Assignment 2 - Neural Networks

1.1 — Helper Functions

ReLU	def **relu**(S): X = np.copy(S) X[S<0] = 0 return X	S: a matrix of layer inputs - each row corresponds to a datapoint - each column corresponds to a neuron input at that layer
softmax	def **softmax**(S): X = np.exp(S) / np.sum(np.exp(S), axis=1, keepdims=True) return X	S: * see above
compute	def **computeLayer**(X, W, b): S = X @ W + b return S	X: a matrix of layer outputs W: weight matrix of a given layer b: bias vector of a given layer
averageCE	def **avgCE**(target, prediction): N = prediction.shape[0] L = - (1/N) * np.sum(target * np.log(prediction)) return L	target: a matrix of one-hot encoded labels prediction: a matrix of predicted labels (a probabilistic output)
gradCE	def **gradCE**(target, prediction): N = prediction.shape[0] dE_dX = - (1/N) * target / prediction return dE_dX	target: * see above prediction: * see above

For the gradCE function the average cross entropy loss was used.
Let x_{ij} denote the softmax of the outer layer predictions
Let y_{ij} denote the value of the one-hot encoded output at class j for datapoint i
Then $E_{in} = -\frac{1}{N}\sum_n \sum_k y_{nk} \ln(x_{nk}) \implies \frac{\partial E_{in}}{\partial x_{ij}} = -\frac{1}{N}\frac{\partial}{\partial x_{ij}} y_{ij} \ln(x_{ij}) = -\frac{1}{N}\frac{y_{ij}}{x_{ij}}$

$$\implies \frac{\partial E_{in}}{\partial X} = -\frac{1}{N}\frac{Y}{X}$$

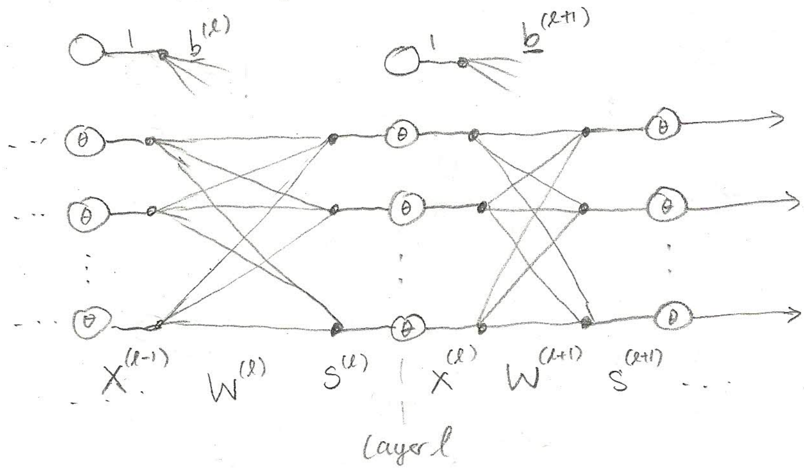
* All these helper functions can be found in Appendix A

1.2 — Backpropagation Derivation

The following pages show how the vectorized forward and backpropagation algorithms are derived for this neural network

VECTORIZING the NEURAL NETWORK

Model of L-layer NN



Let $X^{(l)} = \begin{bmatrix} x_1^{(l)T} \\ \vdots \\ x_N^{(l)T} \end{bmatrix}$ $Y = \begin{bmatrix} y_1^T \\ \vdots \\ y_N^T \end{bmatrix}$ ← one-hot encoded

$\underline{b} = [\, b_1 \cdots b_K \,]$

x_{ij} corresponds to the j^{th} feature of the i^{th} datapoint

$\underline{v}$ ← indicates broadcasting.

FORWARD PROPAGATION VECTORIZATION

$$S_{nj}^{(l)} = \sum_i X_{ni}^{(l-1)} \cdot W_{ij}^{(l)} + b_j^{(l)}$$

$$\boxed{S^{(l)} = X^{(l-1)} W^{(l)} + \underline{b}^{(l)}}$$

$$X_{nj}^{(l)} = \theta(S_{nj}^{(l)})$$

$$\boxed{X^{(l)} = \theta(S^{(l)})}$$

↑ activation function of the l^{th} layer. (ie. ReLu or Softmax)

GRADIENT VECTORIZATION

Let $\delta_{nj}^{(l)} = \dfrac{\partial E_{in}}{\partial S_{nj}^{(l)}}$ the sensitivity of the n^{th} datapoint at the node j layer l.

$$\frac{\partial E_{in}}{\partial w_{ij}^{(l)}} = \sum_{n=1}^{N} \frac{\partial E_{in}}{\partial S_{nj}^{(l)}} \cdot \frac{\partial S_{nj}^{(l)}}{\partial w_{ij}^{(l)}}$$

(This is b/c $E_{in} = \frac{1}{N}\sum_1^N e_n = f(S_1^{(l)}, S_2^{(l)} \cdots S_N^{(l)})$ which means we need to apply a branching chain rule)

$$= \sum_n \delta_{nj}^{(l)} \cdot X_{ni}^{(l-1)} = \sum_n \left(X_{in}^{(l-1)}\right)^T \cdot \delta_{nj}^{(l)} \Rightarrow \boxed{\frac{\partial E_{in}}{\partial W^{(l)}} = [X^{(l-1)}]^T \delta^{(l)}}$$

$$\frac{\partial E_{in}}{\partial b_j^{(l)}} = \sum_1^N \frac{\partial E_{in}}{\partial S_{nj}^{(l)}} \cdot \frac{\partial S_{nj}^{(l)}}{\partial b_j^{(l)}}$$

$$= \sum_n \delta_{nj}^{(l)} \cdot 1 \Rightarrow \boxed{\frac{\partial E_{in}}{\partial \underline{b}^{(l)}} = \underline{1}^T \delta^{(l)}}$$

BACKPROPAGATION VECTORIZATION

$$\delta^{(\ell)} = \frac{\partial E_{in}}{\partial S_{nj}^{(\ell)}} = \frac{\partial E_{in}}{\partial X_{nj}^{(\ell)}} \cdot \frac{\partial X_{nj}^{(\ell)}}{\partial S_{nj}^{(\ell)}}$$

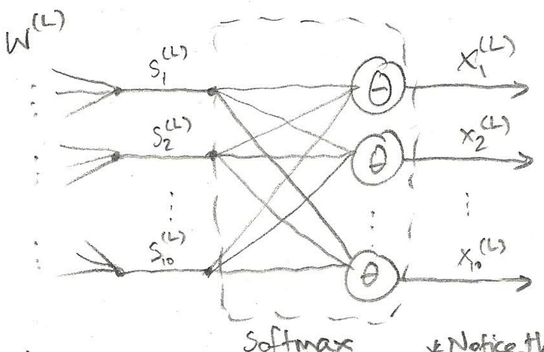
$$= \left(\sum_k \frac{\partial E_{in}}{\partial S_{nk}^{(\ell+1)}} \cdot \frac{\partial S_{nk}^{(\ell+1)}}{\partial X_{nj}^{(\ell)}} \right) \cdot \theta'(S_{nj}^{(\ell)})$$

$$= \left(\sum_k \delta_{nk}^{(\ell+1)} w_{jk}^{(\ell+1)} \right) \cdot \theta'(S_{nj}^{(\ell)}) = \left(\sum_k \delta_{nk}^{(\ell+1)} \left(w_{kj}^{(\ell+1)} \right)^T \right) \cdot \theta'(S_{nj}^{(\ell)})$$

$$\Rightarrow \boxed{\delta^{(\ell)} = \left(\delta^{(\ell+1)} \left[W^{(\ell+1)} \right]^T \right) \otimes \theta'(S^{(\ell)})}$$

in the hidden layer for ReLu.

$\theta'(S^{(\ell)}) = \text{sign}(S^{(\ell)})$



SEEDING the SENSITIVITY

What is $\delta^{(L)}$? L = output layer.

$$\delta_{nj}^{(L)} = \frac{\partial E_{in}}{\partial S_{nj}^{(L)}} = \sum_k \frac{\partial E_{in}}{\partial X_{nk}^{(L)}} \cdot \frac{\partial X_{nk}^{(L)}}{\partial S_{nj}^{(L)}}$$

(Note: unlike for hidden layer activation, the softmax depends on all $S_{nj}^{(L)}$ $\rightarrow \therefore$ we need branching chain rule)

* Notice that the softmax activation depends on all $S_j^{(L)}$

$$E_{in} = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} \ln(X_{nk}^{(L)})$$

$$\frac{\partial E_{in}}{\partial X_{nk}^{(L)}} = -\frac{1}{N} \frac{y_{nk}}{X_{nk}^{(L)}} \Rightarrow \overbrace{\frac{\partial E_{in}}{\partial X^{(L)}} = -\frac{1}{N} \frac{Y}{X^{(L)}}}^{\text{for part 1.1.5}}$$

$$\frac{\partial X_{nk}^{(L)}}{\partial S_{nj}^{(L)}} = \frac{\partial}{\partial S_{nj}^{(L)}} \left(\sigma(S_{nk}^{(L)}) \right) = X_{nk}^{(L)} (\delta_{kj} - X_{nj}^{(L)})$$

dirac delta.

$$\delta_{nj}^{(L)} = \sum_k \frac{\partial E_{in}}{\partial X_{nk}^{(L)}} \cdot \frac{\partial X_{nk}^{(L)}}{\partial S_{nj}^{(L)}}$$

$$= -\frac{1}{N} \sum_k \frac{y_{nk}}{X_{nk}^{(L)}} \cdot X_{nk}^{(L)} (\delta_{kj} - X_{nj}^{(L)})$$

$$= -\frac{1}{N} \sum_k \left(y_{nk} \delta_{kj} - y_{nk} X_{nj}^{(L)} \right)$$

$$= -\frac{1}{N} \left(\sum_k y_{nk} \delta_{kj} - X_{nj}^{(L)} \underbrace{\sum_k y_{nk}}_{\text{is always} = 1 \text{ b/c one-hot encoded.}} \right)$$

$$\Rightarrow \delta^{(L)} = -\frac{1}{N} \left(Y \cdot I - X^{(L)} \right)$$

$$\boxed{\delta^{(L)} = \frac{1}{N} \left(X^{(L)} - Y \right)}$$

For the specific 2-Layer Neural Network we are asked to implement...

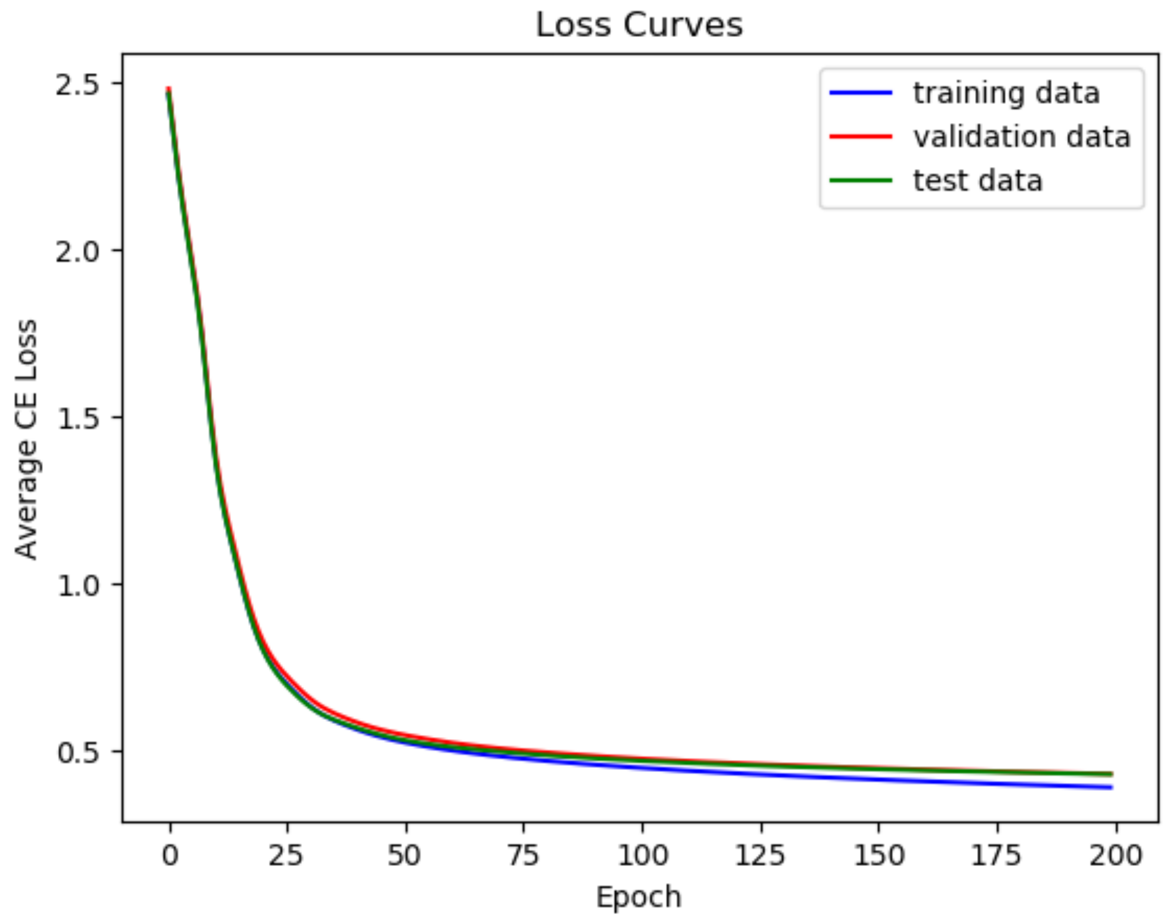
$$\begin{cases} \dfrac{\partial L}{\partial W_o} = \dfrac{\partial E_{in}}{\partial W^{(2)}} = [X^{(1)}]^T \delta^{(2)} \\ \dfrac{\partial L}{\partial b_o} = \dfrac{\partial E_{in}}{\partial \underline{b}^{(2)}} = \underline{1}^T \delta^{(2)} \end{cases}$$

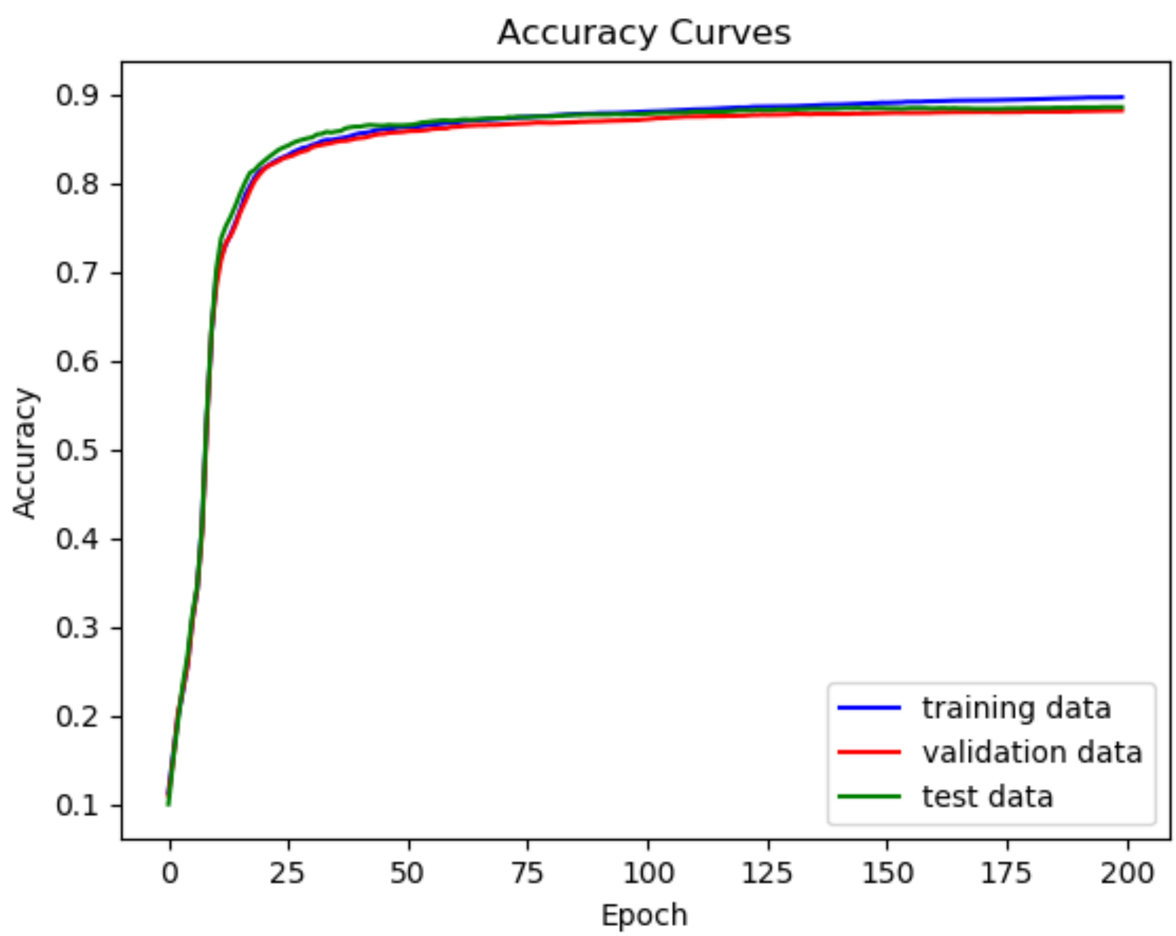
$$\begin{cases} \delta^{(2)} = \dfrac{1}{N}(X^{(2)} - Y) \\ \delta^{(1)} = (\delta^{(2)} [W^{(2)}]^T) \otimes \text{Sign}(S^{(1)}) \end{cases}$$

$$\begin{cases} \dfrac{\partial L}{\partial W_h} = \dfrac{\partial E_{in}}{\partial W^{(1)}} = [X^{(0)}]^T \delta^{(1)} \\ \dfrac{\partial L}{\partial b_h} = \dfrac{\partial E_{in}}{\partial \underline{b}^{(1)}} = \underline{1}^T \delta^{(1)} \end{cases}$$

1.3 — Learning

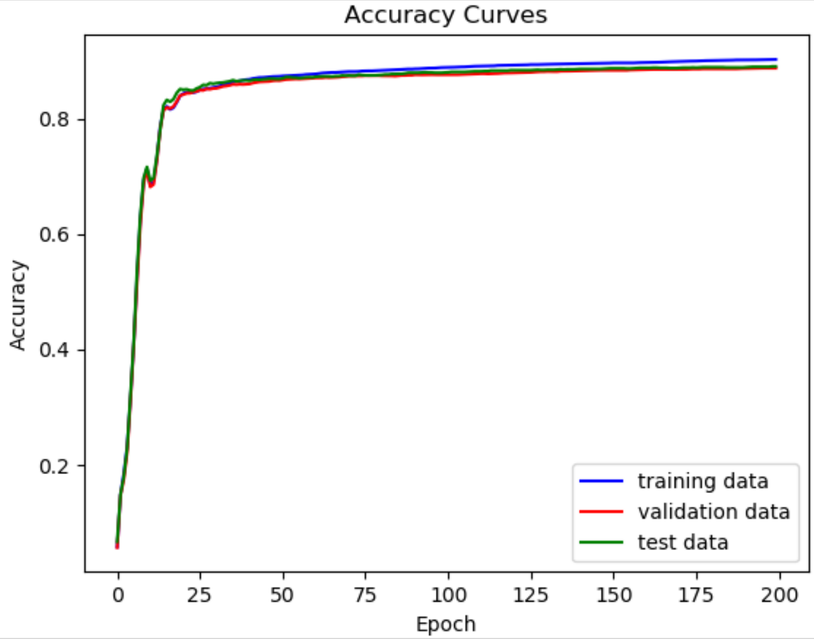
* all code for learning of the neural network can be found in Appendix B — gradient_descent()
* Appendix B — main() executes the NN learning with learning rate = 0.005, epochs = 200, hidden units = 1000





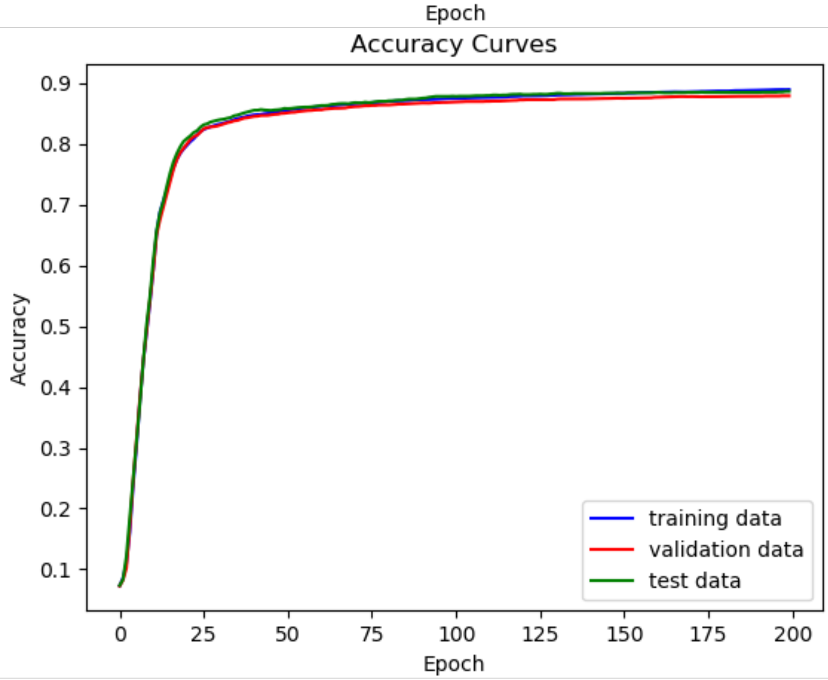
1.4.1 — Number of Hidden Neurons

2000 Neurons



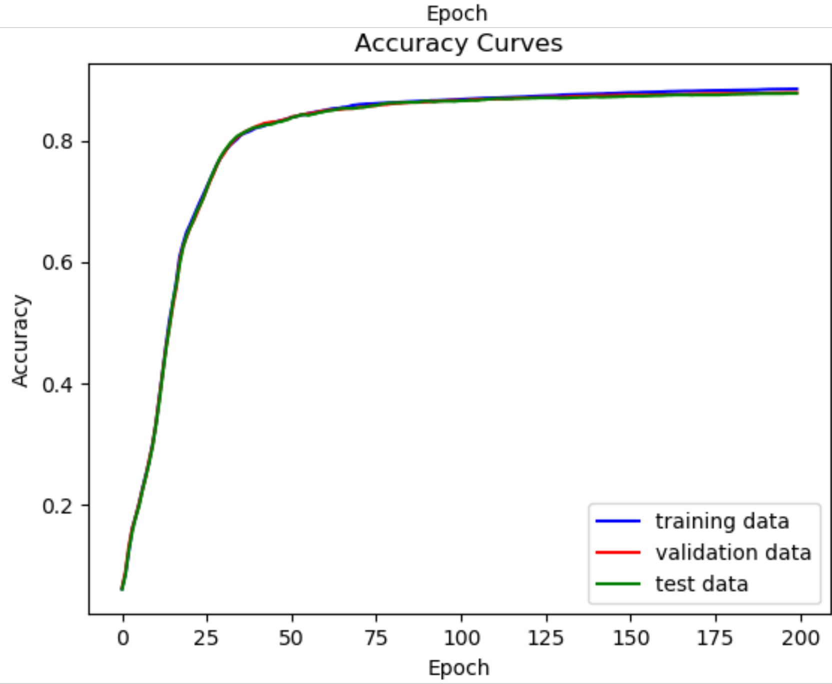

```
TRAINING
----------------------------
Loss:       0.3697683805018834
Accuracy: 0.9023

VALIDATION
----------------------------
Loss:       0.41339738342300447
Accuracy: 0.8871666666666667

TESTING
----------------------------
Loss:       0.4193425232371352
Accuracy: 0.8902349486049926
```

500 Neurons




```
TRAINING
----------------------------
Loss:       0.4139265238179206
Accuracy: 0.8895

VALIDATION
----------------------------
Loss:       0.44251219875802256
Accuracy: 0.879

TESTING
----------------------------
Loss:       0.443531155268261
Accuracy: 0.8861967694566814
```

100 Neurons



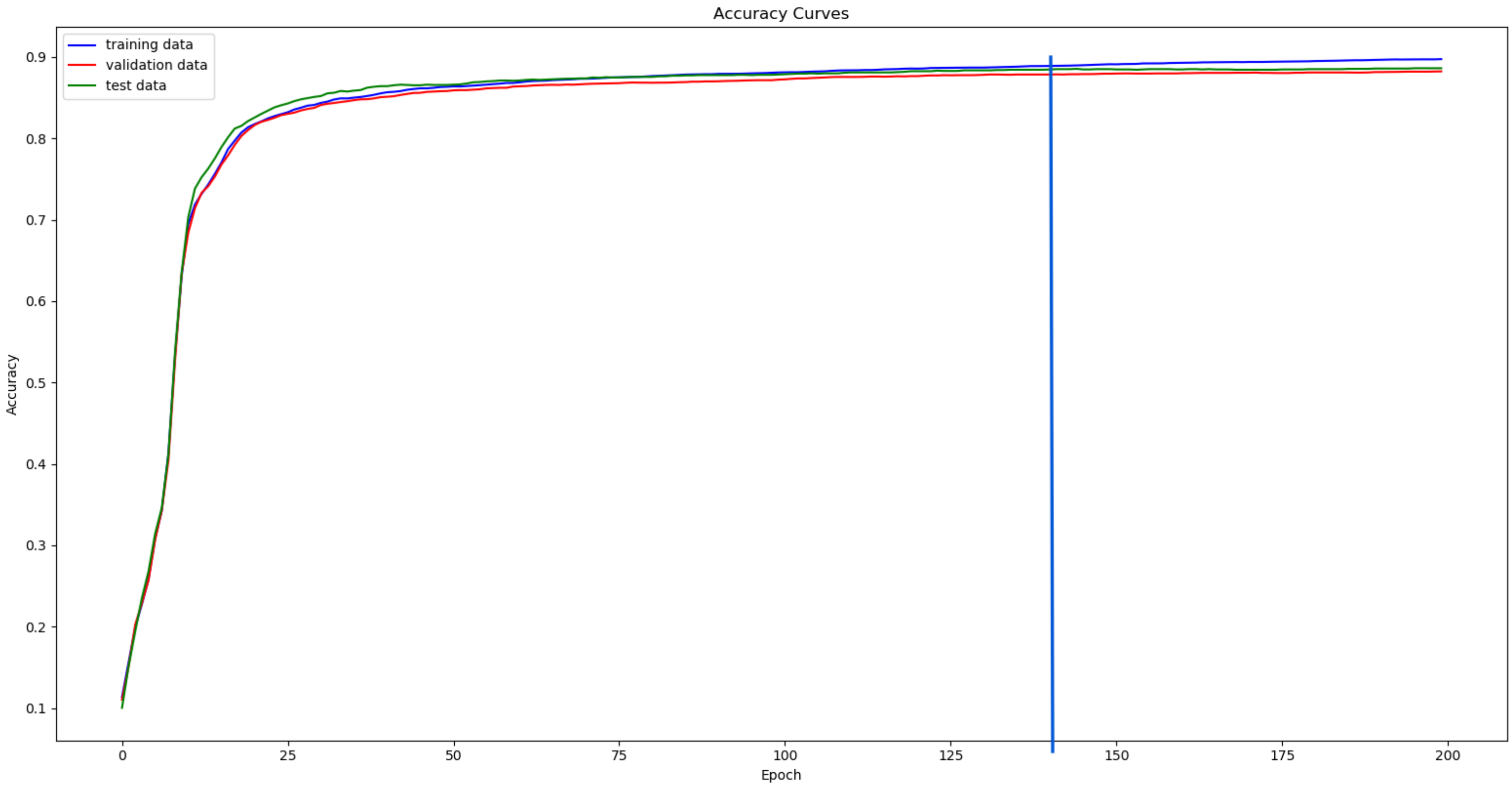

```
TRAINING
----------------------------
Loss:       0.4322297150343103
Accuracy: 0.8851

VALIDATION
----------------------------
Loss:       0.4584138791927889
Accuracy: 0.8791666666666667

TESTING
----------------------------
Loss:       0.46465170276411927
Accuracy: 0.8777533039647577
```

Analysis

By comparing the training accuracies of neural networks with different number of hidden units, we noticed that the more number of hidden units, the lower the total loss and higher the classification accuracy.
This is also true when we compared the test losses and accuracies, although in other cases this may not be true due to overfitting.

1.4.2 — Early Stopping

The early stopping point is the point which has the best test accuracy. In other words, the maximum of the test accuracy curve. Examining the zoomed in accuracy plot from 1.3 we can approximate the early stopping epoch.



The vertical blue line indicates the approximate maximum test accuracy. This occurs at **epoch = 140**

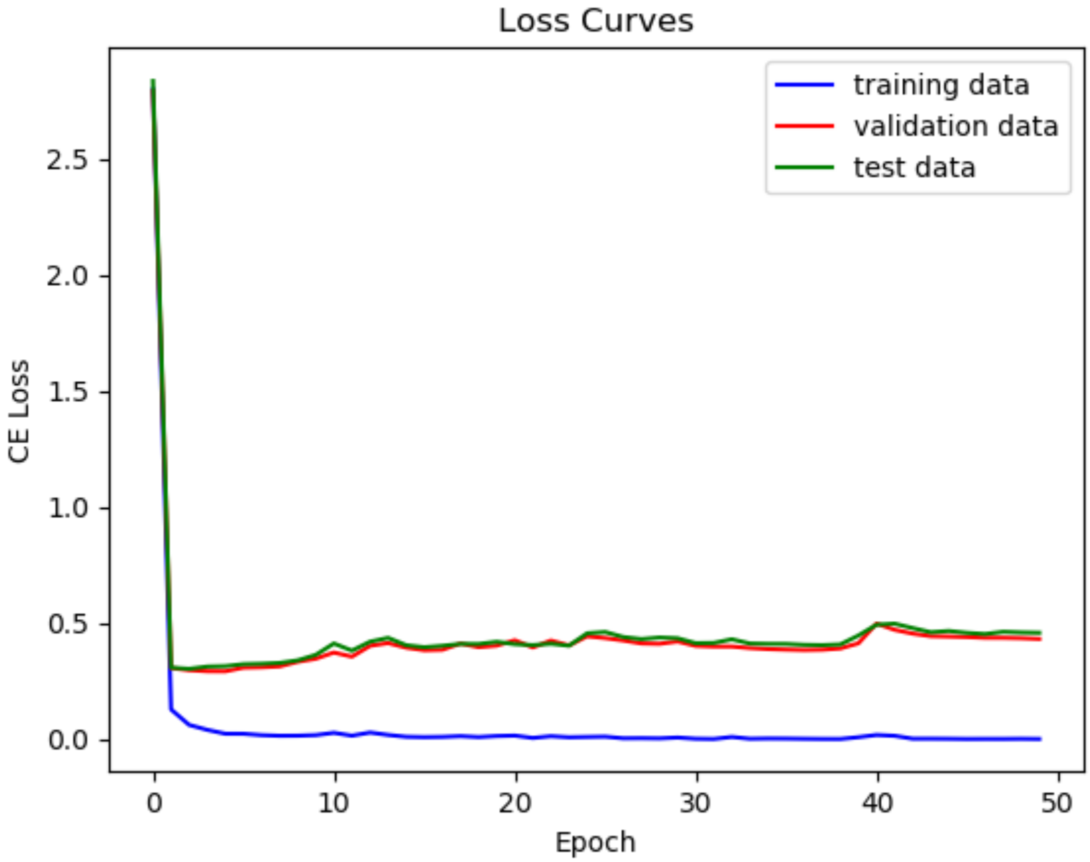
The approximate classification accuracies at epoch = 140 are:
Training Accuracy: 0.883
Validation Accuracy 0.875
Test Accuracy: 0.880

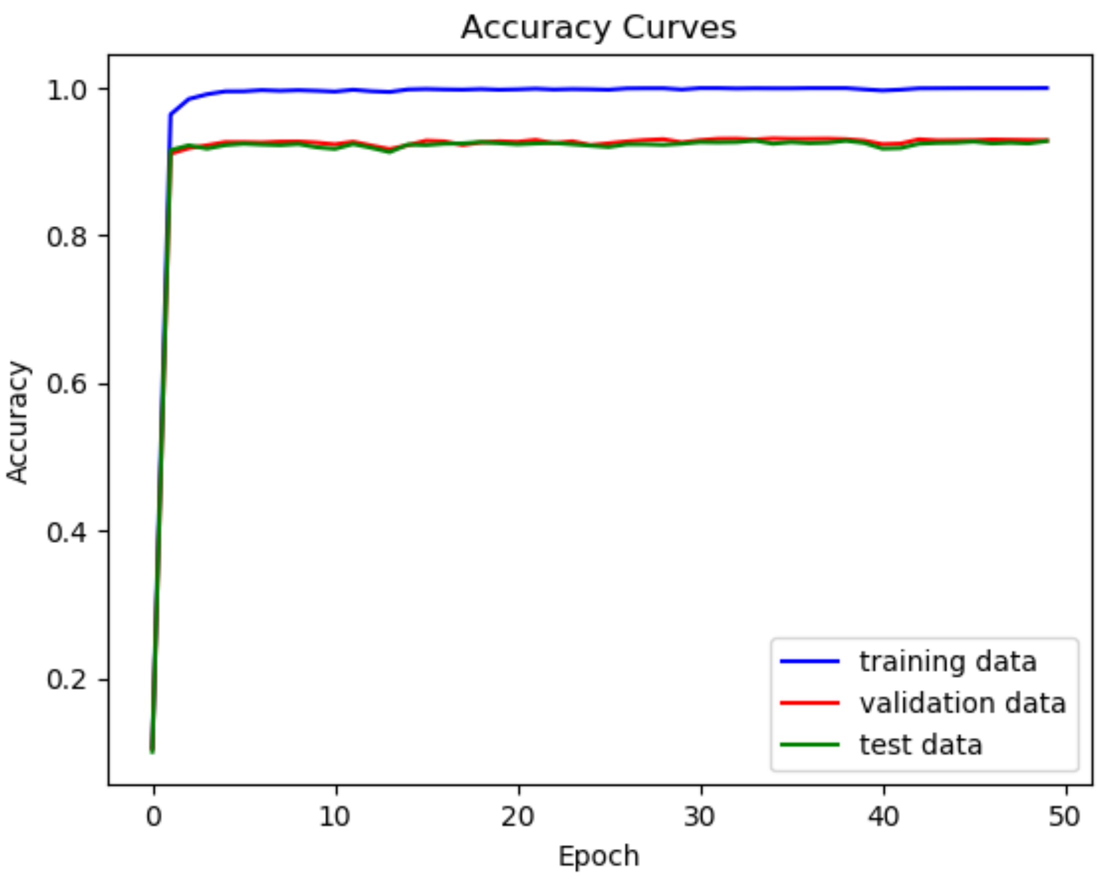
2.1 — Tensorflow CNN Model Implementation

* all code for convolutional neural network model can be found in Appendix C — build_CNN()

2.2 — CNN Training

* all code for learning of the CNN can be found in Appendix C — train_CNN()
* Appendix C — main() executes the NN learning with given parameters (set weight_decay=0 and p=1 for no regularization)





2.3.1 — L2 Regularization

* all code for CNN regularization can be found in Appendix C — main()
* set weight_decay = 0.01 or 0.1 or 0.5 and p=1

	Final Classification Accuracy		
weight decay	**Training Data**	**Validation Data**	**Test Data**
0.01	0.9952	0.93133336	0.9295154
0.1	0.946	0.9206667	0.92217326
0.5	0.8908	0.8843333	0.89133626

The L2 Regularization parameter directly impacts the spread of the classification accuracy between training, validation, and test datasets.

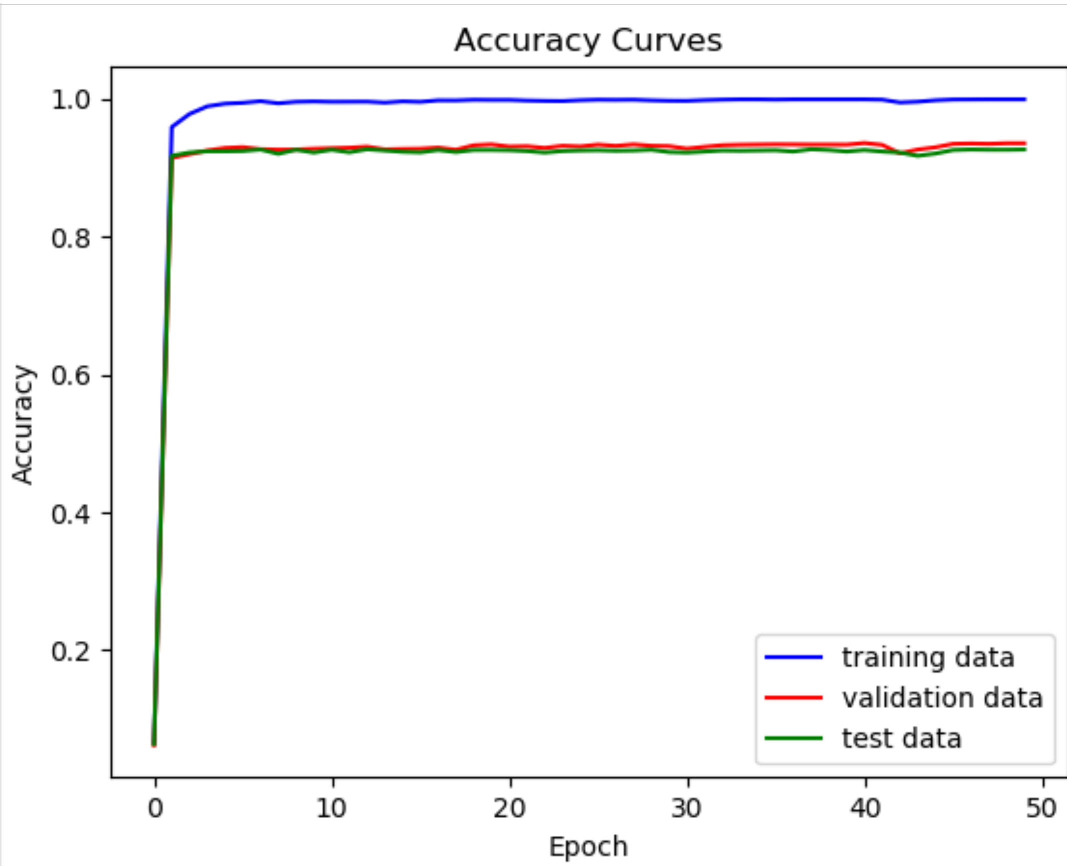
- At low weight decay (0.01) the model is overfitted which causes the training accuracy to be much higher than the validation and test accuracies
- At high weight decay (0.5) the model may be under-fitted, but the difference between the training and test accuracies are much smaller than with a low weight decay

For the given dataset, the test accuracy actually decreases with weight decay greater than 0.1. This suggests that the model may be over-regularized. A more optimal decay value may be found between 0.01 and 0.1.

2.3.2 — Dropout

* all code for CNN dropout can be found in Appendix C — main()
* set weight_decay = 0 and p = 0.9 or 0.75 or 0.5

p=0.9



TRAINING

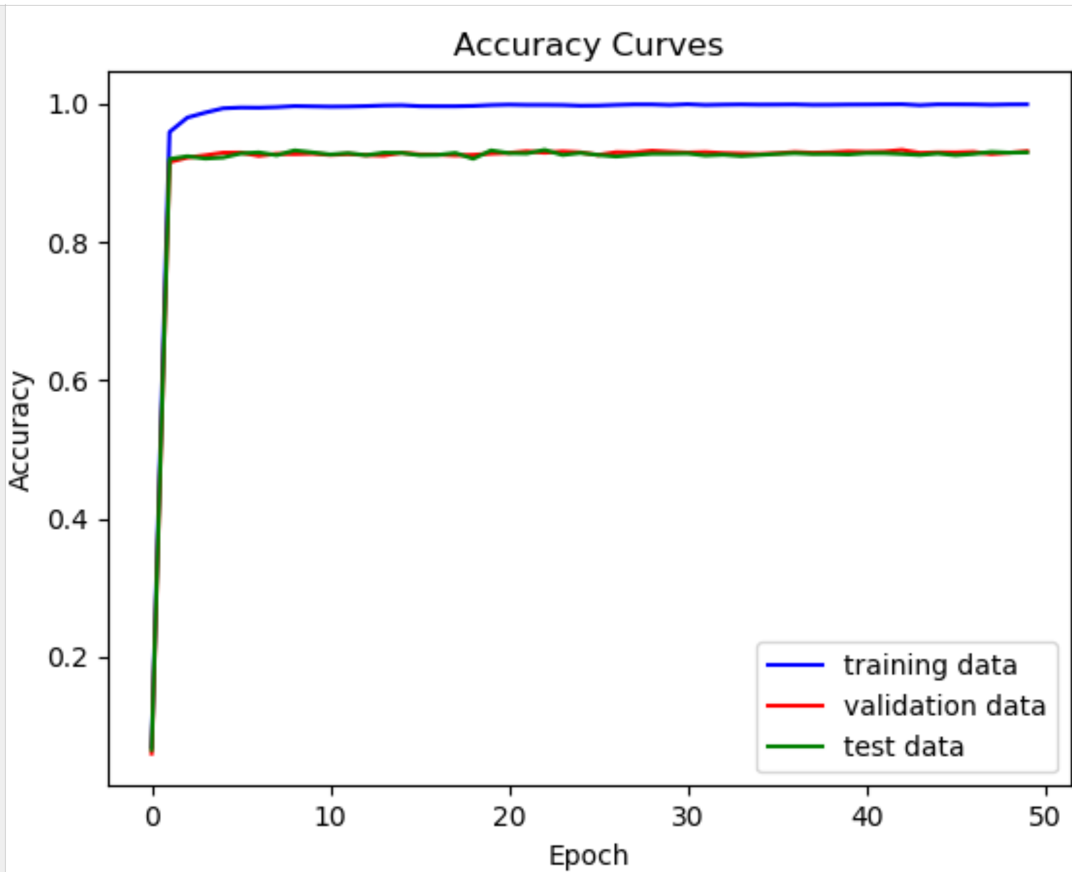
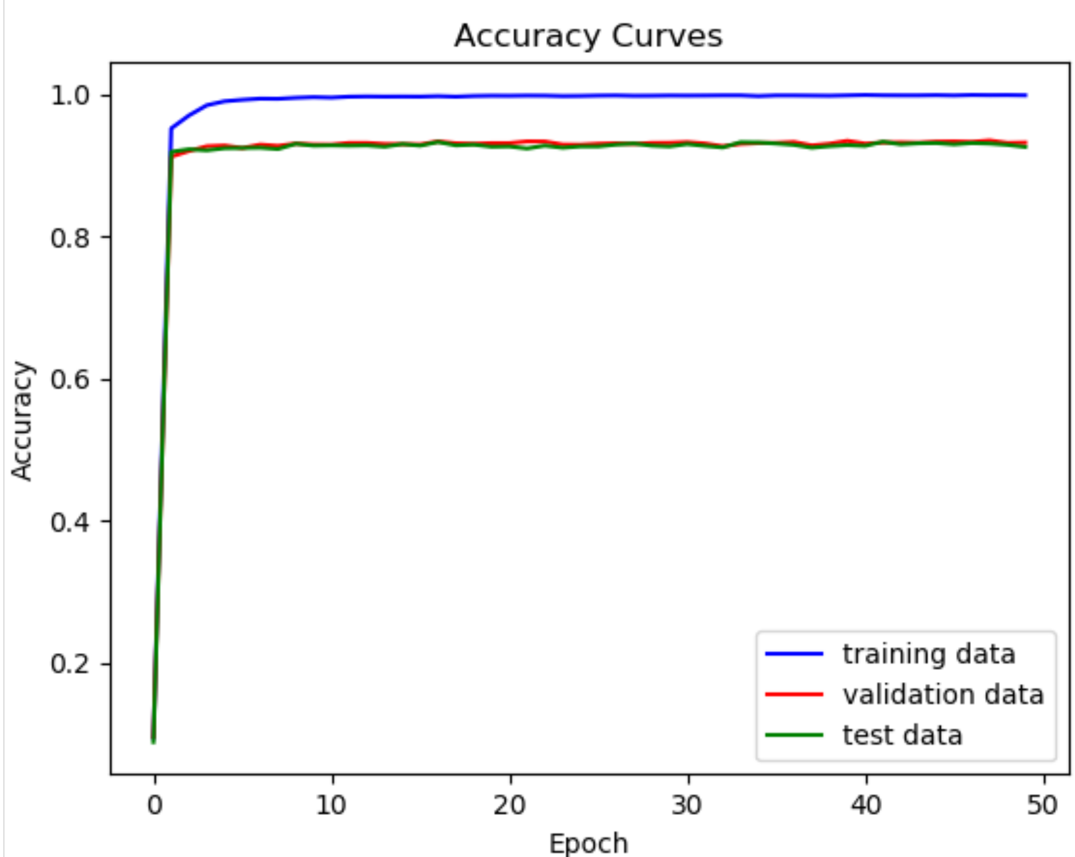
Loss: 0.001539581
Accuracy: 0.9997

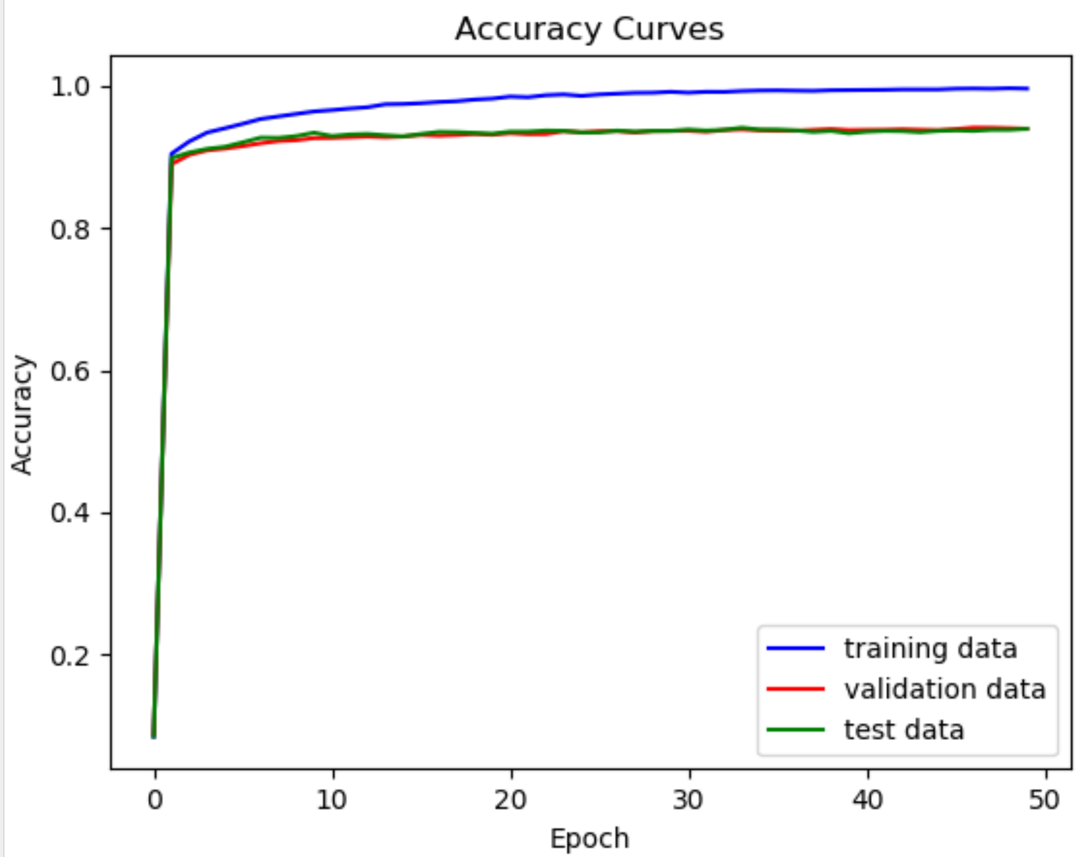
VALIDATION

Loss: 0.39597252
Accuracy: 0.936

TESTING

Loss: 0.45543623
Accuracy: 0.9269457

p=0.75		TRAINING --------------------------- Loss: 0.002391771 Accuracy: 0.9994 VALIDATION --------------------------- Loss: 0.46718428 Accuracy: 0.9316667 TESTING --------------------------- Loss: 0.49651933 Accuracy: 0.9298825
p=0.5		TRAINING --------------------------- Loss: 0.004340526 Accuracy: 0.9992 VALIDATION --------------------------- Loss: 0.45040885 Accuracy: 0.93266666 TESTING --------------------------- Loss: 0.48736116 Accuracy: 0.9265786

p=0.1		TRAINING ---------------------------- Loss: 0.015503812 Accuracy: 0.9955 VALIDATION ---------------------------- Loss: 0.27441043 Accuracy: 0.939 TESTING ---------------------------- Loss: 0.27909115 Accuracy: 0.9390602

* Dropout does not seem very effective when applied to only one fully connected layer in this CNN

Appendix A — starter.py

```
# Implementation of a neural network using only Numpy
#   - trained using gradient descent with momentum
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import time
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

# Load the data
def loadData():
    with np.load("notMNIST.npz") as data:
        Data, Target = data["images"], data["labels"]
        np.random.seed(521)
        randIndx = np.arange(len(Data))
        np.random.shuffle(randIndx)
        Data = Data[randIndx] / 255.0
        Target = Target[randIndx]
        trainData, trainTarget = Data[:10000], Target[:10000]
        validData, validTarget = Data[10000:16000], Target[10000:16000]
        testData, testTarget = Data[16000:], Target[16000:]
    return trainData, validData, testData, trainTarget, validTarget, testTarget

def parseData(data):
    num_data = data.shape[0]
    X = data.reshape(num_data, -1)
    return X

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

# PARSE THE DATA -------------------------------------------------------------------
trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
X_train, X_valid, X_test = parseData(trainData), parseData(validData), parseData(testData)
Y_train, Y_valid, Y_test = convertOneHot(trainTarget, validTarget, testTarget)

def shuffle(trainData, trainTarget):
    np.random.seed(421)
    randIndx = np.arange(len(trainData))
    target = trainTarget
    np.random.shuffle(randIndx)
    data, target = trainData[randIndx], target[randIndx]
    return data, target

def relu(S):
    X = np.copy(S)
    X[S<0] = 0
    return X
```

```
def derivative_relu(S):
    dS = np.zeros_like(S)
    dS[S>0] = 1
    return dS

def softmax(S):
    X = np.exp(S) / np.sum(np.exp(S), axis=1, keepdims=True)
    return X

def computeLayer(X, W, b):
    S = X @ W + b
    return S

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

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

def accuracy(Y, Y_pred):
    j = np.argmax(Y_pred, axis=1)
    i = np.arange(Y.shape[0])
    return np.mean(Y[i, j])
```

Appendix B — neural_network_numpy.py

```
from starter import *
import time

def xavier_init(neurons_in, n_units, neurons_out):
    shape = (neurons_in, n_units)
    var = 2./(neurons_in + neurons_out)
    W = np.random.normal(0, np.sqrt(var), shape)
    return W

def init_weights(n_input, n_hidden, n_output):
    W = []
    W.append(None)
    W.append(xavier_init(n_input, n_hidden, n_output))
    W.append(xavier_init(n_hidden, n_output, 1))
    return W

def init_biases(n_input, n_hidden, n_output):
    b = []
    b.append(None)
    b.append(np.zeros((1, n_hidden)))
    b.append(np.zeros((1, n_output)))
    return b

def forward_propagation(X_input, W, b):
    X, S = [None]*3, [None]*3
    X[0] = X_input

    # UPDATE HIDDEN LAYER
    S[1] = X[0] @ W[1] + b[1]
    X[1] = relu(S[1])

    # UPDATE OUTPUT LAYER
    S[2] = X[1] @ W[2] + b[2]
    X[2] = softmax(S[2])

    return X, S

def backpropagation(X, S, W, Y):
    SENS = [None]*3

    # SEED SENSITIVITY
    N = Y.shape[0]
    SENS[2] = (1/N) * (X[2] - Y)

    # BACKPROPAGATION
    SENS[1] = (SENS[2] @ (W[2]).T) * derivative_relu(S[1])

    return SENS

def compute_gradients(X_input, Y, W, b):
    gradW = [0]*3
    gradb = [0]*3
    N = X_input.shape[0]

    # RUN PROPAGATIONS
    X, S = forward_propagation(X_input, W, b)
    SENS = backpropagation(X, S, W, Y)
```

```
    # GRADIENT of OUTPUT LAYER WEIGHTS + BIASES
    gradW[2] = (X[1]).T @ SENS[2]
    gradb[2] = np.sum(SENS[2], axis=0)

    # GRADIENT of HIDDEN LAYER WEIGHTS + BIASES
    gradW[1] = (X[0]).T @ SENS[1]
    gradb[1] = np.sum(SENS[1], axis=0)

    return gradW, gradb

def measure_performance(W, b):
    Y_pred, S = forward_propagation(X_train, W, b)
    train_loss = avgCE(Y_train, Y_pred[2])
    train_acc = accuracy(Y_train, Y_pred[2])

    Y_pred, S = forward_propagation(X_valid, W, b)
    valid_loss = avgCE(Y_valid, Y_pred[2])
    valid_acc = accuracy(Y_valid, Y_pred[2])

    Y_pred, S = forward_propagation(X_test, W, b)
    test_loss = avgCE(Y_test, Y_pred[2])
    test_acc = accuracy(Y_test, Y_pred[2])

    return train_loss, train_acc, valid_loss, valid_acc, test_loss, test_acc

def gradient_descent(X, Y, n_epochs, alpha, gamma, n_hidden_units):
    n_input_neurons = X.shape[1]
    n_output_neurons = Y.shape[1]

    # INITIALIZE WEIGHTS + BIASES
    W = init_weights(n_input_neurons, n_hidden_units, n_output_neurons)
    b = init_biases(n_input_neurons, n_hidden_units, n_output_neurons)
    VW_o = np.ones_like(W[2]) * 1e-5
    VW_h = np.ones_like(W[1]) * 1e-5
    Vb_o = np.ones_like(b[2]) * 1e-5
    Vb_h = np.ones_like(b[1]) * 1e-5

    # Create Loss/Accuracy dictionaries
    loss = {'train': [], 'valid': [], 'test': []}
    accuracy = {'train': [], 'valid': [], 'test': []}

    for t in range(n_epochs):
        gradW, gradb = compute_gradients(X, Y, W, b)
        print("EPOCH {}".format(t))
        # UPDATE OUTPUT LAYER
        VW_o = gamma * VW_o + alpha * gradW[2]
        W[2] = W[2] - VW_o
        Vb_o = gamma * Vb_o + alpha * gradb[2]
        b[2] = b[2] - Vb_o

        # UPDATE HIDDEN LAYER
        VW_h = gamma * VW_h + alpha * gradW[1]
        W[1] = W[1] - VW_h
        Vb_h = gamma * Vb_h + alpha * gradb[1]
        b[1] = b[1] - Vb_h

        # MEASURE PERFORMANCE
        train_loss, train_acc, valid_loss, valid_acc, test_loss, test_acc = measure_performance(W, b)
        loss['train'].append(train_loss)
        accuracy['train'].append(train_acc)
        loss['valid'].append(valid_loss)
        accuracy['valid'].append(valid_acc)
```

```
        loss['test'].append(test_loss)
        accuracy['test'].append(test_acc)

    return W, b, loss, accuracy

def main():
    start_time = time.time()
    W, b, loss, accuracy = gradient_descent(X_train, Y_train,
                                n_epochs=200,
                                alpha=0.005,
                                gamma=0.9,
                                n_hidden_units=100)
    end_time = time.time()
    print("--- %s seconds ---" % (time.time() - start_time))

    print("TRAINING ----------------------------")
    print("Loss:     ", loss['train'][-1])
    print("Accuracy:", accuracy['train'][-1])

    print("VALIDATION ----------------------------")
    print("Loss:     ", loss['valid'][-1])
    print("Accuracy:", accuracy['valid'][-1])

    print("TESTING ----------------------------")
    print("Loss:     ", loss['test'][-1])
    print("Accuracy:", accuracy['test'][-1])


    plt.plot(loss['train'], color='blue', label='training data')
    plt.plot(loss['valid'], color='red', label='validation data')
    plt.plot(loss['test'], color='green', label='test data')
    plt.legend()
    plt.title('Loss Curves')
    plt.ylabel('Average CE Loss')
    plt.xlabel('Epoch')
    plt.show()

    plt.plot(accuracy['train'], color='blue', label='training data')
    plt.plot(accuracy['valid'], color='red', label='validation data')
    plt.plot(accuracy['test'], color='green', label='test data')
    plt.legend()
    plt.title('Accuracy Curves')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.show()

if __name__ == '__main__':
    main()
```

Appendix C — neural_network_tensorflow.py

```
from starter import *
import time

img_rows = 28
img_cols = 28
img_depth = 1
n_features = 784
n_classes = 10

def convolution_layer(input, n_channels, filter_size, n_filters, name):
    weights = tf.get_variable(
        "weight_{}".format(name),
        shape=[filter_size, filter_size, n_channels, n_filters],
        initializer=tf.contrib.layers.xavier_initializer()
    )
    biases = tf.get_variable(
        "bias_{}".format(name),
        shape=[n_filters],
        initializer=tf.constant_initializer(0.0)
    )
    layer = tf.nn.conv2d(
        input=input,
        filter=weights,
        strides=[1, 1, 1, 1],
        padding="SAME"
    )

    layer = layer + biases
    return layer, weights, biases

def relu_layer(input):
    layer = tf.nn.relu(input)
    return layer

def batch_normalization_layer(input):
    batch_mean, batch_var = tf.nn.moments(input, axes=[0, 1, 2])
    scale = tf.Variable(tf.ones([32]))
    beta = tf.Variable(tf.zeros([32]))

    layer = tf.nn.batch_normalization(input, batch_mean, batch_var,
        offset=beta,
        scale=scale,
        variance_epsilon=1e-3
    )
    return layer

def pooling_layer(input, kernel_size, stride_size):
    layer = tf.nn.max_pool(
        value=input,
        ksize=[1, kernel_size, kernel_size, 1],
        strides=[1, stride_size, stride_size, 1],
        padding="SAME"
    )
    return layer

def flatten_layer(input):
    layer = tf.contrib.layers.flatten(input)
    return layer
```

```
def fully_connected_layer(input, n_inputs, n_outputs, name):
    weights = tf.get_variable(
        "weight_{}".format(name),
        shape=[n_inputs, n_outputs],
        initializer=tf.contrib.layers.xavier_initializer()
    )
    biases = tf.get_variable(
        "bias_{}".format(name),
        shape=[n_outputs],
        initializer=tf.constant_initializer(0.0)
    )

    layer = tf.matmul(input, weights) + biases
    return layer, weights, biases

def build_CNN(learning_rate, weight_decay):
    # Define placeholders for input
    X = tf.placeholder(tf.float32, shape=[None, n_features], name="X")
    Y = tf.placeholder(tf.float32, shape=[None, n_classes], name="Y")
    keep_prob = tf.placeholder(tf.float32, name="keep_prob")

    # INPUT LAYER
    input_layer = tf.reshape(X, shape=[-1, img_rows, img_cols, img_depth])

    # CONVOLUTION LAYER
    conv_layer_1, conv_weights_1, conv_biases_1 = convolution_layer(
        input=input_layer,
        n_channels=1,
        filter_size=3,
        n_filters=32,
        name="conv_1"
    )

    # RELU ACTIVATION
    relu_layer_2 = relu_layer(conv_layer_1)

    # BATCH NORMALIZATION LAYER
    batch_norm_layer_3 = batch_normalization_layer(relu_layer_2)

    # MAX POOLING LAYER (2X2)
    max_pool_layer_4 = pooling_layer(batch_norm_layer_3, kernel_size=2, stride_size=2)

    # FLATTEN LAYER
    flatten_layer_5 = flatten_layer(max_pool_layer_4)

    # FULLY CONNECTED LAYER
    full_connect_layer_6, fc_weights_6, fc_biases_6 = fully_connected_layer(
        input=flatten_layer_5,
        n_inputs=6272, # calculated dimension of flattened_layer_5 = 14x14x32
        n_outputs=784,
        name="fc_layer_1"
    )

    # APPLY DROPOUT
    drop_out = tf.nn.dropout(full_connect_layer_6, keep_prob=keep_prob)

    # RELU ACTIVATION
    relu_layer_7 = relu_layer(drop_out)

    # FULLY CONNECTED LAYER
    full_connect_layer_8, fc_weights_8, fc_biases_8 = fully_connected_layer(
```

```
        input=relu_layer_7,
        n_inputs=784,
        n_outputs=10,
        name="fc_layer_2"
    )

    # SOFTMAX OUTPUT
    Y_pred = tf.nn.softmax(full_connect_layer_8)

    # CROSS ENTROPY LOSS
    cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(
        labels=Y,
        logits=full_connect_layer_8
    )
    # REGULARIZATION
    regularizer = tf.nn.l2_loss(conv_weights_1) \
                + tf.nn.l2_loss(fc_weights_6) \
                + tf.nn.l2_loss(fc_weights_8)
    # TOTAL LOSS
    loss = tf.reduce_mean(cross_entropy) + weight_decay * regularizer

    # CLASSIFICATION ACCURACY
    Y_class = tf.argmax(Y, axis=1)
    Y_pred_class = tf.argmax(Y_pred, axis=1)
    correct_predictions = tf.equal(Y_class, Y_pred_class)
    accuracy = tf.reduce_mean(tf.cast(correct_predictions, tf.float32))

    # ADAM OPTIMIZER
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)

    return optimizer, X, Y, keep_prob, loss, accuracy


def train_CNN(X_data, Y_data, batch_size, n_epochs, learning_rate, weight_decay, p):
    n_datapoints = X_data.shape[0]
    iter_per_epoch = int(np.ceil(n_datapoints/batch_size))
    iterations = iter_per_epoch * n_epochs

    # Create Loss/Accuracy dictionaries to store performance data
    loss_curves = {'train': [], 'valid': [], 'test': []}
    accuracy_curves = {'train': [], 'valid': [], 'test': []}

    # SET UP COMPUTATIONAL GRAPH
    optimizer, X, Y, keep_prob, loss, accuracy = build_CNN(learning_rate, weight_decay)
    global_init = tf.global_variables_initializer()

    with tf.Session() as sess:
        sess.run(global_init)
        for iter in range(iterations):
            # NEW EPOCH
            if iter % iter_per_epoch == 0:
                print("EPOCH {}".format(int((iter+1)/iter_per_epoch)))

                # CALCULATE TRAINING LOSS & ACCURACY
                feed_dict_train = {X: X_data, Y: Y_data, keep_prob: 1.0}
                _loss, _acc = sess.run([loss, accuracy], feed_dict=feed_dict_train)
                loss_curves['train'].append(_loss)
                accuracy_curves['train'].append(_acc)

                # CALCULATE VALIDATION LOSS & ACCURACY
                feed_dict_valid = {X: X_valid, Y: Y_valid, keep_prob: 1.0}
                _loss, _acc = sess.run([loss, accuracy], feed_dict=feed_dict_valid)
```

```
            loss_curves['valid'].append(_loss)
            accuracy_curves['valid'].append(_acc)

            # CALCULATE TEST LOSS & ACCURACY
            feed_dict_test = {X: X_test, Y: Y_test, keep_prob: 1.0}
            _loss, _acc = sess.run([loss, accuracy], feed_dict=feed_dict_test)
            loss_curves['test'].append(_loss)
            accuracy_curves['test'].append(_acc)

            # SHUFFLE DATA ON NEW EPOCH
            X_data, Y_data = shuffle(X_data, Y_data)

        # SELECT MINI-BATCH
        X_batch, Y_batch = X_data[:batch_size], Y_data[:batch_size]

        # GRADIENT DESCENT STEP on mini-batch
        feed_dict_batch = {X: X_batch, Y: Y_batch, keep_prob: p}
        sess.run([optimizer], feed_dict=feed_dict_batch)

        # SHIFT DATA BY BATCH SIZE so next sample is new
        X_data = np.roll(X_data, batch_size, axis=0)
        Y_data = np.roll(Y_data, batch_size, axis=0)

    return loss_curves, accuracy_curves


def main():
    start_time = time.time()
    loss, accuracy = train_CNN(
        X_train,
        Y_train,
        batch_size=32,
        n_epochs=50,
        learning_rate=1e-4,
        weight_decay=0,
        p=1.0
    )
    end_time = time.time()
    print("--- %s seconds ---" % (time.time() - start_time))

    print("TRAINING -----------------------------")
    print("Loss:     ", loss['train'][-1])
    print("Accuracy:", accuracy['train'][-1])

    print("VALIDATION ----------------------------")
    print("Loss:     ", loss['valid'][-1])
    print("Accuracy:", accuracy['valid'][-1])

    print("TESTING -----------------------------")
    print("Loss:     ", loss['test'][-1])
    print("Accuracy:", accuracy['test'][-1])


    plt.plot(loss['train'], color='blue', label='training data')
    plt.plot(loss['valid'], color='red', label='validation data')
    plt.plot(loss['test'], color='green', label='test data')
    plt.legend()
    plt.title('Loss Curves')
    plt.ylabel('CE Loss')
    plt.xlabel('Epoch')
    plt.show()
```

```
    plt.plot(accuracy['train'], color='blue', label='training data')
    plt.plot(accuracy['valid'], color='red', label='validation data')
    plt.plot(accuracy['test'], color='green', label='test data')
    plt.legend()
    plt.title('Accuracy Curves')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.show()


if __name__ == '__main__':
    main()
```