HW 1

Dingrui Lei

The pdf is made from jupyter notebook

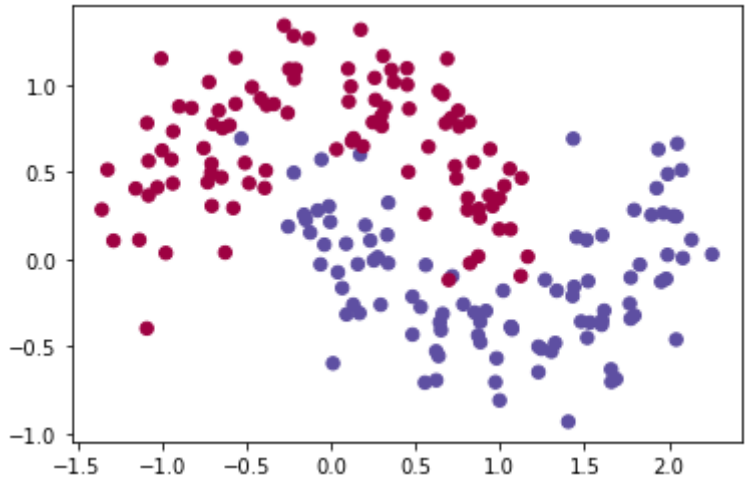
1 Backpropagation in a Simple Neural Network

a) Dataset

In [3]:
```
from three_layer_neural_network import *
```

In [4]:
```
# generate and visualize Make-Moons dataset
X, y = generate_data()
plt.scatter(X[:, 0], X[:, 1], s=40, c=y, cmap=plt.cm.Spectral)
```

Out[4]:
```
<matplotlib.collections.PathCollection at 0x24bbe54c6d0>
```



b) Activation Function

See source code in three `layer_neural_network.py` .

```
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':
        return np.tanh(z)
    if type == 'sigmoid':
        return 1/(1+np.exp(-z))
    if type == 'relu':
        return np.maximum(0,z)
    return None

def diff_actFun(self, z, type):
    '''
    diff_actFun compute 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':
        return (1 - np.tanh(z)**2)
    if type == 'sigmoid':
        return np.exp(-z)/(1+np.exp(-z))**2
    if type == 'relu':
        return (z > 0) * 1.0
    return None
```

c) Build the Neural Network

See source code in three `layer_neural_network.py` .

```
def feedforward(self, X, actFun):
    '''
    feedforward builds a 3-layer neural network and computes the two probabilities,
    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 = np.dot(X, self.W1) + self.b1
    self.a1 = actFun(self.z1)
    self.z2 = np.dot(self.a1, self.W2) + self.b2
    self.probs = self.softmax(self.z2)
    return None

def calculate_loss(self, X, y):
    '''
    calculate_loss compute 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
    y_onehot = np.stack((1-y, y), -1)
    data_loss = -(y_onehot * np.log(self.probs)).sum()

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

d) Backward Pass - Backpropagation

See source code in three layer_neural_network.py.

```
def backprop(self, X, y):
    '''
    backprop run backpropagation to compute the gradients used to update the 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)
    y_onehot = np.stack((1-y, y), -1)

    dW2 = self.a1.T.dot(self.probs - y_onehot)
    db2 = np.ones(num_examples).T.dot(self.probs - y_onehot)
    dW1 = X.T.dot((self.probs - y_onehot).dot(self.W2.T) * self.diff_actFun(self.z1, self.actFun_type))
    db1 = np.ones(num_examples).T.dot((self.probs - y_onehot).dot(self.W2.T) * (self.diff_actFun(self.z1, self.actFun_type)))
    return dW1, dW2, db1, db2
```

e) Train network with different activation functions

Differences that I observe:

Boundary generated by Tanh is more *smooth*.

Boundary generated by Sigmoid is more *smooth*.

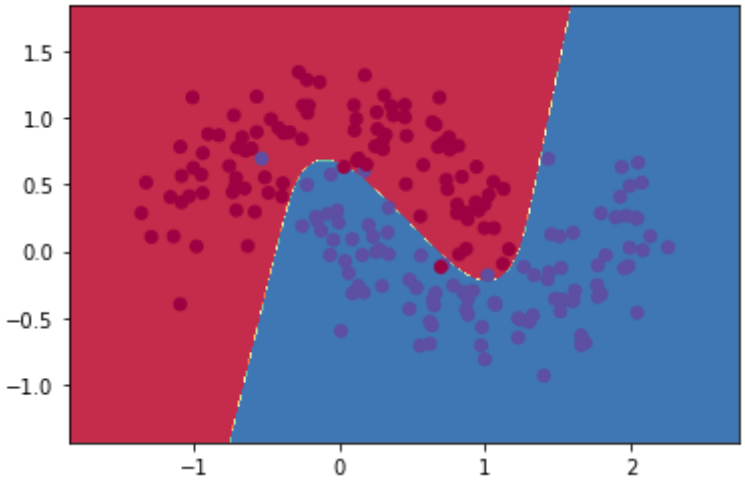
Boundary generated by ReLu is more *sharp*.

Model with Tanh

In [20]:
```
model_tanh = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3 , nn_output_dim=2, actFun
```

```
model_tanh.fit_model(X,y,print_loss=True)
model_tanh.visualize_decision_boundary(X,y)
```

```
Loss after iteration 0: data_loss:0.432166 reg_loss:0.000221
Loss after iteration 1000: data_loss:0.065788 reg_loss:0.003158
Loss after iteration 2000: data_loss:0.065461 reg_loss:0.003445
Loss after iteration 3000: data_loss:0.067217 reg_loss:0.003535
Loss after iteration 4000: data_loss:0.067168 reg_loss:0.003581
Loss after iteration 5000: data_loss:0.067146 reg_loss:0.003605
Loss after iteration 6000: data_loss:0.067135 reg_loss:0.003619
Loss after iteration 7000: data_loss:0.067130 reg_loss:0.003627
Loss after iteration 8000: data_loss:0.067126 reg_loss:0.003631
Loss after iteration 9000: data_loss:0.067124 reg_loss:0.003634
Loss after iteration 10000: data_loss:0.067123 reg_loss:0.003635
Loss after iteration 11000: data_loss:0.067123 reg_loss:0.003636
Loss after iteration 12000: data_loss:0.067122 reg_loss:0.003636
Loss after iteration 13000: data_loss:0.067122 reg_loss:0.003637
Loss after iteration 14000: data_loss:0.067122 reg_loss:0.003637
Loss after iteration 15000: data_loss:0.067122 reg_loss:0.003637
Loss after iteration 16000: data_loss:0.067122 reg_loss:0.003637
Loss after iteration 17000: data_loss:0.067122 reg_loss:0.003637
Loss after iteration 18000: data_loss:0.067122 reg_loss:0.003637
Loss after iteration 19000: data_loss:0.067122 reg_loss:0.003636
```

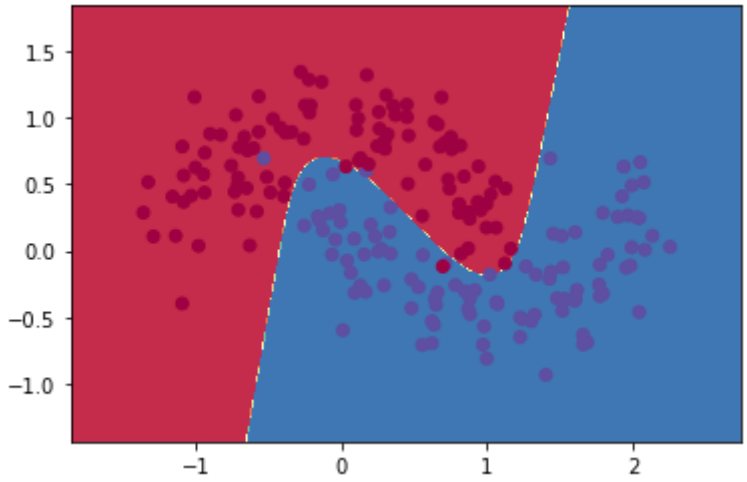


Model with Sigmoid

In [21]:

```
model_sigmoid = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3 , nn_output_dim=2, act
model_sigmoid.fit_model(X,y,print_loss=True)
model_sigmoid.visualize_decision_boundary(X,y)
```

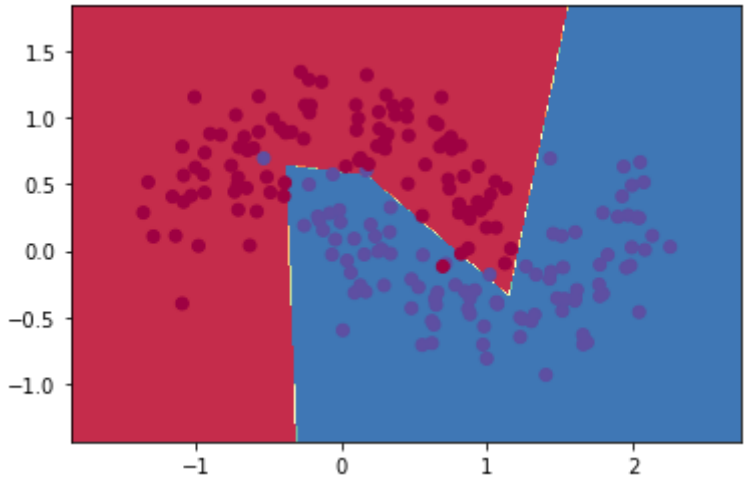
```
Loss after iteration 0: data_loss:0.628365 reg_loss:0.000206
Loss after iteration 1000: data_loss:0.079924 reg_loss:0.008507
Loss after iteration 2000: data_loss:0.069092 reg_loss:0.010506
Loss after iteration 3000: data_loss:0.067554 reg_loss:0.011050
Loss after iteration 4000: data_loss:0.067016 reg_loss:0.011314
Loss after iteration 5000: data_loss:0.066772 reg_loss:0.011460
Loss after iteration 6000: data_loss:0.066649 reg_loss:0.011543
Loss after iteration 7000: data_loss:0.066584 reg_loss:0.011591
Loss after iteration 8000: data_loss:0.066548 reg_loss:0.011618
Loss after iteration 9000: data_loss:0.066528 reg_loss:0.011633
Loss after iteration 10000: data_loss:0.066517 reg_loss:0.011642
Loss after iteration 11000: data_loss:0.066511 reg_loss:0.011647
Loss after iteration 12000: data_loss:0.066507 reg_loss:0.011650
Loss after iteration 13000: data_loss:0.066506 reg_loss:0.011651
Loss after iteration 14000: data_loss:0.066505 reg_loss:0.011651
Loss after iteration 15000: data_loss:0.066504 reg_loss:0.011651
Loss after iteration 16000: data_loss:0.066504 reg_loss:0.011651
Loss after iteration 17000: data_loss:0.066504 reg_loss:0.011651
Loss after iteration 18000: data_loss:0.066505 reg_loss:0.011651
Loss after iteration 19000: data_loss:0.066505 reg_loss:0.011651
```



Model with ReLU

In [23]:
```
model_relu = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3 , nn_output_dim=2, actFun
model_relu.fit_model(X,y,print_loss=True)
model_relu.visualize_decision_boundary(X,y)
```

```
Loss after iteration 0: data_loss:0.560052 reg_loss:0.000222
Loss after iteration 1000: data_loss:0.069051 reg_loss:0.003128
Loss after iteration 2000: data_loss:0.067702 reg_loss:0.003599
Loss after iteration 3000: data_loss:0.067401 reg_loss:0.003758
Loss after iteration 4000: data_loss:0.067367 reg_loss:0.003823
Loss after iteration 5000: data_loss:0.067296 reg_loss:0.003840
Loss after iteration 6000: data_loss:0.067434 reg_loss:0.003842
Loss after iteration 7000: data_loss:0.067249 reg_loss:0.003841
Loss after iteration 8000: data_loss:0.067427 reg_loss:0.003838
Loss after iteration 9000: data_loss:0.067248 reg_loss:0.003836
Loss after iteration 10000: data_loss:0.067256 reg_loss:0.003835
Loss after iteration 11000: data_loss:0.067253 reg_loss:0.003834
Loss after iteration 12000: data_loss:0.067253 reg_loss:0.003833
Loss after iteration 13000: data_loss:0.067238 reg_loss:0.003831
Loss after iteration 14000: data_loss:0.067281 reg_loss:0.003833
Loss after iteration 15000: data_loss:0.067243 reg_loss:0.003831
Loss after iteration 16000: data_loss:0.067280 reg_loss:0.003833
Loss after iteration 17000: data_loss:0.067240 reg_loss:0.003831
Loss after iteration 18000: data_loss:0.067258 reg_loss:0.003833
Loss after iteration 19000: data_loss:0.067387 reg_loss:0.003832
```



Model with Tanh, adding more units

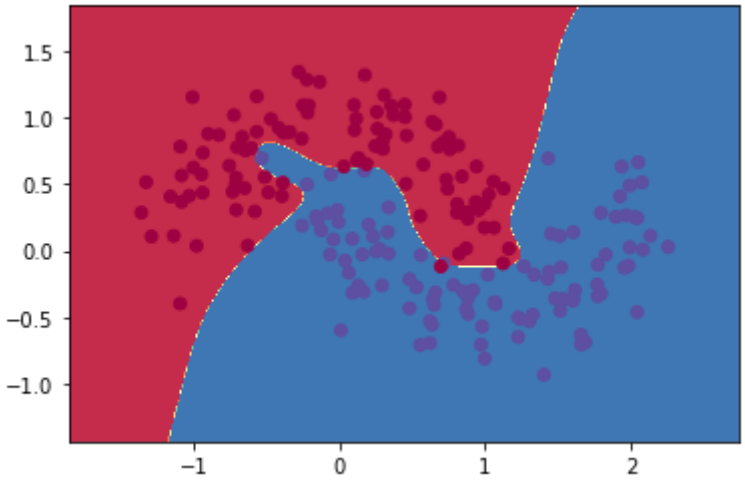
Differences that I observe:

It seems that *overfit* appears.

Boundary generated by tanh with more hidden units is more *wiggly*.

In [22]:
```
model_tanh_tanh = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=10 , nn_output_dim=2,
model_tanh_tanh.fit_model(X,y,print_loss=True)
model_tanh_tanh.visualize_decision_boundary(X,y)
```

```
Loss after iteration 0: data_loss:0.633730 reg_loss:0.000372
Loss after iteration 1000: data_loss:0.041913 reg_loss:0.005464
Loss after iteration 2000: data_loss:0.029982 reg_loss:0.008234
Loss after iteration 3000: data_loss:0.025243 reg_loss:0.009973
Loss after iteration 4000: data_loss:0.022575 reg_loss:0.011165
Loss after iteration 5000: data_loss:0.020757 reg_loss:0.012044
Loss after iteration 6000: data_loss:0.019412 reg_loss:0.012714
Loss after iteration 7000: data_loss:0.018395 reg_loss:0.013229
Loss after iteration 8000: data_loss:0.017638 reg_loss:0.013626
Loss after iteration 9000: data_loss:0.017106 reg_loss:0.013938
Loss after iteration 10000: data_loss:0.016741 reg_loss:0.014169
Loss after iteration 11000: data_loss:0.016487 reg_loss:0.014334
Loss after iteration 12000: data_loss:0.016304 reg_loss:0.014454
Loss after iteration 13000: data_loss:0.016169 reg_loss:0.014542
Loss after iteration 14000: data_loss:0.016067 reg_loss:0.014609
Loss after iteration 15000: data_loss:0.015987 reg_loss:0.014659
Loss after iteration 16000: data_loss:0.015924 reg_loss:0.014698
Loss after iteration 17000: data_loss:0.015873 reg_loss:0.014728
Loss after iteration 18000: data_loss:0.015829 reg_loss:0.014753
Loss after iteration 19000: data_loss:0.015792 reg_loss:0.014773
```



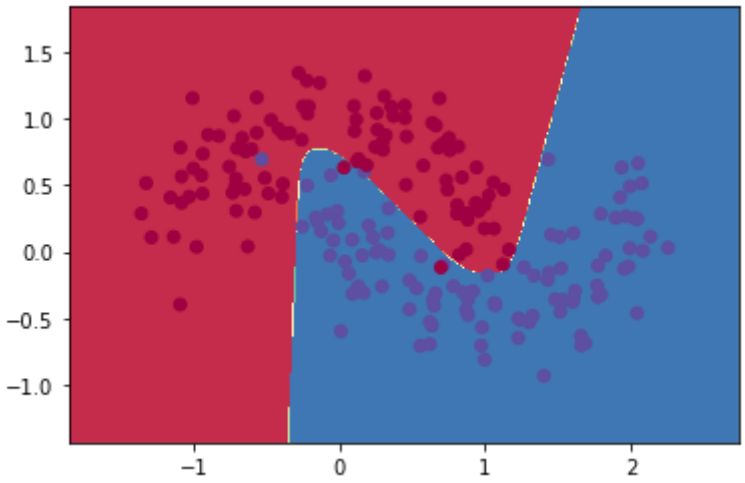
f) Training a Deeper Network!!!

See source code in three `n_layer_neural_network.py` .

In [1]:
```
from n_layer_neural_network import *
```

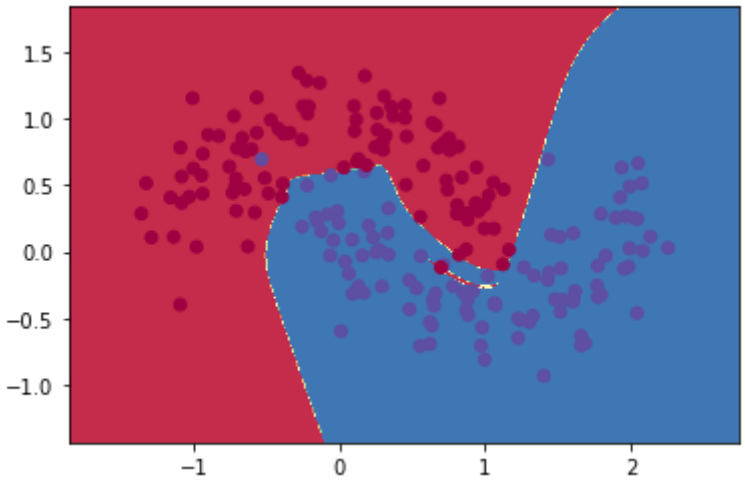
First layer with 1 hidden neuron. Second layer with 3 hidden neuron. It performs normally.

In [5]:
```
model = DeepNeuralNetwrok(n_hidden=1, input_dim=2, hidden_dim=3, output_dim=2, actFu
model.fit_model(X,y,print_loss=False)
model.visualize_decision_boundary(X,y)
```



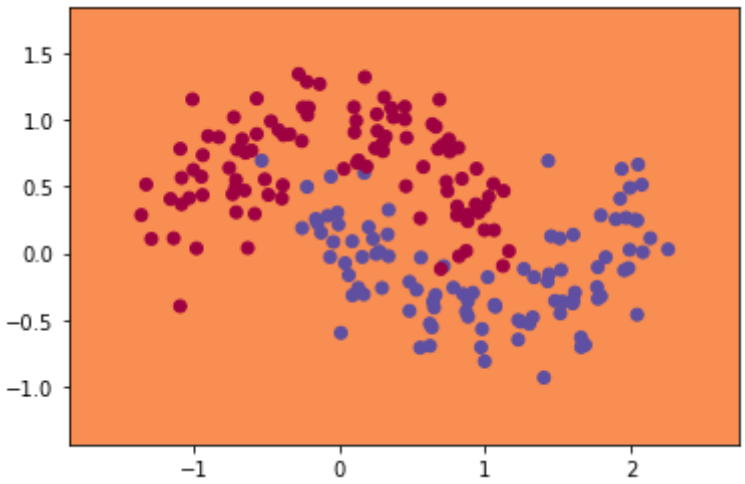
First layer with 3 hidden neuron. Second layer with 5 hidden neuron. The overfit appears with werid boudary and noisy nodes.

In [10]:
```
model = DeepNeuralNetwrok(n_hidden=3, input_dim=2, hidden_dim=5, output_dim=2, actFu
model.fit_model(X,y,print_loss=False)
model.visualize_decision_boundary(X,y)
```



First layer with 6 hidden neuron. Second layer with 10 hidden neuron. The result is overfitted severely.

In [11]:
```
model = DeepNeuralNetwrok(n_hidden=6, input_dim=2, hidden_dim=10, output_dim=2, actF
model.fit_model(X,y,print_loss=False)
model.visualize_decision_boundary(X,y)
```



Import new tesing dataset

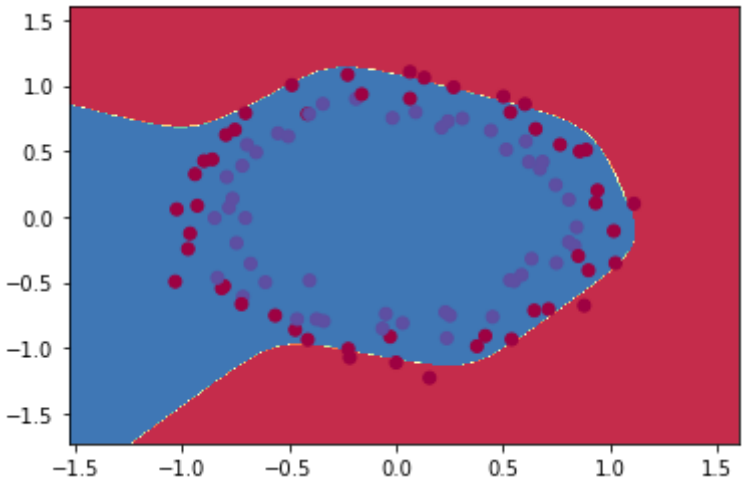
In [20]:
```
from sklearn import datasets

np.random.seed(0)

X, y = datasets.make_circles(100, noise=0.07)
```

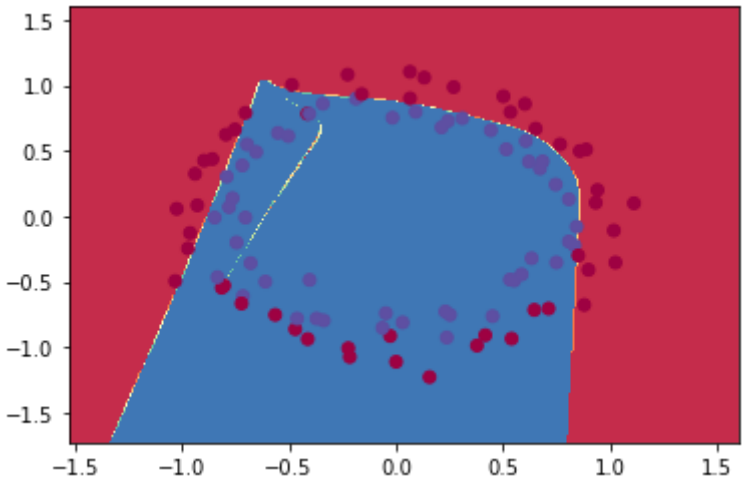
First layer with 1 hidden neuron. Second layer with 3 hidden neuron. The result is bad.

In [21]:
```
model = DeepNeuralNetwrok(n_hidden=1, input_dim=2, hidden_dim=3, output_dim=2,actFun
model.fit_model(X,y,print_loss=False)
model.visualize_decision_boundary(X,y)
```



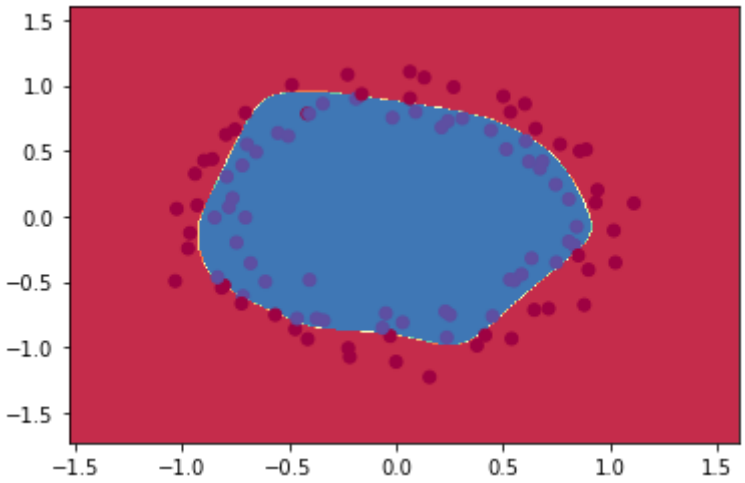
First layer with 3 hidden neuron. Second layer with 5 hidden neuron. The result is bad still.

In [22]:
```
model = DeepNeuralNetwrok(n_hidden=3, input_dim=2, hidden_dim=5, output_dim=2,actFun
model.fit_model(X,y,print_loss=False)
model.visualize_decision_boundary(X,y)
```



First layer with 5 hidden neuron. Second layer with 10 hidden neuron. The classification is close to good.

In [27]:
```
model = DeepNeuralNetwrok(n_hidden=5, input_dim=2, hidden_dim=10, output_dim=2,actFu
model.fit_model(X,y,print_loss=False)
model.visualize_decision_boundary(X,y)
```



2 Training a Simple Deep Convolutional Network on MNIST

a) Build and Train a 4-layer DCN

See source code in three `dcn_mnist.py` .

In [2]:
```
from dcn_mnist import *
```

Run Training

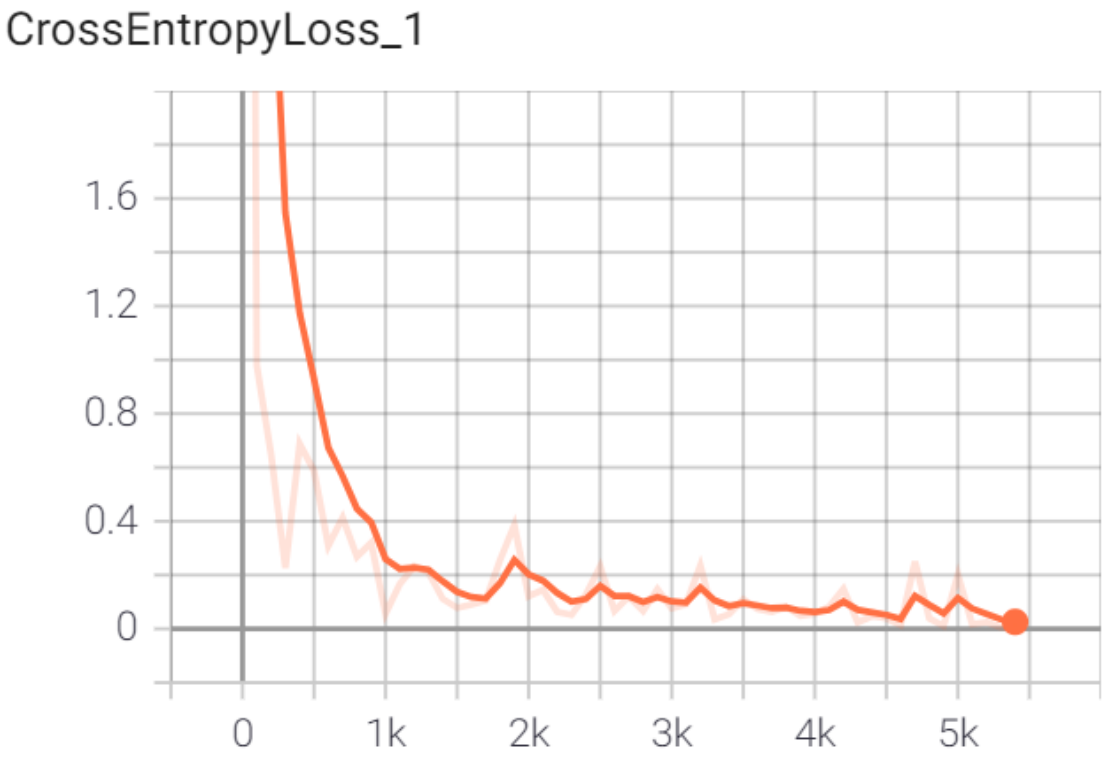
In [5]:
```
main()
```

```
WARNING:tensorflow:From C:\anaconda\lib\site-packages\tensorflow\python\util\dispatc
h.py:201: calling dropout (from tensorflow.python.ops.nn_ops) with keep_prob is depr
ecated and will be removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 - keep_pro
b`.
WARNING:tensorflow:From C:\anaconda\lib\site-packages\tensorflow\python\util\tf_shou
ld_use.py:247: initialize_all_variables (from tensorflow.python.ops.variables) is de
precated and will be removed after 2017-03-02.
Instructions for updating:
Use `tf.global_variables_initializer` instead.
step 0, training accuracy 0.08
step 100, training accuracy 0.82
step 200, training accuracy 0.88
step 300, training accuracy 0.98
step 400, training accuracy 0.84
step 500, training accuracy 0.88
step 600, training accuracy 0.94
step 700, training accuracy 0.88
step 800, training accuracy 0.92
step 900, training accuracy 0.9
step 1000, training accuracy 1
step 1100, training accuracy 1
step 1200, training accuracy 0.92
step 1300, training accuracy 0.94
step 1400, training accuracy 0.98
step 1500, training accuracy 1
step 1600, training accuracy 1
step 1700, training accuracy 0.96
step 1800, training accuracy 0.94
step 1900, training accuracy 0.94
step 2000, training accuracy 0.98
step 2100, training accuracy 0.98
```

```
step 2200, training accuracy 0.98
step 2300, training accuracy 1
step 2400, training accuracy 0.94
step 2500, training accuracy 0.98
step 2600, training accuracy 0.98
step 2700, training accuracy 1
step 2800, training accuracy 1
step 2900, training accuracy 0.98
step 3000, training accuracy 1
step 3100, training accuracy 0.98
step 3200, training accuracy 0.94
step 3300, training accuracy 0.98
step 3400, training accuracy 0.98
step 3500, training accuracy 0.98
step 3600, training accuracy 1
step 3700, training accuracy 0.96
step 3800, training accuracy 1
step 3900, training accuracy 0.98
step 4000, training accuracy 0.98
step 4100, training accuracy 1
step 4200, training accuracy 0.94
step 4300, training accuracy 1
step 4400, training accuracy 0.98
step 4500, training accuracy 1
step 4600, training accuracy 1
step 4700, training accuracy 0.98
step 4800, training accuracy 1
step 4900, training accuracy 1
step 5000, training accuracy 0.94
step 5100, training accuracy 1
step 5200, training accuracy 1
step 5300, training accuracy 1
step 5400, training accuracy 1
test accuracy 0.9858
The training takes 460.587252 second to finish
```

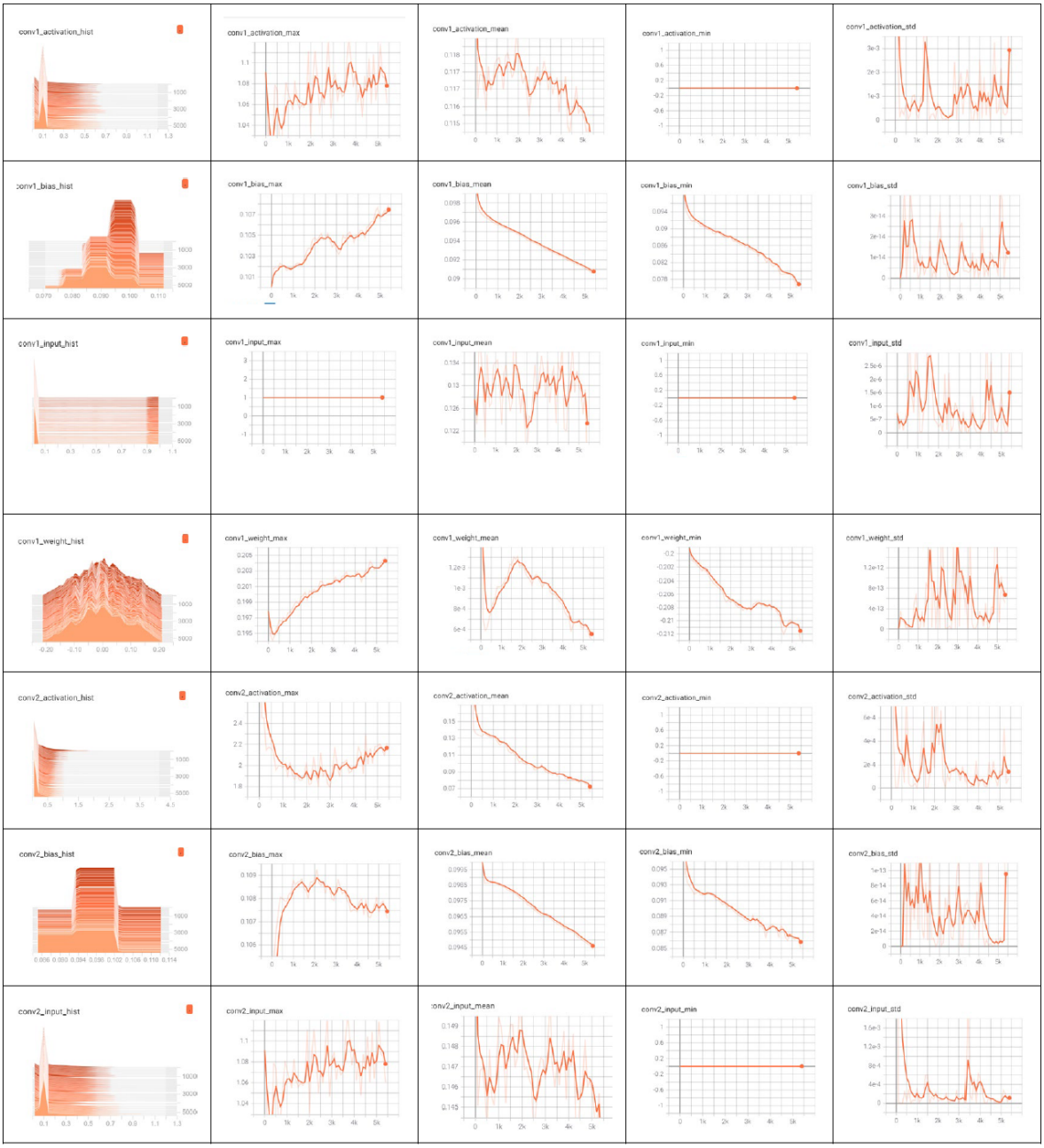
Visualize Training

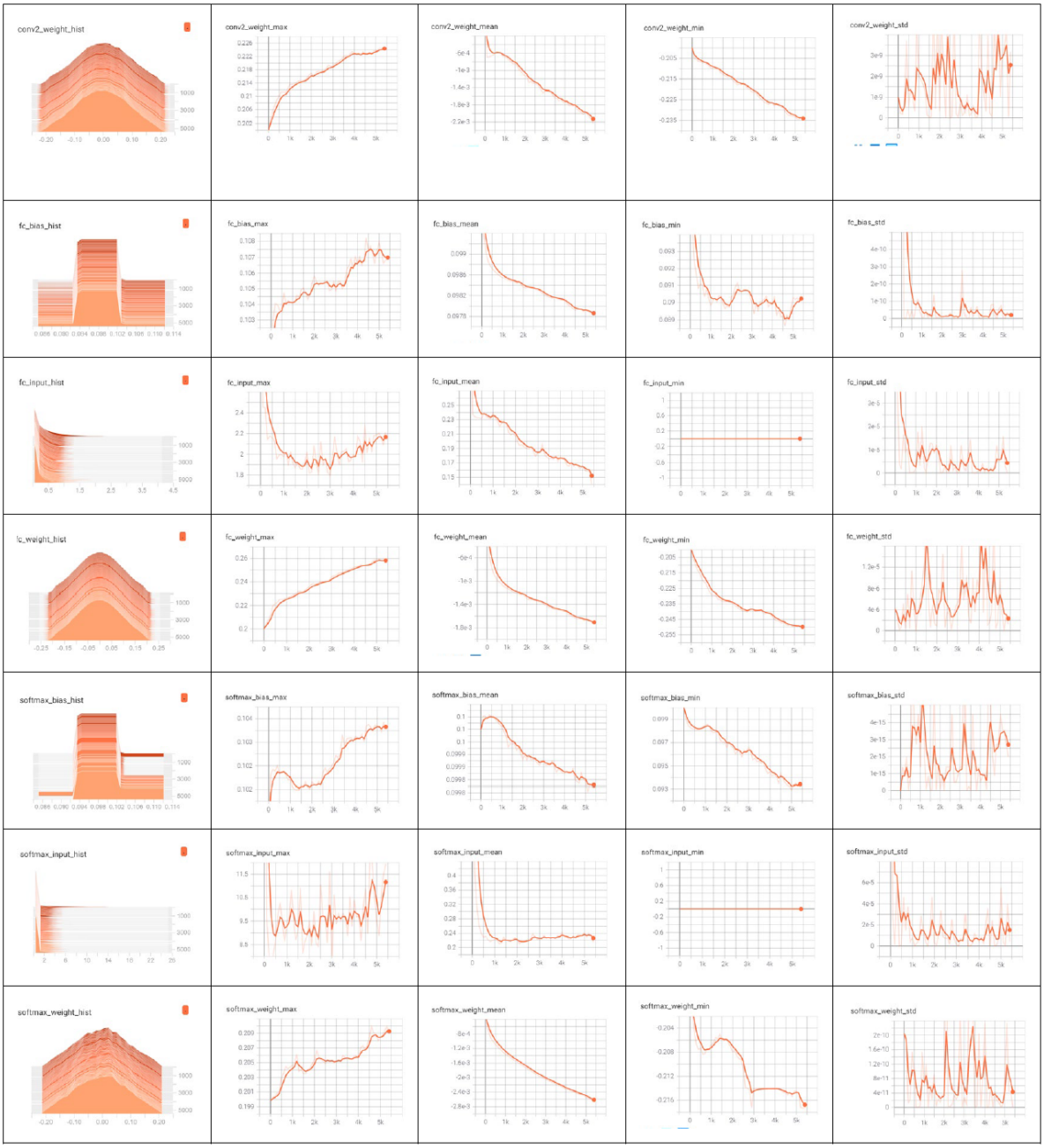
CrossEntropyLoss_1



b) Build and Train a 4-layer DCN

```python
def scalar_summary(para, para_name):
    tf.summary.scalar(para_name + '_mean', tf.reduce_mean(para))
    tf.summary.scalar(para_name + '_max', tf.reduce_max(para))
    tf.summary.scalar(para_name + '_min', tf.reduce_min(para))
    tf.summary.scalar(para_name + '_std', (tf.reduce_sum(para - tf.reduce_mean(para)))**2)
    tf.summary.histogram(para_name + '_hist', para)
tf.summary.scalar(cross_entropy.op.name, cross_entropy)
scalar_summary(W_conv1, 'conv1_weight')
scalar_summary(b_conv1, 'conv1_bias')
scalar_summary(x_image, 'conv1_input')
scalar_summary(h_conv1, 'conv1_activation')
scalar_summary(W_conv2, 'conv2_weight')
scalar_summary(b_conv2, 'conv2_bias')
scalar_summary(h_pool1, 'conv2_input')
scalar_summary(h_conv2, 'conv2_activation')
scalar_summary(W_fc1, 'fc_weight')
scalar_summary(b_fc1, 'fc_bias')
scalar_summary(h_pool2_flat, 'fc_input')
scalar_summary(W_fc2, 'softmax_weight')
scalar_summary(b_fc2, 'softmax_bias')
scalar_summary(h_fc1_drop, 'softmax_input')
```





c) Time for More Fun!!!

See modified code in modified `modified_dcn_mnist.py`

What I observe:

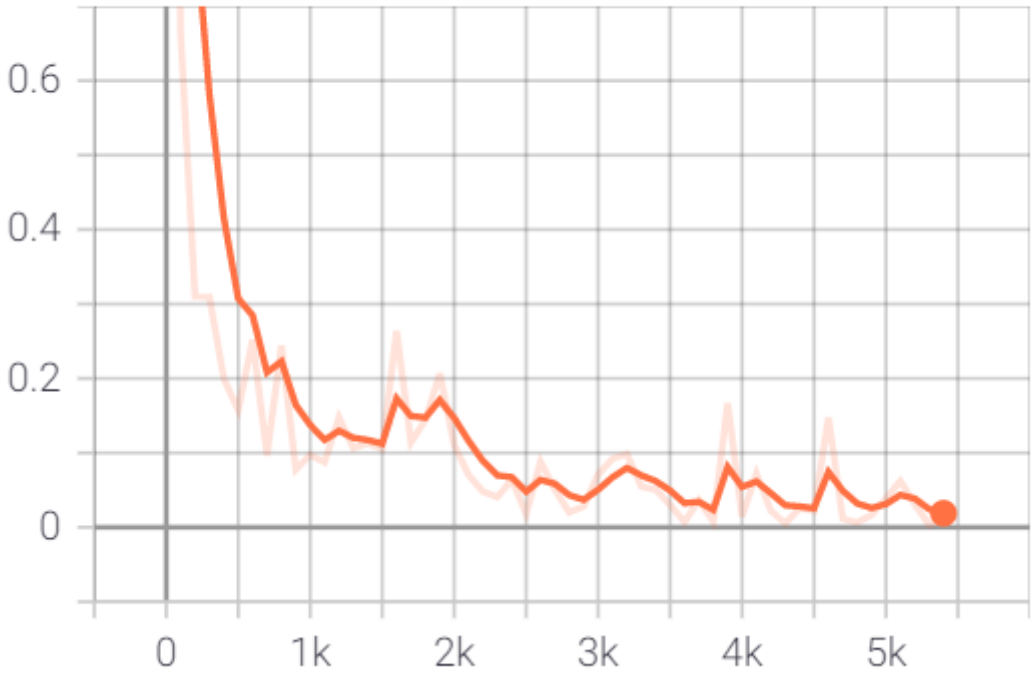
I use substitute ReLU with tanh for activation and initialize the weights and biases by Xavier. From the original histogram generated by tensorboard, It can be clearly seen that most of the neurons in the network are dead with zero activation. During backpropagation, those dead neurons don't let the gradient flow and can cause slower learning. After using ReLU and Xavier, as expected, the distribution of neuron activations more closely resembles a normal distribution, and most neurons in the network have activations. The neural network also learns slightly faster, as shown in the test accuracy graph.

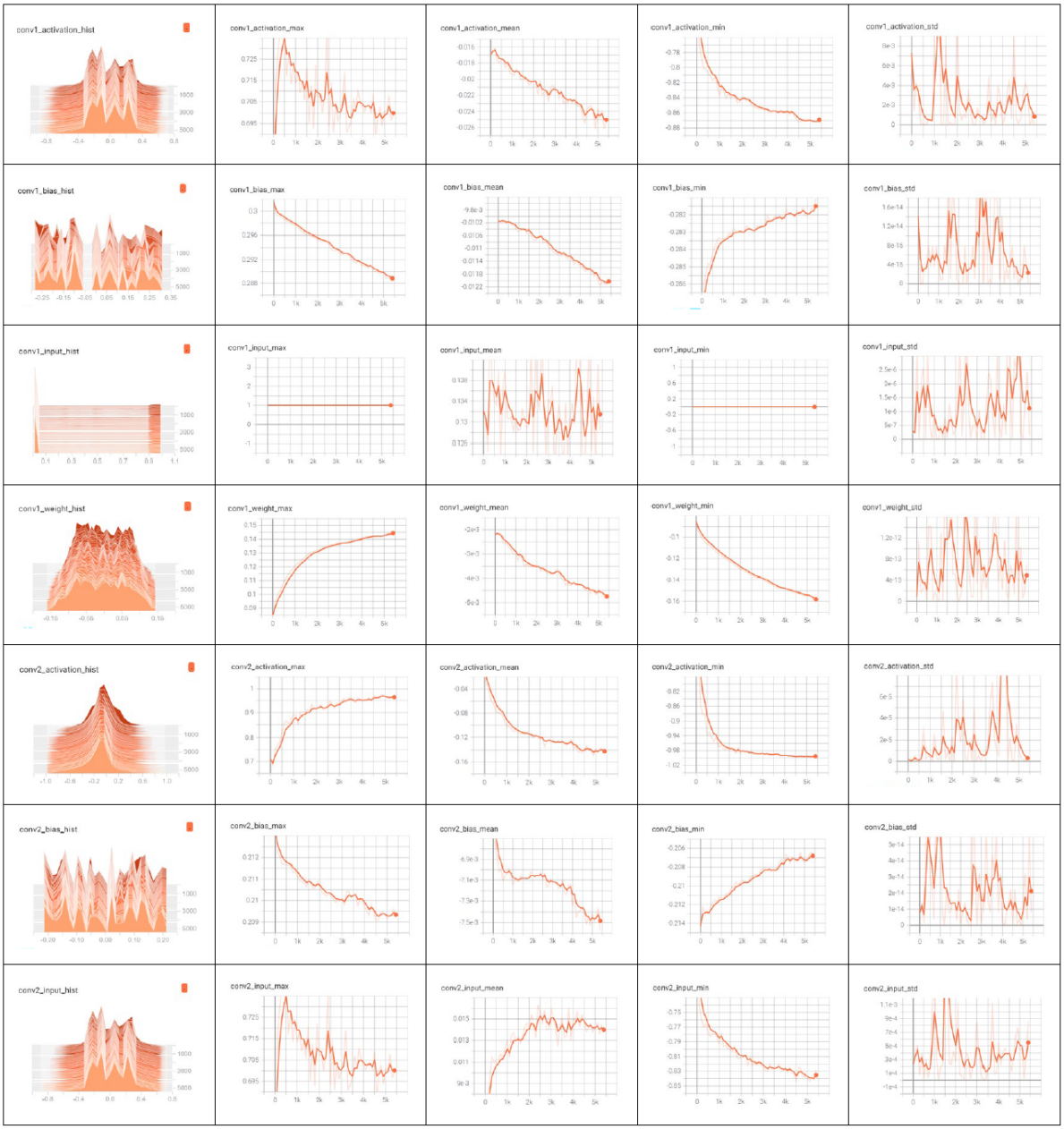
In [3]:
```
import modified_dcn_mnist
```

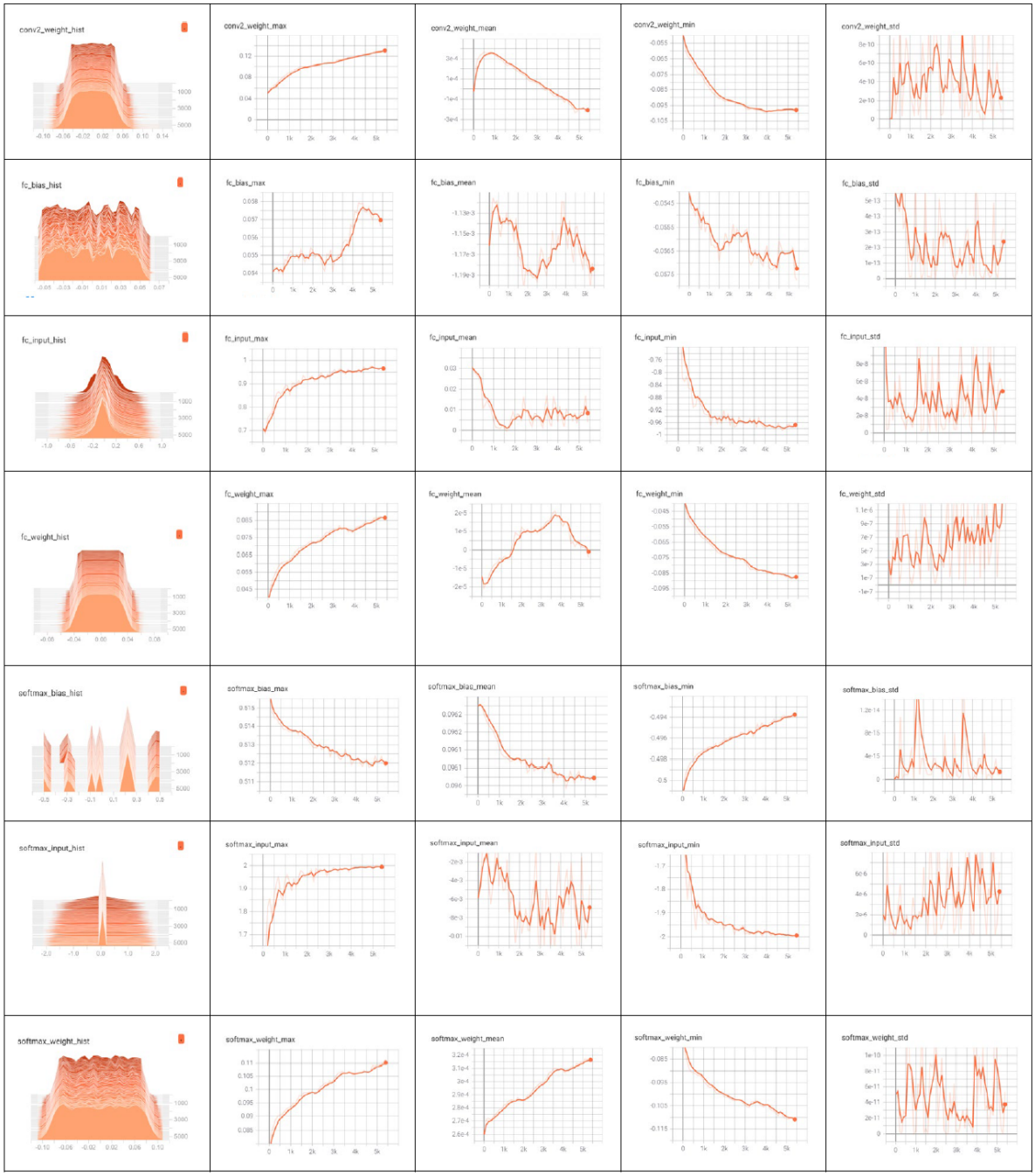
In [2]:
```
modified_dcn_mnist.main()
```

```
WARNING:tensorflow:From C:\anaconda\lib\site-packages\tensorflow\python\util\dispatc
h.py:201: calling dropout (from tensorflow.python.ops.nn_ops) with keep_prob is depr
ecated and will be removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 - keep_pro
b`.
WARNING:tensorflow:From C:\anaconda\lib\site-packages\tensorflow\python\util\tf_shou
ld_use.py:247: initialize_all_variables (from tensorflow.python.ops.variables) is de
precated and will be removed after 2017-03-02.
Instructions for updating:
Use `tf.global_variables_initializer` instead.
step 0, training accuracy 0.08
step 100, training accuracy 0.84
step 200, training accuracy 0.92
step 300, training accuracy 0.86
step 400, training accuracy 0.94
step 500, training accuracy 0.94
step 600, training accuracy 0.92
step 700, training accuracy 0.96
step 800, training accuracy 0.94
step 900, training accuracy 0.98
step 1000, training accuracy 0.96
step 1100, training accuracy 0.96
step 1200, training accuracy 0.94
step 1300, training accuracy 0.98
step 1400, training accuracy 0.94
step 1500, training accuracy 0.98
step 1600, training accuracy 0.92
step 1700, training accuracy 0.96
step 1800, training accuracy 0.98
step 1900, training accuracy 0.96
step 2000, training accuracy 0.96
step 2100, training accuracy 1
step 2200, training accuracy 0.98
step 2300, training accuracy 0.98
step 2400, training accuracy 0.98
step 2500, training accuracy 1
step 2600, training accuracy 0.96
step 2700, training accuracy 1
step 2800, training accuracy 1
step 2900, training accuracy 1
step 3000, training accuracy 1
step 3100, training accuracy 0.98
step 3200, training accuracy 0.96
step 3300, training accuracy 1
step 3400, training accuracy 1
step 3500, training accuracy 1
step 3600, training accuracy 1
step 3700, training accuracy 1
step 3800, training accuracy 1
step 3900, training accuracy 0.98
step 4000, training accuracy 1
step 4100, training accuracy 0.96
step 4200, training accuracy 1
step 4300, training accuracy 1
step 4400, training accuracy 1
step 4500, training accuracy 1
step 4600, training accuracy 0.98
step 4700, training accuracy 1
step 4800, training accuracy 1
step 4900, training accuracy 1
step 5000, training accuracy 1
step 5100, training accuracy 0.98
step 5200, training accuracy 1
step 5300, training accuracy 1
step 5400, training accuracy 1
test accuracy 0.9873
The training takes 529.611185 second to finish
```

CrossEntropyLoss_1




conv1_activation_hist
conv1_activation_max
conv1_activation_mean
conv1_activation_min
conv1_activation_std
conv1_bias_hist
conv1_bias_max
conv1_bias_mean
conv1_bias_min
conv1_bias_std
conv1_input_hist
conv1_input_max
conv1_input_mean
conv1_input_min
conv1_input_std
conv1_weight_hist
conv1_weight_max
conv1_weight_mean
conv1_weight_min
conv1_weight_std
conv2_activation_hist
conv2_activation_max
conv2_activation_mean
conv2_activation_min
conv2_activation_std
conv2_bias_hist
conv2_bias_max
conv2_bias_mean
conv2_bias_min
conv2_bias_std
conv2_input_hist
conv2_input_max
conv2_input_mean
conv2_input_min
conv2_input_std


conv2_weight_hist
conv2_weight_max
conv2_weight_mean
conv2_weight_min
conv2_weight_std
fc_bias_hist
fc_bias_max
fc_bias_mean
fc_bias_min
fc_bias_std
fc_input_hist
fc_input_max
fc_input_mean
fc_input_min
fc_input_std
fc_weight_hist
fc_weight_max
fc_weight_mean
fc_weight_min
fc_weight_std
softmax_bias_hist
softmax_bias_max
softmax_bias_mean
softmax_bias_min
softmax_bias_std
softmax_input_hist
softmax_input_max
softmax_input_mean
softmax_input_min
softmax_input_std
softmax_weight_hist
softmax_weight_max
softmax_weight_mean
softmax_weight_min
softmax_weight_std

In [ ]: