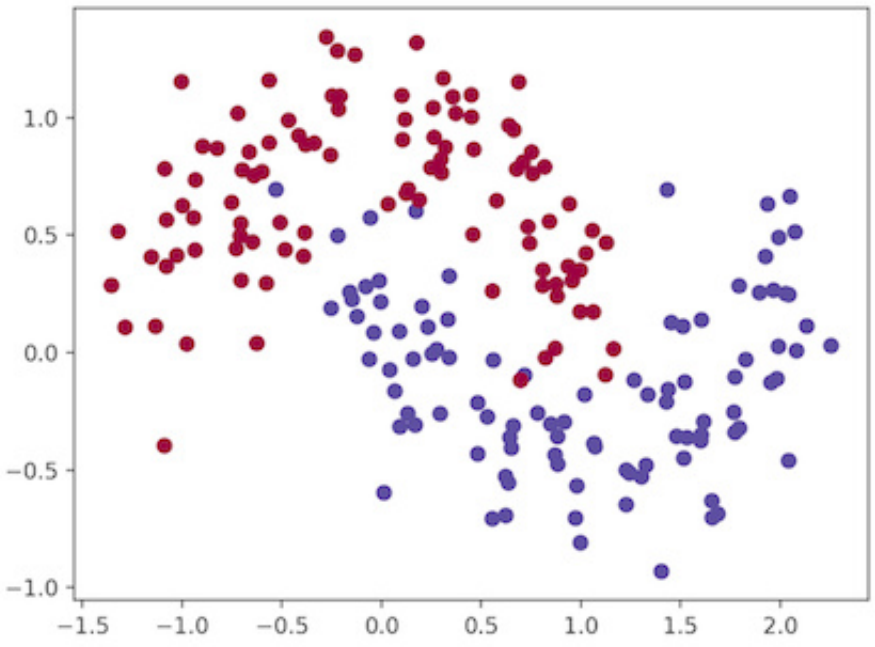
Backpropagation in a Simple Neural Network

Datasets

Run the following code and get the scatter plot.

```
# generate and visualize Make-Moons dataset

X, y = generate_data()

plt.scatter(X[:, 0], X[:, 1], s=40, c=y, cmap=plt.cm.Spectral)
```

The scatter plot is



Activation Function

Implement actFun(self, z, type)

Define the activation function based on their defination.

```
def actFun(self, z, type):
        '''
        actFun computes the activation functions
        :param z: net input
        :param type: Tanh, Sigmoid, or ReLU
        :return: activations
        '''

        # YOU IMPLMENT YOUR actFun HERE
        if type == 'tanh':
            act = np.tanh(z)
        if type == 'sigmoid':
            act = 1. / (1 + np.exp(-z))
        if type == 'relu':
            act = z * (z > 0)

        return act
```

Derive the derivatives of Tanh, Sigmoid, and ReLU

Tanh function

We know that $tanh = \frac{1-e^{-2x}}{1+e^{-2x}}$, and now we will take derivative of this function.

$$\frac{dtanh(x)}{dx} = \frac{2e^{-2x}}{1+e^{-2x}} + \frac{(1-e^{-2x})*2e^{-2x}}{(1+e^{-2x})^2} = \frac{4e^{-2x}}{(1+e^{-2x})^2} = 1 - tanh^2(x)$$

Sigmoid function

$$\frac{dsig(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^{-2}} = sig(x)(1 - sig(x))$$

ReLU function

obviously, the derivative of $\mathbf{x}$ is, 1 for $x_i > 0$ and 0 for $x_i = 0$.

Implement function diff_actFun(self, z, type)

Use the derivative result in 2 to define the function.

```
    def diff_actFun(self, z, type):
        '''
        diff_actFun computes the derivatives of the activation functions wrt the net
input
        :param z: net input
        :param type: Tanh, Sigmoid, or ReLU
        :return: the derivatives of the activation functions wrt the net input
        '''

        # YOU IMPLEMENT YOUR diff_actFun HERE
        if type == 'tanh':
            diff = 1 - np.tanh(z)**2
        if type == 'sigmoid':
            sig = 1 / (1 + np.exp(-z))
            diff = sig * (1 - sig)
        if type == 'relu':
            diff = 1 * (z > 0)

        return diff
```

Build the Neural Network

Implement the function feedforward(self, X, actFun).

Feedforward the model by feeding the model parameters.

```
def feedforward(self, X, actFun):
        '''
        feedforward builds a 3-layer neural network and computes the two probabilitie
s,
        one for class 0 and one for class 1
        :param X: input data
        :param actFun: activation function
        :return:
        '''

        # YOU IMPLEMENT YOUR feedforward HERE

        self.z1 = self.W1 * X + self.b1
        self.a1 = actFun(self.z1)
        self.z2 = self.W2 * self.a1 + self.b2
        exp_scores = np.exp(self.z2)
        self.probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

        return None
```

Fill in the function calculate loss(self, X, y).

Since y is one-hot vector, we could just choose the label column to get the mulplication.

```
 def calculate_loss(self, X, y):
        '''
        calculate_loss computes the loss for prediction
        :param X: input data
        :param y: given labels
        :return: the loss for prediction
        '''
        num_examples = len(X)
        self.feedforward(X, lambda x: self.actFun(x, type=self.actFun_type))
        # Calculating the loss

        # YOU IMPLEMENT YOUR CALCULATION OF THE LOSS HERE
        data_loss = -np.sum(np.log(self.probs)[np.arange(num_examples),
                            y])/num_examples

        # Add regulatization term to loss (optional)
        data_loss += self.reg_lambda / 2 * (np.sum(np.square(self.W1)) + np.sum(np.sq
uare(self.W2)))
        return (1. / num_examples) * data_loss
```

Backward Pass - Backpropagation

Derive the gradients

Sigmoid

1 First, consider the gradient of W_2.

The loss function is $L = -\frac{1}{N}\sum_{n=1}^{N} y_n log\hat{y_n}$, the inner dot product of y_n and $\hat{y_n}$, where y_n is the one-hot vector of n_{th} sample and $\hat{y_n}$ is the output of n_{th} sample. For computation ease, we only consider one sample, $l_1 = -ylog\hat{y}$. Now we have $\frac{\partial l_1}{\partial w_1^{ij}} = \sum_{k,l} \frac{\partial l_1}{\partial y^k} \frac{\partial y^k}{\partial z_2^l} \frac{\partial z_2^l}{\partial w_1^{ij}}$, where ∂z_2^l represents the l_{th} element of z_2.

Obviously the $\frac{\partial l_1}{\partial y^k} = \frac{y^k}{\hat{y}^k}$.

Next, the gradient of softmax function: $\frac{\partial y^k}{\partial z_2^l} = \begin{cases} -y^k y^k & k \neq l \\ y^k(1-y^k) & k = l \end{cases}$.

To ease the computation, now we have $\frac{\partial l_1}{\partial z_2^l} = \sum_k \frac{\partial l_1}{\partial y^k} \frac{\partial y^k}{\partial z_2^l} = y^k - y^l$

Now combine the third part, $\frac{\partial l_1}{\partial w_2^{ij}} = (y^k - y^i)a_1^j$. Notice it is the outer product, we have $\frac{\partial l_1}{\partial W_2} = (\hat{y} - y)a_1^T$

2 Consider the gradient of b_2, we can directly get $\frac{\partial l_1}{\partial b_2} = (\hat{y} - y)$.

3 Next, to calculate the gradient of W_1, We have to go futher steps.

$\frac{\partial l_1}{\partial w_1^{ij}} = \sum_{l,k,h} \frac{\partial l_1}{\partial z_2^l} \frac{\partial z_2^l}{\partial a_1^k} \frac{\partial a_1^k}{\partial z_1^h} \frac{\partial z_1^h}{\partial w_2^{ij}} = \sum_{l,k,h} (y^k - y^l) w_2^{lk} \frac{\partial a_1^k}{\partial z_1^h} \frac{\partial z_1^h}{\partial w_1^{ij}}$, with $\frac{\partial a_1^k}{\partial z_1^h} = \begin{cases} a_1^h(1-a_1^h) & k = h \\ 0 & k \neq h \end{cases}$.

Now we can sum this w.r.t k, that is $\frac{\partial l_1}{\partial w_2^{ij}} = \sum_{l,h} (y^k - y^l) w_2^{lh} a_1^h (1-a_1^h) \frac{\partial z_1^h}{\partial w_1^{i,j}} = \sum_l (y^k - y^l) w_2^{li} a_1^i (1-a_1^i) x_j$.
The matrix form is $W_2^T(\hat{y} - y) .* a_1 .* (1-a_1) x^T$, where $.*$ represents the element-wise product.

4 Finally, the gradient of b_1 is $W_2^T(\hat{y} - y) .* a_1 .* (1-a_1)$.

Tanh

Notice the only different part between Tanh and Sigmoid is the gradient of activation funciton. The gradient of W_2 and b_2 are the same as sigmoid. While the gradient of tanh(z) is $1 - tanh(z)^2$, the gradient of W_1 is $W^T(\hat{y} - y) .* (1 - a_1^T a_1) x^T$

And the gradient of b_1 is $W^T(\hat{y} - y) .* (1 - a_1^T a_1)$.

ReLu

W_2 and b_2 are the same as sigmoid. The gradient of W_1 is $W^T(\hat{y} - y) .* (1 * (y > 0)) x^T$ and the gradient of b_1 is $W^T(\hat{y} - y) .* (1 * (y > 0))$.

implement the function backprop(self, X, y).

Notice the data dimension of W and b are different from the equation, the codes are not strictly align with mathematics models, with some tranposition involved.

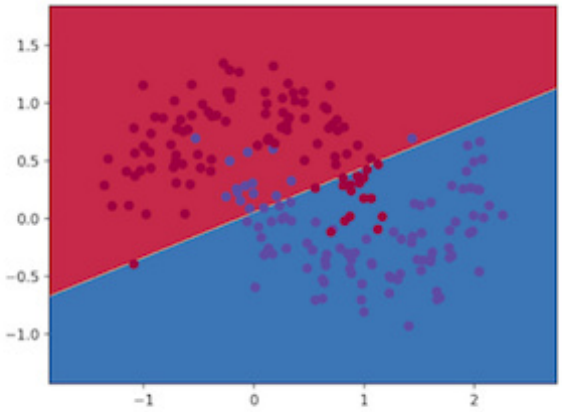
```
def backprop(self, X, y):
        '''
        backprop implements backpropagation to compute the gradients used to update t
he parameters in the backward step
        :param X: input data
        :param y: given labels
        :return: dL/dW1, dL/b1, dL/dW2, dL/db2
        '''

        # IMPLEMENT YOUR BACKPROP HERE
        num_examples = len(X)
        delta3 = self.probs
        delta3[range(num_examples), y] -= 1
        dW2 = 1/num_examples * np.matmul(np.transpose(self.a1),delta3)
        db2 = 1/num_examples * np.sum(delta3, axis = 0)
        dW1 = 1/num_examples * np.matmul(np.transpose(X),np.multiply(
                np.matmul(delta3, np.transpose(self.W2)),
                self.diff_actFun(self.a1, self.actFun_type)))
        db1 = 1/num_examples * np.sum(
                np.multiply(np.matmul(delta3, np.transpose(self.W2)),
                            self.diff_actFun(self.a1, self.actFun_type)),
                            axis = 0)
        return dW1, dW2, db1, db2
```

Time to have fun - Training!

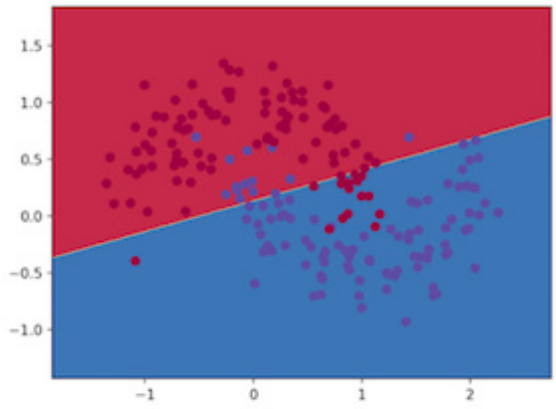
Compare the three activation functions

Now, we can train the model by runnning the three*layer*neural_network.py. First, we explore the difference between different activation functions.

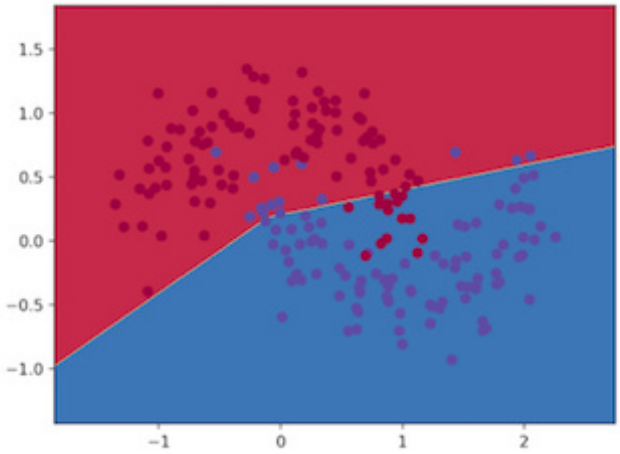
Sigmoid



Tanh



Relu

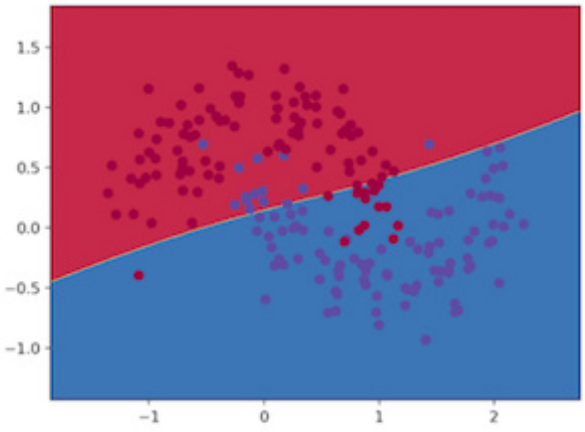


Obviously, Sigmoid and Tanh function with one hidden layer are both acting similarly as linear classfication which will change with the hidden units increasing, while the ReLu activation gives us an obvious unliear classification. The loss of ReLu method is the less than the other two methods, with the next lowest Tanh activation function. This can be seen from the graph, there is more blue points in red part in Sigmoid activation model than the other two.

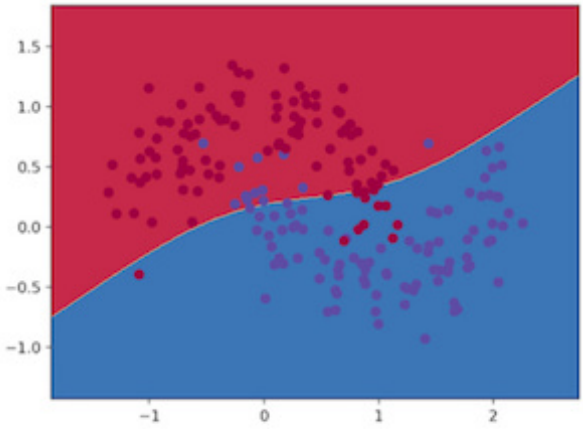
Explore the Tanh activation function

Next, we will change the number of hidden units to explore the differences of models.

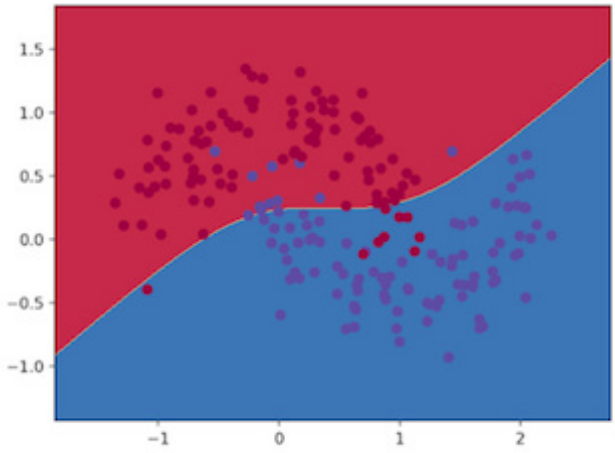
5 hidden units



8 hidden units



50 hidden units



We can see that lower hidden size gives us a more general one, while higher hidden size makes the classfication more precise. However, the percise is not a good thing, with high risk of overfitting. Generally, the loss is decreasing with the hidden units increasing, which can been seen as more information being captured with higher hidden units. The accurate of model with 50 hidden units is higher than model with 5 hidden units. I believe if we train more steps, the accurate will be higher, but with higher risk of overfit.

Even more fun - Training a deeper network!!

Rewritten the neural network class

To train a deep neural network with multiple hidden layers, it is more convenient to construct a new class to forward and backward go through each layer. The Layer class is defined as below:

```
import numpy as np
from configs import configs

config = configs()

class Layer():

    def __init__(self, layer_id, actFun):

        self.actFun = actFun
        self.layer_id = layer_id



    def feedforward(self, input_, W, b):

        if self.layer_id < config.nn_layers - 1:
            self.z = np.matmul(input_, W) + b
            self.a = self.actFun(self.z)

        if self.layer_id == config.nn_layers - 1:
            self.z = np.matmul(input_, W) + b
            exp_scores = np.exp(self.z)
            self.a = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

        return self.a

    def backprop(self, input_, num_examples, delta):

            dW = 1/num_examples * np.dot(np.transpose(input_),delta)
            db = 1/num_examples * np.sum(delta, axis = 0, keepdims = True)

            return dW, db
```

We define two main functions: feedforward and backprop, with parameters feed into these functions. The parameters are updated in the multiple neural network class, while Layer class is only responsible for calculating.

Next, we need to reedit the feedforward, backprop, calculate*loss and fit*model functions in the neural network class.

1.The feedforward codes are like below:

```
def feedforward(self):
        '''
        update the neurons in all layers
        '''

        for i, layer in enumerate(self.layers):
            self.neurons[i+1] = layer.feedforward(self.neurons[i], self.W[i], self.b[
i])

        return None
```

We only need to pass the parameters to the Layer.feedforward(), much more clear version.

2.Backprop

```
def backprop(self):

        for i, layer in enumerate(self.layers[::-1]):

            index = config.nn_layers - 1 - i

            self.dW[index], self.db[index] = layer.backprop(self.neurons[index],
                   len(self.X), self.delta[index])

        return None
```

Also, feed the parameters to Layer.backprop(). Notice, the backprop is updated in reverse order, from the last hidden layer to first hidden layer.

3.calculate_loss

```
def calculate_loss(self):
        '''
        calculate_loss computes the loss for prediction
        :param X: input data
        :param y: given labels
        :return: the loss for prediction
        '''
        num_examples = len(self.X)
        self.feedforward()
        # Calculating the loss

        data_loss = -np.sum(np.log(self.neurons[-1])[np.arange(num_examples),
                            self.y])/num_examples

        # Add regulatization term to loss (optional)
        data_loss += config.reg_lambda / 2 * np.sum([np.sum(np.square(W)) for W
                                                     in self.W] )
        return (1. / num_examples) * data_loss
```

Here, I changed the data_loss slightly to update the name. Also the regulization part is changed.

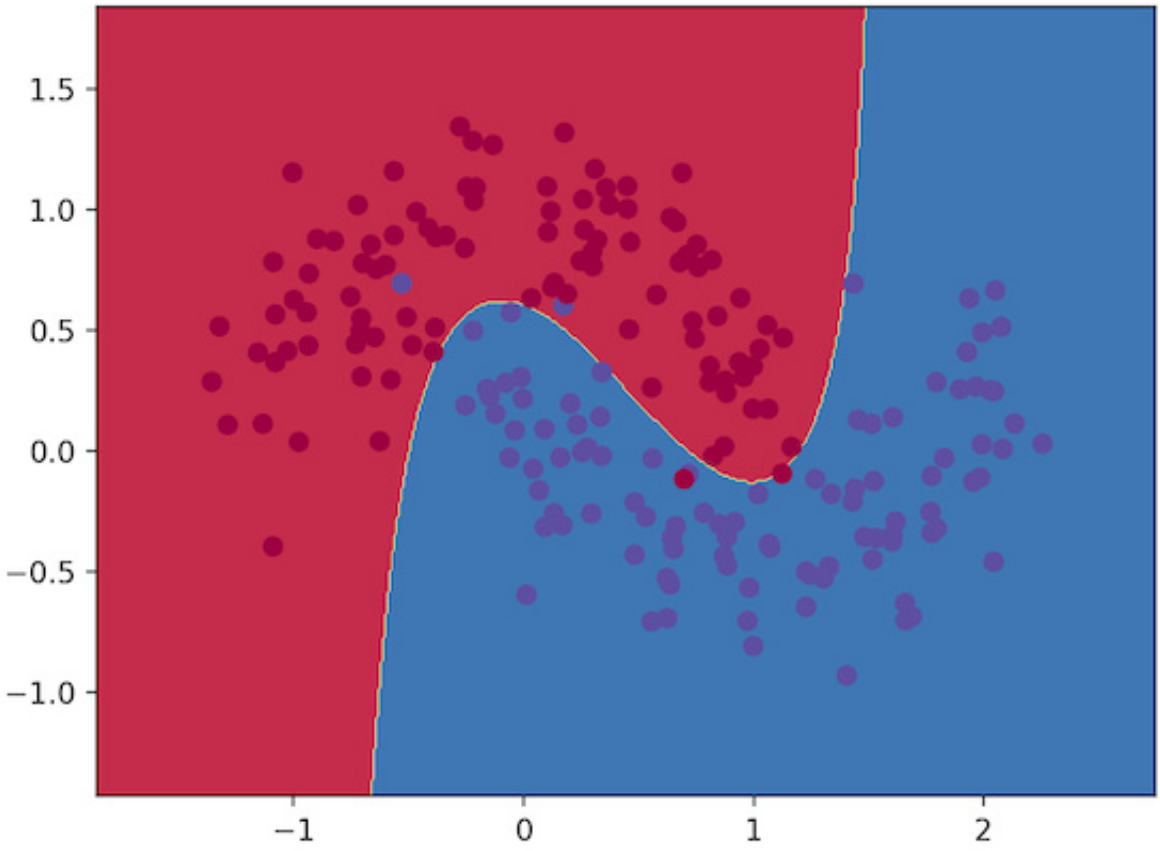
4.fit_model

```
def fit_model(self, epsilon=0.01, num_passes=20000, print_loss=True):

        # Gradient descent.
        for j in np.arange(0, num_passes):
            # Forward propagation
            self.feedforward()
            # Updata deltas
            self.neurons[-1][range(len(self.X)), self.y] -= 1
            self.delta[-1] = np.array(self.neurons[-1])
            for i in np.arange(config.nn_layers)[-2::-1]:
                self.delta[i] = np.multiply(np.dot(self.delta[i+1], self.W[i+1].T),
                          self.diff_actFun(self.neurons[i+1], config.actFun_type))

            # Backpropagation
            self.backprop()

            # Add regularization terms (b1 and b2 don't have regularization terms)
            self.dW = [dw + config.reg_lambda * w for w, dw in zip(self.W, self.dW)]

            # Gradient descent parameter update
            self.W = [w - epsilon * dw for w, dw in zip(self.W, self.dW)]
            self.b = [b - epsilon * db for b, db in zip(self.b, self.db)]

            if print_loss and j % 1000 == 0:
                print("Loss after iteration {}:{}".format(j, self.calculate_loss()))
```

The fit_model part is rewritten to include the list of deltas, which is needed in backprop function to calculate gradients.

Results

This plot is trained with hidden_layer = 4. Obviously, the prediction fits the points nearly perfect. Then we know the more hidden layers, the more accurate the predictions are. However, it is extremely possible to overfit.


1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2

Training a Simple Deep Convolutional Network on MNIST

a) Build and Train a 4-layer DCN

2.Complete the weight and bias function

```
def weight_variable(shape):
    '''
    Initialize weights
    :param shape: shape of weights, e.g. [w, h ,Cin, Cout] where
    w: width of the filters
    h: height of the filters
    Cin: the number of the channels of the filters
    Cout: the number of filters
    :return: a tensor variable for weights with initial values
    '''

    # IMPLEMENT YOUR WEIGHT_VARIABLE HERE
    initial = tf.truncated_normal(shape, stddev=0.1)
    W = tf.Variable(initial)
    return W

def bias_variable(shape):
    '''
    Initialize biases
    :param shape: shape of biases, e.g. [Cout] where
    Cout: the number of filters
    :return: a tensor variable for biases with initial values
    '''

    # IMPLEMENT YOUR BIAS_VARIABLE HERE
    initial = tf.constant(0.1, shape=shape)
    b = tf.Variable(initial)
    return b

def conv2d(x, W):
    '''
    Perform 2-D convolution
    :param x: input tensor of size [N, W, H, Cin] where
```

```
    N: the number of images
    W: width of images
    H: height of images
    Cin: the number of channels of images
    :param W: weight tensor [w, h, Cin, Cout]
    w: width of the filters
    h: height of the filters
    Cin: the number of the channels of the filters = the number of channels of images
    Cout: the number of filters
    :return: a tensor of features extracted by the filters, a.k.a. the results after
convolution
    '''

    # IMPLEMENT YOUR CONV2D HERE
    h_conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
    return h_conv

def max_pool_2x2(x):
    '''
    Perform non-overlapping 2-D maxpooling on 2x2 regions in the input data
    :param x: input data
    :return: the results of maxpooling (max-marginalized + downsampling)
    '''

    # IMPLEMENT YOUR MAX_POOL_2X2 HERE
    h_max = tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1], padding='SAME')
    return h_max
```

3.Build the network

Next, in the main function, we build the network

```
# FILL IN THE CODE BELOW TO BUILD YOUR NETWORK

    # placeholders for input data and input labeles
    x = tf.placeholder(tf.float32, shape=[None, 784])
    y_ = tf.placeholder(tf.float32, shape=[None, 10])

    # reshape the input image
    x_image = tf.reshape(x, [-1, 28, 28, 1])

    # first convolutional layer
    W_conv1 = weight_variable([5, 5, 1, 32])
    b_conv1 = bias_variable([32])
    h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
    h_pool1 = max_pool_2x2(h_conv1)

    # second convolutional layer
    W_conv2 = weight_variable([5, 5, 32, 64])
    b_conv2 = bias_variable([64])
    h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
    h_pool2 = max_pool_2x2(h_conv2)

    # densely connected layer
    W_fc1 = weight_variable([7 * 7 * 64, 1024])
    b_fc1 = bias_variable([1024])
    h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
    h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

    # dropout
    keep_prob = tf.placeholder(tf.float32)
    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

    # softmax
    W_fc2 = weight_variable([1024, 10])
    b_fc2 = bias_variable([10])
    y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

4.Set up training

```
# FILL IN THE FOLLOWING CODE TO SET UP THE TRAINING

    # setup training
    cross_entropy = tf.reduce_mean(
            tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
    train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
    correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

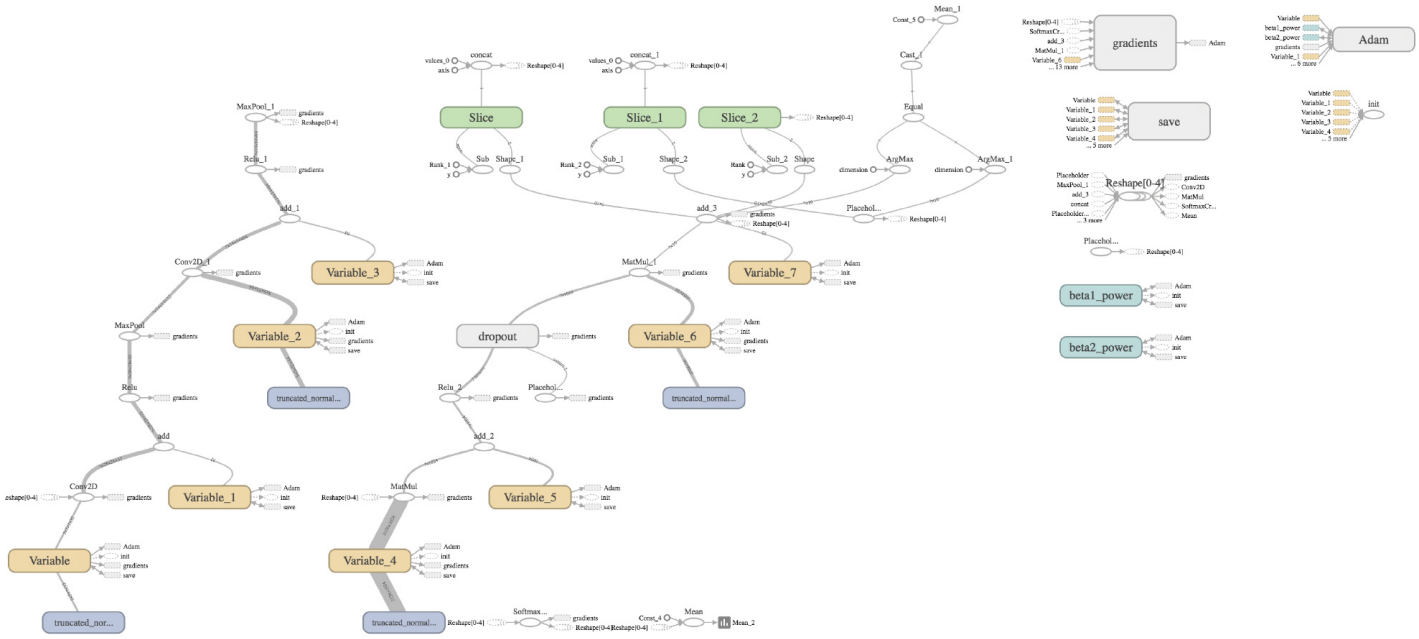
5.Run training.

We run the graph by creating a session, and we also write summary and graph. The final test accuracy is 0.9869.

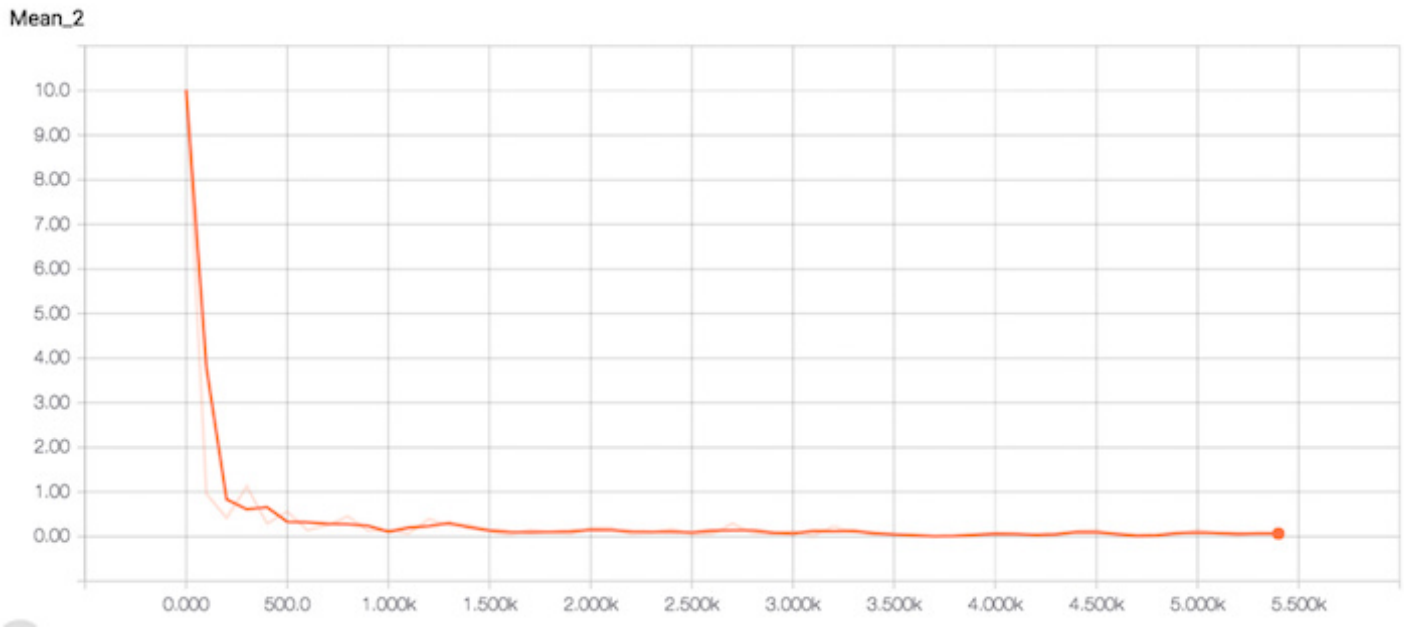
6.Visualize training

In terminal, ~ py$ tensorboard --logdir='/Users/py/Python/comp_576/results'

We get the graph:



Also, we could track the loss path:



b) More on Visualizing Your Training

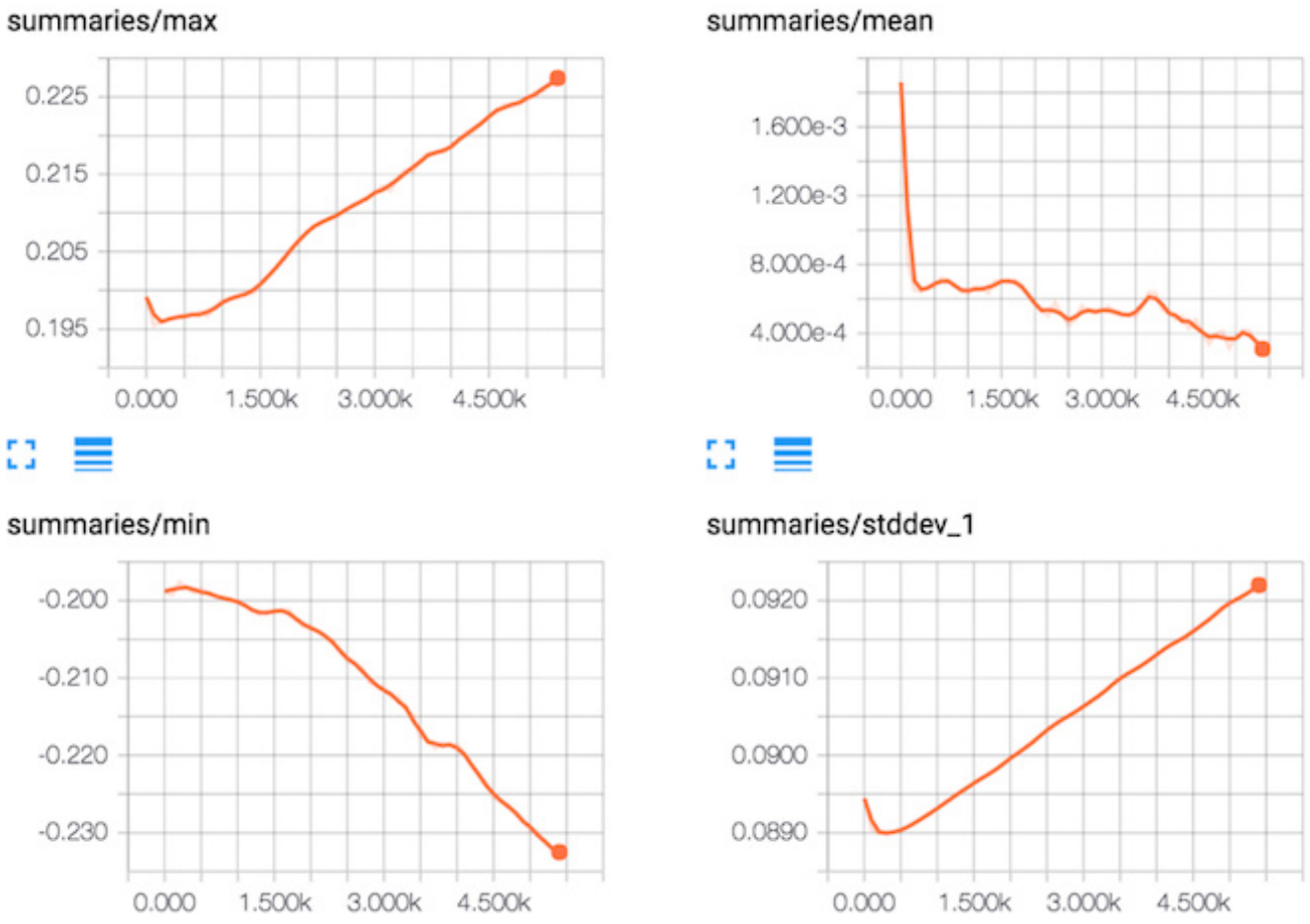
Now, we go a step further on how to monitor a set of variables during the training. As in the tutorial of TensorBoard, let's define a function to summarize statistics of variable.

```
def variable_summaries(var):

  with tf.name_scope('summaries'):
    mean = tf.reduce_mean(var)
    tf.summary.scalar('mean', mean)
    with tf.name_scope('stddev'):
      stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
    tf.summary.scalar('stddev', stddev)
    tf.summary.scalar('max', tf.reduce_max(var))
    tf.summary.scalar('min', tf.reduce_min(var))
    tf.summary.histogram('histogram', var)
```

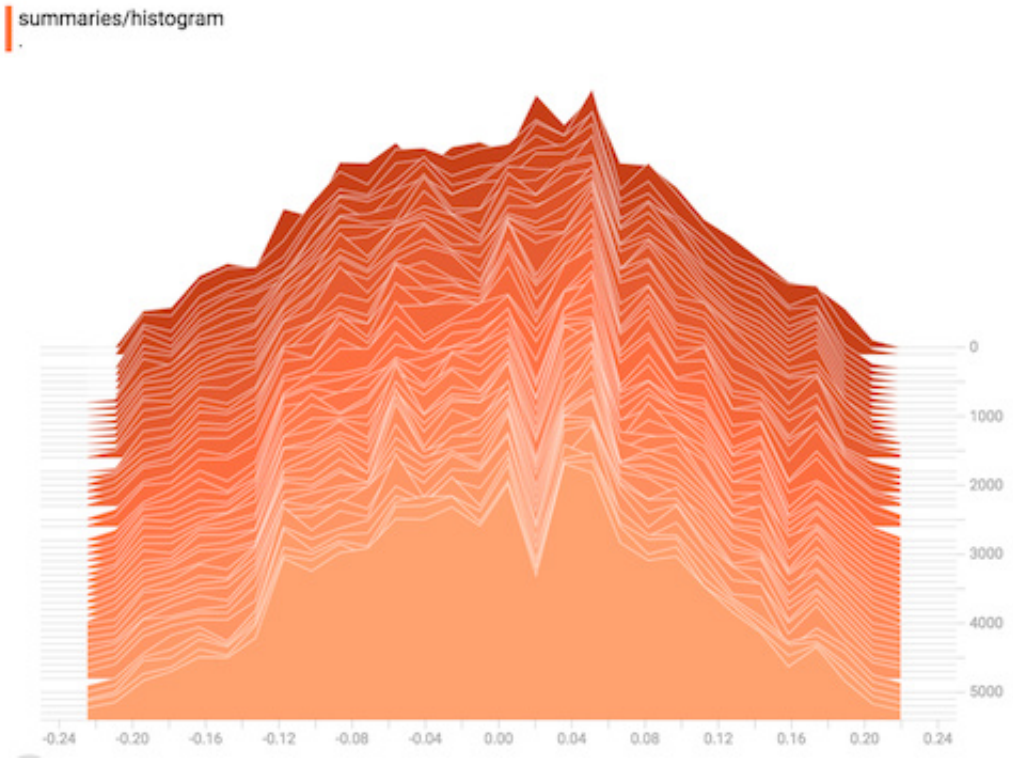
Apply this function on weights, bias, net inputs, activations after Relu at each layer, activations after max-pooling at each layer. Keep using tf.summary.merge_all(), then all the summaries of these variables are created on TensorBoard.

```
# write summaries
    variable_summaries(W_conv1)
    variable_summaries(W_conv2)
    variable_summaries(W_fc1)
    variable_summaries(W_fc2)
    variable_summaries(b_fc2)
    variable_summaries(b_fc1)
    variable_summaries(b_conv2)
    variable_summaries(b_conv1)
    variable_summaries(conv2d(h_pool1, W_conv2) + b_conv2)
    variable_summaries(conv2d(x_image, W_conv1) + b_conv1)
    variable_summaries(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
    variable_summaries(h_conv1)
    variable_summaries(h_conv2)
    variable_summaries(h_fc1)
    variable_summaries(h_pool1)
    variable_summaries(h_pool2)
```

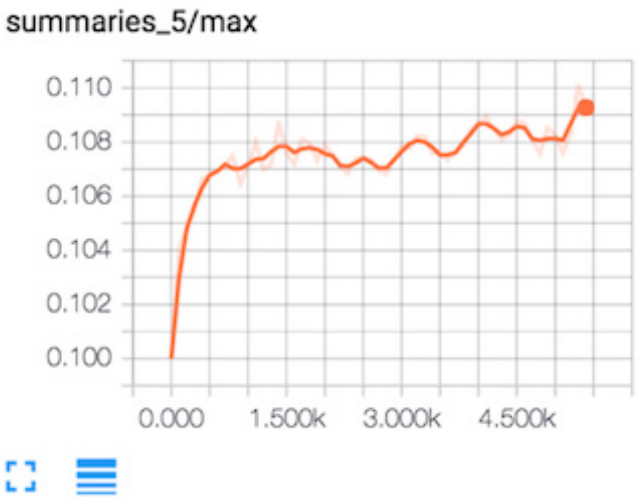
The statistics of weight at 1st layer is like follow:

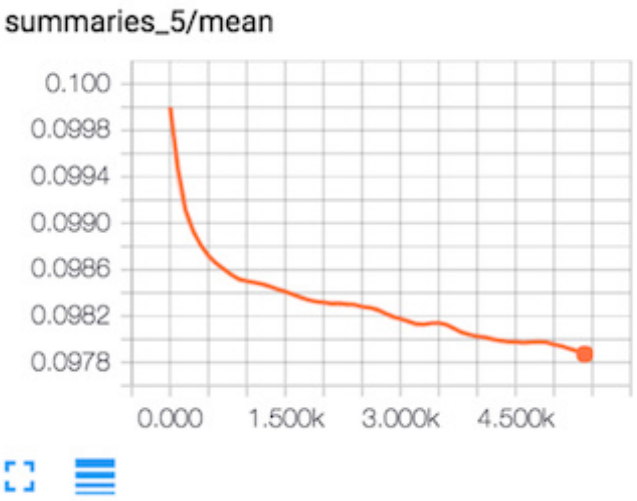


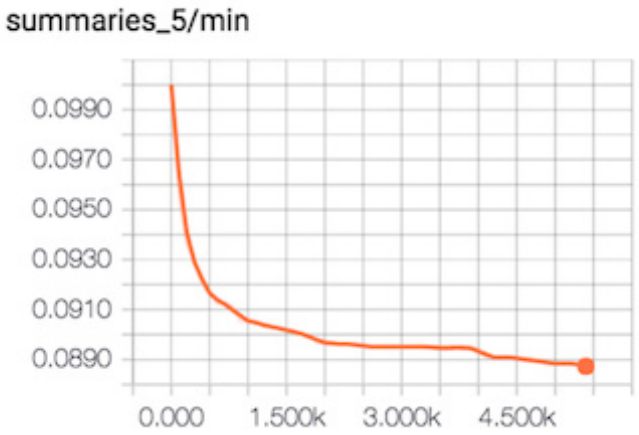
There is also histogram for weight:

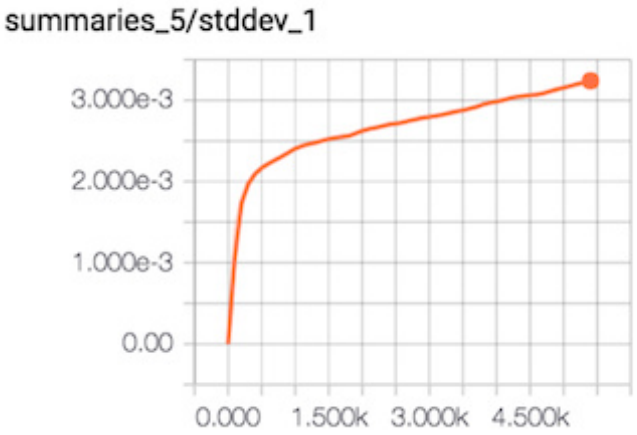


The statistics of bias at 1st layer is also attached:

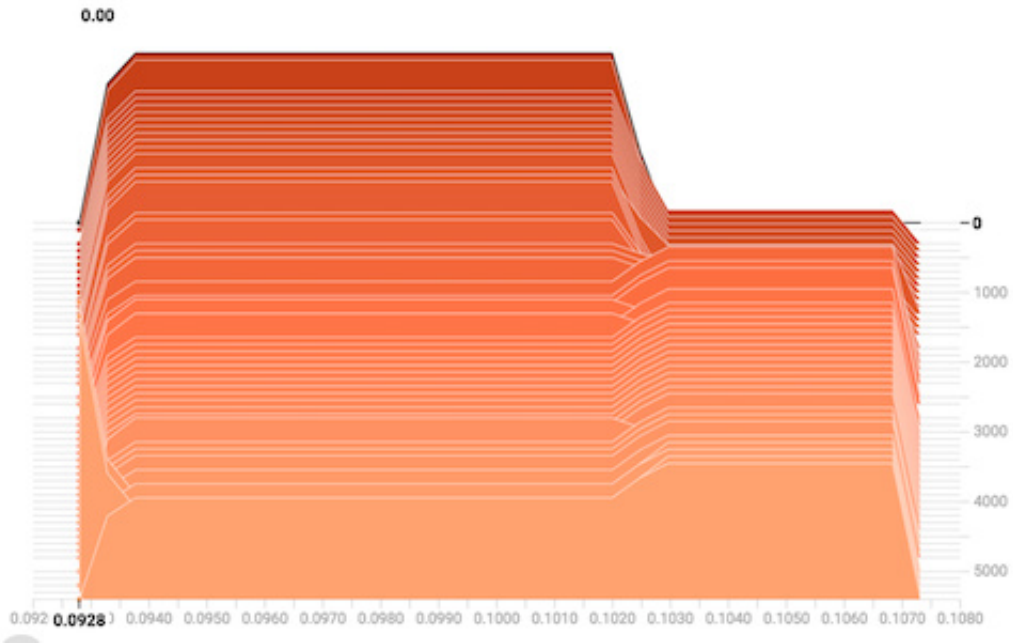








And the histogram of bias at 1st layer:



The mean of weight and bias is reducing, while the standard error is increasing, which means less robost.

Further, create a new FileWrite for test error. Every 1100 steps, we write the summary for test errors, with test set feed into session.

```
test_writer = tf.summary.FileWriter(test_dir)

if i % 1100 == 0 or i == max_step:
            summary_test_err = sess.run(summary_loss, feed_dict = {
                    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0})
            test_writer.add_summary(summary_test_err, i)
            test_writer.flush()
            checkpoint_file = os.path.join(result_dir, 'checkpoint')
            saver.save(sess, checkpoint_file, global_step=i)
```

The monitered test error is like below:

We can see the loss of test set is super low after 1000 steps.

c) Time for More Fun!!!

Here, I explored the model by trying with different nonlinearities (tanh), initialization techniques (Xavier...) and training algorithms (SGD, Momentum-based Methods). To accerlate the process, I reduce checking point steps into 200 and total steps into 1100.

For activation, try use `tf.nn.tanh`

```
h_conv1 = tf.nn.tanh(conv2d(x_image, W_conv1) + b_conv1)
```

For training algorithm, try `tf.train.GradientDescentOptimizer`

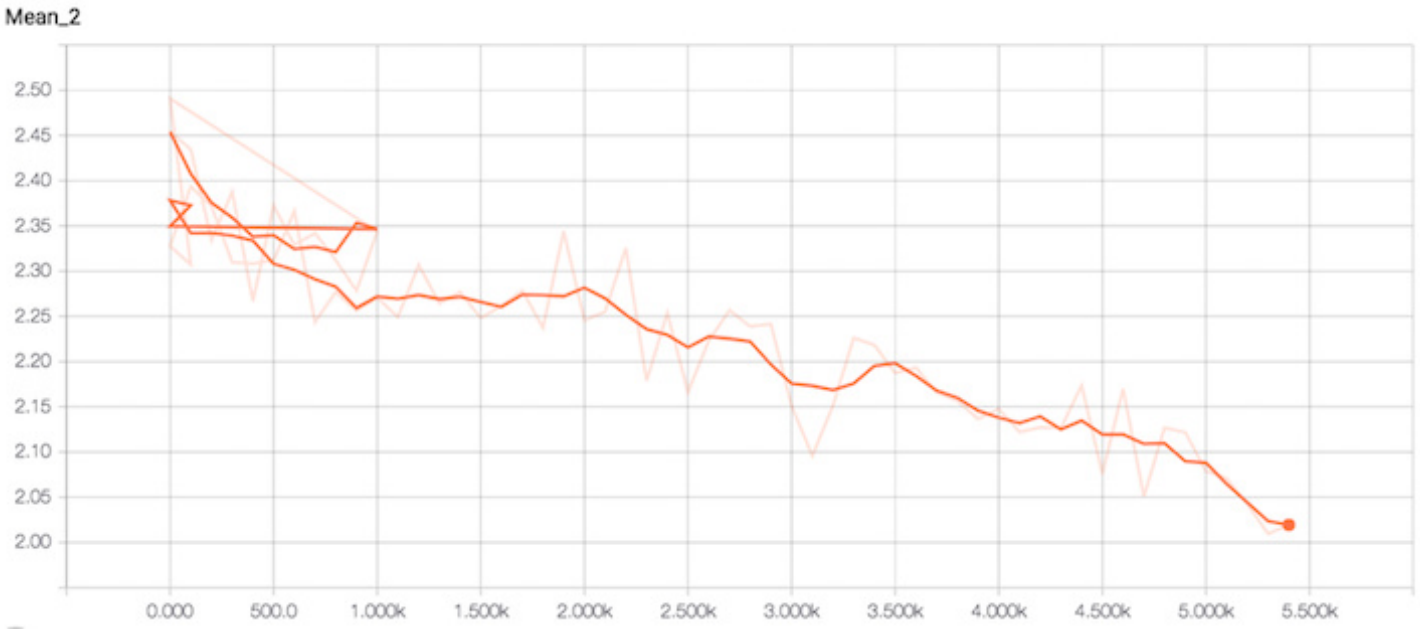
```
train_step = tf.train.GradientDescentOptimizer(1e-4).minimize(cross_entropy)
```

For initialization techniques, try use `tf.contrib.layers.xavier_initializer()` for weights, but unchaged initilizer for bias. The function parameters are slightly different to name the W.

```
def weight_variable(shape, name):

    initial = tf.contrib.layers.xavier_initializer()
    W = tf.get_variable(name = name, shape = shape, initializer = initial)
    return W
```

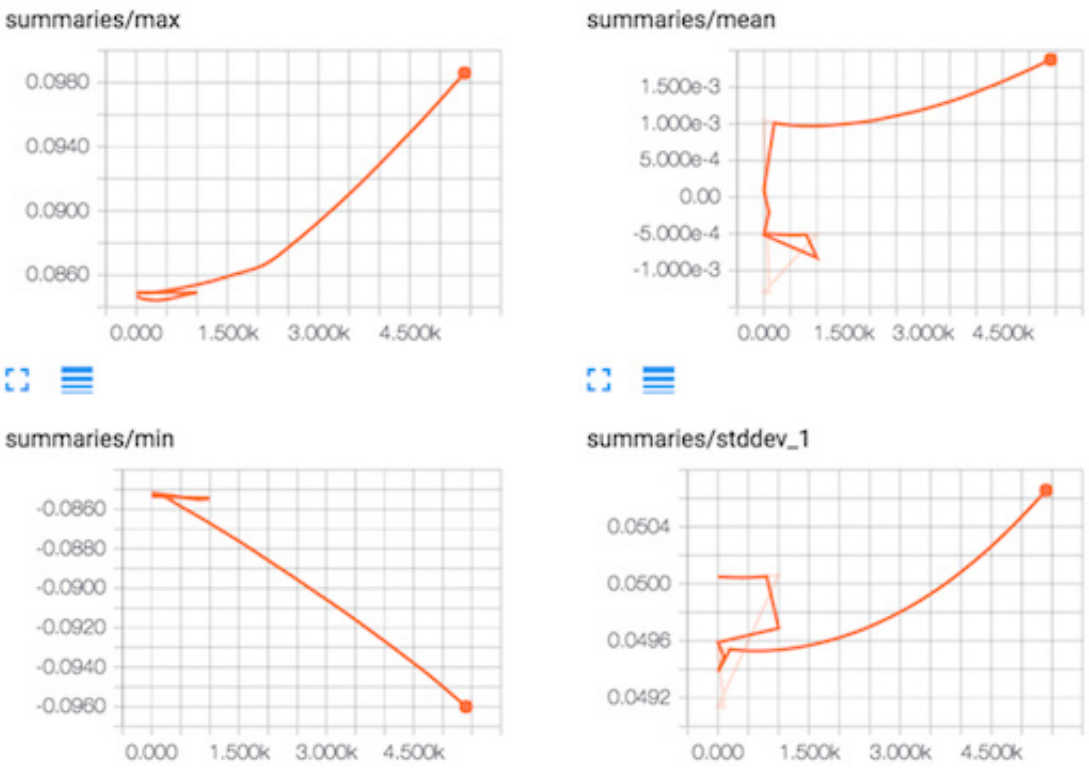
Now, retrain the model, notice the test accuracy for 5500 steps is 0.6585000157356262, which is significantly lower than last model. Besides, it moves slowly after we changed to `tf.train.GradientDescentOptimizer` method. Again, we could moniter the variables on TensorBoard.

1.First, the training loss

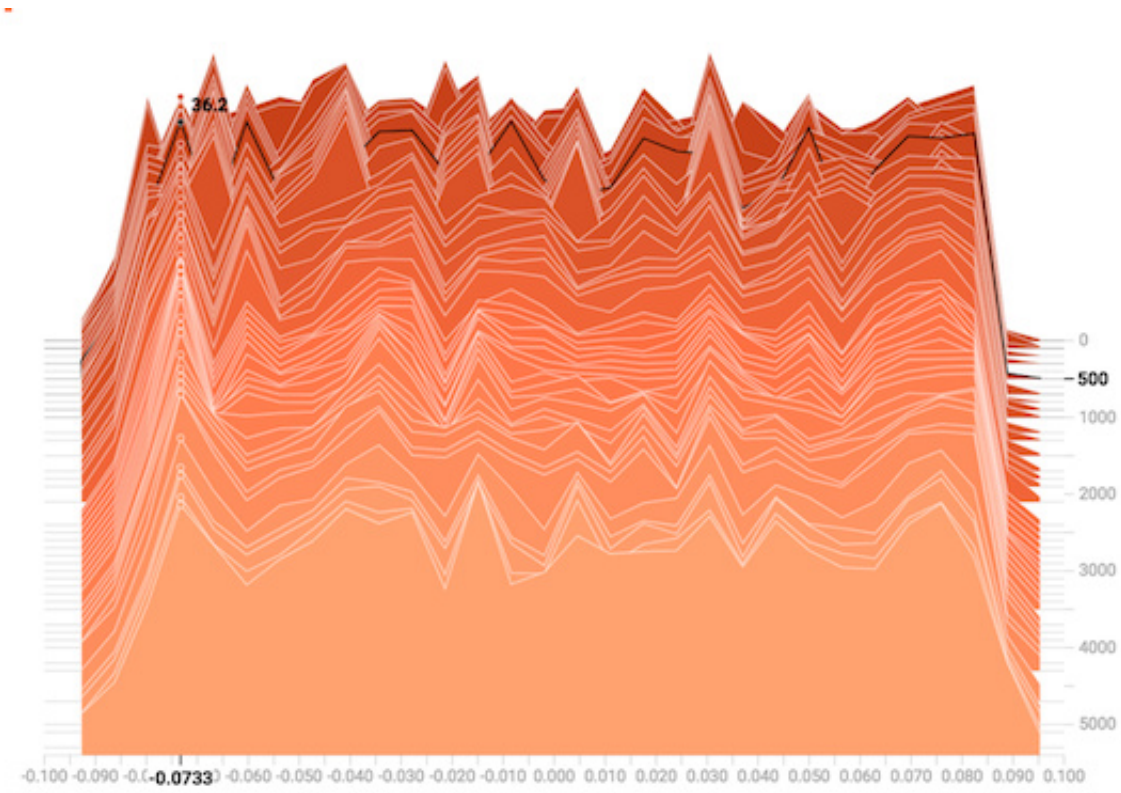


2.Next, the test loss

3.I also list the statistics of weights at 1st layer to compare:



The moving is hazard at first hundreds steps, and the mean is increasing with steps further. The variance is also being larger and larger. In general, this model does not perform as well as the old one. I suppose this is due to the activation function.



The distribution of weight is following the uniform, which is default distribution in `tf.contrib.layers.xavier_initializer()`.

The github page is:
https://github.com/Keyspan/comp576