ON DEEP LEARNING AND NEURAL NETWORKS

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Mathematics

By

Samuel Raymond Lyche

2018

SIGNATURE PAGE

THESIS: ON DEEP LEARNING AND NEURAL NETWORKS

AUTHOR: Samuel Raymond Lyche

DATE SUBMITTED: Spring 2018

Department of Mathematics and Statistics

Dr. Adam King _____
Thesis Committee Chair
Mathematics & Statistics

Dr. Alan Krinik _____
Mathematics & Statistics

Dr. Hao Ji _____
Computer Science

ACKNOWLEDGMENTS

Thank you Jesus, Dr. Adam King, Dr. Alan Krinik, Dr. Hao Ji, Mom, Dad, Noah, Grandma Joy, and the rest of my family and friends. I am blessed to have had your support and could not have done this without you.

Thank you to the members of Dr. Krinik's research group who helped me prepare for my defense: Jeffrey Yeh, Ly Phey, John Kath, Uyen "Winwin" Nguyen, Yoseph Dawit, David Perez, and especially Chon In "Dave" Luk for testing me on the nitty gritty. I appreciate all y'alls help and support on this.

Thank you to Mark Dela for taking the time to read my paper and make it better. Your notes, questions, and comments truly improved my entire thesis.

Soli Deo Gloria.

ABSTRACT

This thesis gives an introductory overview of neural networks and deep learning. A new wave of computational power has brought about a fundamental change in the way that software is written. Only recently have machines become powerful enough to efficiently implement these algorithms. Due also to the massive increase in data storage capabilities and data production, artificial intelligence, of which neural networks form part of the foundation, will soon be affiliated with nearly every industry in the world. We give a high-level overview of the mathematical machinery necessary to implement neural networks, as well as the Python code to actually run the algorithms. Finally, we will show an example of how neural networks can be used to find cell nuclei in medical images as we compete in the 2018 Kaggle Data Science Bowl.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

The name "deep learning" comes from the architecture of the computer algorithms that underlie artificial intelligence. The term apparently originated in 1986 and was coined by Rina Dechter (Dechter, 1986). These algorithms are sometimes collectively referred to as neural networks, deep belief networks (Bengio et al., 2007), or artificial neural networks, to name a few. The flow of information from input to output can be represented by a graph with many layers that feed into one another in various ways. Neural networks can have few or many layers. When people talk about "deep learning," they are generally referring to network architectures with many layers, e.g., 10, 20, 50, 100, or more.

Figure 1.1 on page 2 displays the some of the ways that these graphs can be organized (van Veen, 2016). When layers of graphs are stacked together, this brings about the "deep" part of the name. In this thesis we will explore feedforward fully connected networks and convolutional networks.

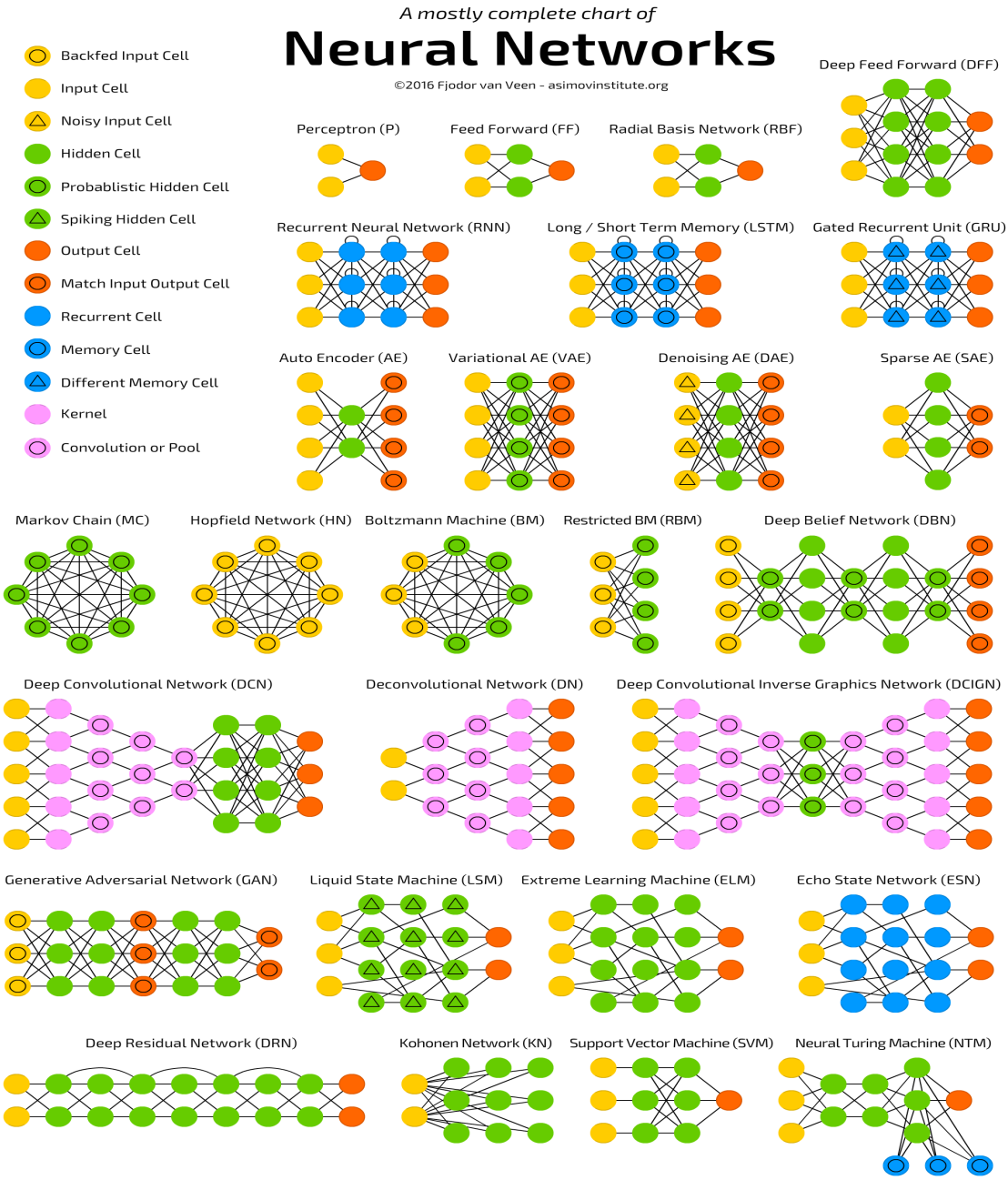


Figure 1.1: Architecture diagrams of many different neural networks (van Veen, 2016).

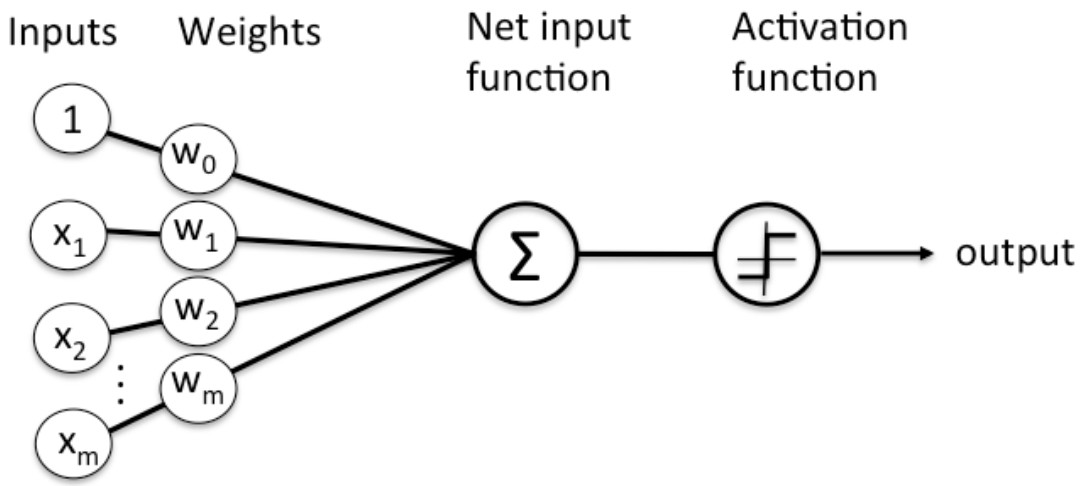


Figure 1.2: Rosenblatt's Perceptron (Raschka, 2015)

1.2 History

In the 1940s, scientists were trying to mathematically model how a brain makes decisions (S. McCulloch and Pitts, 1943). The math was supposed to model how a single neuron in the brain works; hence, the term "neural networks." In 1957, Frank Rosenblatt used the McCulloch-Pitts neuron (S. McCulloch and Pitts, 1943) to build one of the pivotal models in deep learning history, the perceptron (Rosenblatt, 1957). Figure 1.2 on page 3 shows a schematic for the perceptron (Raschka, 2015).

The perceptron was a big step in the right direction towards teaching computers to learn, as it could learn OR/AND/NOT functions. One major downfall, however, was that it could not learn the XOR function, since this function is not linearly separable. See Figure 1.3 on page 4 (Kurenkov, 2015a).

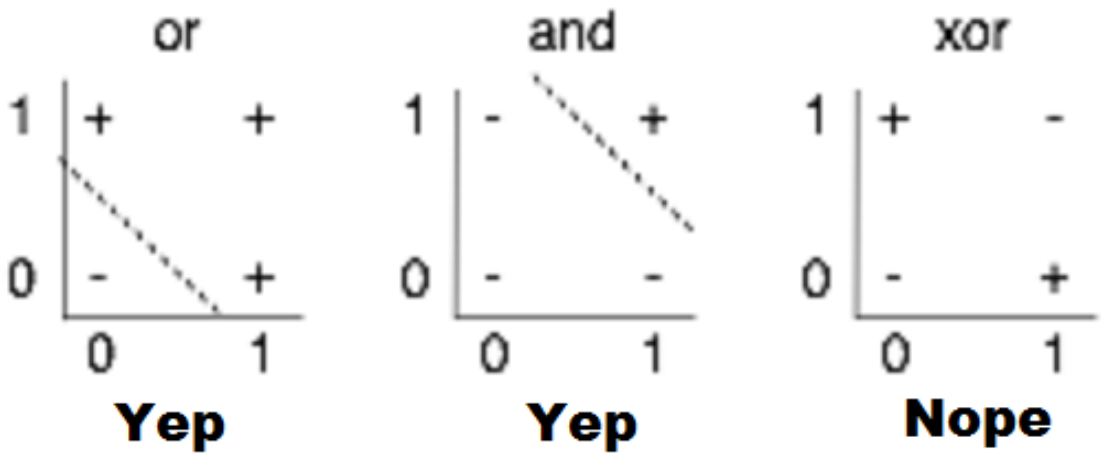


Figure 1.3: Examples of logical functions. Notice XOR function is not linearly separable. (Kurenkov, 2015a)

The Adaptive Linear Neuron algorithm (ADALINE) was another early precursor to modern neural networks (Widrow, 1960). This came out only a few years after the perceptron in 1960 and was published by Bernard Widrow with help from his doctoral student Tedd Hoff. It was a small, but useful, tweak to the algorithm through the use of a linear activation function that made ADALINE different than the perceptron. See Figure 1.4 on 5 (Raschka, 2015).

Training these models, or teaching them to learn useful things, was cumbersome and slow back then, though. The so called AI Winter of the 1970s was, in part, a result of difficult model training, limited capabilities of the algorithms to learn all logical functions, really slow computers, and really expensive computer memory. Also, Marvin Minsky and Seymour Papert's book "Perceptron" came out in 1969, which showed a rigorous analysis of the shortcomings of the perceptron algorithm (Minsky and Papert, 1969).

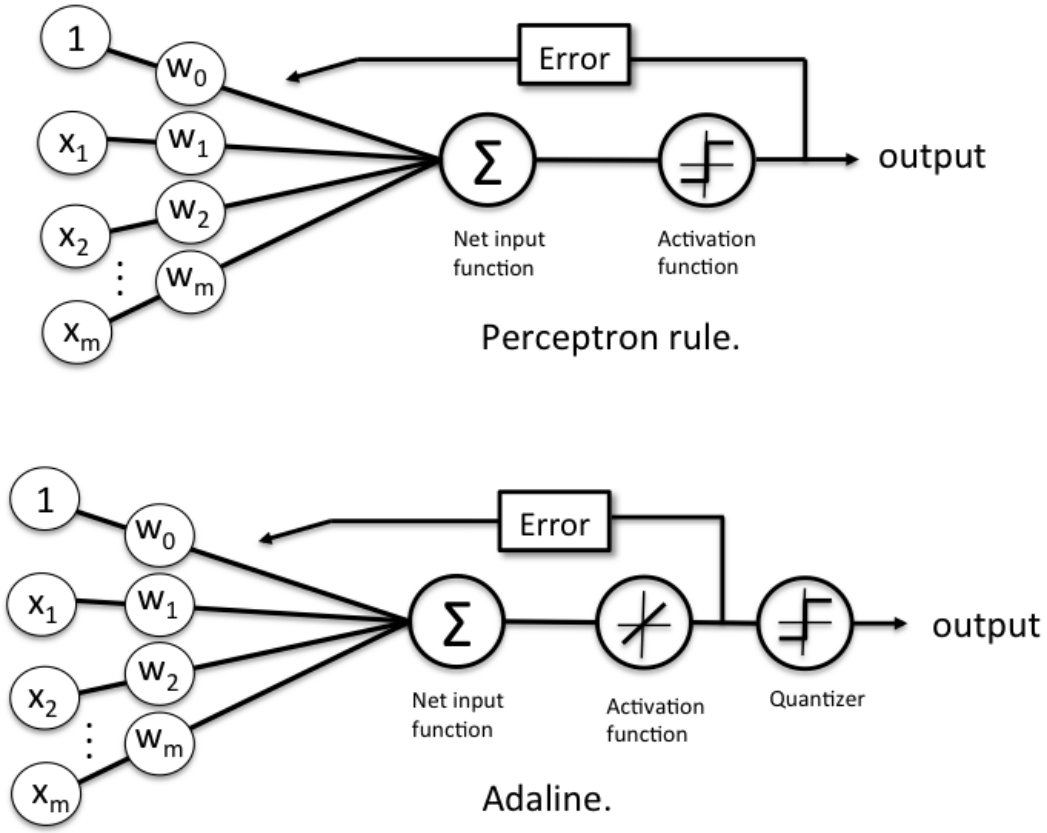


Figure 1.4: Perceptron vs ADALINE (Raschka, 2015)

This book is thought to have scared many away from investing in this field (Kurenkov, 2015a). Research did not totally stop, however.

According to Dettmers (2015), the earliest deep networks with multiple layers of nonlinear features are attributed to Alexey Grigorevich Ivakhnenko and Valentin Grigorevich Lapa. They did not use the now common technique of backpropagation to train their model but relied on layer-by-layer least squares fitting (Ivakhnenko and Lapa, 1967). Dettmers also attributes the first convolutional network to Kunihiko Fukushima (Fukushima, 1980).

The modern method for training deep learning models and neural networks is called backpropagation, which is nothing but the chain rule from calculus. The term itself refers to the idea that input data is fed through the model and then an error value is calculated (the forward phase). Then, this error is distributed back though the layers of the model via the chain rule in order to update the fitted model (the backward phase). This process is repeated until certain criteria of accuracy are met.

Apparently, the backpropagation algorithm was derived by a few researchers in the 1960s (Kurenkov, 2015a) (Dettmers, 2015), but it was not used with neural networks yet. The modern form of this computer algorithm is attributed to Seppo Linnainmaa from his masters thesis in 1970 (Linnainmaa, 1970), but he did not specifically refer to it as a way to train neural networks (Dettmers, 2015). Paul Werbos was the first person to specifically apply backpropagation to neural networks in his Ph.D. thesis in 1974 (Werbos, 1974). Supposedly due to the AI Winter that was going on, he did not publish the results until a few years later. The technique did not become widely used until 1986 when David Rumelhart, Geoffrey Hinton, and Ronald Williams published the classic paper, "Learning representations by back-propagating errors" (Rumelhart et al.,

1986). This mighty technique is still used to train neural networks and deep learning models today.

Now armed with backpropagation, researchers found themselves out of the AI Winter. In 1989, Kurt Hornik, Maxwell Stinchcombe, and Halbert White published the article "Multilayer feedforward networks are universal approximators" (Hornik et al., 1989). As stated in Kurenkov (2015b), their paper "mathematically proved that multiple layers allow neural nets to theoretically implement any function, and certainly XOR." Also in 1989, Yann LeCun and colleagues at AT&T Bell Labs implemented the backpropagation algorithm to accurately recognize handwritten digits in a dataset from the U.S. Postal Service (LeCun et al., 1989). This was a huge step in the history of neural networks since the mathematical theory proved useful in a practical, real-world application.

Another big area of research had to do with speech recognition. This led to many developments in sequence models, or Recurrent Neural Networks (RNN), as they are referred to now (Goodfellow et al., 2016). A type of recurrent network, called Long Short Term Memory (LSTM), became very popular due to its effectiveness (Hochreiter and Schmidhuber, 1997).

Up until now, the algorithms were working, and working well, but they were so very slow. Soon graphical processing units (GPUs), an innovation in the video game industry would forever change the landscape of neural networks and deep learning.

1.3 More Recently

Around 2009, Rajat Raina, Anand Madhavan, and Andrew Ng demonstrated that the training of neural networks could be done with GPUs rather than on CPUs which corresponded with massive speed increases (Raina et al., 2009). They showed that, using

their new technique, they could reduce training time for a model from several weeks to about one single day. This discovery has lead to the creation of gigantic neural network models, some with hundreds of hidden layers and billions of parameters. The graphics cards themselves have seen significant increases in capabilities, specifically those from Nvidia. While these processing cards were mostly used to render graphics for video games, companies like Google are even manufacturing processing units that are more specific to the types of calculations that are needed for deep learning. The Tensor Processing Unit (TPU) is now in its second generation and is reportedly much faster than even a cluster of high end GPUs (Miller, 2017).

With the capability to test bigger models faster, artificial intelligence research will continue to grow. There is a need for more people to know how to use these tools. Andrew Ng created the company DeepLearning.ai for this very purpose. His new series of courses on Coursera on deep learning are meant to give access to this body of knowledge to as many people as possible.

Chapter 2

Neural Networks

In this chapter we will go through the mathematics and theory of two types of neural networks: fully connected feedforward neural networks and convolutional neural networks.

The goal of training a neural network is to find an approximation f^* of some function f. With neural networks, though, we let the computer decide how to build this function f^*, rather than hand-engineering the function, by feeding data to the model and describing what a "good" result should look like in terms of a loss function J. The computer will then update the model through backpropagation to correct for any mistakes that it made.

For example, say we show the neural network a picture of the number five and ask it to guess what number it has been shown. The network makes its guess and then we inform it that the number was actually a five. If the network answered correctly, then the model does not need to be updated. If, on the other hand, it guessed any number other than five, then the model will go back and make some changes to itself, using the knowledge that it gained. Part of the deal with the loss function J is that it will usually provide some detail on the amount of incorrectness of a wrong answer, not just that the answer was wrong.

Say, if the model guessed three, then it would know that the answer was off by negative 2 and adjust accordingly. Now, the last sentence is not exactly how the loss function J works, but it is a somewhat intuitive way to first understand the mechanics.

2.1 Feedforward Neural Networks

The name feedforward is due to the way that information flows through this type of network. The information goes straight through the entire model until the loss function is calculated. Then, backpropagation is applied in order to find an update for the model. This update will be used as part of an optimization algorithm, such as gradient descent, in conjunction with our loss function J. This cycle will repeat itself until we are satisfied with the performance of the model. Sometimes, this will mean that our model is finally working very well (the ideal case). Othertimes, it could mean that we ran out of data and no further improvement can be had or we ran out of time to continue training the model. One other option, is that our model is not improving at all, at which point we will stop the model from training further and change what are known as model hyperparameters before trying to train the model once more.

The hyperparameters are parts of the model that we will need to control. Indeed, the point of deep learning is to let a large part of the model be learned from data. These are the parameters of the model that are trained. However, there are still buttons and knobs that we will have to manually push and turn in order for the computer to learn what it needs to.

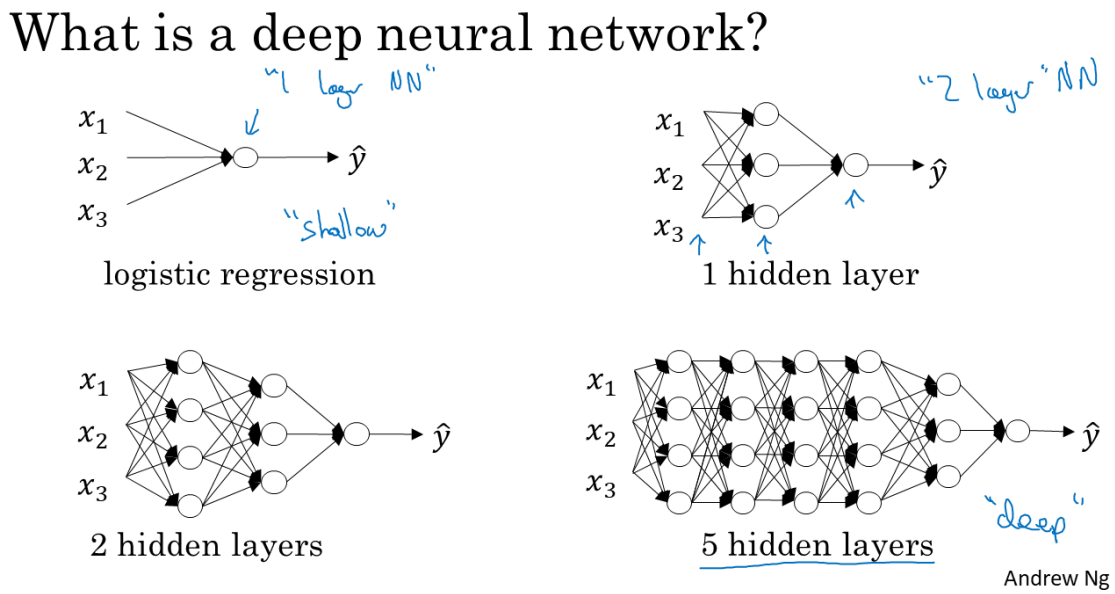


Figure 2.1: Comparison of Some Feedforward Architectures (Ng et al., 2017)

2.1.1 Fully Connected Neural Network

The modern fully connected feedforward neural network is nearly identical to the ADALINE network from Section 1.2, except we are going to use a nonlinear activation function. This will allow us to model nonlinear decision boundaries for complex data. A comparison of neural networks, plus the logistic regression model, which can be thought of as a simplified neural network, is shown in Figure 2.1 on page 11 (Ng et al., 2017).

It is important to note that while many of the early ideas of neural networks were inspired by neuroscience, top researchers in the field now agree that what we call neural networks should not be thought of as representations of how the brain works. Rather, "It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from

what we know about the brain, rather than as models of brain function." (Goodfellow et al., 2016)

As discussed before, we are trying to find some function approximation f^*. We can write $y = f^*(x;\theta)$ where x are the inputs to the network and θ are the parameters that the network will learn during training. The layers of the network can be thought of as compositions of functions. For example, maybe we have $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ where $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ are three functions, corresponding to the first, second and third layers of the network, respectively. Each model is associated with what is called a directed acyclic graph, which describes how the functions are composed, like in Figure 2.1. Each training example, input x is associated with a label y which is used to determine whether and how to tweak the values of θ. We will let θ, the set of parameters to learn, be weight matrices W and bias vectors b. So, now we have $y = f^*(x;\theta) = f^*(x;W,b)$.

We will use an affine transformation of the input, and use superscripts to denote the number of the function as $f^*(x;W,b) = f^{(1)}(x) = W^{(1)}x + b^{(1)}$. Then, we will apply an activation function $f^{(2)} = g$. Here, we will use the rectified linear unit (ReLU), since it is currently considered the best practice for this network architecture. It is defined by $f^{(2)}(x) = g(x) = max(0,x)$. See Figure 2.2 for some plots of common activation functions and Figure 2.3 for plots of their derivatives. So, now we have that the output from each layer looks like $f^{(2)}(f^{(1)}(x)) = max(0, W^{(1)}x + b^{(1)})$. Then, we repeat this for the number of layers that we would like in our network. Thus, the next step would be to let $f^{(3)}$ be the same type of affine transformation as $f^{(1)}$ resulting in $f^{(3)}(f^{(2)}(f^{(1)}(x))) = W^{(2)}max(0, W^{(1)}x + b^{(1)}) + b^{(2)}$.

Note, "The (b) term is often called the bias parameter of the affine transformation. This terminology derives from the point of view that the output of the transformation is biased toward being b in the absence of any input. This term is different from the idea

of a statistical bias, in which a statistical estimation algorithm's expected estimate of a quantity is not equal to the true quantity." (Goodfellow et al., 2016)

Another way to write this algorithm, which looks more like how it will look in the Python code that we see in the next chapter, is as follows:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \tag{2.1}$$

$$a^{[l]} = g^{[l]}(z^{[l]}) \tag{2.2}$$

$$= g^{[l]}(W^{[l]}a^{[l-1]} + b^{[l]}) \tag{2.3}$$

$$= max(0, W^{[l]}a^{[l-1]} + b^{[l]}) \tag{2.4}$$

for $l = 1, 2, ..., L$ where l is the layer number, L is the total number of layers, $W^{[l]}$ is the weight matrix for layer l, $b^{[l]}$ is the bias for layer l, and $g^{[l]}$ is the activation function for layer l. Also, $a^{[0]} = x$, i.e., the input.

2.2 Weight Initialization

Weight initialization turns out to be a big deal for neural networks. We cannot just initialize the weights to zero, because nothing will be learned. If $W = 0$ and $b = 0$, then $\hat{y} = Wx + b = 0$ and the gradients will all be zero and then nothing ever gets updated or learned. So, what is a good way to go about this?

Initializing the weights with small random numbers, e.g., Gaussian with mean zero and standard deviation 0.02 is an acceptable, but naïve, approach for small networks (e.g., less than three layers), but causes problems as networks get deeper. In particular, the problem is known as the exploding/vanishing gradients problem: the choice of the size of the standard deviation of the random initialization can cause all of the gradients

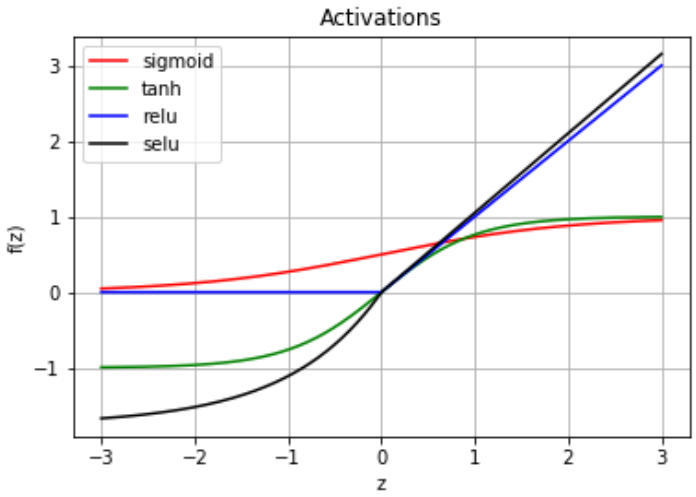


Figure 2.2: Plots of some activation functions

to become increasingly large or to quickly all become zero, which, in either case, halts the ability of the network to learn new information.

In Glorot and Bengio (2010), Xavier Glorot and Yoshua Bengio layed out what has become known as Xavier/Glorot initialization which can significantly help train deep networks by clever weight initialization. Kaiming He and colleagues improved upon this further in 2015 (He et al., 2015). These initializers force the initial random values to have a particular standard deviation which helps the networks continue learning, even with lots of deep layers.

2.3 Activation Functions

Activation functions are what create the nonlinearity of our neural networks. Without them, even with many layers, the matrix multiplication would be equivalent to a single weight matrix and, thus, a linear model.

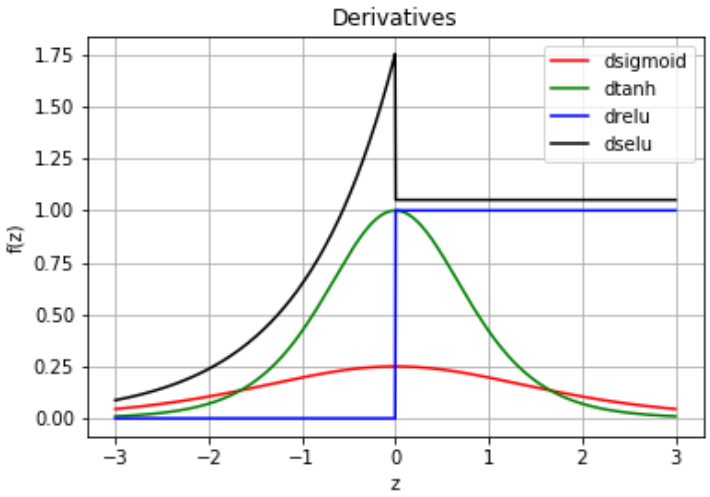


Figure 2.3: Plots of some derivatives of activation functions

Definitions of some common activation functions and their derivatives can be found in Table 2.1 below and Table 2.2 on page 17.

2.4 Loss Functions

In general, a loss (or cost) function will have the form

$$J(W) = \frac{1}{m}\sum_{i=1}^{m} L_i(f(x_i, W), y_i) \tag{2.5}$$

with derivative

$$\nabla_W J(W) = \frac{1}{m}\sum_{i=1}^{m} \nabla_W L_i(f(x_i, W), y_i) \tag{2.6}$$

which we will need to know later when we discuss optimization. To clarify, the L_is are the loss calculated for a particular instance, while the function J is the overall loss.

Table 2.1: Activations and Derivatives

Name	Equation	Derivative w.r.t. z
Sigmoid	$f(z) = \frac{1}{1+e^{-z}}$	$f'(z) = f(z)(1-f(z))$
tanh	$f(z) = \frac{2}{1+e^{-2z}} - 1$	$f'(z) = 1 - f(z)^2$
Rectified Linear Unit (ReLU) (Nair and Hinton, 2010)	$f(z) = \begin{cases} 0, & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases}$	$f'(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } 1 \geq 0 \end{cases}$
Leaky ReLU (Maas et al., 2013)	$f(z) = \begin{cases} 0.01z, & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases}$	$f'(z) = \begin{cases} 0.01, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0 \end{cases}$
Parametric ReLU (He et al., 2015)	$f(\alpha, z) = \begin{cases} \alpha z, & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases}$	$f'(\alpha, z) = \begin{cases} \alpha, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0 \end{cases}$
Exponential Linear Unit (ELU) (Clevert et al., 2015)	$f(\alpha, z) = \begin{cases} \alpha(e^z - 1), & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases}$	$f'(\alpha, z) = \begin{cases} \alpha e^z, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0 \end{cases}$

Table 2.2: Activations and Derivatives (cont)

Name	Equation	Derivative w.r.t. z
Scaled Exponential Linear Unit (selu) (Klambauer et al., 2017)	$f(z) = \lambda \begin{cases} \alpha(e^z - 1), & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases}$ with $\lambda \approx 1.0507$and $\alpha \approx 1.6733$.	$f'(\alpha, z) = \lambda \begin{cases} \alpha e^z, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0 \end{cases}$
Softmax	$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}$ for $i = 1, 2, ..., J$.	$\frac{\partial f_i(\vec{x})}{\partial x_j} = f_i(\vec{x})(\delta_{ij} - f_j(\vec{x}))$ where δ is the Kronecker delta.
Softplus	$f(z) = ln(1 + e^z)$	$f'(z) = \frac{1}{1+e^{-z}}$

This function should describe how well our model's predictions match up with our training labels. The choice of this function depends on the nature of the prediction task at hand. Here are some common loss functions and a description of when they may be useful.

The mean squared error (MSE), which is the same function as is used in least squares regression is a classic loss function is defined by

$$J(y,\hat{y}) = \frac{1}{2m}\sum_{i=1}^{m}(y-\hat{y})^2 \tag{2.7}$$

$$= \frac{1}{2m}\|y-\hat{y}\|_2^2 \tag{2.8}$$

or

$$J(W) = \frac{1}{2m}\sum_{i=1}^{m}(y-(WX+b))^2 \tag{2.9}$$

where y denotes the true value and $\hat{y}$ denotes the predicted value.

The MSE can be used when performing regression with a neural network, that is, when the task is to predict a real number rather than a particular class, e.g., class 0 or class 1.

The binary Cross Entropy with Logits loss function is closely related to the Kullback-Leibler divergence between the empirical distribution and the predicted distribution. This function is defined by

$$J(y,\hat{y}) = -\sum_{i=1}^{m}\hat{y}_i\log(y_i) + (1-\hat{y}_i)\log(1-y_i) \tag{2.10}$$

This loss function can be used when the task is to predict whether an input belongs to one of two classes.

The softmax classifier is a generalization of binary cross entropy with logits and can be used to make predictions when there are more than two classes. The function is defined by

$$J(z)_i = -\log\left(\left[\frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}\right]\right[\tag{2.11}$$

where C is the total number of classes in consideration.

2.5 Regularization

Regularization is a technique to choose the most parsimonious model out of all the choices for parameters that could have been fit. Typically, there are many weight matrices that would produce the same results in prediction, but we would like the model that is the simplest in the sense that we want all of our weights as small as possible. There are many forms of regularization, and this is an active field of research, but to start, we will discuss how to extend a loss function to add some regularization to our models.

If we extend Equation 2.5 by adding a regularization term, we get

$$J(W) = \frac{1}{m}\sum_{i=1}^{m} L_i(f(x_i, W), y_i) + \lambda R(W) \tag{2.12}$$

with derivative

$$\nabla_W J(W) = \frac{1}{m}\sum_{i=1}^{m} \nabla_W L_i(f(x_i, W), y_i) + \lambda \nabla_W R(W) \tag{2.13}$$

Where $R(W)$ is some function that is the regularization function and λ is a hyperparameter that determines the amount of regularization strength. Some common forms of

regularization are the L2 norm (aka weight decay), L1 norm, and elastic net, which is nothing but a combination of the L1 and L2 norms.

The L2 norm regularizer is defined by

$$R(W) = \sum_k \sum_l W_{k,l}^2 \tag{2.14}$$

L2 regularization also corresponds to MAP inference using a Gaussian prior on W in Bayesian statistics. (Li et al., 2017a) It is also known as "ridge regression" with penalty term λ.

The L1 norm regularizer is defined by

$$R(W) = \sum_k \sum_l |W_{k,l}| \tag{2.15}$$

The L1 norm regularizer is also known as the LASSO.

The elastic net regularizer is defined by

$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}| \tag{2.16}$$

The previous methods for regularization essentially penalize values in the weight matrix for being too large. This can also help speed up fitting of a model since the values in the weight matrix do not need to be changed in as large of a magnitude to find the optimal minimum for the cost function.

Other forms of regularization include dropout (Srivastava et al., 2014), data augmentation, and early stopping, and now there are even self-normalizing networks (Klambauer et al., 2017) which have regularization specially built into the architecture.

The way dropout works is by randomly "killing," i.e., setting the weight equal to zero, a select proportion of the nodes in a layer of the network during a training phase. By doing this, we are forcing the still active nodes to learn more than they normally

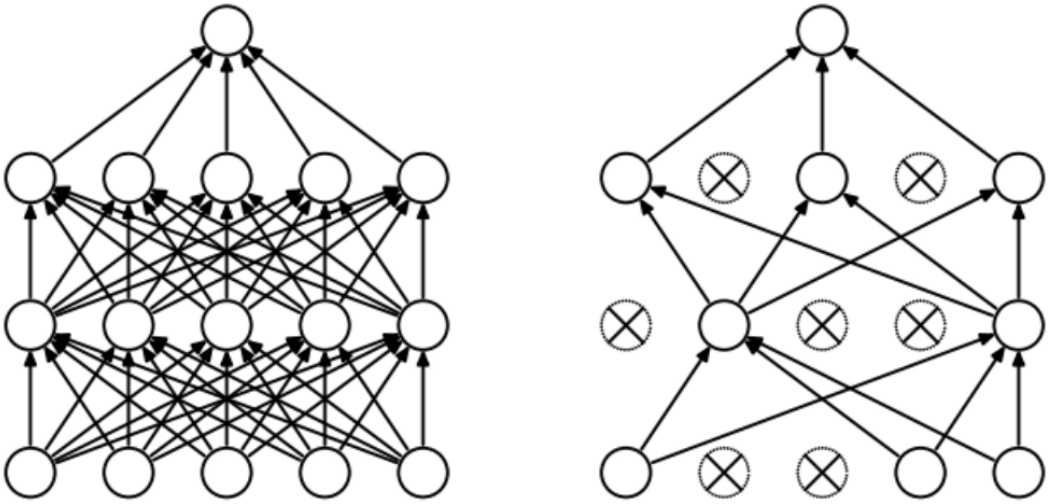


Figure 2.4: Visualization of Dropout (Srivastava et al., 2014)

would have to make a correct prediction. Since the nodes are chosen at random for each training pass, nodes must learn to work with whatever subset of other nodes are available at the time. According to the original authors, "this prevents units from co-adapting too much." (Srivastava et al., 2014) Further, instead of training and then combining the predictions of many different models, dropout has this same affect within just one single model. See Figure 2.4.

Data augmentation refers to taking the data that you already have and distorting it in some way that might happen in real life, e.g., in your test set or in live data in a product system, and using the original plus the distorted data to train your algorithm. For example, if you have images of people's faces, and you want to be able to predict who that person is, then maybe you would rotate each persons face 5, 10, 15, and 20 degrees clockwise and counterclockwise, and add those rotated photos to your dataset. This would allow your algorithm to be less prone to errors due to someone tilting their head,

perhaps. Sometimes changing the lighting of an image can help the model generalize better. Or maybe you have to predict handwritten digits, so you rotate some images, make some of the digits more wavy, or blurry, or skew and shift the images to be able to capture more styles of handwriting. Data augmentation can be done in many ways, and sometimes a few clever tweaks can really improve a learning algorithm.

Early stopping is an important technique because it will not only help prevent overfitting, it will minimize training time. As models train, if one keeps track of the training error and the validation error, there is often a point of greatest success, which, when crossed, actually results in a worse model. The training error will continue to decrease as the loss function is continuously minimized. However, the model will begin to fit just the data that is in the training set, and will not be adaptable to anything else with good accuracy. The goal of early stopping is to find the sweet spot when training error is low enough and validation error is also at its minimum, as shown in Figure 2.5. We can put a line in our code that will keep track of the training and validation errors, and if the validation error stops improving, we will stop training, and use the best model that we have trained up until that point.

2.6 Back Propagation

Back propagation (backprop) is the method of how we update the parameters of our models to learn useful things. It makes use of the chain rule from calculus all the way through the model's layers. In practice, deep learning libraries automatically take care of backprop for us, which is very nice. Automatic differentiation is built into the programs. So, we just have to build the graph of the network that we want, and the library (e.g.,

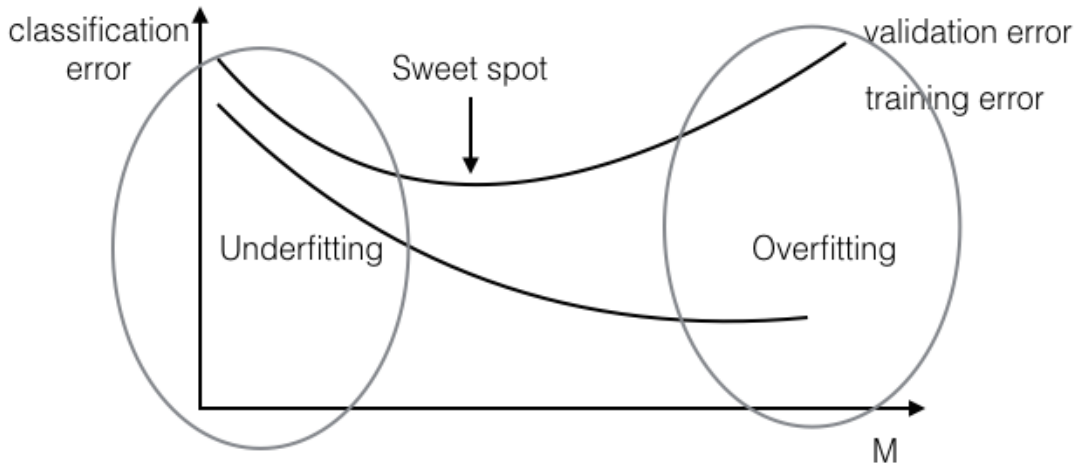


Figure 2.5: As training continues, training data is memorized and generalization strength deteriorates. We want to find the "sweet spot" where the validation error is minimized for the best case of generalization. (Weinberger, 2015)

TensorFlow or Keras or Torch, etc.) takes care of the backprop stage for us. More detailed explanations for how backprop works can be found in Li et al. (2017b).

2.7 Optimization

Once we have performed backpropagation, we have numerical values for the derivatives with respect to our model parameters. Here we will denote this as $\nabla_\theta J(\theta)$, where θ are the parameters from the model, e.g., W or b. Our goal is to minimize the loss function, $J(\theta)$. We can accomplish this by incrementally stepping down the surface of the function J by use of its derivative.

Batch gradient descent refers to when we use the entire training dataset to update the parameters. We will soon see that there are at least a couple other ways to update our parameters, but this is the classical method. A problem that is quickly encountered with this is running out of computer memory. As datasets and algorithms become larger and more complex, more memory is needed to do the computation. We will see that we can either

1. feed all the data through at one time, and repeat this process until our loss function fails to improve (batch gradient descent), or

2. we can update our parameters after each training example goes through, and after all the data has been fed into the model, known as an epoch, shuffle the data and repeat for more epochs until the loss function stops improving (stochastic gradient descent/SGD), or

3. we can feed minibatches into our model, e.g., 16, 32, or 64 examples at a time, and update the model after each minibatch, also shuffling and training for multiple epochs. (minibatch SGD)

2.7.1 Batch Gradient Descent

Let us dig a little more into each of these methods. The update equation for batch gradient descent, often just called gradient descent, is as follows

$$\theta = \theta - \lambda \cdot \nabla_\theta J(\theta) \tag{2.17}$$

where the $\nabla_\theta J(\theta)$ was computed during backpropagation, λ is called the learning rate, and θ is the parameter that you want to update. The learning rate, λ, is a model hyperparameter that controls the size of the step taken in descending the gradient. A hyperparameter is a changeable model parameter that is set before training, rather than learned by the model during training, like W and b. It is often the case that we will have to use some form of cross-validation in order to find the best value for the learning rate. If λ is too large, gradient descent will not converge to the minimum of the loss function, but can oscillate around near that point, or even diverge and cause the loss function to increase. If λ is too small, convergence to the minimum will take way too long. Finding the right value for the learning rate is an important task that can cause real frustration for deep learning practitioners.

2.7.2 Stochastic Gradient Descent

Rather than using the entire training set for each update, stochastic gradient descent performs an update for each training example $x^{(i)}$ and label $y^{(i)}$.

$$\theta = \theta - \lambda \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{2.18}$$

Figure 2.6: Without Momentum

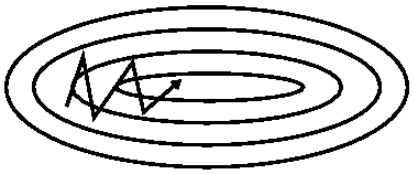


Figure 2.7: With Momentum

(Orr et al., 1999)

This makes the descent more noisy, so it is easier to escape local minima, but it can also speed up training. Another use for SGD is called online learning which is to update an algorithm as new data arrives, for example, as users interact with a website.

2.7.3 Gradient Descent

Minibatch gradient descent is a mix of batch and stochastic gradient descent. Rather than all or one, here we will perform an update for minibatches of n training examples.

$$\theta = \theta - \lambda \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.19}$$

Minibatch gradient descent can be faster than SGD because of parallelization over CPUs or on one or more GPUs.

2.7.4 Momentum

In each of the following sections, we will discuss ways to modify the update equation for gradient descent to improve the learning process. Momentum involves adding a term that will speed up the descent in directions with a steeper gradient and also dampen oscillations (Ruder, 2016). See Figure 2.7.4.

This momentum update is achieved by adding a fraction γ, commonly $\gamma = 0.9$ or higher, of the previous update iteration to the current gradient in order to update the parameters.

The definition of the update is as follows:

$$v_t = \gamma v_{t-1} + \lambda \nabla_\theta J(\theta) \tag{2.20}$$

$$\theta = \theta - v_t \tag{2.21}$$

This momentum is similar to the momentum a ball would pick up while rolling down a mountain especially if you think of a mountain as having peaks and valleys and different kinds of slopes and terrains all over, then the ball will pick up and lose speed in different directions as it continues rolling towards the bottom.

2.7.5 Nesterov Accelerated Gradient

Since we are trying to get to the bottom of the mountain and stay there, rather than allow our momentum to carry us back up the other side of the valley, what if we could somehow incorporate a bit of foresight into our parameter update? That is exactly what Yurii Nesterov figured out how to do (Nesterov, 1983). We know that $\theta - \gamma v_{t-1}$ gives an estimate for what our parameters will be, so we use that to calculate the gradient, rather than using only θ. The update will be as follows:

$$v_t = \gamma v_{t-1} + \lambda \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{2.22}$$

$$\theta = \theta - v_t \tag{2.23}$$

2.7.6 Adagrad

The Adagrad algorithm was introduced by Duchi et al. (2011) and aimed to do away with having to tune the learning rate. It was shown to be rather successful with sparse datasets because it give more weight to rarer instances. Adagrad was useful when Google's algorithm was able to recognize cats in YouTube videos (Dean et al., 2012). The update rule is as follows:

Let g_t be the objective function for the parameter θ at iteration t, that is,

$$g_t = \nabla_\theta J(\theta_t) \tag{2.24}$$

Now, rather than keep the learning rate constant, Adagrad modifies λ after each update iteration based on the past gradients that have been computed for θ:

$$\theta_{t+1} = \theta_t - \frac{\lambda}{\sqrt{G_t + \varepsilon}} * g_t, \tag{2.25}$$

where G_t is a square diagonal matrix with each element $G_{i,i}$ as the sum of the squares of the gradients w.r.t. θ up to update iteration t. "Duchi et al. (2011) give this matrix as an alternative to the full matrix containing the outer products of all previous gradients, as the computation of the matrix square root is infeasible even for a moderate number of parameters." (Ruder, 2016) The ε term is a smoothing term that helps avoid division by zero (usually set around 1e-8).

Adagrads main weakness is that the values in G_t continue growing with each iteration, so eventually the learning rate will be zero, and the model stops learning. How do we fix that?

2.7.7 Adadelta

Zeiler (2012) came up with a method to avoid tuning the learning rate and efficiently calculate a decaying average of the past gradients. This gets rid of the monotonically decreasing learning rate from Adagrad. The running average, $E[g_t^2]$ at update iteration t looks like this:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2, \tag{2.26}$$

where γ is set to something similar to momentum, i.e., ≈ 0.9. Now, replace the diagonal matrix G_t with the decaying average over past squared gradients:

$$\Delta\theta_t = -\frac{\lambda}{\sqrt{E[g^2]_t + \varepsilon}} * g_t \tag{2.27}$$

$$= -\frac{\lambda}{RMS[g]_t} * g_t, \tag{2.28}$$

where $RMS[g]_t$ means the root mean squared error and is just shorthand. Next, if we just use Equation 2.28, the units will not match the parameter that we are updating. So, we need another exponentially decaying average, not of the squared gradients, but of the squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2, \tag{2.29}$$

with shorthand

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \varepsilon}. \tag{2.30}$$

$RMS[\Delta\theta]_t$ is not actually known, however. So, it is approximated using the parameter updates up until the previous update iteration. Thus, the final update rule is

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} * g_t \tag{2.31}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t \tag{2.32}$$

As can be seen from the equations above, there is no learning rate λ to fuss about with this algorithm.

2.7.8 RMSprop

This optimization is famously unpublished and is cited in Geoffrey Hinton's lecture notes from his Coursera class (Hinton et al., 2014). It is similar to Adadelta and the update rule is as follows:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2, \tag{2.33}$$

$$\theta_{t+1} = \theta_t - \frac{\lambda}{\sqrt{E[g^2]_t + \varepsilon}} * g_t. \tag{2.34}$$

2.7.9 Adam

Adaptive Moment Estimation (Adam) by Kingma and Ba (2014) is another algorithm that computes adaptive learning rates for each parameter. Adam keeps track of an exponentially decaying average of past squared gradients v_t, like Adadelta and RMSprop, but it also keeps track of an exponentially decaying average of past gradients m_t, similar to momentum.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t \tag{2.35}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \tag{2.36}$$

These are estimates of the first two moments of the gradients, that is, m_t is an estimate of the gradient's mean, and v_t is an estimate of the gradient's (uncentered) variance. Since m_t and v_t are initialized as zeros, the early steps of the algorithm are biased toward zero, so we correct as follows:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \tag{2.37}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}. \tag{2.38}$$

Then, we use these values in the update rule:

$$\theta_{t+1} = \theta_t - \frac{\lambda}{\sqrt{\hat{v}_t}+\varepsilon}\hat{m}_t \tag{2.39}$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ε. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms (Ruder, 2016).

2.7.10 AdaMax

AdaMax is a generalization of Adam that uses the L_∞ norm rather than the L2 norm of the past gradients. With Adam, the v_t term uses the L2 norm like so:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)|g_t|^2 \tag{2.40}$$

We can generalize to use the LP norm and parameterize β_2 as β_2^p, like so:

$$v_t = \beta_2^p v_{t-1} + (1-\beta_2^p)|g_t|^p \tag{2.41}$$

$$= (1-\beta_2^p)\sum_{i=1}^{t}\beta_2^{p(t-i)}|g_i|^p \tag{2.42}$$

This can be very numerically unstable for large p (Ruder, 2016). But the authors, Kingma and Ba, show that we can use the L_∞ norm, and everything is stable, like the following. We will use u_t to avoid confusion with Adam.

$$u_t = \beta_2^\infty v_{t-1} + (1-\beta_2^\infty)|g_t|^\infty \tag{2.43}$$

$$= \max(\beta_2 u_{t-1}, |g_t|) \tag{2.44}$$

Finally, just replace $\sqrt{\hat{v}_t} + \varepsilon$ with u_t to get the Adamax update rule:

$$\theta_{t+1} = \theta_t - \frac{\lambda}{u_t}\hat{m}_t \tag{2.45}$$

See Kingma and Ba (2014) for more details.

2.7.11 Nadam

The Nadam update rule incorporates NAG into Adam (Dozat, 2015). The update rule becomes as follows:

$$\theta_{t+1} = \theta_t - \frac{\lambda}{\sqrt{\hat{v}_t} + \varepsilon}\left(\left[\beta_1 \hat{m}_t + \frac{(1-\beta_1)g_t}{(1-\beta_1^t)}\right)\right[\tag{2.46}$$

2.7.12 Practical Usage

Practically speaking, TensorFlow (Abadi et al., 2015), Keras (Chollet et al., 2015) or some other deep learning library will be used to implement the entire neural network architecture . Researchers have worked very hard to implement all of the algorithms and necessary tools in the most efficient manner in which a machine can run them. Modern neural networks require training of millions or billions of parameters that are best trained

using GPUs, or, now, even specialized tensor processing units (TPUs). Deep learning experts, like Andrew Ng, suggest using the deep learning libraries rather than attempting to implement the code on your own.

2.8 Further Improve Training Speed

A few other techniques for improving a neural network include shuffling, batchnormalization (Ioffe and Szegedy, 2015), and gradient noise (Neelakantan et al., 2015). Shuffling the dataset before each training epoch helps the network avoid using the same data in the same order. This helps improve generalization of the model. Batchnormalization means that you force the training batches to have the same mean and variance, which makes training more robust and can improve training speed. Adding a bit of random noise to the gradient can help training networks by giving extra little nudges to where the gradient direction goes. Much of this section on optimization used help from Sebastian Ruder's compilation of these results (Ruder, 2016).

2.9 Convolutional Neural Network

Technically, a convolutional neural network is a feedforward neural network because there are no back loops of data flow, but they are still different enough for their own section. Convolutional neural networks have proven extremely useful in computer image analysis. For example, they can identify handwritten numbers like you might find on mail or checks (Lecun et al., 1998). They can perform object detection like finding where pedestrians and cars are in a picture. They can even figure out whether a picture is of a dog. These capabilities are due to the way that the algorithm allows future layers to share spatial information.

2.9.1 Convolutional Architecture

A simple, modern convolutional neural network has a structure as follows: conv, pool, activation, (repeat), fully connected, (repeat), and loss function. A convolution uses a full $H \times W \times C$ image as input, where H is the height (in pixels), W is the width, and C is the number of color channels, rather than a flattened vector. See Figure 2.8 for a general convolutional network architecture.

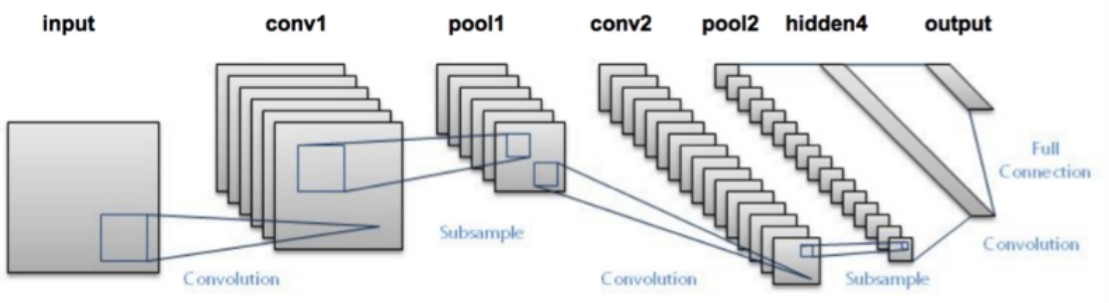


Figure 2.8: Basic Convolutional Network Architecture (Gupta, 2017)

We are going to analyze an image one small piece at a time by only looking at small window (called the convolutional kernel, or just kernel), say 3×3 or 5×5 pixels in size. Starting in the top left corner of the image, we will multiply this slice of the image by a weight matrix, perform a pooling operation (explained on page 36), and then apply our activation function. We repeat this for each slice in the entire image by sliding the window to the right by a number of pixels, called the stride length, until we have reached the right end of the image. Then, we move the window down and and back to the left side of the image, and repeat the sliding process until we have reached the end of the image. When programming these, we can calculate all of the slice times weight matrix operations first, then apply pooling to all of these, and then finally apply the activation functions to those results. As will be seen in the next chapter, a deep learning library will take care of all of these operations for us. We will just need to input the order in which we would like them performed by building the computation graph.

Some important concepts that go along with convolutional neural networks (conv nets) are padding, strides, and subsampling (a.k.a. pooling layers) which we will discuss in a bit more detail in the following paragraphs. For conv nets, the activation functions have not changed, we can still use regularization on the same kinds of loss functions, dropout still works, weight initialization is the same, backpropagation works the same way, and much of what we discussed for fully connected networks still applies.

Padding is an operation that actually increases the dimensions of the input matrices. We add a ring of zeros to the actual image, and in so doing, we help the algorithm to better compute information by the outer edges of the input matrix.

The stride of convolutional kernels refers to how many pixels the kernel is slid over at a time. A stride of one means that we shift the kernel over to the right by one pixel, a stride of two means slide it over two pixels, and so forth. A smaller stride allows more

information to be fed to the next layers of the network, but it also means the algorithm will take longer to run.

Pooling is an operation that is often used with conv nets. It allows us to shrink the number of parameters of the model, which speeds up training and prediction, and it also makes the network "think" about the input matrices in chunks, rather than by individual pixels. Two common forms of pooling include max pooling and average pooling. With max pooling, we pass a kernel over a matrix, and the output is simply the maximum value of the elements of the kernel. Average pooling's output is the average of the values in the kernel.

Deep learning libraries have all of these functions already built in for us. We must choose kernel sizes, stride lengths, number of layers, and certain configurations, but the actual inner workings of the operations are conveniently hidden away from us. We will see more of how to implement all of these networks and layers in Chapters 3 and 4.

2.10 Object Detection and Image Segmentation

This is the goal for the 2018 Kaggle Data Science Bowl. In Chapter 4, we will attempt to use neural networks to compete in this challenge. Here, we will review some of the classical methods for image segmentation on a computer. We will be using an RGB (red/green/blue) colorspace with pixel values in the range [0, 255] in this paper. A pixel with value 0 will be black, while a pixel with value 255 will be full color intensity.

2.10.1 Histograms

Suppose that we want to know the distribution of intensities (pixel values) for an image. We can accomplish this by use of a histogram. This is one way to try to distinguish

between different images, or types of images. Plotting a histogram is a great way to understand the distribution of an image.

Histograms of images help you get a general understanding of the contrast, brightness, and intensity distribution. They also can be used as feature vectors associated with images to feed into neural networks. Further, histogram equalization is a technique to increase the contrast in an image.

2.10.2 Smoothing and Blurring

Sometimes making an image a little more blurry can smooth out edges of objects in the image. If we apply an odd-dimensional (e.g., 3x3 or 5x5) sliding window (a.k.a. convolutional kernel a.k.a. kernel) over the image and transform the center pixel of the kernel to be the average or perhaps median value of the surrounding pixels, this will accomplish our blurring task. Strangely enough, "many image processing and computer vision functions, such as thresholding and edge detection, perform better if the image is first smoothed or blurred." (Rosebrock, 2016)

Other kinds of blurring including Gaussian blurring, where the kernel's center pixel is transformed to be a weighted average of the surrounding pixels, based on how close to the center pixel they are, and bilateral filtering, which blurs the image and helps to retain edges in the image.

2.10.3 Thresholding

"Thresholding is the binarization of an image" (Rosebrock, 2016). We seek to convert all of the pixels of an image from some integer between 0 and 255 to either 0 or 1. A simple example would be to set all pixels less than some threshold value p to 0 and set the rest to 1. A more complex method is adaptive thresholding. This technique

"considers small neighborhoods of pixels and then finds an optimal threshold value T for each neighborhood. This method allows us to handle cases where there may be dramatic ranges of pixel intensities and the optimal value of T may change for different parts of the image" (Rosebrock, 2016).

There exist more methods for thresholding as well, such as Otsu's method, Riddler-Calvard, and Yen's method, to name a few.

2.10.4 Gradients and Edge Detection

"Edge detection embodies mathematical methods to find points in an image where the brightness of pixel intensities changes distinctly" (Rosebrock, 2016). There are many methods that perform this task, including the Laplacian method, Sobel operator, Canny edge detector, and more. There are libraries that implement these methods, and it is best to make use of them when performing these tasks in your own projects. OpenCV and scikit-image are two of these libraries that are commonly used.

2.10.5 Masking

This goes along with object detection and will be used in Chapter 4, but is not completely specific to image segmentation. Masking is the term that describes choosing certain portions of an image to view, or process, by creating a boolean matrix the same shape as your image where the 0's (or False's) are the portion of the image you want to ignore and the 1's (or True's) are the part of the image that you want to work with.

Chapter 3

Deep Learning with Tensorflow and Keras

We are going to be using the MNIST dataset from Yann LeCun. You can get the data from LeCun's website in a somewhat awkward format (LeCun, 1998), or you can download a nice csv version from Joseph Redmon's website (Redmon, 2013). The MNIST dataset (csv version) includes two csv files, one with 60,000 training images and 10,000 test images. Each image is a handwritten digit between 0 and 9. According to LeCun's website, "It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting" (LeCun, 1998). Another benefit of this dataset for beginning practice with neural networks is that it can fit on and be processed in a reasonable amount of time on a laptop. Our goal for this chapter will be to implement neural networks in Python that are able to predict the numbers on the images that we show them.

3.1 Fully Connected Network

First, we need to import some libraries, classes, and functions that we need.

```
import tensorflow as tf
import pandas as pd
import numpy as np

from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras.utils import to_categorical
```

Listing 3.1: Load libraries

TensorFlow is a deep learning library from Google (Abadi et al., 2015) that can be used to directly build neural networks, but we will use it somewhat indirectly as we build our models using the Keras library (Chollet et al., 2015). Keras utilizes TensorFlow code to run, but we will actually be using Keras objects and functions for the most part. Keras was built in order to allow us to more easily create models and get them training faster. Pandas is a package that is useful for data I/O, cleaning, manipulation, and many other things, but we will just use it to read in our data (McKinney, 2010). NumPy is am efficient numerical computation library that we will utilize to store our data into arrays (van der Walt et al., 2011). Pandas and NumPy are part of the larger SciPy ecosystem, open source Python libraries for scientific computing (Jones et al., 2001).

After we have read in our data and inspected it to make sure that everything is in order to be fed into an algorithm, we will center and standardize our data. Note that we center and standardize both our training and testing dataset with the values computed from the training set.

```
df_train = np.array(pd.read_csv('../data/mnist_train.csv', header=None
    ))
df_test = np.array(pd.read_csv('../data/mnist_test.csv', header=None))

X_train = df_train[:,1:]
y_train = df_train[:,0]
X_test = df_test[:,1:]
y_test = df_test[:,0]

mean_vals = np.mean(X_train, axis=0)
std_val = np.std(X_train)

# THIS IS SUPER IMPORTANT!!!
X_train_centered = (X_train - mean_vals) / std_val
X_test_centered = (X_test - mean_vals) / std_val
```

Listing 3.2: Center and Standardize Data

Next, we will create some convenient variables, set some random seeds for reproducibility, and convert our labels into one-hot vectors. One-hot vectors, for our case, will map each label, i.e., an integer from 0-9, to the corresponding index of a vector of length 10. For example, if the label is 2, then the corresponding one-hot vector is `[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`. If the label is 9, then the corresponding one-hot vector is `[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]`.

```
# create some helpful variables
n_features = X_train_centered.shape[1]
random_seed = 7
np.random.seed(random_seed)
tf.set_random_seed(random_seed)

# create onehot vectors for y_train
y_train_onehot = to_categorical(y_train)
n_classes = y_train_onehot.shape[1]
```

Listing 3.3: Convenient Addition

Now, we are ready to build our computation graph using Keras. This model will be a fully connected (FC) network with two layers. The hidden layer will have 50 units, have He weight initialization, zeros for bias initialization, and ReLU activation functions. The second layer will have 10 units (one for each possible class), have He (named after Kaiming He) weight initialization, zeros for bias initialization, and a softmax activation function. We will use an Adam optimizer with learning rate 0.001 and categorical cross entropy loss function.

```
# instantiate Keras Sequential model
model = Sequential()

# add a hidden layer: 50 units, ReLU activation
model.add(
        Dense(
        units=50,
        input_dim=n_features,
        kernel_initializer='he_uniform',
        bias_initializer='zeros',
        activation='relu'))
```

```
12
13
14 # add layer: 10 units, softmax function
15 model.add(
16         Dense(
17         units=n_classes,
18         input_dim=50,
19         kernel_initializer='he_uniform',
20         bias_initializer='zeros',
21         activation='softmax'))
22
23 # create stochastic gradient descent optimizer object
24 optimizer = Adam()
25
26 # compile the graph for the model
27 model.compile(optimizer=optimizer, loss='categorical_crossentropy')
```

Listing 3.4: Build the Computation Graph

Compiling the model sets up everything to run on the GPU. It is a necessary step to take but one for which the details are beyond the scope of this paper. After we have set up our computation graph for our model, it is time to run some training epochs.

```
1 # train the model, holding out 10% for validation
2 history = model.fit(X_train_centered, y_train_onehot,
3                     batch_size=256, epochs=15,
4                     verbose=1,
5                     validation_split=0.1)
```

Listing 3.5: Train the Model

This will use our centered and standardized training set as input. Each epoch, 10% of the training data will be used as a validation set. The validation set is chosen randomly

for each epoch, and the entire training set is shuffled after each epoch. The minibatch size is set to 256 and we will train the model for 15 epochs. The boolean switch "verbose" allows us to see some progress printed to the console as the model trains. The output will look something like the following:

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/15
54000/54000 [==============================]
- 2s 44us/step - loss: 0.4322 - val_loss: 0.1709
...
Epoch 15/15
54000/54000 [==============================]
- 2s 30us/step - loss: 0.0241 - val_loss: 0.0987
```

Finally, it is time to evaluate how well our model works on the training and the held out test set.

```
y_train_pred = model.predict_classes(X_train_centered, verbose=0)
correct_preds = np.sum(y_train == y_train_pred, axis=0)
train_acc = correct_preds / y_train.shape[0]
print('First 3 predictions: ', y_train_pred[:3])
print('Training Accuracy: %.2f%%' % (train_acc * 100))

y_test_pred = model.predict_classes(X_test_centered, verbose=0)
correct_preds = np.sum(y_test == y_test_pred, axis=0)
test_acc = correct_preds / y_test.shape[0]
print('Test accuracy: %.2f%%' % (test_acc * 100))
```

Listing 3.6: Evaluating Our Model

This will output something like the following:

```
First 3 predictions:  [5 0 4]
Training Accuracy: 99.39%
Test accuracy: 97.12%
```

We have built a model that can correctly predict the hand written digits with just over 97% accuracy. This is alright, but we can do better. Let us see how we can improve this by use of a convolutional neural network and some regularization.

3.2 Convolutional Neural Network

To build our convolutional neural network, we will need a few different functions imported, so let us start with those.

```
1  import tensorflow as tf
2  import pandas as pd
3  import numpy as np
4
5  from keras.models import Sequential, load_model
6  from keras.layers import Dense, Dropout, Flatten
7  from keras.layers import Conv2D, MaxPooling2D
8  from keras.optimizers import *
9  from keras.callbacks import EarlyStopping, ModelCheckpoint
10 from keras.utils import to_categorical
11
12
13 from skimage.io import imshow
14
15 import matplotlib.pyplot as plt
```

Listing 3.7: Load Libraries and Functions

We have some of the same libraries as before with our FC network, but we have also included some functions that we need to build our convolutional model graph and add some regularization. Functions Conv2D and MaxPooling2D will be used to create the convolutional layers and implement subsampling. We import all of the optimizer functions by use of the $*$ character in "from keras.optimizers import $*$". EarlyStopping will allow us, in conjunction with ModelCheckpoint, to save our best model as we train, and stop training early when our validation loss fails to improve. There will also be some patience associated with the early stopping, meaning that we will allow the model to keep training even after the validation loss has not improved for a certain number of epochs. Sometimes the model will not improve for a few epochs, and then it will begin to improve again, so we give it some patience. The load_model function will be used to reload our best model that we saved with ModelCheckpoint in order to make predictions.

The scikit-image library will be used to take a look at some input images (van der Walt et al., 2014). This library has many uses for computer vision and image processing that we will use more in chapter 4. Matplotlib is a graphing library that we will use to display the images (Hunter, 2007). These libraries are part of the SciPy ecosystem.

Now, we read in the data just as in the previous section. However, we must resize each input to be a 28x28 pixel image, rather than a 1x784 pixel vector. We will use NumPy's reshape method to accomplish this, as follows:

```
1 # need to make them numpy arrays rather than pandas dataframes to use
    with tensorflow
2 df_train = np.array(pd.read_csv('../data/mnist_train.csv', header=None
    )).astype(float)
3 df_test = np.array(pd.read_csv('../data/mnist_test.csv', header=None))
    .astype(float)
4
```

```
X_train = df_train[:,1:]
y_train = df_train[:,0]
X_test = df_test[:,1:]
y_test = df_test[:,0]

# reshape the datasets to be 28x28 pixel images
X_train_img = X_train.reshape((len(X_train), 28, 28, 1))
X_test_img = X_test.reshape((len(X_test), 28, 28, 1))
```

Listing 3.8: Reshape Images

The extra dimension on the end is necessary to work with Keras, which expects images to be in the form `(IMG_HEIGHT, IMG_WIDTH, NUM_CHANNELS)`. Our images are grayscale, so there is only one color channel. If our images were in RGB format, then the shape would be `(IMG_HEIGHT, IMG_WIDTH, 3)`.

We can take a look at some randomly selected entries of our reshaped dataset to make sure our transformation worked properly.

```
ix = np.random.randint(0,len(X_train)-1)
imshow(np.squeeze(X_train_img[ix]))
plt.show()
```

Listing 3.9: Sanity Check

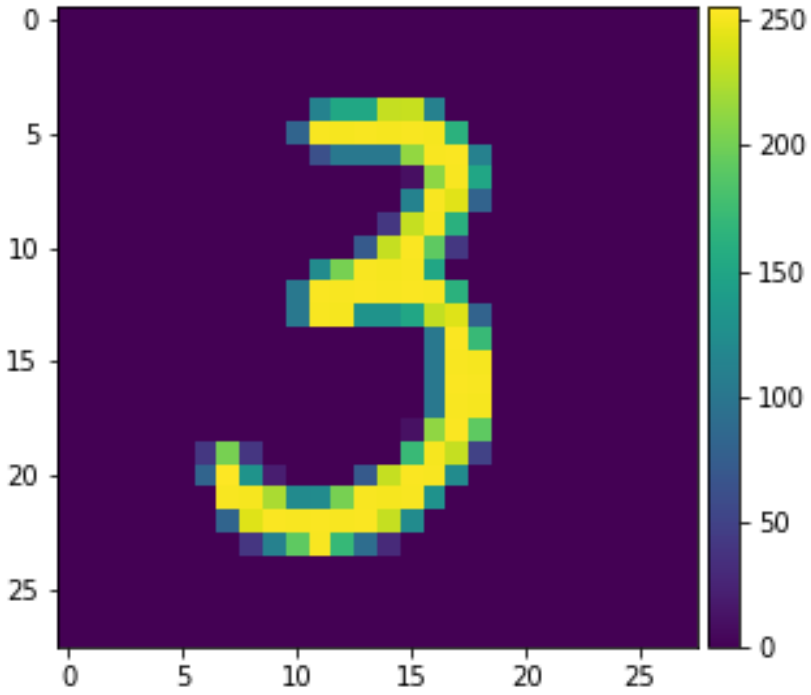


Figure 3.1: Examining our Reshaped Data

It looks like we are doing ok so far. Now, let us center and standardize our data, as before.

```
# standardize both the train and test set with the values computed
    from the training set!!!
mean_val = np.mean(X_train_img)
std_val = np.std(X_train_img)

# THIS IS SUPER IMPORTANT!!!
X_train_centered = (X_train_img - mean_val) / std_val
X_test_centered = (X_test_img - mean_val) / std_val
```

Listing 3.10: Center and Standardize

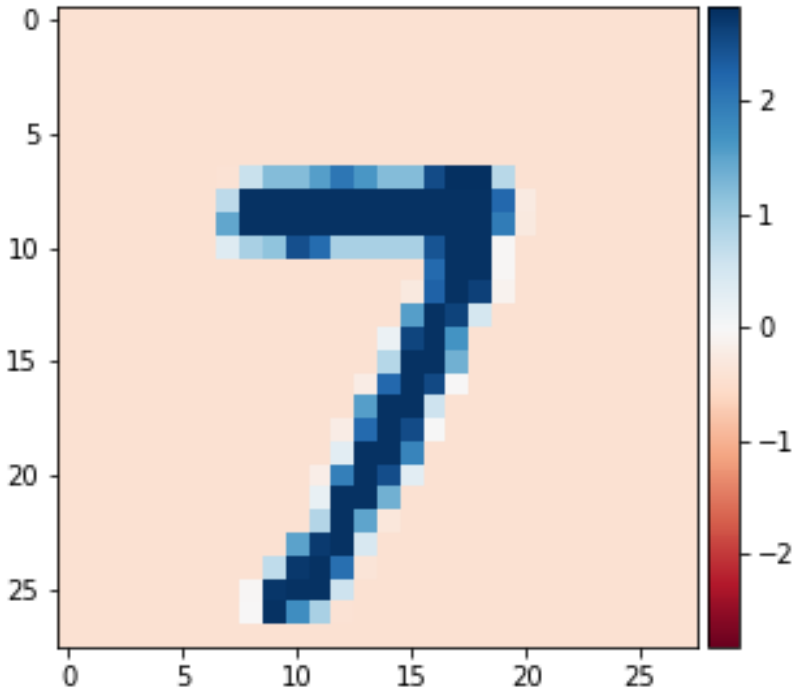


Figure 3.2: Examining our Centered and Standardized Data

Let us take one more look at this newly transformed data just to make sure we are still ok.

```
ix = np.random.randint(0,len(X_train_img)-1)
imshow(np.squeeze(X_train_centered[ix]))
plt.show()
```

Listing 3.11: Check on Centered and Standardized Data

Looks alright. We can note that the value for each pixel was originally in the range [0,255], and now it looks like the values are somewhere near [-3,3]. This is very important to help our optimizer.

Next, let us create some helpful variables and set some random seeds.

```
# create some helpful variables
n_features = X_train_img.shape[1]
random_seed = 7
np.random.seed(random_seed)
tf.set_random_seed(random_seed)

# create onehot vectors for y_train
y_train_onehot = to_categorical(y_train)
n_classes = y_train_onehot.shape[1]
INPUT_SHAPE = X_train_centered.shape[1]
NUM_CHANNELS = X_train_centered.shape[3]
```

Listing 3.12: Some Useful Setup

Now that we set our random seeds, created one-hot labels for each images, and created some other useful variables, let us build our computation graph for a convolutional neural network.

```
# create the convolutional network
model = Sequential()

model.add(Conv2D(32, (3,3), activation='relu', input_shape=(
    INPUT_SHAPE, INPUT_SHAPE, NUM_CHANNELS)))
model.add(Conv2D(32, (3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.1))

model.add(Conv2D(64, (3,3), activation='relu'))
model.add(Conv2D(64, (3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.1))

```

```
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(10, activation='softmax'))

optimizer = Adam(lr=1e-4)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

Listing 3.13: Build Conv Net Computation Graph

We just created a conv net with two conv layers each with a kernel size of 3x3 and 32 channels, a max pooling operation with a kernel size of 2x2, and added dropout which will kill 10% of the units each epoch. This is followed by another module with nearly the same hyperparameters, except we will now increase to 64 channels. Then, we will flatten the resulting tensors to be vectors and use two fully connected layers at the end of the network. These work exactly the same as in the previous section. We have used an Adam optimizer with a learning rate of 0.0001 and used a categorical cross entropy loss function. We can see a summary of the model like so:

```
model.summary()
```

Listing 3.14: Model Summary

which outputs

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
```

```
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 10, 10, 64)        18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 64)          36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               262400
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               32896
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
```

```
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 361,578
Trainable params: 361,578
Non-trainable params: 0
_________________________________________________________________
```

It is interesting to note that the most parameters come from our FC layers, and the convolutional layers have far less parameters to train.

Now it is time to train our model.

```
model_name = 'conv-1'

ES = EarlyStopping(monitor='val_loss', min_delta=0, patience=5,
    verbose=1, mode='auto')
CP = ModelCheckpoint('model_checkpoints/model-{}.h5'.format(model_name
    ), monitor='val_loss',
                     verbose=0, save_best_only=True, mode='auto',
    period=1)

# train the model, holding out 10% for validation
history = model.fit(X_train_centered, y_train_onehot,
                    batch_size=256, epochs=1000,
                    verbose=1,
                    validation_split=0.1,
                    callbacks=[ES, CP])
```

Listing 3.15: Train the Conv Net

We have set a name for our model to be saved as an HDF5 file (The HDF Group, 2018). This is an efficient format to store our model in for I/O purposes. Our early

stopper will make sure that we stop training if our validation loss does not improve with a patience of five epochs. The model checkpointer will save our best model as we train. The model will be fit with the centered and standardized data and one-hot labels as input. We will keep the minibatch size at 256 and we will allow a maximum of 1000 epochs, even though we will most likely not get that far since we have the early stopper. Again, we will use a random 10% of the training data as a validation set for each epoch. After running this code, the output will look very similar to the output from the previous section.

Finally, we load the best model that we got during training, and see how well it performs on the training and test sets.

```
model = load_model('model_checkpoints/model-{}.h5'.format(model_name))

y_train_pred = model.predict_classes(X_train_centered, verbose=0)
correct_preds = np.sum(y_train == y_train_pred, axis=0)
train_acc = correct_preds / y_train.shape[0]
print('First 3 predictions: ', y_train_pred[:3])
print('Training Accuracy: %.2f%%' % (train_acc * 100))

y_test_pred = model.predict_classes(X_test_centered, verbose=0)
correct_preds = np.sum(y_test == y_test_pred, axis=0)
test_acc = correct_preds / y_test.shape[0]
print('Test accuracy: %.2f%%' % (test_acc * 100))
```

Listing 3.16: Make Predictions with Conv Net

which outputs

```
First 3 predictions:  [5 0 4]
Training Accuracy: 99.87%
```

```
Test accuracy: 99.39%
```

Thus, we have improved to over 99% accuracy on the held out test set. In the next chapter, we will combine more machine learning theory, computer vision techniques, and neural networks to compete in the 2018 Kaggle Data Science Bowl.

Chapter 4

Kaggle Data Science Bowl 2018

In this chapter, we will compete in the 2018 Kaggle Data Science Bowl. We will have to use more machine learning theory and skills than have been previously discussed in this paper. To implement models, we have used three different GPUs: a Nvidia GeForce GTX 1080, Nvidia Tesla K80s, and Nvidia Tesla P100s. The GTX 1080 is traditionally a gaming GPU, but it has the ability to run TensorFlow and Keras at speeds much better than CPUs. The K80 and P100 GPUs are even better than the GTX 1080, and the P100 is the best of the three by far. For example, a single training epoch of a large model would take about 5 hours, about 3 hours, and about 50 minutes on one GTX 1080, on four K80s, and two P100s, respectively. The K80s and P100s were used by way of XSEDE (Towns et al., 2014), a supercomputer resource. We will discuss more about XSEDE in Section 4.4.

4.1 Competition Overview

The goal of the 2018 Kaggle Data Science Bowl is to determine where cell nuclei are in medical imaging. According to the competition description, "Identifying the cells' nuclei

is the starting point for most analyses because most of the human body's 30 trillion cells contain a nucleus full of DNA, the genetic code that programs each cell. Identifying nuclei allows researchers to identify each individual cell in a sample, and by measuring how cells react to various treatments, the researcher can understand the underlying biological processes at work." Furthermore, "[f]inding the nucleus helps to locate cells in varied conditions to enable faster cures, free biologists to focus on solutions, improve throughput for research and insight, reduce time-to-market for new drugs - currently 10 years, increase the number of compounds for experiments, and improve health and increase quality of life" (Booz, Allen, Hamilton and Kaggle, 2018).

Here is a description of the evaluation metrics that the competition uses: "This competition is evaluated on the mean average precision at different intersection over union (IoU) thresholds. The IoU of a proposed set of object pixels and a set of true object pixels is calculated as:

$$IoU(A,B) = \frac{A \quad B}{A \bigcup B} \tag{4.1}$$

The metric sweeps over a range of IoU thresholds, at each point calculating an average precision value. The threshold values range from 0.5 to 0.95 with a step size of 0.05: (0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95). In other words, at a threshold of 0.5, a predicted object is considered a "hit" if its intersection over union with a ground truth object is greater than 0.5.

At each threshold value t, a precision value is calculated based on the number of true positives (TP), false negatives (FN), and false positives (FP) resulting from comparing the predicted object to all ground truth objects:

$$\frac{TP(t)}{TP(t)+FP(t)+FN(t)} \tag{4.2}$$

A true positive is counted when a single predicted object matches a ground truth object with an IoU above the threshold. A false positive indicates a predicted object had no associated ground truth object. A false negative indicates a ground truth object had no associated predicted object. The average precision of a single image is then calculated as the mean of the above precision values at each IoU threshold:

$$\frac{1}{|thresholds|}\sum_{t}\frac{TP(t)}{TP(t)+FP(t)+FN(t)} \tag{4.3}$$

Lastly, the score returned by the competition metric is the mean taken over the individual average precisions of each image in the test dataset." (Booz, Allen, Hamilton and Kaggle, 2018)

4.2 Description of Data

The training data set consists of 670 images and the test set consists of 65 images. There are a few different classes of images. That is, most have light nuclei with a dark background, some have dark nuclei with a light background, some have purple nuclei with light background, some have red streaks in them. See Appendix C for a few sample images to demonstrate the different types of images that are included in the datasets. The images are all different dimensions, as well. We will have to resize them to all be the same shape to feed into our models.

4.3 Deep Learning Model

The model that I have used so far is called U-Net (Ronneberger et al., 2015). It was designed for biomedical image segmentation and is especially suited for when the training

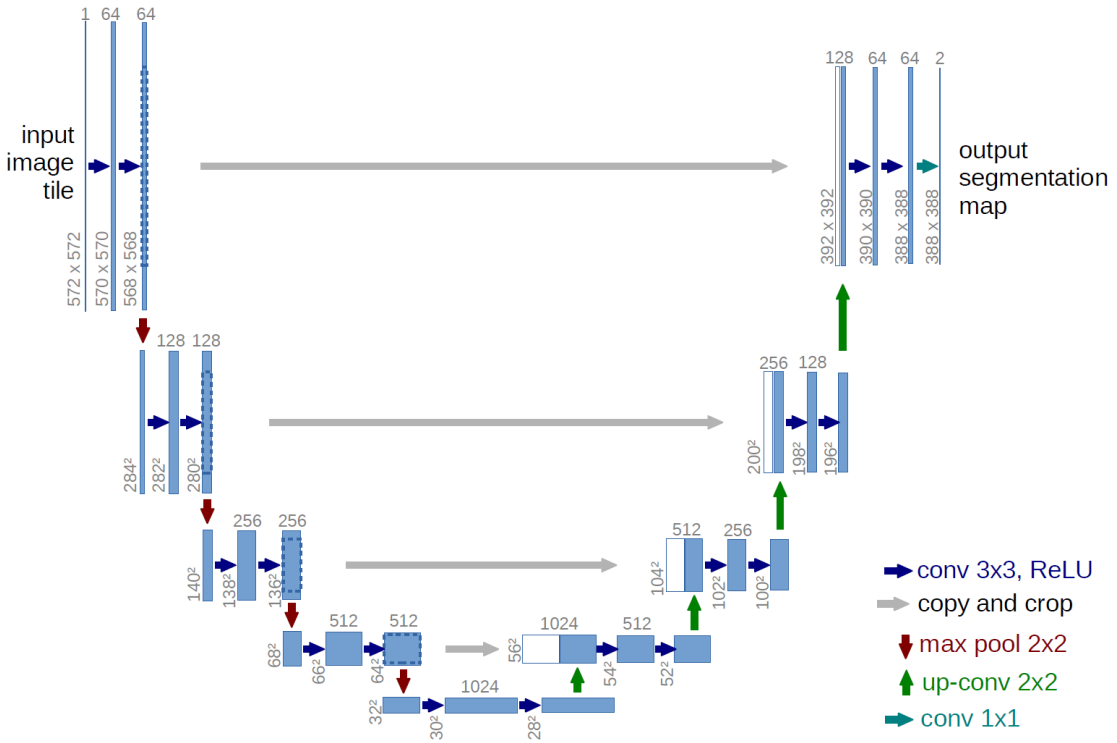


Figure 4.1: U-net architecture (Ronneberger et al., 2015)

data set is not very large, which is a common occurrence in the medical field. The U-Net is a type of convolutional network that uses convolutional layers as well as deconvolutional layers.

4.4 XSEDE and SLURM

As stated by the company website (`https://www.xsede.org/about/what-we-do`) The Extreme Science and Engineering Discovery Environment (XSEDE) "is an NSF-funded virtual organization that integrates and coordinates the sharing of advanced digital services - including supercomputers and high-end visualization and data analysis

resources - with researchers nationally to support science." For this project, we used GPU compute nodes from the Pittsburg Supercomputing Center's (PSC) Bridges GPU resource (Nystrom et al., 2015). This resource allowed us to train larger models at a much faster pace than would have been possible on most personal computers.

To run code on the compute nodes, we have to use the Simple Linux Utility for Resource Management (SLURM) (Jette et al., 2002). Since users from all over the country are trying to use the resources available within the supercomputer, there has to be a way to make sure that only one job is running on a particular resource at any given time. SLURM is what takes care of automatically managing all of the resources. We submit jobs to SLURM, and when the compute resources that we request become available, our code is then run. See Appendix D for an example of a SLURM batch script that will run code to train a model.

Within the SLURM batch script, we define parameters such as which resource we want to use, the maximum amount of time we want our job to run for, what to do if the code begins, ends, or fails, and other information about the job and compute resources. Then, we load any modules that we may need for the job. A module on XSEDE is like an environment that contains prepackaged libraries that we need. For example, we used an Anaconda module so that we had access to all of the Python packages that Anaconda provides, rather than having to list all of them ourselves. Next, we change to the directory where our Python file is and activate our conda environment. The conda environment is another layer of contained Python packages that we added in addition to the Anaconda module. Basically, this is just an extra layer of customization to run the job that we need. Finally, we run the Python script which trains a Keras model.

Essentially, all we do is run the SLURM batch script in order to get the compute resources that we need, which in turn runs the Python script for us after the requested resources become available.

4.5 Discussion

We first resized the images to 512x512. Originally, they were all different shapes and sizes, mostly 256x256 or larger in at least one dimension. We need them to all be the same size in order to feed them into a model. The thought was that this size image would be a reasonable balance between the smaller and larger original image sizes.

Initially, the goal was to overfit training data as much as possible, and then add regularization from there in order to make the model generalize better.

Since the images are not all of the same color scheme, we wanted to make them more uniform to help the model recognize the cell nuclei more easily. We tried converting color images to grayscale, which works alright for most of the images, even though after you convert them you then also have to invert the dark and light to match up properly with the majority of training data. It took extra work to get some of the images with red in them to look more like the majority of the other images. The preprocessing of the images caused some of the greatest improvements in the model. See Appendix A for the code that was used to accomplish this.

To preprocess all of the images, we first apply a bilateral filter to each image. This blurs the images just a bit which is known to help with computer vision and image processing tasks. The next step is to convert all of the images to grayscale. For most of the images, dark background with light nuclei, we convert straight to single channel grayscale images. For the gray images with light backgroud and dark nuclei, we invert

the grayscale image to be more in line with the most common images. For the color images, there were a couple of techniques. Most of the time, we convert them to grayscale, which creates light background and dark nuclei, so we just invert them. For the few images with red streaks in them, we found that, in the training set, the mean of the green channel was always less than 125 and the other color images did not. So, we filtered them out by that criteria, and then chose the color channel which had the highest mean value to convert to grayscale and then invert. See the function `handle_color_imgs()` in Appendix A to see the implementation details.

Our score improved after mean centering the data for each individual image, rather than using the mean of the entire training set on all the images. We believe this is due to the different image types. Most had a very small mean value, e.g., around 5, while others had a mean value of around 200. The practice of centering image data is a standard in computer vision tasks.

We performed cross-validation in both a manual and automated fashion, depending on the task. For creating random, augmented training and validation sets, we have implemented an automated approach using Keras. We have not automated the tasks of tuning hyperparameters such as the optimization learning rate, the model architecture, the momentum coefficient, or whether to use NAG due to computational constraints. We would first use a grid approach to find better learning rate, and would then try to more finely tune that particular model with a random grid search approach. See Appendix B to see the implementation of data augmentation and model training.

A few different network sizes were implemented. As can be seen in Figure 4.1, the first conv layer creates 64 filters. We used this architecture in our final model, but we also tried scaling it down to have 32 and 16 filters in the first layer. The rest of the layers

were then scaled appropriately by dividing the number of filter by 2 (corresponding with the first layer having 32 filters) or 4 (corresponding to the first layer having 16 filters).

The authors of the U-net paper discuss the importance of data augmentation. We implemented this with a generator function which randomly applied shearing, rotation, zooming, and width and height shift on our input data. Keras has a function that automatically does this for us as training is underway. This also takes care of randomly choosing training and validation sets for each training epoch. See the `augment_data()` function in Appendix B for the implementation details.

We tried almost every optimization algorithm our for our model. It turned out that minibatch Nesterov SGD with momentum worked the best by far.

For our loss function, we used a combination of binary cross entropy and a Sorensen-Dice coefficient (aka F1 score) loss (Sørensen, 1948) (Dice, 1945). See the functions `dice_coef()` and `bce_dice_loss()` in Appendix B for implementation details.

As an extra twist to things, we tried converting the input images to their dual matrix and using that as input. Unfortunately, this did not work. The thinking was that we could create a scaled version of the dual matrix so that input values were all very small on both the positive and negative side of zero, so maybe this would be similar to centering and scaling the input images. However, the dual transformation made the nuclei in the images undetectable. It only took about a day to implement the code and run a couple models so it was worth checking on. This idea came from research on Markov chains with Dr. Alan Krinik (Nguyen, 2017). To see the implementation in Python, refer to Appendices E and F. To see some of the mathematics and theory involving dual matrices, see G.

4.6 Results

Our score on the Kaggle Leader Board for this competition was a mean IoU of 0.383. This puts us in 464^{th} place out of 3,634 which is the top 13%. This is far from in the money as far as the competition goes, where the winner has a mean IoU score of 0.631.

4.7 Acknowledgement

Special thanks to Kjetil Åmdal-Sævik for his Kaggle Kernel (Åmdal-Sævik, 2018). This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Åmdal-Sævik, K. (2018). Keras u-net starter - lb 0.277. `https://www.kaggle.com/keegil/keras-u-net-starter-lb-0-277`. Kaggle Kernel for 2018 Kaggle Data Science Bowl. Accessed: 2018-01-20.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In Schölkopf, B., Platt, J. C., and Hoffman, T., editors, *Advances in Neural Information Processing Systems 19*, pages 153–160. MIT Press. `http://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf`.

Booz, Allen, Hamilton and Kaggle (2018). 2018 Data Science Bowl. https://www.kaggle.com/c/data-science-bowl-2018. Kaggle.com, First Accessed: 2018-01-17.

Chollet, F. et al. (2015). Keras. https://github.com/keras-team/keras.

Clevert, D., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289. http://arxiv.org/abs/1511.07289.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., aurelio Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., and Ng, A. Y. (2012). Large scale distributed deep networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc. http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf.

Dechter, R. (1986). Learning while searching in constraint-satisfaction-problems. In *AAAI*, pages 178–185.

Dettmers, T. (2015). Deep learning in a nutshell: History and training. https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training/. Nvidia Blog: Parallel Forall, Accessed: 2018-01-08.

Dice, L. R. (1945). Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302. http://dx.doi.org/10.2307/1932409.

Dozat, T. (2015). Incorporating nesterov momentum into adam. https://web.stanford.edu/~tdozat/files/TDozat-CS229-Paper.pdf.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159. http://dl.acm.org/citation.cfm?id=1953048.2021068.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR. http://proceedings.mlr.press/v9/glorot10a.html.

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

Gupta, D. (2017). Architecture of convolutional neural networks (cnns) demystified. https://www.analyticsvidhya.com/blog/2017/06/architecture-of-convolutional-neural-networks-simplified-demystified/. Analytics Vidhya, Accessed: 2018-02-26.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852. http://arxiv.org/abs/1502.01852.

Hinton, G., Srivastava, N., and Swersky, K. (2014). Neural networks for machine learning, lecture 6e. rmsprop: Divide the gradient by a running average of its recent magnitude. http://www.cs.toronto.edu/~tijmen/csc321/slides/

lecture_slides_lec6.pdf. Coursera: https://www.coursera.org/learn/neural-networks, Lecture Notes Accessed: 2018-02-13.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366. http://www.sciencedirect.com/science/article/pii/0893608089900208.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95. http://aip.scitation.org/doi/abs/10.1109/MCSE.2007.55.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167. http://arxiv.org/abs/1502.03167.

Ivakhnenko, A. and Lapa, V. (1967). *Cybernetics and forecasting techniques*. Modern analytic and computational methods in science and mathematics. American Elsevier.

Jette, M. A., Yoo, A. B., and Grondona, M. (2002). Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag.

Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open source scientific tools for Python. http://www.scipy.org/.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980. http://arxiv.org/abs/1412.6980.

Klambauer, G., Unterthiner, T., Mayr, A., and Hochreiter, S. (2017). Self-normalizing neural networks. *CoRR*, abs/1706.02515. http://arxiv.org/abs/1706.02515.

Kurenkov, A. (2015a). A 'brief' history of neural nets and deep learning, part 1. http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/. Andrey Kurenkov's Web World, Accessed: 2018-03-16.

Kurenkov, A. (2015b). A 'brief' history of neural nets and deep learning, part 2. http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning-part-2/. Andrey Kurenkov's Web World, Accessed: 2018-03-16.

LeCun, Y. (1998). The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/. Yann LeCun's Website, Accessed: 2018-01-16.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551. http://dx.doi.org/10.1162/neco.1989.1.4.541.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324. http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

Li, F.-F., Johnson, J., and Yeung, S. (2017a). Cs231n: Convolutional neural networks for visual recognition. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture3.pdf. Lecture slides, Lecture 3, Accessed: 2018-01-31.

Li, F.-F., Johnson, J., and Yeung, S. (2017b). Cs231n: Convolutional neural networks for visual recognition. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf. Lecture slides, Lecture 4, Accessed: 2018-02-26.

Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. Master's thesis, Univ. Helsinki.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*.

McKinney, W. (2010). Data structures for statistical computing in python. In van der Walt, S. and Millman, J., editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56.

Miller, R. (2017). Google's second generation tpu chips takes machine learning processing to a new level. https://techcrunch.com/2017/05/17/google-announces-second-generation-of-tensor-processing-unit-chips/. TechCrunch, Accessed: 2018-01-11.

Minsky, M. and Papert, S. (1969). *Perceptrons. An Introduction to Computational Geometry*. M.I.T. Press, Cambridge, MA.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Interna-*

tional Conference on Machine Learning, ICML'10, pages 807–814, USA. Omnipress. http://dl.acm.org/citation.cfm?id=3104322.3104425.

Neelakantan, A., Vilnis, L., Le, Q. V., Sutskever, I., Kaiser, L., Kurach, K., and Martens, J. (2015). Adding gradient noise improves learning for very deep networks. *CoRR*, abs/1511.06807. https://arxiv.org/abs/1511.06807.

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$. *Doklady AN USSR*, 269:543–547. https://ci.nii.ac.jp/naid/20001173129/en/.

Ng, A., Katanforoosh, K., and Mourri, Y. B. (2017). Neural networks and deep learning. https://www.coursera.org/learn/neural-networks-deep-learning. Coursera, deeplearning.ai, Accessed: 2017-08-14.

Nguyen, U. (2017). TRIDIAGONAL STOCHASTIC MATRICES. Master's thesis, California State Polytechnic University at Pomona.

Nystrom, N. A., Levine, M. J., Roskies, R. Z., and Scott, J. R. (2015). Bridges: A uniquely flexible hpc resource for new communities and data analytics. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, XSEDE '15, pages 30:1–30:8, New York, NY, USA. ACM. http://doi.acm.org/10.1145/2792745.2792775.

Orr, G., Schraudolf, N., and Cummins, F. (1999). Momentum and learning rate adaptation. https://www.willamette.edu/~gorr/classes/cs449/momrate.html. CS-499: Neural Networks, Willamette University, Accessed: 2018-02-06.

Raina, R., Madhavan, A., and Ng, A. Y. (2009). Largescale deep unsupervised learning using graphics processors. In *International Conf. on Machine Learning*.

Raschka, S. (2015). Single-layer neural networks and gradient descent. http://sebastianraschka.com/Articles/2015_singlelayer_neurons.html. Sebastian Raschka's Website, Accessed: 2018-01-09.

Redmon, J. (2013). Mnist in csv. https://pjreddie.com/projects/mnist-in-csv/. Joseph Redmon's Website, Accessed: 2018-01-16.

Ronneberger, O., Fischer, P., and Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 9351 of *LNCS*, pages 234–241. Springer. (available on arXiv:1505.04597 [cs.CV]) and http://lmb.informatik.uni-freiburg.de/Publications/2015/RFB15a.

Rosebrock, A. (2016). *Practical Python and OpenCV: An Introductory, Example Driven Guide to Image Processing and Computer Vision*. pyimagesearch, 3 edition.

Rosenblatt, F. (1957). *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory. https://books.google.com/books?id=P_XGPgAACAAJ.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747. http://arxiv.org/abs/1609.04747.

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

S. McCulloch, W. and Pitts, W. (1943). A logical calculus of the idea immanent in nervous activity. *Bulletin of Mathematical Biology*, 5:115–133.

Sørensen, T. (1948). A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons. *Kongelige Danske Videnskabernes Selskab*, 5(4):1–34.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958. http://jmlr.org/papers/v15/srivastava14a.html.

The HDF Group (1997-2018). Hierarchical Data Format, version 5. http://www.hdfgroup.org/HDF5/.

Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G. D., Roskies, R., Scott, J. R., and Wilkins-Diehr, N. (2014). Xsede: Accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74. https://doi.ieeecomputersociety.org/10.1109/MCSE.2014.80.

van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30. http://aip.scitation.org/doi/abs/10.1109/MCSE.2011.37.

van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., Gouillart, E., and Yu, T. a. (2014). scikit-image: image processing in python. *PeerJ*, 2:e453. https://doi.org/10.7717/peerj.453.

van Veen, F. (2016). The neural network zoo. www.asimovinstitute.org/neural-network-zoo/. The Asimov Institute, Accessed: 2018-01-08.

Weinberger, K. (2015). Bias/variance and model selection. http://www.cs.cornell.edu/courses/cs4780/2015fa/web/lecturenotes/lecturenote13.html. Cornell CS, Accessed: 2018-02-05.

Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, Cambridge, MA.

Widrow, B. (1960). *Adaptive "Adaline" neuron using chemical "memistors"*. Number Technical Report 1553-2. Stanford Electron. Labs., Stanford, CA.

Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701. http://arxiv.org/abs/1212.5701.

Appendix A

Kaggle Data Pre-Processing

data_exploration_and_preprocessing

March 13, 2018

In [1]:
```
import os
import sys
import random
import warnings

import h5py

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

from itertools import chain
from skimage.io import imread, imshow, imread_collection, concatenate_images
from skimage.transform import resize
from skimage.morphology import label
from skimage.color import rgb2gray
from skimage.util import invert
from skimage.restoration import denoise_bilateral


# Set some parameters
BATCH_SIZE=2**3
IMG_WIDTH = 2**9
IMG_HEIGHT = 2**9
IMG_CHANNELS = 3

DATA_PATH = 'C:/Users/lyche/Desktop/cpp/thesis_stuff/data_science_bowl/input/'
TRAIN_PATH = DATA_PATH + 'stage1_train/'
TEST_PATH = DATA_PATH + 'stage1_test/'

warnings.filterwarnings('ignore', category=UserWarning, module='skimage')
seed = 3

# Get train and test IDs
train_ids = next(os.walk(TRAIN_PATH))[1]
```

```
test_ids = next(os.walk(TEST_PATH))[1]
```

1 Get the data

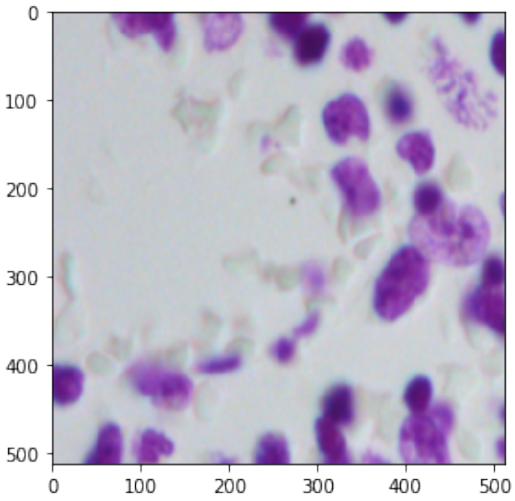
In [2]:
```
# Get and resize train images and masks
X_train = np.zeros((len(train_ids), IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS),
                   dtype=np.uint8)
Y_train = np.zeros((len(train_ids), IMG_HEIGHT, IMG_WIDTH, 1), dtype=np.bool)
print('Getting and resizing training images and masks...')
sys.stdout.flush()
for n, id_ in enumerate(train_ids):
    path = TRAIN_PATH + id_
    img = imread(path + '/images/' + id_ + '.png')[:,:,:IMG_CHANNELS]
    img = resize(img, (IMG_HEIGHT, IMG_WIDTH), mode='constant', preserve_range=True)
    X_train[n] = img
    mask = np.zeros((IMG_HEIGHT, IMG_WIDTH, 1), dtype=np.bool)
    for mask_file in next(os.walk(path + '/masks/'))[2]:
        mask_ = imread(path + '/masks/' + mask_file)
        mask_ = np.expand_dims(resize(mask_, (IMG_HEIGHT, IMG_WIDTH),
                                      mode='constant', preserve_range=True),
                               axis=-1)
        mask = np.maximum(mask, mask_)
    Y_train[n] = mask

# Get and resize test images
X_test = np.zeros((len(test_ids), IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS),
                  dtype=np.uint8)
sizes_test = []
print('Getting and resizing test images...')
sys.stdout.flush()
for n, id_ in enumerate(test_ids):
    path = TEST_PATH + id_
    img = imread(path + '/images/' + id_ + '.png')[:,:,:IMG_CHANNELS]
    sizes_test.append([img.shape[0], img.shape[1]])
    img = resize(img, (IMG_HEIGHT, IMG_WIDTH), mode='constant', preserve_range=True)
    X_test[n] = img

print('Done!')
```

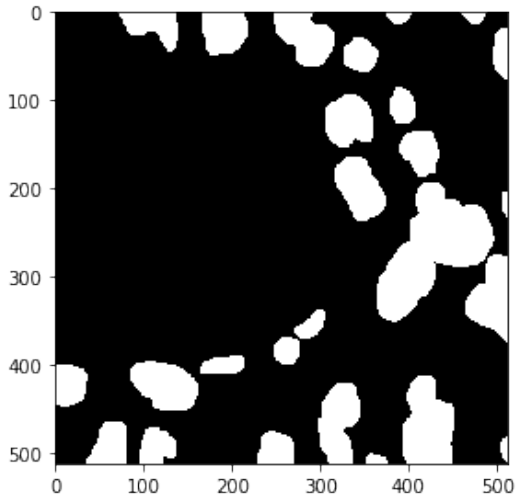
```
Getting and resizing training images and masks...
Getting and resizing test images...
Done!
```

Let's see if things look all right by drawing some random images and their associated masks.

In [3]:
```
# Check if training data looks alright
ix = random.randint(0, len(train_ids))
```

```
imshow(X_train[ix])
plt.show()
imshow(np.squeeze(Y_train[ix]))
plt.show()
```





Let's do some exploration of the color images to see how to best convert them to grayscale in order to best match up with the majority of the input images. That is, darker background and lighter nuclei.

```
In [4]: # find the color images and their indexes in the original training set
        color_imgs = []
        indexes = []
        for i in range(len(X_train)):
            means = []
            for j in range(2):
                means.append(X_train[i,:,:,j].mean())

            if means[0] != means[1]:
                color_imgs.append(X_train[i])
                indexes.append(i)

        color_imgs = np.array(color_imgs, dtype=np.uint8)
```

```
In [5]: print(color_imgs.shape)
        print(len(indexes))
```

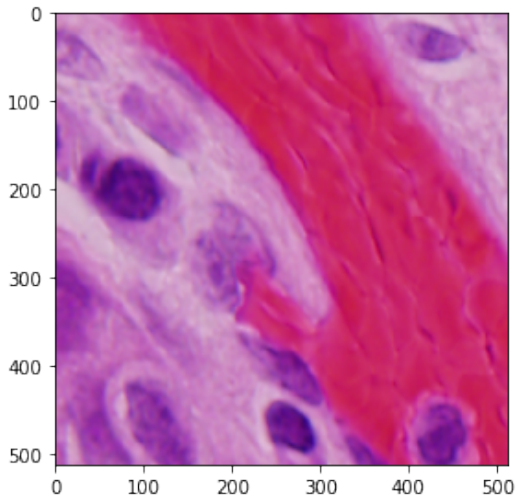
```
(107, 512, 512, 3)
107
```

```
In [7]: # find where the red channel is very high
        # ...in order to find the images with red streaks
        high_red_indexes = []
        for img in range(len(color_imgs)):
            tmp = (color_imgs[img,:,:,0] > 200).astype(bool)
            summation = np.sum(tmp)
            if summation > 0:
                high_red_indexes.append(indexes[img])
```

```
[2, 3, 34, 40, 54, 57, 74, 81, 117, 137, 144, 174, 175, 185, 212, 228, 235, 257, 266, 306, 316
```

We found image with index 174 has a red streak in it. Let's try to see what the pixels look like in the red area and in some other areas of the image.

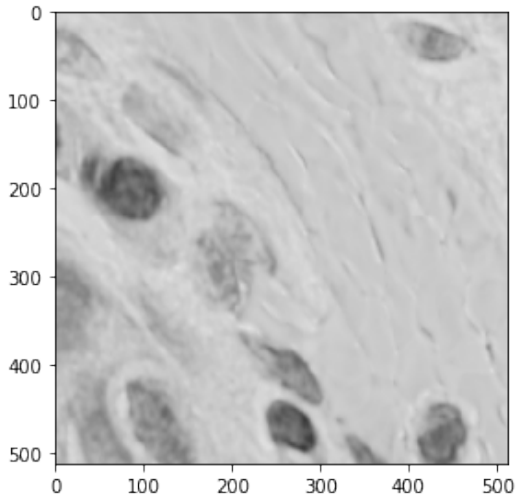
```
In [9]: red = X_train[174]
        imshow(red)
        plt.show()
```



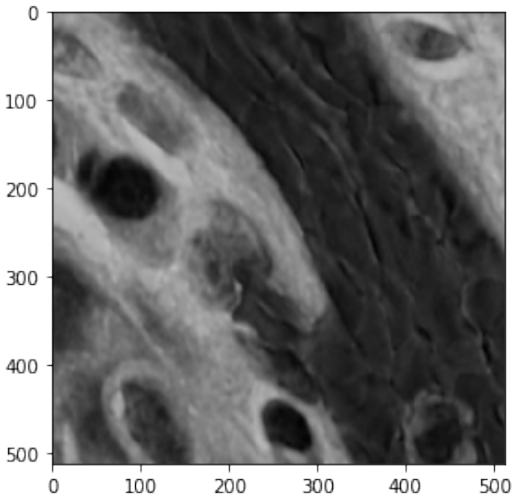

```
In [10]: print(red[351,351,:])
         print(red[100,300,:])
         print(red[480,280,:])
         print(red[200,90,:])
```

```
[205  44  98]
[215  72 134]
[ 95  12 126]
[112  19 139]
```

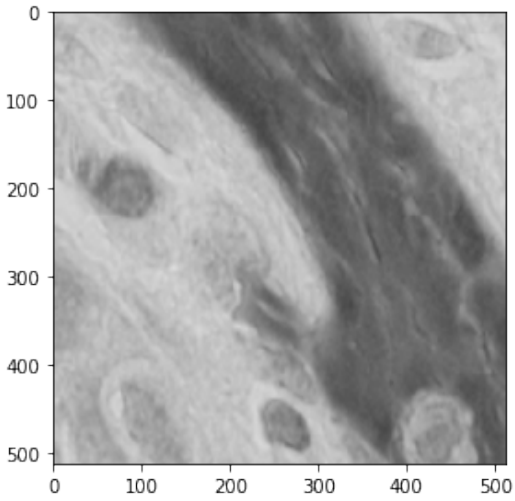
```
In [11]: # Let's see which image channel gives us
         # the nicest version of the image to get rid of
         # the big red streak
         for n in range(3):
             imshow(red[:,:,n])
             plt.show()
             print('Color channel:      ', n)
             print('Color channel mean: ', red[:,:,n].mean())
             print()
```




```
Color channel:       0
Color channel mean:  192.641277313
```

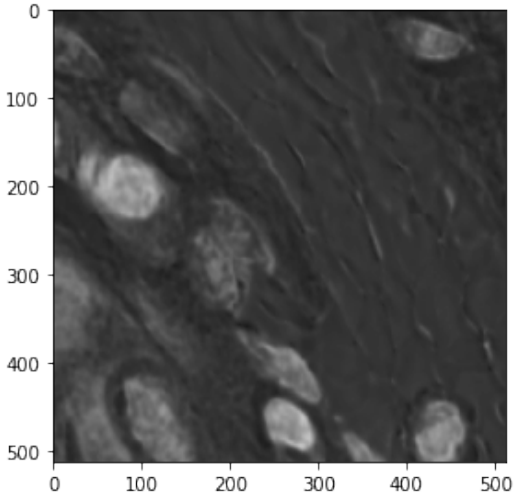


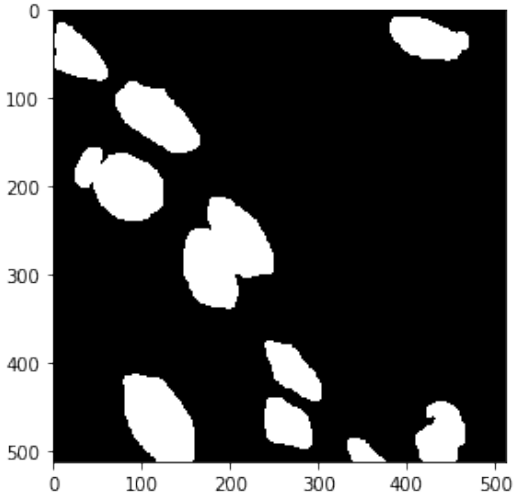

```
Color channel:       1
Color channel mean:  81.3681983948
```




```
Color channel:       2
Color channel mean:  152.419868469
```

In [13]:
```
# Let's see what the image will look like if we use
# the argmax of the channel means as the grayscale image and invert it
n = 174
tmp = np.copy(X_train[n])
tmp = tmp[:,:,np.argmax(np.mean(tmp,axis=(0,1)))]
imshow(invert(rgb2gray(tmp)))
#imshow(rgb2gray(tmp))
#imshow(tmp)
plt.show()
imshow(np.squeeze(Y_train[n]))
plt.show()
```


0
100
200
300
400
500
0
100
200
300
400
500


0
100
200
300
400
500
0
100
200
300
400
500

Now let's create some functions to do our preprocessing.

In [3]:
```
# combines all steps into one function
# denoise, then handle image classes individually to grayscale


# denoise imgs with bilateral filter
def denoise_imgs(X):
    X_out = np.zeros(X.shape, dtype=np.float64)
    for i in range(len(X)):
        X_out[i] = denoise_bilateral(X[i], sigma_color=0.5, sigma_spatial=1)

    return X_out

# preprocess color images with and without red streaks
# ...convert to grayscale and np.float64 datatype
def handle_color_imgs(img):
    img_out = np.zeros((IMG_HEIGHT, IMG_WIDTH, 1), dtype=np.float64)

    # most images have no red streaks so set this to False by default
    red_streak = False

    # change red to True if image fits criteria for a red image
    if img[:,:,1].mean() < 125:
        red_streak = True


    # if no red streak
    if not red_streak:
        # invert the grayscale version of the image
        img_out = np.expand_dims(invert(rgb2gray(tmp)), axis=2)

    # if red streaks
    else:
        # pick the color channel that has the maximum channel mean
        img_out = img[:,:,np.argmax(np.mean(img, axis=(0,1)))]

        # invert the grayscale version of the image
        img_out = np.expand_dims(invert(rgb2gray(img_out)), axis=2)

    return img_out




def make_gray(X):
    X_out = np.zeros((X.shape[0], IMG_HEIGHT, IMG_WIDTH, 1), dtype=np.float64)
    for i in range(len(X)):
```

```
            means = np.mean(X[i], axis=(0,1))

            if means[0] == means[1]:
                if means[0] < 200:

                    # dark backgrounds stay the same
                    X_out[i] = np.expand_dims(X[i,:,:,0], axis=2)
                else:

                    # invert the light backgrounds so that they have dark backgrounds
                    X_out[i] = np.expand_dims(invert(X[i,:,:,0]), axis=2)
            else:

                # handle color images
                X_out[i] = handle_color_imgs(X[i])

        return X_out

    def preprocess(X):
        X_out = np.zeros(X.shape, dtype=np.float64)
        X_out = denoise_imgs(X)
        X_out = make_gray(X_out)
        return X_out
```

In [ ]:
```
# preprocess data
X_train_processed = preprocess(X_train)
X_test_processed = preprocess(X_test)
```

Create some functions to save our datasets to memory.

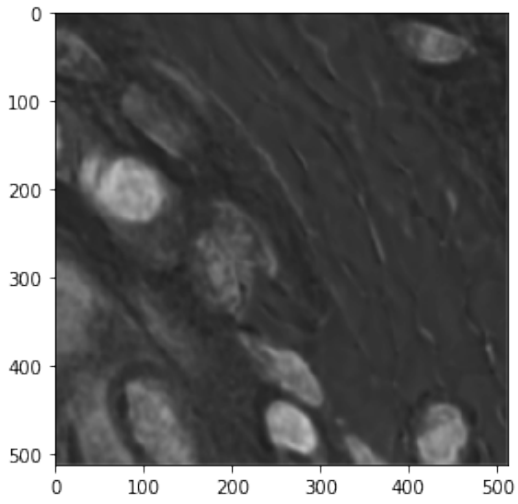
In [4]:
```
# save the new dataset so we don't have to recreate it every time
def data_to_h5(dataset, name):
    h5f = h5py.File(name, "w")
    h5f.create_dataset(name[:len(name)-3], data=dataset)
    h5f.close()

def load_h5_dataset(name):
    h5f = h5py.File(name, 'r')
    data = h5f[name[:len(name)-3]][:]
    h5f.close()
    return data


def save_h5_datasets():
    data_to_h5(X_train, 'X_train.h5')
    data_to_h5(Y_train, 'Y_train.h5')
    data_to_h5(X_test, 'X_test.h5')
    data_to_h5(sizes_test, 'sizes_test.h5')
```

```
    data_to_h5(X_train_processed, 'X_train_processed.h5')
    data_to_h5(X_test_processed, 'X_test_processed.h5')

def load_h5_datasets():
    X_train = load_h5_dataset('X_train.h5')
    Y_train = load_h5_dataset('Y_train.h5')
    X_test = load_h5_dataset('X_test.h5')
    sizes_test = load_h5_dataset('sizes_test.h5')
    X_train_processed = load_h5_dataset('X_train_processed.h5')
    X_test_processed = load_h5_dataset('X_test_processed.h5')
    return X_train, Y_train, X_test, sizes_test, X_train_processed, X_test_processed
```

In [ ]:
```
# save X_train, y_train, X_test, sizes_test,
# X_train_processed, and X_test_processed to h5 format
save_h5_datasets()
```

In [5]:
```
# load the data sets
X_train, Y_train, X_test, sizes_test, \
X_train_processed, X_test_processed = load_h5_datasets()
```

In [6]:
```
# take a look at some preprocessed images
tmp = preprocess(X_train[174:177])
imshow(np.squeeze(tmp[0]))
plt.show()
```



Appendix B

Kaggle Data Augmentation, Model Training, Prediction, and Submission

```
1
2  # coding: utf-8
3
4  # # Import libraries
5
6  # In[ ]:
7
8
9  import os
10 import sys
11 import random
12 import warnings
13
14 import h5py
15
16 import numpy as np
```

```
17 import pandas as pd
18
19 # ------------------------------------------------------------ #
20 # these only work with IPython like in Jupyter Notebooks
21 # remove or comment out if running with python through a shell
22 #     like bash or cmd
23 #
24 #     Also, must remove or comment out blocks where images are
       displayed
25 import matplotlib.pyplot as plt
26 get_ipython().magic('matplotlib inline')
27 from skimage.io import imshow
28 # ------------------------------------------------------------ #
29
30 from skimage.transform import resize
31 from skimage.morphology import label
32 from skimage.color import rgb2gray
33 from skimage.util import invert
34
35 from keras.models import Model, load_model
36 from keras.layers import Input, LeakyReLU
37 from keras.layers.core import Dropout
38 from keras.layers.convolutional import Conv2D, Conv2DTranspose
39 from keras.layers.pooling import MaxPooling2D
40 from keras.layers.merge import concatenate
41 from keras.callbacks import EarlyStopping, ModelCheckpoint
42 from keras import backend as K
43 from keras import optimizers
44 from keras.losses import binary_crossentropy
45 from keras.preprocessing import image
```

```
46
47 import tensorflow as tf
48
49 # Set some parameters
50 # Need to match what you set them to be in
51 #      data_exploration_and_preprocessing.py
52 BATCH_SIZE=2
53 IMG_WIDTH = 2**9
54 IMG_HEIGHT = 2**9
55 IMG_CHANNELS = 1
56
57 # ------------------------------------------------------------------#
58 # expects data to be in folders like
59 # input\stage1_train\<train_img_id>\images
60 # input\stage1_train\<train_img_id>\masks
61 # &
62 # input\stage1_test\<test_img_id>\images
63 DATA_PATH = 'C:/Users/lyche/Desktop/cpp/thesis_stuff/data_science_bowl
       /input/'
64 TRAIN_PATH = DATA_PATH + 'stage1_train/'
65 TEST_PATH = DATA_PATH + 'stage1_test/'
66 # Get train and test IDs
67 train_ids = next(os.walk(TRAIN_PATH))[1]
68 test_ids = next(os.walk(TEST_PATH))[1]
69 # ------------------------------------------------------------------#
70
71 warnings.filterwarnings('ignore', category=UserWarning, module='
       skimage')
72 seed = 3
73
```

```
74
75 # # Load the datasets here
76
77 # In[ ]:
78
79
80 def load_h5_dataset(name):
81     h5f = h5py.File(name, 'r')
82     data = h5f[name[:len(name)-3]][:]
83     h5f.close()
84     return data
85
86 def load_h5_datasets():
87     Y_train = load_h5_dataset('Y_train.h5')
88     X_test = load_h5_dataset('X_test.h5')
89     sizes_test = load_h5_dataset('sizes_test.h5')
90     X_train_processed = load_h5_dataset('X_train_processed.h5')
91     X_test_processed = load_h5_dataset('X_test_processed.h5')
92     return Y_train, X_test, sizes_test, X_train_processed,
    X_test_processed
93
94 Y_train, X_test, sizes_test, X_train_processed, X_test_processed =
    load_h5_datasets()
95
96
97 # In[ ]:
98
99
100 # mean centering per image
101 def per_image_center(X):
```

```
102     X_centered = np.zeros(X.shape)
103     for i in range(len(X)):
104         mu = np.mean(X[i], axis=(0,1))
105         X_centered[i] = X[i] - mu
106 
107     return X_centered
108 
109 # apply the centering function
110 X_train_centered = per_image_center(X_train_processed)
111 X_test_centered = per_image_center(X_test_processed)
112 
113 
114 # In[ ]:
115 
116 
117 # from: https://www.kaggle.com/c0conuts/unet-imagedatagenerator-lb
    -0-336
118 
119 def augment_data(X, Y_train, train_split=0.9):
120     # Creating the training Image and Mask generator
121     image_datagen = image.ImageDataGenerator(shear_range=0.5,
    rotation_range=50, zoom_range=0.2, width_shift_range=0.2,
    height_shift_range=0.2, fill_mode='reflect')
122     mask_datagen = image.ImageDataGenerator(shear_range=0.5,
    rotation_range=50, zoom_range=0.2, width_shift_range=0.2,
    height_shift_range=0.2, fill_mode='reflect')
123 
124     # Keep the same seed for image and mask generators so they fit
    together
125 
```

```
126     image_datagen.fit(X[:int(X.shape[0]*train_split)], augment=True,
        rounds=2, seed=seed)
127     mask_datagen.fit(Y_train[:int(Y_train.shape[0]*train_split)],
        augment=True, rounds=2, seed=seed)
128
129     x=image_datagen.flow(X[:int(X.shape[0]*train_split)],batch_size=
        BATCH_SIZE, shuffle=True, seed=seed)
130     y=mask_datagen.flow(Y_train[:int(Y_train.shape[0]*train_split)],
        batch_size=BATCH_SIZE,shuffle=True, seed=seed)
131
132
133
134     # Creating the validation Image and Mask generator
135     image_datagen_val = image.ImageDataGenerator()
136     mask_datagen_val = image.ImageDataGenerator()
137
138     image_datagen_val.fit(X[int(X.shape[0]*train_split):], augment=
        True, rounds=2, seed=seed)
139     mask_datagen_val.fit(Y_train[int(Y_train.shape[0]*train_split):],
        augment=True, rounds=2, seed=seed)
140
141     x_val=image_datagen_val.flow(X[int(X.shape[0]*train_split):],
        batch_size=BATCH_SIZE,shuffle=True, seed=seed)
142     y_val=mask_datagen_val.flow(Y_train[int(Y_train.shape[0]*
        train_split):], batch_size=BATCH_SIZE,shuffle=True, seed=seed)
143
144     return x, y, x_val, y_val
145
146 #### augment data
147 x, y, x_val, y_val = augment_data(X_train_centered, Y_train)
```

```
148
149
150 # In[ ]:
151
152
153 # Checking if the images fit
154
155 imshow(np.squeeze(x.next()[0]))
156 plt.show()
157 imshow(np.squeeze(y.next()[0]))
158 plt.show()
159 imshow(np.squeeze(x_val.next()[0]))
160 plt.show()
161 imshow(np.squeeze(y_val.next()[0]))
162 plt.show()
163
164
165 # In[ ]:
166
167
168 # creating a training and validation generator
169 # that generate masks and images
170 train_generator = zip(x, y)
171 val_generator = zip(x_val, y_val)
172
173
174 # # Create our Keras metric
175 #
176
177 # In[ ]:
```

```
178
179
180 # Define IoU metric
181
182 def mean_iou(y_true, y_pred):
183     prec = []
184     for t in np.arange(0.5, 1.0, 0.05):
185         y_pred_ = tf.to_int32(y_pred > t)
186         score, up_opt = tf.metrics.mean_iou(y_true, y_pred_, 2)
187         K.get_session().run(tf.local_variables_initializer())
188         with tf.control_dependencies([up_opt]):
189             score = tf.identity(score)
190         prec.append(score)
191     return K.mean(K.stack(prec), axis=0)
192
193
194 # from https://www.kaggle.com/bguberfain/naive-keras
195 smooth = 1.
196
197 # From here: https://github.com/jocicmarko/ultrasound-nerve-
    segmentation/blob/master/train.py
198 def dice_coef(y_true, y_pred):
199     y_true_f = K.flatten(y_true)
200     y_pred_f = K.flatten(y_pred)
201     intersection = K.sum(y_true_f * y_pred_f)
202     return (2. * intersection + smooth) / (K.sum(y_true_f) + K.sum(
    y_pred_f) + smooth)
203
204
205 def bce_dice_loss(y_true, y_pred):
```

```
206     return 0.5 * binary_crossentropy(y_true, y_pred) - dice_coef(
        y_true, y_pred)
207
208
209 # # Build and train our neural network
210 #
211 # Next we build our U-Net model, based on *U-Net: Convolutional
        Networks for Biomedical Image Segmentation* (https://arxiv.org/pdf
        /1505.04597.pdf).
212 #
213 # ![U-net Architecture](images/u-net-architecture.png)
214
215 # In[ ]:
216
217
218 # set IMG_CHANNELS to be either 3 or 1, appropriately for our data
219 IMG_CHANNELS = X_train_centered[0].shape[2]
220
221
222 # In[ ]:
223
224
225 # Build U-Net model
226 def get_unet(first_layer_size=64, act_fn='relu', dropout=0.0,
227              learning_rate=1e-3, opt_fn=optimizers.SGD):
228     '''
229     n determines the size of the first layer...equal to 2**n.
230         So, either 16, 32, or 64
231     '''
232     assert first_layer_size in [16, 32, 64]
```

```
233     assert act_fn in ['relu', 'leaky', 'elu']
234     if first_layer_size == 64:
235         n = 6
236     elif first_layer_size == 32:
237         n = 5
238     else:
239         n = 4
240     inputs = Input((IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))
241 
242     if act_fn != 'leaky':
243         c1 = Conv2D(2**n, (3, 3), input_shape=(IMG_HEIGHT, IMG_WIDTH),
     activation=act_fn, kernel_initializer='he_normal', padding='same')
    (inputs)
244         c1 = Conv2D(2**n, (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same')(c1)
245         p1 = MaxPooling2D((2, 2))(c1)
246     else:
247         c1 = Conv2D(2**n, (3, 3), input_shape=(IMG_HEIGHT, IMG_WIDTH),
     activation=None, kernel_initializer='he_normal', padding='same')(s
    )
248         c1 = LeakyReLU(alpha=0.1)(c1)
249         c1 = Conv2D(2**n, (3, 3), activation=None, kernel_initializer=
    'he_normal', padding='same')(c1)
250         c1 = LeakyReLU(alpha=0.1)(c1)
251         p1 = MaxPooling2D((2, 2))(c1)
252 
253     if act_fn != 'leaky':
254         c2 = Conv2D(2**(n+1), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (p1)
255         c2 = LeakyReLU(alpha=0.1)(c2)
```

```
256         c2 = Conv2D(2**(n+1), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (c2)
257         p2 = MaxPooling2D((2, 2)) (c2)
258     else:
259         c2 = Conv2D(2**(n+1), (3, 3), activation=None,
260                     kernel_initializer='he_normal', padding='same') (
    p1)
261         c2 = LeakyReLU(alpha=0.1)(c2)
262         c2 = Conv2D(2**(n+1), (3, 3), activation=None,
263                     kernel_initializer='he_normal', padding='same') (
    c2)
264         c2 = LeakyReLU(alpha=0.1)(c2)
265         p2 = MaxPooling2D((2, 2)) (c2)
266
267     if act_fn != 'leaky':
268         c3 = Conv2D(2**(n+2), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (p2)
269         c3 = Conv2D(2**(n+2), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (c3)
270         p3 = MaxPooling2D((2, 2)) (c3)
271     else:
272         c3 = Conv2D(2**(n+2), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (p2)
273         c3 = LeakyReLU(alpha=0.1)(c3)
274         c3 = Conv2D(2**(n+2), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (c3)
275         c3 = LeakyReLU(alpha=0.1)(c3)
276         p3 = MaxPooling2D((2, 2)) (c3)
277
278     if act_fn != 'leaky':
```

```
279         c4 = Conv2D(2**(n+3), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (p3)
280         c4 = Conv2D(2**(n+3), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (c4)
281         p4 = MaxPooling2D(pool_size=(2, 2)) (c4)
282     else:
283         c4 = Conv2D(2**(n+3), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (p3)
284         c4 = LeakyReLU(alpha=0.1)(c4)
285         c4 = Conv2D(2**(n+3), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (c4)
286         c4 = LeakyReLU(alpha=0.1)(c4)
287         p4 = MaxPooling2D(pool_size=(2, 2)) (c4)
288
289     if act_fn != 'leaky':
290         c5 = Conv2D(2**(n+4), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (p4)
291         c5 = Conv2D(2**(n+4), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (c5)
292     else:
293         c5 = Conv2D(2**(n+4), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (p4)
294         c5 = LeakyReLU(alpha=0.1)(c5)
295         c5 = Conv2D(2**(n+4), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (c5)
296         c5 = LeakyReLU(alpha=0.1)(c5)
297
298     if act_fn != 'leaky':
299         u6 = Conv2DTranspose(2**(n+3), (2, 2), strides=(2, 2), padding
    ='same') (c5)
```

```
300         u6 = concatenate([u6, c4])
301         c6 = Conv2D(2**(n+3), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (u6)
302         if dropout:
303             c6 = Dropout(dropout) (c6)
304         c6 = Conv2D(2**(n+3), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (c6)
305     else:
306         u6 = Conv2DTranspose(2**(n+3), (2, 2), strides=(2, 2), padding
    ='same') (c5)
307         u6 = concatenate([u6, c4])
308         c6 = Conv2D(2**(n+3), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (u6)
309         c6 = LeakyReLU(alpha=0.1)(c6)
310         if dropout:
311             c6 = Dropout(dropout) (c6)
312         c6 = Conv2D(2**(n+3), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (c6)
313         c6 = LeakyReLU(alpha=0.1)(c6)
314
315     if act_fn != 'leaky':
316         u7 = Conv2DTranspose(2**(n+2), (2, 2), strides=(2, 2), padding
    ='same') (c6)
317         u7 = concatenate([u7, c3])
318         c7 = Conv2D(2**(n+2), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (u7)
319         if dropout:
320             c7 = Dropout(dropout) (c7)
321         c7 = Conv2D(2**(n+2), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (c7)
```

```
322     else:
323         u7 = Conv2DTranspose(2**(n+2), (2, 2), strides=(2, 2), padding
    ='same') (c6)
324         u7 = concatenate([u7, c3])
325         c7 = Conv2D(2**(n+2), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (u7)
326         c7 = LeakyReLU(alpha=0.1)(c7)
327         if dropout:
328             c7 = Dropout(dropout) (c7)
329         c7 = Conv2D(2**(n+2), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (c7)
330         c7 = LeakyReLU(alpha=0.1)(c7)
331
332     if act_fn != 'leaky':
333         u8 = Conv2DTranspose(2**(n+1), (2, 2), strides=(2, 2), padding
    ='same') (c7)
334         u8 = concatenate([u8, c2])
335         c8 = Conv2D(2**(n+1), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (u8)
336         if dropout:
337             c8 = Dropout(dropout) (c8)
338         c8 = Conv2D(2**(n+1), (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (c8)
339     else:
340         u8 = Conv2DTranspose(2**(n+1), (2, 2), strides=(2, 2), padding
    ='same') (c7)
341         u8 = concatenate([u8, c2])
342         c8 = Conv2D(2**(n+1), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (u8)
343         c8 = LeakyReLU(alpha=0.1)(c8)
```

```
344         if dropout:
345             c8 = Dropout(dropout) (c8)
346         c8 = Conv2D(2**(n+1), (3, 3), activation=None,
    kernel_initializer='he_normal', padding='same') (c8)
347         c8 = LeakyReLU(alpha=0.1)(c8)
348 
349     if act_fn != 'leaky':
350         u9 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same
    ') (c8)
351         u9 = concatenate([u9, c1], axis=3)
352         c9 = Conv2D(2**n, (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (u9)
353         if dropout:
354             c9 = Dropout(dropout) (c9)
355         c9 = Conv2D(2**n, (3, 3), activation=act_fn,
    kernel_initializer='he_normal', padding='same') (c9)
356     else:
357         u9 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same
    ') (c8)
358         u9 = concatenate([u9, c1], axis=3)
359         c9 = Conv2D(2**n, (3, 3), activation=None, kernel_initializer=
    'he_normal', padding='same') (u9)
360         c9 = LeakyReLU(alpha=0.1)(c9)
361         if dropout:
362             c9 = Dropout(dropout) (c9)
363         c9 = Conv2D(2**n, (3, 3), activation=None, kernel_initializer=
    'he_normal', padding='same') (c9)
364         c9 = LeakyReLU(alpha=0.1)(c9)
365 
366     outputs = Conv2D(1, (1, 1), activation='sigmoid') (c9)
```

```
367
368     model = Model(inputs=[inputs], outputs=[outputs])
369
370     if opt_fn.__class__ == optimizers.SGD:
371         optimizer = opt_fn(lr=learning_rate, momentum=.99, nesterov=
    True)
372     elif opt_fn.__class__ == optimizers.Adam:
373         optimizer = opt_fn(lr=learning_rate)
374     elif opt_fn.__class__ == optimizers.RMSprop:
375         optimizer = opt_fn(lr=learning_rate)
376     else:
377         optimizer = opt_fn()
378     model.compile(optimizer=optimizer, loss=bce_dice_loss, metrics=[
    mean_iou])
379     #model.summary()
380     return model
381
382
383 # Next we fit the model on the augmented training data. Run model for
    a max of 1000 epochs, but use early stopping with a patience of 30.
384 #
385 # * Use Nesterov SGD with momentum set to 0.99.
386 # * Dropout on the up conv layers with dropout rate 0.1.
387 # * ReLU activation functions
388 # * Learning rate 3e-7
389 # * First layer size 64
390 # * Validation size 10% of training data
391
392 # In[ ]:
393
```

```
394
395 # Fit model
396 model_number = 1
397 model = get_unet(first_layer_size=64, act_fn='relu', dropout=0.1,
        learning_rate=3e-7, opt_fn=optimizers.SGD)
398
399 earlystopper = EarlyStopping(patience=30, verbose=1)
400 checkpointer = ModelCheckpoint('model-dsbowl2018-{}.h5'.format(
        model_number), verbose=1, save_best_only=True)
401
402 results = model.fit_generator(train_generator, validation_data=
        val_generator, validation_steps=10, verbose=1, steps_per_epoch
        =5000, epochs=1000, callbacks=[earlystopper, checkpointer])
403
404
405 # # Make predictions
406 #
407 # Let's make predictions both on the test set, the val set and the
        train set (as a sanity check).
408
409 # In[ ]:
410
411
412 # Predict on train, val and test
413 model = load_model('model-dsbowl2018-{}.h5'.format(model_number),
        custom_objects={'mean_iou': mean_iou, 'dice_coef': dice_coef, '
        bce_dice_loss': bce_dice_loss,})
414 preds_train = model.predict(X_train_centered[:int(X_train_centered.
        shape[0]*0.9)], verbose=1)
```

```
415 preds_val = model.predict(X_train_centered[int(X_train_centered.shape
        [0]*0.9):], verbose=1)
416 preds_test = model.predict(X_test_centered, verbose=1)
417
418 # Threshold predictions
419 preds_train_t = (preds_train > 0.5).astype(np.uint8)
420 preds_val_t = (preds_val > 0.5).astype(np.uint8)
421 preds_test_t = (preds_test > 0.5).astype(np.uint8)
422
423 # Create list of upsampled test masks
424 preds_test_upsampled = []
425 for i in range(len(preds_test)):
426     preds_test_upsampled.append(resize(np.squeeze(preds_test[i]), (
        sizes_test[i][0], sizes_test[i][1]), mode='constant',
        preserve_range=True))
427
428
429 # In[ ]:
430
431
432 # Perform a sanity check on some random training samples
433 ix = random.randint(0, len(preds_train_t))
434 imshow(np.squeeze(X_train_centered[ix]))
435 plt.show()
436 imshow(np.squeeze(Y_train[ix]))
437 plt.show()
438 imshow(np.squeeze(preds_train_t[ix]))
439 plt.show()
440
441
```

```
# In[ ]:


# Perform a sanity check on some random validation samples
ix = random.randint(0, len(preds_val_t))
imshow(np.squeeze(X_train[int(X_train.shape[0]*train_split):][ix]))
plt.show()
imshow(np.squeeze(Y_train[int(Y_train.shape[0]*train_split):][ix]))
plt.show()
imshow(np.squeeze(preds_val_t[ix]))
plt.show()


# # Encode and submit our results
#
# Now it's time to submit our results. I've stolen this (https://www.
    kaggle.com/rakhlin/fast-run-length-encoding-python) excellent
    implementation of run-length encoding.

# In[ ]:


# Run-length encoding stolen from https://www.kaggle.com/rakhlin/fast-
    run-length-encoding-python
def rle_encoding(x):
    dots = np.where(x.T.flatten() == 1)[0]
    run_lengths = []
    prev = -2
    for b in dots:
        if (b>prev+1): run_lengths.extend((b + 1, 0))
```

```
            run_lengths[-1] += 1
            prev = b
    return run_lengths

def prob_to_rles(x, cutoff=0.5):
    lab_img = label(x > cutoff)
    for i in range(1, lab_img.max() + 1):
        yield rle_encoding(lab_img == i)



# Let's iterate over the test IDs and generate run-length encodings
    for each seperate mask identified by skimage ...

# In[ ]:


new_test_ids = []
rles = []
for n, id_ in enumerate(test_ids):
    rle = list(prob_to_rles(preds_test_upsampled[n]))
    rles.extend(rle)
    new_test_ids.extend([id_] * len(rle))



# ... and then finally create our submission!

# In[ ]:

```

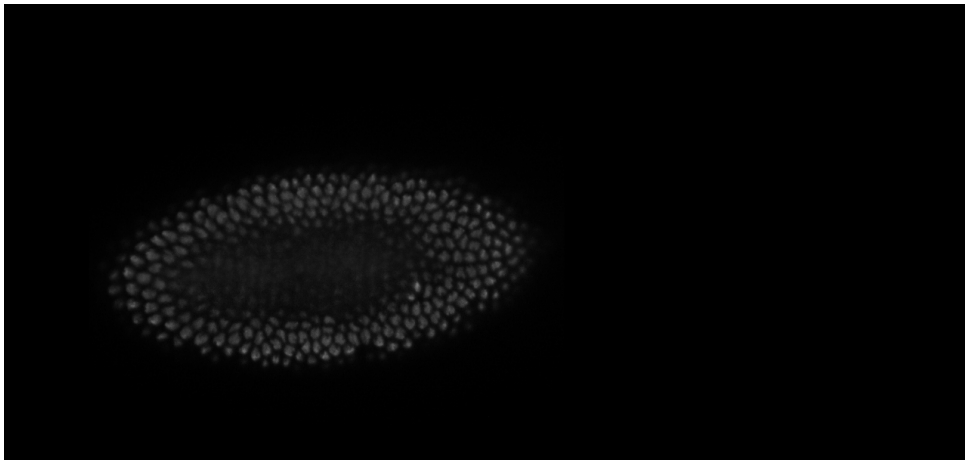
```
498
499 # Create submission DataFrame
500 sub = pd.DataFrame()
501 sub['ImageId'] = new_test_ids
502 sub['EncodedPixels'] = pd.Series(rles).apply(lambda x: ' '.join(str(y)
      for y in x))
503 sub.to_csv('submission-{}.csv'.format(model_number), index=False)
```

Listing B.1: Model Training, Prediction, and Submission Generator

Appendix C

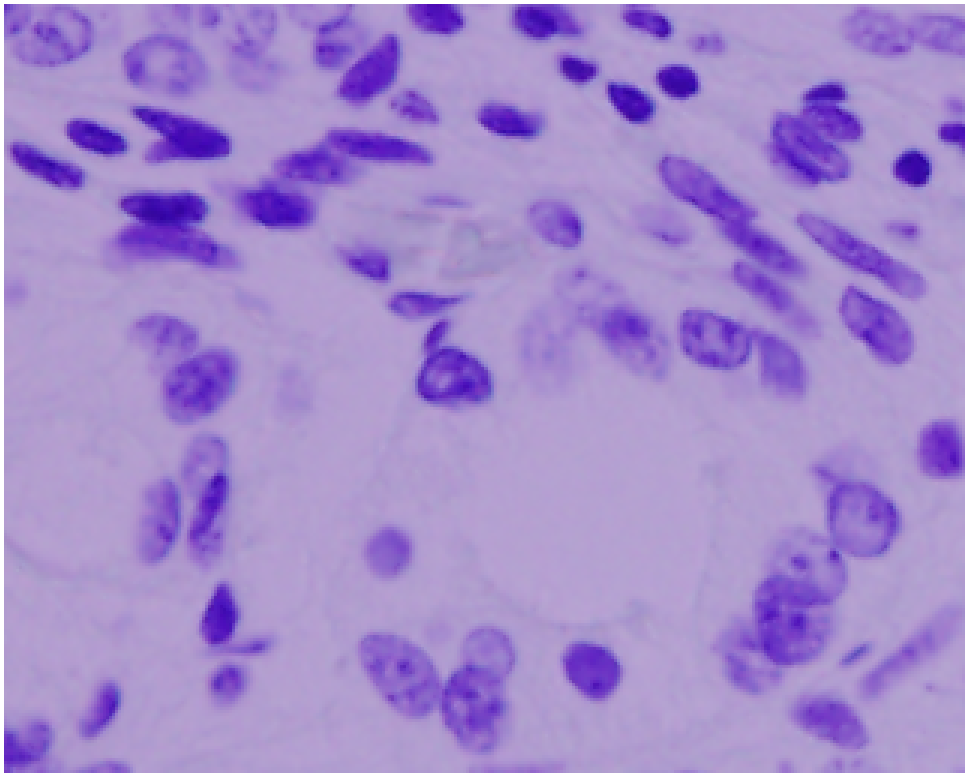
Various Types of Input Images

1. Black Ring

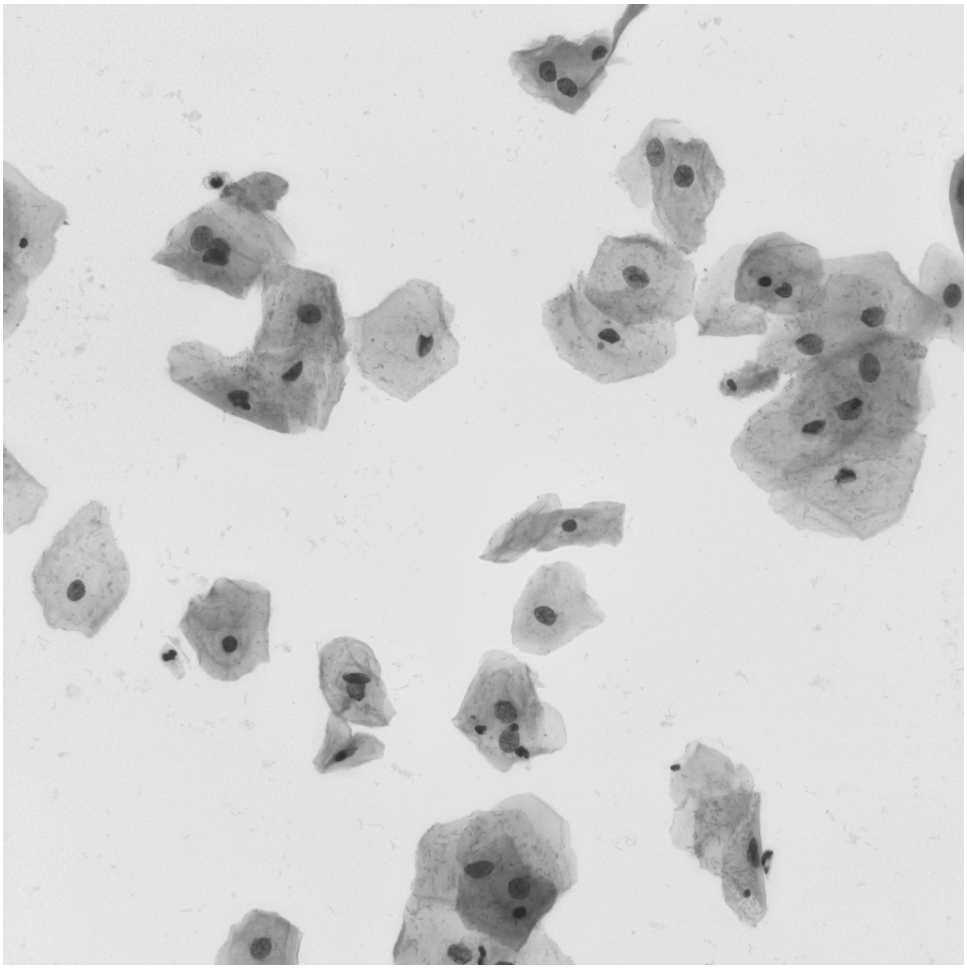


2. Almost nothing there

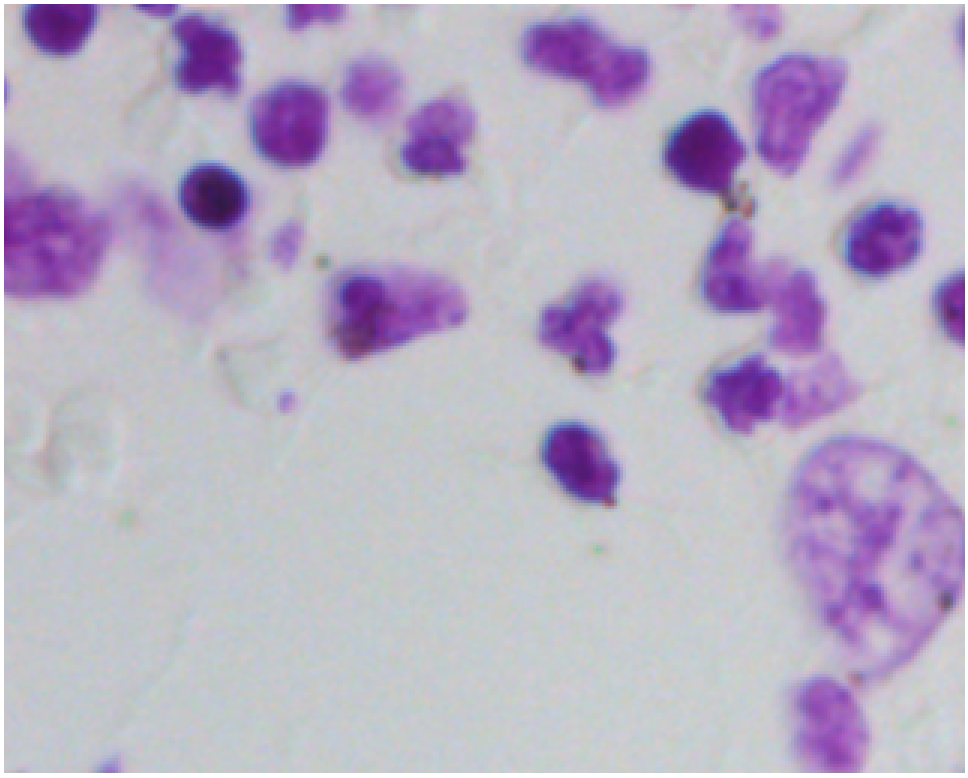
3. Purple with purple background



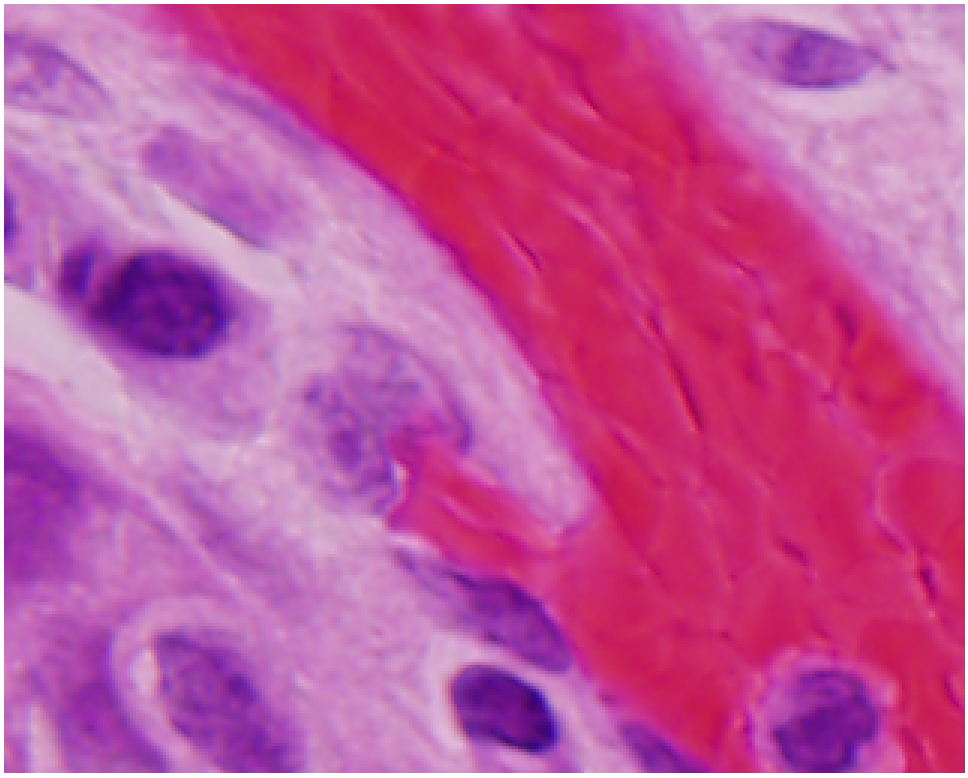
4. Gray little dots with rings and light background



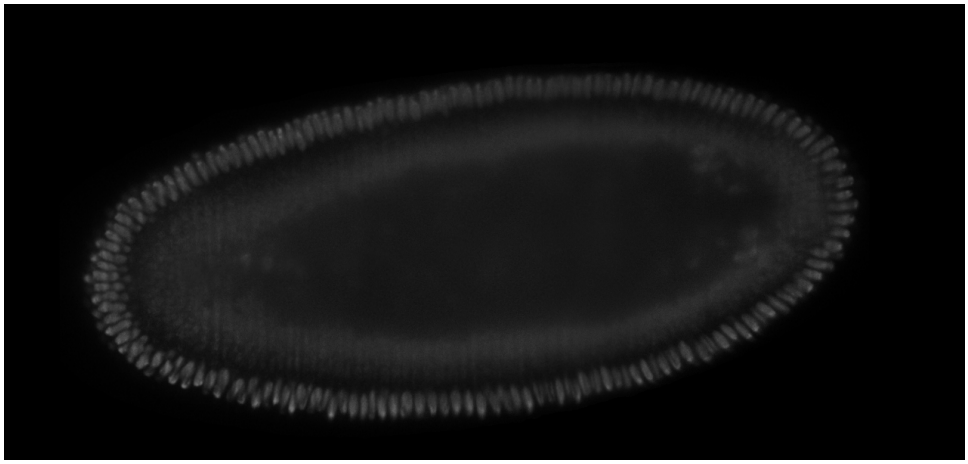
5. Purple with white background



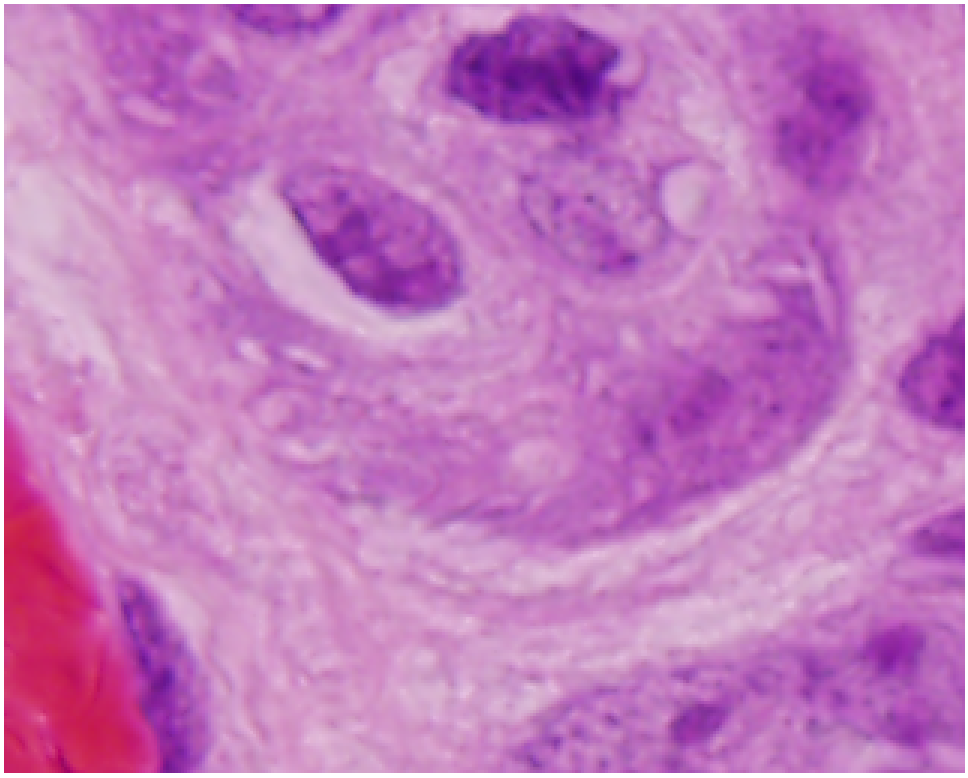
6. Purple and red



7. Ring but inside is nuclei



8. Red and purple; way different than most



Appendix D

Example SLURM Code

```
#!/bin/bash
#SBATCH --job-name="test_1"
#SBATCH --partition=GPU-shared
#SBATCH --gres=gpu:p100:2
#SBATCH --nodes=1
#SBATCH --tasks-per-node=16
#SBATCH --time=48:00:00
#SBATCH --mail-user=srlyche@cpp.edu
#SBATCH --mail-type=BEGIN,END,FAIL
#SBATCH --workdir="/pylon5/cc560sp/lyche32/data_science_bowl"
#SBATCH --output=test_1_%j.o
#SBATCH --error=test_1_%j.e
```

```
# load anaconda5 module
module load anaconda5/5.0.0-3.6

# change to data_science_bowl directory
cd /pylon5/cc560sp/lyche32/data_science_bowl

# activate your conda env
source activate samuel

# run python code
python keras_1.py
```

Appendix E

Python Code to Create Dual Set

```
import numpy as np
from Dual_Matrix import Dual_Matrix

def create_dual_set(X):
    '''
        For use with ndarry of shape
            (NUM_IMAGES, IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS)

        IMG_HEIGHT, IMG_WIDTH, and IMG_CHANNELS were global variables
        in the script that this was used in.
            They could be replaced with X.shape[1], X.shape[2],
            and X.shape[3], respectively
    '''

    # instantiate the dual matrix with an extra row and
    # column than the original
    X_out = np.zeros((X.shape[0], IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS)
    )

```

```
19      # loop over images and channels and insert the dual into X_out
20      for i in range(len(X)):
21          for c in range(IMG_CHANNELS):
22              X_out[i,:,:,c] = Dual_Matrix(X[i,:,:,c]).get_sub_dual()
23
24      return X_out
```

Listing E.1: Create a NumPy ndarray of Dual Matrices

Appendix F

Dual Matrix Python Code

```
#! /usr/bin/python3

from scipy.linalg import toeplitz
import numpy as np

# as a class

class Dual_Matrix():

    def __init__(self, M):
        # get the dimension of the matrix M
        self.M = M
        self.n = M.shape[0]


    def get_R(self):
        '''
        Input
        _____
```

```
20            n : the dimension of the original matrix M
21
22        Output
23        ------
24            R : the matrix to perform the row operations
25        '''
26        '''
27        # this might now be as efficient as using scipy function scipy
    .linalg.toeplitz as below
28        R = s.eye(n)
29        for i in range(n-1):
30            R[i,i+1] = -1
31        '''
32        first_col = np.hstack((1, np.zeros(self.n-1, dtype=int)))  #
    dtype=int b/c default dtype=np.float64, and sympy was acting weird
33        first_row = np.hstack((1, -1, np.zeros(self.n-2, dtype=int)))
    # dtype=int b/c default dtype=np.float64, and sympy was acting
    weird
34        R = toeplitz(c=first_col, r=first_row)
35
36
37        return R
38
39    def get_C(self):
40        '''
41        Input
42        -----
43            n : the dimension of the original matrix M
44
45        Output
```

```
46          ------
47              C : the matrix to perform the column operations
48          '''
49          '''
50          # this might now be as efficient as using scipy function scipy
    .linalg.toeplitz as below
51          C = s.ones(n)  # to get the upper triangular portion to be all
     ones
52
53          # to replace the lower triangular portion to be all zeros
54          for i in range(1, n):
55              for j in range(i):
56                  C[i,j] = 0
57          '''
58          first_col = np.hstack((1, np.zeros(self.n-1, dtype=int))) #
    dtype=int b/c default dtype=np.float64, and sympy was acting weird
59          first_row = np.ones(self.n, dtype=int) # dtype=int b/c default
     dtype=np.float64, and sympy was acting weird
60          C = toeplitz(c=first_col, r=first_row)
61
62          return C
63
64      def get_sub_dual(self):
65          '''
66          Input
67          -----
68              M : an nxn matrix
69
70          Output
71          ------
```

```
72            M_sub_star : an nxn matrix
73                the sub dual matrix of M
74        '''
75        C = self.get_C()
76        R = self.get_R()
77
78        M_sub_star = R.dot(self.M.T).dot(C)
79
80        return M_sub_star
81
82    def get_dual(self):
83        '''
84        Input
85        _____
86            M : an nxn matrix
87
88        Output
89        ______
90            M_star : an (n+1)x(n+1) matrix
91                the dual matrix of the matrix M
92
93            x : scalar
94                the scale value to put M_star values in the range
    [-1,1]
95        '''
96        # get the sub dual matrix... will be nxn
97        M_sub_star = self.get_sub_dual()
98
99        # get the row sums and the max abs value of the row sums
100       row_sums = np.sum(M_sub_star,axis=1)
```

```
101         max_idx = np.argmax(np.abs(row_sums))
102         x = row_sums[max_idx]+2 if row_sums[max_idx] >= 0 else
    row_sums[max_idx]-2
103
104         # create the extra column and row that are needed to create
    the dual
105         extra_col = np.reshape(x - row_sums, (self.n,1))
106         extra_row = np.hstack((np.zeros(self.n), x))
107
108         # create the dual matrix
109         M_star = np.hstack((M_sub_star, extra_col))
110         M_star = np.vstack((M_star, extra_row))
111
112         return M_star
113
114     def get_scaled_dual(self):
115         '''
116         Input
117         -----
118             M : an nxn matrix
119
120         Output
121         ------
122             M_star : an (n+1)x(n+1) matrix
123                 the dual matrix of the matrix M
124
125             x : scalar
126                 the scale value to put M_star values in the range
    [-1,1]
127         '''
```

```
128         # get the sub dual matrix... will be nxn
129         M_sub_star = self.get_sub_dual()
130 
131         # get the row sums and the max abs value of the row sums
132         row_sums = np.sum(M_sub_star,axis=1)
133         max_idx = np.argmax(np.abs(row_sums))
134         x = row_sums[max_idx]+1 if row_sums[max_idx] >= 0 else
    row_sums[max_idx]-1
135 
136         # create the extra column and row that are needed to create
    the dual
137         extra_col = np.reshape(x - row_sums, (self.n,1))
138         extra_row = np.hstack((np.zeros(self.n), x))
139 
140         # create the dual matrix
141         M_star = np.hstack((M_sub_star, extra_col))
142         M_star = np.vstack((M_star, extra_row))
143 
144         # scale M_star by X to make in range [-1,1]
145         M_star /= x
146 
147         return M_star, x
148 
149     def get_scaled_sub_dual(self):
150         return self.get_scaled_dual()[0][:self.n,:self.n]
```

Listing F.1: Dual Matrix Class

Appendix G

Dual Matrices

G.1 The 2×2 Case

Given a matrix M, we can find its dual matrix M^*. For a 2×2 real matrix, this transformation is like so:

Let

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \tag{G.1}$$

Then, the dual matrix, M^*, can be constructed by first augmenting the matrix M to what I will call $\bar{M}$ like so

$$\bar{M} = \begin{bmatrix} 0 & 0 & 0 \\ m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ 0 & 0 & x \end{bmatrix} \tag{G.2}$$

where $x \in \mathbb{R}$.

The dual matrix M^* is then constructed according to the following rule:

$$M^*_{ij} = \sum_{k=i}^{n} \bar{M}_{j+1,k} - \sum_{k=i}^{n} \bar{M}_{j,k} \tag{G.3}$$

for $i = 1, ..., n+1$ where n is the dimension of is matrix M. This results in the dimensions of M^* being $(n+1) \times (n+1)$.

Given the 2×2 M that we have above, we would get

$$M^* = \begin{bmatrix} m_{11}+m_{12} & (m_{21}+m_{22})-(m_{11}+m_{12}) & x-(m_{21}+m_{22}) \\ m_{12} & m_{22}-m_{12} & x-m_{22} \\ 0 & 0 & x \end{bmatrix} \tag{G.4}$$

We want to show that the eigenvalues for the matrix M and the submatrix of M^* defined by $M^*_{1:n,1:n}$ are equal to each other.

For the 2×2 case. This can be shown as follows:

Take the submatrix of M^*, let us call it $\ddot{M}$.

That is,

$$\ddot{M} = \begin{bmatrix} m_{11}+m_{12} & (m_{21}+m_{22})-(m_{11}+m_{12}) \\ m_{12} & m_{22}-m_{12} \end{bmatrix} \tag{G.5}$$

We also want to show that the eigenvalues are the same, so we want the equivalent of

$$det \begin{bmatrix} m_{11}-\lambda & m_{12} \\ m_{21} & m_{22}-\lambda \end{bmatrix} = 0 \tag{G.6}$$

The task then is to show that

$$det \begin{bmatrix} m_{11}+m_{12}-\lambda & (m_{21}+m_{22})-(m_{11}+m_{12}) \\ m_{12} & m_{22}-m_{12}-\lambda \end{bmatrix} \tag{G.7}$$

is equivalent to the left hand side of Equation G.6.

To do so, let us perform some elementary row and column operations on it, which will preserve eigenvalues. First, add column 1 to column 2 and put the result into column 2. We will denote this as $C_1 = \mathfrak{C}_1$, $\mathfrak{C}_1 + C_2 = \mathfrak{C}_2$, and, in general, $\mathfrak{C}_{j-1} + C_j = \mathfrak{C}_j$ for $j = 2, ..., n$. The result is then

$$\begin{bmatrix} m_{11} + m_{12} - \lambda & m_{21} + m_{22} - \lambda \\ m_{12} & m_{22} - \lambda \end{bmatrix} \tag{G.8}$$

Next, subtract row 2 from row from row 1 and put the result into row 1. We will denote this as $R_1 - R_2 = \mathfrak{R}_1$, $R_2 - R3 = \mathfrak{R}_2$, and, in general, $R_i - R_{i+1} = \mathfrak{R}_i$ for $i = 1, ..., n-1$.

The result is

$$\begin{bmatrix} m_{11} - \lambda & m_{21} \\ m_{12} & m_{22} - \lambda \end{bmatrix} \tag{G.9}$$

Notice that the matrix in G.9 is just the transpose of the matrix in G.6. Since a matrix and its transpose have the same eigenvalues, we have shown that the result holds for the 2×2 case.

G.2 The 3×3 Case

Let

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \tag{G.10}$$

This gives us our $\bar{M}$, M^*, and $\ddot{M}$ as follows:

$$\bar{M} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ 0 & 0 & 0 & x \end{bmatrix} \tag{G.11}$$

$$M^*_{row1} = \begin{bmatrix} m_{11}+m_{12}+m_{13} \\ (m_{21}+m_{22}+m_{23})-(m_{11}+m_{12}+m_{13}) \\ (m_{31}+m_{32}+m_{33})-(m_{21}+m_{22}+m_{23}) \\ x-(m_{31}+m_{32}+m_{33}) \end{bmatrix}^T \tag{G.12}$$

$$M^*_{row2} = \begin{bmatrix} m_{12}+m_{13} \\ (m_{22}+m_{23})-(m_{12}+m_{13}) \\ (m_{32}+m_{33})-(m_{22}+m_{23}) \\ x-(m_{32}+m_{33}) \end{bmatrix}^T \tag{G.13}$$

$$M^*_{row3} = \begin{bmatrix} m_{13} \\ m_{23}-m_{13} \\ m_{33}-m_{23} \\ x-m_{33} \end{bmatrix}^T \tag{G.14}$$

$$M^*_{row4} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ x \end{bmatrix}^T \tag{G.15}$$

Alternative representation:

$$M^* = \begin{bmatrix} \sum_{j=1:3} m_{1j} & \sum_{j=1:3} m_{2j} - \sum_{j=1:3} m_{1j} & \sum_{j=1:3} m_{3j} - \sum_{j=1:3} m_{2j} & x - \sum_{j=1:3} m_{3j} \\ \sum_{j=2:3} m_{1j} & \sum_{j=2:3} m_{2j} - \sum_{j=2:3} m_{1j} & \sum_{j=2:3} m_{3j} - \sum_{j=2:3} m_{2j} & x - \sum_{j=2:3} m_{3j} \\ m_{13} & m_{23} - m_{13} & m_{33} - m_{23} & x - m_{33} \\ 0 & 0 & 0 & x \end{bmatrix} \tag{G.16}$$

This gives

$$\ddot{M} = \begin{bmatrix} \sum_{j=1:3} m_{1j} & \sum_{j=1:3} m_{2j} - \sum_{j=1:3} m_{1j} & \sum_{j=1:3} m_{3j} - \sum_{j=1:3} m_{2j} \\ \sum_{j=2:3} m_{1j} & \sum_{j=2:3} m_{2j} - \sum_{j=2:3} m_{1j} & \sum_{j=2:3} m_{3j} - \sum_{j=2:3} m_{2j} \\ m_{13} & m_{23} - m_{13} & m_{33} - m_{23} \end{bmatrix} \tag{G.17}$$

To reduce this to the same matrix as G.11, we apply a similar algorithm as we did to the 2×2 version.

The steps of the algorithm are as follows:

1. $C_1 = \mathfrak{C}_1$
2. $\mathfrak{C}_1 + C_2 = \mathfrak{C}_2$
3. $\mathfrak{C}_2 + C_3 = \mathfrak{C}_3$
4. $R_1 - R_2 = \mathfrak{R}_1$
5. $R_2 - R_3 = \mathfrak{R}_2$

Let us see these steps in action.

1. let $C_1 = \mathfrak{C}_1$. Then, $\mathfrak{C}_1 + C_2 = \mathfrak{C}_2$

$$M^*1 = \begin{bmatrix} \sum_{j=1:3} m_{1j} & \sum_{j=1:3} m_{2j} & \sum_{j=1:3} m_{3j} - \sum_{j=1:3} m_{2j} \\ \sum_{j=2:3} m_{1j} & \sum_{j=2:3} m_{2j} & \sum_{j=2:3} m_{3j} - \sum_{j=2:3} m_{2j} \\ m_{13} & m_{23} & m_{33} - m_{23} \end{bmatrix} \tag{G.18}$$

2. $\mathfrak{C}_2 + C_3 = \mathfrak{C}_3$

$$M^*2 = \begin{bmatrix} m_{11} + m_{12} + m_{13} & m_{21} + m_{22} + m_{23} & m_{31} + m_{32} + m_{33} \\ m_{12} + m_{13} & m_{22} + m_{23} & m_{32} + m_{33} \\ m_{13} & m_{23} & m_{33} \end{bmatrix} \tag{G.19}$$

3. $R_1 - R_2 = \mathfrak{R}_1$

$$M^*3 = \begin{bmatrix} m_{11} & m_{21} & m_{31} \\ m_{12} + m_{13} & m_{22} + m_{23} & m_{32} + m_{33} \\ m_{13} & m_{23} & m_{33} \end{bmatrix} \tag{G.20}$$

4. $R_2 - R_3 = \mathfrak{R}_2$

$$M^*4 = \begin{bmatrix} m_{11} & m_{21} & m_{31} \\ m_{12} & m_{22} & m_{32} \\ m_{13} & m_{23} & m_{33} \end{bmatrix} \tag{G.21}$$

Notice that M^*4 is just M^T. Thus, these two matrices have the same eigenvalues.

G.3 The $n \times n$ Case

We want to show that after some elementary column and row operations, we can transform the subset of the dual matrix M^* defined by $M^*(1:n, 1:n)$ into the transpose of the original matrix M. Thus, we will have shown that the eigenvalues for the original system and the dual system are equal.

Consider the $n \times n$ matrix

$$M = \begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{bmatrix} \tag{G.22}$$

This gives the augmented matrix to be

$$\bar{M} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ m_{11} & m_{12} & \cdots & m_{1n} & 0 \\ m_{21} & m_{22} & \cdots & m_{2n} & 0 \\ \vdots & \vdots & \ddots & \vdots & 0 \\ m_{n1} & m_{n2} & \cdots & m_{nn} & 0 \\ 0 & 0 & 0 & 0 & x \end{bmatrix} \tag{G.23}$$

but we shall assume that the rows of the augmented matrix is zero indexed. That is, the first entry would be labeled as $\bar{M}_{01}$, rather than $\bar{M}_{11}$. This will allow us to only deal with the matrices M and M^*.

The dual matrix has entries

$$M^*_{ij} = \sum_{k=i}^{n} M_{j+1,k} - \sum_{k=i}^{n} M_{j,k} \tag{G.24}$$

but we will only concern ourselves with the submatrix $M^*(1:n,1:n)$ which we will call M^*.

If we successively apply the elementary column operations described by $C_1 = \mathfrak{C}_1$ and $\mathfrak{C}_{j-1} + C_j = \mathfrak{C}_j$ for $j = 2,...,n$, the result is a matrix, $M^{(2)}$ with columns defined by

$$M^{(2)}(:,j) = \mathfrak{C}_j = \begin{bmatrix} \sum_{k=1}^{n} m_{jk} \\ \sum_{k=2}^{n} m_{jk} \\ \vdots \\ \sum_{k=n}^{n} m_{jk} \end{bmatrix} \tag{G.25}$$

which can be alternately written as

$$M^{(2)}(i,j) = \mathfrak{C}_j = \begin{bmatrix} \sum_{k=i}^{n} m_{jk} \end{bmatrix} \tag{G.26}$$

for $i = 1,\ldots,n$.

Then, we apply elementary row operations $R_i - R_{i+1} = \mathfrak{R}_i$ for $i = 1,\ldots,n-1$.

For the first row, we get

$$[R_1 - R_2] = [\mathfrak{R}_1] = [M^{(2)}(1,j) - M^{(2)}(2,j)] \tag{G.27}$$

$$= \sum_{k=1}^{n} m_{jk} - \sum_{k=2}^{n} m_{jk} \tag{G.28}$$

$$= [m_{1j}] \tag{G.29}$$

For the second row, we get

$$[R_2 - R_3] = [\mathfrak{R}_2] = [M^{(2)}(2,j) - M^{(2)}(3,j)] \tag{G.30}$$

$$= \sum_{k=2}^{n} m_{jk} - \sum_{k=3}^{n} m_{jk} \tag{G.31}$$

$$= [m_{2j}] \tag{G.32}$$

For the i^{th} row and j^{th} column, that is, the ij entry of what we will call the resultant matrix M_R, we get

$$M_R(i,j) = \sum_{k=i}^{n} m_{jk} - \sum_{k=i+1}^{n} m_{jk} \tag{G.33}$$

$$= m_{ji} \tag{G.34}$$

Since the ij entry of the resultant matrix M_R is equal the the ji entry of the original matrix M, we have shown that $M_R = M^T$.

We know that a square matrix and its transpose have the same eigenvalues. QED.

A square matrix and its transpose have the same eigenvalues. Proof: $det(A - \lambda I) = det((A - \lambda I)^T) = det(A^T - \lambda I)$ QED.

G.4 Reversing the Algorithm Works, too

What if we start by applying the row operations and then follow them with the column operations? Does this still result in transforming M^* into M^T? Turns out it does work. So, mathematically, the order of the algorithms, columns first or rows first, does not matter. Perhaps there will be some difference in the speed of the algorithm on the computer, though.

Consider the 3×3 version of M^*:

$$M^* = \begin{bmatrix} \sum_{j=1:3} m_{1j} & \sum_{j=1:3} m_{2j} - \sum_{j=1:3} m_{1j} & \sum_{j=1:3} m_{3j} - \sum_{j=1:3} m_{2j} \\ \sum_{j=2:3} m_{1j} & \sum_{j=2:3} m_{2j} - \sum_{j=2:3} m_{1j} & \sum_{j=2:3} m_{3j} - \sum_{j=2:3} m_{2j} \\ m_{13} & m_{23} - m_{13} & m_{33} - m_{23} \end{bmatrix} \tag{G.35}$$

If we apply the row operations first $R_1 - R2 = \mathfrak{R}_1$ we get

$$M^{(2)} = \begin{bmatrix} m_{11} & m_{21} - m_{11} & m_{31} - m_{21} \\ m_{12} + m_{13} & (m_{22} + m_{23}) - (m_{12} + m_{13}) & (m_{32} + m_{33}) - (m_{22} + m_{23}) \\ m_{13} & m_{23} - m_{13} & m_{33} - m_{23} \end{bmatrix} \tag{G.36}$$

Then $R_2 - R_3 = \mathfrak{R}_2$, we get

$$M^{(2)} = \begin{bmatrix} m_{11} & (m_{21} - m_{11}) & (m_{31} - m_{21}) \\ m_{12} & (m_{22} - m_{12}) & (m_{32} - m_{22}) \\ m_{13} & (m_{23} - m_{13}) & (m_{33} - m_{23}) \end{bmatrix} \tag{G.37}$$

Now, we apply the column operations. First, $C_1 + C_2 = \mathfrak{C}_2$:

$$M^{(3)} = \begin{bmatrix} m_{11} & m_{21} & (m_{31} - m_{21}) \\ m_{12} & m_{22} & (m_{32} - m_{22}) \\ m_{13} & m_{23} & (m_{33} - m_{23}) \end{bmatrix} \tag{G.38}$$

And finally, $\mathfrak{C}_2 + C_3 = \mathfrak{C}_3$:

$$M_R = \begin{bmatrix} m_{11} & m_{21} & m_{31} \\ m_{12} & m_{22} & m_{32} \\ m_{13} & m_{23} & m_{33} \end{bmatrix} \tag{G.39}$$

Which shows that $M_R = M^T$.

G.5 Rows First Algorithm for $n \times n$ Case

After transforming our matrix M to the dual and considering on the the entries $1:n, 1:n$ of M^*, we get rows that look like the following:

$$\begin{aligned}
R_1 &= \sum_{k=1}^{n} m_{1,k} \quad \sum_{k=1}^{n} m_{2,k} - \sum_{k=1}^{n} m_{1,k} \quad \cdots \quad \sum_{k=1}^{n} m_{j,k} - \sum_{k=1}^{n} m_{j-1,k} \quad \cdots \\
R_2 &= \sum_{k=2}^{n} m_{1,k} \quad \sum_{k=2}^{n} m_{2,k} - \sum_{k=2}^{n} m_{1,k} \quad \cdots \quad \sum_{k=2}^{n} m_{j,k} - \sum_{k=2}^{n} m_{j-1,k} \quad \cdots \\
&\vdots \\
R_i &= \sum_{k=i}^{n} m_{1,k} \quad \sum_{k=i}^{n} m_{2,k} - \sum_{k=i}^{n} m_{1,k} \quad \cdots \quad \sum_{k=i}^{n} m_{j,k} - \sum_{k=i}^{n} m_{j-1,k} \quad \cdots \\
R_{i+1} &= \sum_{k=i+1}^{n} m_{1,k} \quad \sum_{k=i+1}^{n} m_{2,k} - \sum_{k=i+1}^{n} m_{1,k} \quad \cdots \quad \sum_{k=i+1}^{n} m_{j,k} - \sum_{k=i+1}^{n} m_{j-1,k} \quad \cdots \\
&\vdots
\end{aligned}$$

This implies

$$R_i - R_{i+1} = \quad m_{1,i}\ (m_{2,i} - m_{1,i})\ \ldots\ (m_{j,i} - m_{j-1,i})\ \ldots\ \quad = \mathfrak{R}_{\mathfrak{i}} \tag{G.40}$$

Call this new matrix, after we have applied the row operations, $M^{(2)}$.

Now apply the column operations: $C_1 = \mathfrak{C}_1$ then $\mathfrak{C}_{j-1} + C_j = \mathfrak{C}_j$ for $j = 2, \ldots, n$.

We have that $M^{(2)}[i, j] = [m_{j,i} - m_{j-1,i}]$ for $j = 1, \ldots, n$ with the condition that $m_{0,i} = 0$ for all $i = 1, \ldots, n$.

Thus, our matrix looks like the following:

$$M^{(2)} = \begin{bmatrix} m_{11} & m_{21} - m_{11} & \ldots & m_{j,1} - m_{j-1,1} & \ldots \\ m_{12} & m_{22} - m_{12} & \ldots & m_{j,2} - m_{j-1,2} & \ldots \\ \vdots & \vdots & \ddots & \vdots & \ddots \\ m_{1i} & m_{2i} - m_{1i} & \ldots & m_{ji} - m_{j-1,i} & \ldots \\ \vdots & \vdots & \ddots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \\ \\ \\ \\ \\ \end{bmatrix} \tag{G.41}$$

Since $\mathfrak{C}_1 + C_2 = \mathfrak{C}_2 = [m_{2i}]$ for $i = 1, \ldots, n$, we have that, and if we assume that for $j - 1$ it holds that $\mathfrak{C}_{j-2} + C_{j-1} = \mathfrak{C}_{j-1} = [m_{j-1,i}]$ for $i = 1, \ldots, n$, we have

$$M_R[i, j] = (m_{ji} - m_{j-1,i}) + m_{j-1,i} \tag{G.42}$$

$$= m_{ji} \tag{G.43}$$

Thus, again we have that $M_R = M^T$. QED

G.6 Using Matrices

We can achieve the same results using a bit more linear algebra. We can combine all of the row operations into one matrix R and all of the column operations into a matrix C, rather than using the iterative notation previously discussed.

The matrix R, for the 3×3 case, looks like

$$R = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \tag{G.44}$$

This has the same effect as $R_i - R_{i+1} = \mathfrak{R}_i$.

The matrix C, for the 3×3 case, looks like

$$C = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tag{G.45}$$

This has the same effect as $\mathfrak{C}_{i-1} + C_i = \mathfrak{C}_i$, where $C_1 = \mathfrak{C}_1$.

We then can see that

$$R * M^* * C = M^T \tag{G.46}$$

It is interesting to note that $RC = CR = I$.

In general, the matrix R will have all ones on the main diagonal, all negative ones on the superdiagonal, otherwise zero. That is,

$$R[i,j] = \begin{cases} 1, & \text{if } i = j \\ -1, & \text{if } j = i+1 \\ 0, & \text{otherwise} \end{cases} \tag{G.47}$$

The matrix C is upper triangular with all ones in the upper region. That is

$$C[i,j] = \begin{cases} 1, & \text{if } j \geq i \\ 0, & \text{otherwise} \end{cases} \tag{G.48}$$

G.7 Visualizing $RC = CR = I$

Let consider the 3×3 case, for simplicity. Then

$$R = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \tag{G.49}$$

and

$$C = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tag{G.50}$$

$$RC = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tag{G.51}$$

Then, RC, step by step goes as follows:

$$RC[1,1] = R[1,:] * C[:,1] = 1(1) - 1(0) + 0(0) = 1$$

$$RC[1,2] = R[1,:] * C[:,2] = 1(1) - 1(1) + 0(0) = 0$$

$$RC[1,3] = R[1,:] * C[:,3] = 1(1) - 1(1) + 0(1) = 0$$

$$RC[2,1] = R[2,:] * C[:,1] = 0(0) + 1(0) - 1(0) = 0$$

$$RC[2,2] = R[2,:] * C[:,2] = 0(1) + 1(1) - 1(0) = 1$$

$$RC[2,3] = R[2,:] * C[:,3] = 0(1) + 1(1) - 1(1) = 0$$

$$RC[3,1] = R[3,:] * C[:,1] = 0(1) + 0(0) + 1(0) = 0$$

$$RC[3,2] = R[3,:] * C[:,2] = 0(1) + 0(1) + 1(0) = 0$$

$$RC[3,3] = R[3,:] * C[:,3] = 0(1) + 0(1) + 1(1) = 1$$

Thus, we see that only the entries where $i = j$ have value 1 and everywhere else equals zero, i.e., the identity matrix.

$$CR = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \tag{G.52}$$

For CR, step by step we have:

$$CR[1,1] = C[1,:] * R[:,1] = 1(1) + 1(0) + 1(0) = 1$$

$$CR[1,2] = C[1,:] * R[:,2] = 1(-1) + 1(1) + 1(0) = 0$$

$$CR[1,3] = C[1,:] * R[:,3] = 1(0) + 1(-1) + 1(1) = 0$$

$$CR[2,1] = C[2,:] * R[:,1] = 0(1) + 1(0) + 1(0) = 0$$

$$CR[2,2] = C[2,:] * R[:,2] = 0(-1) + 1(1) + 1(0) = 1$$

$$CR[2,3] = C[2,:] * R[:,3] = 0(0) + 1(-1) + 1(1) = 0$$

$$CR[3,1] = C[3,:] * R[:,1] = 0(1) + 0(0) + 1(0) = 0$$

$$CR[3,2] = C[3,:] * R[:,2] = 0(-1) + 0(1) + 1(0) = 0$$

$$CR[3,3] = C[3,:] * R[:,3] = 0(0) + 0(-1) + 1(1) = 1$$

Which again, is the identity matrix.