ECE421S – Introduction to Machine Learning

Assignment 2

Neural Networks

Hard Copy Due: March 6, 2019 @ BA3014, 4:00-5:00 PM EST

Code Submission: ece421ta2019@gmail.com

March 6, 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 (Work Split)	StudentID
Michael Tang (50%)	1001389525
Khoi To (50%)	1002078477

1. Neural Network Using Numpy

1.1 Helper Functions:

1. ReLu:

```
def relu(x): #implements relu
    return x * (x > 0)
```

2. Softmax:

```
def softmax(x):

    return np.exp(x)/sum(np.exp(x))
```

3. Compute:

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

4. AverageCE:

```
def CE(target, prediction): #input should have one-hot targets and predictions as rows
    N = target.shape[0]
    output = np.sum(target*np.log(prediction + 1e-9), axis=1)
    output = -(1/N)*np.sum(output)
    return output
```

A small value was added to the logarithm to prevent instabilities when the argument is 0.

5. GradCE:

For a single data point:

$$L = -\sum_{k=1}^{K} t_k^n log(s_k^n)$$

$$\frac{dL}{ds} = -\sum_{k=1}^{K} t_k^n * \frac{1}{s_k^n}$$

The targets are one-hot encoded therefore the only term that is not 0 is when $t_k^n = 1$.

$\frac{dL}{ds} = -\frac{1}{s_k^n}$, k == class of data point

Therefore the average gradCE is:

$\frac{dL}{ds} = -\sum_{n=1}^{N} \sum_{k=1}^{K}$ u(t_k^n) $\frac{1}{s_k^n}$, where u(n) = 1 for $n > 0$ *and is* 0 *otherwise.* .

```
def gradCE(target, prediction): #return average gradCE

    #perform row-wise dot product
    N = target.shape[0]
    output = np.sum(target*np.reciprocal(prediction), axis = 0)
    output = -(1/N)*output #vector of averaged gradients according to the dataset

    return output
```

1.2 Backpropagation Derivation

1. Gradient of loss wrt to outer layer weights:
$\frac{\partial L}{\partial W_o} = X_h{}^T(Y-T),\ where\ X_h\ are\ the\ outputs\ from\ the\ nodes\ in\ the\ hidden\ layer,\ Y\ are\ the\ predicted\ tags\ and\ T\ are\ the\ T$

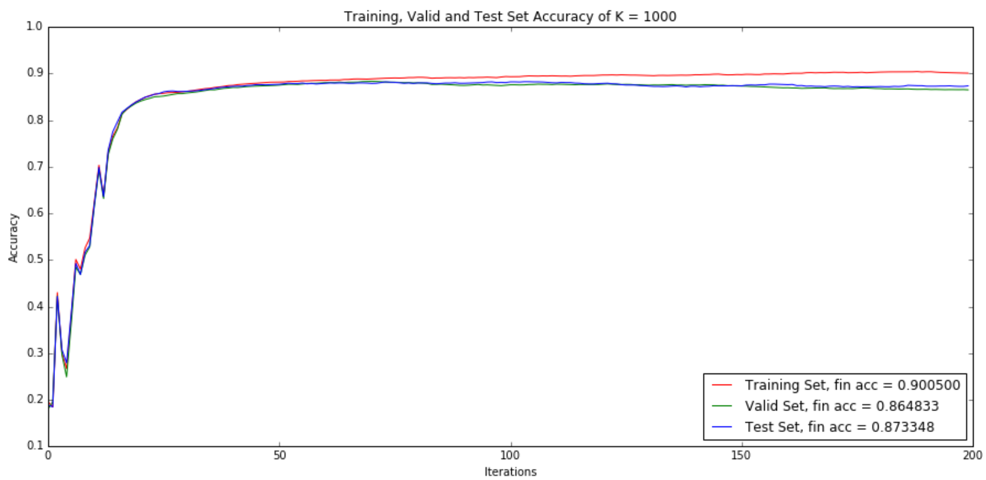
2.
$\frac{\partial L}{\partial b_o}$ = $(Y-T)$

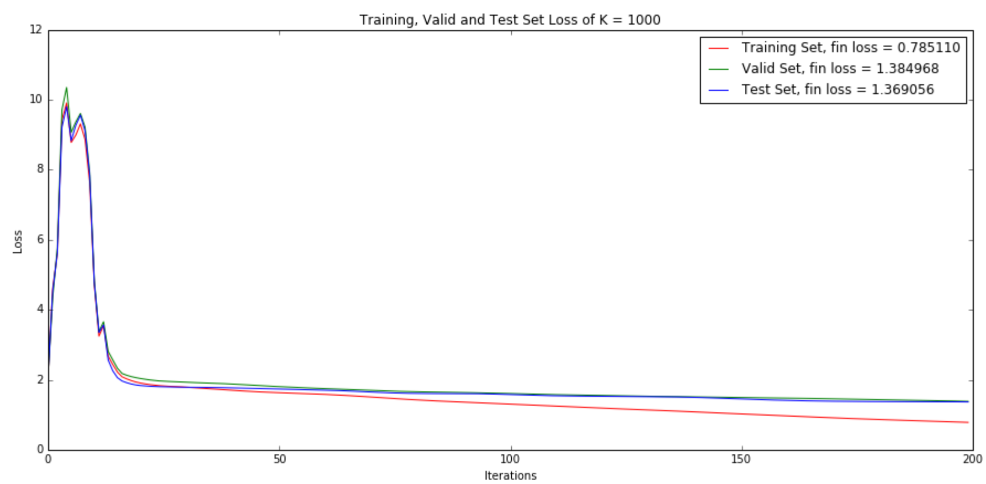
3.
$\frac{\partial L}{\partial W_h} = X_{in}{}^T(Y-T)W_o{}^T \otimes U(Z_h)$, where $\otimes$ denotes element-wise multiplication and U denotes the Heaviside step function and Z_h denotes the sum term matrix before the hidden layer.

4.
$\frac{\partial L}{\partial b_h} = (Y-T)W_o{}^T \otimes U(Z_h)$

1.3 Learning

Code can be found in *Appendix A.1 Part 1 code*.



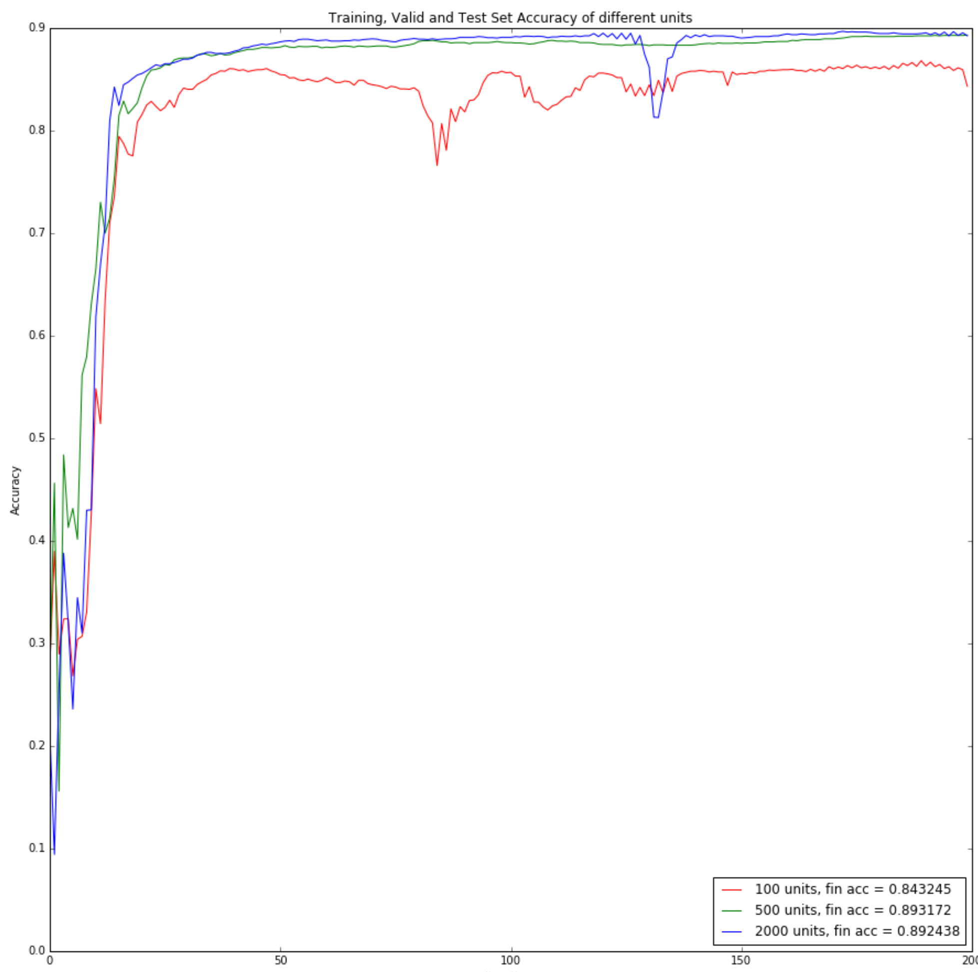


The results from the test are shown in the above graph, with the final accuracies and losses shown in the legends. The learning rate, alpha, was set to 0.01, a small number which is an often recommended rule of thumb and that we found worked well amongst the few that we tried (1, 0.10, 0.01), with alpha = 1 giving a very unstable learning.

1.4 Hyperparameter Investigation

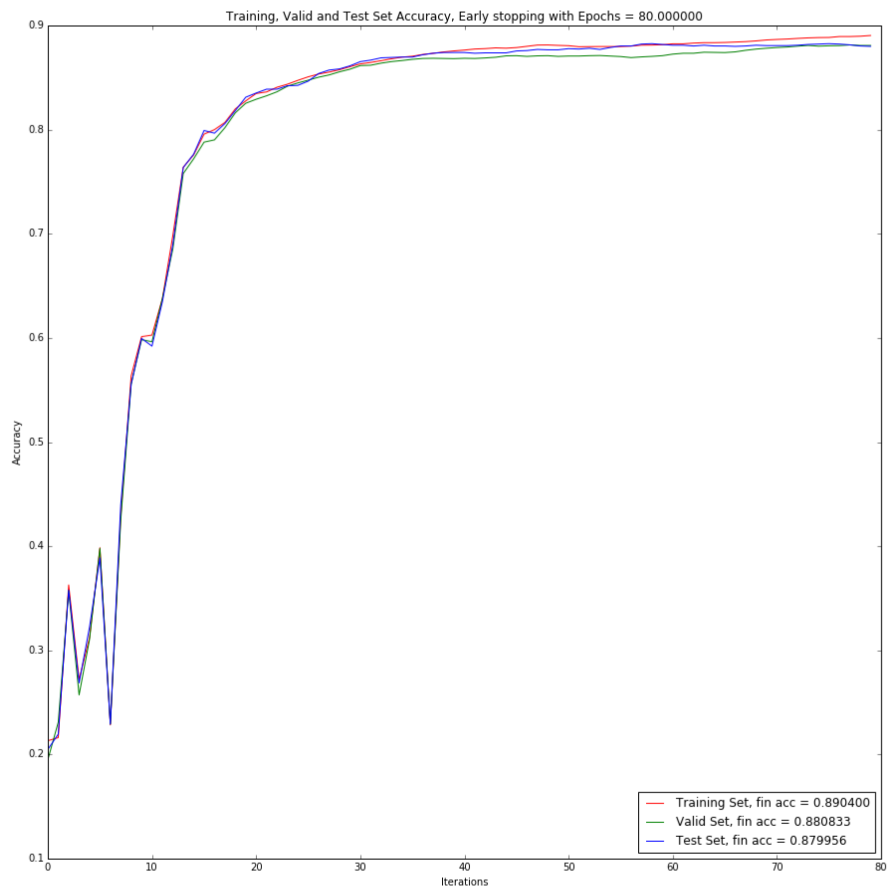
1. Number of hidden units:

In general, more hidden units is able to provide our model with better accuracy. We can see a 5% final accuracy increase (shown in the legend) between 100 and 2000 units. There is, however, a very significant trade-off in training time with 2000 units taking multiple times longer. 500 units seems to be a good 'sweetspot' in that it achieves very high accuracy while not taking too long to train. The accuracy increase can somewhat be explained by the universal approximation theory which state that a sufficient number of hidden units will allow a NN to approximate any arbitrary function.



2. Early Stopping

I identity the iterations at which testing accuracy no longer increases and that further training only increasings training set accuracy. I stop it at epoch 80 to help reduce overfitting. In comparison to 1.3, the testing accuracy increased by about 1%. The final accuracy plot is shown below:



2. Neural Networks in Tensorflow

2.1 Model implementation

Convolutional neural network is implemented as below. For Cross Entropy Loss, *tf.nn.softmax_cross_entropy_with_logits_v2* is employed because this is multiple classification logistic regression model.

```
import tensorflow as tf

class ConvolutionalNeuralNetwork(object):
    def build_model(self,
            seed=421,#tf seed
            alpha=10e-4, #learning rate for ADAM optimizer
            with_dropout=False, p=0.9, #dropout
            with_regularizers=False, beta=0.01 #regulizer
            ):
        #initialize
        tf.reset_default_graph()
        tf.set_random_seed(seed)


        # label
        self.y = tf.placeholder(tf.float32, [None, 10], 'y')

        # 1. input layer (dim: 28x28x1)
        self.x = tf.placeholder(tf.float32, [None, 28, 28], 'x')
        x_reshaped = tf.reshape (self.x, [-1, 28, 28, 1])

        # 2. 3 × 3 conv layer, 32 filters, vertical/horizontal strides of 1
        W_conv = tf.get_variable(
            'W_conv',
            shape=(3,3,1,32), #0,1:filter size, 2: channel, 3: filter numbers
            initializer=tf.contrib.layers.xavier_initializer() #Xavier scheme
            )
        b_conv = tf.get_variable(
            'b_conv',
            shape=(32),
            initializer=tf.contrib.layers.xavier_initializer()
            )
```

```
conv_layer = tf.nn.conv2d(
   input=x_reshaped,
   filter=W_conv,
   strides=[1,1,1,1], #0: image number, 1,2:h/v stride, 3: # of channel
   padding='SAME',
   name='conv_layer'
   )
conv_layer = tf.nn.bias_add(conv_layer, b_conv) #output dim: 28x28x32

# 3. ReLU activation
relu_conv = tf.nn.relu (conv_layer)

# 4. A batch normalization layer
mean, variance = tf.nn.moments(relu_conv, axes=[0])

bnorm_layer = tf.nn.batch_normalization(
   relu_conv,
   mean=mean,
   variance=variance,
   offset=None, scale=None,
   variance_epsilon=1e-3
   )

# 5. A 2 × 2 max pooling layer (dim: 14x14x32)
maxpool2x2_layer = tf.nn.max_pool(
   bnorm_layer,
   ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
   padding='SAME'
   )

# 6. Flatten layer
flatten_layer = tf.reshape(maxpool2x2_layer, [-1, 14*14*32])

# 7. Fully connected layer (with 784 output units, i.e. corresponding to each pixel)
W_fcl_784 = tf.get_variable(
      'W_fcl_784',
      shape=(14*14*32, 784),
      initializer=tf.contrib.layers.xavier_initializer() #Xavier scheme
      )
b_fcl_784 = tf.get_variable(
      'b_fcl_784',
      shape=(784),
```

```
        initializer=tf.contrib.layers.xavier_initializer() #Xavier scheme
        )
    fullconn784_layer = tf.add(tf.matmul(flatten_layer, W_fcl_784), b_fcl_784)

    #drop out
    if with_dropout:
      fullconn784_layer = tf.nn.dropout(fullconn784_layer, keep_prob=p)

    # 8. ReLU activation
    relu_fcl = tf.nn.relu (fullconn784_layer)

    # 9. Fully connected layer (with 10 output units, i.e. corresponding to each class)
    W_fcl_10 = tf.get_variable(
        'W_fcl_10',
        shape=(784, 10),
        initializer=tf.contrib.layers.xavier_initializer() #Xavier scheme
        )
    b_fcl_10 = tf.get_variable(
        'b_fcl_10',
        shape=(10),
        initializer=tf.contrib.layers.xavier_initializer() #Xavier scheme
        )
    fullconn10_layer = tf.add(tf.matmul(relu_fcl, W_fcl_10), b_fcl_10)

    # 10. Softmax output
    y_hat = tf.nn.softmax(fullconn10_layer)

    # 11. Cross Entropy loss
    # normal loss
    self.loss = tf.reduce_mean (tf.nn.softmax_cross_entropy_with_logits_v2(logits=y_hat,
labels=self.y))
    # loss with l2 regularizers
    if with_regularizers:
      regularizers = tf.nn.l2_loss(W_conv) + tf.nn.l2_loss(W_fcl_784) + tf.nn.l2_loss(W_fcl_10)
      self.loss = tf.reduce_mean(self.loss + beta*regularizers)

    # optimizer
    self.optimizer = tf.train.AdamOptimizer(learning_rate=alpha).minimize(self.loss)

    # compute accuracy
    correct_prediction = tf.equal(tf.argmax(y_hat, 1), tf.argmax(self.y, 1))
    self.accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

2.2 Model Training

Stochastic Gradient Descent class is implemented in *sgd.py*. Epochs and batch size are set to 50 and 32 respectively by default.

```
from cnn import ConvolutionalNeuralNetwork
import tensorflow as tf

class StochasticGradientDescent(object):
    def __init__(self, data, recorder, cnn):
        self.dt  = data
        self.rc  = recorder
        self.cnn = cnn

    def build_trainer(self, epochs=50, batch_size=32):
        dt = self.dt
        cnn = self.cnn
        init = tf.global_variables_initializer()

        with tf.Session() as sess:
            sess.run(init)
            n = len(dt.y_train_oh)
            x = cnn.x
            y = cnn.y
            # SGD
            for i in range(epochs):
                #shuffle
                x_shuffled, y_shuffled = dt.shuffle(dt.x_train, dt.y_train_oh)
                #go through all batches
                for j in range(0, n, batch_size):
                    x_batch, y_batch = x_shuffled[j:j+batch_size], y_shuffled[j:j+batch_size]
                    # run optimizer
                    sess.run (cnn.optimizer, feed_dict = {x: x_batch, y: y_batch})

                loss_train, acc_train = sess.run ([cnn.loss, cnn.accuracy], feed_dict = {x: dt.x_train, y: dt.y_train_oh})
                loss_valid, acc_valid = sess.run ([cnn.loss, cnn.accuracy], feed_dict = {x: dt.x_valid, y: dt.y_valid_oh})
                loss_test, acc_test = sess.run ([cnn.loss, cnn.accuracy], feed_dict = {x: dt.x_test, y: dt.y_test_oh})
                print ("Iteration: ", i,
                       "Train: ", loss_train, acc_train, ' \ '
                       "Valid: ", loss_valid, acc_valid, ' \ '
                       "Test: ", loss_test, acc_test)

                self.rc.train = self.rc.train.append({'loss': loss_train, 'accuracy': acc_train}, ignore_index=True)
                self.rc.valid = self.rc.valid.append({'loss': loss_valid, 'accuracy': acc_valid}, ignore_index=True)
                self.rc.test = self.rc.test.append({'loss': loss_test, 'accuracy': acc_test}, ignore_index=True)
```

Basic test is conducted in *main.py* with default/required values (learning rate: 10^-4, epochs: 50, batch size: 32)

```
from cnn import ConvolutionalNeuralNetwork
from sgd import StochasticGradientDescent
from data import Data
from recorder import Recorder
from plotter import Plotter

import matplotlib.pyplot as plt

dt = Data()
rc = Recorder()
cnn = ConvolutionalNeuralNetwork()
sgd = StochasticGradientDescent(dt, rc, cnn)
plotter = Plotter(rc)

dt.load('notMNIST.npz')

#%% 2.1 + 2.2 Convolutional Neural Network +  Stochastic Gradient Descent

cnn.build_model(
   seed=421,#tf seed
   alpha=1e-4, #learning rate for ADAM optimizer
   with_dropout=False, p=0.9, #dropout
   with_regularizers=False, beta=0.01 #regulizer
)
sgd.build_trainer (
   epochs=50,
   batch_size=32
)

# plot loss & accuracy
plotter.plot_train_valid_test('img/basic')
```

Below figures are the results for train, valid, and test sets:

Figure 2.2.1 Train Loss & Accuracy over 50 epochs

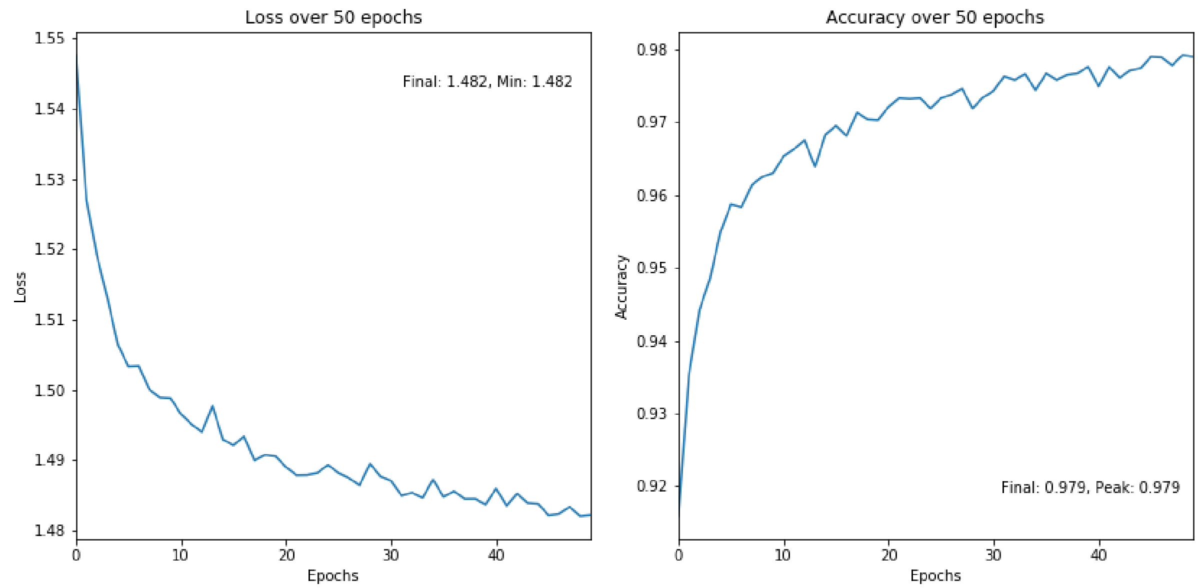


Figure 2.2.2 Valid Loss & Accuracy over 50 epochs

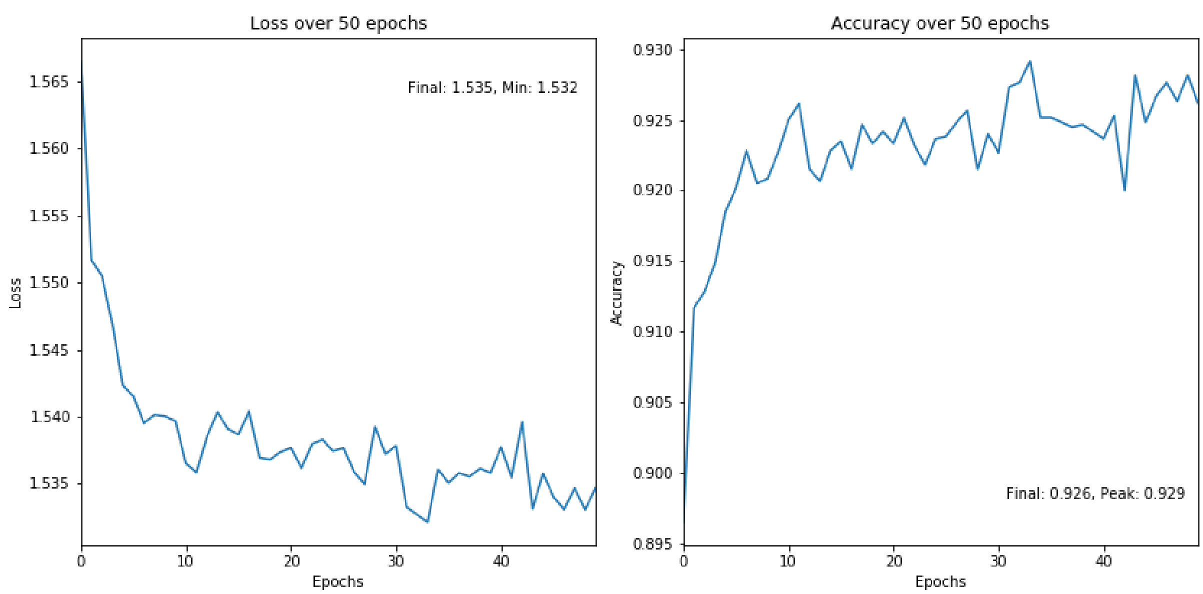


Figure 2.2.3 Test Loss & Accuracy over 50 epochs

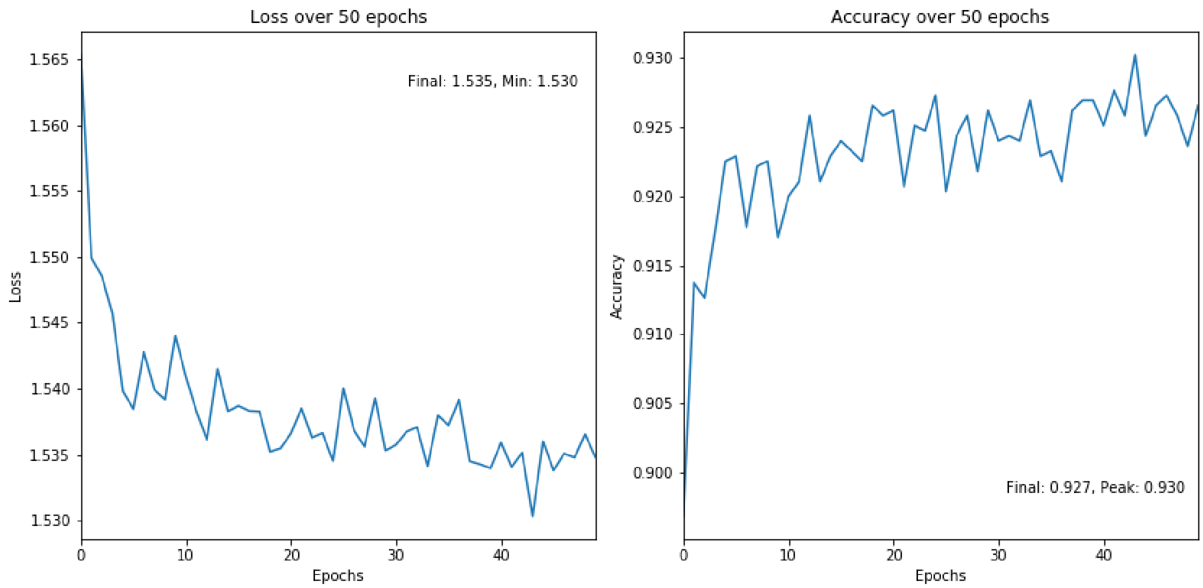


Table 2.2 Final training, validation and test accuracies

Set	Training	Validation	Test
Accuracy	**97.9%**	**92.6%**	**92.7%**

Observation:

- Loss and accuracy curves for validation and test sets seems to fluctutate strongly over the whole course of training. It means in every epoch, the model is trying to overfit to a batch of the training data. As a result, it is less fit to the validation and test data.
- At the end of training, training accuracy is able to reach 97.9%, meanwhile, valid and test accuracy only reach 92.6% & 92.7% respectively. The gap is approximately 5%. This means the model is a bit overfitting towards the training set.
- However, training accuracy curve converges very early at around 10 epoch.

2.3 Hyperparameter Investigation

1. L2 Normalization

Setup code can be found in *Appendix A.2.3.1 L2 Normalization*.

1.1 Weight decay coefficient = 0.01

Figure 2.3.1.1.1 Train Loss & Accuracy over 50 epochs

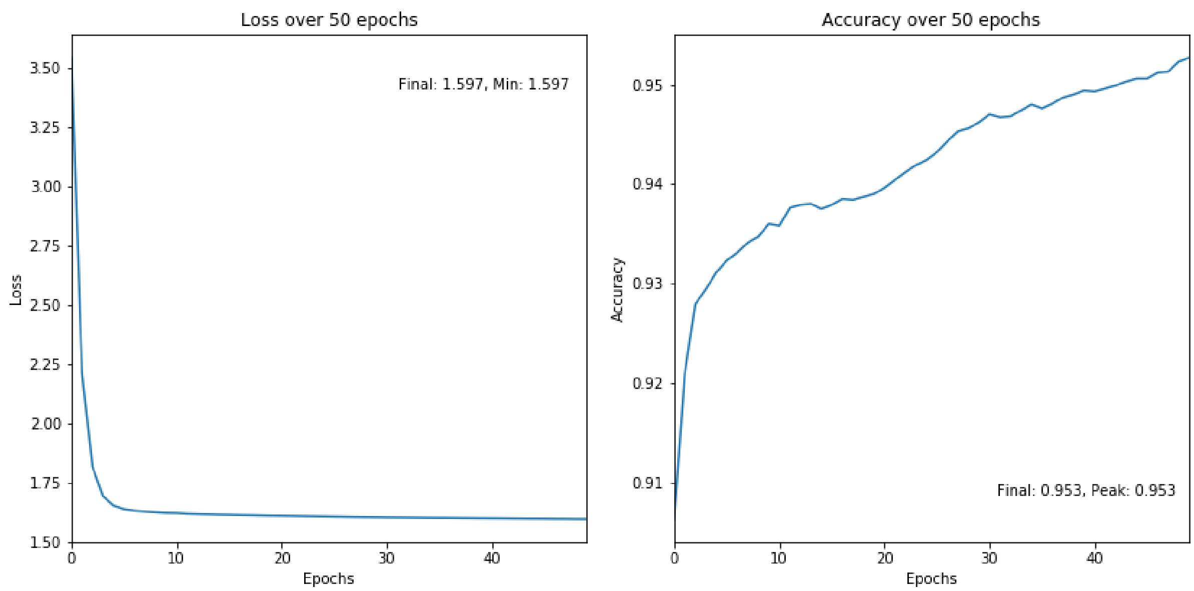


Figure 2.3.1.1.2 Valid Loss & Accuracy over 50 epochs

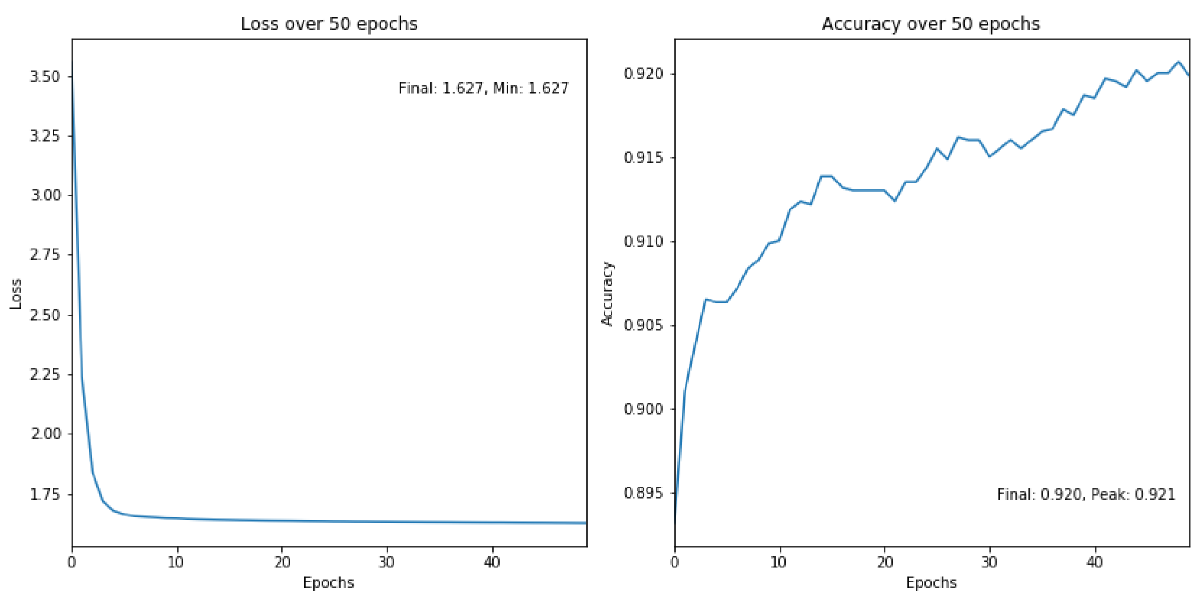


Figure 2.3.1.1.3 Test Loss & Accuracy over 50 epochs

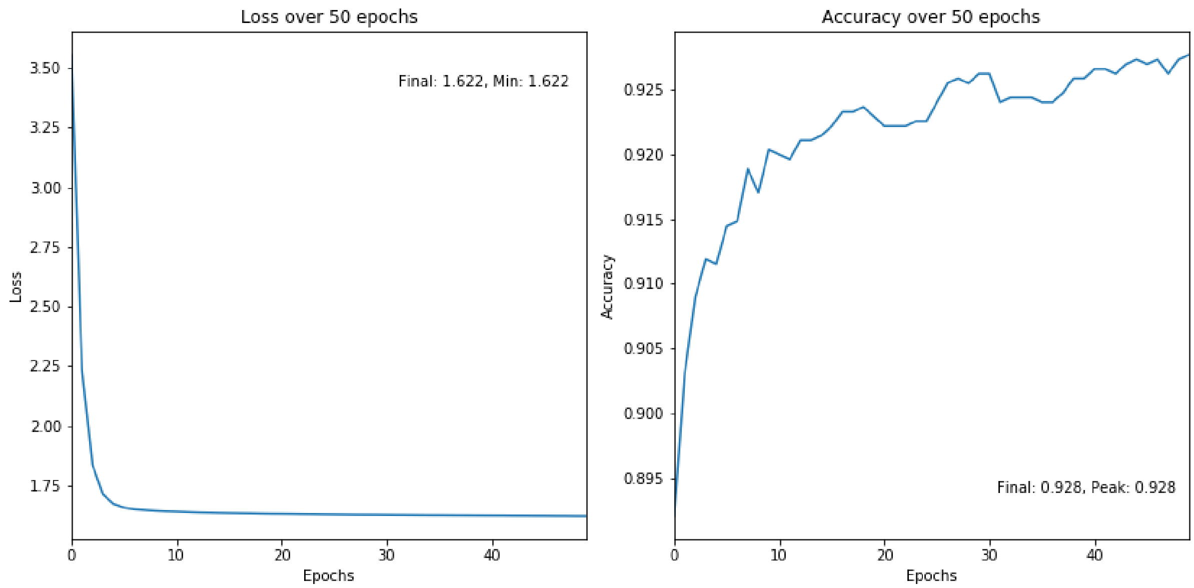


Table 2.2 Final training, validation and test accuracies

Set	Training	Validation	Test
Accuracy	**95.3%**	**92.0%**	**92.8%**

Observation:

- After L2 Normalization is applied, we can clearly see that loss and accuracy curves for training, validation, and test cases are smooth out.
- Training accuracy is decreased by around 2.7% comparing to the very first training process, however, accuracy gap between training and validation/test cases becomes smaller at around 3%.
- The fact that training accuracy is lost by 2.7% proves that L2 regularization makes the model less overfitting to the training set, giving a slightly higher test accuracyy.
- However, the 3 accuracy curves cannot converge propably as an effect of regularization (it probably requires more epochs).

1.2 Weight decay coefficient = 0.1

Figure 2.3.1.2.1 Train Loss & Accuracy over 50 epochs

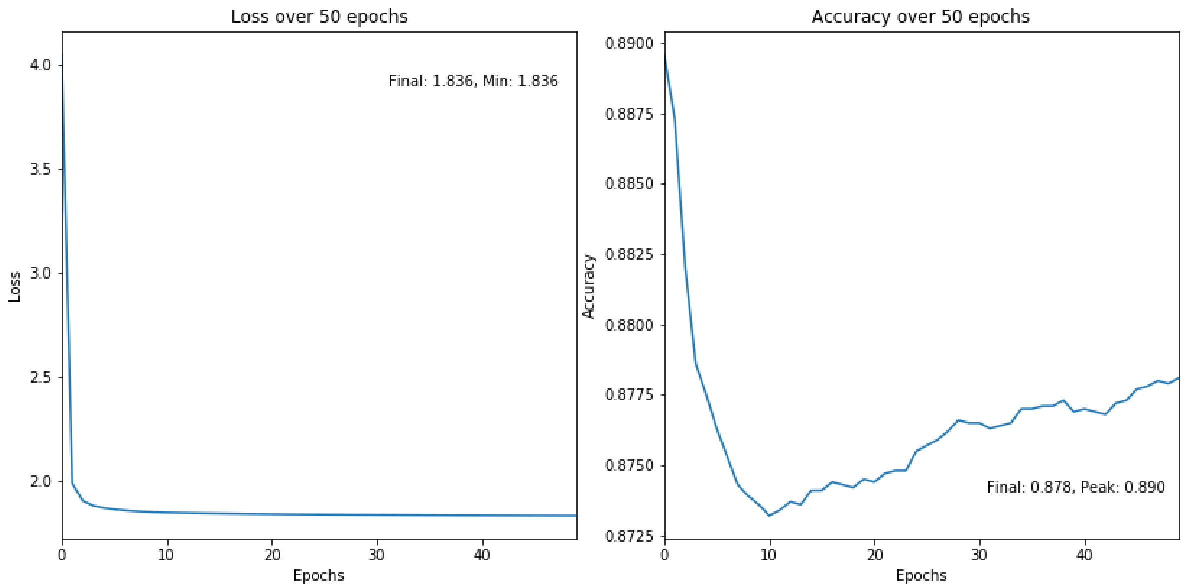


Figure 2.3.1.2.2 Valid Loss & Accuracy over 50 epochs

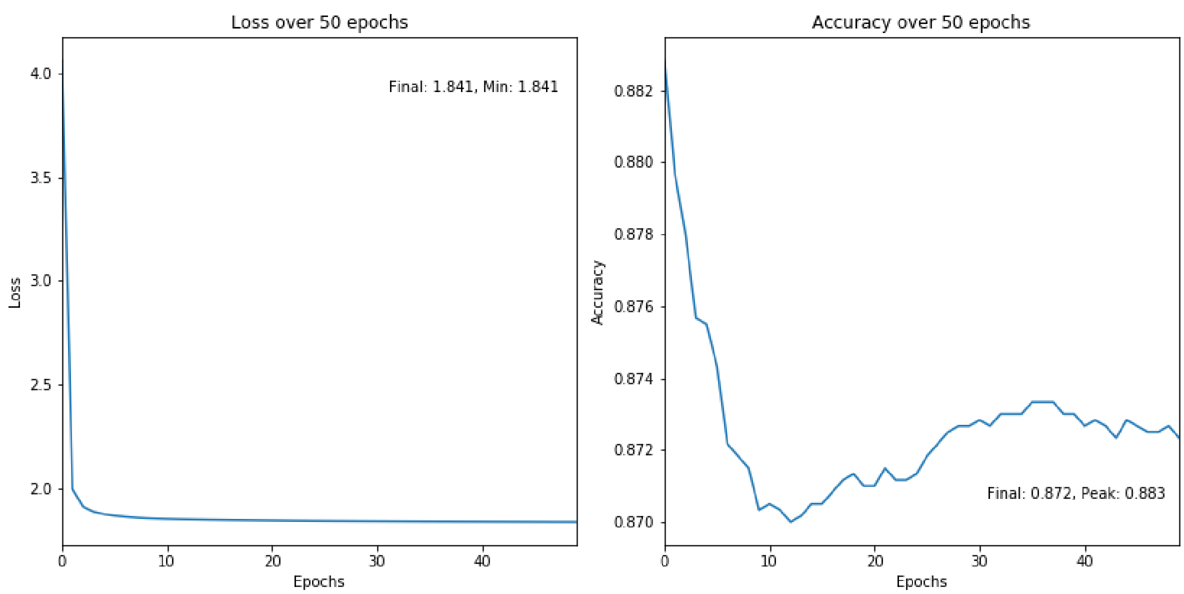


Figure 2.3.1.2.3 Test Loss & Accuracy over 50 epochs

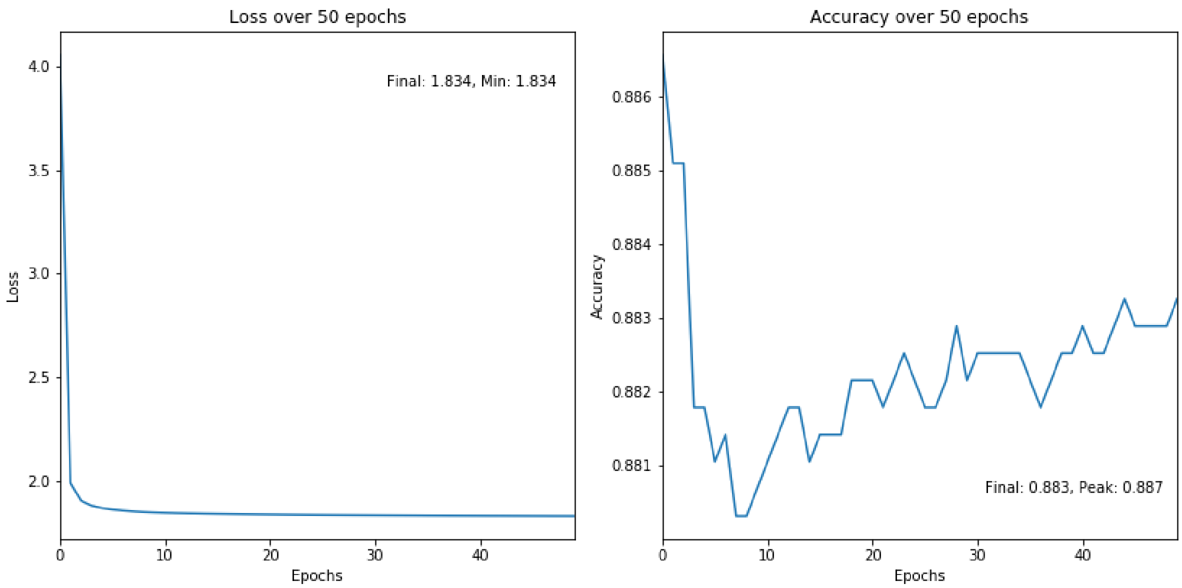


Table 2.2 Final training, validation and test accuracies

Set	Training	Validation	Test
Accuracy	**87.8%**	**87.2%**	**88.3%**

Observation:

- After increase the weight decay to 0.1, accuracy of 3 cases is suprisingly decreased for the first 10 epochs. For example, test accuracy decreases from 88.3% to 87.2%.
- The loss and accuracy curves seems to flutuate.
- Training, validation, and test accuracy is very close (difference is about 0.5%). This means the model almost solves the overfitting problem.
- However, comparing to the training without L2 regularization, the training, validation, and test accuracy drops by about 10.1%, 5.4%, and 4.4%. This means the model is starting to be underfitting, we are overconstraining our model with such a high regularization and it is unable to predict the labels accurately.

1.3 Weight decay coefficient = 0.5

Figure 2.3.1.3.1 Train Loss & Accuracy over 50 epochs

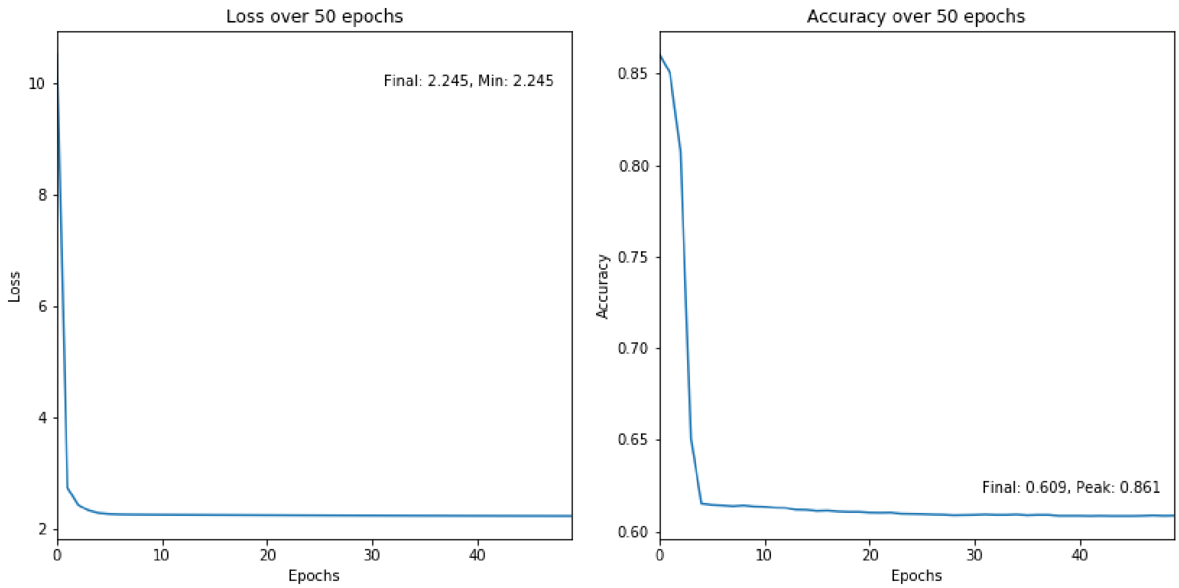


Figure 2.3.1.3.2 Valid Loss & Accuracy over 50 epochs

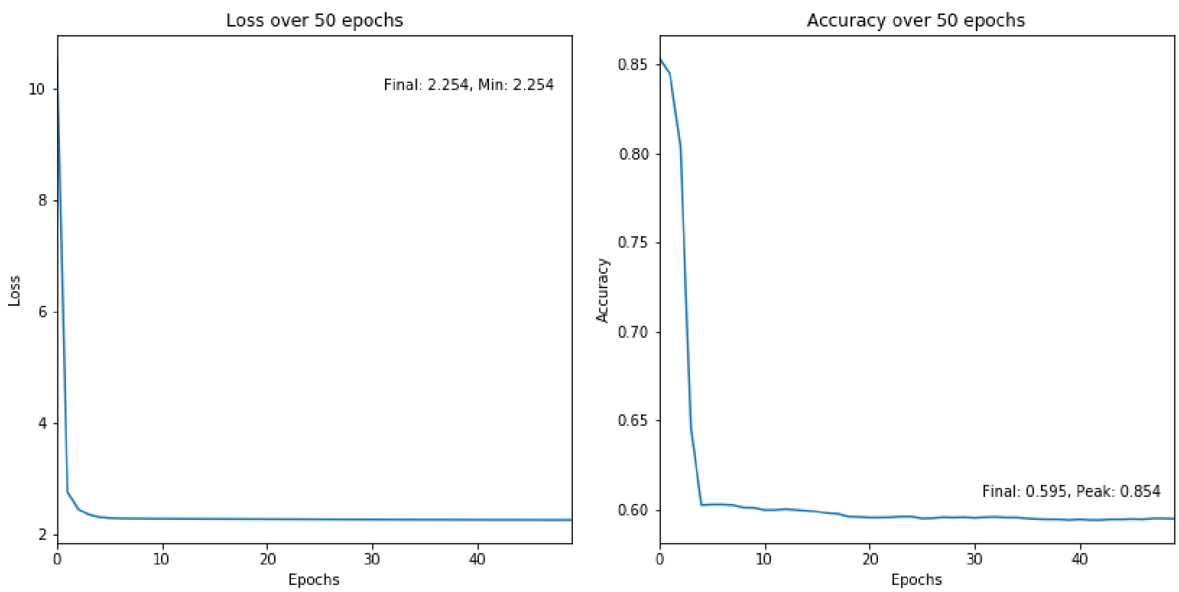
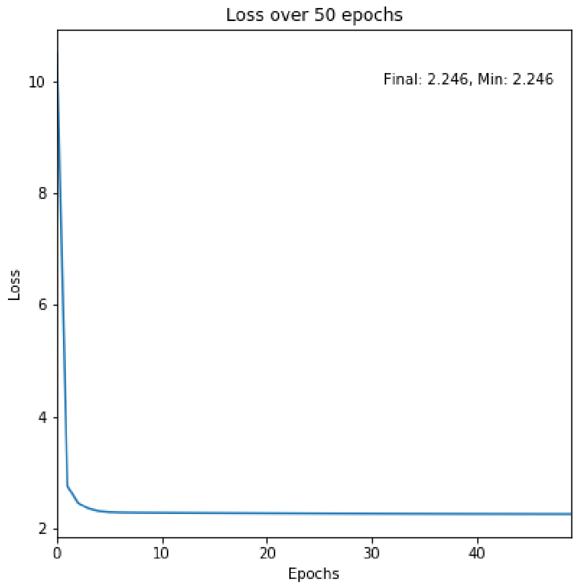


Figure 2.3.1.3.3 Test Loss & Accuracy over 50 epochs



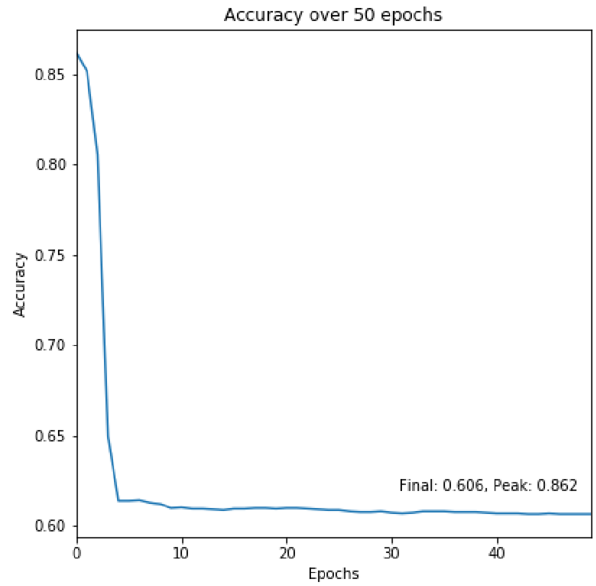


Table 2.2 Final training, validation and test accuracies

Set	Training	Validation	Test
Accuracy	**60.9%**	**59.5%**	**60.6%**

Observation:

- Accuracy curves of 3 cases completely drop from about 86% to 60% and converge after 5 epochs.
- Although accuracy gap is very small, accuracy losses by more than 30% for 3 sets comparing to the first experiment.
- We shall conclude that the model is already underfitting, or extremely constrained such that it no longer has the needed ability to discriminate properly between the images. Overall, we see that too much L2 regularization can decrease the accuracies.

2. Dropout

Setup code can be found in *Appendix A.2.3.2 Dropout*.

2.1 P = 0.9

Figure 2.3.2.1.1 Training Loss & Accuracy over 50 epochs

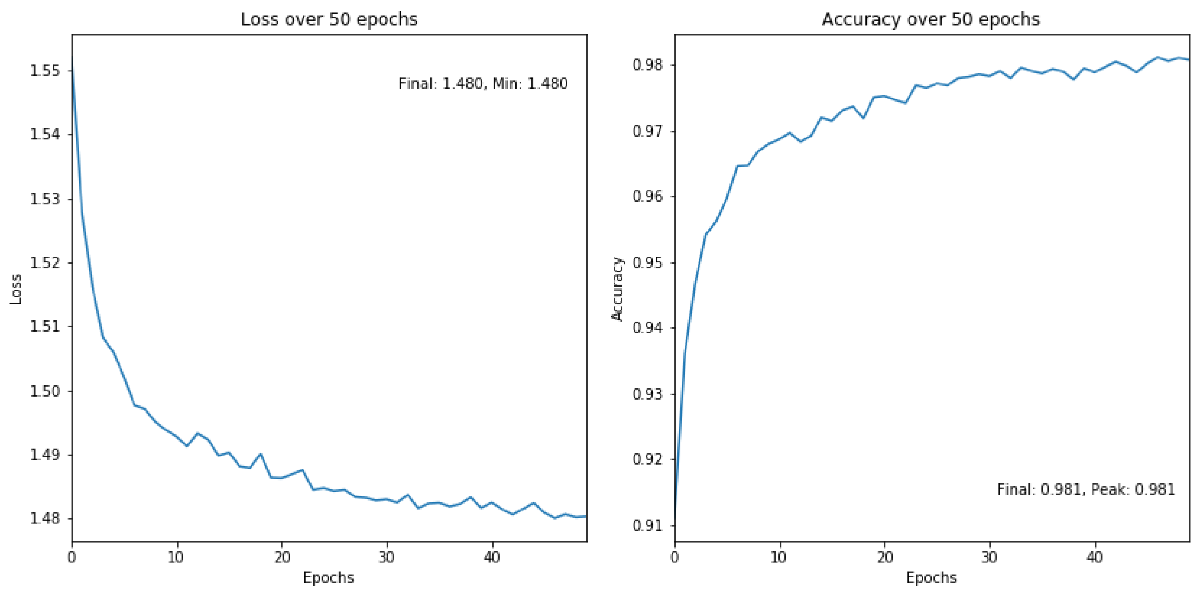


Figure 2.3.2.1.2 Validation Loss & Accuracy over 50 epochs

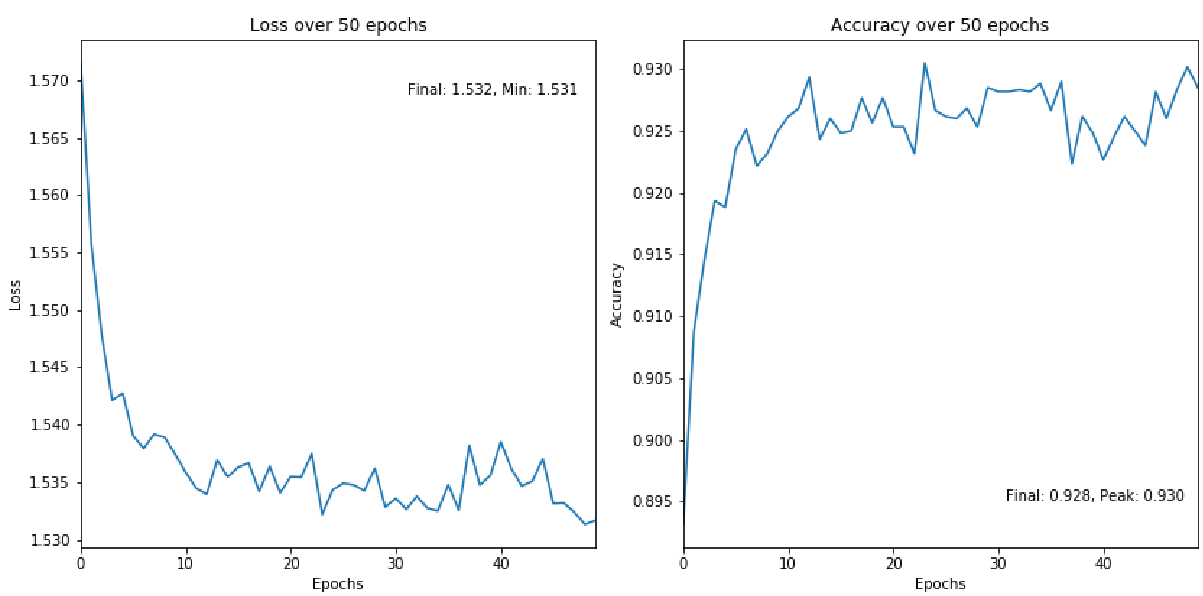


Figure 2.3.2.1.3 Test Loss & Accuracy over 50 epochs

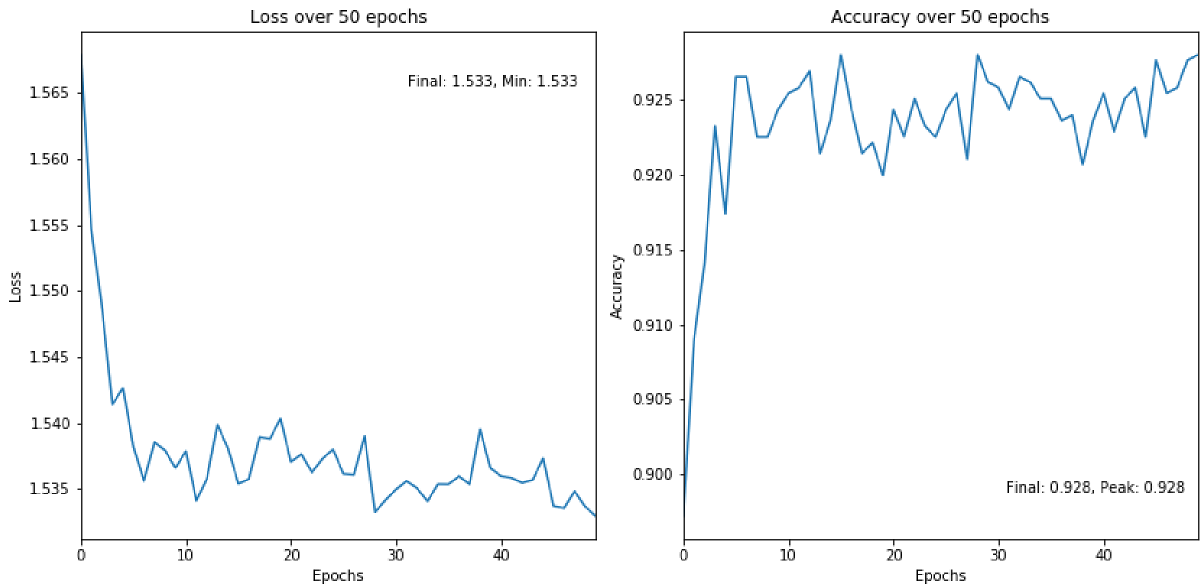


Table 2.3.1 Final training, validation and test accuracies

Set	Training	Validation	Test
Accuracy	**98.1%**	**92.8%**	**92.8%**

Observation:

- Dropout at p=0.9 seems to have small impact on the model by reducing the fluctuation of training accuracy curve. The model is less likely to overfitting in the sense that the testing accuracy increases very rapidly. The model is quicker to reach a point where it generalizes well against unseen data. We can see that a higher dropout probability should thus decrease training time in addition to helping to combat overfitting

2.2 P = 0.75

Figure 2.3.2.2.1 Training Loss & Accuracy over 50 epochs

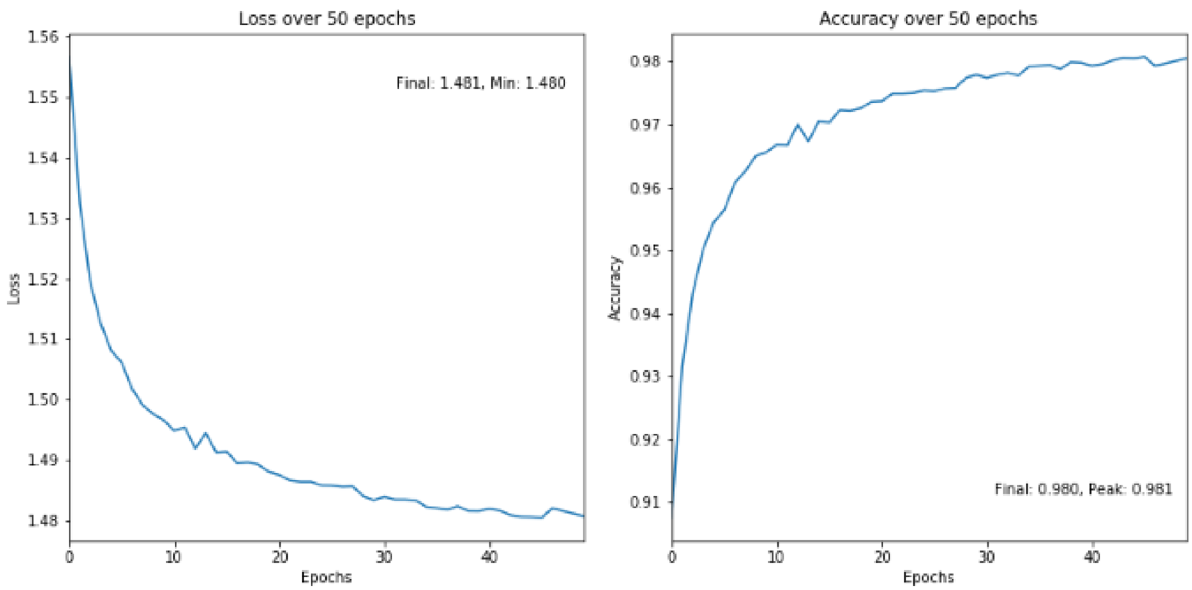


Figure 2.3.2.2.2 Validation Loss & Accuracy over 50 epochs

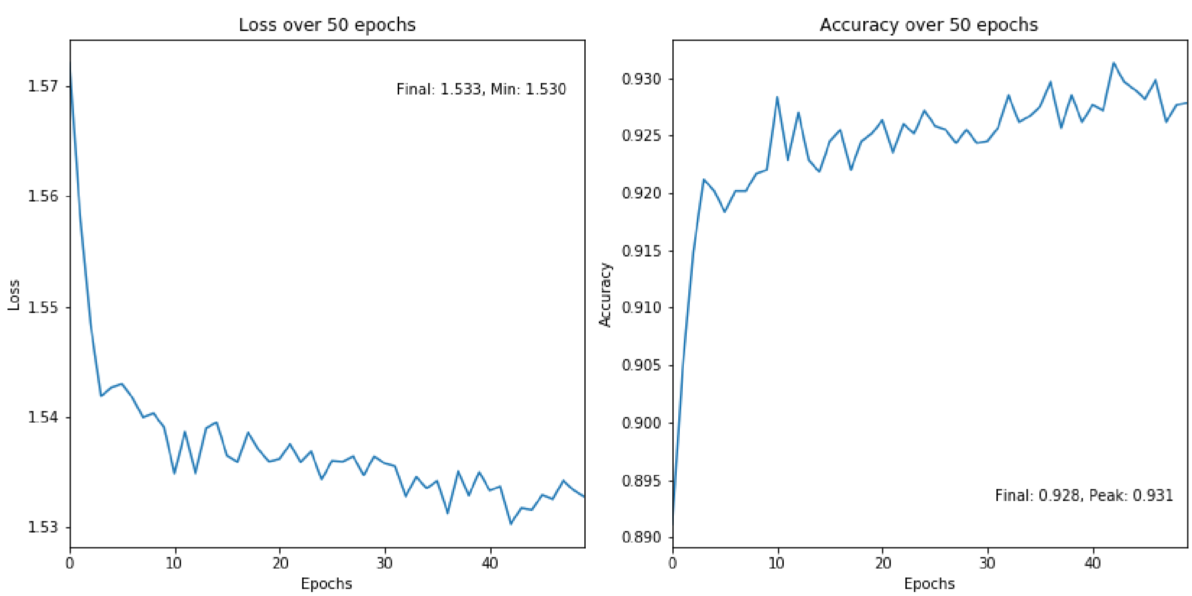


Figure 2.3.2.2.3 Test Loss & Accuracy over 50 epochs

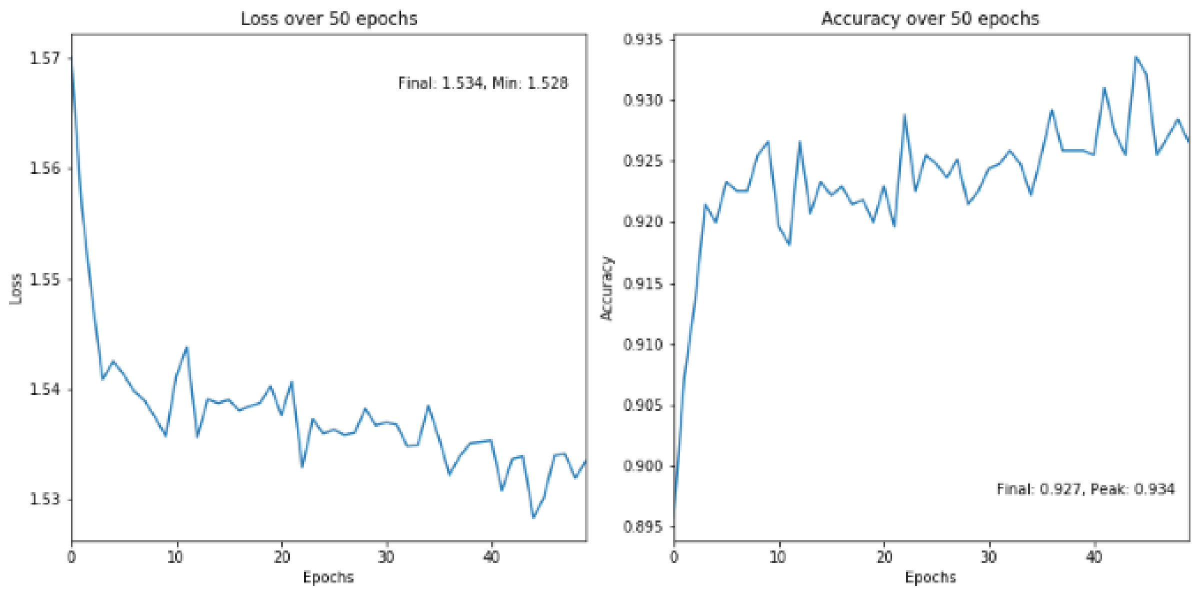


Table 2.3.2 Final training, validation and test accuracies

Set	Training	Validation	Test
Accuracy	**98.0%**	**92.8%**	**92.7%**

Observation:

- The final accuracies stays almost the same comparing to previous p(0.5), but the training accuracy curve is smoothier.
- The training accuracy curve starts to converge slower.

2.3. P = 0.5

Figure 2.3.2.3.1 Training Loss & Accuracy over 50 epochs

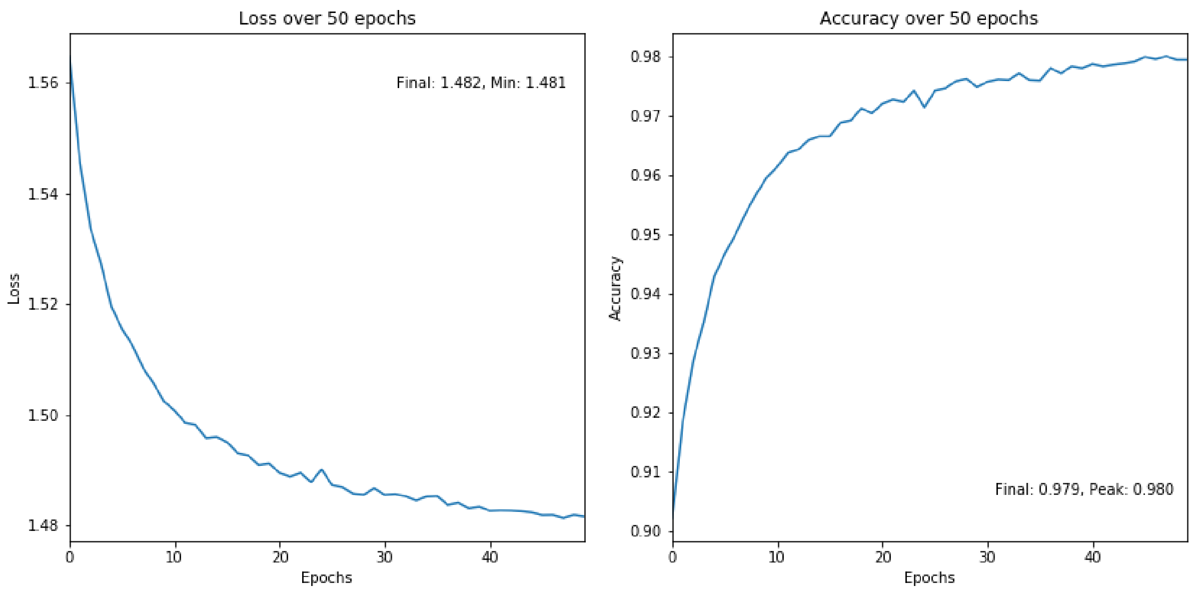


Figure 2.3.2.3.2 Validation Loss & Accuracy over 50 epochs

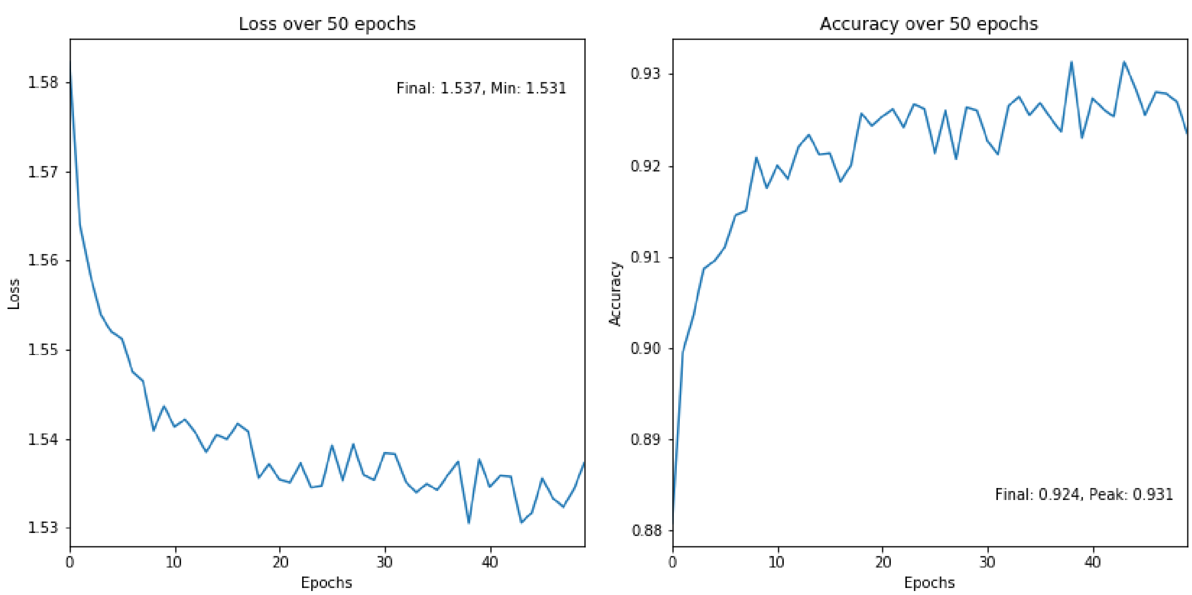


Figure 2.3.2.3.3 Test Loss & Accuracy over 50 epochs

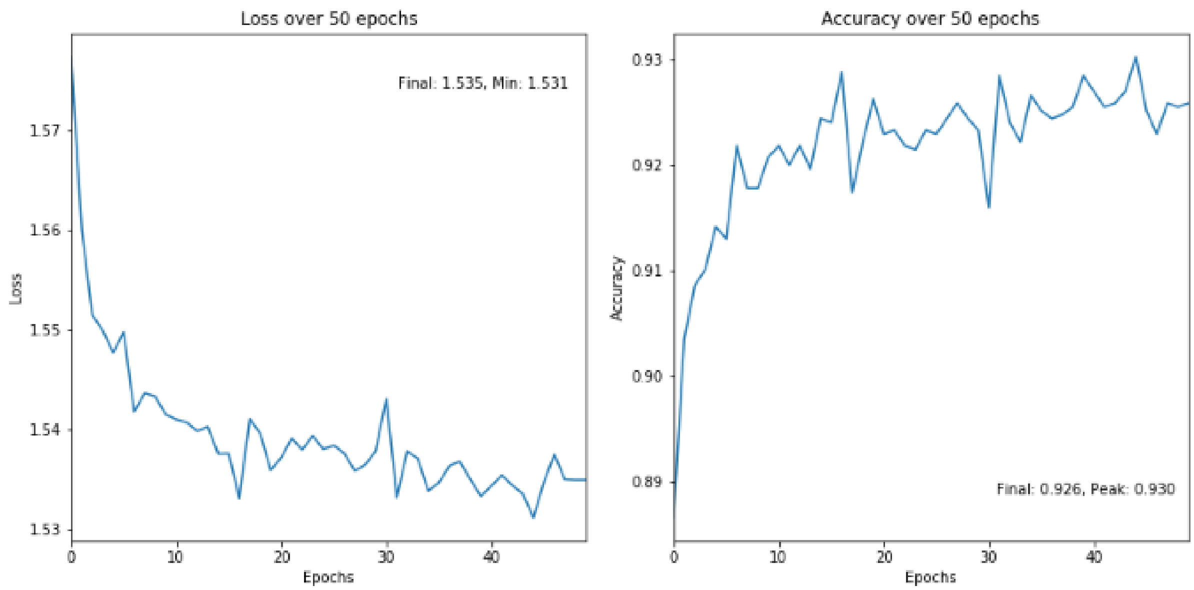


Table 2.3.2 Final training, validation and test accuracies

Set	Training	Validation	Test
Accuracy	**97.9%**	**92.4%**	**92.6%**

Observation:

- The final accuracies stays almost the same comparing to previous values of p, but the training accuracy curve is the smoothiest.
- After step 7 (fully connected layer with 784 output units), a dropout of p=0.5 is applied.
- We testing accuracy increases at the fastest rate in the higher probability dropout training. In this sense, the model generalizes earlier in training well.

Appendix

A.1 Part 1 code

```
import starter as starter
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

#%% Load Data and One-hot ecoding and parameters
trainData, validData, testData, trainTarget, validTarget, testTarget = starter.loadData()
#Reshape data to vectors
trainData = trainData.reshape([trainData.shape[0],784]) #train data matrix
testData = testData.reshape([testData.shape[0],784]) #test data matrix
validData = validData.reshape([validData.shape[0],784]) #validation data matrix

# Change from labels to one-hot encoding
trainTarget, validTarget, testTarget = starter.convertOneHot(trainTarget, validTarget, testTarget)



#%% 1.3 Create NN

#Xavier weight initialization parameters
K = 1000 #number of hidden nodes
epochs = 200

numClasses = testTarget.shape[1] #10
numFeature = trainData.shape[1] #1000

Wh = np.random.normal(0, (2/(numFeature + K)), (numFeature, K))
bh = np.random.normal(0, 0.01, (1, K)) #normal dist with small variance
Wo = np.random.normal(0, (2/(testTarget.shape[1] + K)), (K, numClasses)) #(testTarget.shape[1] + K) is number
of classes
bo = np.random.normal(0, 0.01, (1, (numClasses)))
Vh =  np.full((numFeature, K), 1e-5)
Vo =  np.full((K, numClasses), 1e-5)

alpha = 0.01#0.1 #Leaning Rate
gamma = 0.99
Vold_h = 0
Vold_o = 0
Vnew_h = 0
```

```
Vnew_o = 0

costs = []
accuracy = []

perfRecord = starter.Performance()

for epoch in range(epochs):
        #Forward Pass
        Z1 = np.matmul(trainData, Wh)  #sums of hidden layer, in rows
        X1 = starter.relu(Z1 + bh) #outputs of hidden layyer
        Z2 = np.matmul(X1, Wo) #sums of output layer, in rows
        Y = starter.softmax((Z2 + bo).T).T #final outputs

        #Backward pass
        delta2 = Y - trainTarget
        delta1 = (np.matmul(delta2,Wo.T))*np.heaviside(Z1,1)

        #Weight update
        Vnew_o = gamma*Vold_o + alpha*X1.T.dot(delta2)
        Vold_o = Vnew_o
        bo =  bo - alpha*(delta2).sum(axis = 0)
        Wh = Wh - Vnew_h

        Vnew_h = gamma*Vold_h + alpha*trainData.T.dot(delta1)
        Vold_h = Vnew_h
        Wh = Wh - Vnew_h
        bh = bh - alpha*(delta1).sum(axis=0)


        loss = starter.CE(trainTarget, Y)
        costs.append(loss)



        #Test score
        YoneHot, predict = starter.forwardPass(testData, Wh, Wo, bh, bo);
        targets = np.argmax(testTarget, axis = 1)
        testAccuracy = np.mean(predict == targets)
        testLoss = starter.CE(testTarget, YoneHot)

        #Train score
        YoneHot, predict = starter.forwardPass(trainData, Wh, Wo, bh, bo);
        targets = np.argmax(trainTarget, axis = 1)
        trainLoss = starter.CE(trainTarget, YoneHot)
        trainAccuracy = np.mean(predict == targets)
```

```
        #Valid Score
        YoneHot, predict = starter.forwardPass(validData, Wh, Wo, bh, bo);
        targets = np.argmax(validTarget, axis = 1)
        validAccuracy = np.mean(predict == targets)
        validLoss = starter.CE(validTarget, YoneHot)

        perfRecord.errorTrain.append(trainLoss)
        perfRecord.errorValid.append(validLoss)
        perfRecord.errorTest.append(testLoss)

        perfRecord.trainSetAcc.append(trainAccuracy)
        perfRecord.validSetAcc.append(validAccuracy)
        perfRecord.testSetAcc.append(testAccuracy)

        print('Epoch is', epoch,'Loss is ', loss, 'Test Acc is ', testAccuracy, 'Train Acc is', trainAccuracy)




#%% Part 1 plots

#Accuracy plots
plt.figure(figsize=(15,15))
plt.subplot(2, 1, 1)
plt.ylabel('Accuracy')
plt.xlabel('Iterations')
plt.title('Training, Valid and Test Set Accuracy of K = 1000')
plt.plot(perfRecord.trainSetAcc, 'r')
plt.plot(perfRecord.validSetAcc, 'g')
plt.plot(perfRecord.testSetAcc, 'b')
plt.legend(['Training Set, fin acc = %f' % perfRecord.trainSetAcc[-1]
        , 'Valid Set, fin acc = %f' % perfRecord.validSetAcc[-1]
        ,'Test Set, fin acc = %f' % perfRecord.testSetAcc[-1], ], loc = 4)

#Loss plots
plt.subplot(2, 1, 2)
plt.ylabel('Loss')
plt.xlabel('Iterations')
plt.title('Training, Valid and Test Set Loss of K = 1000')
plt.plot(perfRecord.errorTrain, 'r')
plt.plot(perfRecord.errorValid, 'g')
plt.plot(perfRecord.errorTest, 'b')
plt.legend(['Training Set, fin loss = %f' % perfRecord.errorTrain[-1]
        , 'Valid Set, fin loss = %f' % perfRecord.errorValid[-1]
        ,'Test Set, fin loss = %f' % perfRecord.errorTest[-1], ], loc = 1)

plt.savefig("Accuracy and loss plots 1.3.png")
```

```
#%% 1.4 Hypterparameter invest
epochs = 200
params = [100, 500, 2000]

for K in params:

            numClasses = testTarget.shape[1] #10
            numFeature = trainData.shape[1] #1000

            Wh = np.random.normal(0, (2/(numFeature + K)), (numFeature, K))
            bh = np.random.normal(0, 0.01, (1, K)) #normal dist with small variance
            Wo = np.random.normal(0, (2/(testTarget.shape[1] + K)), (K, numClasses)) #(testTarget.shape[1] + K) is
number of classes
            bo = np.random.normal(0, 0.01, (1, (numClasses)))
            Vh =  np.full((numFeature, K), 1e-5)
            Vo =  np.full((K, numClasses), 1e-5)

            alpha = 0.01#0.1 #Leaning Rate
            gamma = 0.92
            Vold_h = 0
            Vold_o = 0
            Vnew_h = 0
            Vnew_o = 0

            costs = []
            accuracy = []

            perfRecord = starter.Performance()

            for epoch in range(epochs):
            #Forward Pass
            Z1 = np.matmul(trainData, Wh)  #sums of hidden layer, in rows
            X1 = starter.relu(Z1 + bh) #outputs of hidden layyer
            Z2 = np.matmul(X1, Wo) #sums of output layer, in rows
            Y = starter.softmax((Z2 + bo).T).T #final outputs

            #Backward pass
            delta2 = Y - trainTarget
            delta1 = (np.matmul(delta2,Wo.T))*np.heaviside(Z1,1)

            #Weight update
            Vnew_o = gamma*Vold_o + alpha*X1.T.dot(delta2)
            Vold_o = Vnew_o
            bo =  bo - alpha*(delta2).sum(axis = 0)
            Wh = Wh - Vnew_h

            Vnew_h = gamma*Vold_h + alpha*trainData.T.dot(delta1)
```

```
        Vold_h = Vnew_h
        Wh = Wh - Vnew_h
        bh = bh - alpha*(delta1).sum(axis=0)


        loss = starter.CE(trainTarget, Y)
        costs.append(loss)



        #Test score
        YoneHot, predict = starter.forwardPass(testData, Wh, Wo, bh, bo);
        targets = np.argmax(testTarget, axis = 1)
        testAccuracy = np.mean(predict == targets)
        testLoss = starter.CE(testTarget, YoneHot)

        #Train score
        YoneHot, predict = starter.forwardPass(trainData, Wh, Wo, bh, bo);
        targets = np.argmax(trainTarget, axis = 1)
        trainLoss = starter.CE(trainTarget, YoneHot)
        trainAccuracy = np.mean(predict == targets)

        #Valid Score
        YoneHot, predict = starter.forwardPass(validData, Wh, Wo, bh, bo);
        targets = np.argmax(validTarget, axis = 1)
        validAccuracy = np.mean(predict == targets)
        validLoss = starter.CE(validTarget, YoneHot)

        perfRecord.errorTrain.append(trainLoss)
        perfRecord.errorValid.append(validLoss)
        perfRecord.errorTest.append(testLoss)

        perfRecord.trainSetAcc.append(trainAccuracy)
        perfRecord.validSetAcc.append(validAccuracy)
        perfRecord.testSetAcc.append(testAccuracy)

        print('Epoch is', epoch,'Loss is ', loss, 'Test Acc is ', testAccuracy, 'Train Acc is', trainAccuracy)

        if K == 100:
        perfRecord100 = perfRecord
        elif K == 500:
        perfRecord500 = perfRecord
        elif K == 2000:
        perfRecord2000 = perfRecord

#%%
```

```
#Accuracy plots
plt.figure(figsize=(15,15))
plt.ylabel('Accuracy')
plt.xlabel('Iterations')
plt.title('Training, Valid and Test Set Accuracy of different units')
plt.plot(perfRecord100.testSetAcc, 'r')
plt.plot(perfRecord500.testSetAcc, 'g')
plt.plot(perfRecord2000.testSetAcc, 'b')
plt.legend(['100 units, fin acc = %f' % perfRecord100.testSetAcc[-1]
            , '500 units, fin acc = %f' % perfRecord500.testSetAcc[-1]
            ,'2000 units, fin acc = %f' % perfRecord2000.testSetAcc[-1], ], loc = 4)

plt.savefig("Accuracy 1.4a.png")




#%% 1.4b Early stopping

#Xavier weight initialization parameters
K = 1000 #number of hidden nodes


numClasses = testTarget.shape[1] #10
numFeature = trainData.shape[1] #1000

Wh = np.random.normal(0, (2/(numFeature + K)), (numFeature, K))
bh = np.random.normal(0, 0.01, (1, K)) #normal dist with small variance
Wo = np.random.normal(0, (2/(testTarget.shape[1] + K)), (K, numClasses)) #(testTarget.shape[1] + K) is number
of classes
bo = np.random.normal(0, 0.01, (1, (numClasses)))
Vh =  np.full((numFeature, K), 1e-5)
Vo =  np.full((K, numClasses), 1e-5)

alpha = 0.01#0.1 #Leaning Rate
gamma = 0.99
Vold_h = 0
Vold_o = 0
Vnew_h = 0
Vnew_o = 0

costs = []
accuracy = []

perfRecord = starter.Performance()
```

```
epochs = 80
for epoch in range(epochs):
        #Forward Pass
        Z1 = np.matmul(trainData, Wh)  #sums of hidden layer, in rows
        X1 = starter.relu(Z1 + bh) #outputs of hidden layyer
        Z2 = np.matmul(X1, Wo) #sums of output layer, in rows
        Y = starter.softmax((Z2 + bo).T).T #final outputs

        #Backward pass
        delta2 = Y - trainTarget
        delta1 = (np.matmul(delta2,Wo.T))*np.heaviside(Z1,1)

        #Weight update
        Vnew_o = gamma*Vold_o + alpha*X1.T.dot(delta2)
        Vold_o = Vnew_o
        bo =  bo - alpha*(delta2).sum(axis = 0)
        Wh = Wh - Vnew_h

        Vnew_h = gamma*Vold_h + alpha*trainData.T.dot(delta1)
        Vold_h = Vnew_h
        Wh = Wh - Vnew_h
        bh = bh - alpha*(delta1).sum(axis=0)


        loss = starter.CE(trainTarget, Y)
        costs.append(loss)



        #Test score
        YoneHot, predict = starter.forwardPass(testData, Wh, Wo, bh, bo);
        targets = np.argmax(testTarget, axis = 1)
        testAccuracy = np.mean(predict == targets)
        testLoss = starter.CE(testTarget, YoneHot)

        #Train score
        YoneHot, predict = starter.forwardPass(trainData, Wh, Wo, bh, bo);
        targets = np.argmax(trainTarget, axis = 1)
        trainLoss = starter.CE(trainTarget, YoneHot)
        trainAccuracy = np.mean(predict == targets)

        #Valid Score
        YoneHot, predict = starter.forwardPass(validData, Wh, Wo, bh, bo);
        targets = np.argmax(validTarget, axis = 1)
        validAccuracy = np.mean(predict == targets)
        validLoss = starter.CE(validTarget, YoneHot)
```

```python
        perfRecord.errorTrain.append(trainLoss)
        perfRecord.errorValid.append(validLoss)
        perfRecord.errorTest.append(testLoss)

        perfRecord.trainSetAcc.append(trainAccuracy)
        perfRecord.validSetAcc.append(validAccuracy)
        perfRecord.testSetAcc.append(testAccuracy)

        print('Epoch is', epoch,'Loss is ', loss, 'Test Acc is ', testAccuracy, 'Train Acc is', trainAccuracy)




#%% Part 1 plots

#Accuracy plots
plt.figure(figsize=(15,15))
plt.ylabel('Accuracy')
plt.xlabel('Iterations')
plt.title('Training, Valid and Test Set Accuracy, Early stopping with Epochs = %f' % epochs)
plt.plot(perfRecord.trainSetAcc, 'r')
plt.plot(perfRecord.validSetAcc, 'g')
plt.plot(perfRecord.testSetAcc, 'b')
plt.legend(['Training Set, fin acc = %f' % perfRecord.trainSetAcc[-1]
        , 'Valid Set, fin acc = %f' % perfRecord.validSetAcc[-1]
        ,'Test Set, fin acc = %f' % perfRecord.testSetAcc[-1], ], loc = 4)


plt.savefig("Accuracy and loss plots 1.4b early stop.png")

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'

class Performance:
        def __init__(self): #needed to initialize arrays
        self.iterations = []
        self.errorTrain  = []
        self.errorValid = []
        self.errorTest = []
        self.trainSetAcc = []
        self.validSetAcc = []
        self.testSetAcc = []
```

```
# 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

# Implementation of a neural network using only Numpy - trained using gradient descent with momentum
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


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(x): #implements relu
        return x * (x > 0)

def softmax(x):
        #np.exp(x)/sum(np.exp(x)) #This implementation is not stable
https://deepnotes.io/softmax-crossentropy
        # Below is a more stable implementation, from same website
        exps = np.exp(x - np.max(x))
        return exps / np.sum(exps)
```

```
	#return np.exp(x)/sum(np.exp(x)) less stable

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

def forwardPass(trainData, Wh, Wo, bh, bo):
	Z1 = np.matmul(trainData, Wh)  #sums of hidden layer, in rows
	X1 = relu(Z1 + bh) #outputs of hidden layyer
	Z2 = np.matmul(X1, Wo) #sums of output layer, in rows
	Y =  softmax((Z2 + bo).T).T
	YoneHot = Y
	Yclass = np.argmax(Y, axis = 1) #1 hot
#==============================================================================
#	newY = np.zeros((Y.shape[0], 10))
#	for item in range(0, Y.shape[0]):
#	newY[item][Y[item]] = 1
#==============================================================================

	return YoneHot, Yclass #returns labels

def CE(target, prediction): #input should have one-hot targets and predictions as rows
	N = target.shape[0]
	output = np.sum(target*np.log(prediction + 1e-9), axis=1)
	output = -(1/N)*np.sum(output)
	return output

def gradCE(target, prediction): #return average gradCE

	#perform row-wise dot product
	N = target.shape[0]
	output = np.sum(target*np.reciprocal(prediction), axis = 0)
	output = -(1/N)*output #vector of averaged gradients according to the dataset

	return output
```

A.2.2 Model Training: Stochastic Gradient Descent

A.2.2.1 Basic Test: CNN + SGD

```
from cnn import ConvolutionalNeuralNetwork
from sgd import StochasticGradientDescent
from data import Data
from recorder import Recorder
from plotter import Plotter

import matplotlib.pyplot as plt

dt = Data()
rc = Recorder()
cnn = ConvolutionalNeuralNetwork()
sgd = StochasticGradientDescent(dt, rc, cnn)
plotter = Plotter(rc)

dt.load('notMNIST.npz')

#%% 2.1 + 2.2 Convolutional Neural Network +  Stochastic Gradient Descent

cnn.build_model(
    seed=421,#tf seed
    alpha=1e-4, #learning rate for ADAM optimizer
    with_dropout=False, p=0.9, #dropout
    with_regularizers=False, beta=0.01 #regulizer
)
sgd.build_trainer (
    epochs=50,
    batch_size=32
)

# plot loss & accuracy
plotter.plot_train_valid_test('img/basic')
```

A.2.3.1 L2 Normalization

A.2.3.1.1 Decay 0.01

```
#%% 2.3 L2 Decay 0.01

cnn.build_model(
    seed=421,#tf seed
    alpha=1e-4, #learning rate for ADAM optimizer
    with_dropout=False, p=0.9, #dropout
    with_regularizers=True, beta=0.01 #regulizer beta: weight decay
)
sgd.build_trainer (
    epochs=50,
    batch_size=32
)

# plot loss & accuracy
plotter.plot_train_valid_test('img/decay001')
```

A.2.3.1.2 Decay 0.1

```
#%% 2.3 L2 Decay 0.1

cnn.build_model(
    seed=421,#tf seed
    alpha=1e-4, #learning rate for ADAM optimizer
    with_dropout=False, p=0.9, #dropout
    with_regularizers=True, beta=0.1 #regulizer beta: weight decay
)
sgd.build_trainer (
    epochs=50,
    batch_size=32
)

# plot loss & accuracy
plotter.plot_train_valid_test('img/decay01')
```

A.2.3.1.2 Decay 0.5

```
#%% 2.3 L2 Decay 0.5

cnn.build_model(
    seed=421,#tf seed
    alpha=1e-4, #learning rate for ADAM optimizer
    with_dropout=False, p=0.9, #dropout
    with_regularizers=True, beta=0.5 #regulizer beta: weight decay
)
sgd.build_trainer (
    epochs=50,
    batch_size=32
)

# plot loss & accuracy
plotter.plot_train_valid_test('img/decay05')
```

A.2.3.2 Dropout

A.2.3.2.1 P 0.9

```
#%% 2.3.2 Dropout Decay 0.9

cnn.build_model(
    seed=421,#tf seed
    alpha=1e-4, #learning rate for ADAM optimizer
    with_dropout=True, p=0.9, #dropout
    with_regularizers=False, beta=0.5 #regulizer beta: weight decay
)
sgd.build_trainer (
    epochs=50,
    batch_size=32
)

# plot loss & accuracy
plotter.plot_train_valid_test('img/dropout09')
```

A.2.3.2.2 P 0.75

```
#%% 2.3.2 Dropout Decay 0.75

cnn.build_model(
    seed=421,#tf seed
    alpha=1e-4, #learning rate for ADAM optimizer
    with_dropout=True, p=0.75, #dropout
    with_regularizers=False, beta=0.5 #regulizer beta: weight decay
)
sgd.build_trainer (
    epochs=50,
    batch_size=32
)

# plot loss & accuracy
plotter.plot_train_valid_test('img/dropout075')
```

A.2.3.2.3 P 0.5

```
#%% 2.3.2 Dropout Decay 0.5

cnn.build_model(
    seed=421,#tf seed
    alpha=1e-4, #learning rate for ADAM optimizer
    with_dropout=True, p=0.5, #dropout
    with_regularizers=False, beta=0.5 #regulizer beta: weight decay
)
sgd.build_trainer (
    epochs=50,
    batch_size=32
)

# plot loss & accuracy
plotter.plot_train_valid_test('img/dropout05')
```