```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

data = input_data.read_data_sets("MNIST_data/", one_hot = True)

import matplotlib.pyplot as plt
import numpy as np
import time

# Function to update the plot for each epoch and error
def dynamic_plot(x, y, y_1, ax, ticks, title, colors = ['b']):
    ax.plot(x, y, 'b', label = 'Train Loss')
    ax.plot(x, y_1, 'r', label = 'Test Loss')
    if len(x) == 1:
        plt.legend()
        plt.title(title)
    plt.yticks(ticks)
    fig.canvas.draw()

# Network parameters
n_hidden_1 = 512  # first hidden layer
n_hidden_2 = 128  # second hidden layer
n_input    = 784  # Mnist data input (img shape: 28x28)
n_classes  = 10   # mnist total classes (0-9 : digits)

# Placehoders which will be used for feeding input.
# represent it as 2D tensor of floating point numbers
# None means, it can be dimension of any length

x = tf.placeholder(tf.float32, shape = [None, n_input])
y_ = tf.placeholder(tf.float32, shape = [None, n_classes])

# keep_prob: will be used in case of dropouts for testing
keep_prob = tf.placeholder(tf.float32)

# keep_prob_input : for dropout in training
keep_prob_input = tf.placeholder(tf.float32)

# weights initialization
# SGD: Xavier/Glorot Normal initialization.
# stddev = sqrt[2/fan_in + fan_out]

weight_sgd = {
     # 784x512
    'h1' : tf.Variable(tf.random_normal([n_input, n_hidden_1], stddev = 0.039, mean = 0.0)
     # 512x128
    'h2' : tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2], stddev = 0.055, mean = 0
     # 128x10
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes], stddev = 0.120, mean = 0.
}
```

```
biases ={
       # 512x1
    'b1' : tf.Variable(tf.random_normal([n_hidden_1])),
       # 128x1
    'b2' : tf.Variable(tf.random_normal([n_hidden_2])),
       # 10x1
    'out': tf.Variable(tf.random_normal([n_classes]))
}


# Parameters
training_epochs = 15
learning_rate = 0.001
batch_size = 100
display_step = 1
```

Model 1: input(784) - sigmoid(512) - sigmoid(128) - softmax(output

```
# https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/
# multi_layer_perceptron_mnist.html
# Create Model
def multilayer_perceptron(x, weights, biases):
    print('x:',x.get_shape(), 'w[h1]:',weights['h1'].get_shape(),
          'b[h1]:',biases["b1"].get_shape())

    # Hidden layer1 with sigmoid activation
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.sigmoid(layer_1)
    print('layer_1:',layer_1.get_shape(), 'w[h2]:',weights['h2'].get_shape(),
          'b[h2]:',biases["b2"].get_shape())

    # Hidden layer2 with sigmoid ativation
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.sigmoid(layer_2)
    print('layer_2:',layer_2.get_shape(), 'w[out]:',weights['out'].get_shape(),
          'b[out]:',biases["out"].get_shape())

    # output layer with sigmoid activation
    output_layer = tf.add(tf.matmul(layer_2, weights['out']), biases['out'])
    output_layer = tf.nn.sigmoid(output_layer)
    print('output_layer:',output_layer.get_shape())

    return output_layer
```

Model 1 + Adam Optimizer

```
# Since sigmoid activation units are used here, so we will use the weights_sgd and biases
# above
y_sgd = multilayer_perceptron(x, weight_sgd, biases)

# Cost_function
```

```
cost_sgd = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y_sgd, labels=y_)

# optimizer to minimize the cost
optimizer_adam = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost_sgd)
optimizer_sgdc = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(c


# Starting the Session
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    fig, ax = plt.subplots(1,1)
    ax.set_xlabel('epoch')
    ax.set_ylabel("Softmax Cross Entropy Loss")
    xs, y_trs, y_tes = [],[],[]
    for epoch in range(training_epochs):
        train_avg_cost = 0.
        test_avg_cost = 0.
        total_batch = int(data.train.num_examples/batch_size)

        for i in range(total_batch):
            batch_xs, batch_ys = data.train.next_batch(batch_size)

            feed_dict = {x:batch_xs, y_:batch_ys}
            _,c,w = sess.run([optimizer_adam, cost_sgd, weight_sgd], feed_dict = feed_dict
            train_avg_cost += c / total_batch

            c = sess.run(cost_sgd, feed_dict = {x:data.test.images, y_:data.test.labels})
            test_avg_cost += c / batch_size

        xs.append(epoch)
        y_trs.append(train_avg_cost)
        y_tes.append(test_avg_cost)
        dynamic_plot(xs, y_trs, y_tes, ax, np.arange(1.3, 1.8, step = 0.04), "input-sigmoi

        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch + 1), 'train_cost ={:.9f}'.format(train_avg_co
                 'test_cost ={:.9f}'.format(test_avg_cost))
    dynamic_plot(xs, y_trs, y_tes, ax, np.arange(1.3, 1.8, step = 0.04), "input-sigmoid(51


    # Calculating the final Accuracy on test set
    correct_prediction = tf.equal(tf.argmax(y_sgd, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    print("Accuracy: ", accuracy.eval({x:data.test.images, y_: data.test.labels}))
```

```
# Plotting weight distribution at the end of training
import seaborn as sns
h1_w = w['h1'].flatten().reshape(-1,1)
h2_w = w['h2'].flatten().reshape(-1,1)
out_w = w['out'].flatten().reshape(-1,1)

fig = plt.figure()
plt.subplot(1,3,1)
plt.title("Weight matrix")
ax = sns.violinplot(y = h1_w, color='b')
plt.xlabel("hidden layer 1")

plt.subplot(1,3,2)
plt.title("Weight matrix")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel("hidden layer 2")

plt.subplot(1,3,3)
plt.title("Weight matrix")
ax = sns.violinplot(y=out_w, color='y')
plt.xlabel("output layer")
plt.show()
```

Model 1 + Grandient Descent optimizer

```
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    fig, ax = plt.subplots(1,1)
    ax.set_xlabel("epoch")
    ax.set_ylabel("Softmax Cross Entropy Loss")
    xs, y_trs, y_tes = [], [], []

    for epoch in range(training_epochs):
        train_avg_cost = 0.
        test_avg_cost = 0.
        total_batch = int(data.train.num_examples / batch_size)

        for i in range(total_batch):
            batch_xs, batch_ys = data.train.next_batch(batch_size)

            feed_dict = {x:batch_xs, y_:batch_ys}
            _,c,w = sess.run([optimizer_sgdc, cost_sgd, weight_sgd], feed_dict=feed_dict)
            train_avg_cost += c / total_batch

            c = sess.run(cost_sgd, feed_dict={x:data.test.images, y_:data.test.labels})
            test_avg_cost += c / total_batch

        xs.append(epoch)
        y_trs.append(train_avg_cost)
        y_tes.append(test_avg_cost)
        dynamic_plot(xs,y_trs,y_tes,ax,np.arange(2,2.6, step = 0.05), "input-sigmoid(512)-

        if epoch % display_step == 0:
            print("Epoch: ",'%04d' % (epoch +1), 'train_cost{:0.9f}'.format(train_avg_cost
    dynamic_plot(xs,y_trs,y_tes,ax,np.arange(2,2.6, step = 0.05), "input-sigmoid(512)-sigm

    # Calucating the final accuracy
    correct_prediction = tf.equal(tf.arg_max(y_sgd, 1), tf.arg_max(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    print("Accuracy: ", accuracy.eval({x:data.test.images, y_:data.test.labels}))
```

```
# Plotting the weights distribution after the end of training
h1_w = w['h1'].flatten().reshape(-1,1)
h2_w = w['h2'].flatten().reshape(-1,1)
out_w = w['out'].flatten().reshape(-1,1)

fig = plt.figure()
plt.subplot(1,3,1)
plt.title("Weight matrix")
ax = sns.violinplot(y=h1_w, color='b')
plt.xlabel("hidden layer 1")

plt.subplot(1,3,2)
plt.title("Weight matrix")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel("hidden layer 2")

plt.subplot(1,3,3)
plt.title("Weight matrix")
ax = sns.violinplot(y=out_w, color='y')
plt.xlabel("output layer")
plt.show()
```

Model 2: input(784) - reul(512) - relu(128) - sigmoid(output 10)

```
def multilayer_perceptron_relu(x, weights, biases):
    print("x: ",x.get_shape(), "w[h1]: ",weights['h1'].get_shape(),
          "b[h1]: ",biases['b1'].get_shape())
    # first hidden layer
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    print("layer_1: ", layer_1.get_shape(), "w[h2]: ",weights['h2'].get_shape(),
         "b[h2]: ",biases['b2'].get_shape())

    # second hidden layer
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)
    print("layer_2: ",layer_2.get_shape(), "w[out]: ",weights['out'].get_shape(),
         "b[out]: ",biases['out'].get_shape())

    # output layer
    output_layer = tf.add(tf.matmul(layer_2, weights['out']), biases['out'])
    output_layer = tf.nn.sigmoid(output_layer)
    print("Output_layer: ", output_layer.get_shape())

    return output_layer
```

Input-ReLu(512)-ReLu(128)-sigmoid(output) - AdamOptimizer

```
## For relu activation
## He initialization
## std = sqrt(2/fan_in + 1)
## +1 is for Zero division error
weights_relu = {
     'h1' : tf.Variable(tf.random_normal([n_input, n_hidden_1], stddev=0.062, mean=0.0)),
     'h2' : tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2], stddev=0.125, mean=0.0)
     'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes], stddev=0.120, mean=0.0))
 }


biases ={
    'h1' : tf.Variable(tf.random_normal([n_hidden_1])),
```

```
    'b1' : tf.Variable(tf.random_normal([n_hidden_1])),
    'b2' : tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}


# for relu activation
y_relu = multilayer_perceptron_relu(x, weights_relu, biases)

# cost function
cost_relu = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y_relu, labels=y_

# optimizers
optimizer_relu_adam = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost_re
optimizer_relu_sgdc = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minim


# Starting the session
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)

    fig, ax = plt.subplots(1,1)
    ax.set_xlabel("epoch")
    ax.set_ylabel("Softmax Cross Entropy loss")
    xs, y_trs, y_tes = [],[],[]

    for epoch in range(training_epochs):
        train_avg_cost = 0.
        test_avg_cost = 0.
        total_batch = int(data.train.num_examples / batch_size)

        for i in range(total_batch):
            batch_xs, batch_ys = data.train.next_batch(batch_size)

            feed_dict = {x:batch_xs, y_:batch_ys}
            _,c,w = sess.run([optimizer_relu_adam, cost_relu, weights_relu], feed_dict=fee
            train_avg_cost += c / total_batch

            c = sess.run(cost_relu, feed_dict={x:data.test.images, y_:data.test.labels})
            test_avg_cost += c / total_batch

        xs.append(epoch)
        y_trs.append(train_avg_cost)
        y_tes.append(test_avg_cost)
        dynamic_plot(xs,y_trs,y_tes,ax,np.arange(1.3,1.8,step=0.04),"Input-ReLu(512)-ReLu(


        if epoch % display_step == 0:
            print("Epoch: ",'%04d' % (epoch + 1), 'train_cost={:0.9f}'.format(train_avg_co
                'test_cost={:0.9f}'.format(test_avg_cost))
    # Plotting final results
    dynamic_plot(xs,y_trs,y_tes,ax,np.arange(1.3,1.8,step=0.04),"Input-ReLu(512)-ReLu(128)

    # Calculating the final accuracy on test data
    correct_prediction = tf.equal(tf.argmax(y_relu, 1), tf.argmax(y_, 1))
```

```
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    print("Accuracy: ", accuracy.eval({x:data.test.images, y_:data.test.labels}))
```

```
# Plotting the weights distribution after the end of training
h1_w = w['h1'].flatten().reshape(-1,1)
h2_w = w['h2'].flatten().reshape(-1,1)
out_w = w['out'].flatten().reshape(-1,1)

fig = plt.figure()
plt.subplot(1,3,1)
plt.title("Weight matrix")
sns.violinplot(y=h1_w, color = 'b')
plt.xlabel('hidden layer 1')

plt.subplot(1,3,2)
plt.title("Weight matrix")
sns.violinplot(y=h2_w, color = 'r')
plt.xlabel("hidden layer 2")

plt.subplot(1,3,3)
plt.title("Weight matrix")
```

```
sns.violinplot(y=out_w, color='y')
plt.xlabel("output layer")

plt.show()
```

Model 3 + Gradient Descent Optimizer

```
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)

    fig,ax = plt.subplots(1,1)
    ax.set_xlabel("epoch")
    ax.set_ylabel("Softmax Cross Entropy loss")

    xs,y_trs,y_tes = [],[],[]
    for epoch in range(training_epochs):
        train_avg_cost = 0.
        test_avg_cost = 0.
        total_batch = int(data.train.num_examples / batch_size)

        for i in range(total_batch):
            batch_xs, batch_ys = data.train.next_batch(batch_size)
            feed_dict = {x:batch_xs, y_: batch_ys}
            _,c,w = sess.run([optimizer_relu_sgdc, cost_relu, weights_relu], feed_dict=fee
            train_avg_cost += c / total_batch

            c = sess.run(cost_relu, feed_dict={x:data.test.images, y_:data.test.labels})
            test_avg_cost += c / total_batch


        xs.append(epoch)
        y_trs.append(train_avg_cost)
        y_tes.append(test_avg_cost)
        dynamic_plot(xs,y_trs,y_tes,ax,np.arange(1.5,2.4,step=0.05),"input-ReLu(512)-ReLu(

        if epoch % display_step == 0:
            print("Epoch:",'%04d' % (epoch +1), "train_cost={:0.9f}".format(train_avg_cost
```

```
                "test_cost={:0.9f}".format(test_avg_cost))

    # Plotting the final results
    dynamic_plot(xs,y_trs,y_tes,ax,np.arange(1.5,2.4,step=0.05),"input-ReLu(512)-ReLu(128)

    # calculating the final accuracy on test set
    correct_prediction = tf.equal(tf.argmax(y_relu, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    print("Accuracy: ", accuracy.eval({x:data.test.images, y_:data.test.labels}))
```

```
# Plotting weights distribution after the end of training

h1_w = w['h1'].flatten().reshape(-1,1)
h2_w = w['h2'].flatten().reshape(-1,1)
out_w = w['out'].flatten().reshape(-1,1)

fig = plt.figure()
plt.subplot(1,3,1)
plt.title("Weight matrix")
sns.violinplot(y=h1_w, color = 'b')
plt.xlabel("hidden layer 1")

plt.subplot(1,3,2)
plt.title("Weight matrix")
sns.violinplot(y=h2_w, color = 'r')
```

```
sns.violinplot(y=h2_w, color = 'r')
plt.xlabel("hidden layer 2")

plt.subplot(1,3,3)
plt.title("Weight matrix")
sns.violinplot(y=out_w, color = 'y')
plt.xlabel("output layer")
plt.show()
```

Batch Normalization

```
# Input - Sigmoid(BatchNormalization(512)) - Sigmoid(BatchNormalization(128))- Sigmoid(out

epsilon = 1e-3  # to ignore the zero division error
def multilayer_perceptron_batch(x,weights,biases):
  print("x:",x.get_shape(), "w[h1]:",weights['h1'].get_shape(),"b[h1]:",biases['b1'].get_s

  ########################################################################################
  # hidden layer 1 with sigmoid activation and batch normalization
  layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])

  # calculating the mean and variance of x
  batch_mean_1, batch_var_1 = tf.nn.moments(x=layer_1, axes = [0])

  scale_1 = tf.Variable(tf.ones([n_hidden_1]))
  beta_1 = tf.Variable(tf.zeros([n_hidden_1]))

  layer_1 = tf.nn.batch_normalization(layer_1, batch_mean_1, batch_var_1, beta_1, scale_1,
  layer_1 = tf.nn.sigmoid(layer_1)

  print("layer_1:",layer_1.get_shape(), "w[h2]:",weights['h2'].get_shape(),
        "b[h2]:",biases['b2'].get_shape())

  ########################################################################################
  # hidden layer 2 with sigmoid activation and batch normalization
  layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])

  # calculating mean and variance
```

```
    batch_mean_2, batch_var_2 = tf.nn.moments(x=layer_2, axes=[0])

    scale_2 = tf.Variable(tf.ones([n_hidden_2]))
    beta_2  = tf.Variable(tf.zeros([n_hidden_2]))

    layer_2 = tf.nn.batch_normalization(layer_2,batch_mean_2,batch_var_2,beta_2,scale_2,epsi
    layer_2 = tf.nn.sigmoid(layer_2)

    print("layer_2:",layer_2.get_shape(),'w[out]:',weights['out'].get_shape(),'b[out]:',bias

    ##########################################################################################
    # output layer with sigmoid activation
    output_layer = tf.add(tf.matmul(layer_2, weights['out']), biases['out'])
    output_layer = tf.nn.sigmoid(output_layer)

    print("output_layer:",output_layer.get_shape())

    return output_layer
```

Model + adam optimizer

```
y_batch =  multilayer_perceptron_batch(x,weight_sgd,biases)

# cost function
cost_batch = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y_batch, labels

# optimizer
optimizer_batch_adam = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost_b
optimizer_batch_sgdc = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).mini
```

```
# Session
with tf.Session() as sess:
  init = tf.global_variables_initializer()
  sess.run(init)

  fig,ax = plt.subplots(1,1)
  ax.set_xlabel("epoch")
  ax.set_ylabel("Softmax Cross Entropy loss")

  xs,y_trs,y_tes = [],[],[]
  for epoch in range(training_epochs):
    train_avg_cost = 0.
    test_avg_cost = 0.
    total_batch = int(data.train.num_examples / batch_size)

    for i in range(total_batch):
      batch_xs, batch_ys = data.train.next_batch(batch_size)

      feed_dict = {x:batch_xs, y_:batch_ys}
```

```
      _,c,w = sess.run([optimizer_batch_adam,cost_batch,weight_sgd], feed_dict=feed_dict)
      train_avg_cost += c / total_batch

      feed_dict = {x:data.test.images, y_:data.test.labels}
      c = sess.run(cost_batch, feed_dict=feed_dict)
      test_avg_cost += c / total_batch

    xs.append(epoch)
    y_trs.append(train_avg_cost)
    y_tes.append(test_avg_cost)
    dynamic_plot(xs,y_trs,y_tes,ax,np.arange(1.3, 1.8, step=0.04), "input-Sigmoid(BN(512))

    if epoch % display_step == 0:
      print("Epoch:",'%04d' % (epoch + 1), "train_cost={:0.9f}".format(train_avg_cost),
            "test_cost={:0.9f}".format(test_avg_cost))

  # Plotting the final results
  dynamic_plot(xs,y_trs,y_tes,ax,np.arange(1.3, 1.8, step=0.04), "input-Sigmoid(BN(512))-S

  # Calculating the accurancy on test set
  correct_prediction = tf.equal(tf.argmax(y_batch, 1), tf.argmax(y_, 1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
  print("Accuracy:", accuracy.eval({x:data.test.images, y_:data.test.labels}))
```

```
h1_w = w['h1'].flatten().reshape(-1,1)
h2_w = w['h2'].flatten().reshape(-1,1)
out_w = w['out'].flatten().reshape(-1,1)


fig = plt.figure()
plt.subplot(1, 3, 1)
plt.title("Weight matrix")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Weight matrix ")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Weight matrix ")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```

Model 3 + GradientDescentOptimizer

```
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    fig,ax = plt.subplots(1,1)
    ax.set_xlabel('epoch') ; ax.set_ylabel('Soft Max Cross Entropy loss')
    xs, ytrs, ytes = [], [], []
    for epoch in range(training_epochs):
        train_avg_cost = 0.
        test_avg_cost = 0.
        total_batch = int(data.train.num_examples/batch_size)

        # Loop over all batches
```

```
        for i in range(total_batch):
            batch_xs, batch_ys = data.train.next_batch(batch_size)

            # here we use GradientDescentOptimizer
            _, c, w = sess.run([optimizer_batch_sgdc, cost_batch, weight_sgd], feed_dict={
            train_avg_cost += c / total_batch
            c = sess.run(cost_batch, feed_dict={x: data.test.images, y_: data.test.labels}
            test_avg_cost += c / total_batch

        xs.append(epoch)
        ytrs.append(train_avg_cost)
        ytes.append(test_avg_cost)
        dynamic_plot(xs, ytrs, ytes, ax,np.arange(1.5, 2.4, step=0.05), "input-Sigmoid(BN(

        if epoch%display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "train cost={:.9f}".format(train_avg_cost)

    # plot final results
    dynamic_plot(xs, ytrs, ytes, ax, np.arange(1.5, 2.4, step=0.05), "input-Sigmoid(BN(512

    # we are calculating the final accuracy on the test data
    correct_prediction = tf.equal(tf.argmax(y_batch,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    print("Accuracy:", accuracy.eval({x: data.test.images, y_: data.test.labels}))
```

```
h1_w = w['h1'].flatten().reshape(-1,1)
h2_w = w['h2'].flatten().reshape(-1,1)
out_w = w['out'].flatten().reshape(-1,1)


fig = plt.figure()
plt.subplot(1, 3, 1)
plt.title("Weight matrix")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Weight matrix ")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Weight matrix ")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```

Model 4: Input - ReLu(512) - Dropout - ReLu(128)- Dropout -Sigmoid(output)

```
# Createing the model
def multilayer_perceptron_dropout(x, weights, biases):
  print("x:",x.get_shape(), "w[h1]: ",weights['h1'].get_shape(), "b['h1']: ",biases['b1'].

  # first hidden layer with relu activation
  layer_1 = tf.add(tf.matmul(x,weights['h1']), biases['b1'])
  layer_1 = tf.nn.relu(layer_1)

  # Adding the dropout layer
  layer_1_drop = tf.nn.dropout(layer_1, keep_prob)

  print("layer_1_drop: ",layer_1_drop.get_shape(), "w[h2]: ",weights['h2'].get_shape(),
        "b[h2]: ",biases['b2'].get_shape())
```

```
  # Second dropout layer with relu activation
  layer_2 = tf.add(tf.matmul(layer_1_drop, weights['h2']), biases['b2'])
  layer_2 = tf.nn.relu(layer_2)

  # dropout layer
  layer_2_drop = tf.nn.dropout(layer_2, keep_prob)

  print("layer_2_drop: ",layer_2_drop.get_shape(), "w[out]: ",weights['out'].get_shape(),
        "b[out]: ",biases['out'].get_shape())

  # Output layer
  output_layer = tf.add(tf.matmul(layer_2_drop, weights['out']), biases['out'])
  output_layer = tf.nn.sigmoid(output_layer)
  print("output_layer: ", output_layer.get_shape())

  return output_layer
```

Model + Adamoptimizer

```
y_drop = multilayer_perceptron_dropout(x, weights_relu, biases)

# cost function
cost_drop = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y_drop, labels=y_

# Optimizers
optimizer_drop_adam = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost_dr
optimizer_drop_sgdc = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minim


# Session
with tf.Session() as sess:
  init = tf.global_variables_initializer()
  sess.run(init)

  fig,ax = plt.subplots(1,1)
  ax.set_xlabel("epoch")
  ax.set_ylabel("Softmax Cross Entropy Loss")

  xs,y_trs,y_tes = [],[],[]
  for epoch in range(training_epochs):
    train_avg_cost = 0.
    test_avg_cost = 0.
    total_batch = int(data.train.num_examples / batch_size)

    for i in range(total_batch):
      batch_xs, batch_ys = data.train.next_batch(batch_size)

      feed_dict = {x:batch_xs, y_:batch_ys, keep_prob:0.5}
      _,c,w = sess.run([optimizer_drop_adam,cost_drop,weights_relu], feed_dict = feed_dict
      train_avg_cost += c / total_batch

      c = sess.run(cost_drop, feed_dict={x:data.test.images, y_:data.test.labels, keep_pro
      test_avg_cost += c / total_batch
```

```
    xs.append(epoch)
    y_trs.append(train_avg_cost)
    y_tes.append(test_avg_cost)
    dynamic_plot(xs,y_trs,y_tes,ax,np.arange(1,1.8,step=0.05),"Input - ReLu(512) - Dropout


    if epoch % display_step == 0:
      print("Epoch: ","%04d" % (epoch+1), "train_cost = {:0.9f}".format(train_avg_cost),
            "test_cost = {:0.9f}".format(test_avg_cost))

  # Plotting the final plot
  dynamic_plot(xs,y_trs,y_tes,ax,np.arange(1,1.8,step=0.05),"Input - ReLu(512) - Dropout -

  # Calculating the accuracy on test set
  correct_prediction = tf.equal(tf.argmax(y_drop, 1), tf.argmax(y_, 1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

  print("Accuracy: ", accuracy.eval({x:data.test.images, y_: data.test.labels, keep_prob:1
```

```
# Plotting the weight distribution after the end of trainng
import seaborn as sns
h1_w = w['h1'].flatten().reshape(-1,1)
h2_w = w['h2'].flatten().reshape(-1,1)
out_w = w['out'].flatten().reshape(-1,1)


fig = plt.figure()
plt.subplot(1, 3, 1)
plt.title("Weight matrix")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Weight matrix ")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Weight matrix ")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```