which is 1×1. Table 4.1 explains this architecture in details.

[10]From `https://en.wikipedia.org/wiki/Hausdorff_distance`

Table 4.1: Detailed Architecture of U-Net

Layer	Output Shape	Param #	Connected to
Input	(192, 192, 1)	0	
Conv2D_1	(192, 192, 32)	320	Input
Conv2D_2	(192, 192, 32)	9248	Conv2D_1
MaxPool_1	(96, 96, 32)	0	Conv2D_2
Conv2D_3	(96, 96, 64)	18496	MaxPool_1
Conv2D_4	(96, 96, 64)	36928	Conv2D_3
MaxPool_2	(48, 48, 64)	0	Conv2D_4
Conv2D_5	(48, 48, 128)	73856	MaxPool_2
Conv2D_6	(48, 48, 128)	147584	Conv2D_5
MaxPool_3	(24, 24, 128)	0	Conv2D_6
Conv2D_7	(24, 24, 256)	295168	MaxPool_3
Conv2D_8	(24, 24, 256)	590080	Conv2D_7
MaxPool_4	(12, 12, 256)	0	Conv2D_8
Conv2D_9	(12, 12, 512)	1180160	MaxPool_4
Conv2D_10	(12, 12, 512)	2359808	Conv2D_9
MaxPool_5	(6, 6, 512)	0	Conv2D_10
Conv2D_11	(6, 6, 1024)	4719616	MaxPool_5
Conv2D_12	(6, 6, 1024)	9438208	Conv2D_11
TransposeConv2D_1	(12, 12, 512)	2097664	Conv2D_12
Concatenate_1	(12, 12, 1024)	0	TransposeConv2D_1 | Conv2D_10
Conv2D_13	(12, 12, 512)	4719104	Concatenate_1
Conv2D_14	(12, 12, 512)	2359808	Conv2D_13
TransposeConv2D_2	(24, 24, 256)	524544	Conv2D_14
Concatenate_2	(24, 24, 512)	0	TransposeConv2D_2 | Conv2D_8
Conv2D_15	(24, 24, 256)	1179904	Concatenate_2
Conv2D_16	(24, 24, 256)	590080	Conv2D_15
TransposeConv2D_3	(48, 48, 128)	131200	Conv2D_16
Concatenate_3	(48, 48, 256)	0	TransposeConv2D_3 | Conv2D_6
Conv2D_17	(48, 48, 128)	295040	Concatenate_3
Conv2D_18	(48, 48, 128)	147584	Conv2D_17
TransposeConv2D_4	(96, 96, 64)	32832	Conv2D_18
Concatenate_4	(96, 96, 128)	0	TransposeConv2D_4 | Conv2D_4
Conv2D_19	(96, 96, 64)	73792	Concatenate_4
Conv2D_20	(96, 96, 64)	36928	Conv2D_19
TransposeConv2D_5	(192, 192, 32)	8224	Conv2D_20
Concatenate_5	(192, 192, 64)	0	TransposeConv2D_5 | Conv2D_2
Conv2D_21	(192, 192, 32)	18464	Concatenate_5
Conv2D_22	(192, 192, 32)	9248	Conv2D_21
Conv2D_23	(192, 192, 1)	33	Conv2D_22

Table 4.2: Comparison of U-Net variations

	Original	**Uniform Filter**	**Shallow**
Dice Score	0.93	0.92	0.93
Hausdorff Distance	2.02	2.30	2.33
Trainable Params	31,093,921	31,028,929	465,953
Non-trainable params	0	64,992	0
Training time (ms/step)	20		10
Prediction time (ms/step)	7		4

4.5 Results

Table 4.2 summarizes the experiments conducted. Analysis and visualizations are presented in the following sections.

4.5.1 Training Progression

In Figure 4.7, the training progression is shown for the original U-Net architecture. After 100 epochs, the Dice score converges to 0.93 on the test set. Another interesting observation here is how fast the loss drops at the beginning of the training. Loss is defined as the negative of the dice score, in other words maximizing the dice score is equal to minimizing the loss.

Figure 4.8 shows the training progression plot for 100 epochs for the shallow U-Net architecture. Dice score converges to 0.93 for the test set at the end of 100 epochs of training which is the same as deep U-Net. Loss function value drops even faster in the case of the shallow U-Net compared to the original deep U-Net, and after a few epochs, it gradually reaches its plateau.

The training progression for uniform U-Net is shown in Figure 4.9. One noticeable difference here is the number of epochs needed in the training stage to achieve a comparable dice score to other variations of U-Net discussed before. There are more jittering and fluctuations during the training phase, but eventually, the Dice score converges to 0.91 on the test set. However, we need to train this network for 200 epochs as opposed to 100 in order to achieve a comparable Dice score.

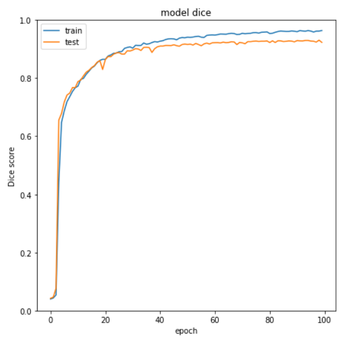


Figure 4.7: Original U-Net

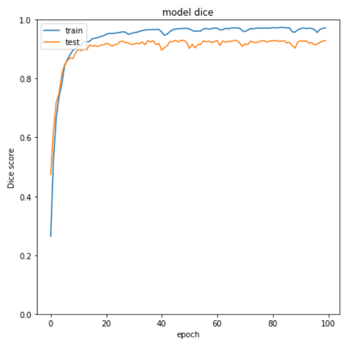


Figure 4.8: Shallow U-Net

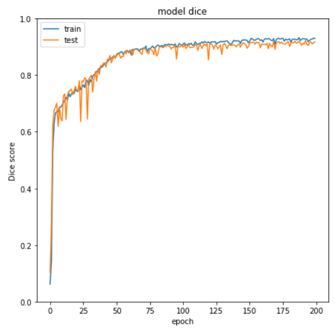


Figure 4.9: Fourier U-Net

4.5.2 Good Segmentation Results

In this section, we provide examples of good segmentation results produced by the original U-Net architecture on the test set. Test set cases have been handpicked here in a way to show the strength of the model in segmenting a variety of input images that not only differ in the shape and area of the left ventricle itself but also different in terms of the brightness and shape of the whole slice of the MRI image.

In Figure 4.10 through Figure 4.13, the image on the left is the input test example, the middle image is the correct gold standard output, and the image on the right is the predicted output produced by the network.

In Figure 4.10, there is no left ventricle to be segmented since the input slice is at the end of the cardiac cycle where we have the most contraction. We can see that the network prediction perfectly captures this. On the other hand in Figure 4.11, the left ventricle consists of a bigger portion of the whole image and also it is not exactly in the center of the image. Figure 4.12 and Figure 4.13 show two very different input images in which both have very small left ventricle regions and close to the center.

4.5.3 Failed Segmentation Results

Failure cases are rarely discussed in the literature. Authors think reporting the failure cases of their models will undermine the value of their paper which is a very wrong assumption. The community should be aware of the strengths and weak-

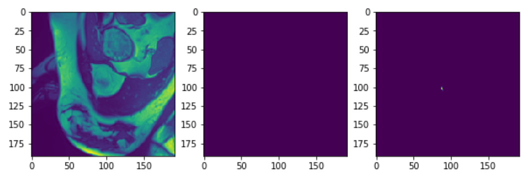


Figure 4.10: Test Example 1

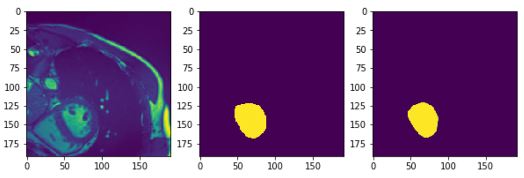


Figure 4.11: Test Example 2

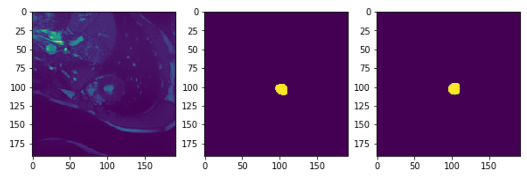


Figure 4.12: Test Example 3

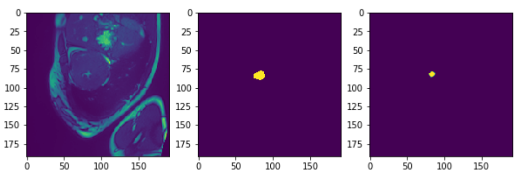


Figure 4.13: Test Example 4

nesses of the proposed approaches. This is why in this section we report the failure cases of our model and investigate how different are the weaknesses for different shallow and deep models.

Figure 4.14 through Figure 4.15 show the failure cases of the U-Net. In each figure, the image on the left is the input, the middle depicts the correct actual segmentation, and the produced segmentation is on the right. Failure cases include a mix of small and big left ventricle regions. Sometimes the neural net segments a region that should not be segmented and sometimes it does not segment a region that should be segmented.

4.5.4 Creating Shallow from Gradient Flow

By plotting the histogram of the gradients of the early layers of the U-Net and also the deeper layers, one can see that the gradient flow at the bottom layers is much less than the early layers. Hence the motivation behind removing the bottom layers from the U-Net in order to make it more shallow.

Figure 4.18 shows the histogram of the gradient values for the first layer in the deep U-Net, and Figure 4.19 shows the same plot for the middle layer (*i.e.*, the bottom layer) in the U-Net. The x-axis shows the actual value of the gradient and the y-axis is the number of elements in the gradient matrix with that value. We can

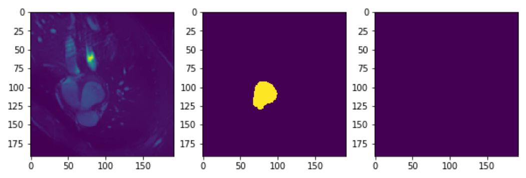


Figure 4.14: Test failure 1

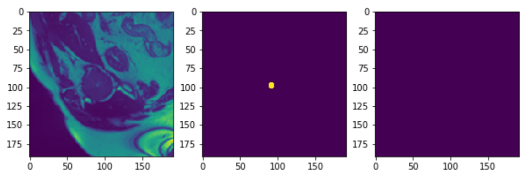


Figure 4.15: Test failure 2

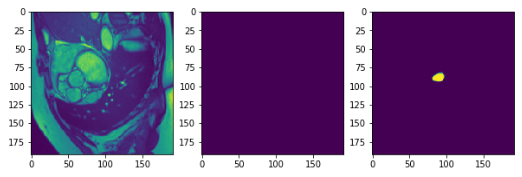


Figure 4.16: Test failure 3

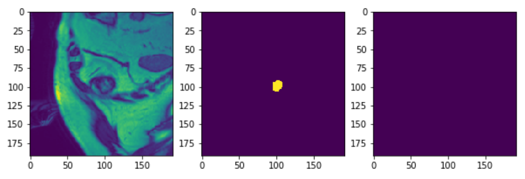


Figure 4.17: Test failure 4

see that the gradient value in the deeper layers is much closer to zero compared the early layers.

One can infer that there is not much learning of the weights in the bottom layers and hence it might be beneficial to remove those layers and reassess the network performance.

Figure 4.20 and Figure 4.21 show the histogram of the gradients for the first layer and the middle layer for the shallow U-Net, respectively.

Comparing Figure 4.19 and Figure 4.21 shows how different are the values of the gradients along the x-axis of those two plots. In the deeper network, the values are close to zero but that is not the case for the shallow network.

4.5.5 Shallow Net Segmentation Results

Both the shallow and deeper versions of the U-Net produce the same Dice score on the test set. Therefore, one would expect that both these network produce the same results. However, the assessment of individual segmentation results in each image prove this to be wrong. They actually produce pretty different segmentation for some of the cases on the test set. Figure 4.22 through Figure 4.28 show different examples of this contrast in the segmentation of the shallow and deep U-Net approaches.

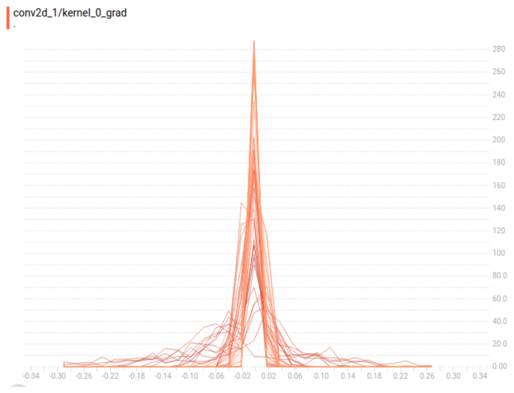


Figure 4.18: Deep First Layer Gradient

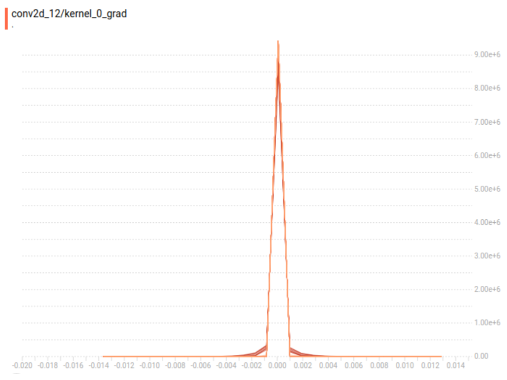


Figure 4.19: Deep Bottom Layer Gradient

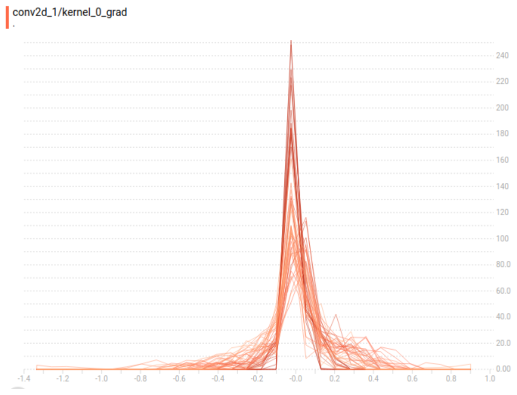


Figure 4.20: Shallow First Layer Gradient

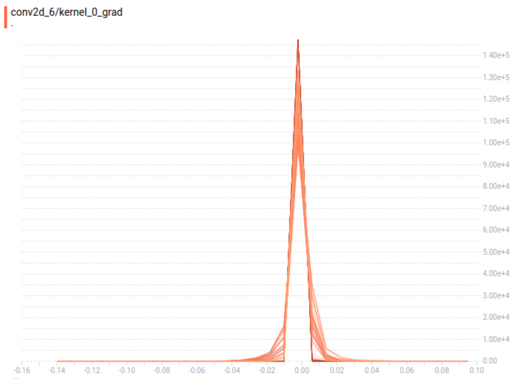


Figure 4.21: Shallow Bottom Layer Gradient

In each figure, the top row is the segmentation of the shallow U-Net and the bottom row is the segmentation result for the deep U-Net. In each row, the first column is the input image, the middle column is the correct segmentation, and the last column is the prediction by the neural net.

4.5.6 Understanding U-Net Functionality Using Fourier Analysis

In this section, we compare the uniform filter U-Net with the original U-Net through the lens of Fourier analysis. In other words, we visualize each convolution layer and its corresponding Fourier transform to see if one can recognize a pattern that could lead us to replacing the convolution kernels with a known filter such as the uniform filter.

Input/Output Test Images

The input and the gold standard output used for visualizations in this section are shown in Figure 4.29 and 4.30, respectively.

Layer 1

One can see the convolution results of the original U-Net filters from layer 1 in Figure 4.31, and its corresponding Fourier transform in Figure 4.33. From observation, it is obvious that the original U-Net layer 1 filters can be approximated by uniform filters. In other words, as the original network get optimized over time, the layer 1 filters converge to a solution very close to the one produced by uniform filters. Following this observation, we replace the original network layer 1 filters with actual uniform filters. One can see in Figure 4.32 results of the convolution and in Figure 4.34 results of applying Fourier Transform to the said convolutions.

Layer 2 through Layer 4

A similar strategy can be applied to layers 2 through 4 following the same reasoning applied to layer 1.

Layer 5 through Layer 10

For these layers, we were unable to replace convolution filter weights with uniform filters. Therefore. starting from layer 5, the optimized U-Net architecture convolution weights are learned using the back-propagation.

Layer 11

This is the bottom of the U-shaped structure in the U-Net architecture. As one can see, these filters cannot be approximated by uniform filters.

Layer 13 through Layer 21

After the bottom layer of the encoder, we start the decoder part of the U-Net.

Layer 22

This is the last convolution layer. After this layer is the output segmentation.

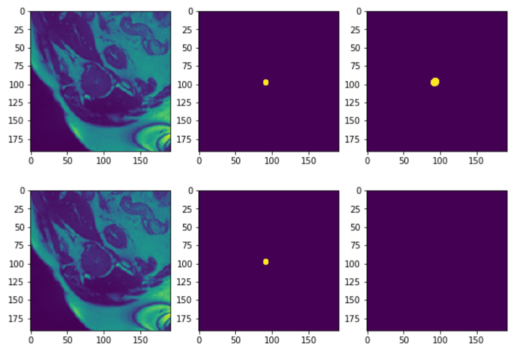


Figure 4.22: Shallow vs Deep example 1

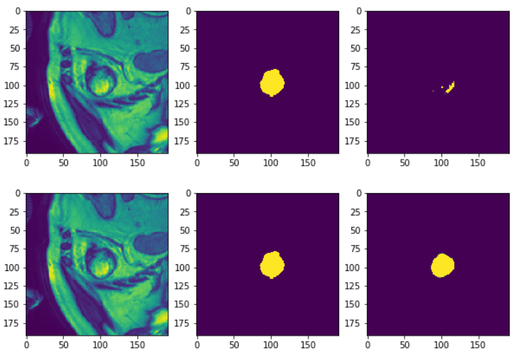


Figure 4.23: Shallow vs Deep example 2

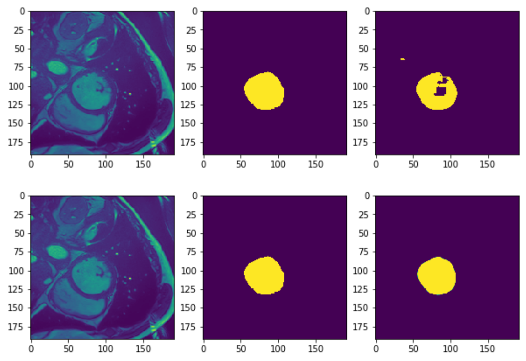


Figure 4.24: Shallow vs Deep example 3

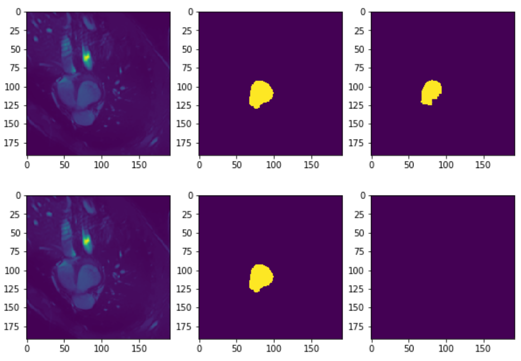


Figure 4.25: Shallow vs Deep example 4

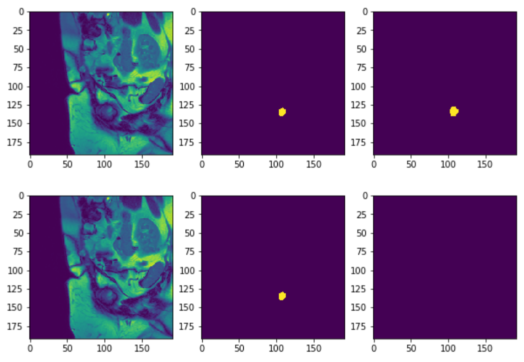


Figure 4.26: Shallow vs Deep example 5

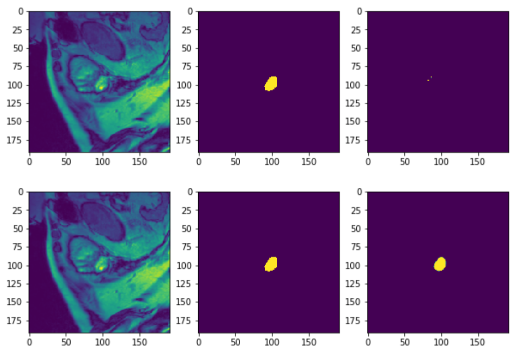


Figure 4.27: Shallow vs Deep example 6

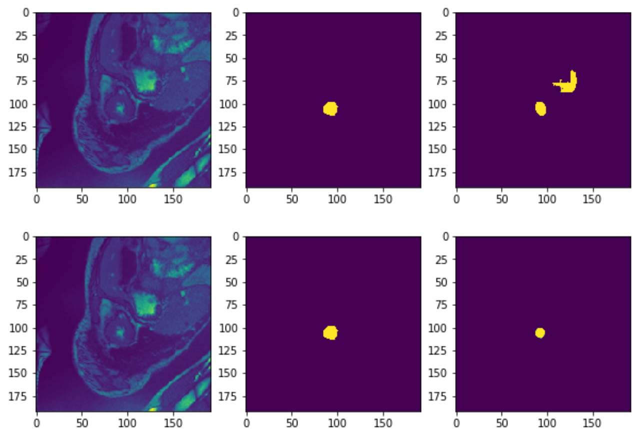


Figure 4.28: Shallow vs Deep U-Net example 7

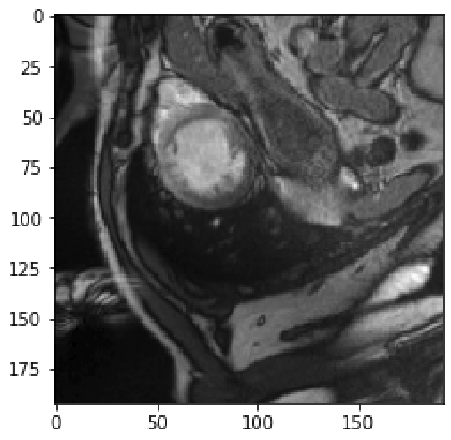


Figure 4.29: Input image

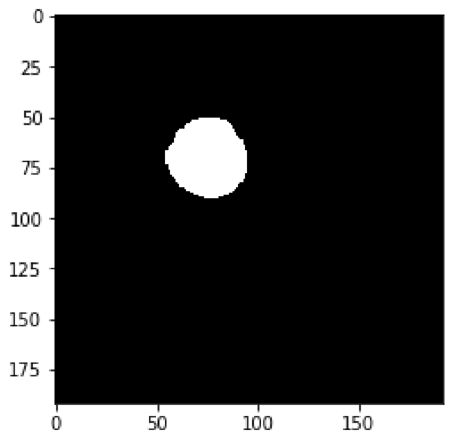


Figure 4.30: Output image

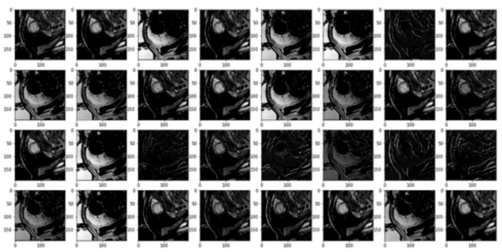


Figure 4.31: L1 Conv

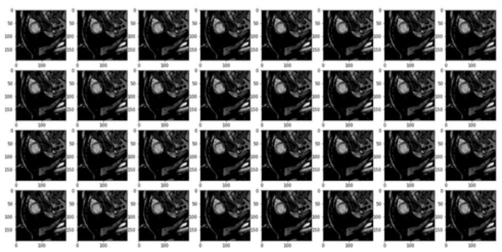


Figure 4.32: L1 Optimized Conv

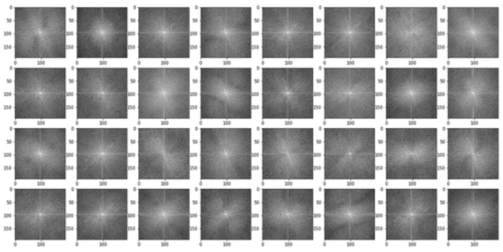


Figure 4.33: L1 Fourier

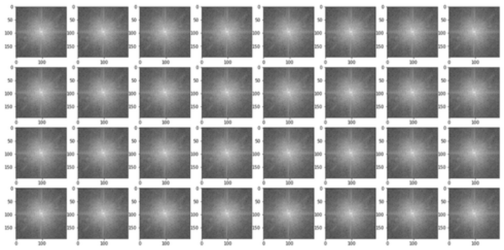


Figure 4.34: L1 Optimized Fourier

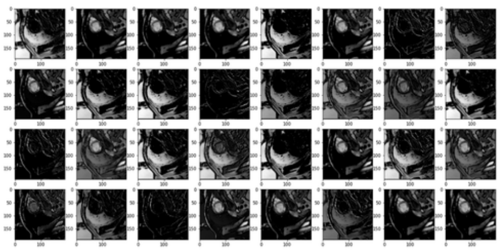


Figure 4.35: L2 Conv

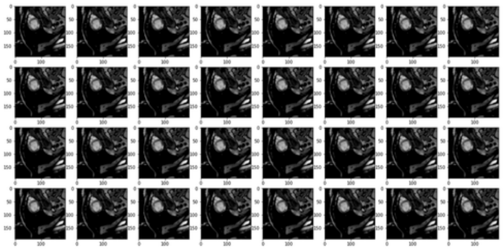


Figure 4.36: L2 Optimized Conv

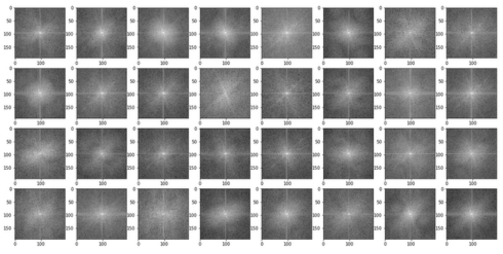


Figure 4.37: L2 Fourier

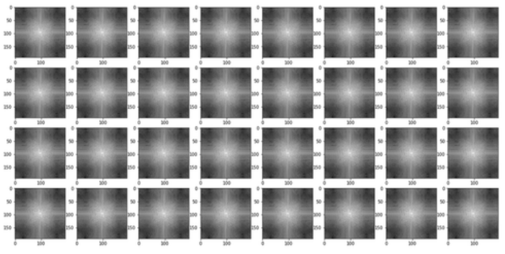


Figure 4.38: L2 Optimized Fourier

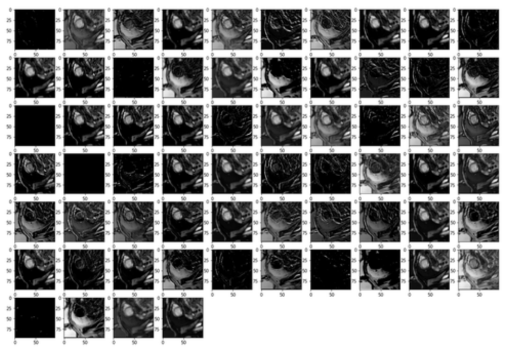


Figure 4.39: L3 Conv

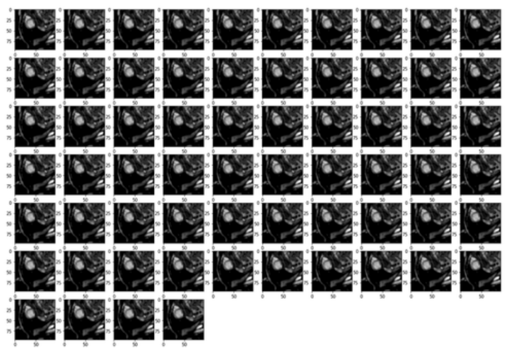


Figure 4.40: L3 Optimized Conv

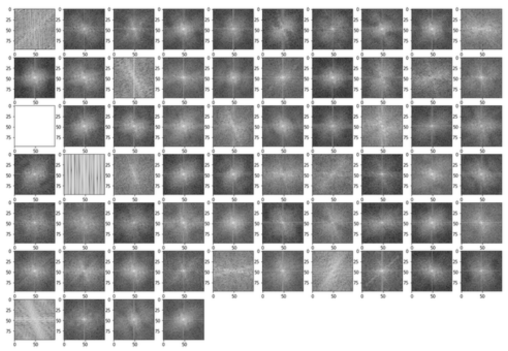


Figure 4.41: L3 Fourier

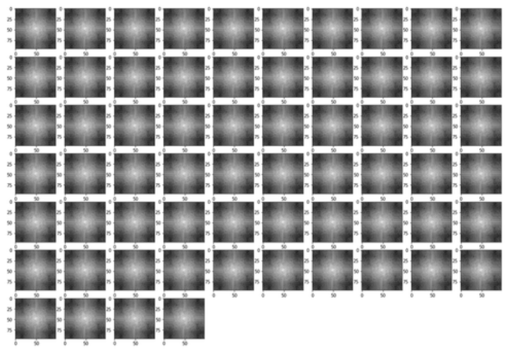


Figure 4.42: L3 Optimized Fourier

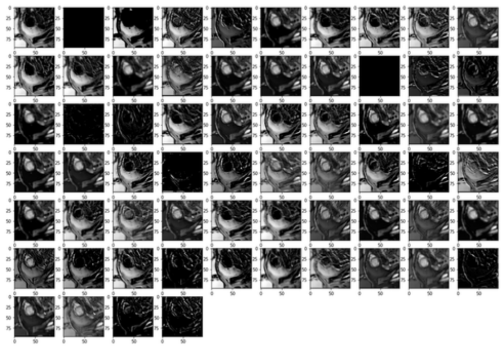


Figure 4.43: L4 Conv

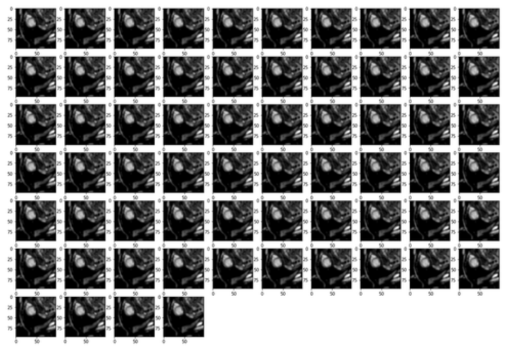


Figure 4.44: L4 Optimized Conv

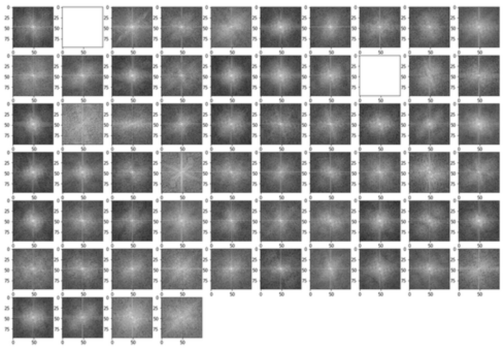


Figure 4.45: L4 Fourier

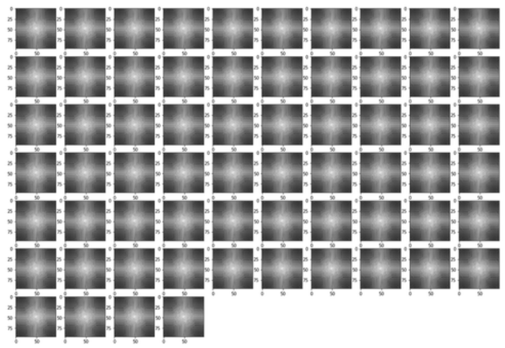


Figure 4.46: L4 Optimized Fourier

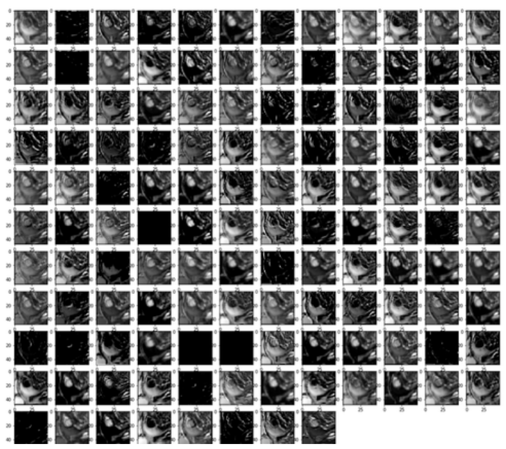


Figure 4.47: L5 Conv

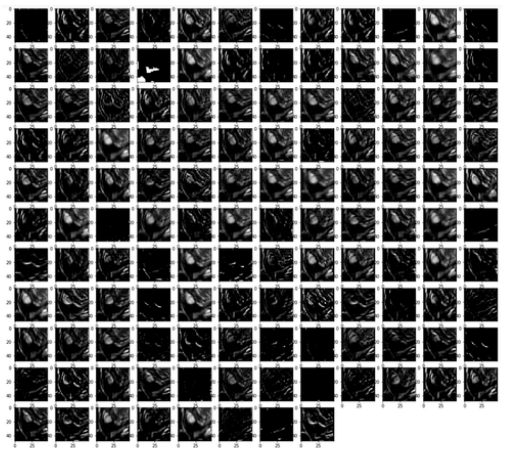


Figure 4.48: L5 Optimized Conv

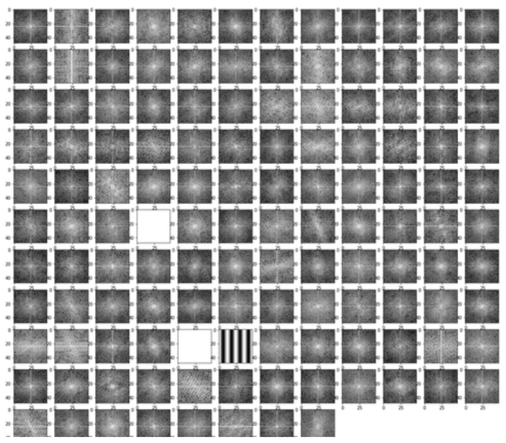


Figure 4.49: L5 Fourier

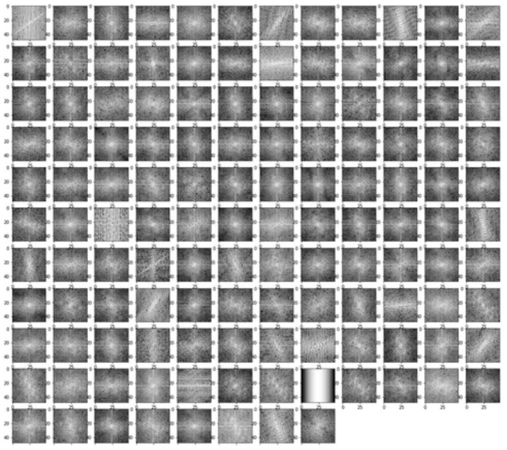


Figure 4.50: L5 Optimized Fourier

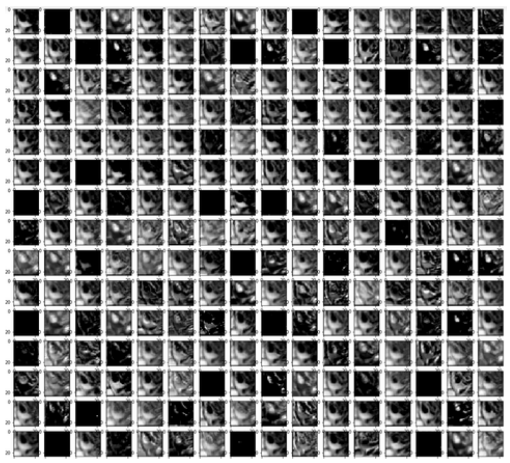


Figure 4.51: L7 Conv

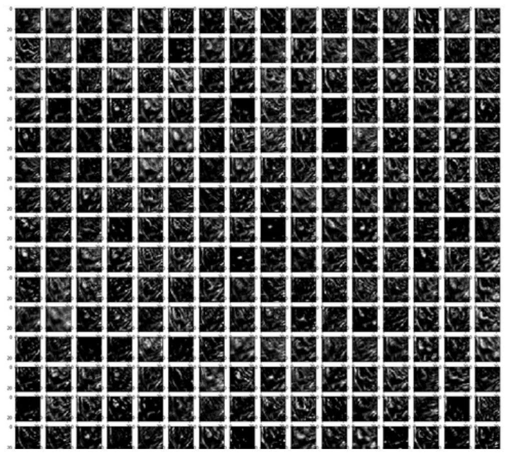


Figure 4.52: L7 Optimized Conv

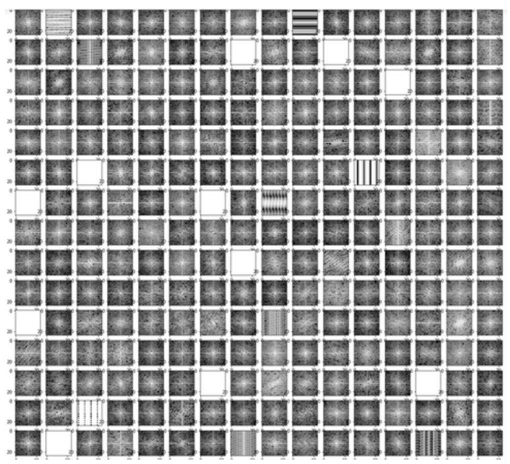


Figure 4.53: L7 Fourier

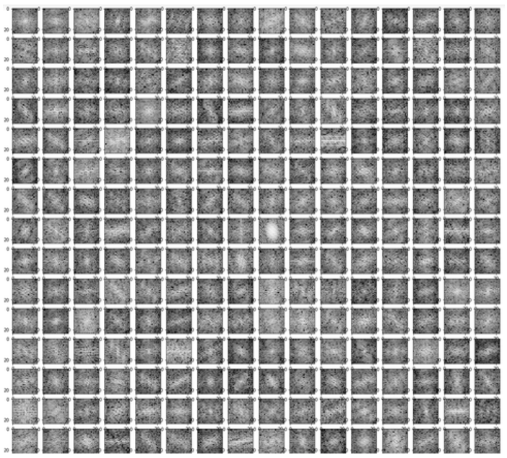


Figure 4.54: L7 Optimized Fourier

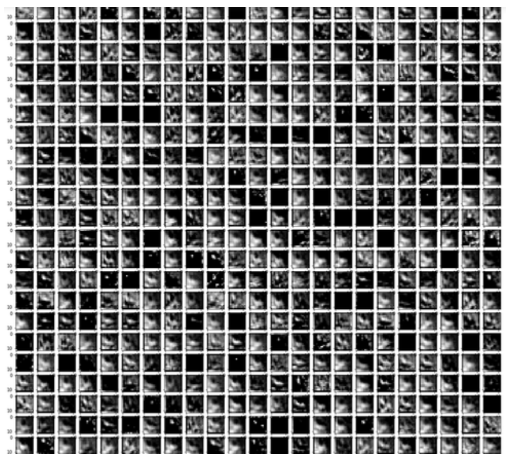


Figure 4.55: L9 Conv

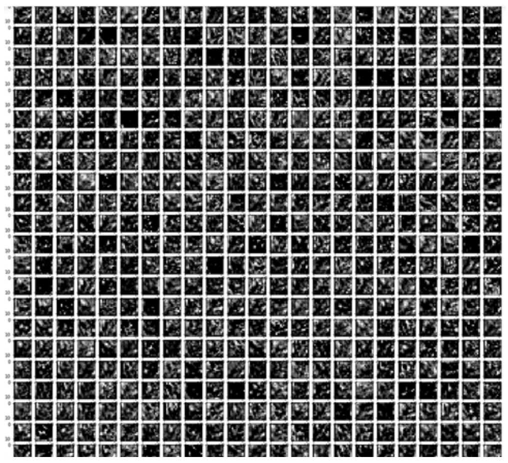


Figure 4.56: L9 Optimized Conv

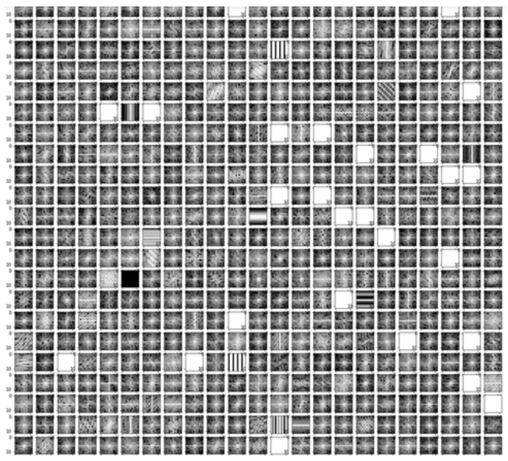


Figure 4.57: L9 Fourier

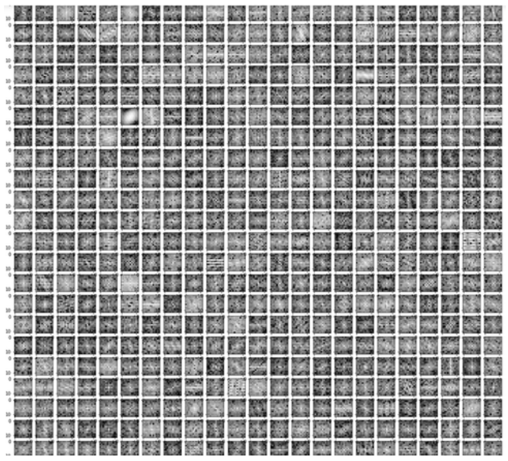


Figure 4.58: L9 Optimized Fourier

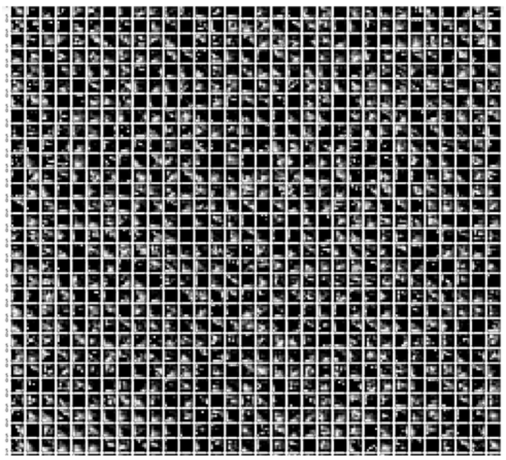


Figure 4.59: L11 Conv

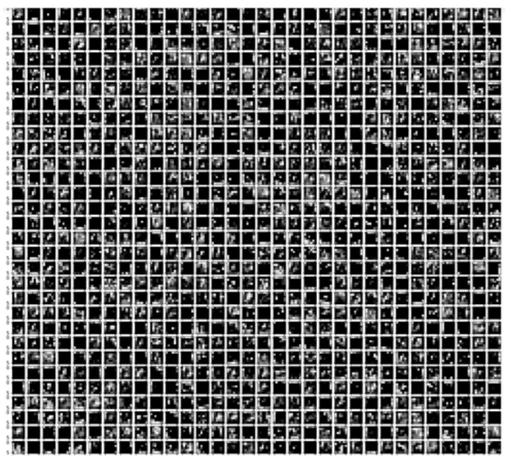


Figure 4.60: L11 Optimized Conv

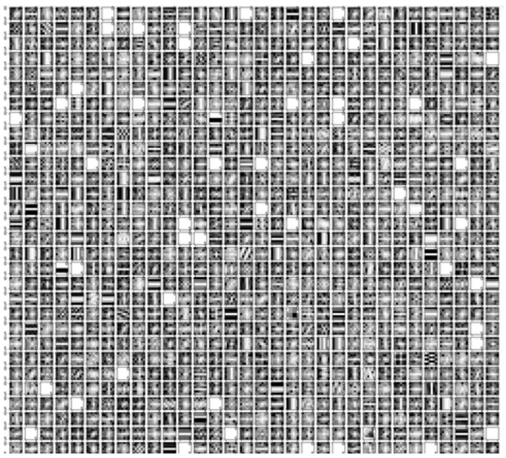


Figure 4.61: L11 Fourier

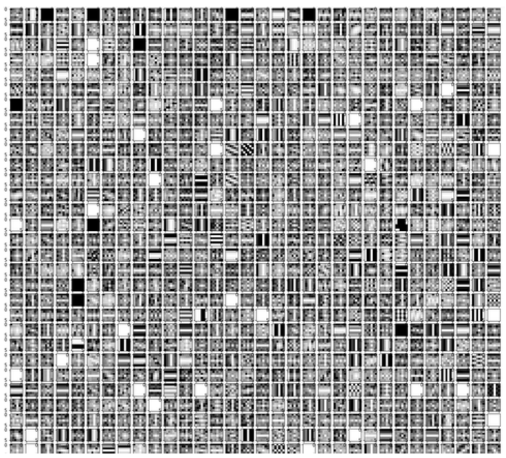


Figure 4.62: L11 Optimized Fourier

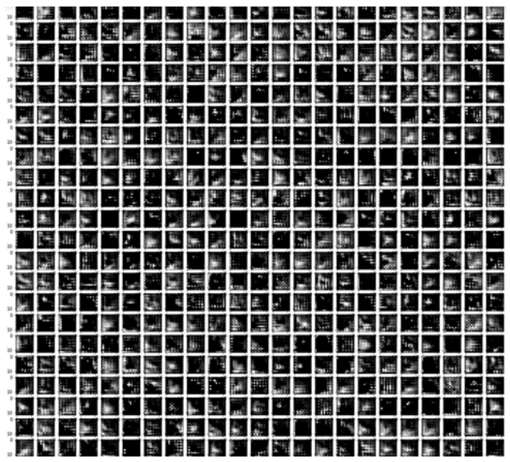


Figure 4.63: L13 Conv

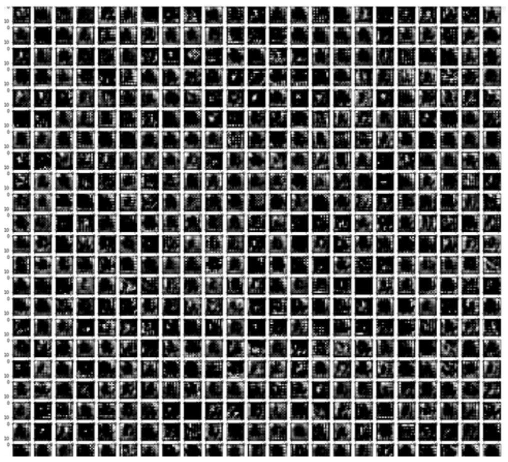


Figure 4.64: L13 Optimized Conv

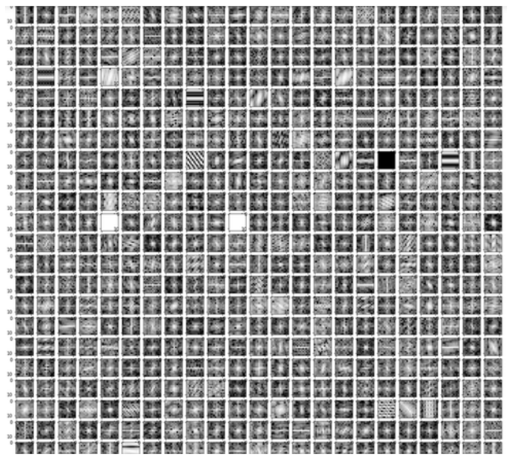


Figure 4.65: L13 Fourier

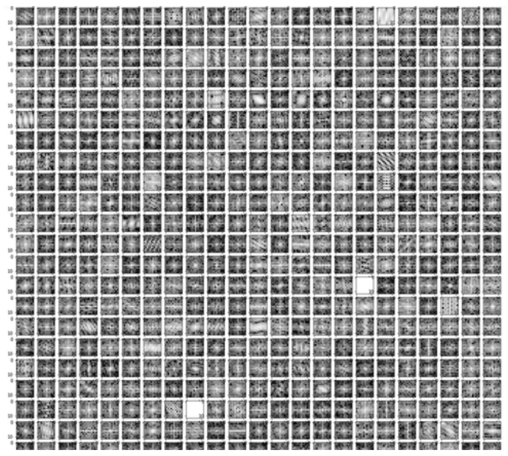


Figure 4.66: L13 Optimized Fourier

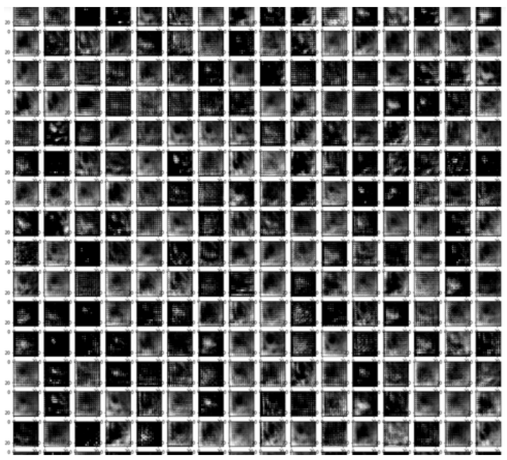


Figure 4.67: L15 Conv

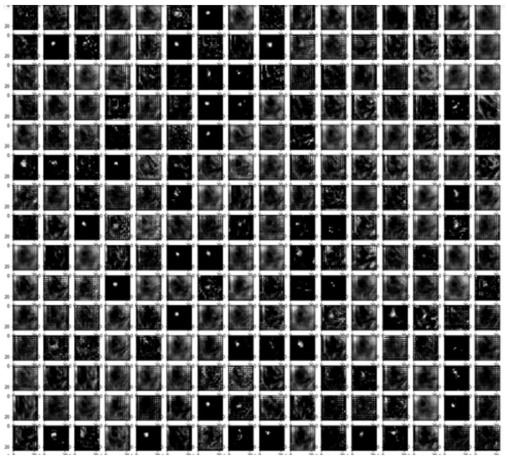


Figure 4.68: L15 Optimized Conv

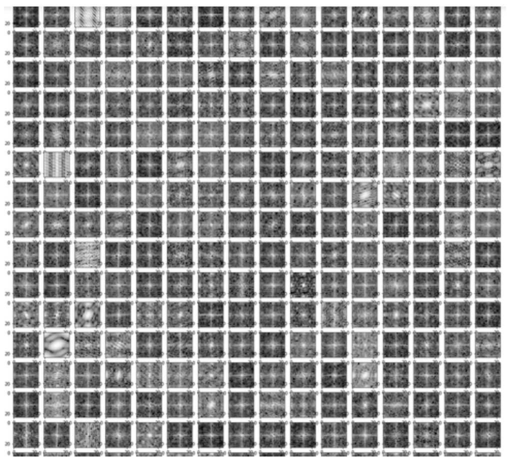


Figure 4.69: L15 Fourier

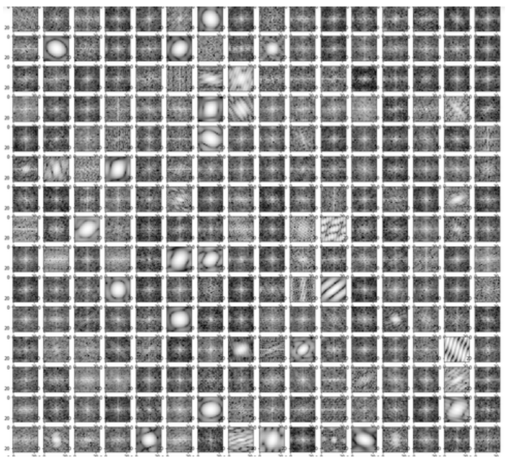


Figure 4.70: L15 Optimized Fourier

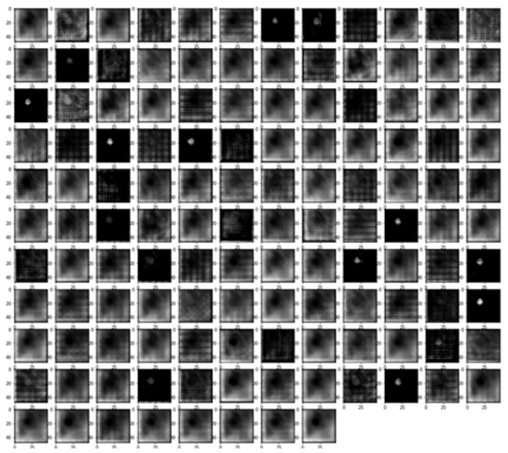


Figure 4.71: L17 Conv

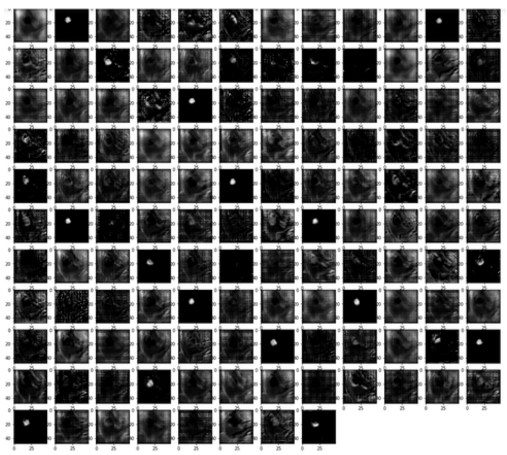


Figure 4.72: L17 Optimized Conv

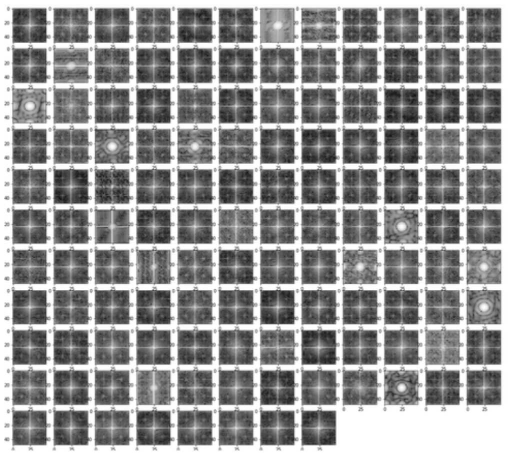


Figure 4.73: L17 Fourier

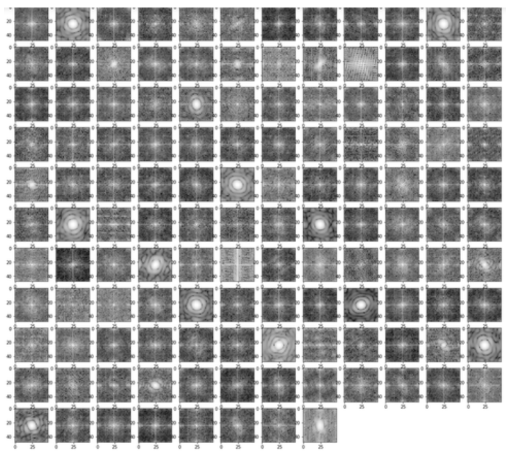


Figure 4.74: L17 Optimized Fourier

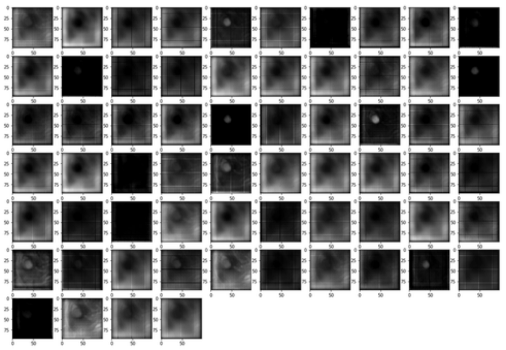


Figure 4.75: L19 Conv

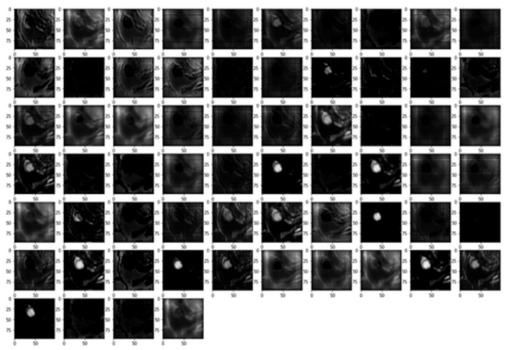


Figure 4.76: L19 Optimized Conv

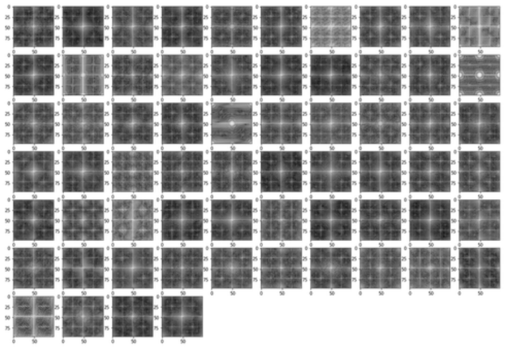


Figure 4.77: L19 Fourier

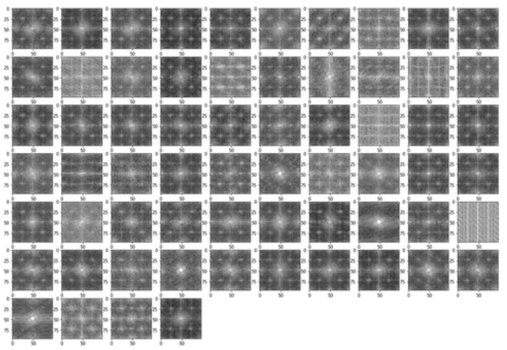


Figure 4.78: L19 Optimized Fourier

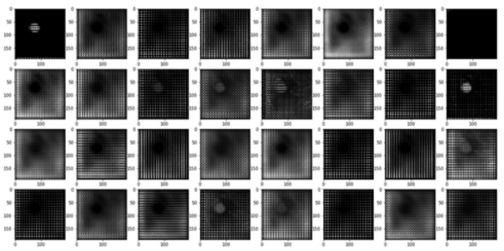


Figure 4.79: L21 Conv

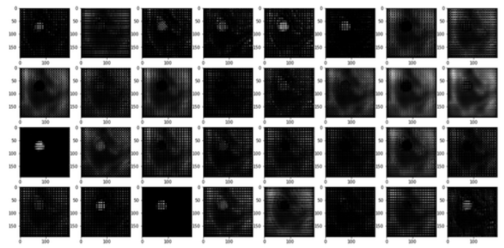


Figure 4.80: L21 Optimized Conv

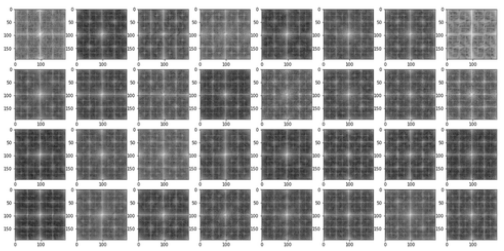


Figure 4.81: L21 Fourier

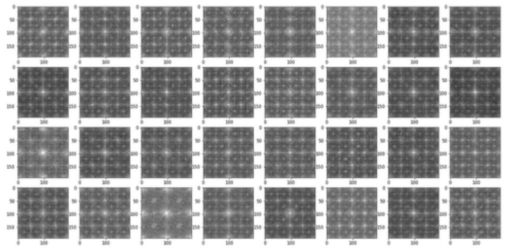


Figure 4.82: L21 Optimized Fourier

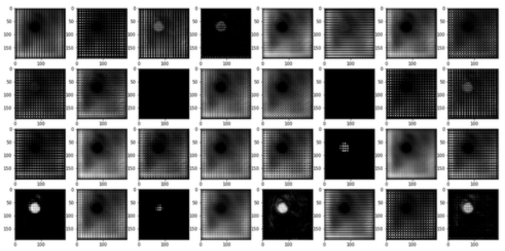


Figure 4.83: L22 Conv

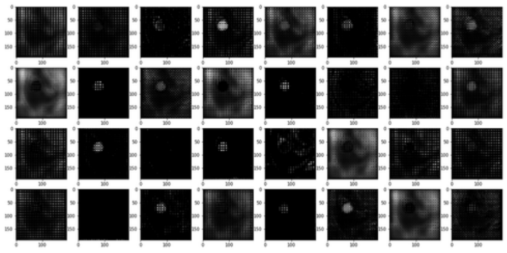


Figure 4.84: L22 Optimized Conv

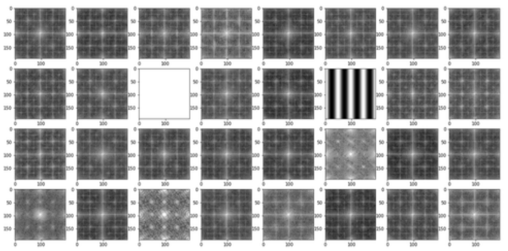


Figure 4.85: L22 Fourier

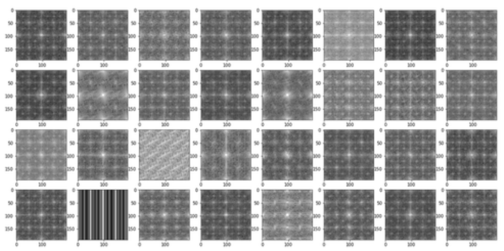


Figure 4.86: L22 Optimized Fourier

Chapter 5

Conclusion and Future Work

The original U-Net architecture works very well for medical image segmentation tasks, especially if the training set is not big enough for standard CNN architectures. However, the architecture does not need to be as complicated as it is explained in the original paper in [22].

In this work, we presented experiments and results indicating the U-Net for the left ventricle segmentation does not need to be as deep as suggested. We showed this by performing a gradient analysis in the deeper layers that shows that the gradient flow is very sparse in those layers. Hence, removing most of the bottom layers in the U-Net can decrease the number of free parameters and increase the training and inference speed significantly.

More analysis on the first layers of the U-Net also shows that most of them are learning uniform filters and thus replacing those convolution kernel weights with fixed uniform filters should not affect the performance of the network by much. Experiments and results are presented to support this hypothesis, and visualization of U-Net layers in Fourier domain provides further evidence regarding this.

5.1 Future Work

There are still lots of work to be done to understand what is exactly happening in the deep convolutional neural networks. This is specifically important for the medical tasks as the need for interpretable and explainable models will continue to rise. One aspect that was shown in this thesis is the fact that although both the shallow and deep U-Nets achieve the same Dice score, their resulting segmentations on the test set show a big difference on how these two models behave. In other words, each model has its own strengths and weaknesses, but they yield similar computational performances on the test set overall. One possible future direction for this thesis is to find these strengths and weaknesses.

Bibliography

[1] MR Avendi, Arash Kheradvar, and Hamid Jafarkhani. A combined deep-learning and deformable-model approach to fully automatic segmentation of the left ventricle in cardiac mri. *Medical image analysis*, 30:108–119, 2016.

[2] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, (1):1–127, 2009.

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[4] C. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[5] C. Bishop. *Pattern Recognition and Machine Learning*. Springer Verlag, New York, 2006.

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[7] D. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, New Jersey, 2002.

[8] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feed-forward neural networks. In *Artificial Intelligence and Statistics, International Conference on*, pages 249–256, 2010.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[11] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *Computing Research Repository*, abs/1207.0580, 2012.

[12] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[13] K. Jarrett, K. Kavukcuogl, M. Ranzato, and Y. LeCun. What is the best multistage architecture for object recognition? In *Computer Vision, International Conference on*, pages 2146–2153, 2009.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[16] Y. LeCun, K. Kavukvuoglu, and C. Farabet. Convolutional networks and applications in vision. In *Circuits and Systems, International Symposium on*, pages 253–256, 2010.

[17] 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.

[18] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[19] Caroline Petitjean and Jean-Nicolas Dacher. A review of segmentation methods in short axis cardiac mr images. *Medical image analysis*, 15(2):169–84, Apr 2011.

[20] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[21] Sandro Queirs, Daniel Barbosa, Brecht Heyde, Pedro Morais, Joo L Vilaa, Denis Friboulet, Olivier Bernard, and Jan Dhooge. Fast automatic myocardial segmentation in 4d cine cmr datasets. *Medical image analysis*, 18(7):1115–1131, 2014.

[22] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition. chapter Learning Representations by Back-Propagating Errors, pages 318–362. MIT Press, Cambridge, 1986.

[24] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[25] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[26] Gustavo Carneiro Tuan Anh Ngo, Zhi Lu. Combining deep learning and level set for the automated segmentation of the left ventricle of the heart from cardiac cine magnetic resonance. *Medical image analysis*, 35:159–171, Jan 2017.

[27] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *Computing Research Repository*, abs/1311.2901, 2013.

[28] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.