Deep Quaternion Networks

Prospectus

by

Chase Gaudet

March 18, 2018

Supervisor:
Dr. Anthony Maida

University of Louisiana at Lafayette
Faculty of Computer Science

Abstract

The field of deep learning has seen significant advancement in recent years. However, much of the existing work has been focused on real-valued numbers. Recent work has shown that a deep learning system using the complex numbers can be deeper for a fixed parameter budget compared to its real-valued counterpart. In this work, we explore the benefits of generalizing one step further into the hyper-complex numbers, quaternions specifically, and provide the architecture components needed to build deep quaternion networks. We go over quaternion convolutions, present a quaternion weight initialization scheme, and present algorithms for quaternion batch-normalization. These pieces are tested in a classification model by end-to-end training on the CIFAR-10 and CIFAR-100 data sets and a segmentation model by end-to-end training on the KITTI Road Segmentation data set. The quaternion networks show improved convergence compared to real-valued and complex-valued networks, especially on the segmentation task.

Dedication

PLACEHOLDER

Contents

Chapter 1

Quaternions

The quaternions are a number system that extends the complex numbers. They were first described by Irish mathematician William Rowan Hamilton in 1843 and applied to mechanics in three-dimensional space. In modern mathematical language, quaternions form a four-dimensional associative normed division algebra over the real numbers, and therefore also a domain. In this chapter we will define operations on the quaternion algebra, draw connection to the complex numbers and $\mathbb{R}^3$, and show some real world use cases of quaternions.

1.1 Quaternion Algebra

In 1833 Hamilton proposed complex numbers $\mathbb{C}$ be defined as the set $\mathbb{R}^2$ of ordered pairs (a, b) of real numbers. He then began working to see if triplets (a, b, c) could extend multiplication of complex numbers. In 1843 he discovered a way to multiply in four dimensions instead of three, but the multiplication lost commutativity. This construction is now known as quaternions. Quaternions are composed of four components, one real part, and three imaginary parts. Typically denoted as

$$\mathbb{H} = \{a + bi + cj + dk \ : \ a, b, c, d \in \mathbb{R}\} \tag{1.1}$$

where a is the real part, (i, j, k) denotes the three imaginary axis, and (b, c, d) denotes the three imaginary components. Sometimes a is referred to as the scalar part and (a, b, c) as the vector part. Quaternions are governed by the following arithmetic:

$$i^2 = j^2 = k^2 = ijk = -1 \tag{1.2}$$

which, by enforcing distributivity, leads to the noncommutative multiplication rules

$$ij = k,\ jk = i,\ ki = j,\ ji = -k,\ kj = -i,\ ik = -j \tag{1.3}$$

1.1.1 Addition and Multiplication

The addition of two quaternions acts component wise, exactly the same as two complex numbers. Consider the quaternion q

$$q = q_0 + q_1 i + q_2 j + q_3 k \tag{1.4}$$

and the quaternion p

$$p = p_0 + p_1 i + p_2 j + p_3 k. \tag{1.5}$$

There addition is given by

$$p + q = (p_0 + q_0) + (p_1 + q_1)i + (p_2 + q_2)j + (p_3 + q_3)k. \tag{1.6}$$

The product of two quaternions will produce another quaternion and is given by

$$\begin{aligned} pq &= (p_0 + p_1 i + p_2 j + p_3 k)(q_0 + q_1 i + q_2 j + q_3 k) \\ &= p_0 q_0 - (p_1 q_1 + p_2 q_2 + p_3 q_3) \\ &\quad + p_0(q_1 i + q_2 j + q_3 k) + q_0(p_1 i + p_2 j + p_3 k) \\ &\quad + (p_2 q_3 - p_3 q_2)i + (p_3 q_1 - p_1 q_3)j + (p_1 q_2 p - p_2 q_1)k. \end{aligned}$$

This can be greatly simplified by utilizing the inner and cross products of two vectors in $\mathbb{R}^3$:

$$pq = p_0 q_0 - \mathbf{p} \cdot \mathbf{q} + p_0 \mathbf{q} + q_0 \mathbf{p} + \mathbf{p} \times \mathbf{q} \tag{1.7}$$

where $\mathbf{p} = (p_1, p_2, p_3)$ and $\mathbf{q} = (q_1, q_2, q_3)$ are the vector parts of p and q respectively.

1.1.2 Conjugate, Norm, and Inverse

The *conjugate* of q, denoted as q^*, is defined as a negation of the vector part of q

$$q^* = q_0 - \mathbf{q} \tag{1.8}$$

and is constructed such that

$$qq^* = q_0 q_0.$$

The *norm* of a quaternion q, denoted as $|q|$, is the scalar

$$|q| = \sqrt{q^* q}$$

and a quaternion is said to be a *unit quaternion* if its norm is 1.

The *inverse* of a quaternion q is defined as

$$q^{-1} = \frac{q^*}{|q|^2},$$

which gives $q^{-1}q = qq^{-1} = 1$. For all unit quaternions, the inverse is equal to the conjugate.

1.2 Geometric Representation of Quaternions

Here we will try to show a more intuitive geometric interpretation of quaternions by showing their link to complex numbers and how they can be used to represent rotations. To do this we will first give a brief reminder of complex numbers and their geometric interpretation of rotating a 2D plane.

1.2.1 Complex Algebra

Complex numbers are composed of two components, one real part, and one imaginary part. This is usually denoted as

$$\mathbb{C} = \{a + bi \ : \ a, b \in \mathbb{R}, \ \ i^2 = -1\} \tag{1.9}$$

where a is the real part, i denotes the single imaginary axis, and b denotes the single imaginary component.

Let c and d be two complex numbers, their addition is given by

$$c + d = (c_0 + c_1 i) + (d_0 + d_1 i) = (c_0 + d_0) + (c_1 + d_1)i \tag{1.10}$$

and their multiplication by

$$cd = (c_0 d_0 - c_1 d_1) + (c_0 d_1 + c_1 d_0)i. \tag{1.11}$$

Any complex number has a length given by

$$|c| = |c_0 + c_1 i| = \sqrt{c_0^2 + c_1^2} \tag{1.12}$$

and like quaternions, any complex number with a length of 1 is called a *unit complex number*.

1.2.2 Complex Rotation Operation

The set of unit complex numbers lies on the unit circle in $\mathbb{C}$ and Leonhard Euler showed that

$$e^{i\theta} = \cos\theta + i \sin\theta. \tag{1.13}$$

If we multiply this by any positive number r, we get a complex number of length r. Therefore, by adjusting the length r and the angle θ, we can write any complex number. This form goes by the name *polar coordinates*.

They are a great way to multiply complex numbers. Instead of (1.11) let us write each complex in polar coordinates

$$c = (c_0 + c_1 i) = re^{i\theta}, \quad d = (d_0 + d_1 i) = se^{i\phi}$$

and then multiply

$$cd = re^{i\theta} se^{i\phi} = rse^{i(\theta+\phi)}.$$

This tells us that to multiply two complex numbers, multiply their lengths and add their angles. In particular, if we multiply a given complex number by a unit complex number the resulting length is the same, but we have rotated it by θ degrees. This gives us some hints that quaternions may also be able to perform rotation operation in higher dimensional space.

1.2.3 Quaternion Rotation Operation

We will stick to interpretations of quaternions in $\mathbb{R}^3$ as it is easier to visualize, but how can a quaternion, which exists in $\mathbb{R}^4$, operate on a 3D vector? Recall that the imaginary components of a quaternion is called the vector part and a vector $\mathbf{v} \in \mathbb{R}^3$ is a *pure quaternion* whose real part is zero.

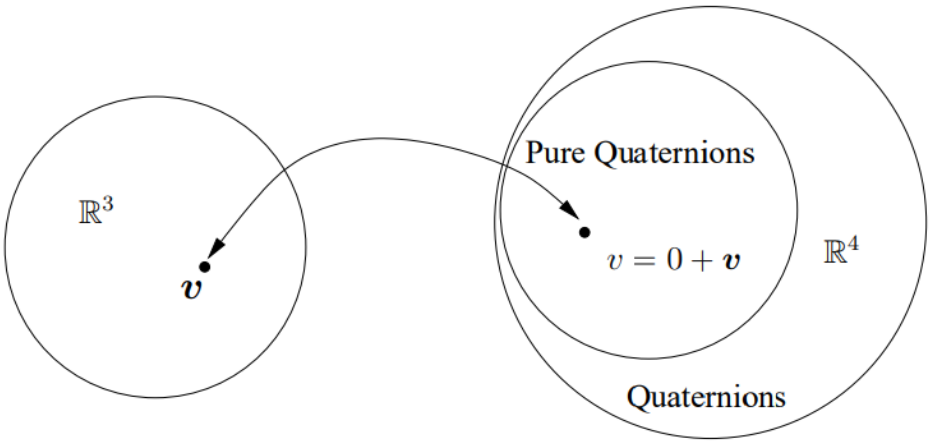


Figure 1.1: $\mathbb{R}^3$ can be viewed as a subspace of quaternions called pure quaternions which have a real part of zero.

Using the unit quaternion q let us define a function $L_q : \mathbb{R}^3 \to \mathbb{R}^3$ on vectors $\mathbf{v} \in \mathbb{R}^3$:

$$\begin{aligned} L_q(\mathbf{v}) &= q\mathbf{v}q^* \\ &= (q_0^2 - ||\mathbf{q}||^2)\mathbf{v} + 2(\mathbf{q} \cdot \mathbf{v})\mathbf{q} + 2q_0(\mathbf{q} \times \mathbf{v}). \end{aligned} \tag{1.14}$$

Two observations to note are that first, the operation (1.14) does not modify the length of the vector $\mathbf{v}$:

$$\begin{aligned} ||L_q(\mathbf{v})|| &= ||q\mathbf{v}q^*|| \\ &= |q| \cdot ||\mathbf{v}|| \cdot |q^*| \\ &= ||\mathbf{v}||. \end{aligned}$$

And second, the direction of $\mathbf{v}$, if along $\mathbf{q}$, is left unchanged by the function. To verify let $\mathbf{v} = k\mathbf{q}$

$$\begin{aligned} L_q(\mathbf{v}) &= q(k\mathbf{q})q^* \\ &= (q_0^2 - ||\mathbf{q}||^2)k\mathbf{q} + 2(\mathbf{q} \cdot k\mathbf{q})\mathbf{q} + 2q_0(\mathbf{q} \times k\mathbf{q}) \\ &= k(q_0^2 + ||\mathbf{q}||^2)\mathbf{q} \\ &= k\mathbf{q}. \end{aligned}$$

Using our insights from complex numbers this lets us guess that the function (1.14) acts like a rotation about $\mathbf{q}$.

Theorem 1. *For any unit quaternion*

$$q = q_0 + \boldsymbol{q} = cos\frac{\theta}{2} + \hat{\mathbf{u}}sin\frac{\theta}{2}, \tag{1.15}$$

and for any vector $\boldsymbol{v} \in \mathbb{R}^3$ the result of the function

$$L_q(\boldsymbol{v}) = q\boldsymbol{v}q^*$$

on $\boldsymbol{v}$ is equivalent to a rotation of the vector through an angle θ about $\hat{\mathbf{u}}$ as the axis of rotation.

Proof. Given a vector $\mathbf{v} \in \mathbb{R}^3$, we decompose it as $\mathbf{v} = \mathbf{a} + \mathbf{n}$, where $\mathbf{a}$ is the component along the vector $\mathbf{q}$ and $\mathbf{n}$ is the component normal to $\mathbf{q}$. Then we show that under the function L_q, $\mathbf{a}$ is invariant, while $\mathbf{n}$ is rotated about $\mathbf{q}$ through an angle θ.

Earlier we showed that $\mathbf{a}$ is invariant under L_q so let us see how L_q transform $\mathbf{n}$.

$$\begin{aligned} L_q(\mathbf{n}) &= (q_0^2 - ||\mathbf{q}||^2)\mathbf{n} + 2(\mathbf{q} \cdot \mathbf{n})\mathbf{q} + 2q_0(\mathbf{q} \times \mathbf{n}) \\ &= (q_0^2 - ||\mathbf{q}||^2)\mathbf{n} + 2q_0(\mathbf{q} \times \mathbf{n}) \\ &= (q_0^2 - ||\mathbf{q}||^2)\mathbf{n} + 2q_0||\mathbf{q}||(\hat{\mathbf{u}} \times \mathbf{n}), \end{aligned}$$

where in the last step we introduced $\hat{\mathbf{u}} = \mathbf{q}/||\mathbf{q}||$. Let $\mathbf{n}_\perp = \hat{\mathbf{u}} \times \mathbf{n}$ to get

$$L_q(\mathbf{n}) = (q_0^2 - ||\mathbf{q}||^2)\mathbf{n} + 2q_0||\mathbf{q}||\mathbf{n}_\perp. \tag{1.16}$$

Also note that $\mathbf{n}_\perp$ and $\mathbf{n}$ have the same length:

$$||\mathbf{n}_\perp|| = ||\mathbf{n} \times \hat{\mathbf{u}}|| = ||\mathbf{n}|| \cdot ||\hat{\mathbf{u}}|| \sin\frac{\pi}{2} = ||\mathbf{n}||.$$

Then rewriting (1.16) we arrive at

$$\begin{aligned} L_q(\mathbf{n}) &= \left(\cos^2\frac{\theta}{2} - \sin^2\frac{\theta}{2}\right)\mathbf{n} + \left(2\cos\frac{\theta}{2}\sin\frac{\theta}{2}\right)\mathbf{n}_\perp \\ &= \cos\,\theta\mathbf{n} + \sin\,\theta\mathbf{n}_\perp. \end{aligned}$$

The resulting vector is a rotation of $\mathbf{n}$ through an angle θ in the plane defined by $\mathbf{n}$ and $\mathbf{n}_\perp$. □

Chapter 2

Introduction To Convolutional Neural Networks

The Convolutional Neural Network (CNN) is a well-known deep learning architecture that was loosely inspired by the natural visual perception mechanism of some living organisms. This comes from a paper by Hubel and Wiesel [HW68] in 1959 where they found that their were specific cells in the visual cortex of animals that were responsible for detecting light in the receptive field, which eventually won them a Nobel Prize. It was not until 1990 that LeCun et al. [LBD+90] released the crucial paper that would establish modern framework for CNNs. Their model was a multiple layer neural network called LeNet-5 that could classify 28x28 greyscale hand written digit images with very high accuracy. As seen in Fig. 2.1 LeNet-5 has multiple layers and can be trained with the back-propagation algorithm [HN+88], but used a low number of parameter values for its size. To achieve this LeNet-5 used two core components, which we will discuss in the next section.

2.1 Basic CNN Components

While there are many different layer types in current literature, here we will focus on the two introduced in LeNet-5, which are still used today along with

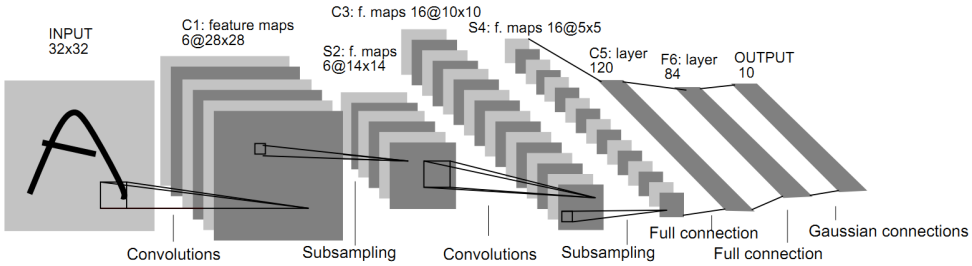


Figure 2.1: The architecture for LeNet-5, which is composed of repeating convolution and pooling layers.

some variants of them.

2.1.1 Convolutional Layer

The first is the convolution layer. Its purpose is to learn feature representations of the inputs. Each convolution layer has a specified number of kernels where each kernel is unique as to generate a distinct feature map. Each neuron of a feature map is connected to a region of neighboring neurons in the previous layer. The size of the connection is known as the kernel size or the receptive field size. The feature map is obtained by first convolving the input with the kernel and then applying an element-wise nonlinear activation function on the result. Because it is a convolution operation each kernel is shared with all spatial locations of the input. This spatial weight sharing is what creates the low parameter number compared to a regular neural network which fully connects all inputs and outputs. It also allows the CNN to take advantage of the underlying structure in images. Topological information, i.e., spatial information about the structure in an image, such as adjacency and rotations are also taken into account. The equation for finding the feature value at location (i, j) in the k^{th} feature map of the l^{th} layer, $z^l_{i,j,k}$ is

$$z^l_{i,j,k} = {\mathbf{w}^l_k}^T \mathbf{x}^l_{i,j} + b^l_k \tag{2.1}$$

where $\mathbf{w}^l_k$ and b^l_k are the weight vector, or convolution kernel, and the bias term of the l^{th} layer respectively, and $\mathbf{x}^l_{i,j}$ is the input patch centered at location (i, j) of the l^{th} layer. An example of this operation can be seen in

Fig. 2.2. After the convolution is applied the result is then put through the nonlinear activation function:

$$a^l_{i,j,k} = a(z^l_{i,j,k}) \tag{2.2}$$

where $a(\cdot)$ is the nonlinear activation function. We will discuss different activation functions in great detail in Chapter 3.

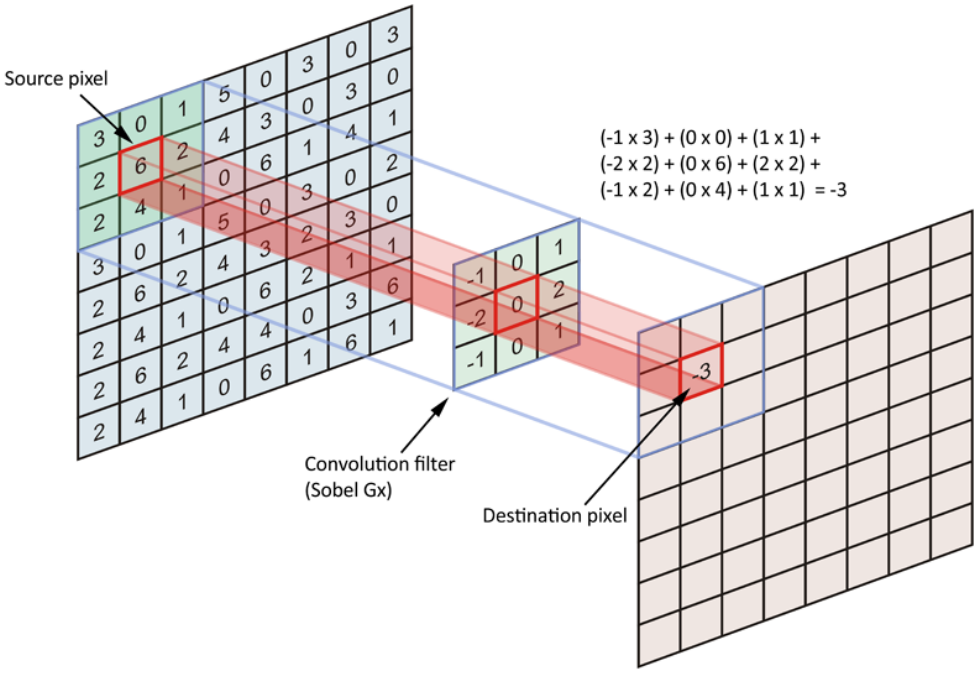


Figure 2.2: Convolution operation performed on a single feature location and one convolution kernel [App00].

2.1.2 Pooling Layer

The other key layer is the pooling layer shown in Fig.2.1 as subsampling. The pooling layer comes after a convolution layer and it performs a reduction in the resolution of the feature map. This layer works typically with two arguments: the spatial size of the pooling window F and the stride, or shift per operation, S. It is accomplished in a similar manner to convolution, starting at the top right a window of size $F \times F$ is grabbed. From this

window the pooling function is used, which is commonly a **MAX** function. The window then moves over by S pixels and the above operation happens again. This is repeated until the whole image is covered. For example, if one chooses $F = 2$ and $S = 2$ they will be selecting the maximum pixel in a 2×2 window and moving 2 pixels over exactly as shown in Fig. 2.3. Note that this reduces the original image by a factor of $F = 2$. This serves the purpose of reducing the number of multiplications the architecture must perform and also aims to achieve shift invariance. We will discuss different pooling techniques in Chapter 3.

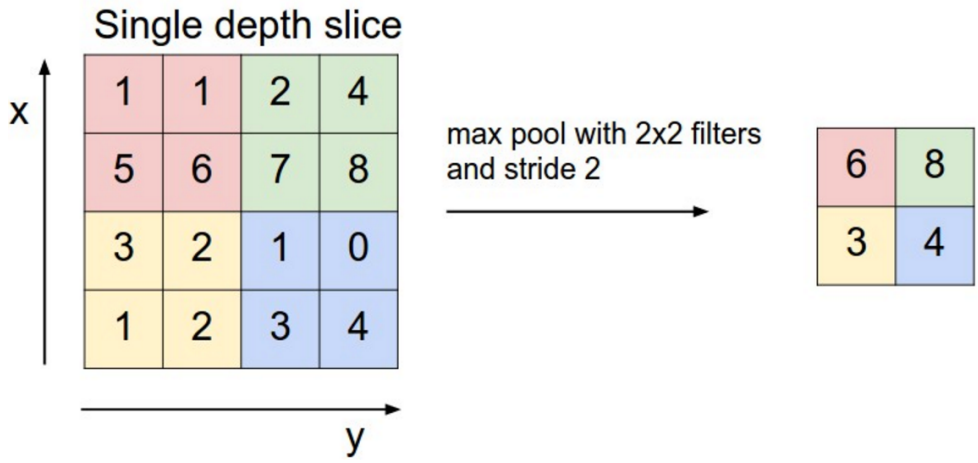


Figure 2.3: Example max pooling operation with a size of 2x2. A 2x2 window is moved over the input with a stride of 2 and the maximum value of the window is taken [Kar13].

There is also the inverse of pooling, often called upsampling, that will increase the resolution of the feature maps. There are several cases where this is a desired effect which we will discuss later.

2.1.3 CNN Common Tasks

The last layers of LeNet-5 are fully connected layers like that of a regular Feed-Forward Neural Network (FNN), sometimes called Multilayer Percep-

trons [RHW85]. The second to last layer is sometimes called a decision layer as it takes the outputs from the feature mapping and combines them together. The last layer is a fully connected layer of size 10, which corresponds to the number of classes in that particular data set. The classes, which are digits numbered 0 through 9, are represented as a 'one hot' vector, meaning that if the class is the digit '0' the first element of the output vector should be 1 and the rest should be 0. This can be thought of as a probability distribution on the classes. This is an important representation because it allows one to use loss functions that minimizes the differences of distributions opening up many potential loss functions. This was the common use case for CNNs for a while, given an image predict a distribution over a set of classes. This task is known as *classification*.

Recently CNNs have seen use in another area known as *semantic segmentation*. Semantic segmentation, also sometimes just called segmentation, is the process of partitioning an image into multiple segments, or sets of pixels. More precisely, segmentation is the process of assigning a class from our classification set to each pixel, or potentially area of pixels, in an image. Unlike classification, the output of this CNN would typically be in the form of an image. Typically the network will output a distribution over a set of classes *per pixel* and then the pixel is labeled with the class with the highest probability. There are many applications where image classification alone does not giving enough information and one needs pixel-level labels. Examples include: detecting road signs [MBLAGJ+07], detecting tumors in medical imaging [LJH15, LPAL15, KPU15, HDWF+17], detecting objects of interest from satellite photos [CXLP13], finding pedestrians [DEKLD16], and many more. Having pixel level labels allows multiple objects of different classes to be detected in one image compared to image level labels where there is a single output per image.

In recent years, CNNs have obtained state of the art results on almost all classification and segmentation data sets. The improvements in CNNs have come from a few different areas including better computers, improvements to initialization of network weights, and specific architectures to combat problem areas of deep networks. In this next chapter we will go through a list of improvements and key papers.

Chapter 3

State of the Art

In this chapter we will go over the latest improvements for CNNs. They range from the activation function used within each neuron to major architecture overhauls from the original LeNet-5.

3.1 Recent Advances in Training Neural Networks

3.1.1 Activation Functions

There have been many recent papers on improving the results of NNs in general. One focus has been on the activation functions used inside the network. The current trend has been using activation functions that are non-saturated, meaning they do not have very small derivatives at large input values. All mentioned activation functions can be seen in Fig. 3.1.

ReLU: The most notable non-saturated activation function is the rectified linear unit (ReLU) [NH10]. This is defined as:

$$f(x) = \max(0, x). \tag{3.1}$$

The benefits of this unit include: faster calculation since the $\max(\cdot)$ operation is faster than sigmoid or tanh, it creates sparsity in the hidden units by giving true zero values often to help sparse representations form, and does not suffer from vanishing gradients in deep models. ReLU has been shown to work better than sigmoid or tanh in several tasks and shows fast convergence even without pretraining [GBB11, KSH12, ZRM$^+$13, MHN13].

LReLU: The ReLU has zero gradient whenever its inputs add to a negative value or when a large gradient changes the weights to large negatives value making future input into the unit always have a negative value. These units will never be updated and learning will not occur, which can be a problem. Mass *et al.* [MHN13] found a solution to this by introducing a non zero component to the ReLU when the input is negative. They called this unit the Leaky ReLU as it 'leaks' a positive gradient on the negative side of its graph by having a very small slope. This leaky ReLU or LReLU is defined as:

$$f(x) = \max(0, x) + \lambda \min(0, x) \tag{3.2}$$

where $\lambda \in (0, 1)$ and is predefined per model. This unit allows for small gradients when the unit is not active, which helps the problems of the ReLU unit.

PReLU: Rather than trying to find an optimal value for λ in the LReLU, He et al. [HZRS15b] introduced the Parametric ReLU which adapts the λ during training. It is defined as:

$$f(x) = \max(0, x) + \lambda_k \min(0, x) \tag{3.3}$$

where λ_k is the k-th channel. Since the number of parameters in networks is often very large compared to the number of total channels, the extra computational cost to learn the values of λ_k's is not much.

ELU: Next is the Exponential Linear Unit (ELU) [CUH15]. ELUs are similar to the above mentioned units, but they have a saturating function on their negative side. The saturation function decreases the variation of the

units if they are not activated, which gives those units a chance to update while making them more robust to noise. The ELU function is defined as:

$$f(x) = \max(0, x) + \min(0, \lambda(e^x - 1)) \tag{3.4}$$

where λ is predefined as in the LReLU.

PELU: A natural extension of the ELU is the Parametric ELU [TGCd16]. Unlike the PReLU which learns one parameter to modify the LReLU, the PELU learns two parameters during training to modify the ELU. This gives the network more control over the vanishing gradients. The PELU is defined as:

$$f(x) = \max(0, \frac{a}{b}x) + \min(0, a(e^{x/b} - 1)) \tag{3.5}$$

where $a, b > 0$. Both a and b change the slope of the linear function together, b effects the scale of the exponential decay, and a is the saturation point in the negative side of the function. In [TGCd16] it is shown that PELU does a better job than all the above functions in several different models and data sets.

3.1.2 Regularization

Some of the biggest improvements to NNs have been through regularizing the network in different ways to prevent overfitting.

Dropout: Introduced by Hinton *et al.* [HSK+12], Dropout has shown to be very effective at improving network results and reducing overfitting. In their paper Dropout is applied to fully-connected layers where the output of the Dropout layer is

$$\mathbf{y} = \mathbf{r} * a(\mathbf{W}^T\mathbf{x}), \tag{3.6}$$

where $\mathbf{x} = [x_1, x_2, ..., x_n]^T$ is the input to the fully-connected layer, $a(\cdot)$ is an activation function, $\mathbf{W}$ is the weight matrix, $*$ is an element-wise multiplication, and $\mathbf{r}$ is a binary vector whose elements are independently drawn from a Bernoulli distribution with parameter p. This means that at every update to the network, each neuron has a p chance of getting multiplied by

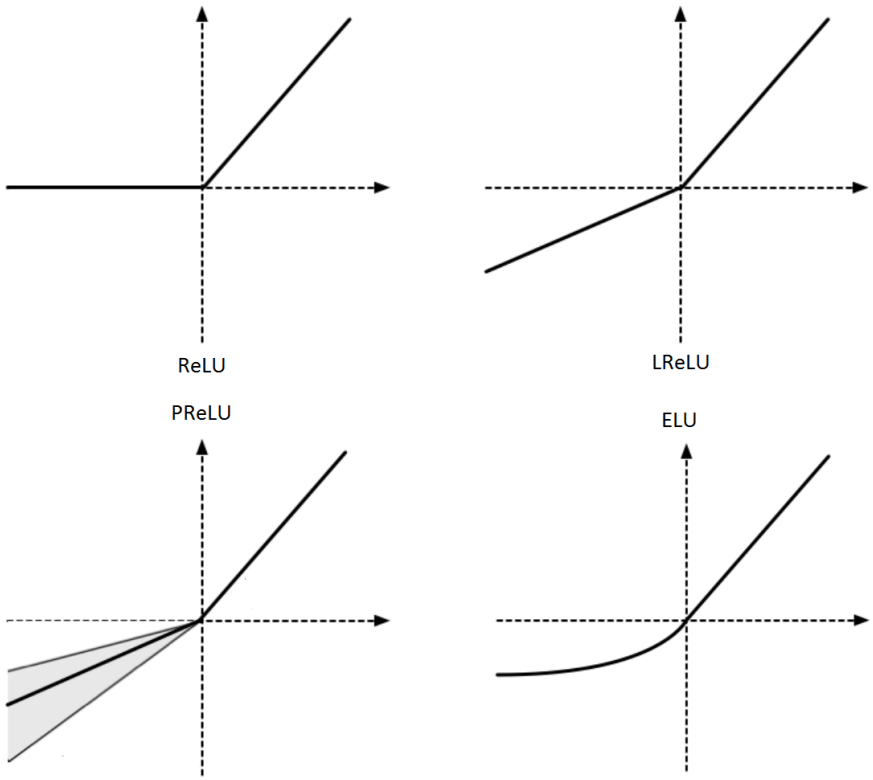


Figure 3.1: Examples of some of the discussed activation functions. From top to bottom and left to right: ReLU, LReLU, PReLU, and ELU. []

zero. Dropout usually prevents the network from becoming too reliant on a few neurons and helps the network generalize by spreading feature information. Some extensions to Dropout have been proposed like in [WM13] where a method to improve the speed of training while using Dropout are introduced. Similar to the learned parameters of the activation functions, [BF13] learn the p in the Dropout distribution adaptively. Finally, in [TGJ$^+$15] they modify Dropout with CNNs specifically in mind by extending the Dropout value across the entire feature map, meaning that during training each feature map has a p chance of being completely ignored for each update step. They call this method Spatial Dropout and it seems to help the network learn more general features, which improves results on limited size data sets.

Batch Normalization: It is standard practice to normalize data to have zero-mean and unit variance before feeding it into a NN, but as the data goes through the network, especially deep networks, the data will lose this property. This change to the input distribution is known as internal covariance shift. To keep data normed through the network [IS15] introduced an efficient method called Batch Normalization (BN). BN works by normalizing the mean and variance of each layers input using each batch rather than the entire training set. To demonstrate BN, let $\mathbf{x} = [x_1, x_2, ..., x_n]^T$ be a n dimensional input to a layer. The k-th dimensions is normalized by:

$$\hat{x}_k = (x_k - \mu_B)/\sqrt{\sigma_B^2 + \epsilon} \tag{3.7}$$

where μ_B and σ_B^2 are the mean and variance of the batch respectively and ϵ is some small constant value that will guarantee the root term is always defined. Finally the input $\hat{x}_k$ is further transformed:

$$y_k = \mathrm{BN}_{\alpha,\beta}(x_k) = \alpha\hat{x}_k + \beta \tag{3.8}$$

where α and β are parameters learned during training. There are many benefits to using BN. By reducing internal covariance shift the network will converge faster. By making sure no inputs get too large or small BN makes it possible to use saturating activation functions without fear of getting stuck in a saturating section and having vanishing gradients. This is very important when training GANs, which are extremely sensitive to the issues BN addresses.

3.1.3 Architecture Improvements

There are a couple of recent architecture improvements to NNs in general that inspired some of the newer ideas in segmentation. The first was introduced in [LCY13] where they use fully connected layers on the outputs of each convolution operation called Network-in-network. The idea was an alternative to stacking multiple convolutional layers (each with a number of their own filters) to get deepness in a neural network model, by replacing each filter with a multi-layer perceptron, which is essentially a small neural network that slides across the image like a convolutional filter. The math works out so that a $1 \times 1 \times U$ convolution filter convolved across a V-channel

image emulates a $U \times V$ matrix multiplied by each V-channel pixel, which is the same as running a single-layer neural network across every pixel of your input as if each pixel were an example vector in a training set. Chaining together such $1 \times 1 \times U$ convolutions, you get the same result as if running a many-layered neural network at each input pixel of V channels. The idea from the paper was to turn a convolution filter which is a generalized linear model, into a non-linear model. It allows the network to combine channels from previous layer in a non-linear fashion, which can lead to more advanced features.

Google's team was inspired by the Network-in-network paper and saw the power of using the 1x1 convolutions not only for non-linearaity, but to reduce computational burden of deep models by using the cross-channel pooling aspect to reduce their feature maps before convolution layers. In [SLJ+15] Christian Szegedy and his team introduced the Inception module as seen in Fig.3.2.

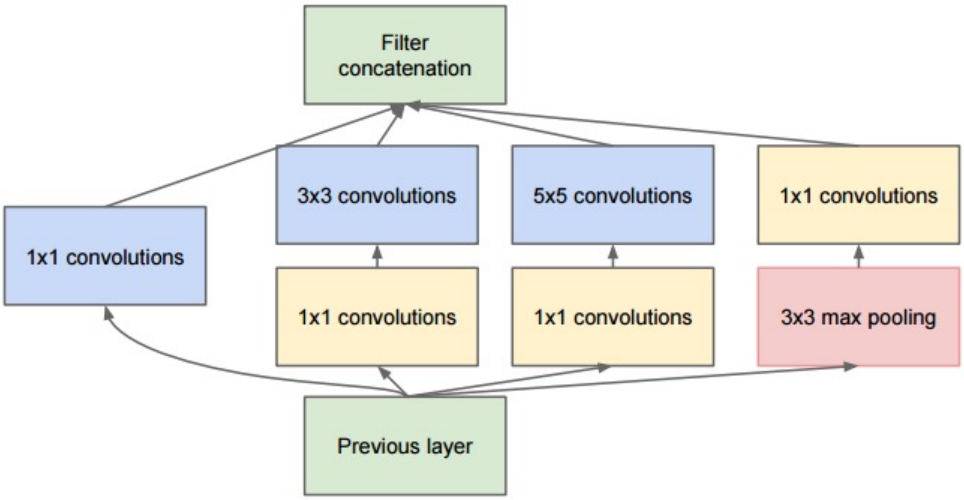


Figure 3.2: An inception block from [SLJ+15]. Notice the 1x1 convolutions before the 3x3 and 5x5 convolutions are reducing the number of feature channels. This reduces the number of multiplications that must be performed.

Another work that focused on fixing the vanishing gradient problem of deep networks was Residual Nets (ResNets) [HZRS15a] where they used what

is now called shortcut connections. These connections were inspired by Long Short Term Memory (LSTM) units, a type of recurrent neural network unit that feeds information back into itself with weighted gates. Instead of using weighted gates shortcut connections pass information with the identify function so it is untransformed. This means that the activation of deep units can be written as the sum of the activation of some shallower unit and a residual function, which is a series of network layers. Or to put simply, the input to a layer group is added to the output of that layer group. This is called a Residual Block and can be seen in Fig.3.3. By stacking these residual blocks together one gets a fully residual model. This design allows the gradients to be directly propagated to shallower units allowing for networks of 100s of layers to be trained.

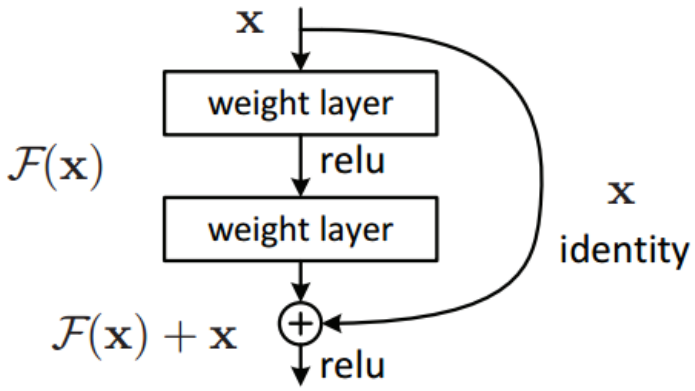


Figure 3.3: A residual block from [HZRS15a].

3.2 Recent Advances in CNNs for Semantic Segmentation

The area of segmentation has recently seen many improvements using methods mentioned in the Section 3.1. The highlight papers that brought big improvements to the area and that inspired the work in this thesis will be discussed each in their own section.

3.2.1 Fully Convolutional Networks for Semantic Segmentation

In [LSD15] Long *et al.* used 1x1 convolutional kernels at the end of their network, instead of a fully connected layer, to produce a dense heatmap prediction of their classes as seen in Fig. 3.5. They then used *backwards convolution*, sometimes called *deconvolution* or *transposed convolution*, with some input stride to upsample this dense heatmap back up to the original input image size. By using this method to upsample they achieve a non-linear learned upsampling. Backwards convolution works by taking the input image and padding zeros between the pixels based on the stride of the convolution shown in Fig. 3.4. The result is a segmentation of classes of the same size as the input image trained fully end to end. However, the results are very coarse

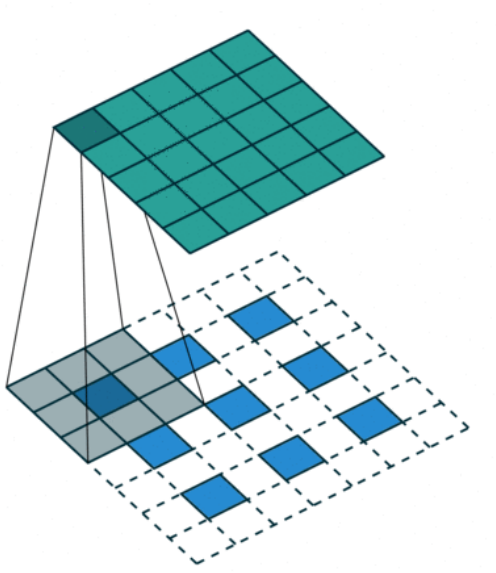


Figure 3.4: Backwards convolution with a stride of 2 upsampling a 3x3 image to a 5x5 image. Notice the 3x3 image is padded with zeros, the white tiles. The shaded grey area is the actual convolution kernel, which is learned during training.

even with learned upsampling from the deconvolution layers. To remedy this they used the idea of short connections to forward information from shallow

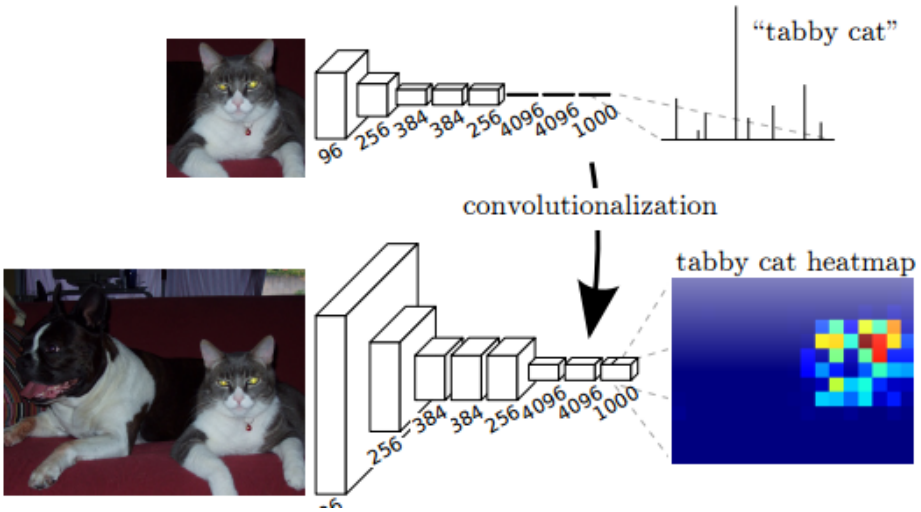


Figure 3.5: Transforming fully connected layers into convolution layers enables a classification net to output a heatmap [LSD15]. This is achieved by using 1x1 convolutions instead of fully connected layers and will produce a tiny heatmap of classes. This change also make the network become independent of input image size.

layers, upsample them to the correct matching size of the current layer, and then combine them with the current layer. This adds the finer grain detail that the shallow layers possess that get lost after pooling operations. Some of the different architectures can be seen in Fig. 3.6. Their three models used increasing number of short connections, which increased the accuracy with each additional connection. It is important to note that in this paper they used weights from a pre-trained classification network and only the new backward convolution layers and onward were learned from the segmentation maps.

3.2.2 U-Net: Convolutional Networks for Biomedical Image Segmentation

Ronneberger *et al.* in [RFB15] directly mention that they are trying to improve upon results from [LSD15]. Since their application is medical seg-

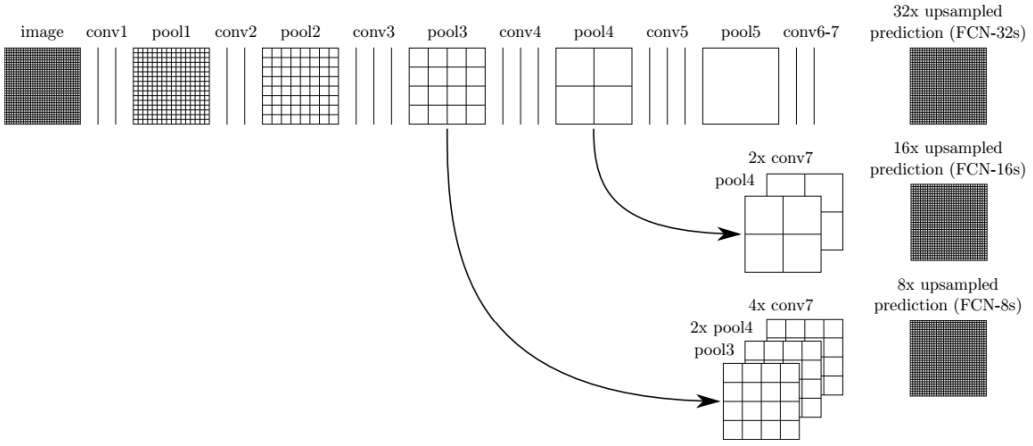


Figure 3.6: Three different architectures from [LSD15]. The first uses no short connections, the second forwards information from pool4, and the third forwards information from both pool3 and pool4.

mentation the maps cannot be coarse so their aim is to improve in that area specifically as well as allowing for the network to be trained end-to-end, or no loading pre-trained weights. Their architecture consists of a contracting path (left side) into an expansive path (right side). The left side is a typical CNN and the right side is the inverse of the left, using deconvolution to upsample. Heavy use of short connections is used by forwarding information from every layer from the first half of their model to the matching size layer of the second half where it is then concatenated as shown in Fig. 3.7. This short connection from every layer in the contracting side differs from Long *et al.*, which used only two at most. They also use an overlapping tile prediction strategy, which combats the edge artifacts from padding convolution operations with zeros. This means the prediction is only done on some middle portion of the input image.

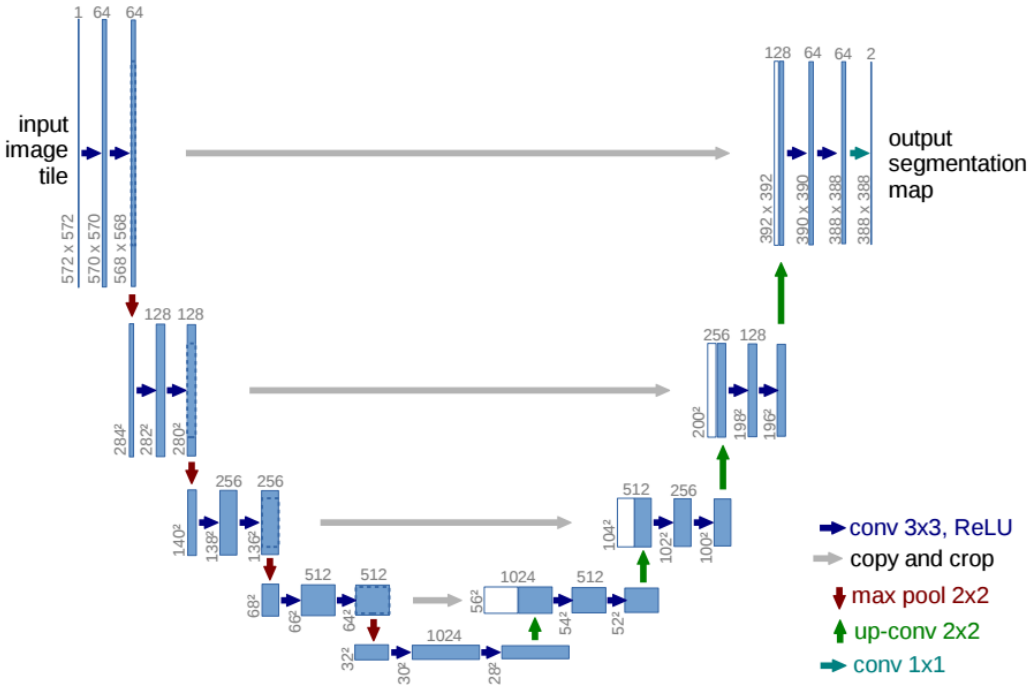


Figure 3.7: U-Net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations. [RFB15].

3.2.3 Learning Iterative Processes with Recurrent Neural Networks to Correct Satellite Image Classification Maps

Maggiori *et al.* at Inria in [MTCA16] introduced a post processing network RNN whose sole purpose was to enhance segmentation results from a CNN. They formulate a generic partial differential equation (PDE) diffusion process applied to a map u, which is the segmentation result from the first network. They take as input a score map u_k (for class k) and an arbitrary number of feature maps $g_1, g_2, ..., g_p$ derived from image I. Then convolution kernels $M_1, M_2, ...$ and $N_1, N_2, ...$ are applied to the heat map u_k and the features g_j

derived from image I respectively. These filters are learned during training. These feature maps from applying these convolutions are gathered into a set:

$$\Phi(u_k, I) = \left\{ M_i * u_k,\ N_l^j * g_j(I);\ \forall i, j, l \right\}. \tag{3.9}$$

Then u_k can be expressed as changing in time as

$$\frac{\partial u_k(x)}{\partial t} = f_k(\Phi(u_k, I)(x)), \tag{3.10}$$

where f_k is a function that takes as input the values of all the features in $\Phi(u_k, I)$ at an image point x and combines them. Since it is discretized in time it takes the form:

$$u_{k,t+1}(x) = u_{k,t}(x) + \delta u_{k,t}(x), \tag{3.11}$$

where $\delta u_{k,t}$ is the overall update of $u_{k,t}$ at time t. This $\delta u_{k,t}$ is the value that the post processing network will be trying to approximate using convolution results fed into a fully connected layer, known to be able to approximate any function, shown in Fig.3.8. They do this in an iterative fashion where the update $u_{k,t+1}(x)$ is fed into the same network with shared weights N number of times. They use $N = 5$ in their paper. This iterative step is a recurrent network and has a couple of advantages. First the weight sharing cuts the number of parameters down by a factor of N, making it easier to train. And second, the weight sharing is more physically realistic to a PDE. At each update the physics should remain unchanged, which would not be modeled properly if each iteration had a unique set of weights. An example of the complete RNN can be seen in Fig.3.9.

3.2.4 Semantic Segmentation using Adversarial Networks

Brief overview of GANs

We will start with a brief overview of the basics of a Generative Adversarial Network (GAN). Goodfellow *et al.* introduced GANs in [GPAM$^+$14] as a new way to estimate generative models, which are models that generate new

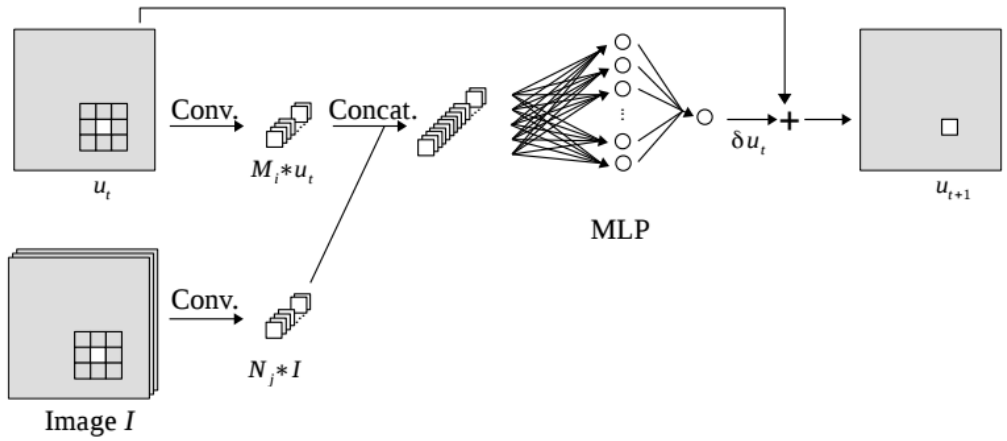


Figure 3.8: One iteration of the RNN represented as common neural network layers. The block approximates $\delta u_{k,t}$ and then adds it to $u_{k,t}$ to estimate $\delta u_{k,t}$. [MTCA16]

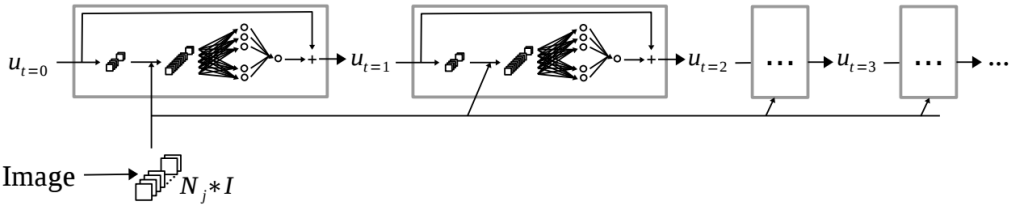


Figure 3.9: The whole RNN shown by multiple blocks from Fig.3.8 with shared weights. [MTCA16].

data with parameters significantly smaller than the amount of data we train them on. So the models are forced to discover and efficiently internalize the essence of the data in order to generate it.

The basic idea of GANs is to set up a game or battle between two networks. One is the generator, which creates samples that are suppose to come from the training set distribution. The other network is the discriminator, which examines an input and predicts whether it is real or fake. For example let us say our generator takes a random array of numbers sampled from a Gaussian distribution and from that array produces an image. The discriminator takes in the image and does a binary classification of either real or

fake, that is, to indicate whether it is from the generator distribution or not. To fool the discriminator, the generator must learn to create images that are indistinguishable from an image from the training set.

Let us write our two networks as two functions, each of which is differentiable with respect to its own inputs and respect to its own parameters. The discriminator is a function D that takes x as input and uses θ^D as parameters. The generator is a function G that takes z as input and uses θ^G as parameters. Each network has its own cost function that it seeks to minimize. The discriminator wants to minimize $J^D(\theta^D, \theta^G)$, but must do so while only controlling θ^D. Likewise, the generator wants to minimize $J^G(\theta^D, \theta^G)$, but must do so while only controlling θ^G. The solution to this coupled loss system is a Nash equilibrium [RBS13]. In this context a Nash equilibrium is a tuple (θ^D, θ^G) that is a local minimum of J^D with respect to θ^D and a local minimum of J^G with respect to θ^G.

GANs are notoriously hard to train and much work has been done on stabilizing training. The training process involves simultaneous minimization updates to each network. On each update two batches are sampled: a batch of x values from the training set and a batch of z values drawn from the generator. Then two gradient steps are made simultaneously: one updating θ^D to reduce J^D and the other updating θ^G to reduce J^G.

After the first GAN paper many people began working on the issue of training stability and trying to produce larger, more realistic images. The first major work released was Radford *et al.* [RMC15] that showed some significant improvements. They saw that using Batch Normalization in both the discriminator and generator is a must, having any fully connected layers is a bad idea, and instead of pooling use a strided convolution.

The next big improvement came from Salimans *et al.* [SGZ+16] where they did slight modifications to the learning procedure. Instead of having the generator trying to fool the discriminator, they proposed a new objective function. This objective requires the generator to generate data that matches the statistics of the real data. Then the discriminator is only used to specify which statistics are worth matching. They also showed that smoothing the training labels by making the discriminator target output from [0=fake image, 1=real image] to [0=fake image, 0.9=real image] improved training

stability. Their modifications focused on the loss function of the GAN, which we will see in the next two papers is a large area for improvement.

Recently some new papers have come out showing using different loss functions can greatly improve training stability and results. The first is Arjovsky *et al.* [ACB17] where they use the Wasserstein distance instead of the typical Kullback-Leibler divergence. Unlike Kullback-Leibler divergence, which is not defined if they generator's learned distribution and the true distribution do not overlap well, the Wasserstein distance has a well defined gradient everywhere. The second is Mao *et al.* [XM17] where they reformulate the discriminator's loss as a least squares loss. The binary loss typically used does not provide more weight from a sample right next to the decision boundary to a sample very far away from the boundary. Meaning that samples that barely pass the boundary are treated the same as samples that did very well. By using the least square loss they fix this issue.

GANs for segmentation

One major issue of using CNNs for segmentation is that spatial contiguity is typically lacking, meaning that the CNN does not use long range information when classifying a pixel. We have shown some ways to combat the issue above like forwarding information from earlier layers to later layers and in Section 3.2.3 where a RNN model is run on the CNNs results to improve boundary sharpness. Luc *et al.* [LCCV16] tackled this issue by formulating the segmentation task in a way that a GAN model could be trained to accomplish as shown in Fig. 3.10. Their model consists of the segmentor, which is the generator in the typical GAN framework, and the adversarial network is the discriminator. The segmentor takes an image and produces a segmentation of class predictions. The discriminator is then fed the same image and either the actual ground truth segmentation label or the segmentor's segmentation. It then predicts if the segmentation is the ground truth label or the segmentor's result. By using a GAN they combine the conventional cross-entropy loss over each pixel with an adversarial term. The adversarial term encourages the segmentation model, which is the generator, to produce label maps that cannot be distinguished from the ground-truth labels of the training set by the discriminator. Since the adversarial model can

assess the joint configuration of many label variables, it can enforce forms of higher-order consistency that cannot be enforced using pair-wise terms, nor measured by a per-pixel cross-entropy loss.

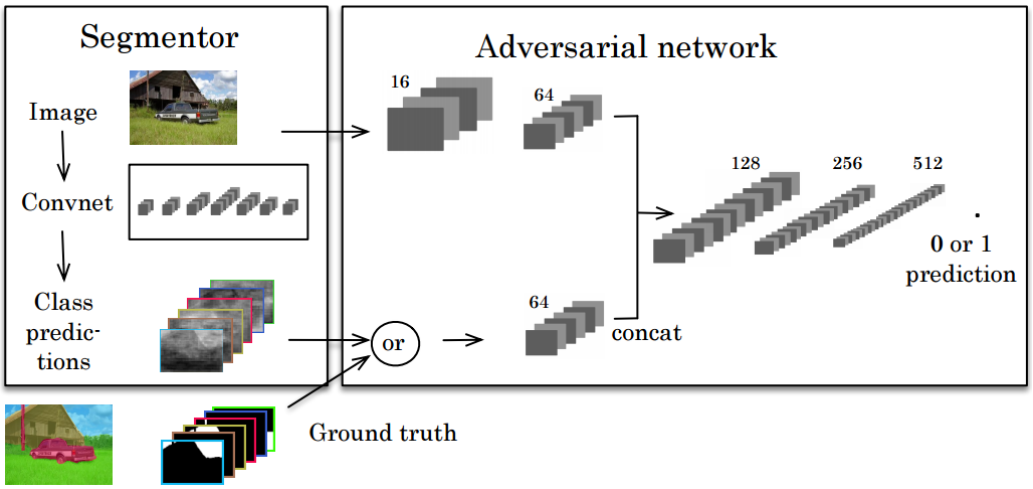


Figure 3.10: GAN formulation of semantic segmentation from [LCCV16]. Left: segmentation net takes RGB image as input, and produces per-pixel class predictions. Right: Adversarial net takes label map as input and produces class label (1=ground truth, or 0=synthetic).

3.3 Open Questions

There have been so many individual contributions to segmentation in the past couple of years, but combining these improvements into mixing architectures has not been studied. We intend to explore different combinations of good architectures to see if state-of-the-art can be improved. By creating the best mixed architecture we will not only have a better stand alone segmentation model, but we can hopefully further improve those results by both using it as a generator of a GAN model with untested loss functions for GAN segmentation and applying the iterative RNN process to the resulting segmentation heatmaps.

Chapter 4

Work Completed

The goal is to find which combination of new techniques provides the greatest overall segmentation results. To that end we have taken a particularly challenging data set, the 2D EM segmentation challenge [onsiEs12]. The training data is a set of 30 sections from a serial section Transmission Electron Microscopy (ssTEM) data set of the Drosophila first instar larva ventral nerve cord (VNC). The microcube measures 2 x 2 x 1.5 microns approx., with a resolution of 4x4x50 nm/pixel. The corresponding binary labels are provided in an in-out fashion, i.e. white for the pixels of segmented objects and black for the rest of pixels (which correspond mostly to membranes). This data set has many fine lines to segment, is heavily biased towards the membrane class, and has few training examples. The test set for the model does not have ground truth labels released to the public and instead one must submit their predictions for an accuracy measure. Because of this our result figures can only show our predictions and no ground truth label to compare next to it.

4.1 Models

Here we describe the models we have used for results so far. The overall basic shape of the model is shown in Fig. 4.1 and the individual components are

shown in Fig. 4.2. The overall shape was inspired by the U-Net model from Section 3.2.2. Instead of using any type of pooling layers we instead use a strided convolution to down sample feature maps. Also we differ from the U-Net model in the short connections. In U-Net the feature maps forwarded by the short connections are concatenated, but in our model they are summed. Along with the residual connections inside each ResBlock, these long residual connections make our model a fully residual model. The full residual model is shown to help information flow within and across levels in the network [QHJ16]. Also by not concatenating we reduce the number of feature maps in the expanding path of the model. For the multi-scale residual block we hope to increase the models spatial contiguity by giving it multiple fields of view around each local point.

The first half of the network is called the encoding path. Along this path the number of feature maps is doubled after each downsampling. After the center ResBlock begins the decoding path. In the decoding part, the number of feature maps is halved per level to maintain the network symmetry. The downsampling and upsampling are done with convolution layers. These convolutional layers serve as a connector to bridge the input feature maps and the residual block because the number of feature maps from the previous layer may differ from that of the residual block. The proposed network performs an end-to-end segmentation from the input data to the final prediction of the segmentation. We train the network with image set pairs corresponding to the image and its segmentation label, compare the output with manual segmentation, and use a loss function to back-propagate to adjust the weights of the network.

4.2 Training

For training we select an image size s and a batch size B. We then randomly sample a $s \times s$ image from a randomly selected training image. That sub-image is then modified by first adding a very small amount of Gaussian noise, randomly flipped left to right or top to bottom, and randomly rotated a small angle between -20 and 20 degrees. Sub-images are grabbed in this manner until there are B of them. The batch is then run through the model and the weights are updated. We have a set number of batches that we called

an epoch, but each epoch is a different set of batches since all augmentation happens online. After each epoch we observe the model's loss on the validation set and keep a record of it along with the corresponding weights. After the total number of epochs is done running we restore the weights to the point of lowest validation loss.

4.3 Experimental Setup

The proposed deep network is implemented using Keras open-source deep learning library [Cho15]. This library provides an easy-to-use high-level programming API written in Python, and Theano or TensorFlow can be chosen for a backend deep learning engine. Since the work was done on a Windows PC the Theano backend was chosen as TensorFlow did not support GPU processing on Windows. Training and deployment of the network is conducted on a PC equipped with an Intel i7 CPU with a 64 GB main memory and an NVIDIA GTX Geforce 980 Ti GPU.

4.4 Results

This section will show comparison results from many different models so we will break this section into subsections and begin each subsection with a description of the models being compared.

4.4.1 Basic Comparison

In this section we are establishing a baseline for each of the three main models. We are comparing: the U-Net [RFB15], our model shown in Fig. 4.1 with the basic residual block shown in Fig. 4.2a, and our model shown in Fig. 4.1 with the multi-scale residual block shown in Fig. 4.2b. This baseline is using the most common non-saturating activation function, the ReLU shown in Eq. 3.1. Also we did not apply Batch Normalization in any of these baseline models. The loss function used was a binary cross-entropy on each pixel of

the output. The loss plots are shown in Fig. 4.3 and a sample segmentation is shown in Fig. 4.4.

4.4.2 Applying Batch Normalization

Most state of the art models use Batch Normalization. We apply it directly before every non-linearity in the model. This is to keep the values, which can become large from the addition operations of the residual connections, stay mean centered. The loss plots are shown in Fig. 4.5 and a sample segmentation is shown in Fig. 4.6. Compare how smooth the losses are compared to the baseline model's loss in Fig. 4.3. The resulting segmentations also come out much more crisp, especially around the edges of the membranes compared to the basic model's segmentation in Fig. 4.4. Another interesting observation is that the multi-scale segmentation completely segmented the large membrane structure in the bottom right, while the other two models left the center open. This shows that the wider field of view helped the multi-scale model in this instance.

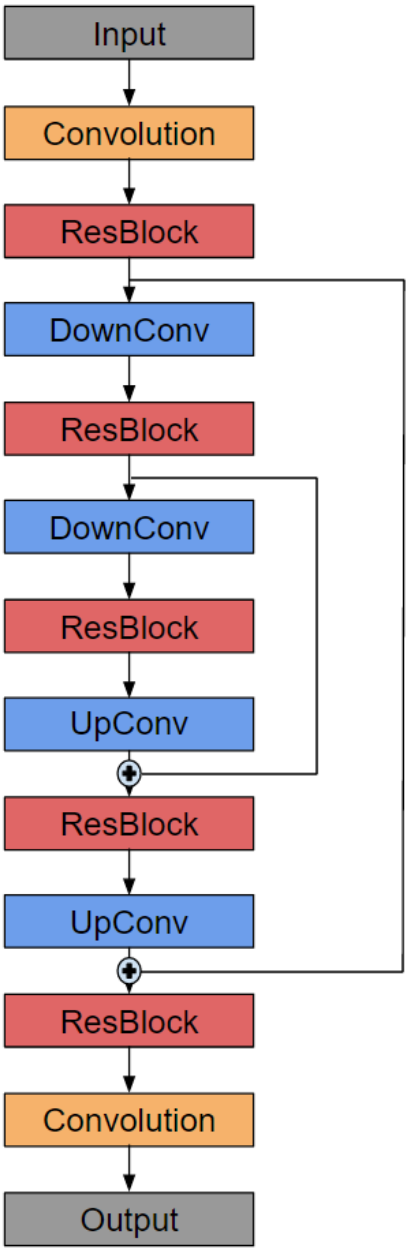
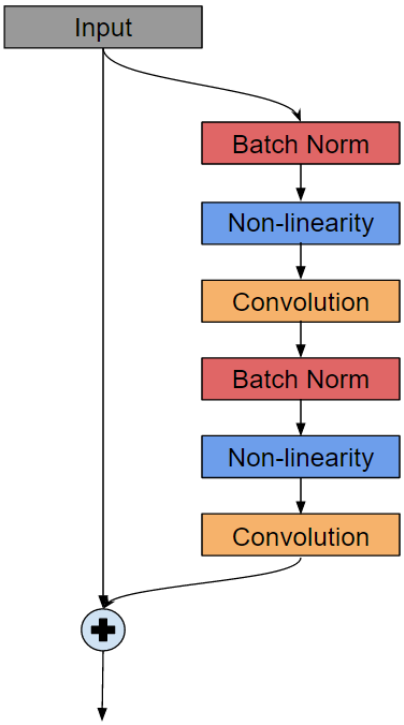
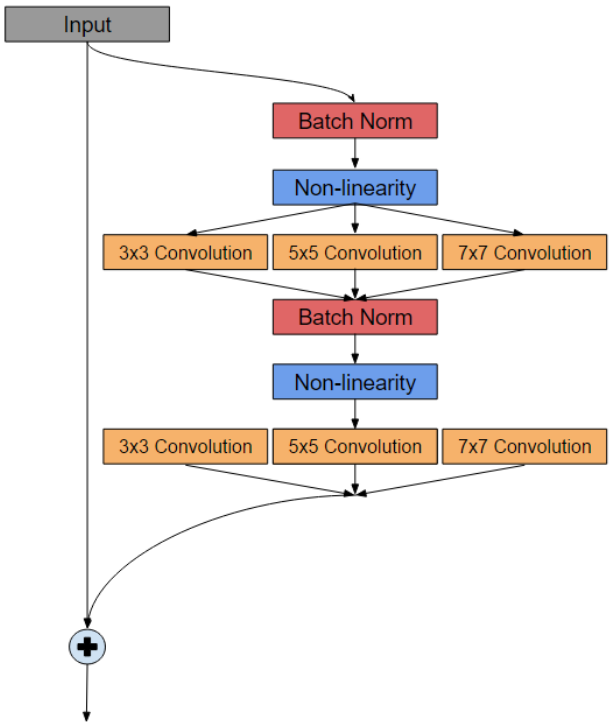


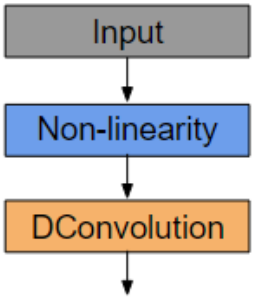
Figure 4.1: The model architecture. Feature maps from the down path get summed with feature maps of the corresponding size in the up path. The ResBlock types are shown in Figs. 4.2a & 4.2b and the DConvolution and UConvolution are shown in Fig. 4.2c and Fig. 4.2d respectively.



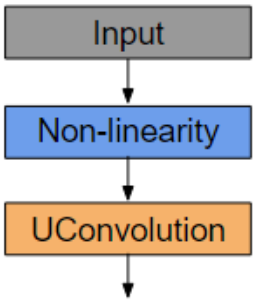
(a) A basic residual block.



(b) A multi-scale residual block.



(c) A down sampling block.



(d) An upsampling block.

Figure 4.2: Model components.

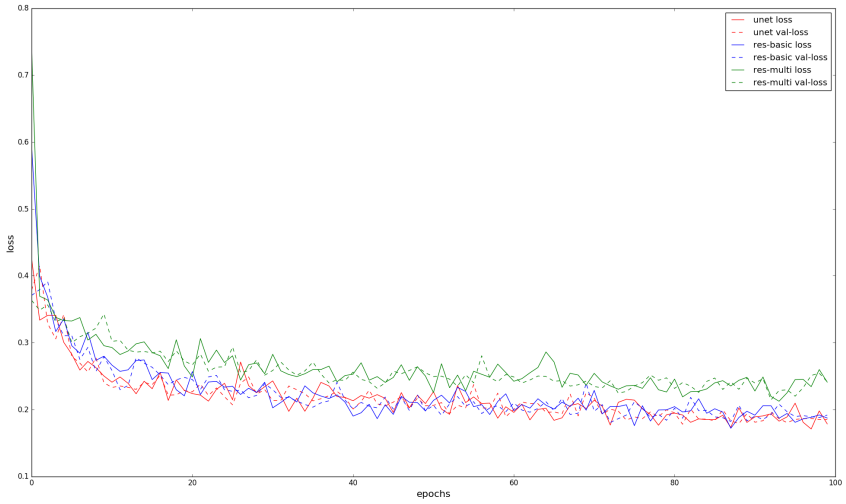


Figure 4.3: The loss and validation loss during training for the baseline models.

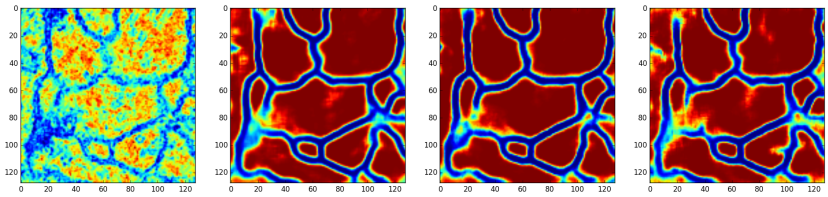


Figure 4.4: Sample segmentation on a test image for the baseline models. The test image is shown in its normalize stated, which is scaling the values between minus one and one. From left to right: test image, U-Net segmentation, our model, and our model with multi-scale convolution. The segmentations are shown in a red to blue color map where red is 0 and blue is 1.

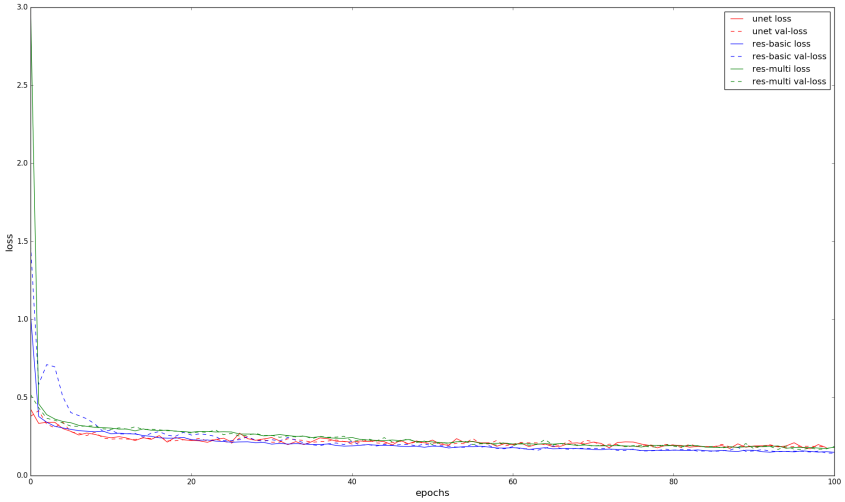


Figure 4.5: The loss and validation loss during training for the batchnorm models.

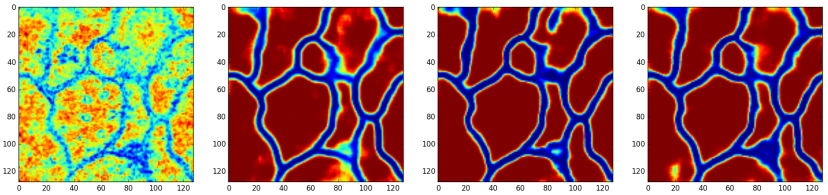


Figure 4.6: Sample segmentation on a test image for the batchnorm models. From left to right: test image, U-Net segmentation, our model, and our model with multi-scale convolution.

Chapter 5

Additional Research

In the above we have explored the effects of combining several new breakthroughs in deep neural networks for the task of semantic segmentation. Going forward we would like to explore using GAN models using the newly proposed Wasserstein and the least square loss functions to combat the training instabilities often seen in GANs. This would be the first to our knowledge implementation of GANs for segmentation with those loss functions. We would use the models we have already explored as the generators for these GANs and could publish a stand alone paper on these results.

We also want to look at the results of running a RNN refinement type model on the outputs of all previous models we have explored as well as the GANs outputs. While this may improve our GANs results, we do not believe it would be enough for its own paper.

We also intend to run all of models on more data sets. For the second data set we chose the Larval Zebrafish EM data, which is another neuronal segmentation challenge. However, this set has released the ground truth labels for the test data. And lastly we want to run on a more natural image challenge so we chose the COCO 2016 Detection Challenge segmentation challenge. The COCO data set has more than 200,000 images and 80 object categories.

Chapter 6

Timeline

The additional research to be carried out could be finished over the summer months. The code base is written in a very modular way allowing changes to any part of the models with a single variable change.

6.1 Refinement RNN

The code for the refinement RNN is already written, but no results have been generated yet. The biggest challenge with this model is that it requires a large amount of GPU memory and it takes a long time to train compared to the other models in this paper. To combat this we intend to deploy this model on AWS instances so it does not tie up my main machine from other work. I am very familiar with using AWS so the setup time will be very minimal.

6.2 GAN Model

The majority of the time will come from the implementation of the GAN model. For only this model we will need to add some new code that handles

training the discriminator and generator in simultaneous fashion. We also must implement the proposed loss functions since they are not build into any existing deep learning framework. We plan to attack this problem in steps. First we will implement a simple GAN to generate MNIST examples in a too be determined deep learning framework. Once this is working we will create the new loss functions and test them on the MNIST GAN. After we are confident in our ability to run the simultaneous training steps and that our custom loss functions are working we will implement the segmentation GAN.

Bibliography

[ACB17] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[App00] Apple. Performing convolution operations. https://developer.apple.com/library/content/documentation/Performance/Concep 2000.

[BF13] Jimmy Ba and Brendan Frey. Adaptive dropout for training deep neural networks. In *Advances in Neural Information Processing Systems*, pages 3084–3092, 2013.

[Cho15] Franois Chollet. keras. https://github.com/fchollet/keras, 2015.

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[CXLP13] Xueyun Chen, Shiming Xiang, Cheng-Lin Liu, and Chun-Hong Pan. Vehicle detection in satellite images by parallel deep convolutional neural networks. In *Pattern Recognition (ACPR), 2013 2nd IAPR Asian Conference on*, pages 181–185. IEEE, 2013.

[DEKLD16] Xianzhi Du, Mostafa El-Khamy, Jungwon Lee, and Larry S Davis. Fused dnn: A deep neural network fusion approach to fast and robust pedestrian detection. *arXiv preprint arXiv:1610.03466*, 2016.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[HDWF+17] Mohammad Havaei, Axel Davy, David Warde-Farley, Antoine Biard, Aaron Courville, Yoshua Bengio, Chris Pal, Pierre-Marc Jodoin, and Hugo Larochelle. Brain tumor segmentation with deep neural networks. *Medical image analysis*, 35:18–31, 2017.

[HN+88] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[HSK+12] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[HW68] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[HZRS15a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[HZRS15b] 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.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[Kar13] Andrej Karpathy. Convolutional neural networks (cnns / convnets). http://cs231n.github.io/convolutional-networks/#pool, 2013.

[KPU15] Philipp Kainz, Michael Pfeiffer, and Martin Urschler. Semantic segmentation of colon glands with deep convolutional neural networks and total variation segmentation. *arXiv preprint arXiv:1511.06919*, 2015.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LBD+90] Y LeCun, B Boser, JS Denker, D Henderson, RE Howard, W Hubbard, and LD Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems 2, NIPS 1989*, pages 396–404. Morgan Kaufmann Publishers, 1990.

[LCCV16] Pauline Luc, Camille Couprie, Soumith Chintala, and Jakob Verbeek. Semantic segmentation using adversarial networks. *arXiv preprint arXiv:1611.08408*, 2016.

[LCY13] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[LJH15] Wen Li, Fucang Jia, and Qingmao Hu. Automatic segmentation of liver tumor in ct images with deep convolutional neural networks. *Journal of Computer and Communications*, 3(11):146, 2015.

[LPAL15] Mark Lyksborg, Oula Puonti, Mikael Agn, and Rasmus Larsen. An ensemble of 2d convolutional neural networks for tumor segmentation. In *Scandinavian Conference on Image Analysis*, pages 201–211. Springer, 2015.

[LSD15] 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.

[MBLAGJ+07] Saturnino Maldonado-Bascon, Sergio Lafuente-Arroyo, Pedro Gil-Jimenez, Hilario Gomez-Moreno, and Francisco López-Ferreras. Road-sign detection and recognition based on support vector machines. *IEEE transactions on intelligent transportation systems*, 8(2):264–278, 2007.

[MHN13] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013.

[MTCA16] Emmanuel Maggiori, Yuliya Tarabalka, Guillaume Charpiat, and Pierre Alliez. Learning iterative processes with recurrent neural networks to correct satellite image classification maps. *arXiv preprint arXiv:1608.03440*, 2016.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[onsiEs12] ISBI Challenge: Segmentation of neuronal structures in EM stacks. Em stacks. http://brainiac2.mit.edu/isbi_challenge/home, 2012.

[QHJ16] Tran Minh Quan, David GC Hilderbrand, and Won-Ki Jeong. Fusionnet: A deep fully residual convolutional neural network for image segmentation in connectomics. *arXiv preprint arXiv:1612.05360*, 2016.

[RBS13] Lillian J Ratliff, Samuel A Burden, and S Shankar Sastry. Characterization and computation of local nash equilibria in continuous games. In *Communication, Control, and Computing (Allerton), 2013 51st Annual Allerton Conference on*, pages 917–924. IEEE, 2013.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 234–241. Springer, 2015.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[SGZ+16] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2226–2234, 2016.

[SLJ+15] 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.

[TGCd16] Ludovic Trottier, Philippe Giguère, and Brahim Chaib-draa. Parametric exponential linear unit for deep convolutional neural networks. *arXiv preprint arXiv:1605.09332*, 2016.

[TGJ+15] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler. Efficient object localization using convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 648–656, 2015.

[WM13] Sida I Wang and Christopher D Manning. Fast dropout training. In *ICML (2)*, pages 118–126, 2013.

[XM17] Haoran Xie Raymond Y.K. Lau Zhen Wang Xudong Mao, Qing Li. Least squares generative adversarial networks. *arXiv preprint arXiv:1611.04076*, 2017.

[ZRM+13] Matthew D Zeiler, M Ranzato, Rajat Monga, Min Mao, Kun Yang, Quoc Viet Le, Patrick Nguyen, Alan Senior, Vincent Vanhoucke, Jeffrey Dean, et al. On rectified linear units for speech processing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3517–3521. IEEE, 2013.

List of Figures

List of Tables