ECE 421 Assignment 2

Jerry He 1003979180
Contribution: 33%
Calvin Ma 1003803805
Contribution: 33%
Eric Li 1004015852
Contribution: 33%

March 6, 2020

Jerry He 1003979180
Calvin Ma 1003803805
Eric Li 1004015852

ECE 421
March 6, 2020

Assignment 2

1 Neural Networks using Numpy

In this report, matrices will be denoted with bolded symbols ($\boldsymbol{X}$), vectors will be denoted with underlined symbols ($\underline{y}$), and scalars will be denoted with no accent (z).

1.1 Helper Functions

1. *ReLU()*: $\text{ReLU}(x) = \max(x, 0)$

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

2. *softmax()*:

$$\sigma(\boldsymbol{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, j = 1 \dots K, \text{ for } K \text{ classes}$$
$$= \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \frac{e^{-z_{max}}}{e^{-z_{max}}}$$
$$= \frac{e^{z_j - z_{max}}}{\sum_{k=1}^{K} e^{z_k - z_{max}}}$$

Implementing this in vectorized form:

```python
def softmax(z):
  zTilde = z - np.transpose(np.tile(np.amax(z, axis=1), (z.shape[1],1)
    ))
  return np.exp(zTilde)/np.sum(np.exp(zTilde), axis=1)[:,None]
```

3. *compute()*: $\underline{s} = \boldsymbol{W}^T \underline{x} + \underline{b}$ or $\boldsymbol{s} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$

Note that the Python implementation below uses the transposes of $\boldsymbol{x}$, and outputs the transpose of $\boldsymbol{s}$:

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

4. *averageCE()*:

$$averageCE = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_k^n log(p_k^n)$$

Implementing this in vectorized form:

```python
def averageCE(target, prediction):
  # The following line prevents division by zero errors in np.log
  prediction[prediction == 0] = 1e-15
  return -np.sum(np.multiply(target, np.log(prediction))) / target.
    shape[0]
```

Jerry He 1003979180
Calvin Ma 1003803805
Eric Li 1004015852

Assignment 2

5. *gradCE()*: First, take the partial derivative of the total cross-entropy loss with respect to the ith element of the softmax input. The following is for a single example in the dataset:

$$\begin{aligned}\mathcal{L}_{CE} &= -\sum_{k=1}^{K} y_k log(p_k) \\ \frac{\partial \mathcal{L}_{CE}}{\partial z_i} &= -\sum_{k=1}^{K} y_k \frac{1}{p_k}\frac{\partial p_k}{\partial z_i}\end{aligned} \tag{1}$$

Next, take the partial derivative of the kth element of the softmax output with respect to the ith element of the softmax input. If $i = k$:

$$\begin{aligned}p(\boldsymbol{z})_k &= \frac{e^{z_k}}{\sum_{j=1}^{J} e^{z_j}} \\ \frac{\partial p_k}{\partial z_i} &= \frac{e^{z_k}\sum_{j=1}^{J} e^{z_j} - e^{z_k}e^{z_k}}{(\sum_{j=1}^{J} e^{z_j})^2} \\ &= \frac{e^{z_k}(\sum_{j=1}^{J} e^{z_j} - e^{z_k})}{(\sum_{j=1}^{J} e^{z_j})^2} \\ &= \frac{e^{z_k}}{\sum_{j=1}^{J} e^{z_j}}\frac{\sum_{j=1}^{J} e^{z_j} - e^{z_k}}{\sum_{j=1}^{J} e^{z_j}} \\ &= p_k(1-p_k) \\ &= p_k(1-p_i)\end{aligned}$$

If $i \neq k$:

$$\begin{aligned}p(\boldsymbol{z})_k &= \frac{e^{z_k}}{\sum_{j=1}^{J} e^{z_j}} \\ \frac{\partial p_k}{\partial z_i} &= e^{z_k}\frac{-e^{z_i}}{(\sum_{j=1}^{J} e^{z_j})^2} \\ &= -\frac{e^{z_k}}{\sum_{j=1}^{J} e^{z_j}}\frac{e^{z_i}}{\sum_{j=1}^{J} e^{z_j}} \\ &= -p_k p_i\end{aligned}$$

In general, $\frac{\partial p_k}{\partial z_i}$ can be expressed as

$$\frac{\partial p_k}{\partial z_i} = p_k(\delta_{i,k} - p_i) \tag{2}$$

where

$$\delta_{i,k} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{if } i \neq k \end{cases}$$

Using (2) in (1) results in

$$\begin{aligned}\frac{\partial \mathcal{L}_{CE}}{\partial z_i} &= -\sum_{k=1}^{K} y_k \frac{1}{p_k} p_k(\delta_{i,k} - p_i) \\ &= -\sum_{k=1}^{K} (y_k \delta_{i,k} - y_k p_i) \\ &= -y_i + \sum_{k=1}^{K} y_k p_i\end{aligned}$$

Since $\underline{y}$ is one-hot encoded, $\sum_{k=1}^{K} y_k = 1$. Therefore,

$$\frac{\partial \mathcal{L}_{CE}}{\partial z_i} = p_i - y_i \tag{3}$$

Or, in matrix form:

$$\frac{\partial \mathcal{L}_{CE}}{\partial \boldsymbol{z}} = \boldsymbol{p} - \boldsymbol{y}$$

where $\frac{\partial \mathcal{L}_{CE}}{\partial \boldsymbol{z}} \in \mathbb{R}^{10\times N}$, $\boldsymbol{p} \in \mathbb{R}^{10\times N}$, and $\boldsymbol{y} \in \mathbb{R}^{10\times N}$. Note that the Python implementation below uses the transposes of $\boldsymbol{p}$, $\boldsymbol{y}$, and outputs the transpose of $\frac{\partial \mathcal{L}_{CE}}{\partial \boldsymbol{z}}$:

```python
def gradCE(target, prediction):
  return prediction - target
```

1.2 Backpropogation Derivation

1. $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_o}$, the gradient of the loss with respect to the outer layer weights. Shape: $(K \times 10)$, with K units.

   $\underline{z} \in \mathbb{R}^{10\times 1}$, the input to the softmax function $\underline{p}(\underline{z}) \in \mathbb{R}^{10\times 1}$, is of the form

   $$\underline{z} = \boldsymbol{W}_o^T \underline{x}^{(1)} + \underline{b}_o$$

   where $\underline{x}^{(1)} \in \mathbb{R}^{K\times 1}$ is the output of the hidden layer and $\boldsymbol{W}_o \in \mathbb{R}^{K\times 10}$ and $\underline{b}_o \in \mathbb{R}^{10\times 1}$ are the output layer weight matrix and biases, respectively. The element-wise partial derivative of $\underline{z}$ with respect to the weight matrix $\boldsymbol{W}_o$ is

   $$\frac{\partial z_i}{\partial \left(W_o\right)_{j,i}} = x_j^{(1)} \tag{4}$$

   Using (3) and (4), the element-wise partial derivative of the loss $\mathcal{L}$ with respect to the outer layer weight matrix $\boldsymbol{W}_o$ is

   $$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \left(W_o\right)_{j,i}} &= \frac{\partial \mathcal{L}}{\partial z_i} \frac{\partial z_i}{\partial \left(W_o\right)_{j,i}} \\ &= \left(p_i - y_i\right) x_j^{(1)}\end{aligned}$$

Therefore,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_o} = \left(\underline{p} - \underline{y}\right) \underline{x}^{(1)^T} \tag{5}$$

Full gradient descent requires the summation of (5) across all N samples:

$$\sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_o} = \sum_{n=1}^{N} \left(\underline{p}_{(n)} - \underline{y}_{(n)}\right) \underline{x}_{(n)}^{(1)^T} \tag{6}$$

Vectorized in matrix form, (6) becomes

$$\sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_o} = \left(\boldsymbol{p} - \boldsymbol{y}\right) \boldsymbol{x}^{(1)^T} \tag{7}$$

where $\boldsymbol{p} \in \mathbb{R}^{10\times N}$, $\boldsymbol{y} \in \mathbb{R}^{10\times N}$ and $\boldsymbol{x}^{(1)} \in \mathbb{R}^{K\times N}$. The transpose of (7) has dimensions $(K \times 10)$, as required. Note that the Python implementation below uses the transposes of $\boldsymbol{p}$, $\boldsymbol{y}$, and $\boldsymbol{x}^{(1)}$, and outputs the transpose of (7):

```
def gradLossOuterWeights(target, prediction, X1):
  X1Trans = np.transpose(X1)
  gradLossSoftmaxInput = gradCE(target, prediction)
  return np.matmul(X1Trans, gradLossSoftmaxInput)
```

2. $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_o}$, the gradient of the loss with respect to the outer layer biases. Shape: (1×10).

The element-wise partial derivative of $\underline{z}$ with respect to the weight matrix $\boldsymbol{b}_o$ is

$$\frac{\partial z_i}{\partial \left(b_o\right)_i} = 1 \tag{8}$$

Using (3) and (8), the element-wise partial derivative of the loss $\mathcal{L}$ with respect to the outer layer biases $\boldsymbol{b}_o$ is

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \left(b_o\right)_i} &= \frac{\partial \mathcal{L}}{\partial z_i} \frac{\partial z_i}{\partial \left(b_o\right)_i} \\ &= \left(p_i - y_i\right) \end{aligned}$$

Therefore,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_o} = \left(\underline{p} - \underline{y}\right) \tag{9}$$

Full gradient descent requires the summation of (9) across all N samples:

$$\sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_o} = \sum_{n=1}^{N} \left(\underline{p}_{(n)} - \underline{y}_{(n)}\right) \tag{10}$$

Vectorized in matrix form, (10) becomes

$$\sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_o} = (\boldsymbol{p} - \boldsymbol{y})\,\mathbf{1} \tag{11}$$

where $\boldsymbol{p} \in \mathbb{R}^{10\times N}$, $\boldsymbol{y} \in \mathbb{R}^{10\times N}$, and $\mathbf{1} \in \mathbb{R}^{N\times 1}$ is a vector of ones. The transpose of (11) has dimensions (1×10), as required. Note that the Python implementation below uses the transposes of $\boldsymbol{p}$, $\boldsymbol{y}$, and $\mathbf{1}$, and outputs the transpose of (11):

```python
def gradLossOuterBias(target, prediction):
  onesRow = np.ones((1, target.shape[0]))
  gradLossSoftmaxInput = gradCE(target, prediction)
  return np.matmul(onesRow, gradLossSoftmaxInput)
```

3. $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_h}$, the gradient of the loss with respect to the hidden layer weights. Shape: $(F \times K)$, with F features, K units.

$\underline{s}^{(1)} \in \mathbb{R}^{K\times 1}$, the input to the ReLU function $\underline{x}^{(1)}(\underline{s}^{(1)}) \in \mathbb{R}^{K\times 1}$, is of the form

$$\underline{s}^{(1)} = \boldsymbol{W}_h^T \underline{x}^{(0)} + \underline{b}_h$$

where $\underline{x}^{(0)} \in \mathbb{R}^{F\times 1}$ is the input layer and $\boldsymbol{W}_h \in \mathbb{R}^{F\times K}$ and $\underline{b}_h \in \mathbb{R}^{K\times 1}$ are the hidden layer weight matrix and biases, respectively.

Adapting Equation (7.4) from [1], the following expression for $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_h}$ is obtained:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_h} = \underline{x}^{(0)} \left(\frac{\partial \mathcal{L}}{\partial \underline{s}^{(1)}} \right)^T \tag{12}$$

Adapting Equation (7.5) from [1], the following expression for $\frac{\partial \mathcal{L}}{\partial \underline{s}^{(1)}}$ is obtained:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \underline{s}^{(1)}} &= \frac{\partial \underline{x}^{(1)}}{\partial \underline{s}^{(1)}} \otimes \left(\boldsymbol{W}_o \frac{\partial \mathcal{L}}{\partial \underline{z}} \right) \\ &= \frac{\partial \underline{x}^{(1)}}{\partial \underline{s}^{(1)}} \otimes \left(\boldsymbol{W}_o \left(\underline{p} - \underline{y} \right) \right) \end{aligned} \tag{13}$$

The ReLU function $\underline{x}^{(1)}(\underline{s}^{(1)})$ is defined element-wise as follows:

$$x_i^{(1)}(s_i^{(1)}) = \max\left(s_i^{(1)}, 0 \right)$$

The element-wise partial derivative of $\underline{x}^{(1)}$ with respect to $\underline{s}^{(1)}$ is

$$\left(\frac{\partial x_i^{(1)}}{\partial s_i^{(1)}} \right)_i = \begin{cases} 1 & \text{if } s_i^{(1)} \geq 0 \\ 0 & \text{if } s_i^{(1)} < 0 \end{cases} \tag{14}$$

Therefore, $\frac{\partial \underline{x}_i^{(1)}}{\partial \underline{s}_i^{(1)}}$ is a $(K \times 1)$ vector that satisfies the property in (14).

Using (14) in (13), and then using (13) in (12), the following is obtained:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_h} = \underline{x}^{(0)} \left(\frac{\partial \underline{x}^{(1)}}{\partial \underline{s}^{(1)}} \otimes \left(\boldsymbol{W}_o \left(\underline{p} - \underline{y} \right) \right) \right)^T \tag{15}$$

Full gradient descent requires the summation of (15) across all N samples:

$$\sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_h} = \sum_{n=1}^{N} \underline{x}^{(0)} \left(\frac{\partial \underline{x}_{(n)}^{(1)}}{\partial \underline{s}_{(n)}^{(1)}} \otimes \left(\boldsymbol{W}_o \left(\underline{p}_{(n)} - \underline{y}_{(n)} \right) \right) \right)^T \tag{16}$$

Vectorized in matrix form, (16) becomes

$$\sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_h} = \boldsymbol{x}^{(0)} \left(\frac{\partial \boldsymbol{x}_{(n)}^{(1)}}{\partial \boldsymbol{s}_{(n)}^{(1)}} \otimes \left(\boldsymbol{W}_o \left(\boldsymbol{p}_{(n)} - \boldsymbol{y}_{(n)} \right) \right) \right)^T \tag{17}$$

where $\boldsymbol{x}^{(0)} \in \mathbb{R}^{F \times N}$, $\frac{\partial \boldsymbol{x}_{(n)}^{(1)}}{\partial \boldsymbol{s}_{(n)}^{(1)}} \in \mathbb{R}^{K \times N}$, $\boldsymbol{W}_o \in \mathbb{R}^{K \times 10}$, $\boldsymbol{p} \in \mathbb{R}^{10 \times N}$, and $\boldsymbol{y} \in \mathbb{R}^{10 \times N}$. (17) has dimensions $(F \times K)$, as required. Note that the Python implementation below uses the transposes of $\boldsymbol{x}^{(0)}$, $\frac{\partial \boldsymbol{x}_{(n)}^{(1)}}{\partial \boldsymbol{s}_{(n)}^{(1)}}$, $\boldsymbol{p}$, and $\boldsymbol{y}$:

```python
def gradLossHiddenWeights(X0, S1, W_o, target, prediction):
  S1[S1 > 0] = 1
  S1[S1 < 0] = 0
  X0Trans = np.transpose(X0)
  gradLossSoftmaxInput = gradCE(target, prediction)
  W_o_trans = np.transpose(W_o)
  return np.matmul(X0Trans, S1 * np.matmul(gradLossSoftmaxInput,
    W_o_trans))
```

4. $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_h}$, the gradient of the loss with respect to the hidden layer biases. Shape: $(1 \times K)$, with K units.

The element-wise partial derivative of $\underline{s}^{(1)}$ with respect to the hidden layer biases $\boldsymbol{b}_h$ is

$$\frac{\partial s_i^{(1)}}{\partial \left(b_h \right)_i} = 1 \tag{18}$$

Similar to (15), the following is obtained:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_h} = \left(\frac{\partial \underline{x}^{(1)}}{\partial \underline{s}^{(1)}} \otimes \left(\boldsymbol{W}_o \left(\underline{p} - \underline{y} \right) \right) \right)^T \tag{19}$$

Full gradient descent requires the summation of (19) across all N samples:

$$\sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_h} = \left(\frac{\partial \underline{x}^{(1)}}{\partial \underline{s}^{(1)}} \otimes \left(\boldsymbol{W}_o \left(\underline{p} - \underline{y} \right) \right) \right)^T \tag{20}$$

Vectorized in matrix form, (20) becomes

$$\sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_h} = \mathbf{1} \left(\frac{\partial \boldsymbol{x}_{(n)}^{(1)}}{\partial \boldsymbol{s}_{(n)}^{(1)}} \otimes \left(\boldsymbol{W}_o \left(\boldsymbol{p}_{(n)} - \boldsymbol{y}_{(n)} \right) \right) \right)^T \tag{21}$$

where $\frac{\partial \boldsymbol{x}_{(n)}^{(1)}}{\partial \boldsymbol{s}_{(n)}^{(1)}} \in \mathbb{R}^{K \times N}$, $\boldsymbol{W}_o \in \mathbb{R}^{K \times 10}$, $\boldsymbol{p} \in \mathbb{R}^{10 \times N}$, $\boldsymbol{y} \in \mathbb{R}^{10 \times N}$, and $\mathbf{1} \in \mathbb{R}^{1 \times N}$ is a vector of ones. (21) has dimensions $(K \times 1)$, as required. Note that the Python implementation below uses the transposes of $\frac{\partial \boldsymbol{x}_{(n)}^{(1)}}{\partial \boldsymbol{s}_{(n)}^{(1)}}$, $\boldsymbol{p}$, and $\boldsymbol{y}$:

```python
def gradLossHiddenBias(S1, W_o, target, prediction):
  S1[S1 > 0] = 1
  S1[S1 < 0] = 0
  onesRow = np.ones((1, target.shape[0]))
  gradLossSoftmaxInput = gradCE(target, prediction)
  W_o_trans = np.transpose(W_o)
  return np.matmul(onesRow, S1 * np.matmul(gradLossSoftmaxInput,
   W_o_trans))
```

1.3 Learning

Table 1: Final Accuracies with 1000 Hidden Units

Training Accuracy	Validation Accuracy	Test Accuracy
0.9342	0.9027	0.89868

In this section, a neural network with 1000 hidden units was trained for 200 epochs. The weight matrices were initialized following the Xavier initialization scheme (each element is randomly selected from a Gaussian distribution with $\mu = 0$ and variance $\sigma^2 = \frac{2}{\text{units in+units out}}$). The bias vectors were initialized with zeros. The following update rule was used:

$$\boldsymbol{\nu}_{\textbf{new}} \leftarrow \gamma \boldsymbol{\nu}_{\textbf{old}} + \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \boldsymbol{\nu}_{\textbf{new}}$$

When the learning was set to $\alpha = 10^{-5}$ as stipulated in the assignment, the model displayed divergent behaviour and failed to train correctly. When the learning rate was set

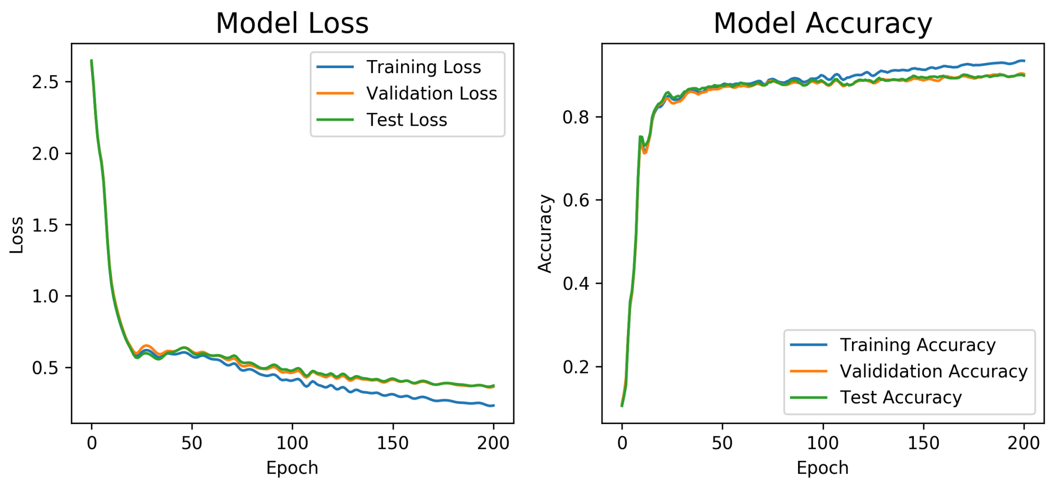


Figure 1: NumPy Neural Network with 1000 Hidden Units

to $\alpha = 10^{-6}$, the model trained correctly, so a learning rate of $\alpha = 10^{-6}$ was used instead. γ was set to 0.99 and each element of the $\boldsymbol{\nu}$ matrices was initialized to 10^{-5}. Training this model took 430.598 seconds. The final accuracies are shown in Table 1. The NumPy implementation of the neural network is shown below:

```
def learning(W_o, W_h, b_o, b_h, v_W_o, v_W_h, v_b_o, v_b_h, epochs, gamma
    , alpha, trainData, trainTarget, validData, validTarget, testData,
    testTarget):
  trainAccuracy = []
  validAccuracy = []
  testAccuracy = []
  trainLoss = []
  validLoss = []
  testLoss = []

  for i in range(epochs):
    S1 = computeLayer(trainData, W_h, b_h)
    X1 = ReLU(S1)
    Z = computeLayer(X1, W_o, b_o)
    P = softmax(Z)
    trainLoss.append(averageCE(trainTarget, P))
    predictions = np.argmax(P, axis = 1)
    labels = np.argmax(trainTarget, axis = 1)
    numberCorrect = np.sum(np.equal(predictions, labels) == True)
    trainAccuracy.append(numberCorrect / trainData.shape[0])

    S1Valid = computeLayer(validData, W_h, b_h)
    X1Valid = ReLU(S1Valid)
    ZValid = computeLayer(X1Valid, W_o, b_o)
    PValid = softmax(ZValid)
    validLoss.append(averageCE(validTarget, PValid))
```

```python
    predictionsValid = np.argmax(PValid, axis = 1)
    labelsValid = np.argmax(validTarget, axis = 1)
    numberCorrectValid = np.sum(np.equal(predictionsValid, labelsValid) == True)
    validAccuracy.append(numberCorrectValid / validData.shape[0])

    S1Test = computeLayer(testData, W_h, b_h)
    X1Test = ReLU(S1Test)
    ZTest = computeLayer(X1Test, W_o, b_o)
    PTest = softmax(ZTest)
    testLoss.append(averageCE(testTarget, PTest))
    predictionsTest = np.argmax(PTest, axis = 1)
    labelsTest = np.argmax(testTarget, axis = 1)
    numberCorrectTest = np.sum(np.equal(predictionsTest, labelsTest) == True)
    testAccuracy.append(numberCorrectTest / testData.shape[0])

    print("Epoch:", i)
    v_W_o = gamma * v_W_o + alpha * gradLossOuterWeights(trainTarget, P, X1)
    W_o = W_o - v_W_o
    v_b_o = gamma * v_b_o + alpha * gradLossOuterBias(trainTarget, P)
    b_o = b_o - v_b_o

    v_W_h = gamma * v_W_h + alpha * gradLossHiddenWeights(trainData, S1, W_o, trainTarget, P)
    W_h = W_h - v_W_h
    v_b_h = gamma * v_b_h + alpha * gradLossHiddenBias(S1, W_o, trainTarget, P)
    b_h = b_h - v_b_h

  return W_o, W_h, b_o, b_h, trainAccuracy, validAccuracy, testAccuracy, trainLoss, validLoss, testLoss

trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
trainData = trainData.reshape((trainData.shape[0], trainData.shape[1] * trainData.shape[2]))
validData = validData.reshape((-1, validData.shape[1] * validData.shape[2]))
testData = testData.reshape((-1, testData.shape[1] * testData.shape[2]))

############################### PARAMETERS ###############################

epochs = 50
alpha = 0.00001
gamma = 0.99
hiddenUnits = 1000
output_image_filename = 'Numpy_1000_Hidden_Units.png'

```

```
# For initializing weight matrices
mean = 0
variance_o = 2 / (hiddenUnits + 10)
variance_h = 2 / (trainData.shape[1] + hiddenUnits)

#########################################################################

trainOneHot, validOneHot, testOneHot = convertOneHot(trainTarget,
    validTarget, testTarget)

W_o = np.random.normal(mean, np.sqrt(variance_o), (hiddenUnits, 10))
W_h = np.random.normal(mu, np.sqrt(stddev_h), (trainData.shape[1],
    hidden_units))

b_o = np.zeros((1, 10))
b_h = np.zeros((1, hiddenUnits))

v_W_o = np.full((hiddenUnits, 10), 1e-5)
v_W_h = np.full((trainData.shape[1],hidden_units), 1e-5)

v_b_o = np.full((1, 10), 1e-5)
v_b_h = np.full((1, hiddenUnits), 1e-5)

W_o, W_h, b_o, b_h, trainAccuracy, validAccuracy, testAccuracy, trainLoss,
     validLoss, testLoss = learning(W_o, W_h, b_o, b_h, v_W_o, v_W_h, v_b_o
    , v_b_h, epochs, gamma, alpha, trainData, trainOneHot,validData,
    validOneHot, testData, testOneHot)

S1 = computeLayer(trainData, W_h, b_h)
X1 = ReLU(S1)
Z = computeLayer(X1, W_o, b_o)
P = softmax(Z)
trainLoss.append(averageCE(trainOneHot, P))
predictions = np.argmax(P, axis = 1)
labels = np.argmax(trainOneHot, axis = 1)
numberCorrect = np.sum(np.equal(predictions, labels) == True)
trainAccuracy.append(numberCorrect / trainData.shape[0])
print("Final training accuracy:", trainAccuracy[-1])

S1Valid = computeLayer(validData, W_h, b_h)
X1Valid = ReLU(S1Valid)
ZValid = computeLayer(X1Valid, W_o, b_o)
PValid = softmax(ZValid)
validLoss.append(averageCE(validOneHot, PValid))
predictionsValid = np.argmax(PValid, axis = 1)
labelsValid = np.argmax(validOneHot, axis = 1)
numberCorrectValid = np.sum(np.equal(predictionsValid, labelsValid) ==
    True)
validAccuracy.append(numberCorrectValid / validData.shape[0])
print("Final validation accuracy:", validAccuracy[-1])
```

```

S1Test = computeLayer(testData, W_h, b_h)
X1Test = ReLU(S1Test)
ZTest = computeLayer(X1Test, W_o, b_o)
PTest = softmax(ZTest)
testLoss.append(averageCE(testOneHot, PTest))
predictionsTest = np.argmax(PTest, axis = 1)
labelsTest = np.argmax(testOneHot, axis = 1)
numberCorrectTest = np.sum(np.equal(predictionsTest, labelsTest) == True)
testAccuracy.append(numberCorrectTest / testData.shape[0])
print("Final testing accuracy:", testAccuracy[-1])

epochs = len(trainLoss)

fig = plt.gcf()
fig.set_size_inches(10, 4)

plt.subplot(1, 2, 1)
plt.plot(range(epochs), trainLoss)
plt.plot(range(epochs), validLoss)
plt.plot(range(epochs), testLoss)
plt.legend(['Training Loss', 'Validation Loss', 'Test Loss'], loc='upper
    right')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.title('Loss Curve', fontsize=16)

plt.subplot(1, 2, 2)
plt.plot(range(epochs), trainAccuracy)
plt.plot(range(epochs), validAccuracy)
plt.plot(range(epochs), testAccuracy)
plt.legend(['Training Accuracy', 'Valididation Accuracy', 'Test Accuracy'
    ], loc='lower right')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.title('Accuracy Curve', fontsize=16)

plt.suptitle('1000 Hidden Units', fontsize=16)

plt.savefig(output_image_filename, dpi = 300)
```

1.4 Hyperparameter Investigation

1. **Number of Hidden Units**:

Table 2: Final Accuracies with 100, 500, and 2000 Hidden Units

	Training Accuracy	Validation Accuracy	Test Accuracy	Training Time
100 Hidden Units	0.9068	0.8882	0.8917	61.786s
500 Hidden Units	0.9248	0.896	0.8946	229.178s
2000 Hidden Units	0.9333	0.8995	0.8957	824.714s

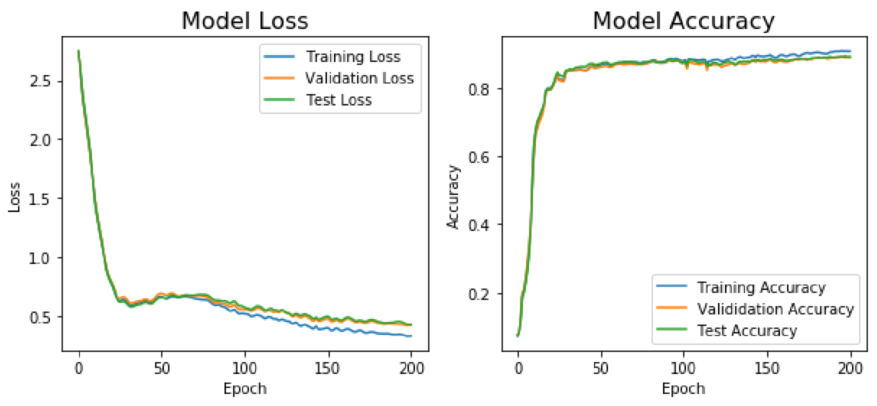
As seen in Table 2, increasing the number of hidden units generally results in higher training, validation, and test accuracies. However, training the model with 2000 hidden units took nearly four times as long as training with 500 hidden units, with negligible increases in accuracy. Additionally, as the number of hidden units was increased from 500 to 2000, the final training accuracy increased but the validation and test accuracies remained the same, which is a sign of overfitting. In subfigures 2b and 2c, the training loss and accuracy curves continue to drop and climb, respectively, whereas the validation and test curves plateaued. This means that the model is no longer learning useful features and is potentially overfitting. Based on these results, 500 is a good choice for the number of hidden units in this model.

2. **Early Stopping**:

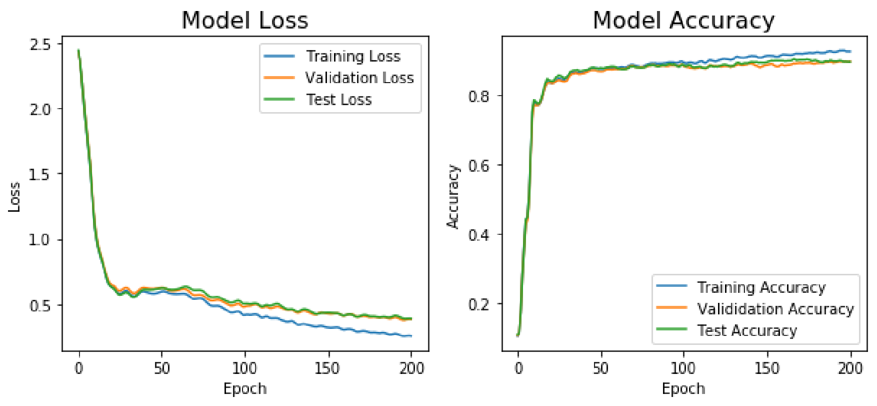
As seen from the loss and accuracy curves in Figure 1 of Section 1.3, overfitting does not occur within 200 epochs (i.e., the validation and test loss curves do not climb and the validation and test accuracy curves do not drop). Over the 200 epochs, the training, validation, and testing losses and accuracies all decrease and increase, respectively, as the number of epochs increases. However, it is noted that the amount of validation and test accuracy increase between epochs 100-200 is very small relative to the increase in accuracies in the initial 100 epochs. Consequently, it may be wise to perform early stoppage prior to epoch 200, to save computation time in exchange for a small decrease in accuracy.

Table 3: Final Accuracies at 200 and 100 Epochs (Early Stoppage)

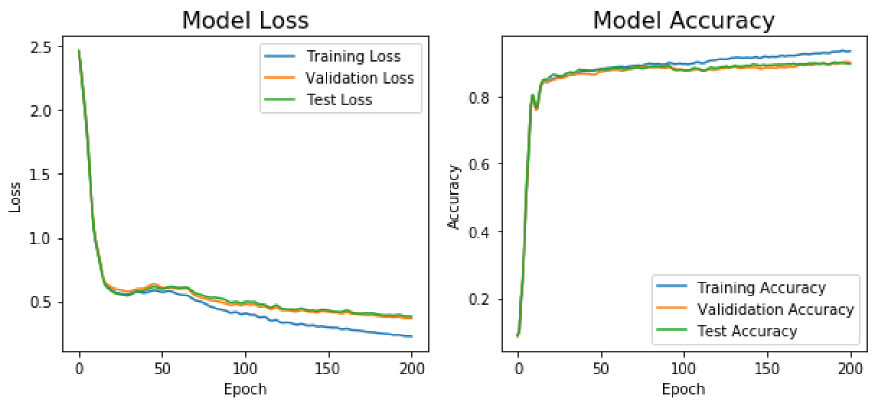
	Training Accuracy	Validation Accuracy	Test Accuracy	Training Time
100 Epochs	0.8986	0.8838	0.8847	222.132
200 Epochs	0.9342	0.9027	0.8987	430.598



(a) Loss and Accuracy Curves for NumPy Neural Network with 100 Hidden Units



(b) Loss and Accuracy Curves for NumPy Neural Network with 500 Hidden Units



(c) Loss and Accuracy Curves for NumPy Neural Network with 2000 Hidden Units

Figure 2: Loss and Accuracy Curves for NumPy Neural Network with 100, 500, and 2000 Hidden Units

2 Neural Networks in Tensorflow

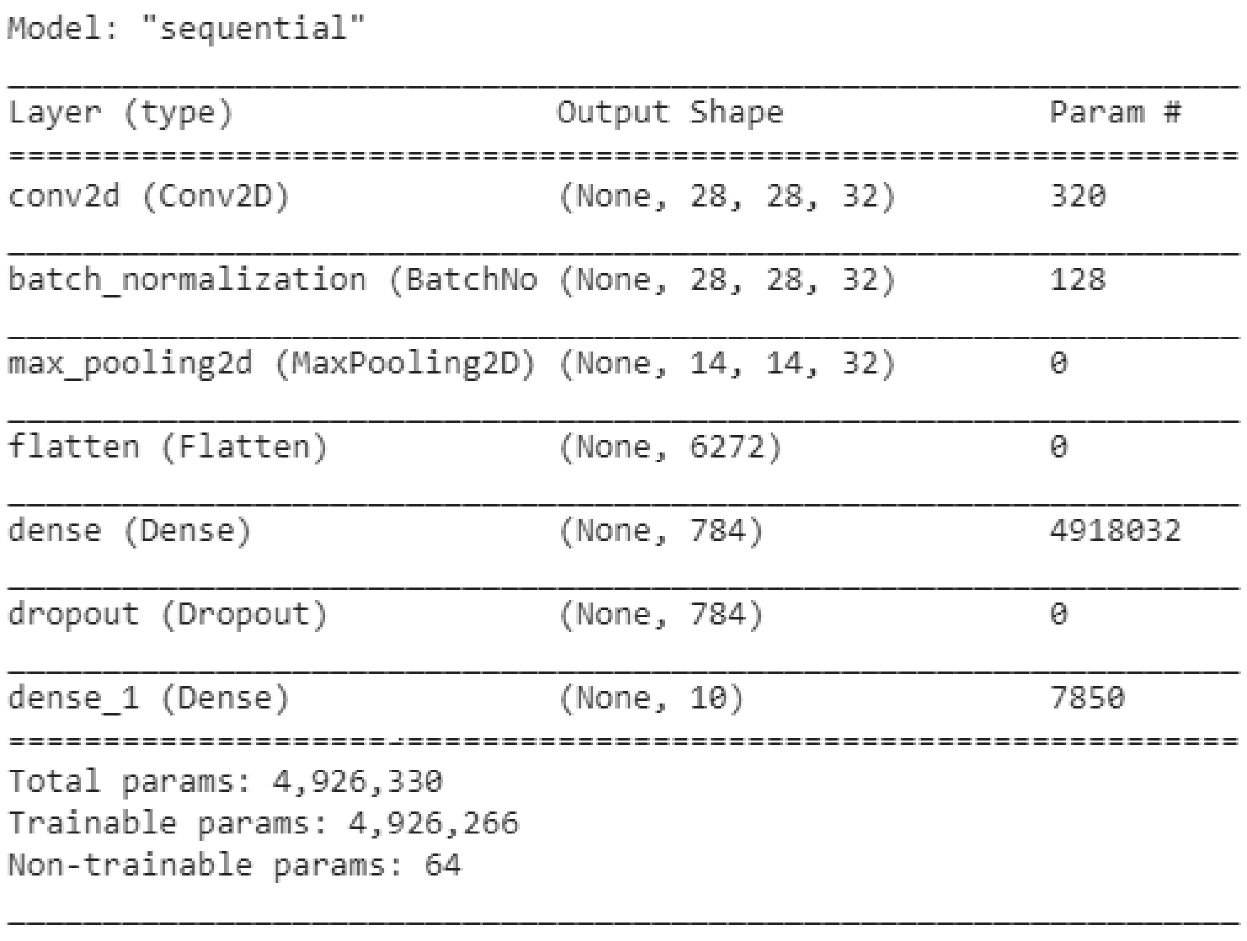
2.1 Model Implementation

According to discussion on the course's Piazza, in this part we have the flexibility to implement the convolutional neural network (CNN) using any state-of-the-art framework. Since our model architecture involves the use of well-known layers, implementation of the CNN would be more convenient using the higher level Tensorflow.Keras API.

In general, constructing a neural net in Keras involves stacking up layers in a model. Here, we construct our model according to the assignment handout.

```
model = Sequential()
# 1       Input Layer
# 2 - 3   3x3 Convolutional Layer
model.add(Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), padding='same', activation='relu', input_shape=(28, 28, 1), kernel_initializer='glorot_normal', bias_initializer='glorot_normal'))

# 4       Batch Normalization
model.add(BatchNormalization())

# 5       2x2 Max Pooling
model.add(MaxPooling2D(pool_size=(2, 2)))

# 6       Flatten Layer
model.add(Flatten())

# 7 - 8   Fully Connected Layer
model.add(Dense(units=784, activation='relu', kernel_initializer='glorot_normal', bias_initializer='glorot_normal', kernel_regularizer=l2(L2_LAMBDA)))

# Dropout Layer
model.add(Dropout(rate=DROPOUT))

# 9 - 10 Fully Connected Layer
model.add(Dense(units=10, activation='softmax', kernel_initializer='glorot_normal', bias_initializer='glorot_normal'))

model.summary()

sgd = SGD(lr=1E-4)
adam = Adam(lr=1E-4)
model.compile(optimizer=adam, loss='categorical_crossentropy', metrics=['accuracy'])
```

Note that some constants (`L2_LAMBDA`, `DROPOUT`) are referenced. These will be used in Section 2.3. Here the values are set to 0.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320
_________________________________________________________________
batch_normalization (BatchNo (None, 28, 28, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
_________________________________________________________________
dense (Dense)                (None, 784)               4918032
_________________________________________________________________
dropout (Dropout)            (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                7850
=================================================================
Total params: 4,926,330
Trainable params: 4,926,266
Non-trainable params: 64
_________________________________________________________________
```

Figure 3: Model Summary

We now have a summary of the model's structure, shown in Figure 3.

2.2 Model Training

Next, we proceed to train the models. First, we want to pre-process our data.

```python
def convertOneHot(trainTarget, validTarget, testTarget):
    newtrain = np.zeros((trainTarget.shape[0], 10))
    newvalid = np.zeros((validTarget.shape[0], 10))
    newtest = np.zeros((testTarget.shape[0], 10))
    for item in range(0, trainTarget.shape[0]):
        newtrain[item][trainTarget[item]] = 1
    for item in range(0, validTarget.shape[0]):
        newvalid[item][validTarget[item]] = 1
    for item in range(0, testTarget.shape[0]):
        newtest[item][testTarget[item]] = 1
    return newtrain, newvalid, newtest
```

```python

trainTarget, validTarget, testTarget = convertOneHot(trainTarget,
    validTarget, testTarget)
trainData = trainData.reshape(-1, 28, 28, 1)
validData = validData.reshape(-1, 28, 28, 1)
testData = testData.reshape(-1, 28, 28, 1)
```

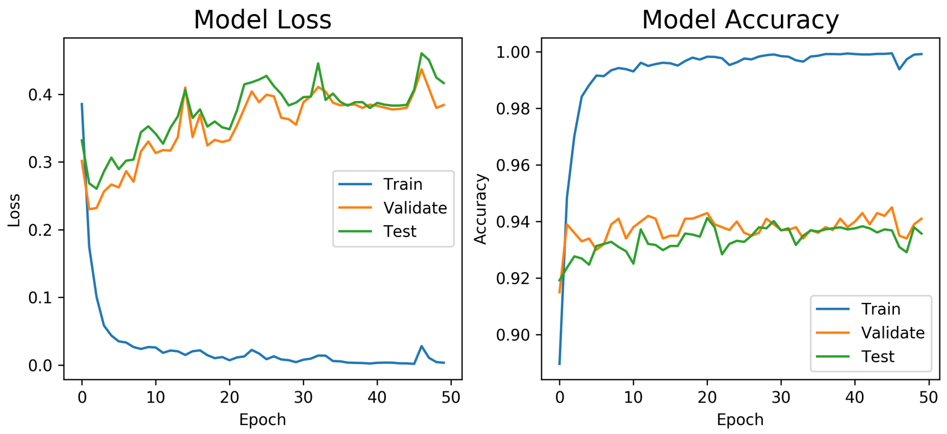
To keep track of test losses and accuracy, we need a custom "callback" function. This will save losses and accuracies for the three data sets at the end of every epoch.

```python
# Define Callback for Validation and Test Accuracy and Loss
class EvalValidationAndTestCB(Callback):
    def __init__(self, testData, testTarget):
        super(EvalValidationAndTestCB, self).__init__()
        self.testData, self.testTarget = testData, testTarget
        self.train_loss, self.train_acc = [], []
        self.val_loss, self.val_acc = [], []
        self.test_loss, self.test_acc = [], []

    def on_train_begin(self, logs):
        self.train_loss, self.train_acc = [], []
        self.val_loss, self.val_acc = [], []
        self.test_loss, self.test_acc = [], []

    def on_epoch_end(self, epoch, logs):
        # Save Training Loss and Acc
        self.train_loss.append(logs['loss'])
        self.train_acc.append(logs['acc'])

        # Save Validation Loss and Acc
        self.val_loss.append(logs['val_loss'])
        self.val_acc.append(logs['val_acc'])

        # Evaluate and Save Test Loss and Acc
        test_scores = self.model.evaluate(x=self.testData, y=self.
    testTarget)
        self.test_loss.append(test_scores[0])
        self.test_acc.append(test_scores[1])

cb = EvalValidationAndTestCB(testData, testTarget)
```

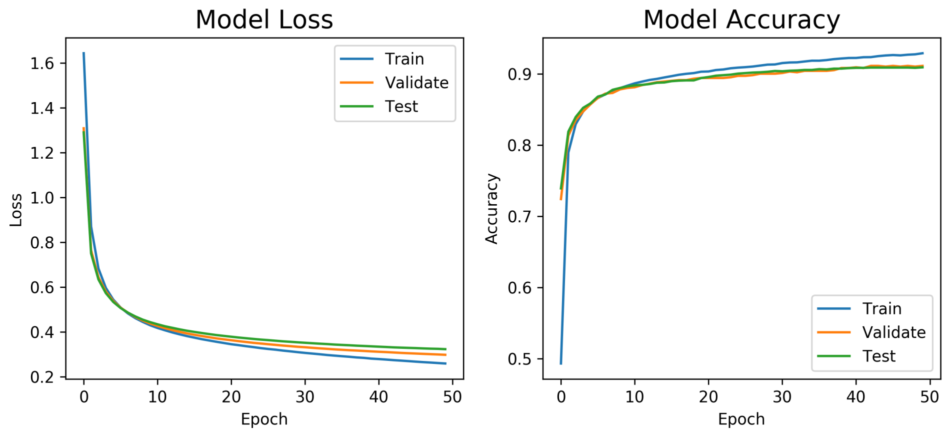
We now have all the ingredients to train our neural network.

```python
history = model.fit(trainData, trainTarget, epochs=50, batch_size=32,
    callbacks=[cb], validation_data=(validData, validTarget))
```

The results (using Adam and SGD, respectively) are shown in Figure 4 in the form of training curves.



(a) CNN Training Curve (Adam Optimizer)



(b) CNN Training Curve (SGD Optimizer)

Figure 4: Training Curves of CNN (Tensorflow) without Dropout, without L2 Regularization

2.3 Hyperparameter Investigation

1. **L2 Regularization**:

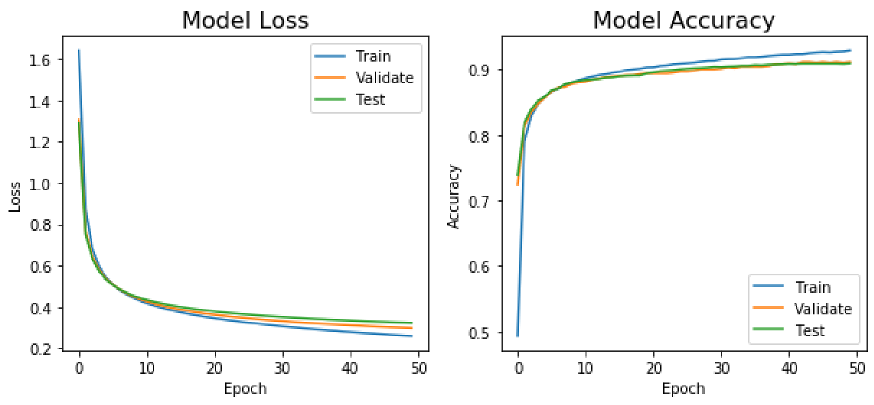
Table 4: Final Accuracies of CNN (Tensorflow) Neural Net with Weight Decay Coefficients of $\lambda = 0.01$, 0.1, and 0.5.

	$\lambda = 0.01$	$\lambda = 0.1$	$\lambda = 0.5$
Training data accuracy	0.9873	0.9549	0.9251
Validation data accuracy	0.9360	0.9260	0.9070
Test data accuracy	0.9302	0.9295	0.9085

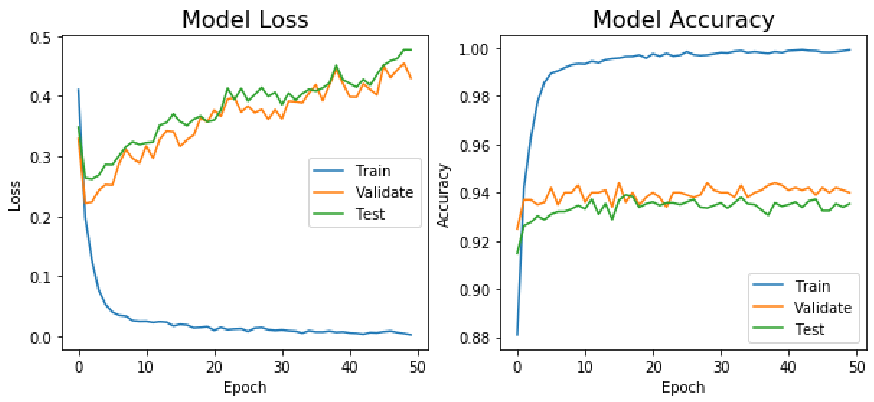
Based on the accuracies in Table 4, it can be seen that by increasing the regularization factor (λ), the final validation and test accuracies decrease. Given that increasing the λ correlates to increasing the loss attributed by the L2 norm of the weight vector, a larger λ prevents values in the weight vector from getting particularly large. This in turn helps mitigate over fitting when given validation and testing datasets. In this scenario, it is likely that the optimal weight vector has larger weight values, resulting in a decrease in accuracy as λ increases.

2. **Dropout**:

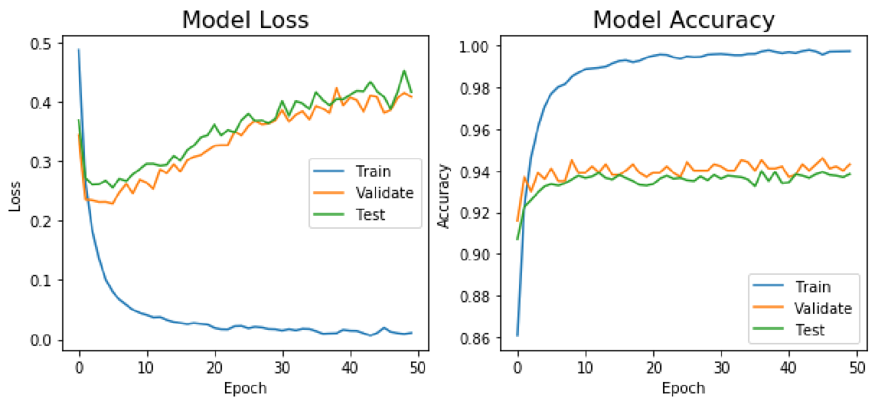
Note: In Figure 5, $p = 0.9$, 0.75, and 0.5 indicate the probabilities that a node is kept.



(a) Training with $p = 0.9$ Keeping Probability



(b) Training with $p = 0.75$ Keeping Probability



(c) Training with $p = 0.5$ Keeping Probability

Figure 5: Training Curves of CNN (Tensorflow) with Keeping Probabilities $p = 0.9$, 0.75, and 0.5, without L2 Regularization

References

[1] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, Learning from data: a short course. Estados Unidos: AMLBook, 2012.