ECE 532 Project: Image Classification Using Neural Networks

Justin Cummings, Akhil Sundararajan, Alex Watras

December 7, 2015

Contents

1 Introduction

The general idea behind neural networks is that by mimicking the function of the human brain, we can design a machine which is good at the same sorts of tasks as the brain can perform. In this lab we will learn how to design, build, and train a neural network to perform a basic image classification task.

Neural networks are considered one of the most exciting methods in current machine learning. The inspiration behind the neural network is the human nervous system. The basic building block of both the human nervous system and the neural network algorithm is a neuron. In biology, a neuron is a type of cell that takes in inputs from other neurons or physical stimulus and then transmits an output signal somewhere else in the body. A biological neuron model is showing below in figure 1.

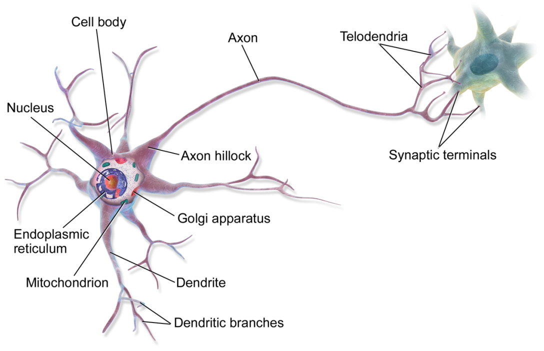

Figure 1: Biological Neuron Cell [11]

As seen in figure (1) above the neuron, on the left side of the figure, takes in outside stimulus from the dendrites. Once the neuron has enough current to be considered above an "activation\firing" threshold it will discharge\fire a signal down the axon.

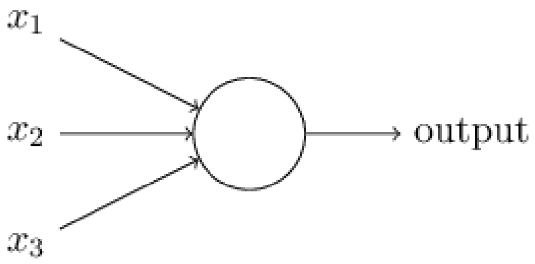

Figure 2: Neural Network Cell [12]

As seen in figure (2) The artificial neuron also takes inputs from outside sources until its inputs exceed a threshold, causing our neuron to fire.

The neural network model attempts to copy the behavior of a biological neuron. Each computational "neuron" combines a variety of inputs into a single output. By chaining together neurons, we create a network that is capable of emulating a wide variety of complex linear or nonlinear functions.

2 Overview

2.1 Neuron Model

Neural networks are based around combining basic nodes called neurons. The neuron will take in our input variables, weight them, and then combine them via an activation function into a single numerical output. By weighting the input variables, we can set which inputs each neuron cares about and set the scenario in which the neuron fires. Below is code to implement a neuron capable of two of the more common activation functions used in neural networks.

```
function h = neuron(X,W,TYPE)
if nargin < 3
    TYPE = 1;
end

X = X(:);
W = W(:);

z = W'*X;
if TYPE == 1
    % gradient = sig(z)*(1-sig(z))
    h = 1./(1+exp(-1*z));
end
if TYPE == 2
    % graddient = 1 - sig(z)^2;
    h = (exp(z) - exp(-z))./(exp(z) + exp(-z));
end
end
```

The two activation functions that are implemented by this code are the sigmoid and the hyperbolic tangent. Each of the different activation functions can be seen in figures (3) and (4) below.

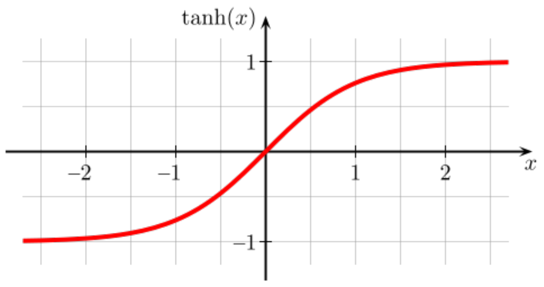

Figure 3: Hyperbolic Tangent function [13]

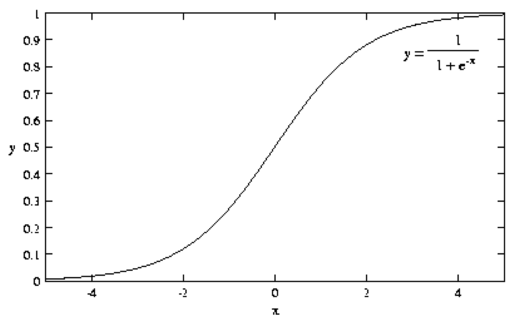

Figure 4: Sigmoid function [14]

As seen in the figures (3) and (4), these functions approach limits very quickly as you travel away from 0. This means that for inputs of a large magnitude, the output of the activation function will be mostly invariant to small changes in the inputs. The reasons these functions are chosen over others is that they have gradients which make training algorithms like back-propagation, discussed in section (3.2.2) and (3.2.1), much easier.

3 Warm Up

3.1 Logic Function Implementation Using Neurons

In this section we will show how the above neuron can be used to create common boolean logic functions. Text in blue denotes an exercise for you to complete and code for this lab can be found here: `http://tinyurl.com/z3azbkl` .

Let's consider the AND function:

X1	X2	Out
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: AND Truth Table

In order to implement this function with our neuron, we need to choose our weights such that we get the desired outputs. Look at the implementation below.

```
function h = andnet(X);
X = X(:);
X = [X;1];
W = [20,20,-30]';
out = neuron(X,W);
h = round(out);
end
```

We know that at an input value of 10 or -10, the sigmoid is far enough from the origin to push the output close to either 0 or 1. Therefore, we know that we want to push the input to be greater than 10 in the case that $X_1 = 1$ and $X_2 = 1$ and to be less than -10 in all the other cases. Therefore, the output of the network is very close to the desired output and by rounding our network output off, we get the exact outputs in the table.

Now see if you can do the same for the OR function as in the table provided. (Hint: consider the following table)

X1	X2	Out
0	0	0
0	1	1
1	0	1
1	1	1

Table 2: OR Truth Table

Solution:
Correct weights should be something like W = [20,20,-10]’

```
function h = ornet(X)
X = X(:);
X = [X;1];
W =

out = neuron(X,W);
h = round(out);
end
```

Some functions can not be easily created with a single neuron of this form. An example of this would be the XOR function classified by the following table:

X1	X2	Out
0	0	0
0	1	1
1	0	1
1	1	0

Table 3: XOR Truth Table

If you attempt to graph this table, you will see that the data is not linearly seperable. However, by chaining multiple neurons together, we can still accurately reproduce the XOR function. Below we have sketched out the basic form of a neural network which could act as an XOR gate.

What weights will successfully create an XOR function? (Hint: How can you create an XOR out of AND and OR gates.)

```
function h = xornet(X)

X = X(:);
X = [X;1];

W11 = []';
a1 = neuron(X,W11);

W12 = []';
a2 = neuron(X,W12);

x2 = [a1 a2 1]';
W21 = []';
out = neuron(x2,W21);

h =round(out);
end
```

Solution:
Correct weights should be something like W11 = [30,-30,-20]', W12 = [-30 30 -20]', W21 = [20 20 -10]

3.2 Neural Network Training Algorithms

3.2.1 Perceptron Algorithm

In this section we will look at the simplest possible neuron example, which utilizes a simple threshold activation function. Figure (5) shows our model of the neuron for this section. This model is uniquely named the *perceptron*.

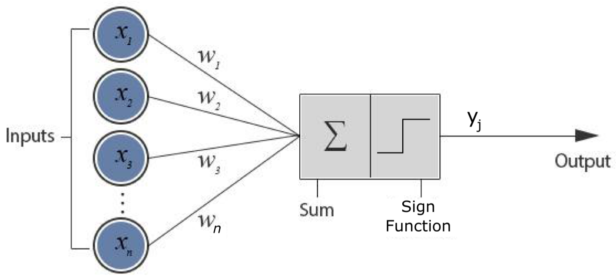

Figure 5: Perceptron Neuron [15]

As depicted in figure (5), the perceptron takes as input the $x^{(j)^{th}}$ training example. From here each feature $x_i^{(j)}$ is weighted by it's respective weight w_i and then passed to the neuron. The neuron sums all weighted feature elements and then passes this output to the activation function which looks at the sign of the weighted sum to decide if the output should be labeled $+1$ or -1. The mathematical operation of the perceptron can be described as follows:

$$y^{(j)} = \text{sign}\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) \tag{1}$$

Below we provide a simple algorithm that will find the optimal weights of $w_i : i \in \{i, ...n\}$.

The perceptron algorithm for training is as follows:

1. Initialize $w_i = \text{random} \ \forall \ i$
2. loop $w_{t+1} = w_t + \eta \left(y^{(j)} - \hat{y}^{(j)} \right) x^{(j)}$
   where $\eta \left(y^{(j)} - \hat{y}^{(j)} \right) x^{(j)} = \begin{cases} 0, & \text{if } y^{(j)} x^{T(j)} w > 0. \\ 2\eta y^{(j)} x^{(j)}, & \text{otherwise} \end{cases}$

The perceptron algorithm is not very interesting to examine for our purposes because it is very limited in its nature. In the following section we will look at how to extend this neuron model to use sigmodial activation functions.

3.2.2 Extension to Sigmoidal Functions

In the previous section we over-viewed the model of a neuron and analyzed how to train it when the activation function is the sign of the weighted sum of the training example. In this section we are going to look at a similar model, but using a sigmoidal activation function.

Our model of the neuron is similar to what we saw earlier in section 3.2.1, with the exception that we will change the activation function of the neuron as seen in figure (6) below.

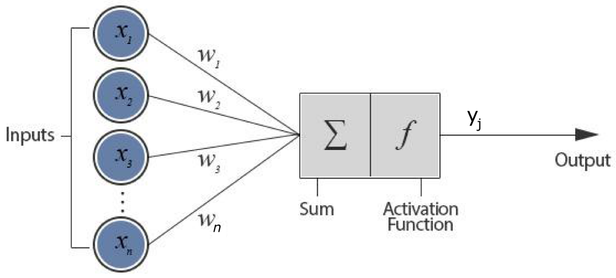

Figure 6: Sigmoidal Neuron [15]

As seen in figure (6) above, x is an n-dimensional training example as the input to our neuron. Each feature element, x_i, is weighted by it's respective weight w_i. These weighted x_i's are then summed at the neuron prior to being passed to the sigmoid function. Mathematically we can represent the output of the neuron as:

$$y^{(j)} = f\left(\sum_{i=1}^{n} w_i x_i^{(j)} \right) \tag{2}$$

such that $f(*)$ is a sigmoidal function. Some common choices for $f(*)$ include the *hyperbolic tangent* function:

$$f(*) = tanh(*) = \frac{e^{(*)} - e^{(-*)}}{e^{(*)} + e^{(-*)}}$$

and the *logistic* function:

$$f(*) = \frac{1}{(1 + e^{-*})}$$

As with most problems we have encountered in class before we want to find a w such that we minimize the following loss function:

$$\begin{aligned} J(x,y,w) &= \frac{1}{2}\sum_{j=1}^{m}\left(\hat{y}^{(j)} - y^{(j)}\right)^2 \\ &= \frac{1}{2}\sum_{j=1}^{m}\left(f\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) - y^{(j)}\right)^2 \end{aligned} \tag{3}$$

1. In the following exercise please show how to minimize the loss function shown in equation (3) using gradient descent by choice of w_i. Instead of minimizing over all m training examples lets analyze for the $(j)^{\text{th}}$ training example. Show your work. (hint: we have started the derivation for you as follows)

   We want to minimize:

$$\min_{w_i}\left\{\left[f\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) - y^{(j)}\right]^2\right\}$$

   Start by taking the derivative of with respects to w_i:

$$\frac{d}{dw_i}\left\{\left[f\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) - y^{(j)}\right]^2\right\} =$$

   Solution:

$$\begin{aligned} &\frac{d}{dw_i}\left\{\left(f\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) - y^{(j)}\right)^2\right\} = \\ &= 2\left(f\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) - y^{(j)}\right)\frac{d}{dw_i}\left(f\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) - y^{(j)}\right)\frac{d}{dw_i}\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) \\ &= 2\left(f\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) - y^{(j)}\right)f'\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right)x_i \\ &= 2\left(\hat{y}^{(j)} - y^{(j)}\right)f'\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right)x_i^{(j)} \end{aligned}$$

It is important to note that most literature will define a new variable δ such that:

$$\delta^{(j)} = 2\left(\hat{y}^{(j)} - y^{(j)}\right)f'\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) \tag{4}$$

From the analysis above we can conclude that the gradient of the loss function with respects to w and the j$^{\text{th}}$ training example is:

$$\frac{d}{dw}\left\{\left[f\left(\sum_{i=1}^{n} w_i x_i^{(j)}\right) - y^{(j)}\right]^2\right\} = \delta^{(j)} \bullet x^{(j)} \tag{5}$$

where $\bullet$ represents the element-wise multiplication function.

2. Using the result that you found above, show how to update the weights w using gradient descent. (hint: recall that gradient descent requires the use of all training examples)

   Solution:

$$w_{t+1} = w_t - \left(\frac{\eta}{2m}\right)\left(\sum_{j=1}^{m} \delta_j \bullet x^{(j)}\right)$$

We have shown up to this point how *back-propagation* works with a single layer neural network. In the next section we will explore how *back-propagation* extends to a multi-layer neural network.

3.2.3 Multi-layer Back-Propagation

In the following section we will go through an example of how back-propagation works with a multi-layer neural network. In particular we will work through the derivations using the example the network shown in figure (7) below. In the end of this section we will summarize how the derivations can be extended to general n-dimensional feature examples.

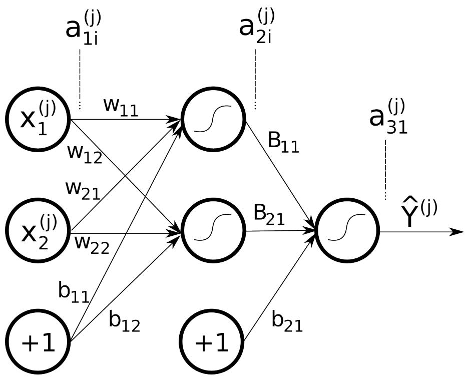

Figure 7: 2-Layer Neural Network

The network we will be using as an example for our derivations is shown above in figure (7). The network consists of an input layer with 2 feature neurons and a bias neuron. The first layer is then fully connected to the second layer consisting of 2 neurons and a bias neuron. The 3rd and final layer of the network consists on a single neuron taking in all inputs from the second layer. Lets look in a litte more detail on how each layer is represented mathematically starting with the second layer.

Mathematically we can represent the output of the second layer and the $k^{(\text{th})}$ neuron as the sum of weighted feature elements plus the bias weight, shown below in equation (6).

$$a_{2k}^{(j)} = f\left(\sum_{i=1}^{n}\left(w_{ik}x_i^{(j)}\right) + b_{1k}\right) \tag{6}$$

Next it can be observed that the output of the third and final layer is:

$$\hat{y}^{(j)} = a_{31}^{(j)} = f\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right) \tag{7}$$

1. Expand equation (7) above using the equation we found for the output of the second layer $a_{2k}^{(j)}$. Show your work. (Understanding how the output of the third layer will become important in a future derivation)

   Solution:

   $$\hat{y}^{(j)} = a_{31}^{(j)} = f\left(\sum_{k=1}^{n}\left[B_{k1}f\left(\sum_{i=1}^{n}\left(w_{ik}x_i^{(j)}\right) + b_{1k}\right)\right] + b_{21}\right)$$

In the following paragraphs we will show how to update the weights at each layer using a gradient descent method. We will start with the final layer where the neuron weights are B_{k1} and the bias weight is b_{21}. For this we will need to take the derivative of the loss function $J(*)$ with respects to the B_{k1}^{th} weight. Recall that the loss function we are using is:

$$J = \frac{1}{2}\sum_{j=1}^{m}\left(\hat{y}^{(j)} - y^{(j)}\right)^2 \tag{8}$$

To update the B_{k1}^{th} weight using gradient descent let's first start by taking the derivative of the loss function at the j^{th} training example:

2. Using the knowledge from section 3.2.3 show, using gradient descent, how to update the weight vector B, then verify your results with the following solution. Do your results agree with the derivation shown below?

   Solution:

   The student should find that the updates of the weights B is as follows

   $$B_{t+1} = B_t - \frac{\eta}{2m}\left(\sum_{j=1}^{m}\delta^{(j)} \bullet a_2^{(j)}\right)$$

   and the full derivation is included step by step in the text for them to follow if they are unsuccessful.

$$\begin{aligned}
&\frac{d}{dB_{k1}}\left\{\left(\hat{y}^{(j)} - y^{(j)}\right)^2\right\} = \\
&= 2\left(\hat{y}^{(j)} - y^{(j)}\right)\frac{d}{dB_{k1}}\left\{\hat{y}^{(j)}\right\} \\
&= 2\left(\hat{y}^{(j)} - y^{(j)}\right)\frac{d}{dB_{k1}}\left\{f\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right)\right\} \\
&= 2\left(\hat{y}^{(j)} - y^{(j)}\right)f'\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right)\frac{d}{dB_{k1}}\left\{\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right\} \\
&= 2\left(\hat{y}^{(j)} - y^{(j)}\right)f'\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right)a_{2k}^{(j)}
\end{aligned}$$

From the analysis above we can simplify the result as:

$$\frac{d}{dB}\left\{\left(\hat{y}^{(j)}-y^{(j)}\right)^2\right\}=2\left(\hat{y}^{(j)}-y^{(j)}\right)f'\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right)+b_{21}\right)\bullet a_2^{(j)} \tag{9}$$

where $\bullet$ represents the element-wise multiplication operation. As before with the single layer cause previously shown, for simplification lets define a new variable δ:

$$\delta^{(j)}=2\left(\hat{y}^{(j)}-y^{(j)}\right)f'\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right)+b_{21}\right) \tag{10}$$

We can then update the weight vector B as follows using gradient descent:

$$B_{t+1}=B_t-\frac{\eta}{2m}\left(\sum_{j=1}^{m}\delta^{(j)}\bullet a_2^{(j)}\right) \tag{11}$$

You should notice at this time that the bias weight b_{21} is not included in the neuron weights as it was in a previous section. This means that we will need to derive an equation on how to update it as well.

3. Find an equation that describes how to update the bias weight b_{21}, then check your answer with the solution provided below. Does your answer agree with the provided derivation? (Hint: find the gradient of the loss function with respect to b_{21})

   Solution:
   The student should find that to update the bias weight b_{21} as:

   $$b_{21}(t+1)=b_{21}(t)-\frac{\eta}{2m}\left(\sum_{j=1}^{m}\delta^{(j)}\right)$$

   and the complete derivation is included in the text if they experience difficulties finding it.

$$\begin{aligned}
&\frac{d}{db_{21}}\left\{\left(\hat{y}^{(j)}-y^{(j)}\right)^2\right\}=\\
&=2\left(\hat{y}^{(j)}-y^{(j)}\right)\frac{d}{db_{21}}\left\{\hat{y}^{(j)}\right\}\\
&=2\left(\hat{y}^{(j)}-y^{(j)}\right)\frac{d}{db_{21}}\left\{f\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right)+b_{21}\right)\right\}\\
&=2\left(\hat{y}^{(j)}-y^{(j)}\right)f'\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right)+b_{21}\right)\frac{d}{db_{21}}\left\{\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right)+b_{21}\right\}\\
&=2\left(\hat{y}^{(j)}-y^{(j)}\right)f'\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right)+b_{21}\right)(1)
\end{aligned}$$

Here we notice that the solution we found for the derivative for the loss function with respect to the second layer bias weight is equal to the variable we previously defined as $\delta^{(j)}$:

$$\frac{d}{db_{21}}\left\{\left(\hat{y}^{(j)} - y^{(j)}\right)^2\right\} = \delta^{(j)} \tag{12}$$

Then by using a gradient descent method we can update the bias weight b_{21}:

$$b_{21}(t+1) = b_{21}(t) - \frac{\eta}{2m}\left(\sum_{j=1}^{m}\delta^{(j)}\right) \tag{13}$$

Up to this point we have only shown how to update the top layer of the network using a gradient descent method. Now what we are going show how to update the weights, w_{ik}, and bias weights, b_{1k}, of the first layer using gradient method.

4. Start by taking the derivative of the loss function with respects to the w_{ik}^{th} weight (hint: remember to use the chain rule for derivatives.)

   The derivation has been started below:

$$\frac{d}{dw_{ik}}\left\{\left(\hat{y}^{(j)} - y^{(j)}\right)^2\right\} =$$
$$= 2\left(\hat{y}^{(j)} - y^{(j)}\right)\frac{d}{dw_{ik}}\left\{\hat{y}^{(j)}\right\}$$
$$= 2\left(\hat{y}^{(j)} - y^{(j)}\right)\frac{d}{dw_{ik}}\left\{f\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right)\right\}$$

   Solution:

$$= 2\left(\hat{y}^{(j)} - y^{(j)}\right) f'\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right)\frac{d}{dw_{ik}}\left\{\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right\}$$
$$= 2\left(\hat{y}^{(j)} - y^{(j)}\right) f'\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right)\left(B_{k1}\frac{d}{dw_{ik}}\left(a_{2k}^{(j)}\right)\right)$$
$$= \delta^{(j)}B_{k1}f'\left(\sum_{i=1}^{n}\left(w_{ik}x_i^{(j)}\right) + b_{1k}\right)\frac{d}{dw_{ik}}\left(\sum_{i=1}^{n}\left(w_{ik}x_i^{(j)}\right) + b_{1k}\right)$$
$$= \delta^{(j)}B_{k1}f'\left(\sum_{i=1}^{n}\left(w_{ik}x_i^{(j)}\right) + b_{1k}\right)x_i^{(j)}$$

Now let us define a new variable $\bar{\delta}_k^{(j)}$ such that:

$$\bar{\delta}_k^{(j)} = \delta^{(j)}B_{k1}f'\left(\sum_{i=1}^{n}\left(w_{ik}x_i^{(j)}\right) + b_{1k}\right) \tag{14}$$

Using the derivation above we can update the weights, w_{ik} using gradient descent as follows:

$$w_{ik}(t+1) = w_{ik}(t) - \frac{\eta}{2m}\left(\sum_{j=1}^{m}\bar{\delta}_k^{(j)}x_i^{(j)}\right) \tag{15}$$

Lastly we will need to update the bias weights, b_{1k}, at the first layer. As we have done before we will need to differentiate the loss function with respects to the bias weight b_{1k} as:

$$
\begin{aligned}
&\frac{d}{db_{1k}}\left\{\left(\hat{y}^{(j)} - y^{(j)}\right)^2\right\} = \\
&= 2\left(\hat{y}^{(j)} - y^{(j)}\right)\frac{d}{db_{1k}}\left\{\hat{y}^{(j)}\right\} \\
&= 2\left(\hat{y}^{(j)} - y^{(j)}\right)\frac{d}{db_{1k}}\left\{f\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right)\right\} \\
&= 2\left(\hat{y}^{(j)} - y^{(j)}\right)f'\left(\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right)\frac{d}{db_{1k}}\left\{\sum_{k=1}^{n}\left(B_{k1}a_{2k}^{(j)}\right) + b_{21}\right\} \\
&= \delta^{(j)}B_{k1}f'\left(\sum_{i=1}^{n}\left(w_{ik}x_i^{(j)}\right) + b_{1k}\right)\frac{d}{db_{1k}}\left(\sum_{i=1}^{n}\left(w_{ik}x_i^{(j)}\right) + b_{1k}\right) \\
&= \delta^{(j)}B_{k1}f'\left(\sum_{i=1}^{n}\left(w_{ik}x_i^{(j)}\right) + b_{1k}\right)(1)
\end{aligned}
$$

5. What does the above derivation of $\frac{d}{db_{1k}}\left\{\left(\hat{y}^{(j)} - y^{(j)}\right)^2\right\}$ simplify to? (hint: look at previously defined variables)

   Solution:
   Here we notice that the solution we found for the derivative for the loss function with respect to the first layer bias weight is equal to the variable we previously defined as $\bar{\delta}_k^{(j)}$:

   $$\frac{d}{db_{1k}}\left\{\left(\hat{y}^{(j)} - y^{(j)}\right)^2\right\} = \bar{\delta}_k^{(j)}$$

Now we can define the update of the b_{ik}^{th} bias weight as:

$$b_{1k}(t+1) = b_{1k}(t) - \frac{\eta}{2m}\left(\sum_{j=1}^{m}\bar{\delta}_k^{(j)}\right) \tag{16}$$

Hopefully by the end of this section you have a better understanding of how back-propagation works for a multi-layer neural network. It ultimately boils down to using gradient descent at the top layer of the network and then propagating it back a layer using the chain rule for derivatives.

4 Main Lab Activities

4.1 2D Data Classification Example

4.1.1 Single Layer Neural Network

In this section we are going to look at how a single layer single neuron works to train and classify a 2D data set. The data set we are going to train and classify is shown below in figure (8). Text in blue denotes an exercise for you to complete!

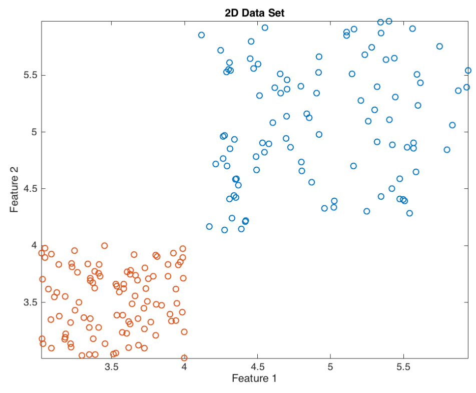

Figure 8: Linearly Separable 2D Data Set

The 2D data set above in figure (8) is generated in two highly separable clusters. The left cluster is labeled -1 and the right cluster is labeled $+1$. We acknowledge that this may be a somewhat over simplified data set, but it is meant as an introduction to before we get to the more complicated non-linear dataset and MNIST handwritten numbers dataset.

1. Start by opening the "*neural_network_sl_sn.m*" MATLAB code provided with the tutorial.

2. Run the script until the error between the true y vector and the estimated y vector $\hat{y} < 1.5 * 10^{-1}$

3. Report the error rate found by the single layer neural network.

   Solution:
   The students should find that the error rate for the single layer neural network is approximately `err_rate_nn = 0`.

4. Implement a regularized least squares solution on the 2D data set. How does the error rate compare to the error rate found with the neural network?

   Solution:
   The students should provide code for the regularized least squares implementation similar to:

```
X = X(:,1:2);
w_rls = pinv(X'*X + lambda*eye(size(X'*X)))*X'*y;
err_rate_rls = sum(sign(X*w_rls) ~= y)/length(y)
```

   For the RLS implementation they should find that the error rater is approximately `err_rate_rls = 0.0150`. Results may vary slightly, but in general the NN should do better.

5. Run the final section of the MATLAB code in the provided .m file to generate a decision boundary for the neural network. On the same figure, plot the decision boundary you found with the regularized least squares solution.

   Solution:
   The students should include a plot similar to the one shown below:

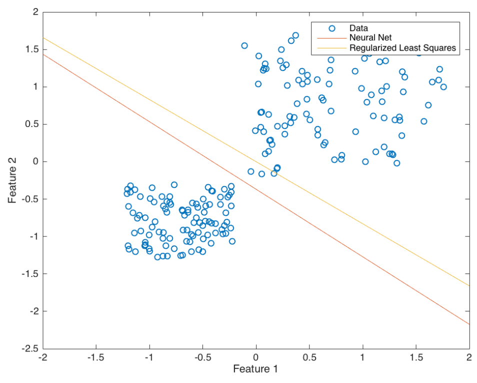

Figure 9: Single Layer NN and RLS Comparison

4.1.2 Multi-layer Neural Network

In this section of the lab we will explore our MATLAB implementation of a multi-layer neural network. The neural network that will be used in this section is designed to replicate the one we examined mathematically in section (3.3.3). The 2D dataset that we will be classifying is different from the one that we previously used in section (4.1.1). This time our data set is **not** linearly separable. The new dataset we will be using is shown below in figure (10).

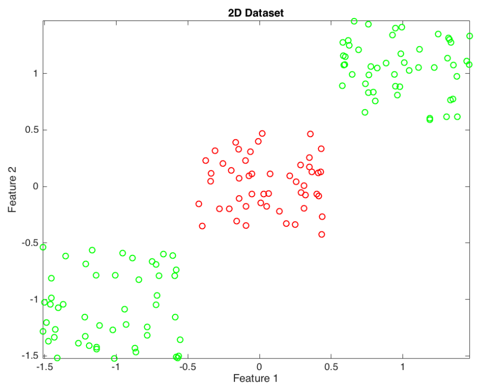

Figure 10: Non-linearly separable 2D Data Set

The 2D data set above in figure (10) is generated in three highly separable clusters, but the trick here is they are not linearly separable. The left cluster is labeled -1, the middle cluster is labeled $+1$, and the right cluster is labeled -1. In this exercise we will see how multi-layer neural networks can find a non-linear solution to this dataset.

1. Start by opening the "*neural_network_dl_dn.m*" MATLAB code provided with the tutorial.

2. Run the script until the error between the true y vector and the estimated y vector, $\hat{y}$, is less than $5 * 10^{-1}$

3. Report the error rate found by the multi-layer neural network.

   Solution:
   The students should find that the error rate for the multi-layer neural network is approximately `err_val = 0`.

4. If you were to attempt to classify the dataset above using a single-layer neural network, would the classification performance be better or worse that the same data run on a multi-layer neural network. (you do not have to carry out actual calculations)

   Solution:
   If you were to attempt to classify this data set using the single layer single neuron code we developed previously, you would find that you are not able to classify at least 50% of the data because the single layer can only deal with linearly separable data sets

5. Explain the purpose of the following code snippet taken from the multi-layer neural network:

```
eta = etano/(1+sqrt(count)*.01);
```

   (Note: η is known as our *learning rate* parameter)

   Solution:
   The code snippet above shows that we are using a dynamically adjusted learning rate or step size. This allows us to take larger steps at the beginning of the algorithm and smaller once we get close to the actual solution. It also helps prevent divergence issues when using a fixed rate.

6. Run the final section of the MATLAB code in the provided .m file to generate a decision boundary for the neural network. Include a plot of the decision boundary with the original data.

   Solution:
   The students should include a plot similar to the one shown below:

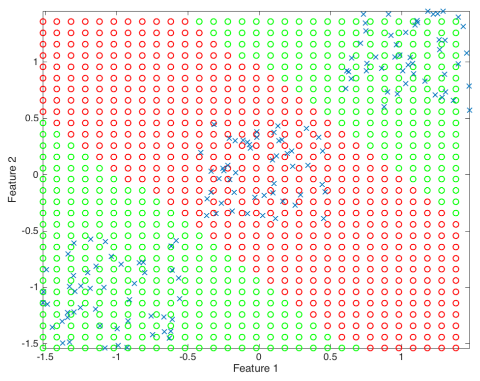

Figure 11: Multi-layer NN Decision Boundary

7. From the two examples we have seen, What is the general purpose of using multi-layer neural networks?

   Solution:
   From the examples we have seen in this lab, adding additional layers to the neural network allows it classify non-linear data

4.2 Classification of MNIST Handwritten Digits

We will now explore different approaches to classifying images using a subset of the sixes and sevens from the MNIST handwritten digits database. This dataset contains a collection of handwritten digits that can be used to test image classification algorithms [3]. While the original MNIST database contains 50,000 training digits and 10,000 test digits, we will be using 1000 sixes and 1000 sevens for training and testing in this lab. In order to access this dataset, first download the data6 and data7 files from: `http://cis.jhu.edu/~sachin/digit/digit.html` or load the files provided with the tutorial. Our goal is to design a binary classifier which can identify whether an image is of a six or a seven.

4.2.1 Support Vector Machines

Recall that a support vector machine seeks to find a set of weights to classify unlabeled examples by solving the regularized hinge loss optimization

$$\sum_{i=1}^{m}(1 - y_i \mathbf{x_i}^T \mathbf{w})_+ + \lambda ||\mathbf{w}||_2^2$$

The following questions will explore using a support vector machine to perform binary classification of sixes and sevens. The file `svm_digits.m` loads the digit data (1000 sixes and 1000 sevens), then takes a random sampling of 100 sixes and 100 sevens as training data and the remainder of the digits as test data. Cross validation is implemented to compute average error across 100 iterations.

1. Train an SVM classifier for the MNIST sixes and sevens using the MATLAB file `svm_digits.m`. What is your average error on the test data? The

average test error is reported in the variable `avg_err`.

Make sure to fill in the blanks in lines 82 and 83 of `svm_digits.m` before running the file:

```
svmStruct = svmtrain(???,???);
  pred = svmclassify(???,???);
```

Solution:
The argument to `svmtrain` and `svmclassify` should be as follows:
`svmStruct = svmtrain(A_train,b_train)`
`pred = svmclassify(svmStruct,A_test)`.
Using the code provided in `svm_digits.m`, the average test error rate is approximately 0.64%.

2. Now come up with a kernelized SVM classifier for the digits using a Gaussian kernel and cubic polynomial kernel by setting the boolean variables `turnOnGK = 1` and `turnOnPK = 1`. What is the average test error using these nonlinear kernels? What do your results suggest about the linear separability of the sixes and sevens?
Solution:
The average test error using the SVM with a Gaussian kernel is 41.43%. The average test error using the SVM with a cubic polynomial kernel is 2.61%. The higher error rate using the Gaussian kernel and polynomial kernel relative to the linear kernel suggests that the data are linearly separable.

3. Use the sigmoid (MLP) kernel option in `svmtrain` by turning on the variable `turnOnMLP`. The Sigmoid Kernel has the following form [1]:

$$k(x_i, x_j) = tanh(P1 \times (x_i^T x_j) + P2)$$

Choose the parameters `P1` and `P2` and specify them in the `'mlp_params'` option as shown in line 127. Try out several values for the parameters, with `P1>0` and `P2<0`. What is the lowest test error rate you can achieve? HINT: Start out by trying values of `P1` small and `|P2|`>0.5.
Solution:
With parameters `P1=0.001` and `P2=-1.5`, The average test error using the SVM with an MLP kernel is 0.62%.

4. Repeat your experiments above with different sizes of training data by changing the vector `training_sizes` (Uncomment line 48). Set `showPlot=1` and use the plotting code at the end of the file to plot average test error versus training size. What is the smallest training size in the case of each kernel which yields an average error rate of $< 5\%$?
Solution:
Here is the plot showing error rate vs. training size. Using polynomial, MLP, and linear kernels, the an error rate of ¡5% can be achieved with approximately 50 training examples. The Gaussian Kernel never yields an error rate as low as 5%.

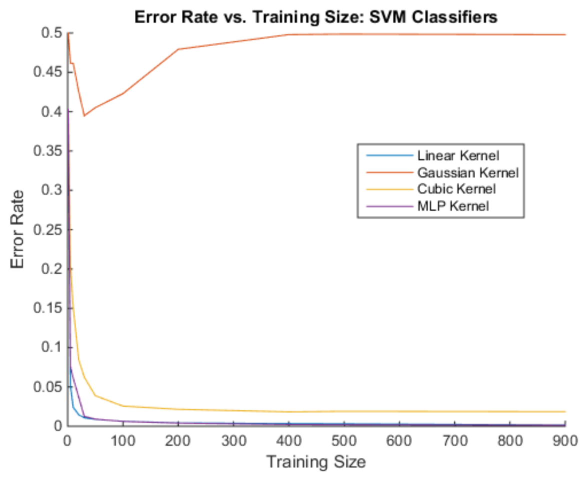

Figure 12: Error Rate vs. Training Size using Different SVM Kernels.

4.2.2 Multi-Layer Neural Network

In this section of the lab we will look at how our original MATLAB script implementation of a multi-layer neural network performs when tasked with classifying 6's and 7's from the MNIST handwritten number data set. We have a total of 1000 examples of $28x28$ images of handwritten 6's and 7's.

1. Start by opening the "*neural_network_dl_dn_mnist_crossval.m*" MATLAB code provided with the tutorial.

The script will use a training set of 100 examples of 6's and 100 examples of 7's. Of these 200 examples, we train on 150 and check the error on the remaining 50 examples. The training process will repeat until the norm of the error is less than the defined error tolerance. This process is repeated 5 times per generated training set. We then check to see which weights out of the 5 passes provides the best performance on the 50 training examples. The best weights vectors are selected and then then remaining 1800 examples of 6's and 7's are classified using the best found weight. We repeat this whole process 5 times to get a good idea of the overall performance of the system (making sure to generate random permutations of the training and validation sets each iteration).

2. Run the script with an error tolerance of $5 * 10^{-1}$.

3. Report the average performance of the multi-layer neural network.

   Solution:
   The average performance of the multi-layer neural network should be approximately `avg_err_rate = 0.0091` or equivocally 1%

4. [Optional]: To improve the performance of the multi-layer neural network, decrease the error tolerance to $1 * 10^{-1}$.

   (Warning: This significantly increases training time in the code)

   Solution:
   The average performance of the multi-layer neural network should be approximately `avg_err_rate = 0.0075` or equivocally 0.75%, note that this

run took approximately 4 hours, if this algorithm was run for days or weeks we would expect to see it perform better than the SVM classifier.

In the provided code there is a variable called `l2_extran`. This variable allows you to increase the number of neurons at the second layer from `n` to `l2_extran + n`.

5. [Optional]: Experiment with increasing the value of `l2_extran`. In what situations do you think that increasing this parameter would increase performance.

   Solution:
   In certain situations increasing the number of neurons in the second layer could increase the ability of the network to define non-linear decision boundaries. For our case this should not matter because the MNIST dataset is linearly separable.

4.2.3 MATLAB Neural Network Toolbox

Matlab has a built in toolkit to implement neural networks. In this section, We will walk you through how to use matlab's neural network toolbox to solve the handwritten data image classification problem above.

1. Run the HWDataimport2 script to generate the dataset we will be using.

2. Open the Neural Network Fitting Toolbox in matlab. It can be found under the apps tab in Matlab 2015a. A GUI should open up. Click next

3. You should be at a menu labeled select data. If you are working from the previous script, X is our input, y is our desired output, and our samples are matrix columns.

4. For now you may use the default values for validation and Test Data and Network Architecture. Train the network with the Levenberg-Marquardt training algorithm.

5. Finally, on the save results screen, choose to save data to the workspace. You now have a working neural network. This should create a neural network object named net that we will use in the next steps.

6. Test the neural network by running the command stem(net(Xtest)). This will create a graph of the neural network output given from each sample. If your network is working well, this plot should contain almost entirely values close to one in the first thousand samples and almost entirely values close to zero in the second thousand samples.
   Solution:
   Students should get a figure similar to the one shown below.

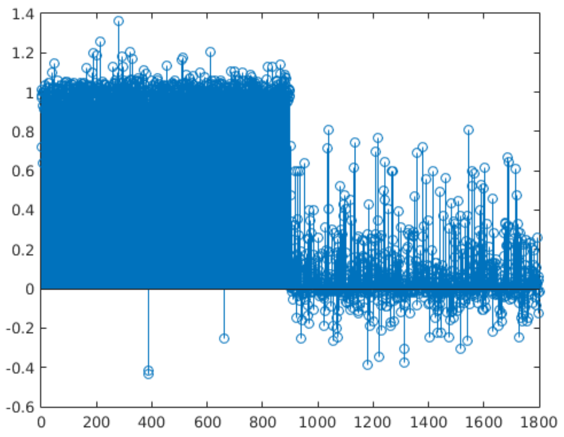

Figure 13: Output of stem(net(Xtest))

7. Design a classifier using the output of this neural network. What sort of error rate are you getting?
   Soution:
   Using just the round function as the classifier, for Levenberg-Marquadt we should get an error rate of 3.9444 percent. For bayesian regularization we should get an error rate of 0 percent.

8. [Optional:]Try repeating the previous steps using the Bayesian Regularization training algorithm.**WARNING: This may take several hours to complete the training step.** What do you think are the tradeoffs between the two methods?

   Solution:
   The Levenberg-Marquardt is much quicker but does not fit the data quite as well.

9. Compare the performance of classifying the MNIST handwritten dataset between the different implementations found in section 4.2 (SVM, Original MATLAB code, and MATLAB Tool Box).
   Solution:
   In computation speed: Levenberg-Marquadt > SVM > Original Code > Bayesian Regularization
   In final Error rate: Bayesian Regularization > Original Code > SVM > Levenberrg-Marquadt

4.3 Convolutional Neural Networks and Caffe

Neural networks involving many layers are sometimes called ”deep” neural networks, and can be a powerful way to approach image classification. In this section of the lab, we introduce convolutional neural networks, a class of deep neural networks that is popular today because they offer important capabilities to image processing problems such as spatial invariance. Using Caffe, an open source deep learning framework, we will run a demo to classify the MNIST data using the LeNet convolutional neural network.

In the neural networks we have seen so far, each layer in the network was "fully connected", meaning each neuron was connected to every input in the layer before it. Convolutional neural networks have sparse connectivity so as to exploit the spatial correlation of images; that is, neurons in a hidden layer m are only connected to a subset of the input neurons in layer $m-1$, as shown in Figure 14 below [8].

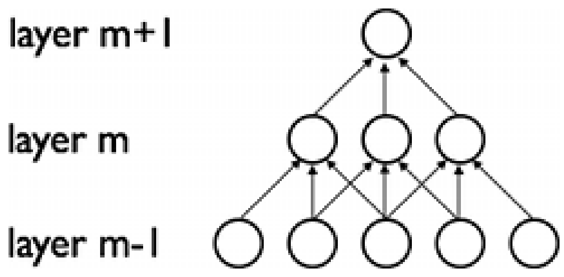

Figure 14: Sparse Connectivity in a Convolutional Neural Network [8]

For our handwritten digits, each image is 28-by-28 pixels, so an n-by-n subset of the pixels maps to a neuron in the next layer. This n-by-n subset of pixels is known as a *receptive field* [7].

1. Suppose the receptive field size used in a convolutional neural network is 5-by-5. For a stride length of 2, i.e., if the receptive field is shifted across the image two pixels at a time, how many neurons will be in the subsequent hidden layer?
   Solution:
   There will be 13-by-13 neurons in the next layer.

Associated with each receptive field is a set of weights which determine the activation of the neuron in the hidden layer. These weights are shared among all neurons in the hidden layer, and thus constitute a linear mapping from the input image to the hidden layer. The output of this filtering operation is called a *feature map*, and with activation function σ, is given by the following equation [8]

$$h_{ij}^k = \sigma((W^k * x)_{ij} + b_k)$$

where h is the feature map, k represents the k^{th} feature map (in general, there can be multiple feature maps in a given layer), W are the shared weights, x_{ij} are the pixel data, and b_k is the bias for feature map k. Convolution is denoted by the $(*)$ operator and for two-dimensional signals, is given by the following [8]

$$f[m,n] * g[m,n] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[u,v]g[m-u,n-v]$$

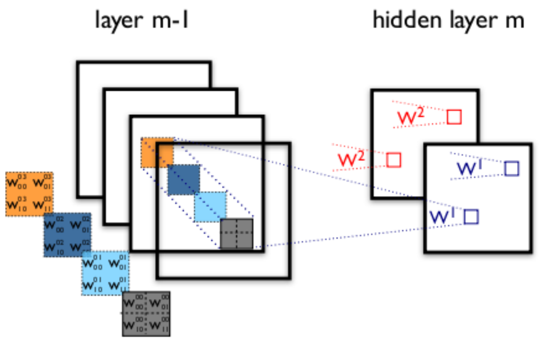

Figure 15: Convolutional Layers [8]

In addition to convolutional layers, a convolutional neural network includes layers for pooling, a process which seeks to sample the feature map (output of a convolutional layer) for the next layer of neurons. The LeNet network we introduce next uses max-pooling, which takes a feature map output and returns the maximum activation of each m-by-m subset of pixels from the feature map, thus reducing computation in subsequent layers of the network [7].

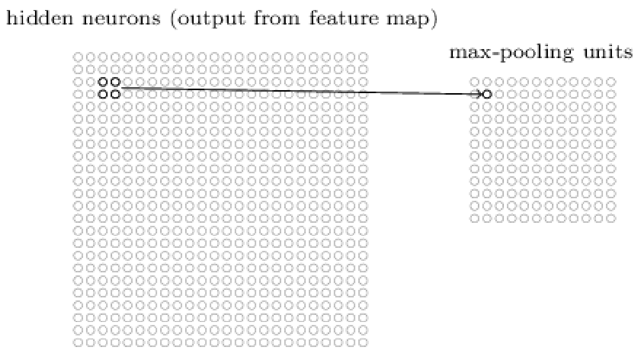

Figure 16: Max-pooling [7]

2. If the output of a convolutional layer in our network consists of 24-by-24 neurons, and we use max-pooling units of size 4-by-4, how many neurons are in the layer as a result of max-pooling?
   Solution:
   There will be 6-by-6 neurons in the next layer.

Let us now use Caffe to run a demo to classify the entire MNIST handwritten digit set. In this example, Caffe will build the LeNet convolutional network, which contains convolutional and pooling layers followed by two fully connected layers of neurons. The activation function used in this LeNet network is the rectified linear unit, or ”ReLU” function [2].

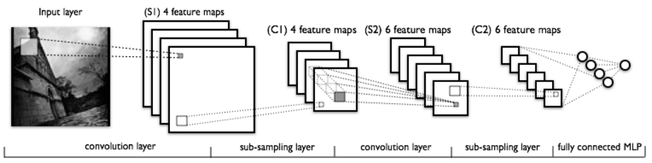

Figure 17: LeNet Family Architecture [8]

You must run the following Caffe example in the Windows 64-bit Operating System. Running Caffe requires installing a number of dependencies first, but Niu Zhiheng has ported Caffe to Windows and made a package available with which we can run LeNet on the MNIST data [10].

3. Go to `https://github.com/niuzhiheng/caffe` and download the standalone Caffe package file **(scroll to the middle of the page, the link is in the Windows Installation Readme where it says "You can download the windows x64 standalone package...")**. Unzip the file into your working directory.

4. Open up the Windows command prompt (Start → Command Prompt) and `cd` to the `standalone` directory.

5. Open the file `lenet_solver.prototxt` and set the following variables as follows:

```
max_iter:  1000
display:  100
snapshot:  100
```

6. Enter `runMNIST.bat` in the command line to run the MNIST classification. What is the testing accuracy after 1000 iterations?
   Solution:
   The accuracy after 1000 iterations is 97.89%.

   You will see each layer of LeNet being created, followed by reports of testing accuracy every 100 iterations. Here's what the Caffe should log while the script is running. Test Score 0 reports the testing accuracy, while Test Score 1 reports the value of the testing loss function.

```
[solver.cpp:84] Testing net
[solver.cpp:111] Test Score #0:  0.9605
[solver.cpp:111] Test Score #1:  0.129299
```

7. Caffe uses stochastic gradient descent to compute weights for the network. Experiment with a different learning rate policy in the solver by changing the `lr_policy` parameter in `lenet_solver.prototxt` to `"fixed"`. How might this affect the runtime and test accuracy?
   Solution:
   A fixed learning rate could lead to divergence if the learning rate is not small enough. In this case, the base learning rate of 0.01 is small enough so that the accuracy achieved with a 'fixed' learning rate policy is about the same as when the 'inv' policy is used.

The "inv" learning rate policy calculates a step size as follows 2,

```
base_lr * (1+gamma*iter)^(- power)
```

while the "fixed" learning rate policy always chooses base_lr as the step size.

8. Try running the script for more iterations (by changing max_iter). How accurate do your test results get?

   Solution:
   After 2000 iterations, the accuracy increases to 98.49%. Observe that the value of the testing loss function has also decreased.

We have now explored several implementations of neural networks. Hopefully you have a better understanding of the various methods that exist to tackle image classification problems.

5 References

1. C.J.C. Burges. "Geometry and Invariance in Kernel Based Methods," in *Advances in Kernel Methods: Support Vector Learning.* Cambridge, MA: MIT Press, 1999, ch.7,sec.7,p.98.

2. Y. Jia. Caffe: An open source convolutional architecture for fast feature embedding. http://caffe.berkeleyvision.org/, 2013.

3. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.

4. LISA lab. "Convolutional Neural Networks (LeNet)", 2010. http://deeplearning.net/tutorial/lenet.html

5. Mazur, Matt. "A Step by Step Backpropagation Example." 17 Mar. 2015. Web. 06 Dec. 2015. http://mattmazur.com /2015/03/17/a-step-by-step-backpropagation-example/.

6. Andrew Ng, Jiquan Ngiam, et al. "UFLDL - Multi-Layer Neural Network", . http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/

7. M. Nielsen. "Neural Networks and Deep Learning", Determination Press, 2015. http://neuralnetworksanddeeplearning.com

8. Openclassroom.stanford.edu, 'Unsupervised Feature Learning and Deep Learning', 2015. http://openclassroom.stanford.edu/MainFolder/ CourseP-age.php?course=ufldl.

9. Genevieve Orr, Nici Schraudolph, and Fred Cummins. "CS-449: Neural Networks", 1999. https://www.willamette.edu/ gorr/classes/cs449/intro.html

10. N. Zhiheng. Caffe Windows Port. https://github.com/niuzhiheng/caffe.

11. *Figure 1.* Digital image. N.p., n.d. Web. http://www.wikiwand.com/en/ Neuron.

12. *Figure 2.* Digital image. N.p., n.d. Web. http://neuralnetworksanddeeplearning.com/images/tikz0.png.

13. *Figure 3.* Digital image. N.p., n.d. Web. http://upload.wikimedia.org/wikipedia/commons/thumb/8/87 /Hyperbolic_Tangent.svg/490px-Hyperbolic_Tangent.svg.png.

14. *Figure 4.* Digital image. N.p., n.d. Web. http://pcrochat.online.fr/webus/ tutorial/sigmoid.png.

15. *Figure 6.* Digital image. N.p., n.d. Web. http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-part-1/7.