Assignment 2

Wednesday, March 13th, 2019 — ECE1513
University of Toronto — Introduction to Machine Learning

1 Neural Networks using Numpy

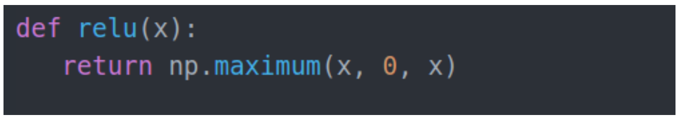
In this section, we build a conventional (not convolutional!) neural network using Numpy and train the network using Gradient Descent with Momentum.

1.1 Helper Functions

We first start by building a few helper functions to modularize the code. The functions are built in vectorized form to reduce the computational time needed to calculate the results.

1.1.1 ReLU()

This is the activation function used in the hidden layer.

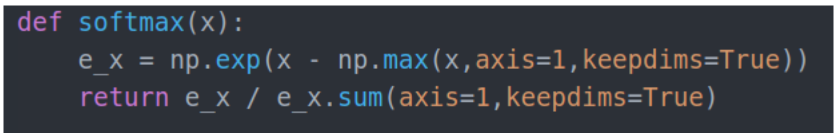


```python
def relu(x):
    return np.maximum(x, 0, x)
```

Figure 1: ReLu Function Code

1.1.2 softmax()

This function is used at the outer layer to output the logits.

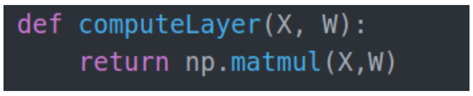


```python
def softmax(x):
    e_x = np.exp(x - np.max(x,axis=1,keepdims=True))
    return e_x / e_x.sum(axis=1,keepdims=True)
```

Figure 2: Softmax Function Code

1.1.3 computeLayer()

This function is to calculate the product between matrices.

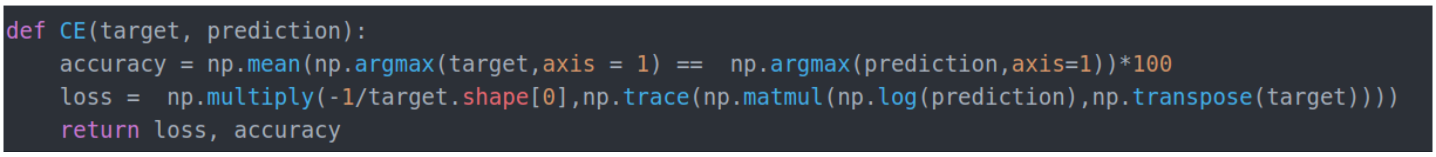


```python
def computeLayer(X, W):
    return np.matmul(X,W)
```

Figure 3: ComputeLayer Function Code

1.1.4 averageCE()

This function returns the average Cross-Entropy cost



```python
def CE(target, prediction):
    accuracy = np.mean(np.argmax(target,axis = 1) ==  np.argmax(prediction,axis=1))*100
    loss =  np.multiply(-1/target.shape[0],np.trace(np.matmul(np.log(prediction),np.transpose(target))))
    return loss, accuracy
```

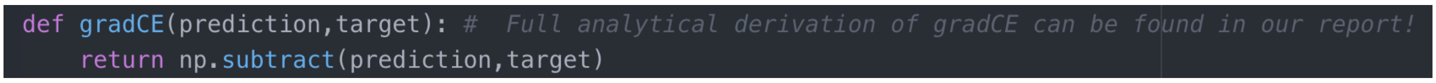
Figure 4: Cross Entropy Function Code

1.1.5 gradCE()

In the code, we do not use this, instead we use the product of this function with the terms shown in Section 1.2.1 to simplify the calculation. The end results just ends up being the difference between the prediction and the target. Nevertheless, below is the derivative of the cost function with respect to the output of the softmax.

$$\frac{\partial \mathcal{L}}{\partial s_k} = -\frac{\partial}{\partial s_k}\left(\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^n log s_k^n\right) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\frac{t_k^n}{s_k^n}$$

The snippet below shows the function in the code in the form where it just becomes the subtraction of the predicted and target value:



```
def gradCE(prediction,target): #  Full analytical derivation of gradCE can be found in our report!
    return np.subtract(prediction,target)
```

Figure 5: gradCE Resultant Code

1.2 Backpropogation Derivation

1.2.1 Output Layer Weights

The partial derivative of the cross entropy loss function with respect to the output layer weights is given by the chain rule:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_O} &= \frac{\partial \mathcal{L}}{\partial s_O}\frac{\partial s_O}{\partial z_O}\frac{\partial z_O}{\partial w_O}\\ &= \frac{\partial}{\partial s_O}\Big(-\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^n log(s_k^n)\Big)\frac{\partial}{\partial z_O}\Big(\frac{\exp z_j}{\sum_{k=1}^{K}\exp z_k}\Big)\frac{\partial}{\partial w_O}\Big(w_O^T s_H + b_O\Big)\end{aligned}$$

Where $\mathcal{L}$ is the cross entropy loss function provided in the assignment, s_O is the activated output of the softmax function being applied to the logit from the output layer, and z_O is the logit from the output layer. Starting with the partial derivative of the softmax function with respect to the output logit:

$$\begin{aligned}\frac{\partial \sigma(z_O)}{\partial z_{O,i}} &= \frac{\partial(\frac{\exp(z_{O,j})}{\sum_{k=1}^{K}\exp(z_{O,k})})}{\partial z_{O,i}}\\ \text{if } i=j: \frac{\partial \sigma(z_O)}{\partial z_{O,i}} &= \frac{\exp(z_{O,i})\sum_{k=1}^{K}\exp(z_{O,k}) - \exp(z_{O,i}\exp(z_{O,i})}{\Big(\sum_{k=1}^{K}\exp(z_{O,k})\Big)^2}\\ &= \frac{\exp(z_{O,i})}{\sum_{k=1}^{K}\exp(z_{O,k})}\frac{\sum_{k=1}^{K}\exp(z_{O,k}) - \exp(z_{O,i})}{\sum_{k=1}^{K}\exp(z_{O,k})}\\ &= \frac{\exp(z_{O,i})}{\sum_{k=1}^{K}\exp(z_{O,k})}\Big(1-\frac{\exp(z_{O,i})}{\sum_{k=1}^{K}\exp(z_{O,k})}\Big) = \boldsymbol{y_i(1-y_i)}\\ \text{if } i\neq j: \frac{\partial \sigma(z_O)}{\partial z_{O,i}} &= \frac{0-\exp(z_{O,i})\exp(z_{O,j})}{\Big(\sum_{k=1}^{K}\exp(z_{O,k})\Big)^2} = \boldsymbol{-y_i y_j}\end{aligned}$$

Using this knowledge, we now take the partial derivative of the cross entropy loss function with respect to the input to the softmax function:

$$\frac{\partial \mathcal{L}}{\partial s_O}\frac{\partial s_O}{\partial z_O} = \frac{\partial \mathcal{L}}{\partial z_O} = -\sum_{k=1}^{K}\frac{\partial(t_k log(y_k))}{\partial z_{O,i}} = -\sum_{k=1}^{K}\frac{t_k}{y_k}\frac{\partial y_k}{\partial z_{O,i}}$$

Using the two different cases outlined above,

$$\begin{aligned}&= -\frac{t_i}{y_i}\frac{\partial y_i}{\partial z_{O,i}} - \sum_{k\neq 1}^{K}\frac{t_k}{y_k}\frac{\partial y_k}{\partial z_{O,i}} = -\frac{t_i}{y_i}y_i(1-y_i) - \sum_{k\neq i}^{K}\frac{t_k}{y_k}(-y_k y_i)\\ &= t_i y_i - t_i + \sum_{k\neq i}^{K} t_k y_i = \sum_{k=1}^{K} t_k y_i - t_i = y_i - t_i = \boldsymbol{y} - \boldsymbol{t} = \boldsymbol{\delta_O}\end{aligned}$$

This is the same as the derivative for the logistic function from the previous assignment! Taking the final partial derivative:

$$\frac{\partial z_O}{\partial w_O} = \frac{\partial(w_O^T s_H + b_O)}{\partial w_O} = s_H^T$$

We finally have out expression for the gradient of the cross entropy loss function with respect to the output layer weights:

$$\frac{\partial \mathcal{L}}{\partial w_O} = (y-t)s_H^T = \delta_O s_H^T$$

1.2.2 Output Layer bias

The derivation of the partial derivative of the cross entropy loss function with respect to the output layer biases is similar. The partial derivative of the cross entropy loss function with respect to the input to the softmax function remains the same, however the partial derivative of the softmax input with respect to the output layer bias is just 1:

$$\frac{\partial z_O}{\partial b_O} = \frac{\partial(w_O^T s_H + b_O)}{\partial b_O} = 1$$

Therefore, the partial derivative of the cross entropy loss function with respect to the output layer biases is given by:

$$\frac{\partial \mathcal{L}}{\partial b_O} = \delta_O = (y - t)$$

1.2.3 Hidden Layer Weights

We continue our back propagation algorithm by finding the partial derivative of the cross entropy loss function with respect to the hidden layer weights:

$$\frac{\partial \mathcal{L}}{\partial w_H} = \frac{\partial \mathcal{L}}{\partial s_O}\frac{\partial s_O}{\partial z_O}\frac{\partial z_O}{\partial s_H}\frac{\partial s_H}{\partial z_H}\frac{\partial z_H}{\partial w_H}$$

The first two terms are already defined by our output error, δ_O. The third term is given by:

$$\frac{\partial z_O}{\partial s_H} = \frac{\partial}{\partial s_H}(w_O^T s_H + b_O) = w_O^T$$

The fourth term is the derivative of the ReLU activation function, which is given by:

$$ReLU(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

The derivative of this function is simply:

$$\frac{\partial}{\partial x} ReLU(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

Finally, the fifth term is similar to the third term from the derivation of the partial derivative of the cross entropy loss function with respect to the output layer weights:

$$\frac{\partial z_H}{\partial w_H} = \frac{\partial (w_H^T x_I + b_H)}{\partial w_H} = x_I^T$$

Thus, the total partial derivative of the cross entropy loss function with respect to the hidden layer weights is given by:

$$\frac{\partial L}{\partial w_H} = \delta_O w_O^T \frac{\partial}{\partial s_H} ReLU(s_H) x_I^T = \delta_H x_I^T$$

1.2.4 Hidden Layer Biases

The derivation of the partial derivative of the cross entropy loss function with respect to the hidden layer biases is similar. The partial derivative of the cross entropy loss function with respect to the input to the ReLU function remains the same, however, the partial derivative of the ReLU input with respect to the hidden layer bias is just 1:

$$\frac{\partial z_H}{\partial b_H} = \frac{\partial (w_H^T x_I + b_H)}{\partial b_H} = 1$$

Therefore, the partial derivative of the cross entropy loss function with respect to the hidden layer biases is given by:

$$\frac{\partial \mathcal{L}}{\partial b_H} = \delta_H = \delta_O w_O^T \frac{\partial}{\partial s_H} ReLU(s_H)$$

1.3 Learning

As no learning rate was given to use, we had to experiment with a couple of values. The value we settled on was 10^{-5}. At this rate, the error converges, and the graph is smooth. We set $\gamma = 0.95$. The resulting accuracy and loss plots are shown below:

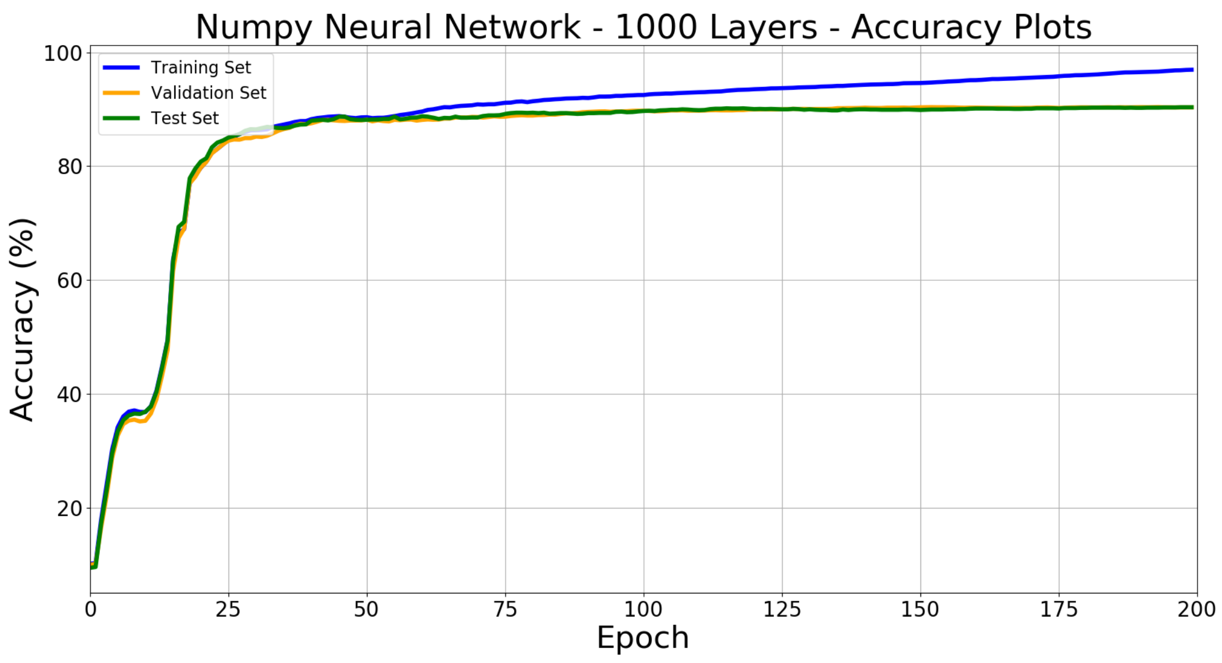


Figure 6: Numpy Neural Network with 1000 Layers Accuracy Curves

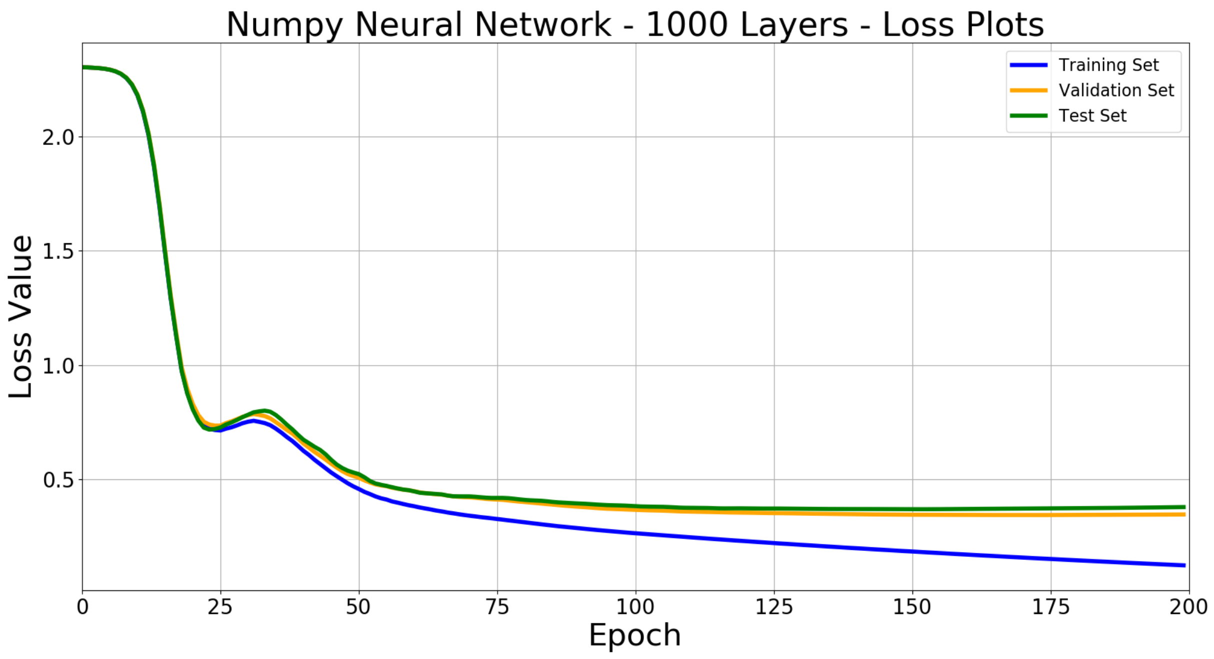


Figure 7: Numpy Neural Network with 1000 Layers Loss Curves

The table below summarizes the results at the final epoch:

Time Taken(s)	**Training Set Loss/Accuracy**	**Validation Set Loss/Accuracy**	**Test Set Loss/Accuracy**
232.41	0.122815/96.93%	0.345485/90.32%	0.377778/90.35%

Table 1: Numpy Neural Network with 1000 Layers Summary

The time taken was based on running the test on an AMD Ryzen 5 2600 CPU.

1.4 Hyperparameter Investigation

1.4.1 Number of Hidden Units

The table below summarizes the results

Number of Hidden Units	Time Taken (s)	Test Set Accuracy	Test Set Loss
100	27.56	89.90%	0.388943
500	129.44	90.05%	0.383086
2000	426.98	90.38%	0.367983

Table 2: Hyperparameter Investigation Summary

From Table 2, it can be seen that increasing the number of hidden units increases the final test accuracy and lowers the final test set loss. The cost of achieving a higher accuracy is in the time taken. There is an almost linear correlation between the number of hidden units and the time taken, however the gain in accuracy isn't anywhere as good. For example, if we compare the accuracy of the 1000 hidden unit and 2000 hidden unit models, there is only 0.03% improvement going from the former to the latter, but requires twice as much time to train the model. Having more hidden units also increases the time taken for the neural network weights to converge. This is shown in Figure 8, where the lower the number of hidden units, the quicker the time required for the network to reach its plateau. Comparing the different hidden layers, it can be said that using either 500 or 1000 hidden units best balances accuracy vs time taken.

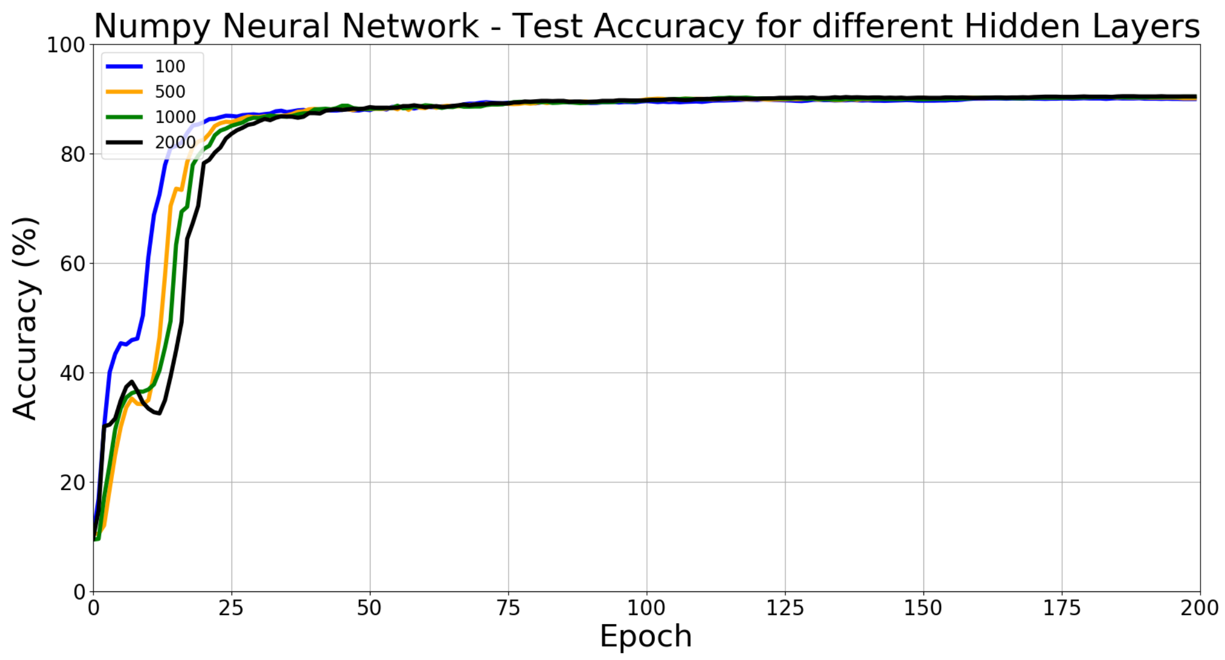


Figure 8: Numpy Neural Network with 1000 Layers Loss Curves

1.4.2 Early Stopping

Early stopping occurs before training accuracy and loss start diverging from the validation and test set. In the 1000 hidden unit neural network mode, this occurs at the 50th epoch as seen on Figure 6. Table 3 shows the training, validation, and test set accuracy at both the 50th and 200th epoch:

Epoch	Training Set Accuracy	Validation Set Accuracy	Accuracy
50	88.60%	88.15%	88.14%
200	96.93%	90.32%	90.35%
Difference	8.33%	2.17%	2.20%

Table 3: Numpy Neural Network Epoch Comparison

From the table, it can be seen that the training set improves 4 times more than the validation and test set. Before the 50th epoch, the training, validation, and test data sets are improving at relatively similar rates as seen in Figure 6.

2 Neural Networks with TensorFlow

This section details the construction of a neural net using TensorFlow and the ADAM optimizer. The structure of the model is described below:

- **Convolutional Input Layer** - The input to the neural net is reshaped back into its original 28×28 pixel shape to allow 32 3×3 filters to pass over the image. Each layer uses vertical and horizontal strides of 1 resulting in 676 convolutions for each filter.

  Convolutional Neural networks are most commonly used for image recognition. The convolution operation emulates the visual response of an individual neuron to visual stimuli. During the forward pass of a convolutional layer, a 2 dimensional activation map is created by computing the dot product between the entries of the filter and the input to the layer. This is repeated across with width and height of the input volume, allowing the network to learn filters which activate when a specific type of feature at some spatial position is detected. These activation maps are then stacked to form the output volume of the layer.

- **Relu Activation layer** - This is a standard Relu activation layer, which maps the output from the convolutional layer to zero, if the value is less than zero, and otherwise leaves the output unchanged.

- **Batch Normalization Layer** - This layer normalizes the output of the relu activation, to ensure that the values do not vary in magnitude by a large amount. This allows the training time of the neural net to decrease, and also reduces the sensitivity to network initialization.

- **Max Pooling Layer** - This layer combines the outputs of the neuron clusters of the previous layer into a single neuron in the subsequent layer. Max pooling takes the maximum value from each cluster in the previous layer. Pooling layers reduce the spatial size of the representation in the network. This allows for the reduction of the amount of parameters and computation required within the network.

- **Flatten Layer** - Once the number of parameters has been reduced by the pooling layer, the output is flattened into a 2 dimensional array, of dimension number of input samples by (input width $\times$ input height $\times$ number of filters).

- **Fully Connected Layer** - This is a standard neural network perceptron hidden layer, where the input is multiplied by weights and a bias is added.

- **(Optional) Dropout Layer** - A dropout layer removes random neurons within a layer by ignoring them during the forward and backward pass. For each hidden layer, for each training sample, and for each iteration, a random fraction of nodes is dropped. Individual nodes are ignored with a probability '1-p'. This reduces overfitting of the model by preventing neurons from developing co-dependency on one another. It is an efficient method of model averaging in a neural network.

- **Relu Activation Layer** - This is a standard Relu activation layer, which maps the output from the convolutional layer to zero, if the value is less than zero, and otherwise leaves the output unchanged.

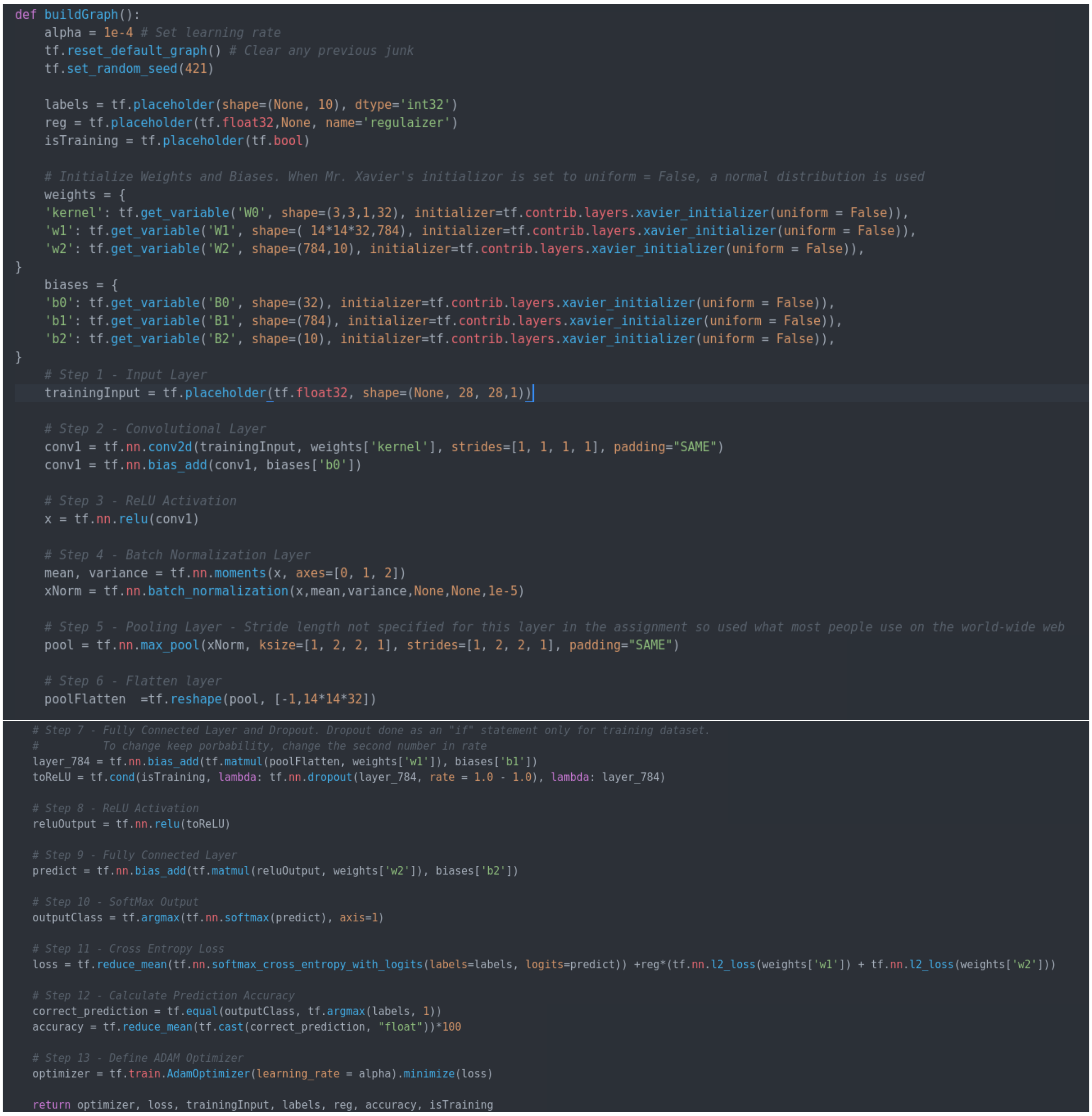
- **Fully Connected Layer** - This is a standard neural network perceptron hidden layer, where the input is multiplied by weights and a bias is added.

- **Softmax Output Layer** - This is the final activation layer, where the second fully connected layer output is fed through the softmax function. This maps the K outputs of the second fully connected layer to a normalized probability distribution consisting of K probabilities, each probability being in the interval of 0 and 1, and all probabilities adding up to zero. This allows us to classify the different training samples as a letter with a probability of it being that letter.

- **Cross Entropy Loss Function** - The loss of the neural net is calculated by comparing the target label of a sample with the highest probability output from the softmax layer for each sample.

2.1 Model Implementation

The following code snippet summarizes the implementation of the TensorFlow CNN.



```python
def buildGraph():
    alpha = 1e-4 # Set learning rate
    tf.reset_default_graph() # Clear any previous junk
    tf.set_random_seed(421)

    labels = tf.placeholder(shape=(None, 10), dtype='int32')
    reg = tf.placeholder(tf.float32,None, name='regulaizer')
    isTraining = tf.placeholder(tf.bool)

    # Initialize Weights and Biases. When Mr. Xavier's initializor is set to uniform = False, a normal distribution is used
    weights = {
    'kernel': tf.get_variable('W0', shape=(3,3,1,32), initializer=tf.contrib.layers.xavier_initializer(uniform = False)),
    'w1': tf.get_variable('W1', shape=( 14*14*32,784), initializer=tf.contrib.layers.xavier_initializer(uniform = False)),
    'w2': tf.get_variable('W2', shape=(784,10), initializer=tf.contrib.layers.xavier_initializer(uniform = False)),
}
    biases = {
    'b0': tf.get_variable('B0', shape=(32), initializer=tf.contrib.layers.xavier_initializer(uniform = False)),
    'b1': tf.get_variable('B1', shape=(784), initializer=tf.contrib.layers.xavier_initializer(uniform = False)),
    'b2': tf.get_variable('B2', shape=(10), initializer=tf.contrib.layers.xavier_initializer(uniform = False)),
}
    # Step 1 - Input Layer
    trainingInput = tf.placeholder(tf.float32, shape=(None, 28, 28,1))

    # Step 2 - Convolutional Layer
    conv1 = tf.nn.conv2d(trainingInput, weights['kernel'], strides=[1, 1, 1, 1], padding="SAME")
    conv1 = tf.nn.bias_add(conv1, biases['b0'])

    # Step 3 - ReLU Activation
    x = tf.nn.relu(conv1)

    # Step 4 - Batch Normalization Layer
    mean, variance = tf.nn.moments(x, axes=[0, 1, 2])
    xNorm = tf.nn.batch_normalization(x,mean,variance,None,None,1e-5)

    # Step 5 - Pooling Layer - Stride length not specified for this layer in the assignment so used what most people use on the world-wide web
    pool = tf.nn.max_pool(xNorm, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME")

    # Step 6 - Flatten layer
    poolFlatten  =tf.reshape(pool, [-1,14*14*32])
```

```python
    # Step 7 - Fully Connected Layer and Dropout. Dropout done as an "if" statement only for training dataset.
    #          To change keep porbability, change the second number in rate
    layer_784 = tf.nn.bias_add(tf.matmul(poolFlatten, weights['w1']), biases['b1'])
    toReLU = tf.cond(isTraining, lambda: tf.nn.dropout(layer_784, rate = 1.0 - 1.0), lambda: layer_784)

    # Step 8 - ReLU Activation
    reluOutput = tf.nn.relu(toReLU)

    # Step 9 - Fully Connected Layer
    predict = tf.nn.bias_add(tf.matmul(reluOutput, weights['w2']), biases['b2'])

    # Step 10 - SoftMax Output
    outputClass = tf.argmax(tf.nn.softmax(predict), axis=1)

    # Step 11 - Cross Entropy Loss
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=labels, logits=predict)) +reg*(tf.nn.l2_loss(weights['w1']) + tf.nn.l2_loss(weights['w2']))

    # Step 12 - Calculate Prediction Accuracy
    correct_prediction = tf.equal(outputClass, tf.argmax(labels, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))*100

    # Step 13 - Define ADAM Optimizer
    optimizer = tf.train.AdamOptimizer(learning_rate = alpha).minimize(loss)

    return optimizer, loss, trainingInput, labels, reg, accuracy, isTraining
```

Figure 9: TensorFlow CNN buildGraph Function

2.2 Model Training

The following figures show the training, validation and test loss and accuracy curves for the SGD TensorFlow model which uses a batch size of 32, trained for 50 epochs, with a learning rate of $\alpha = 10^{-4}$:

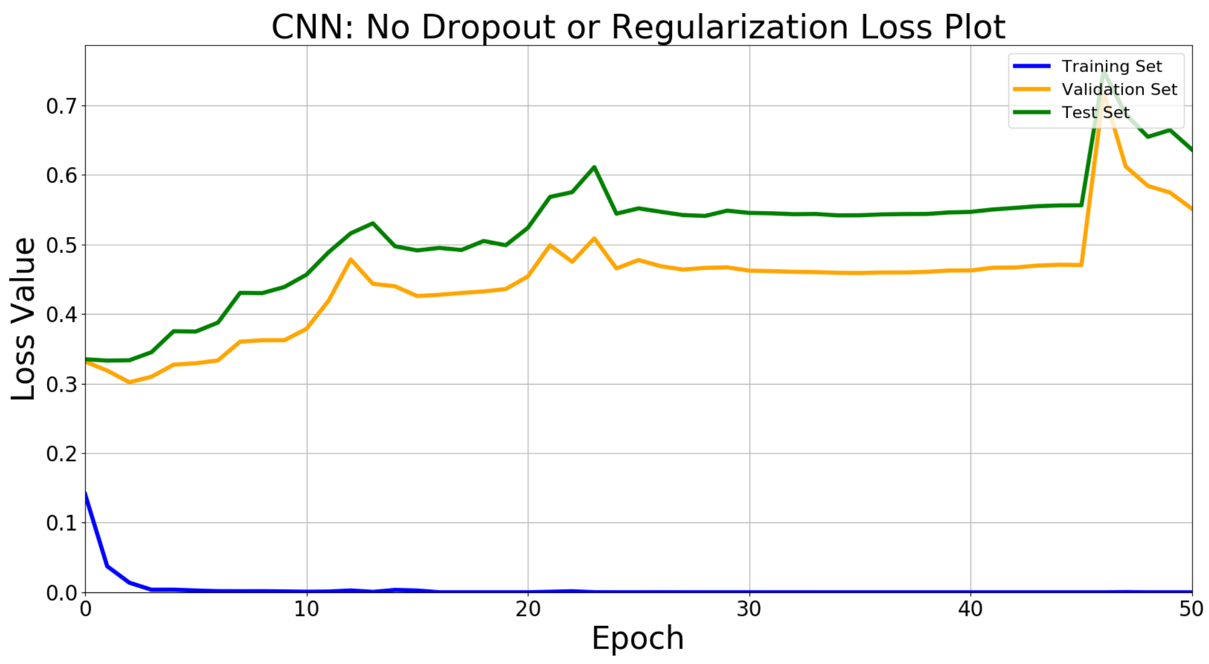


Figure 10: Training, validation and test loss versus epoch for the TensorFlow implementation.

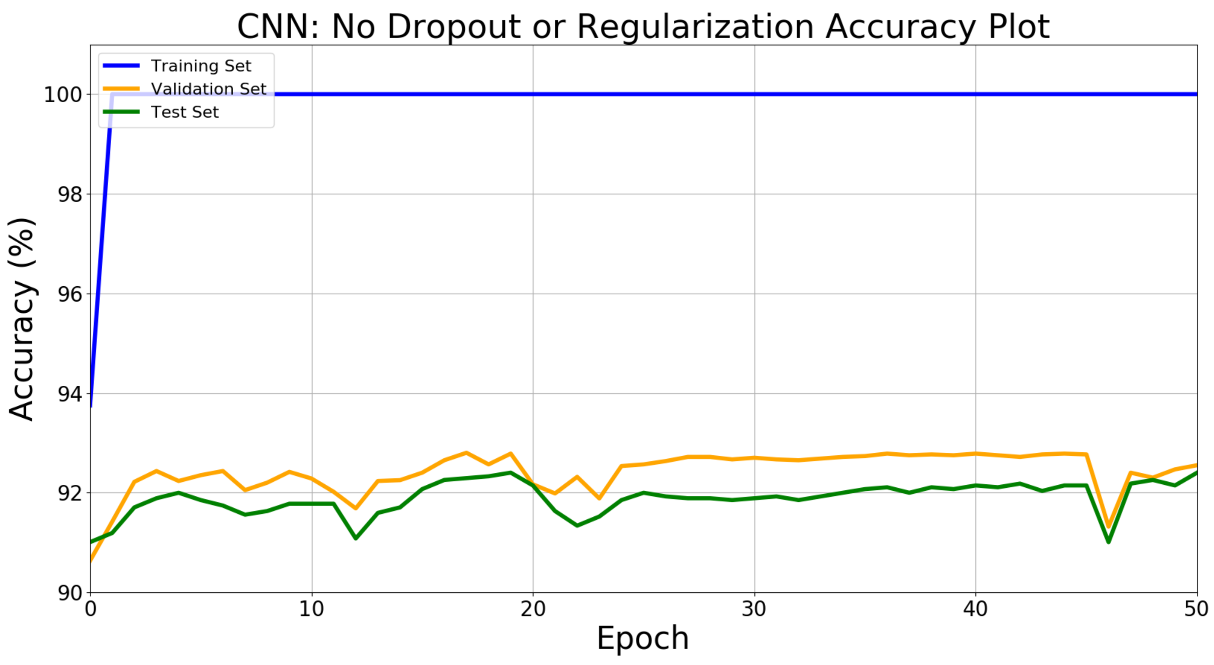


Figure 11: Training, validation and test accuracy versus epoch for the TensorFlow implementation.

2.3 Hyperparamater Investigation

2.3.1 L2 Normalization

The effects of L2 normalization on the TensorFlow model were investigated. The regularization constant was varied between values $\lambda = \{0.01, 0.1, 0.5\}$. The accuracies of the model for each regularization value are plotted below, and summarized in a table at the bottom.

Data Subset	λ	Final CE Loss	Final Accuracy
Train	0.01	0.118	100%
Valid	0.01	0.370	92.92%
Test	0.01	0.391	92.55%
Train	0.1	0422	93.75%
Valid	0.1	0.508	92.70%
Test	0.1	0.506	92.99%
Train	0.5	1.063	93.75%
Valid	0.5	0.970	89.82%
Test	0.5	0.961	90.31%

Table 4: Loss and accuracy for the model when using regularization λ.

Representing visually, we compared the different values of λ on the test set. This is shown in the figure below:

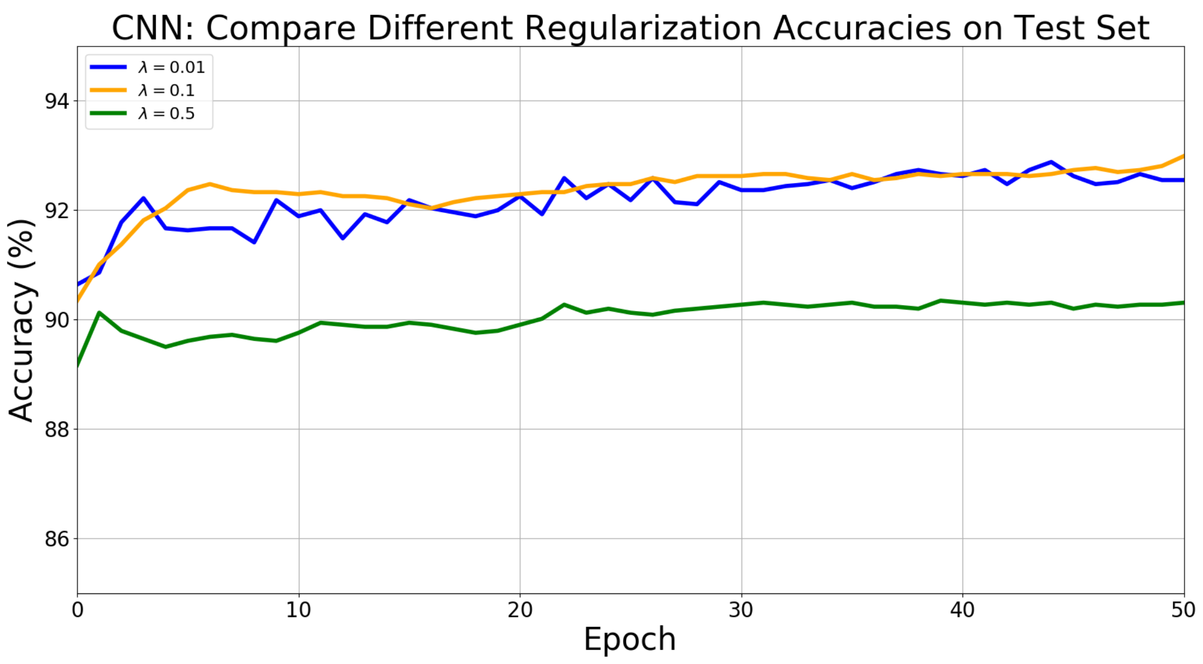


Figure 12: Test accuracy versus epoch for different λs

As seen from the graph, when $\lambda = 0.5$ the accuracy drops a couple of percent. This is likely due to under-fitting. Between $\lambda = 0.01$ and $\lambda = 0.1$, the difference on the final test set is small. To eliminate the risk of over-fitting, it is probably more optimal to use the higher value to be more conservative.

2.3.2 Dropout

The effects on dropout were investigated to determine its impact on the overfitting within the model. The dropout probability was varied in value between $p = \{0.9, 0.75, 0.5\}$, and the accuracies of the model for each dropout probability are plotted below, then summarized in a table at the bottom.

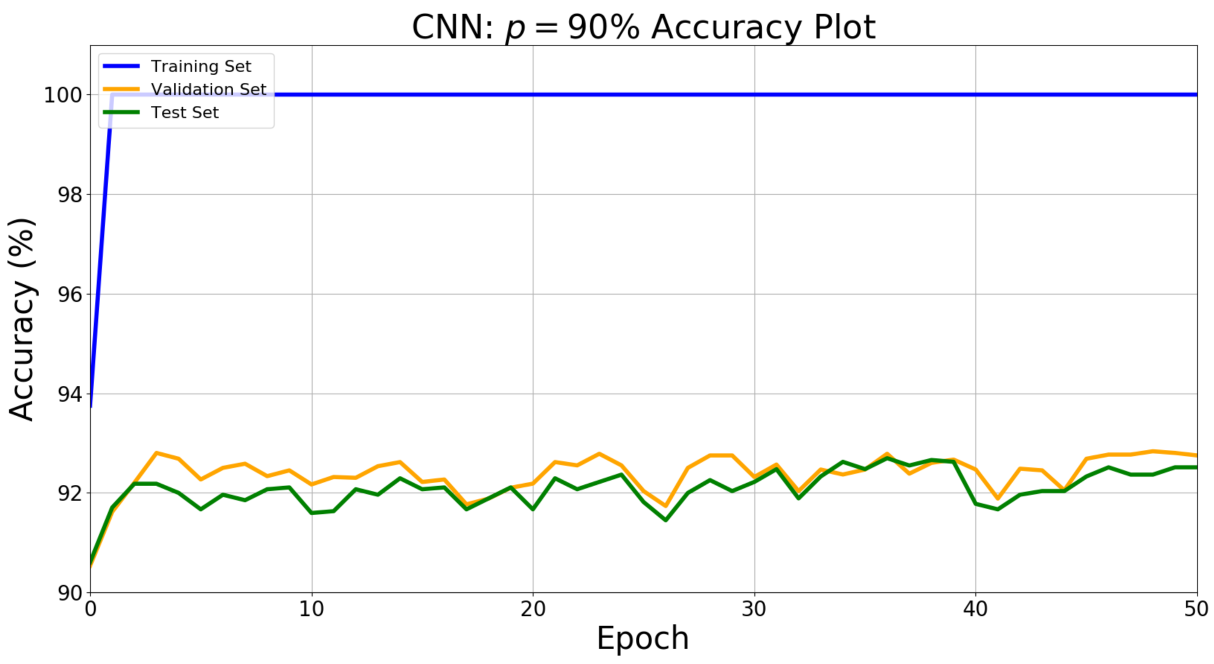


Figure 13: Training, validation and test accuracy versus epoch for the TensorFlow implementation with $p = 0.9$.

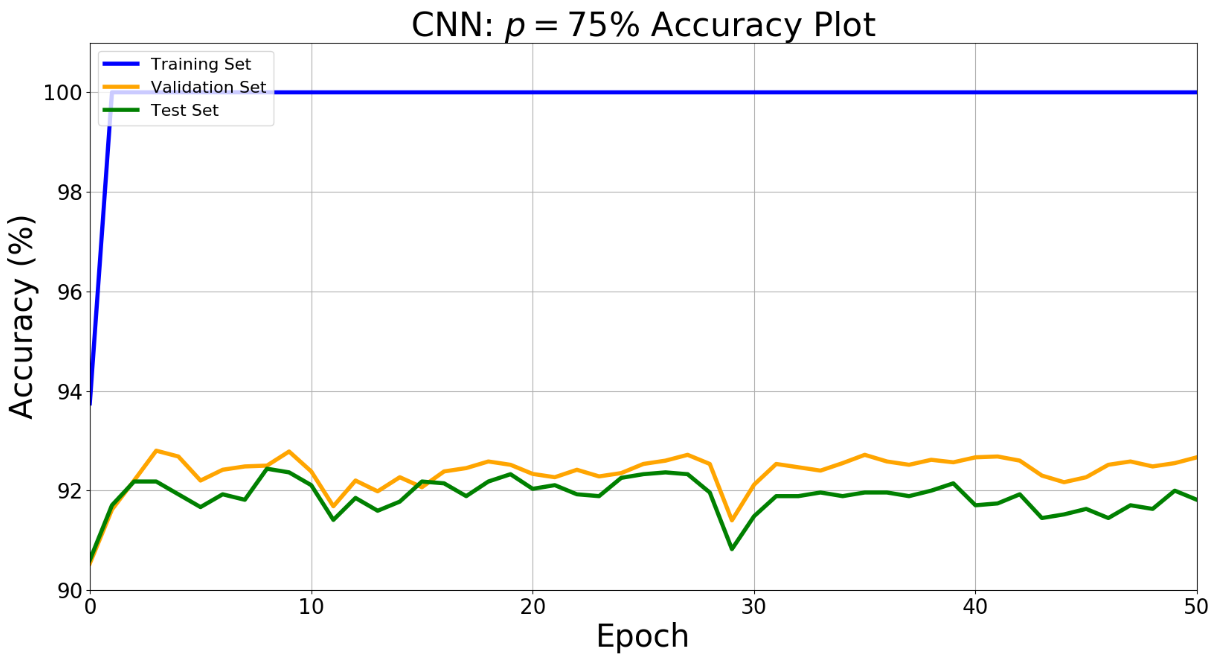


Figure 14: Training, validation and test accuracy versus epoch for the TensorFlow implementation with $p = 0.75$.

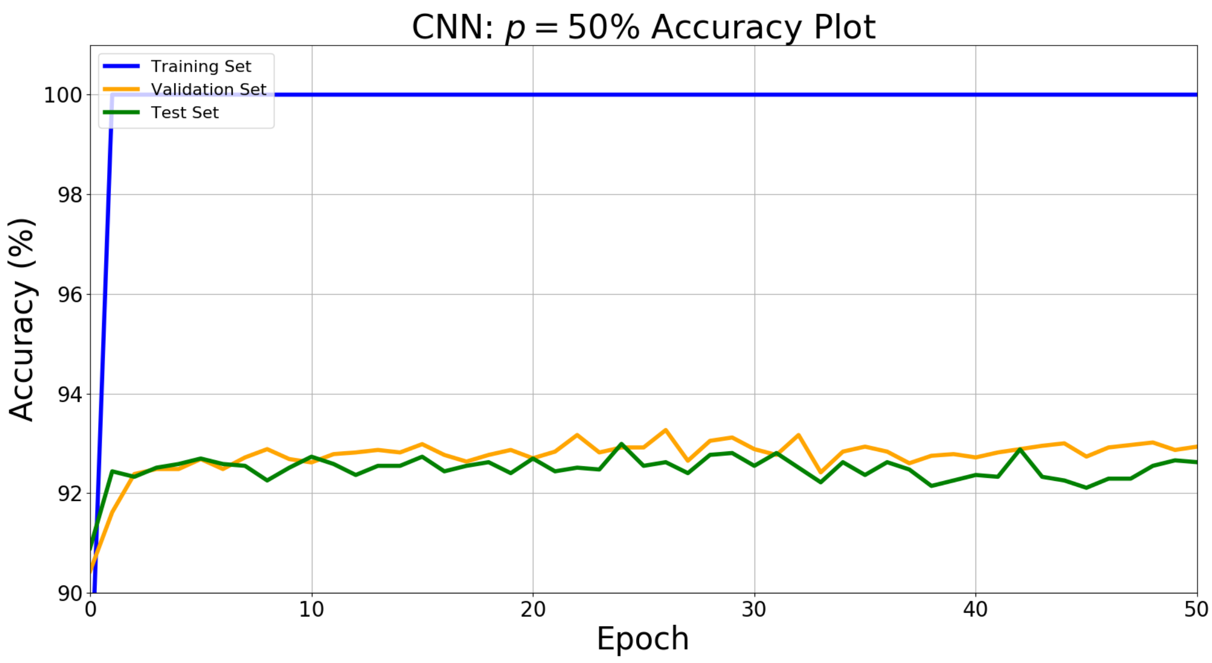


Figure 15: Training, validation and test accuracy versus epoch for the TensorFlow implementation with $p = 0.5$.

Data Subset	p	Final CE Loss	Final Accuracy
Train	0.9	1.196e-5	100%
Valid	0.9	0.568	92.75%
Test	0.9	0.656	92.51%
Train	0.75	5.111e-6	100%
Valid	0.75	0.583	92.67%
Test	0.75	0.668	91.18%
Train	0.5	4.560e-6	100%
Valid	0.5	0.536	92.93%
Test	0.5	0.589	92.62%

Table 5: Loss and accuracy for the model when using dropout probability p.

As seen in the figures above and Table 5, varying the drop-out rate had minimal effect on improving the accuracy of the CNN. This might be due to the fact that the drop-out layer was added towards the end of the CNN, as opposed to the middle of it. Another reason why drop-out didn't impact the final accuracy might also be to the fact that the CNN was not deep enough for it to have an affect.

3 Concluding Remarks

In this report, 2 types of neural nets where implemented to classify 10 letters: a conventional one using Numpy, and a convolutional network using TensorFlow. The Numpy model's biggest advantage is its simplicity. Using just one hidden layer, we were able to predict letters with an approximately 90% success rate. You can visualize the predicted images using the *playGame* function. 9 out of 10 times, the model predicts the right letter.

In some cases, 90% accuracy is not good enough. To try to achieve higher accuracy, a Convolutional Neural Network was implemented. Even though this model had a more complicated set-up, it ran much faster than the Numpy model. This is due to the use of a GPU and a CPU to run this model as opposed to using just the CPU in the former. The increase in complexity enabled us to achieve approximately a 2.5% improvement over the Numpy model. That may not seem like much, but that small percent improvement translates into 68 more images being classified correctly from the test dataset. Between changing the regularization and changing the percent dropout, we felt the former had a stronger impact on classifying the data more accurately. Unlike the Numpy model, the CNN model was able to classify the training set with 100% accuracy, showing the strength of this network. However, the challenge of replicating that accuracy on validation and test sets remains, due to the variation between samples.