ECE421 Introduction to Machine Learning

Assignment 2:

Neural Networks

Due Date: Friday, March 6, 2020

|  |  |
| --- | --- |
| **FIRST NAME** | Saud |
| **LAST NAME** | Badar |
| **STUDENT #** | 1002554595 |
| **SECTION:** | LEC: 0102 TUT: 0101 |

|  |  |
| --- | --- |
| **FIRST NAME** | Kyu Bum |
| **LAST NAME** | Kim |
| **STUDENT #** | 1003969100 |
| **SECTION:** | LEC: 0102 TUT: 0101 |

**1 Neural Networks using Numpy [14 pts]**

**1.1 Helper Functions [4 pts]**

1. *ReLU():* ReLU(x) = max(x, 0)

|  |
| --- |
| **def** **relu**(x):  **return** np.maximum(0, x) |

1. *softmax():* σ(**z**)j= , j = 1 . . . K, for K classes.

|  |
| --- |
| **def** **softmax**(x):  **return** (np.exp(x - np.max(x))) / np.sum(np.exp(x - np.max(x))) |

1. *compute():*

|  |
| --- |
| **def** **computeLayer**(X, W, b):  layer = np.matmul(W.transpose(), X) + b  **return** (layer) |

1. *averageCE():*

**def** **averageCE**(true, prediction):  
 **return** (-np.mean(true\*np.log(prediction)))

1. *gradCE():*

So= , keep in mind **s =** softmax (o) = σ(**o**)j= , j = 1 . . . K, for K classes a

And **o** = WTX + b

First let's find :

LCE = - y(log(s))

= -y()

Secondly we need to find the partial derivative of the sigmoid function with respect to the input ():

=

=

= s(1 - s)

When we multiply both the values together we get:

=

= [-y()][s(1 - s)]

=( s - y)

|  |
| --- |
| **def** **gradCE**(target, prediction):  **return** prediction - target |

**1.2 Backpropagation Derivatives [4 pts]**

We need to make the assumption that we have **N** inputs:

1. , the gradient of the loss with respect to the outer layer weights. Shape: (K × 10), with K units. We need to keep in mind the output layer has an softmax activation function. So we need to solve:

=

Zo = WjkL Sk(L-1)+bjL

= Sk(L-1) = **Sh** ε RN x K is the output of the hidden layer

=

= so(1 - so)

= -y ()

= (so - y)

**= ShT(so - y)**

**Zo** and **so** are both the input and output of the outer layer respectively.

|  |
| --- |
| **def** **back\_wrt\_weights\_out**(predication, true, hidden\_output):  softmax\_grad = gradCE(predication, true)  grad\_weight\_out = np.matmul(hidden\_output.transpose(), softmax\_grad)  **return**(grad\_weight\_out) |

1. , the gradient of the loss with respect to the outer layer biases. Shape: (1 × 10).

=

= **1**, from the previous step we know = (so - y)

= **1T(so - y)**

Here the **1**, is a vector in which all elements are ones. **1** ε RN x 1

|  |
| --- |
| **def** **back\_wrt\_bias\_out**(predication, true):  softmax\_grad = gradCE(predication, true)  ones = np.ones((1, true.shape[0]))  grad\_bias\_out = np.matmul(ones, softmax\_grad)  **return**(grad\_bias\_out) |

1. , the gradient of the loss with respect to the hidden layer weights. Shape: (F × K), with F features, K units. We need to keep in mind that the hidden output has an ReLu activation function. Due to this we have:

=

= (so - y)

Zo= WjkL Sk(L-1)+bjL

= Wo

= Si

is 0 if Zh is < 0 and 1 if Zh is > 0

**= SiT()(so - y)WoT**

**Zh** and **Si** is the input to the hidden layer and the output of output of the input layer.

|  |
| --- |
| **def** **back\_wrt\_weights\_hidden**(predictions, true, weights\_out, inputs, hidden\_in):  softmax\_grad = gradCE(predication, true)  **if** hidden\_in > 0:  hidden\_in = 1  **else**:  hidden\_in = 0  grad\_hidden\_weight\_partial = hidden\_in \* np.matmul(softmax\_grad, weights\_out.transpose())  grad\_hidden\_weight = np.matmul(inputs.transpose(), grad\_hidden\_weight\_partial)  **return**(grad\_hidden\_weight) |

1. , the gradient of the loss with respect to the hidden layer biases. Shape: (1 × K), with K units.

=

= (so - y)

= Wo

= is 0 if Zh is < 0 and 1 if Zh is > 0

= **1**

**= 1T(so - y)WoT**

Here the **1**, is a vector in which all elements are ones. **1** ε RK x 1

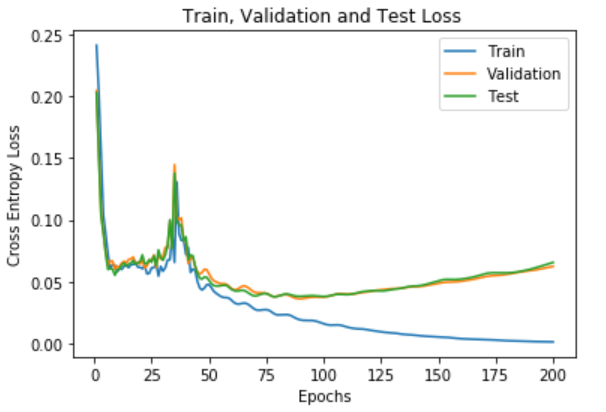
|  |
| --- |
| **def** **back\_wrt\_bias\_hidden**(predictions, true, weights\_out, hidden\_in):  softmax\_grad = gradCE(predication, true)  **if** hidden\_in > 0:  hidden\_in = 1  **else**:  hidden\_in = 0  grad\_hidden\_weight\_partial = hidden\_in \* np.matmul(softmax\_grad, weights\_out.transpose())  **return**(grad\_hidden\_weight) |

**1.3 Learning [6 pts]**

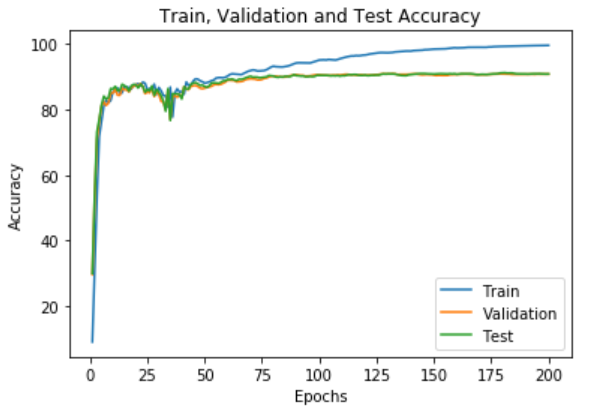
The plots below represent the training curve obtained when testing the notMNIST dataset with the given default hyperparameters

|  |
| --- |
| **def** **NN**():  np.random.seed(21)  trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()  trainTarget, validTarget, testTarget = convertOneHot(trainTarget, validTarget, testTarget)   *#Change the shape of all the dataset*  trainData = trainData.reshape((trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))  validData = validData.reshape((validData.shape[0], validData.shape[1]\*validData.shape[2]))  testData = testData.reshape((testData.shape[0], testData.shape[1]\*testData.shape[2]))   *#Set up the parameters*  hidden\_units = 1000  num\_epochs = 200  classes = 10  learning\_rate = 1e-5  gamma = 0.99   *#Set up the array for the loss and accuracy*  train\_loss, val\_loss, test\_loss = [], [], []  train\_acc, val\_acc, test\_acc = [], [], []  *#Set up all the weights and bias following Xiaver initialization scheme*  W\_h = np.random.normal(0, np.sqrt(2.0/(784 + hidden\_units)), (784, hidden\_units))  W\_o = np.random.normal(0, np.sqrt(2.0/(classes + hidden\_units)), (hidden\_units, classes))   b\_h = np.zeros((1, hidden\_units))  b\_o = np.zeros((1, classes))   *#Set up all the learning weights*  vW\_h = np.full((784, hidden\_units), 1e-5)  vW\_o = np.full((hidden\_units, classes), 1e-5)  vb\_h = np.full((1, hidden\_units), 1e-5)  vb\_o = np.full((1, classes), 1e-5)   **for** epoch **in** range(num\_epochs):  *#Shuffle the data each epoch*  trainData, trainTarget = shuffle(trainData, trainTarget)   *#Forward propagation*  s\_h, x\_h, s\_o, x\_o = forward\_propagation(trainData, W\_h, b\_h, W\_o, b\_o)   *#Back propagation*  grad\_W\_h, grad\_W\_o, grad\_b\_h, grad\_b\_o = back\_propagation (trainTarget, trainData, s\_h, s\_o, x\_h, W\_o)   *#Change the learning weights*  vW\_h = (learning\_rate\*grad\_W\_h) + (gamma\*vW\_h)  vW\_o = (learning\_rate\*grad\_W\_o) + (gamma\*vW\_o)  vb\_h = (learning\_rate\*grad\_b\_h) + (gamma\*vb\_h)  vb\_o = (learning\_rate\*grad\_b\_o) + (gamma\*vb\_o)   *#Add the learning weight to the actual weights*  W\_h = W\_h - vW\_h  W\_o = W\_o - vW\_o  b\_h = b\_h - vb\_h  b\_o = b\_o - vb\_o   *#Find the accuracy for the training dataset*  acc = accuracy(x\_o, trainTarget)  train\_acc.append(acc\*100)   *#Find the loss for the training dataset*  loss = CE(trainTarget, x\_o)  train\_loss.append(loss)   *#Find the x\_o for the validation set*  s\_h\_val, x\_h\_val, s\_o\_val, x\_o\_val = forward\_propagation(validData, W\_h, b\_h, W\_o, b\_o)   *#Find the accuracy for the validation dataset*  acc = accuracy(x\_o\_val, validTarget)  val\_acc.append(acc\*100)   *#Find the loss for the validation dataset*  loss = CE(validTarget, x\_o\_val)  val\_loss.append(loss)   *#Find the x\_o for the test set*  s\_h\_test, x\_h\_test, s\_o\_test, x\_o\_test = forward\_propagation(testData, W\_h, b\_h, W\_o, b\_o)   *#Find the accuracy for the testing dataset)*  acc = accuracy(x\_o\_test, testTarget)  test\_acc.append(acc\*100)   *#Find the loss for the testing dataset*  loss = CE(testTarget, x\_o\_test)  test\_loss.append(loss)   print("Epoch: {}".format(epoch))    *#First Graph for the Loss*  n = len(train\_loss)  plt.title("Train, Validation and Test Loss ")  plt.plot(range(1,n+1), train\_loss, label="Train")  plt.plot(range(1,n+1), val\_loss, label="Validation")  plt.plot(range(1, n+1), test\_loss, label = "Test")  plt.xlabel("Epochs")  plt.ylabel("Cross Entropy Loss")  plt.legend(loc='best')  plt.show()   *#Second Graph for the Accuracy*  n = len(train\_acc)  plt.title("Train, Validation and Test Accuracy")  plt.plot(range(1,n+1), train\_acc, label="Train")  plt.plot(range(1,n+1), val\_acc, label="Validation")  plt.plot(range(1, n+1), test\_acc, label = "Test")  plt.xlabel("Epochs")  plt.ylabel("Accuracy")  plt.legend(loc='best')  plt.show() |

Loss Graph (epoch = 200, **hidden units = 1000**, gamma = 0.99 and learning rate = 1e-5)



Accuracy Graph (epoch = 200, **hidden units = 1000**, gamma = 0.99 and learning rate = 1e-5)



We used three important functions for our computation *forward\_propagation, backward\_propagation* and *accuracy*. The name of the functions tell us the purpose of these three functions.

|  |
| --- |
| **def** **forward\_propagation**(x, W\_h, b\_h, W\_o, b\_o):  s\_h = computeLayer(x, W\_h, b\_h)  x\_h = relu(s\_h)  s\_o = computeLayer(x\_h, W\_o, b\_o)  x\_o = softmax(s\_o)  **return** (s\_h, x\_h, s\_o, x\_o) |

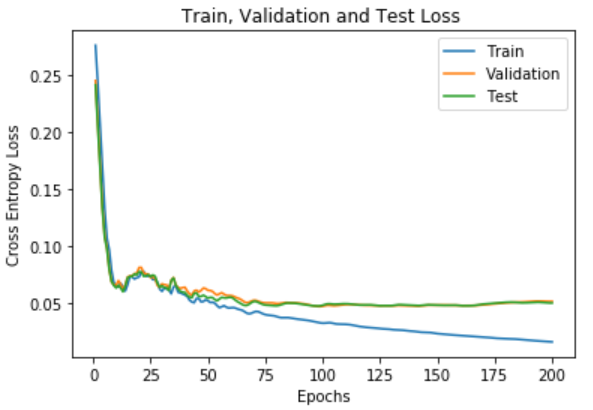
|  |
| --- |
| **def** **back\_propagation**(trainTarget, trainData, s\_h, s\_o, x\_h, W\_o):  change\_out = gradCE(trainTarget, s\_o)  grad\_relu = gradRelu(s\_h)  step = np.matmul(change\_out,W\_o.transpose())  change\_hidden = grad\_relu \* step   grad\_W\_h = np.matmul(trainData.transpose(), change\_hidden)  grad\_W\_o = np.matmul(x\_h.transpose(), change\_out)  grad\_b\_h = np.sum(change\_hidden, axis = 0)  grad\_b\_o = np.sum(change\_out, axis = 0)  **return**(grad\_W\_h, grad\_W\_o, grad\_b\_h, grad\_b\_o)  **def** **accuracy**(prediction, target):   *#(b, 10) (b, 10)*  *# print(prediction)*  max\_indices\_pred = prediction.argmax(axis=1)  indices\_target = target.argmax(axis=1)  *# print(max\_indices\_pred, indices\_target)*  accurate\_arr = np.equal(max\_indices\_pred, indices\_target)  **return** np.sum((accurate\_arr==**True**))/(indices\_target.shape[0]) |

**1.4 Hyperparameter Investigation [4 pts]**

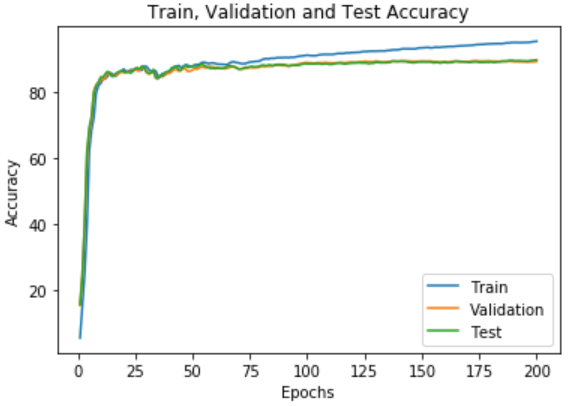
**1.4.1 Number of hidden units [2 pts]**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Hidden Units** | **100** | **500** | **1000** | **2000** |
| **Test Accuracy** | **89.54%** | **90.86%** | **90.89%** | **91.37%** |

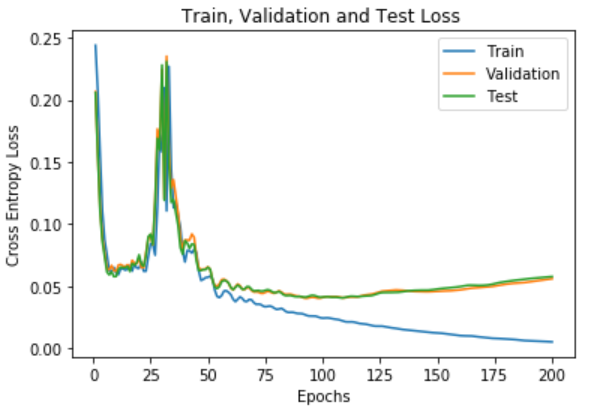
Loss Graph (epoch = 200, **hidden units = 100**, gamma = 0.99 and learning rate = 1e-5)



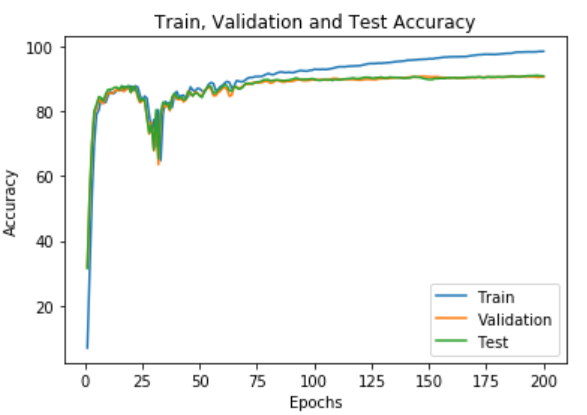
Accuracy Graph (epoch = 200, **hidden units = 100,** gamma = 0.99 and learning rate = 1e-5)



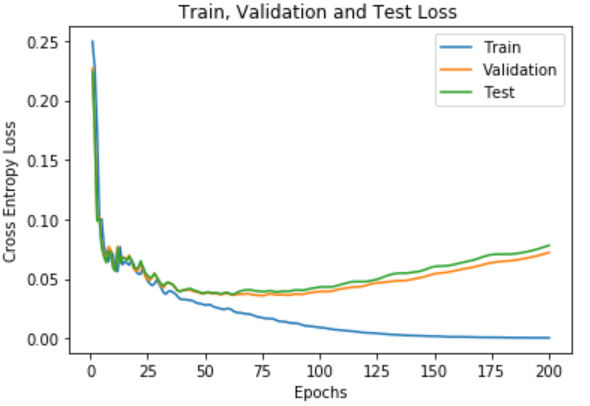
Loss Graph (epoch = 200, **hidden units = 500**, gamma = 0.99 and learning rate = 1e-5)



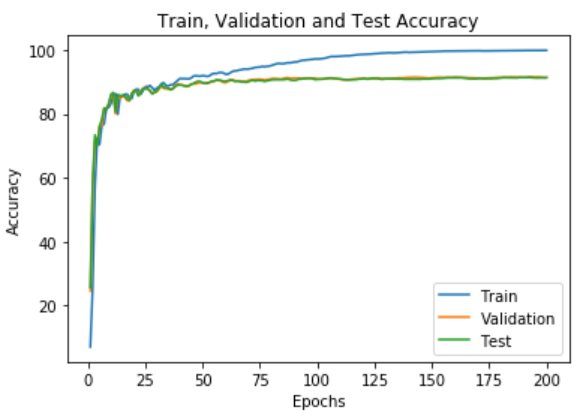
Accuracy Graph (epoch = 200, **hidden units = 500,** gamma = 0.99 and learning rate = 1e-5)



Loss Graph (epoch = 200, **hidden units = 2000**, gamma = 0.99 and learning rate = 1e-5)



Accuracy Graph (epoch = 200, **hidden units = 2000,** gamma = 0.99 and learning rate = 1e-5)



As shown in the plots above, increasing the number of hidden units within the neural network results in a faster approach to the optimal weight values. This can be shown through the training curves. As the hidden units increase, the training error and loss decrease at a faster rate as the optimal has been reached resulting in slight overfitting of the data. This can be seen through the 2000 hidden units plots compared to the 100 hidden units plots. When the training reaches around 75 epochs, the 2000 hidden unit plots training accuracy and loss begins to deviate from the valid and test at a greater rate than the 100 hidden units. You can observe this gradual change through all three sets of plots. This is because, as the number of neurons increases, the input data is distributed more sparsely among all the neurons allowing for more features to be handled at once. Therefore, each feature will be determined by more neurons allowing for a faster convergence of the optimal solution. With less neurons, each neuron will take in more information causing the optimal solution to be harder to reach, as weights will change based on more features. More features means more errors when calculating the error. Therefore, more hidden units results in faster convergence and a larger possibility of overfitting.

**1.4.2 Early Stopping [2 pts]**

From the graphs above we can see that around 75 epochs, the accuracy of the training and the validation and testing accuracy start to diverge from one another. We can make a rough guess that the graph will be overfitting to the training dataset around 75 epochs. The accuracy begins to stabilize for the testing and validation set, while the training accuracy keeps on increasing.

|  |  |  |  |
| --- | --- | --- | --- |
| **Datasets** | **Training Dataset** | **Validation Dataset** | **Testing Dataset** |
| **Early Stopping (75 epochs)** | **90.94%** | **89.57%** | **89.97%** |

**2 Neural Networks in Tensorflow [14 pts]**

**2.1 Model Implementation [4 pts]**

So let's start off by describing our neural network architecture. We are given an image of the size of 28 x 28 = 728. For the dimension of the input we need to use the placeholder which will be of the form of [None, 28, 28, 1], the *None* in this case is for the batch size. As we don’t know the batch size we will leave it empty. We also have an output placeholder which has the shape of [ None, 10].

We will also initialize all three pairs of weight and biases to the Xavier scheme. The first pair is input connected to the convolutional layer, the second pair connects the output of convolutional layer to the input first fully connected layer. The last pair connects the output of the first fully connected layer to the the input of the second fully connected layer.

The next part we need to define is the convolutional layer, we know that the convolution layer takes in one channel and outputs 32 channels. It also has a kernel of the size of 3x3, from all this information we can define the weight shape as [3, 3, 1, 32]. The first two values are the shape of the kernel and the third and fourth values are the input channels and output channel respectively. Keep in mind the bias size will also be 32

After the convolutional, our value will pass through the *ReLu* activation function, this will not change the shape of the output. The next step will be the max pooling as the the max pooling has the shape of 2x2, the original image shape of 28 x 28 x 1 will change to 14 x 14 x 1.

Next we will flatten this so we could pass it onto the first fully connected layer, the input of the fully connected layer will be 14 x 14 x 32 and the output will be of 784. This means that the second weight shape is [14 x 14 x 32, 784]. While the bias size will be 784. The last part will be connecting the first fully connected layer to the second fully connected layer. The input size of the the second fully connected layer is 784 and the output size is 10. The weight shape will be [784, 10], while the bias size will be 10.

We also have to create two functions *conv2d()* and *maxpool2d()*. These functions will work as a convolutional and max pool layer respectively. The *conv2d()* function will take in three inputs: *x*, *weight, bias*. While on the other hand the *maxpool2d()* function will take in two inputs: *x* and *size*.

This is an example of how we initized our biases and weights.

|  |
| --- |
| x = tf.placeholder("float", [**None**, 28,28,1]) true = tf.placeholder("float", [**None**, 10]) *# This is the first pair of weights and biases* **with** tf.variable\_scope("Weight\_1", reuse = tf.AUTO\_REUSE):  weight\_in = tf.get\_variable('Weight\_1', shape = (3, 3, 1, 32), initializer = tf.contrib.layers.xavier\_initializer()) **with** tf.variable\_scope("Bias\_1", reuse = tf.AUTO\_REUSE):  bias\_in = tf.get\_variable('Bias\_1', shape = (32), initializer = tf.contrib.layers.xavier\_initializer()) *#This is the second pair of weights and biases for the weights between conv2d and fc1* **with** tf.variable\_scope("Weight\_2", reuse = tf.AUTO\_REUSE):  weight\_fc1 = tf.get\_variable('Weight\_2', shape = (14\*14\*32, 784), initializer = tf.contrib.layers.xavier\_initializer()) **with** tf.variable\_scope("Bias\_2", reuse = tf.AUTO\_REUSE):  bias\_fc1 = tf.get\_variable('Bias\_2', shape = (784), initializer = tf.contrib.layers.xavier\_initializer()) *#This is the third pair of weights and biases for the weights between fc1 and fc2* **with** tf.variable\_scope("Weight\_3", reuse = tf.AUTO\_REUSE):  weight\_fc2 = tf.get\_variable('Weight\_3', shape = (784, 10), initializer = tf.contrib.layers.xavier\_initializer()) **with** tf.variable\_scope("Bias\_3", reuse = tf.AUTO\_REUSE):  bias\_fc2 = tf.get\_variable('Bias\_3', shape = (10), initializer = tf.contrib.layers.xavier\_initializer()) |

|  |
| --- |
| **def** **conv2d**(x, W, b):  x = tf.nn.conv2d(x, W, strides =[1, 1, 1, 1], padding='SAME')  x = tf.nn.bias\_add(x, b)  **return** (x)  **def** **maxpool2d**(x, size):  x = tf.nn.max\_pool(x, ksize=[1, size, size, 1], strides=[1, size, size, 1], padding='SAME')  **return**(x) |

Our *CNN()* will take in one value, it will take in the *x* which is the input we just created using placeholder*.* This function builds up the architecture used in CNN. We also need to keep in mind to implement *softmax()* on the output of this function as described on the handout. We do this as we have a multi-class classification problem.

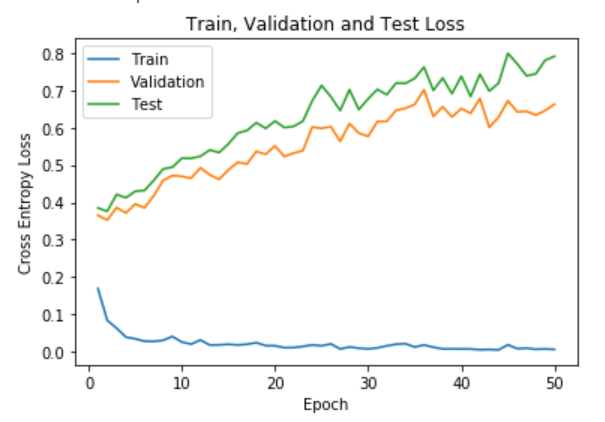
|  |
| --- |
| **def** **CNN**(x):  *#1. Input Layer(This has already been created in the code above)*  *#2.A 3 × 3 convolutional layer, with 32 filters, using vertical and horizontal strides of 1.*  *#(use my own function)*  convolutional\_layer = conv2d(x, weight\_in, bias\_in)  *#3. ReLU activation*  relu\_layer = tf.nn.relu(convolutional\_layer)  *#4. A batch normalization layer*  *#4.1 Get the mean and variance*  batchMean, batchVar = tf.nn.moments(relu\_layer, [0])  *#4.2 Normalizing the data*  normal\_layer = tf.nn.batch\_normalization(relu\_layer, mean = batchMean, variance= batchVar, offset = 0, scale = **None**, variance\_epsilon= 1e-3)  *#5. A 2 × 2 max pooling layer (use my own function)*  max\_pool\_layer = maxpool2d(normal\_layer, 2)  *#6. Flatten layer*  flatten\_layer = tf.reshape(max\_pool\_layer, [-1, weight\_fc1.get\_shape().as\_list()[0]])  *#7. Fully connected layer (with 784 output units, i.e. corresponding to each pixel)*  fully\_connected\_1\_unbiased = tf.matmul(flatten\_layer, weight\_fc1)  fully\_connected\_one = tf.add(fully\_connected\_1\_unbiased, bias\_fc1)  *#8. ReLU activation*  relu\_layer\_two = tf.nn.relu(fully\_connected\_one)  *#9. Fully connected layer (with 10 output units, i.e. corresponding to each class)*  fully\_connected\_2\_unbiased = tf.matmul(relu\_layer\_two, weight\_fc2)  out = tf.add(fully\_connected\_2\_unbiased, bias\_fc2)  **return** (out) *# In this step we get the value of*  output = CNN(x) *#10. Softmax Output*  Softmax = tf.nn.softmax(output) *#11. Cross Entropy loss* cost\_CE = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits\_v2(labels= true, logits = output)) |
|  |

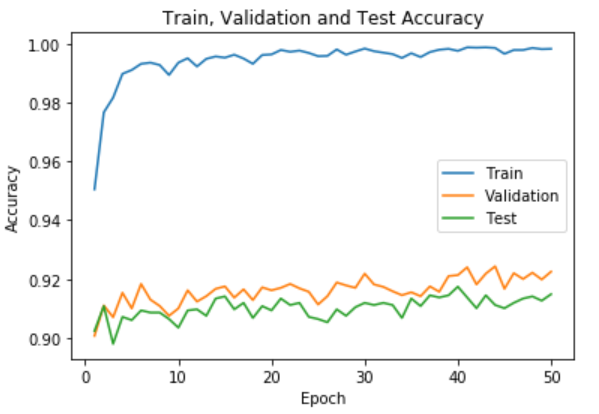
**2.2 Model Training [4 pts.]**

In this section I will go through the how I used SGD to find the local minimum for the weight-loss graph. I created six arrays in total for both the loss and accuracy of the training, validation and testing dataset. I also created a double for loop, the inner for loop was for the batch size and it would optimize the weights in each iteration. The outer for loop was for the number of epochs, it was tasked to keep going until it hit the max epoch. I also estimated all the loss and accuracy factors for each dataset at the end of batch size. After both the for loops were done, I printed the graphs with the help of the six arrays that I created. Keep in mind our learning rate is 0.0001, batch size is 32 and our epoch is 50.

|  |
| --- |
| train\_loss, val\_loss, test\_loss = [], [], [] train\_acc, val\_acc, test\_acc = [], [], []  **with** tf.Session() **as** sess:  sess.run(tf.global\_variables\_initializer())  **for** i **in** range(epochs):  batch\_loop = len(trainData)//batch\_size  *#Second for loop for the batch size*  **for** j **in** range(batch\_loop):  batch\_start = j \* batch\_size  batch\_end = min(((j + 1) \* batch\_size), len(trainData))   batch\_data = trainData[batch\_start:batch\_end]    batch\_end = min(((j + 1) \* batch\_size), len(trainTarget))  batch\_target = trainTarget[batch\_start: batch\_end]  optimize = sess.run(optimizer, feed\_dict= {x: batch\_data, true: batch\_target})    *#Now we will calculate the loss for each iteration*  epoch\_train\_loss = sess.run(cost\_CE, feed\_dict={x: trainData, true: trainTarget})  epoch\_train\_acc = sess.run(final\_acc, feed\_dict ={x: trainData, true: trainTarget})  epoch\_val\_loss = sess.run(cost\_CE, feed\_dict={x: validData, true: validTarget})  epoch\_val\_acc = sess.run(final\_acc, feed\_dict ={x: validData, true: validTarget})  epoch\_test\_loss = sess.run(cost\_CE, feed\_dict={x: testData, true: testTarget})  epoch\_test\_acc = sess.run(final\_acc, feed\_dict ={x: testData, true: testTarget})  *#Now we add it to the list*  train\_loss.append(epoch\_train\_loss)  train\_acc.append(epoch\_train\_acc)  val\_loss.append(epoch\_val\_loss)  val\_acc.append(epoch\_val\_acc)  test\_loss.append(epoch\_test\_loss)  test\_acc.append(epoch\_test\_acc)  *#Now shuffle the data after each epoch*  trainData, trainTarget = shuffle(trainData, trainTarget) *#First Graph for the Loss* n = len(train\_loss) plt.title("Train, Validation and Test Loss") n = len(train\_loss) *# number of epochs* plt.plot(range(1,n+1), train\_loss, label="Train") plt.plot(range(1,n+1), val\_loss, label="Validation") plt.plot(range(1, n+1), test\_loss, label = "Test") plt.xlabel("Epoch") plt.ylabel("Cross Entropy Loss") plt.legend(loc='best') plt.show() *#Second Graph for the Accuracy* plt.title("Train, Validation and Test Accuracy") plt.plot(range(1,n+1), train\_acc, label="Train") plt.plot(range(1,n+1), val\_acc, label="Validation") plt.plot(range(1, n+1), test\_acc, label = "Test") plt.xlabel("Epoch") plt.ylabel("Accuracy") plt.legend(loc='best') plt.show() |

Cross-Entropy Loss (α = 0.0001, batch\_size = 32 and epochs = 50)

Accuracy (α = 0.0001, batch\_size = 32 and epochs = 50)



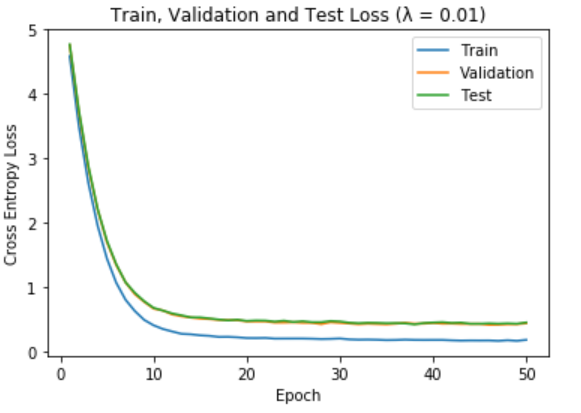
**2.3 Hyperparameter Investigation [6 pts.]**

**2.3.1 L2 Regularization [3 pts.]**

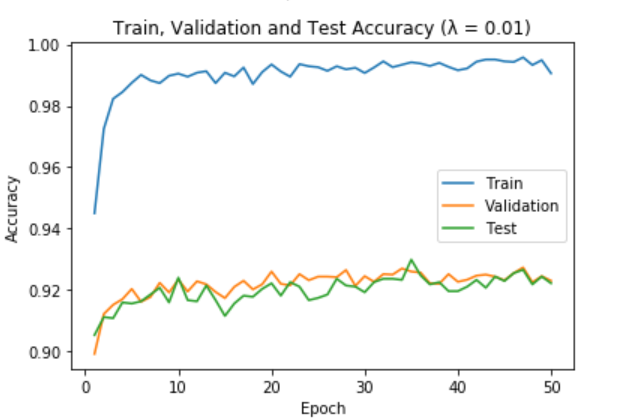
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **λ = 0** | **λ = 0.01** | **λ = 0.1** | **λ = 0.5** |
| **Training Accuracy** | **99.83%** | **99.06%** | **94.83%** | **89.99%** |
| **Validation Accuracy** | **92.25%** | **92.30%** | **91.87%** | **89.15%** |
| **Testing Accuracy** | **91.48%** | **92.21%** | **91.85%** | **89.54%** |

We can see from the plots created by changing the L2 regularization causes the accuracy of the validation and testing set to converge to the training accuracy. Which tells us that the model may not be overfitting. The regularization parameter is directly proportional to the accuracy and if we make the parameter bigger the accuracy of all the dataset comes closer to one another. Increasing the parameter will also introduce more noise to our graph. We can see this occuring when we set λ = 0.5. Overall if we use the regularization parameter is a safe manner we can prevent overfitting while also not introducing noise to the graphs.

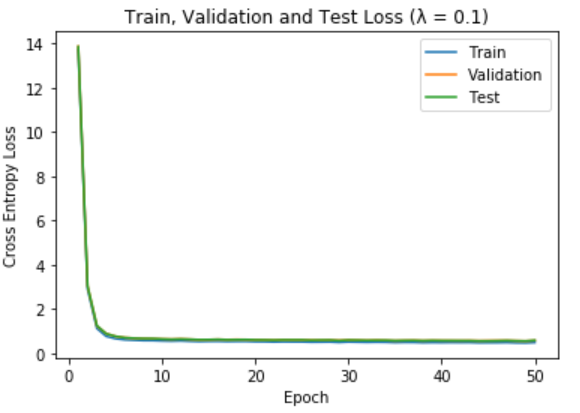
Cross-Entropy Loss (α = 0.0001, batch\_size = 32, epochs = 50 and λ = 0.01)



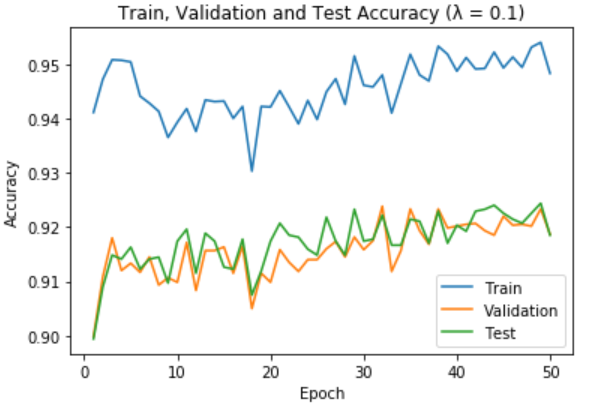
Accuracy (α = 0.0001, batch\_size = 32, epochs = 50 and λ = 0.01)



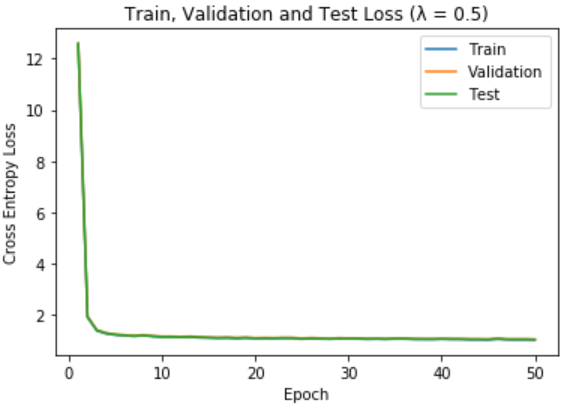
Cross-Entropy Loss (α = 0.0001, batch\_size = 32, epochs = 50 and λ = 0.1)



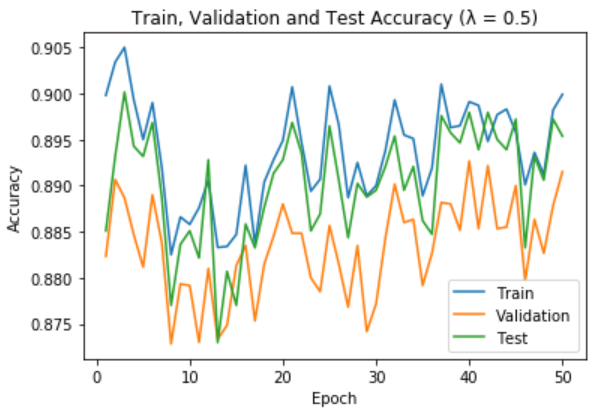
Accuracy (α = 0.0001, batch\_size = 32, epochs = 50 and λ = 0.1)



Cross-Entropy Loss (α = 0.0001, batch\_size = 32, epochs = 50 and λ = 0.5)



Accuracy (α = 0.0001, batch\_size = 32, epochs = 50 and λ = 0.5)

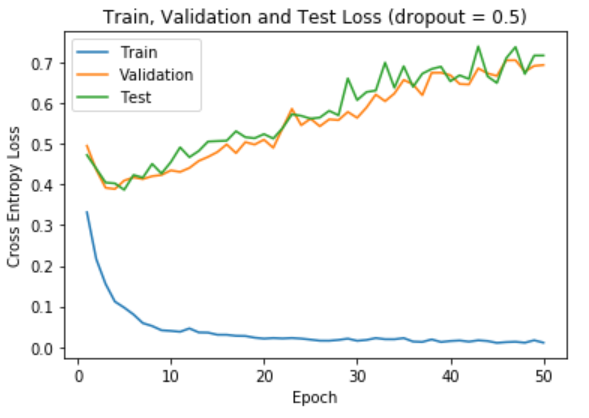


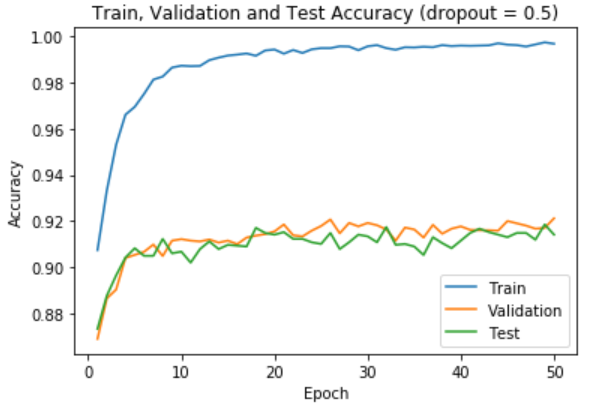
**2.3.2 Dropout [3 pts.]:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Accuracy/Dropout** | **P = 0** | **P = 0.5** | **P = 0.75** | **P = 0.9** |
| **Training Accuracy** | **99.83%** | **99.82%** | **98.93%** | **93.95%** |
| **Validation Accuracy** | **92.25%** | **91.67%** | **91.62%** | **89.13%** |
| **Testing Accuracy** | **91.48%** | **91.96%** | **91.48%** | **88.98%** |

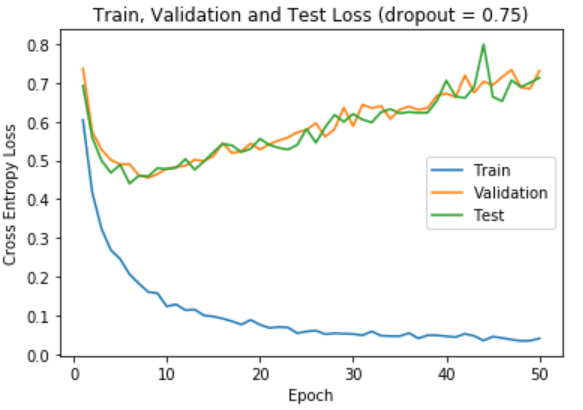
Dropout is another method used to prevent overfitting of the training dataset in the table above. We can see that 0.9 means that 90% of the hidden units will be kept the same while the rest of the hidden units will be set to 0.

Cross-Entropy Loss (α = 0.0001, batch\_size = 32, epochs = 50 and dropout= 0.5)

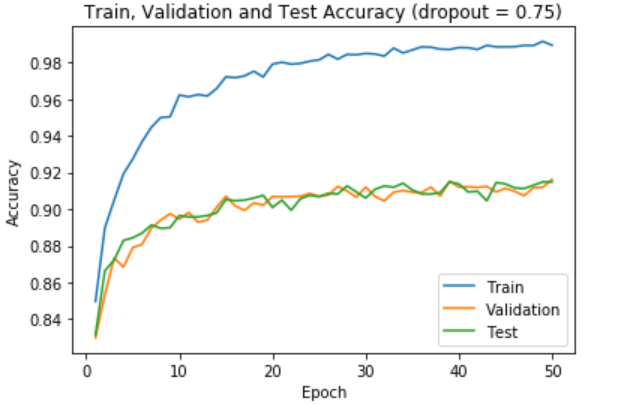
Accuracy (α = 0.0001, batch\_size = 32, epochs = 50 and dropout = 0.5)



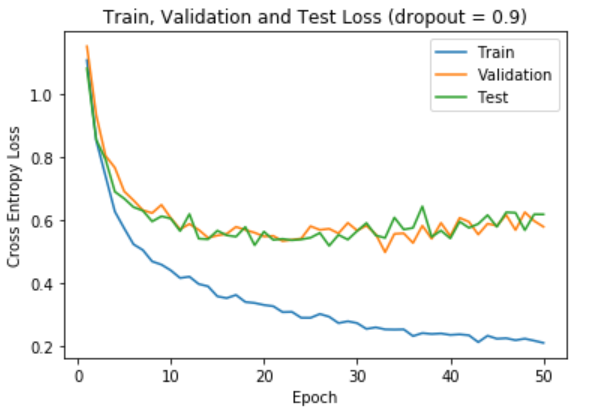
Cross-Entropy Loss (α = 0.0001, batch\_size = 32, epochs = 50 and dropout= 0.75)

****

Accuracy (α = 0.0001, batch\_size = 32, epochs = 50 and dropout = 0.75)

****

Cross-Entropy Loss (α = 0.0001, batch\_size = 32, epochs = 50 and dropout= 0.9)



Accuracy (α = 0.0001, batch\_size = 32, epochs = 50 and dropout = 0.9)

