ECE421S – Introduction to Machine Learning

Assignment 2

Neural Networks

Hard Copy Due: March 13, 2019 @ BA3014, 4:00-5:00 PM EST

Code Submission: ece421ta2019@gmail.com March 13, 2019 @ 5:00 PM EST

General Notes:

- Attach this cover page to your hard copy submission
- For assignment related questions, please contact Matthew Wong (matthewck.wong@mail.utoronto.ca)
- For general questions regarding Python or Tensorflow, please contact Tianrui Xiao (tianrui.xiao@mail.utoronto.ca) or see him in person in his office hours, Tuesdays, 4:00-6:00 PM in BA-3128 (Robotics Lab)

Please circle section to which you would like the assignment returned

Tutorial Sections

001	002	003	004
005	006	007	Graduate

Group Members		
Names	StudentID	Contribution %
Andy Linzi Zhou	**1002296730**	**55**
Ryan Do	**1001254117**	**45**

1. Neural Networks using Numpy

1.1 Helper Functions

1.1.5 gradCE() derivation

$$\frac{\partial CE}{\partial S} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\frac{t_k^n}{S_k^n} \qquad for\ K\ classes\ and\ N\ examples$$

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

def GradReLU(x):
    return np.where(x <= 0, 0, 1)

def softmax(x):
    e_x = np.exp(x)
    denom = e_x.sum(axis=1)
    denom = np.tile(denom, (10, 1)).T
    return np.divide(e_x, denom)

def computeLayer(X, W, b):
    return np.matmul(X, W) + np.tile(b, (X.shape[0], 1))

def CE(target, prediction):
    N = target.shape[0]
    return -(1/N) *np.sum(np.multiply(target, np.log(prediction)))

def gradCE(target, prediction):
    N = target.shape[0]
    return (-1 / N) * np.sum(np.division(target, prediction))
```

Figure 1: Code snippet of Helper Functions

1.2 Backpropagation Derivation

Vector Form

Note: $\otimes$ *Outer product between two vectors*

$\odot$ *Inner product between two vectors (element − wise)*

1. K x 10 units, With respect to the i th column of W^o

$$\frac{\partial L}{\partial W_i^o} = \frac{1}{N}\sum_{n=1}^{N}\left(\sigma(z)_i - y_{n,i}\right)\boldsymbol{x}^h$$

We obtain in matrix form

define $X_h = \begin{bmatrix} \boldsymbol{x}_1^h \\ \vdots \\ \boldsymbol{x}_N^h \end{bmatrix}$

$\boldsymbol{\Sigma}(\boldsymbol{z}) = \begin{bmatrix} \sigma(\boldsymbol{z})_1 \\ \vdots \\ \sigma(\boldsymbol{z})_{\boldsymbol{N}} \end{bmatrix}$, $\mathrm{Y} = \begin{bmatrix} y_1 \\ \vdots \\ y_{\boldsymbol{N}} \end{bmatrix}$

$$\frac{\partial L}{\partial W_o} = \frac{1}{N}\sum_{n=1}^{N}(\sigma(\boldsymbol{z}) - y)_n \otimes \boldsymbol{x}_n^h$$

$$\frac{\partial L}{\partial W_o} = \frac{1}{N}\boldsymbol{X}_h^T(\boldsymbol{\Sigma}(\boldsymbol{S_o}) - \boldsymbol{Y})$$

x_n^h : k x 1

y_n: 10 x 1 one hot labeled

$\sigma(\mathbf{z}) = [\sigma(z)_1 \ldots\ldots \sigma(z)_{10}]$

2. 1 x 10 units, With respect to an element j ϵ {1…10}

$$\frac{\partial L}{\partial b_j^o} = \frac{1}{N}\sum_{n=1}^{N}\left(\sigma(z)_j - y_{n,j}\right)$$

We obtain in matrix form

$$\frac{\partial L}{\partial b_o} = \frac{1}{N}\sum_{n=1}^{N}(\sigma(\boldsymbol{z}) - y_n)$$

$$\frac{\partial L}{\partial b_o} = \frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{\Sigma}(\boldsymbol{S_o}) - \boldsymbol{Y})_n$$

3. F x K units, with respect to the i th column of W^h

$$\frac{\partial L}{\partial W_i^h} = \frac{1}{N}\sum_{n=1}^{N}\boldsymbol{x}^{input} \otimes \nabla\left(ReLU\left(s_i^h\right)\right)\sum_{k=1}^{10}\left(\sigma(z)_k - y_{n,k}\right) w_{i,k}^o$$

We obtain in matrix form

$$\frac{\partial L}{\partial W_h} = \frac{1}{N}\sum_{\boldsymbol{n=1}}^{\boldsymbol{N}}\boldsymbol{x}_{input} \otimes (\nabla\left(ReLU\left({W_h}^T\boldsymbol{x}^{input}\right)\right) \odot (W^o(\sigma(\boldsymbol{z}) - y_n)))$$

$$\frac{\partial L}{\partial W_h} = \frac{1}{N}\ \boldsymbol{X_{input}^T}\ (\nabla(ReLU(\boldsymbol{S}_h)) \odot ((\boldsymbol{\Sigma}(\boldsymbol{S_o}) - \boldsymbol{Y})\ \boldsymbol{W_o^T}))$$

$$\nabla(ReLU(\boldsymbol{S}_h)) = \begin{cases}1, & \boldsymbol{S}_h > 0\\ 0, & \boldsymbol{S}_h \le 0\end{cases}$$

W^o : K x 10

W_h: F x K

4. 1 x K units, with respect to an element j ϵ {1…K}

$$\frac{\partial L}{\partial b_j^h} = \frac{1}{N}\sum_{n=1}^{N} \nabla\left(ReLU\left(s_i^h\right)\right) \sum_{k=1}^{10}\left(\sigma(z)_k - y_{n,k}\right) w_{j,k}^o$$

We obtain in matrix form

$$\frac{\partial L}{\partial b_h} = \frac{1}{N}\sum_{n=1}^{N} (\nabla\left(ReLU\left(W_h{}^T \boldsymbol{x}^{input}\right)\right) \odot (W^o(\sigma(\boldsymbol{z}) - y_n)^T))$$

$$\frac{\partial L}{\partial b_h} = \frac{1}{N}\sum_{n=1}^{N} (\nabla\left(ReLU(\boldsymbol{S}_h)\right) \odot ((\boldsymbol{\Sigma}(\boldsymbol{S_o}) - \boldsymbol{Y})\, \boldsymbol{W}_o^T))_n$$

- $\boldsymbol{X_{input}}$ is the output of the input layer
- $\boldsymbol{S_o}$ is the input into the output layer (perform softmax)
- $\boldsymbol{S}_h$ is the input into the hidden layer
- $y_{n,i}$ is value of the ith position in the one-hot label for data point n
- $\mathbf{x}_n^h$ is a vector of outputs from the hidden layer for data point n: $\mathbf{x}_n^h = [x_{1n}, x_{2n}, \ldots x_{Kn}]$
- K is the number of neurons in the hidden layer
- F is the number of input neurons = 784

Shapes of matrices

$$X_h: N\ x\ K$$

$$X_{input}: N\ x\ 784$$

$$W_h: 784\ x\ K$$

$$W_o: K\ x\ 10$$

```python
def backprop_gradients(data, labels, S_h, X_h, X_o, W_o):

    N = labels.shape[0]

    dW_o = (1 / N) * np.matmul(X_h.T, X_o - labels)  # shape: (Kx10)
    db_o = (1 / N) * np.sum(X_o - labels, axis=0)  # shape: (1x10)

    mat1 = np.multiply(GradReLU(S_h), np.matmul(X_o - labels, W_o.T))
    dW_h = (1 / N) * np.matmul(data.T, mat1)

    db_h = (1 / N) * np.sum(mat1, axis=0)

    return dW_o, db_o, dW_h, db_h
```

Figure 2: Backpropagation Function code

1.3 Learning

Note: the x axis represents the epoch

The hyperparameters used to obtain these plots are:

$\alpha = 0.01$

$\gamma = 0.9$

$\text{epochs} = 200$

$\text{hidden size} = 1000$

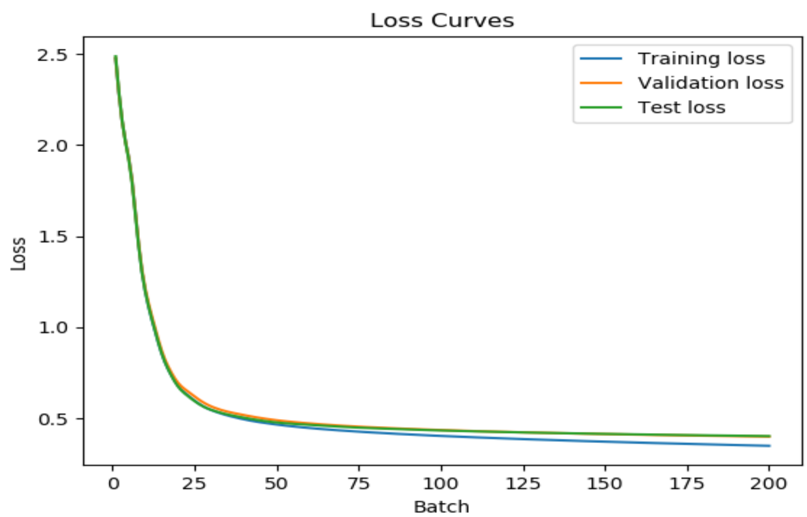


Figure 3: Loss Curves for Section 1.3

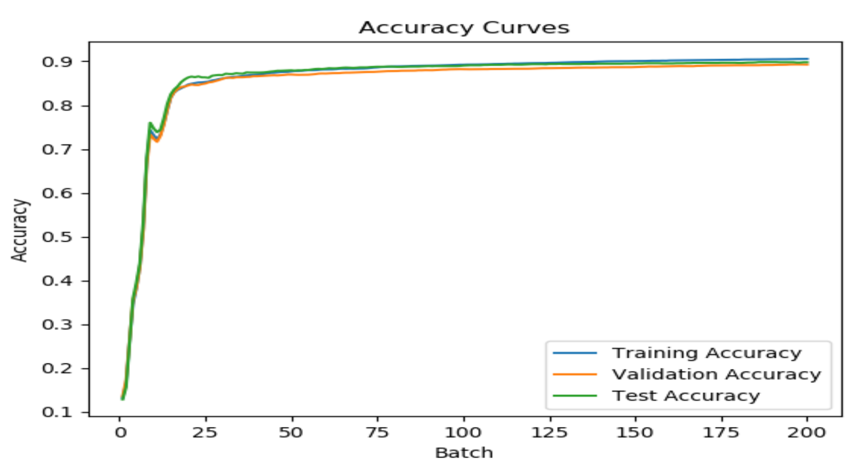
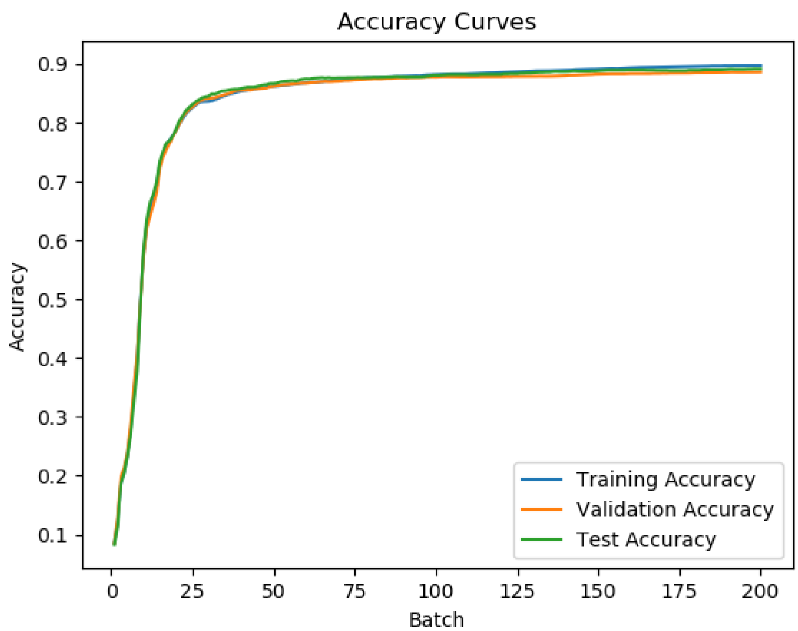
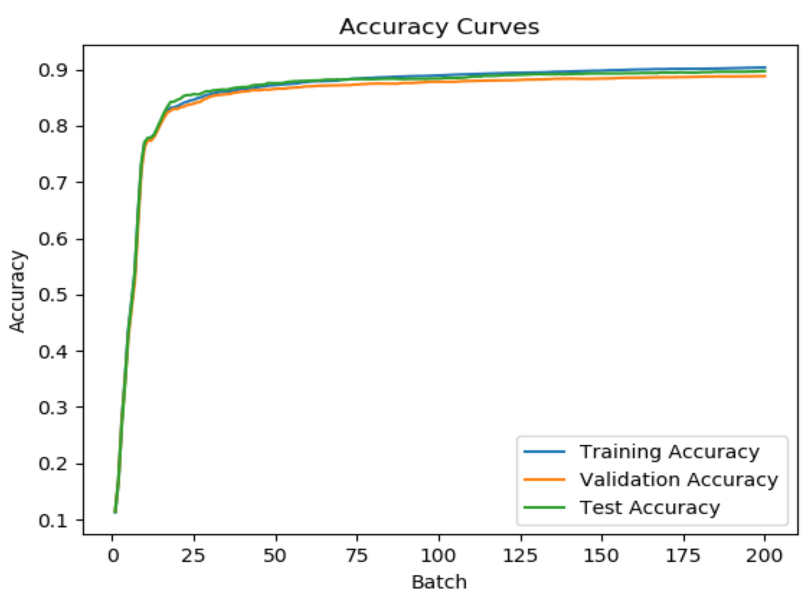
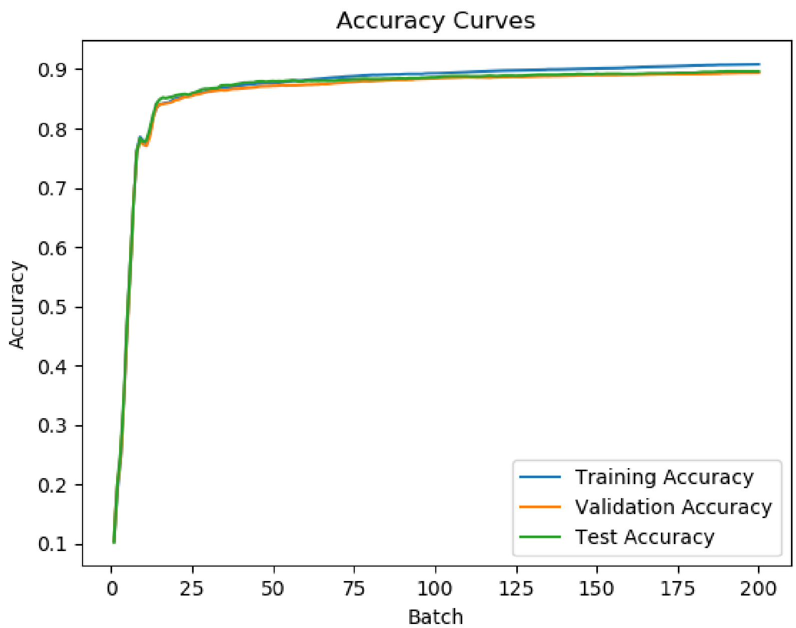


Figure 4: Accuracy Curves for Section 1.3

1.4 Hyperparameter Investigation

1. Number of hidden units

Number of hidden units	Accuracy Curves	Final Test Accuracy
100		0.89170
500		0.89721

2000		0.89648

The number of hidden units had minimal effect on the final test accuracies. However, we can observe that increasing the number of hidden units lead to a faster increase/convergence in test accuracy through the epochs. A hidden layer size of 100 seems to be sufficient for optimizing the model accuracy. Perhaps changing the number of layers would lead to a more sizable impact.

2. Early Stopping

Based on the training/validation/test loss curves in 1.3, there seems to be minimal overfitting. However, the 40 epoch point is where validation and training losses start to diverge. Early stopping should take place at this point.

At an early stop of 40 epochs: The training accuracy is 87.0%. The validation accuracy is 86.6%. The test accuracy is 87.4%.

2. Neural Networks in Tensorflow

2.1 Model Implementation

```
def CNN_model(x, weights, biases):

    #x = tf.reshape(x, shape=[-1, 28, 28, 1])
    # first conv2d layer
    conv1 = tf.nn.conv2d(x, filter=weights['conv2d_filter1'], strides=[1, 1, 1, 1], padding='SAME')
    conv1 = tf.nn.bias_add(conv1, biases['bias1'])

    # first relu layer
    relu1 = tf.nn.relu(conv1)

    # batch normalization layer

    mean, variance = tf.nn.moments(relu1, axes=[0])
    bn_layer = tf.nn.batch_normalization(relu1, mean=mean, variance=variance, offset=None, scale=None, variance_epsilon=1e-8)

    # 2x2 max pooling layer

    maxpool = tf.nn.max_pool(bn_layer, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

    # shape of maxpool = (batch_size, 14, 14, 32)
    # Flatten Layer
    flatten = tf.reshape(maxpool, [-1, weights['fc1_weight'].get_shape().as_list()[0]])

    # fully connected layer 1 (784 output units)
    fc1 = tf.add(tf.matmul(flatten, weights['fc1_weight']), biases['fc1_bias'])

    # dropout layer
    dropout = tf.layers.dropout(fc1, rate=0.5, seed=0)

    # second RELU layer
    relu2 = tf.nn.relu(dropout)

    # fully connected layer 2 (10 output units)
    fc2 = tf.add(tf.matmul(relu2, weights['out_weight']), biases['out_bias'])

    # softmax the output

    out = tf.nn.softmax(fc2)

    return out
```

Figure 5: Model of Neural Network

```
x = tf.placeholder("float", [None, 28, 28, 1])
y = tf.placeholder("float", [None, num_classes])

weights = {
    'conv2d_filter1': tf.get_variable('W1', shape=(3, 3, 1, 32), initializer=tf.contrib.layers.xavier_initializer()),
    'fc1_weight': tf.get_variable('W2', shape=(32*14*14, 784), initializer=tf.contrib.layers.xavier_initializer()),
    'out_weight': tf.get_variable('W6', shape=(784, num_classes), initializer=tf.contrib.layers.xavier_initializer())
}


biases = {
    'bias1': tf.get_variable('B1', shape=(32), initializer=tf.contrib.layers.xavier_initializer()),
    'fc1_bias': tf.get_variable('B2', shape=(784), initializer=tf.contrib.layers.xavier_initializer()),
    'out_bias': tf.get_variable('B3', shape=(num_classes), initializer=tf.contrib.layers.xavier_initializer())
}
```

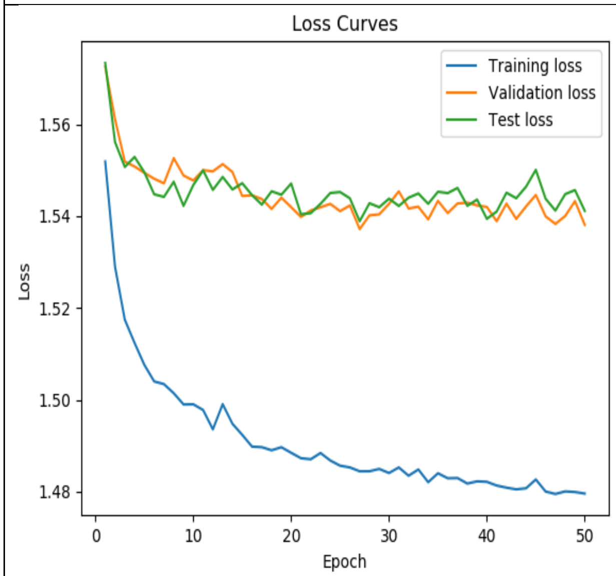
Figure 6: Definition of variables, weights and biases

2.2 Model Training

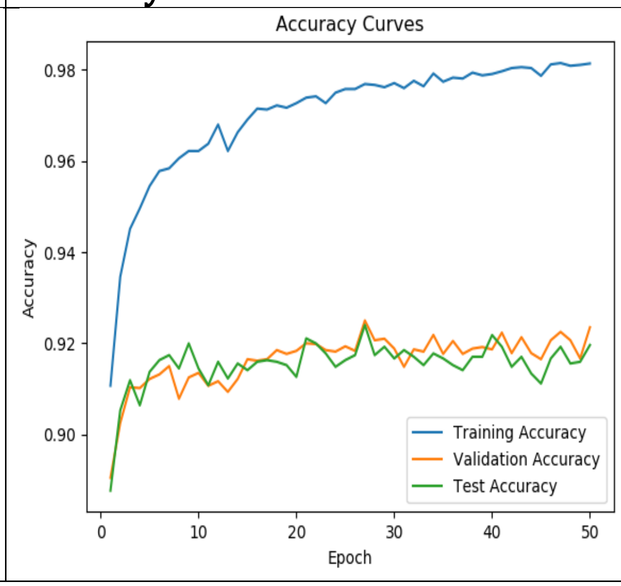
Hyperparameters

- Epochs: 50
- Batch size: 32
- Learning rate: $1x10^{-4}$

Loss curve



Accuracy Curve



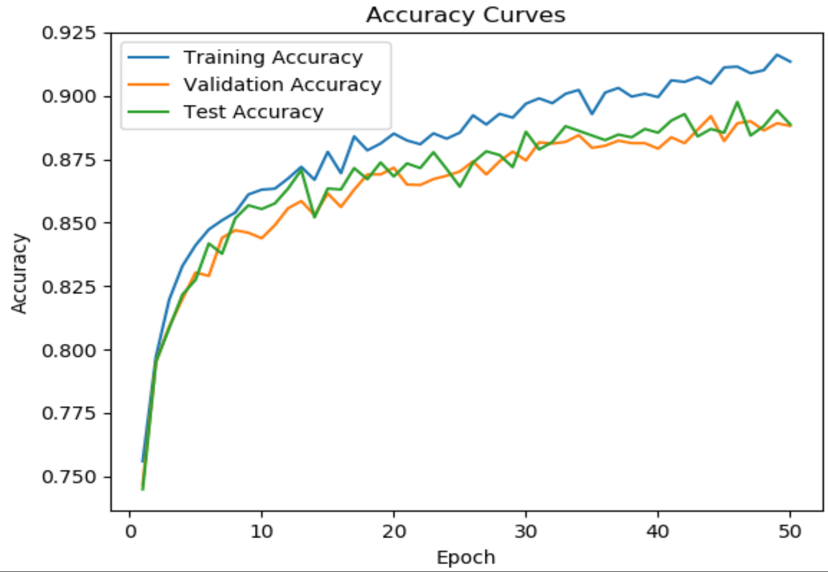
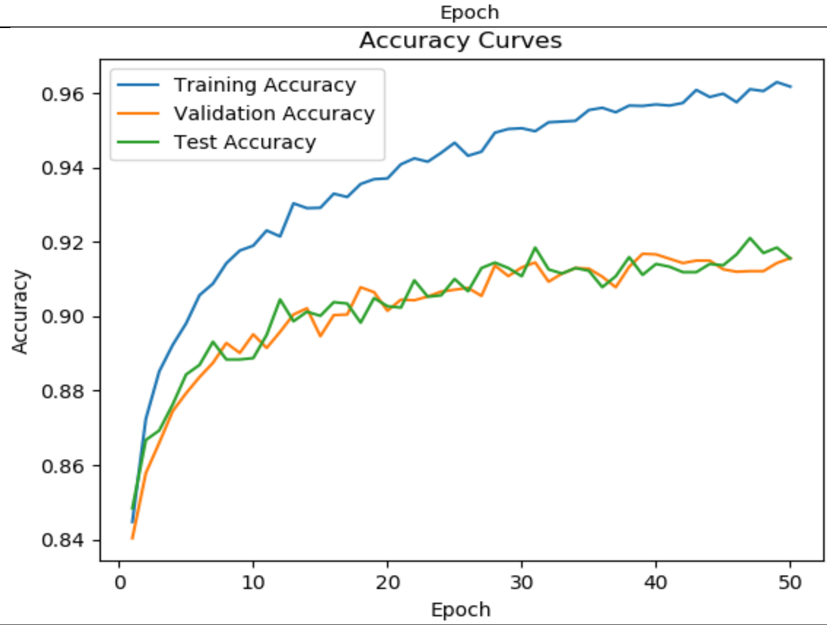
2.3 HyperParameter Investigation

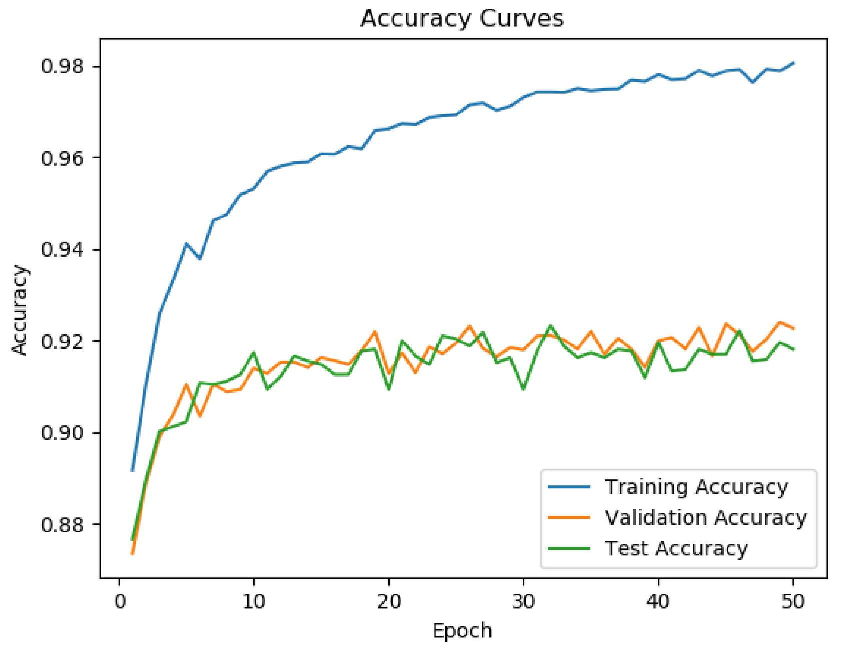
1. L2 Regularization

Weight decay	Final Training Accuracy	Final Validation Accuracy	Final Test Accuracy
$\lambda = 0.01$	0.96230	0.92483	0.92327
$\lambda = 0.1$	0.90390	0.89667	0.89978
$\lambda = 0.5$	0.75970	0.75117	0.75808

L2 Regularization puts more penalty on the norm of the weight matrices as λ increases. As we increase the weight decay coefficient, the final test and validation accuracies decrease. This is to be expected since a greater weight decay coefficient discourages large weights that lead to overfitting, but also performs worse on accuracy.

2. Dropout

Dropout Probability	Accuracy Curves
p = 0.9	
p = 0.75	

p = 0.5	

With higher dropout probability, we are able to control the amount of overfitting in the model.