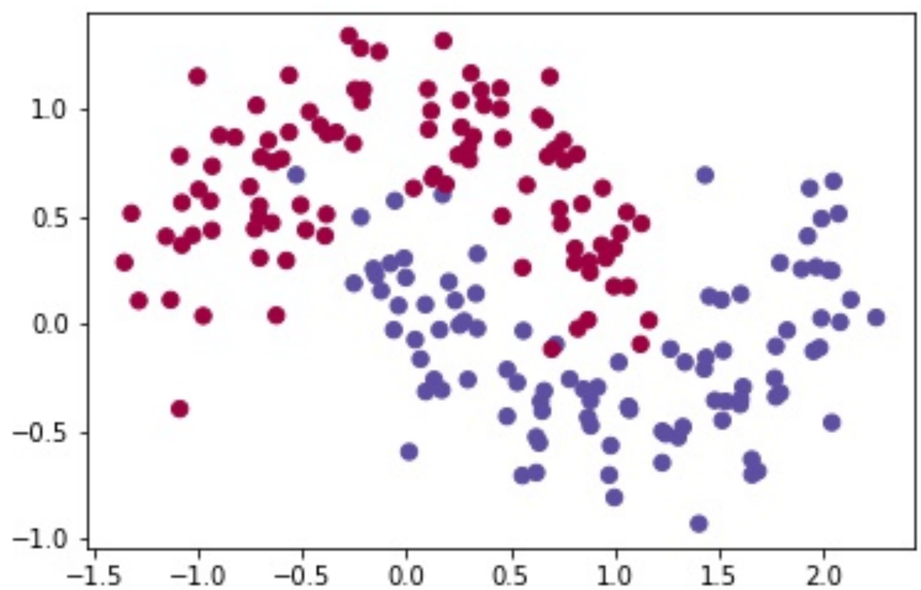
Assignment 1, COMP576

Sophie Sun (ys97)
Oct 3th, 2022

Task 1 : Backpropagation in a Simple Neural Network

1. Backpropagation in a Simple Neural Network

- a) **Dataset**



- b) **Activation Function**

1. Implement function actFun(self, z, type) in three layer neural network.py. This function computes the activation function where z is the net input and type ∈ {'Tanh', 'Sigmoid', 'ReLU'}.

$$\mathrm{Tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$
$$\mathrm{Sigmoid}\sigma(z) = \frac{1}{1 + e^{-z}}$$
$$\mathrm{ReLU}(z) = \max(z, 0)$$

In [2]:
```
import numpy as np
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
    act_func_dict = {
        "tanh":np.tanh(z),
        'sigmoid':1/ (1+np.exp(-z)),
        "relu":np.maximum(0,z)
    }
    if type in act_func_dict:
        return act_func_dict[type]
    return None
```

- b)

2. Derive the derivatives of Tanh, Sigmoid and ReLU

$$
\begin{aligned}
\frac{d}{dz}\tanh(z) &= \frac{(e^z - e^{-z})'(e^z + e^{-z}) - (e^z + e^{-z})'(e^z - e^{-z})}{(e^z + e^{-z})^2} \\
&= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z + e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} \\
&= \frac{(e^z + e^{-z})^2 - (e^z + e^{-z})^2}{(e^z + e^{-z})^2} \\
&= 1 + \tanh^2(z) \\
\frac{d}{dz}\sigma(z) &= \frac{d}{dz}\frac{1}{1 + e^{-z}} \\
&= \frac{0 + e^{-z}}{(1 + e^{-z})^2} \\
&= \frac{e^{-z}}{(1 + e^{-z})^2} \\
&= \sigma(z)(1 - \sigma(z)) \\
\frac{d}{dz}\mathrm{ReLU}(z) &= \begin{cases} 1, z > 0 \\ 0, z < 0 \end{cases}
\end{aligned}
$$

- b)

3. Implement function diff actFun(self, z, type) in three layer neural network.py. This function computes the derivatives of Tanh, Sigmoid and ReLU.

In [4]:
```
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.square(np.tanh(z))
    elif type == 'sigmoid':
        tmp = 1/ (1 + np.exp(-z))
        return tmp * (1 - tmp)
    elif type == "relu":
        return np.where(z>1,1,0)
    else:
        return None
```

- c) **Build the Neural Network**

1. In three layer neural network.py, implement the function feedforward(self, X, actFun). This function builds a 3-layer neural network and computes the two probabilities (self.probs in the code or a2 in Eq. 4), one for class 0 and one for class 1. X is the input data, and actFun is the activation function. You will pass the function actFun you implemented in part b into feedforward(self, X, actFun).

In [5]:
```
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
    exp_scores = np.exp(self.z2)
    self.probs = exp_scores / np.sum(exp_scores,
                                     axis = 1, keepdims=True)
    return None
```

- c)

2. In three layer neural network.py, fill in the function calculate loss(self, X, y). This function computes the loss for prediction of the network. Here X is the input data, and y is the given labels.

In [6]:
```
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
    probs = np.exp(self.z2) / np.sum(np.exp(self.z2),
                                     axis=1, keepdims=True)
    data_loss_single = -np.log(probs[range(num_examples),y])
    data_loss = np.sum(data_loss_single)

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

- d) **Backward Pass - Backpropagation**

1. Derive the following gradients : $\frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_2}, \frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial b_1}$ mathematically.

$$L = -\frac{1}{N}\sum_{n=1}^{N} L(n) = -\frac{1}{N}\sum_{n=1}^{N} y_{(n)} \log \hat{y}_{(n)}$$

$$\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_2} = -\frac{1}{N}\sum_{n=1}^{N} \frac{y_{(n)}}{1-\sigma(z_2)}\frac{\partial \sigma(z_2)}{\partial z_2}$$

$$= -\frac{1}{N}\sum_{n=1}^{N} \frac{y_{(n)}}{1-\sigma(z_2)} - \sigma(z_2)(1-\sigma(z_2)) = -\frac{1}{N}\sum_{n=1}^{N}(y_{(n)} - \sigma(z_2))$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial W_2} = -\frac{1}{N}(y - \sigma(z_2))a_1^T$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial b_2} = -\frac{1}{N}(y - \sigma(z_2))$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial z_1}\frac{\partial z_1}{\partial W_1} = -\frac{1}{N}W_2^T(y \circ (1 - \sigma(z_2)))X^T$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial z_1}\frac{\partial z_1}{\partial b_1} = -\frac{1}{N}W_2^T(y \circ (1 - \sigma(z_2)))$$

- d)

2. In three layer neural network.py, implement the function backprop(self, X, y). Again, X is the input data, and y is the given labels. This function implements backpropagation (i.e., computing the gradients above).

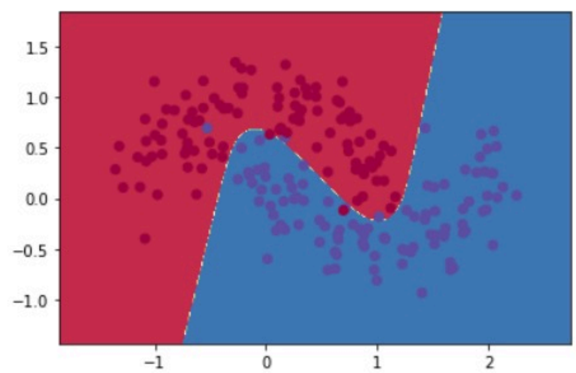
In [7]:
```
def backprop(self, X, y):
    '''
    backprop run backpropagation to compute the gradients used to
    update the parameters in the backward step
    :param X: input data
    :param y: given labels
    :return: dL/dW1, dL/b1, dL/dW2, dL/db2
    '''

    # IMPLEMENT YOUR BACKPROP HERE
    num_examples = len(X)
    delta3 = self.probs
    delta3[range(num_examples),y] -=1

    # dW2 = dL/dW2
    # db2 = dL/db2
    # dW1 = dL/dW1
    # db1 = dL/db1
    dW2 = np.dot(self.a1.T, delta3)
    db2 = np.sum(delta3, axis=0, keepdims=True)
    diff = self.diff_actFun(self.z1, type=self.actFun_type)
    delta2 = np.dot(delta3, self.W2.T) * diff
    dW1 = np.dot(X.T, delta2)
    db1 = np.sum(delta2, axis=0, keepdims=False)
    return dW1, dW2, db1, db2
```

- e) **Time to Have Fun - Training!**

1. Train the network using different activation functions (Tanh, Sigmoid and ReLU). Describe and explain the differences that you observe. Include the figures generated in your report. In order to train the network, uncomment the main() function in three layer neural network.py, take out the following lines, and run three layer neural network.py.

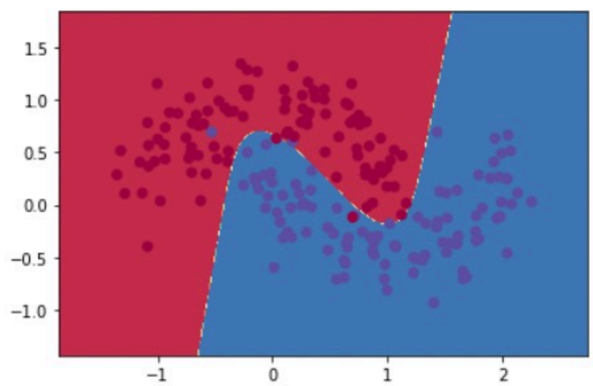
- activation function: tanh

```
Loss after iteration 0: 0.432387
Loss after iteration 1000: 0.068947
Loss after iteration 2000: 0.068943
Loss after iteration 3000: 0.070752
Loss after iteration 4000: 0.070748
Loss after iteration 5000: 0.070751
Loss after iteration 6000: 0.070754
Loss after iteration 7000: 0.070756
Loss after iteration 8000: 0.070757
Loss after iteration 9000: 0.070758
Loss after iteration 10000: 0.070758
Loss after iteration 11000: 0.070758
Loss after iteration 12000: 0.070758
Loss after iteration 13000: 0.070758
Loss after iteration 14000: 0.070758
Loss after iteration 15000: 0.070758
Loss after iteration 16000: 0.070758
Loss after iteration 17000: 0.070758
Loss after iteration 18000: 0.070758
Loss after iteration 19000: 0.070758
```



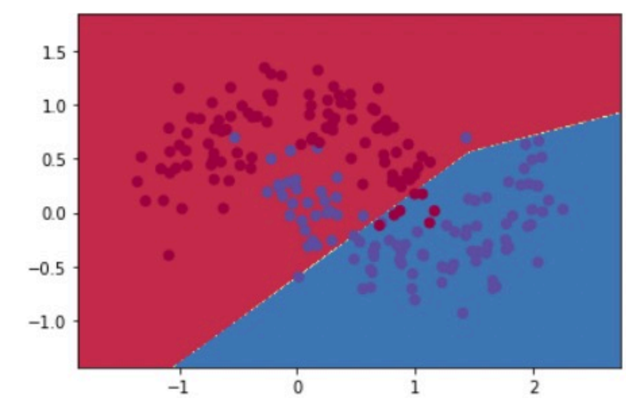
- activation function: sigmoid

```
Loss after iteration 0: 0.628571
Loss after iteration 1000: 0.088431
Loss after iteration 2000: 0.079598
Loss after iteration 3000: 0.078604
Loss after iteration 4000: 0.078330
Loss after iteration 5000: 0.078233
Loss after iteration 6000: 0.078192
Loss after iteration 7000: 0.078174
Loss after iteration 8000: 0.078166
Loss after iteration 9000: 0.078161
Loss after iteration 10000: 0.078159
Loss after iteration 11000: 0.078158
Loss after iteration 12000: 0.078157
Loss after iteration 13000: 0.078156
Loss after iteration 14000: 0.078156
Loss after iteration 15000: 0.078156
Loss after iteration 16000: 0.078156
Loss after iteration 17000: 0.078156
Loss after iteration 18000: 0.078156
Loss after iteration 19000: 0.078155
```



- activation function: relu

```
Loss after iteration 0: 0.550123
Loss after iteration 1000: 0.317963
Loss after iteration 2000: 0.325877
Loss after iteration 3000: 0.331841
Loss after iteration 4000: 0.331677
Loss after iteration 5000: 0.332447
Loss after iteration 6000: 0.333270
Loss after iteration 7000: 0.333799
Loss after iteration 8000: 0.334405
Loss after iteration 9000: 0.334646
Loss after iteration 10000: 0.334946
Loss after iteration 11000: 0.335177
Loss after iteration 12000: 0.335134
Loss after iteration 13000: 0.335445
Loss after iteration 14000: 0.335343
Loss after iteration 15000: 0.335370
Loss after iteration 16000: 0.335376
Loss after iteration 17000: 0.335560
Loss after iteration 18000: 0.335826
Loss after iteration 19000: 0.335648
```



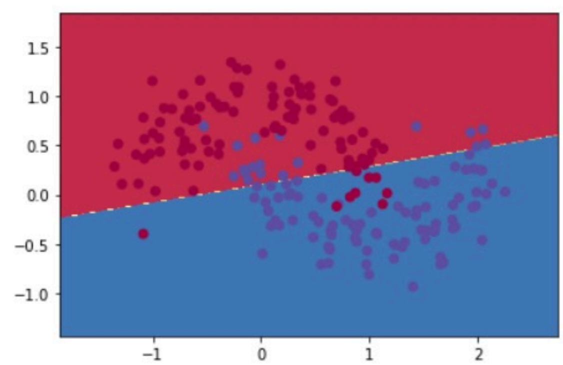
- Observation and Explain: It seems that Tanh and Sigmoid produce similar predict result with lower false compared with Relu. One interesting thing is that, the relu activation function have the vanishing gradient problem, maybe because it's nof differentialbe at 0.

2. Increase the number of hidden units (nn hidden dim) and retrain the network using Tanh as the activation function. Describe and explain the differences that you observe. Include the figures generated in your report.

Tanh

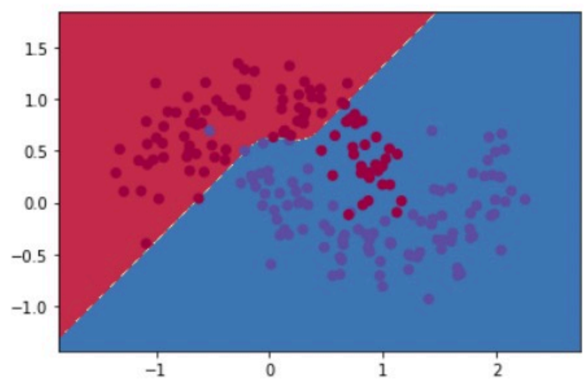
- nn_hidden_dim =1

```
Loss after iteration 0: 0.567280
Loss after iteration 1000: 0.333524
Loss after iteration 2000: 0.333505
Loss after iteration 3000: 0.333489
Loss after iteration 4000: 0.333476
Loss after iteration 5000: 0.333466
Loss after iteration 6000: 0.333457
Loss after iteration 7000: 0.333450
Loss after iteration 8000: 0.333444
Loss after iteration 9000: 0.333439
Loss after iteration 10000: 0.333435
Loss after iteration 11000: 0.333432
Loss after iteration 12000: 0.333430
Loss after iteration 13000: 0.333428
Loss after iteration 14000: 0.333426
Loss after iteration 15000: 0.333424
Loss after iteration 16000: 0.333423
Loss after iteration 17000: 0.333422
Loss after iteration 18000: 0.333421
Loss after iteration 19000: 0.333421
```



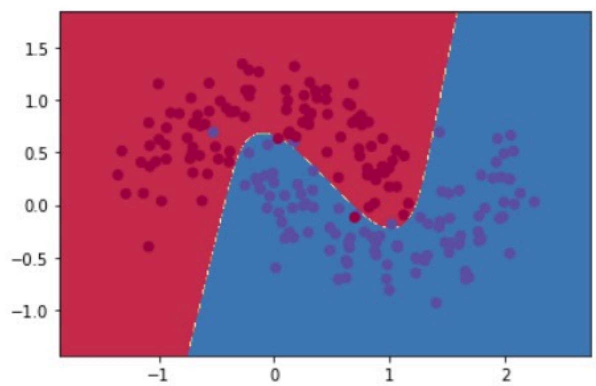
- nn_hidden_dim =2

```
Loss after iteration 0: 0.546544
Loss after iteration 1000: 0.324081
Loss after iteration 2000: 0.321476
Loss after iteration 3000: 0.320915
Loss after iteration 4000: 0.324282
Loss after iteration 5000: 0.324079
Loss after iteration 6000: 0.319419
Loss after iteration 7000: 0.323011
Loss after iteration 8000: 0.325401
Loss after iteration 9000: 0.324052
Loss after iteration 10000: 0.326436
Loss after iteration 11000: 0.349009
Loss after iteration 12000: 0.290238
Loss after iteration 13000: 0.322130
Loss after iteration 14000: 0.324169
Loss after iteration 15000: 0.326871
Loss after iteration 16000: 0.368909
Loss after iteration 17000: 0.313245
Loss after iteration 18000: 0.322188
Loss after iteration 19000: 0.321829
```



- nn_hidden_dim =3

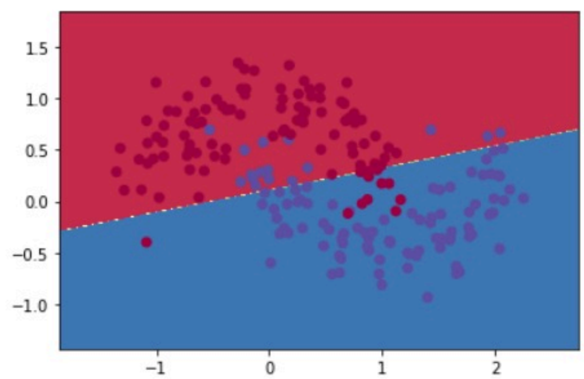
```
Loss after iteration 0: 0.432387
Loss after iteration 1000: 0.068947
Loss after iteration 2000: 0.068943
Loss after iteration 3000: 0.070752
Loss after iteration 4000: 0.070748
Loss after iteration 5000: 0.070751
Loss after iteration 6000: 0.070754
Loss after iteration 7000: 0.070756
Loss after iteration 8000: 0.070757
Loss after iteration 9000: 0.070758
Loss after iteration 10000: 0.070758
Loss after iteration 11000: 0.070758
Loss after iteration 12000: 0.070758
Loss after iteration 13000: 0.070758
Loss after iteration 14000: 0.070758
Loss after iteration 15000: 0.070758
Loss after iteration 16000: 0.070758
Loss after iteration 17000: 0.070758
Loss after iteration 18000: 0.070758
Loss after iteration 19000: 0.070758
```



Sigmoid

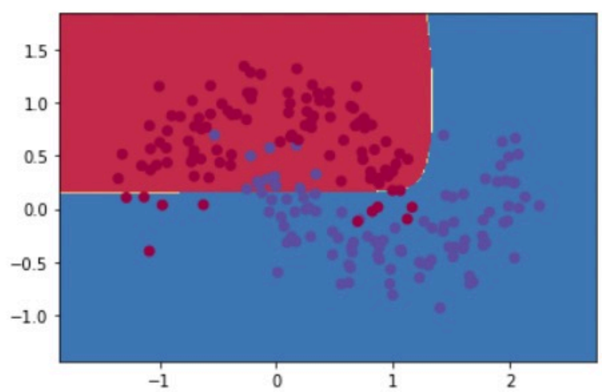
- nn_hidden_dim =1

```
Loss after iteration 0: 0.649653
Loss after iteration 1000: 0.305516
Loss after iteration 2000: 0.310857
Loss after iteration 3000: 0.310835
Loss after iteration 4000: 0.310822
Loss after iteration 5000: 0.310811
Loss after iteration 6000: 0.310802
Loss after iteration 7000: 0.310795
Loss after iteration 8000: 0.310790
Loss after iteration 9000: 0.310785
Loss after iteration 10000: 0.310781
Loss after iteration 11000: 0.310778
Loss after iteration 12000: 0.310775
Loss after iteration 13000: 0.310773
Loss after iteration 14000: 0.310771
Loss after iteration 15000: 0.310770
Loss after iteration 16000: 0.310769
Loss after iteration 17000: 0.310768
Loss after iteration 18000: 0.310767
Loss after iteration 19000: 0.310766
```



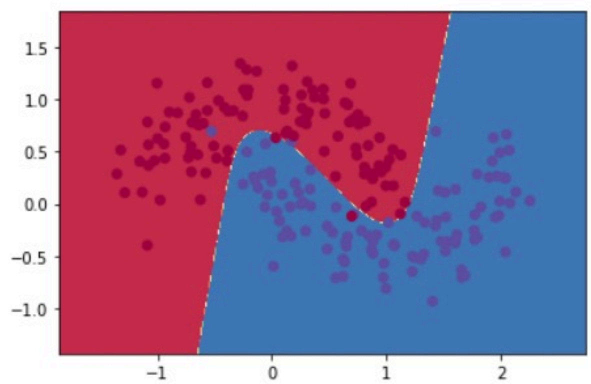
- nn_hidden_dim =2

```
Loss after iteration 0: 0.785921
Loss after iteration 1000: 0.303810
Loss after iteration 2000: 0.298855
Loss after iteration 3000: 0.290810
Loss after iteration 4000: 0.274627
Loss after iteration 5000: 0.264113
Loss after iteration 6000: 0.259089
Loss after iteration 7000: 0.257833
Loss after iteration 8000: 0.257245
Loss after iteration 9000: 0.256922
Loss after iteration 10000: 0.256731
Loss after iteration 11000: 0.256612
Loss after iteration 12000: 0.256535
Loss after iteration 13000: 0.256484
Loss after iteration 14000: 0.256450
Loss after iteration 15000: 0.256426
Loss after iteration 16000: 0.256410
Loss after iteration 17000: 0.256611
Loss after iteration 18000: 0.256548
Loss after iteration 19000: 0.256526
```



- nn_hidden_dim =3

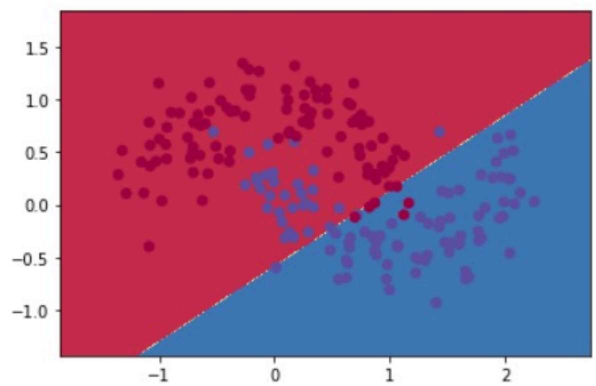
```
Loss after iteration 0: 0.628571
Loss after iteration 1000: 0.088431
Loss after iteration 2000: 0.079598
Loss after iteration 3000: 0.078604
Loss after iteration 4000: 0.078330
Loss after iteration 5000: 0.078233
Loss after iteration 6000: 0.078192
Loss after iteration 7000: 0.078174
Loss after iteration 8000: 0.078166
Loss after iteration 9000: 0.078161
Loss after iteration 10000: 0.078159
Loss after iteration 11000: 0.078158
Loss after iteration 12000: 0.078157
Loss after iteration 13000: 0.078156
Loss after iteration 14000: 0.078156
Loss after iteration 15000: 0.078156
Loss after iteration 16000: 0.078156
Loss after iteration 17000: 0.078156
Loss after iteration 18000: 0.078156
Loss after iteration 19000: 0.078155
```



Relu

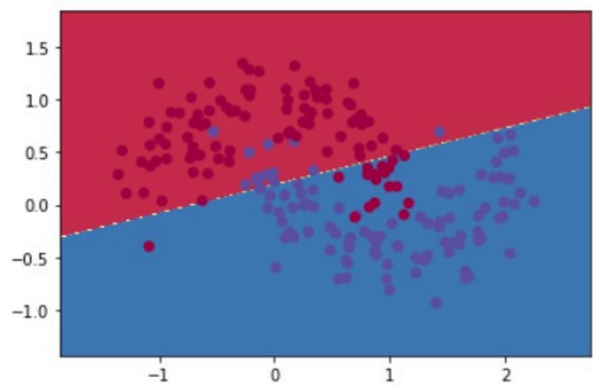
- nn_hidden_dim =1

```
Loss after iteration 0: 0.623982
Loss after iteration 1000: 0.402943
Loss after iteration 2000: 0.404122
Loss after iteration 3000: 0.405079
Loss after iteration 4000: 0.405690
Loss after iteration 5000: 0.405725
Loss after iteration 6000: 0.405743
Loss after iteration 7000: 0.405780
Loss after iteration 8000: 0.405809
Loss after iteration 9000: 0.405820
Loss after iteration 10000: 0.405819
Loss after iteration 11000: 0.405836
Loss after iteration 12000: 0.405819
Loss after iteration 13000: 0.405840
Loss after iteration 14000: 0.405835
Loss after iteration 15000: 0.405835
Loss after iteration 16000: 0.405824
Loss after iteration 17000: 0.405830
Loss after iteration 18000: 0.405836
Loss after iteration 19000: 0.405838
```



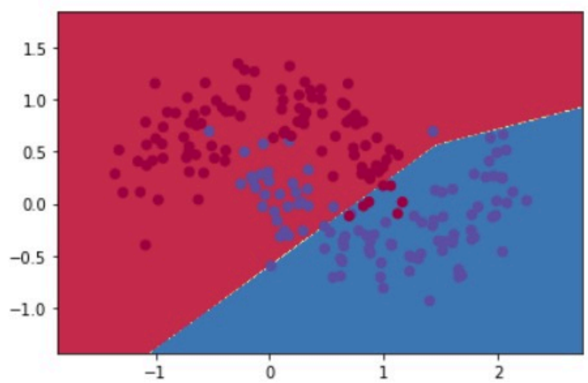
- nn_hidden_dim =2

```
Loss after iteration 0: 0.721074
Loss after iteration 1000: 0.327940
Loss after iteration 2000: 0.330377
Loss after iteration 3000: 0.330188
Loss after iteration 4000: 0.330001
Loss after iteration 5000: 0.329718
Loss after iteration 6000: 0.330382
Loss after iteration 7000: 0.327239
Loss after iteration 8000: 0.327867
Loss after iteration 9000: 0.330256
Loss after iteration 10000: 0.327246
Loss after iteration 11000: 0.327597
Loss after iteration 12000: 0.327470
Loss after iteration 13000: 0.327319
Loss after iteration 14000: 0.327906
Loss after iteration 15000: 0.327576
Loss after iteration 16000: 0.329724
Loss after iteration 17000: 0.328209
Loss after iteration 18000: 0.327518
Loss after iteration 19000: 0.329733
```



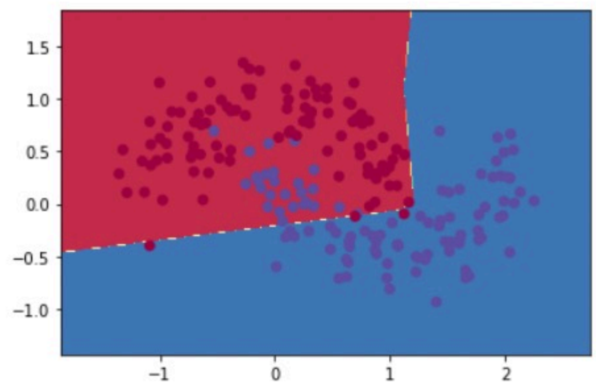
- nn_hidden_dim =3

```
Loss after iteration 0: 0.550123
Loss after iteration 1000: 0.317963
Loss after iteration 2000: 0.325877
Loss after iteration 3000: 0.331841
Loss after iteration 4000: 0.331677
Loss after iteration 5000: 0.332447
Loss after iteration 6000: 0.333270
Loss after iteration 7000: 0.333799
Loss after iteration 8000: 0.334405
Loss after iteration 9000: 0.334646
Loss after iteration 10000: 0.334946
Loss after iteration 11000: 0.335177
Loss after iteration 12000: 0.335134
Loss after iteration 13000: 0.335445
Loss after iteration 14000: 0.335343
Loss after iteration 15000: 0.335370
Loss after iteration 16000: 0.335376
Loss after iteration 17000: 0.335560
Loss after iteration 18000: 0.335826
Loss after iteration 19000: 0.335648
```



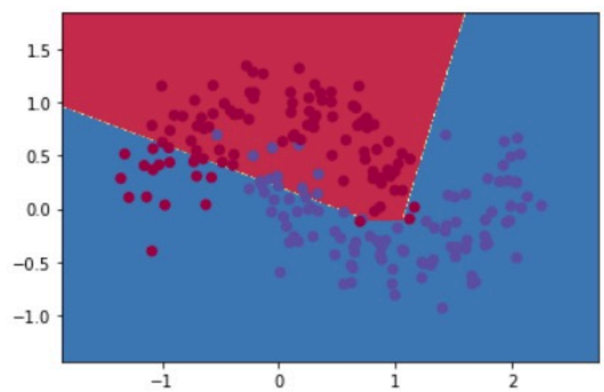
- nn_hidden_dim =4

```
Loss after iteration 0: 0.560801
Loss after iteration 1000: 0.257142
Loss after iteration 2000: 0.264000
Loss after iteration 3000: 0.263616
Loss after iteration 4000: 0.263448
Loss after iteration 5000: 0.264686
Loss after iteration 6000: 0.268400
Loss after iteration 7000: 0.270379
Loss after iteration 8000: 0.275302
Loss after iteration 9000: 0.282313
Loss after iteration 10000: 0.292010
Loss after iteration 11000: 0.303117
Loss after iteration 12000: 0.313895
Loss after iteration 13000: 0.318449
Loss after iteration 14000: 0.328910
Loss after iteration 15000: 0.337974
Loss after iteration 16000: 0.344730
Loss after iteration 17000: 0.347879
Loss after iteration 18000: 0.347880
Loss after iteration 19000: 0.347329
```



- nn_hidden_dim =5

```
Loss after iteration 0: 1.568747
Loss after iteration 1000: 0.317275
Loss after iteration 2000: 0.314504
Loss after iteration 3000: 0.316509
Loss after iteration 4000: 0.319858
Loss after iteration 5000: 0.321658
Loss after iteration 6000: 0.322227
Loss after iteration 7000: 0.321963
Loss after iteration 8000: 0.321891
Loss after iteration 9000: 0.322287
Loss after iteration 10000: 0.323017
Loss after iteration 11000: 0.324039
Loss after iteration 12000: 0.325540
Loss after iteration 13000: 0.327382
Loss after iteration 14000: 0.329511
Loss after iteration 15000: 0.332208
Loss after iteration 16000: 0.335358
Loss after iteration 17000: 0.338788
Loss after iteration 18000: 0.343055
Loss after iteration 19000: 0.347604
```



- Observation and Explain: With the increase of nn_hidden_dim, the performance of model getter better for 3 differenct activation function. It is because when the value is too small, the model will underfit the data. But when the value is too big, the model will overfit the data. So if we can find the best number for nn_hidden_dim, it will be better.

- f) Even More Fun - Training a Deeper Network!!!

Write your own n_layer_neural_network.py that builds and trains a neural network of n layers. Solution in n_layer_neural_network.py. The different part of the coding as following: funcions of feedworwar, backprob, calculate_loss, fit_model

In [ ]:

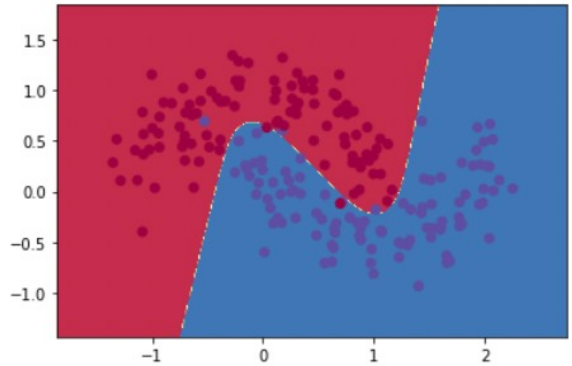
```
def feedforward(self, X, actFun):
    '''
    feedforward builds a 3-layer neural network and
    computes the two probabilities,
    one for class 0 and one for class 1
    :param X: input data
    :param actFun: activation function
```

```python
    :return:
    '''

    # YOU IMPLEMENT YOUR feedforward HERE

    self.z = []
    self.a = []
    for i in range(len(self.W)):
        if i == 0:
            self.z.append(np.dot(X, self.W[i]) + self.b[i])
        else:
            self.z.append(np.dot(self.a[i-1],
                                 self.W[i]) + self.b[i])
        if i != len(self.W) - 1:
            self.a.append(actFun(self.z[i]))
    exp_scores = np.exp(self.z[len(self.z)-1])
    self.probs = exp_scores / np.sum(exp_scores,
                                     axis=1, keepdims=True)
    return None

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

    probs = np.exp(self.z[len(self.z)-1]) / \
            np.sum(np.exp(self.z[len(self.z)-1]),
                   axis=1, keepdims=True)
    data_loss_single = -np.log(probs[range(num_examples), y])
    data_loss = np.sum(data_loss_single)

    # Add regulatization term to loss (optional)
    W_sum = 0
    for i in len(self.W):
        W_sum += np.sum(np.square(self.W[i]))
    data_loss += self.reg_lambda / 2 * W_sum
    return (1. / num_examples) * data_loss

def predict(self, X):
    '''
    predict infers the label of a given data point X
    :param X: input data
    :return: label inferred
    '''
    self.feedforward(X, lambda x: self.actFun(x,
```

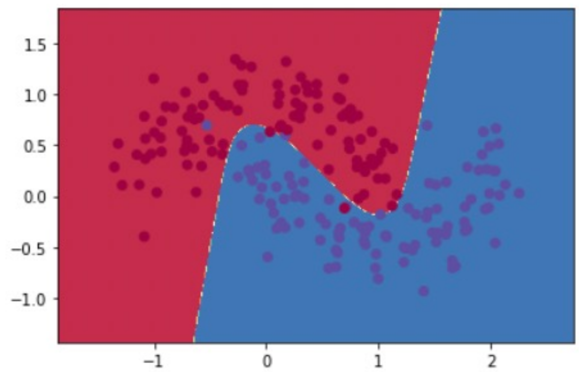
```
        self.feedforward(X, lambda x: self.actFun(x,
                                    type=self.actFun_type))
        return np.argmax(self.probs, axis=1)

    def backprop(self, X, y):
        '''
        backprop implements 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, ... dL/dn,
        dL/bn in two lists
        '''

        # IMPLEMENT YOUR BACKPROP HERE
        num_examples = len(X)
        delta = self.probs
        delta[range(num_examples), y] -= 1

        dW = []
        db = []
        for i in range(len(self.z)):
            index = len(self.z) - i - 1
            if index != 0:
                dW.insert(0, np.dot(self.a[index - 1].T,
                                    delta))
                db.insert(0, np.sum(delta,
                                    axis=0, keepdims=True))
                delta = np.dot(delta, self.W[index].T) * \
                        self.diff_actFun(self.z[index-1],
                                         type=self.actFun_type)
            else:
                dW.insert(0, np.dot(X.T, delta))
                db.insert(0, np.sum(delta, axis=0, keepdims=False))

        return dW, db

    def fit_model(self, X, y, epsilon=0.01,
                  num_passes=20000, print_loss=True):
        '''
        fit_model uses backpropagation to train the network
        :param X: input data
        :param y: given labels
        :param num_passes: the number of times that the
        algorithm runs through the whole dataset
        :param print_loss: print the loss or not
        :return:
        '''
        # Gradient descent.
        for i in range(0, num_passes):
            # Forward propagation
            self.feedforward(X, lambda x: self.actFun(x,
                                    type=self.actFun_type))
            # Backpropagation
            dW, db = self.backprop(X, y)
```

```
        dW, db = self.backprop(X, y)

        # Add regularization terms (b1 and b2 don't have
        # regularization terms)
        for i in range(len(dW)):
            # print(dW[i].shape)
            # print(self.W[i].shape)
            dW[i] += self.reg_lambda * self.W[i]

        # Gradient descent parameter update
        for i in range(len(self.W)):
            self.W[i] += -epsilon * dW[i]
            self.b[i] += -epsilon * db[i]

        # Optionally print the loss.
        # This is expensive because it uses the whole
        # dataset, so we don't want to do it too often.
        if print_loss and i % 1000 == 0:
```

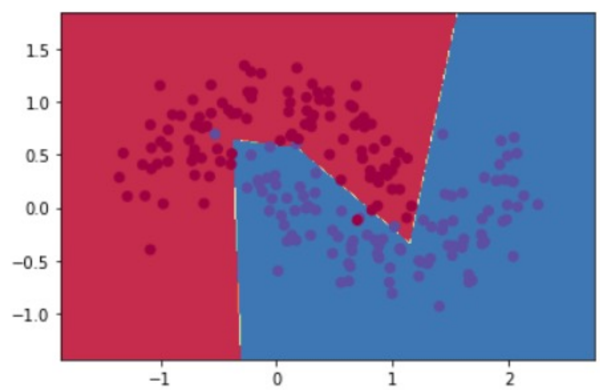
- run the class of DeepNeuralNetwork with 3 different actitation function
- Tanh



- Sigmoid



- Relu



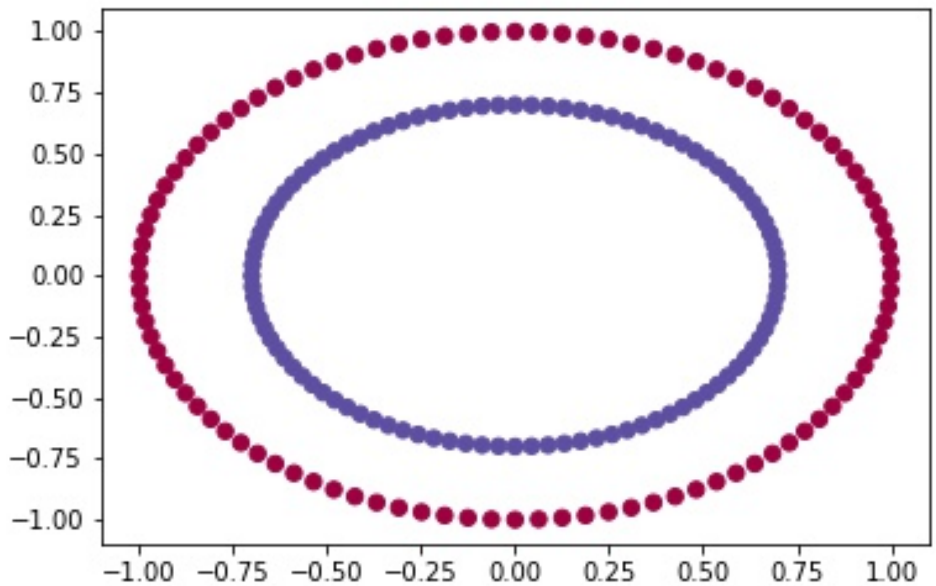
The observation and explanation: We can see , with same nn_hidden_dims = 3, the prediction of relu with DeepNeuralNetwork class works better than the NeuralNetwork class.

Type *Markdown* and LaTeX: α^2

1.f change dataset When the dataset is changed to be make_circles:

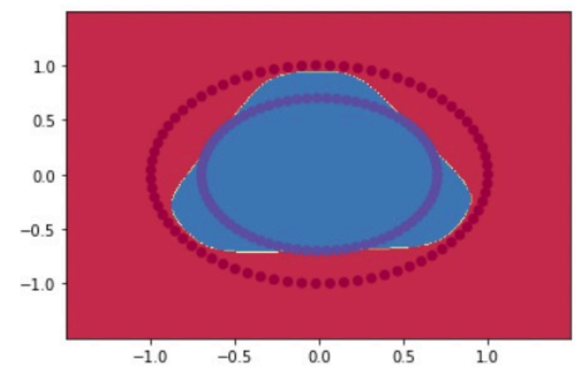
In [3]:
```
from sklearn import datasets
X, y = datasets.make_circles(n_samples=100,
    shuffle=True, noise=None, random_state=None, factor=0.8)
```

- New data set with make_circles



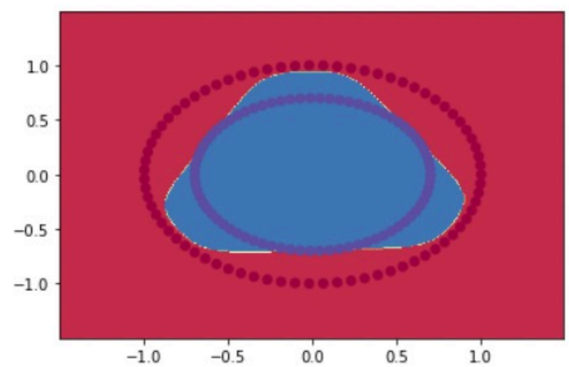
- with data size of 100

```
Loss after iteration 0: 0.696409
Loss after iteration 1000: 0.233862
Loss after iteration 2000: 1.350100
Loss after iteration 3000: 0.169909
Loss after iteration 4000: 0.191831
Loss after iteration 5000: 0.201922
Loss after iteration 6000: 0.256229
Loss after iteration 7000: 0.225861
Loss after iteration 8000: 0.187237
Loss after iteration 9000: 0.242718
Loss after iteration 10000: 0.215607
Loss after iteration 11000: 0.226070
Loss after iteration 12000: 0.178402
Loss after iteration 13000: 0.180204
Loss after iteration 14000: 0.145848
Loss after iteration 15000: 0.223959
Loss after iteration 16000: 0.245700
Loss after iteration 17000: 0.263645
Loss after iteration 18000: 0.160721
Loss after iteration 19000: 2.721298
```



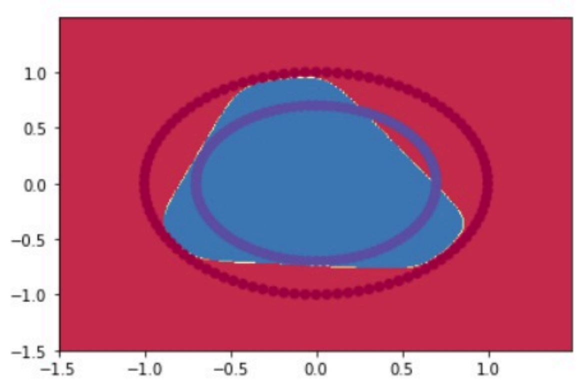
- with data size of 100

```
Loss after iteration 0: 0.696409
Loss after iteration 1000: 0.233862
Loss after iteration 2000: 1.350100
Loss after iteration 3000: 0.169909
Loss after iteration 4000: 0.191831
Loss after iteration 5000: 0.201922
Loss after iteration 6000: 0.256229
Loss after iteration 7000: 0.225861
Loss after iteration 8000: 0.187237
Loss after iteration 9000: 0.242718
Loss after iteration 10000: 0.215607
Loss after iteration 11000: 0.226070
Loss after iteration 12000: 0.178402
Loss after iteration 13000: 0.180204
Loss after iteration 14000: 0.145848
Loss after iteration 15000: 0.223959
Loss after iteration 16000: 0.245700
Loss after iteration 17000: 0.263645
Loss after iteration 18000: 0.160721
Loss after iteration 19000: 2.721298
```



- with data size of 200

```
Loss after iteration 0: 0.693833
Loss after iteration 1000: 0.248702
Loss after iteration 2000: 0.256833
Loss after iteration 3000: 0.223822
Loss after iteration 4000: 0.251005
Loss after iteration 5000: 0.243078
Loss after iteration 6000: 0.229056
Loss after iteration 7000: 0.231975
Loss after iteration 8000: 0.272167
Loss after iteration 9000: 0.260909
Loss after iteration 10000: 1.742223
Loss after iteration 11000: 0.256263
Loss after iteration 12000: 0.236326
Loss after iteration 13000: 0.248164
Loss after iteration 14000: 0.206065
Loss after iteration 15000: 0.221233
Loss after iteration 16000: 0.230755
Loss after iteration 17000: 0.213492
Loss after iteration 18000: 0.285917
Loss after iteration 19000: 0.265185
```



- We can see, after I change the data from Make_Moons to Make_Circles in Sklearn, my model of neuralNetwork and help to predict the data. It worked well when the data size is only 100, but get worse when I increase the data size to 150 and 200.

Task2 : Training a Simple Deep Convolutional Network on MNIST

- a) Build and Train a 4-layer DCN

2. Complete functions weight variable(shape), bias variable(shape), conv2d(x, W), max pool 2x2(x) in dcn mnist.py. The first two functions initialize the weights and biases in the network, and the last two functions will implement convolution and max-pooling operators, respectively.

In [ ]:
```
def weight_variable(shape):
    result = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(result)

def bias_variable(shape):
    result = tf.constant(0.1, shape = shape)
    return tf.Variable(result)

def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1,1,1,1], PADDING = 'SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1,2,2,1],
                          strides=[1,2,2,1], padding='SAME')
```

3. Build your network: In dcn mnist.py, you will see ”FILL IN THE CODE BELOW TO BUILD YOUR NETWORK”. Complete the following sections in dcn mnist.py: placeholders for input data and input labeles, first convolutional layer, convolutional layer, densely connected layer, dropout, softmax.

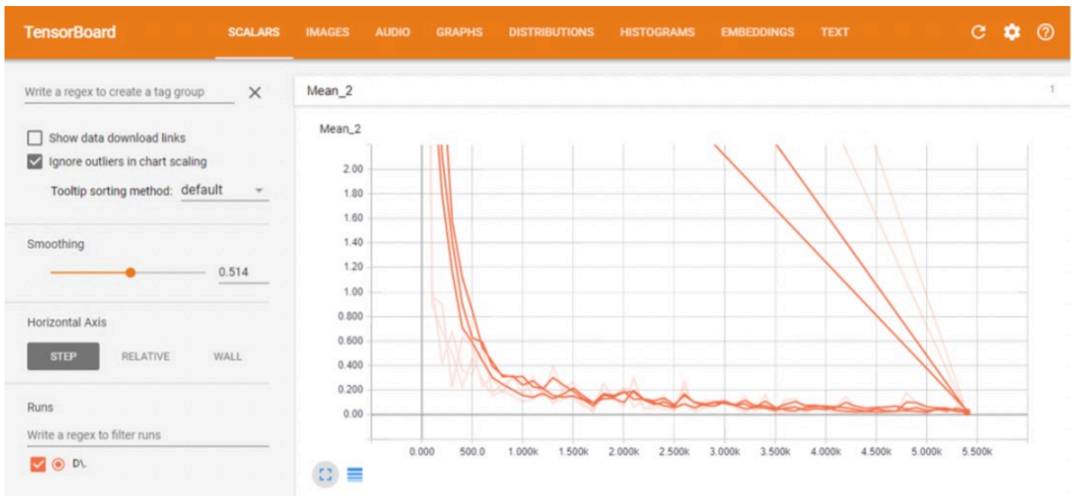
In [ ]:
```python
# FILL IN THE CODE BELOW TO BUILD YOUR NETWORK

# placeholders for input data and input labeles
x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.int64, [None])

# 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 * 64,1204])
b_fc1 = bias_variable([1204])
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([1204,10])
b_fc2 = bias_variable([10])
y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

4. Set up Training: In dcn mnist.py, you will see ”FILL IN THE FOLLOWING CODE TO SET UP THE TRAINING”. Complete section setup training in dcn_mnist.py.

In [ ]:
```python
# FILL IN THE FOLLOWING CODE TO SET UP THE TRAINING

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

5. Run Training: Study the rest of dcn mnist.py. Notice that, different from the tutorial Deep MNIST for Expert, I use the summary operation (e.g. summary op, summary writer, ...) to monitor the training. Here, I only monitor the training loss value. Now, run dcn mnist.py. What is the final test accuracy of your network? Note that I set the batch size to 50, and to save time, I set the max step to only 5500. Batch size is the number of MNIST images that are sent to the DCN at each iteration, and max step is the maximum number of training iterations. max step = 5500 means the training will stop after 5500 iterations no matter what. When batch size is 50, 5500 iterations is equivalent to 5 epochs. Remind that, in each epoch, the DCN will see the whole training set once. In this case, since there are 55K training images, each epoch is consisted of 55K/50 = 1100 iterations.

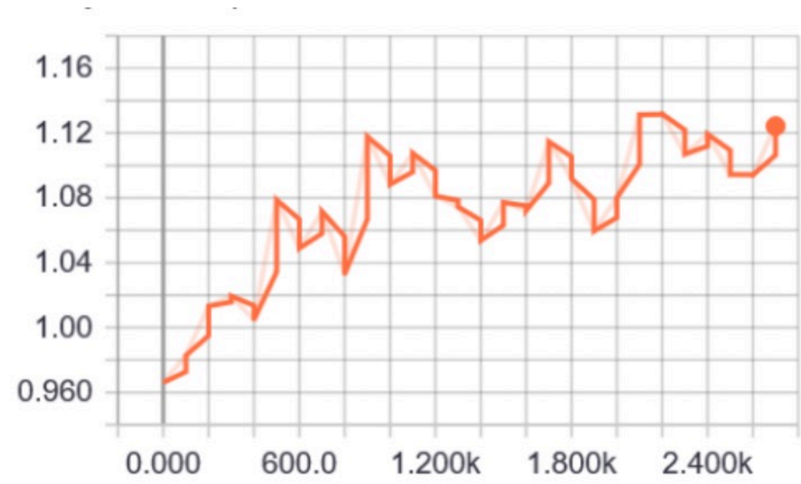
- The final test accuracy for this run is 0.9865.



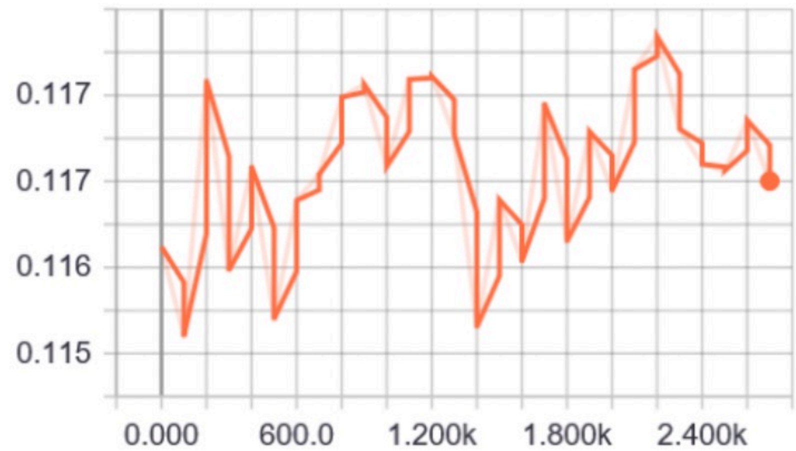
- b) More on Visualizing Your Training

In part (a) of this problem, you only monitor the training loss during the training. Now, let's visualize your training more! Study dcn mnist.py and this tutorial TensorBoard: Visualizing Learning to learn how to monitor a set of variables during the training. Then, modify dcn mnist.py so that you can monitor the statistics (min, max, mean, standard deviation, histogram) of the following terms after each 100 iterations: weights, biases, net inputs at each layer , activations after ReLU at each layer, activations after Max-Pooling at each layer. Also monitor the test and validation error after each 1100 iterations (equivalently, after each epoch). Run the training again and visualize the monitored terms in TensorBoard. Include the resultant figures in your report.

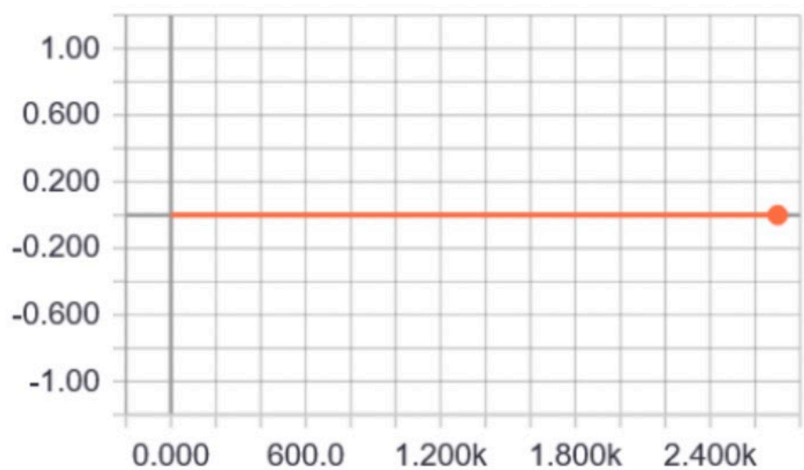
- ConvLayer1 max for Wx_plust_b



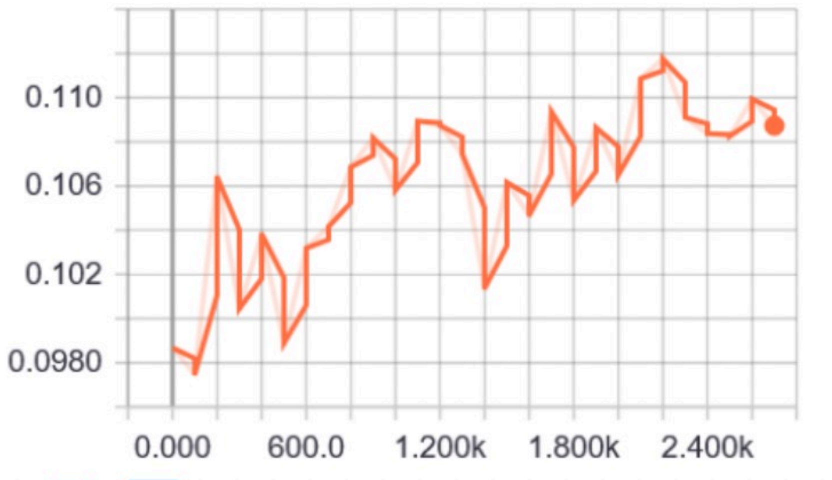
- ConvLayer1 min for Wx_plust_b



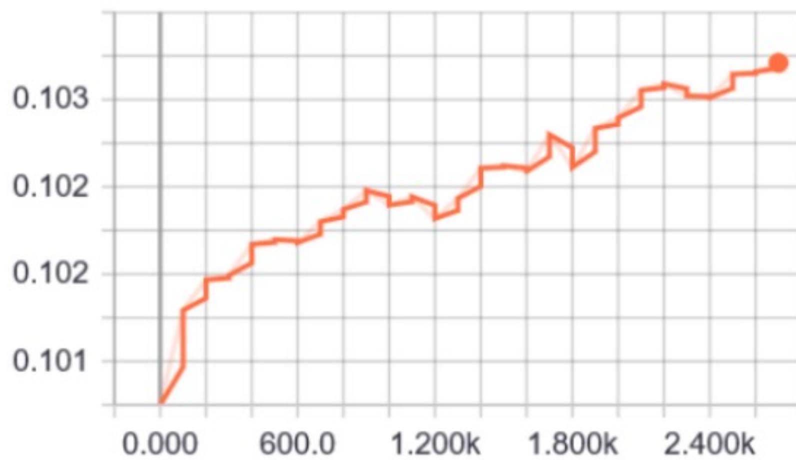
- ConvLayer1 mean for Wx_plust_b



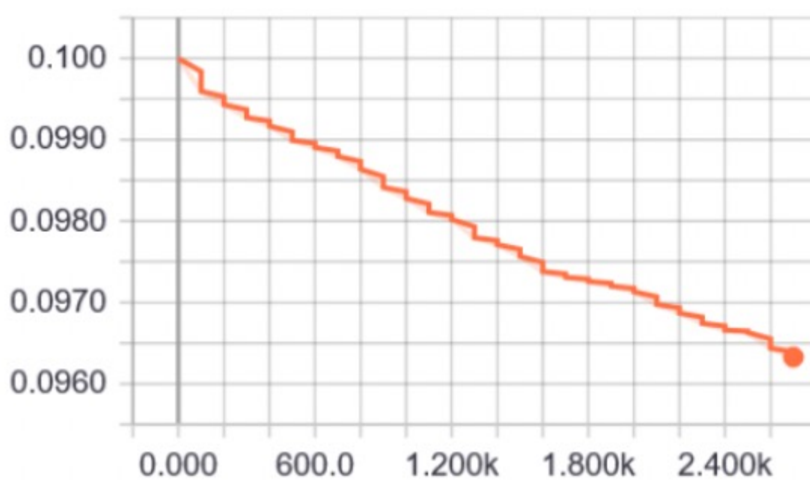
- ConvLayer1 stddev for Wx_plust_b



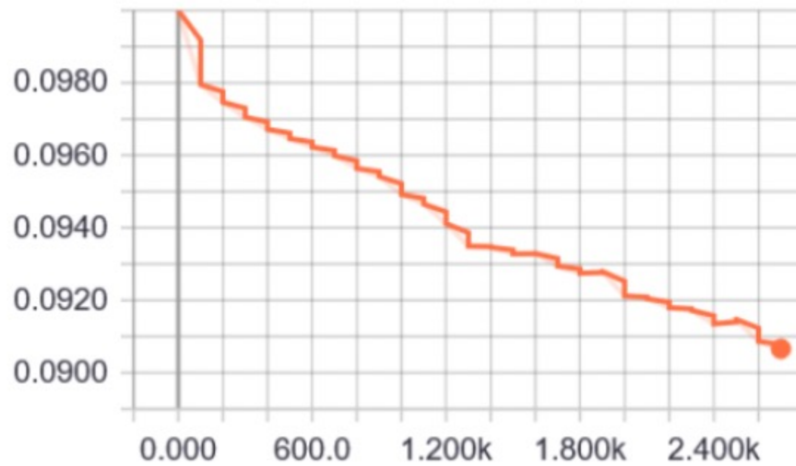
- ConvLayer1 max for biases



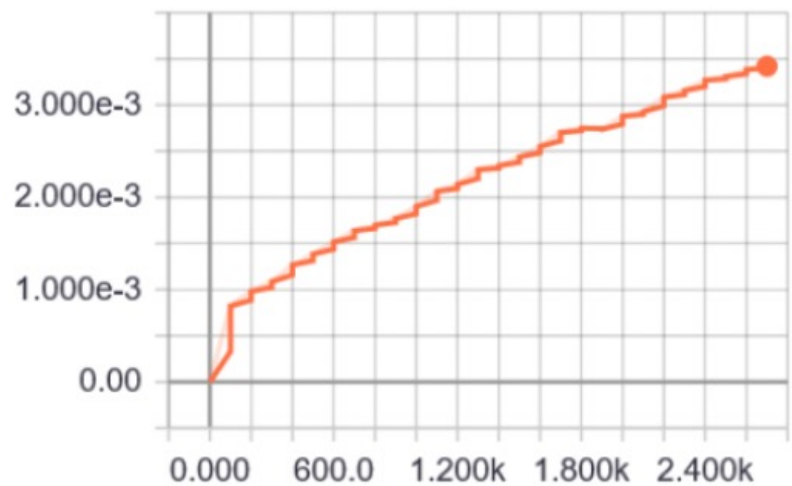
- ConvLayer1 min for biases



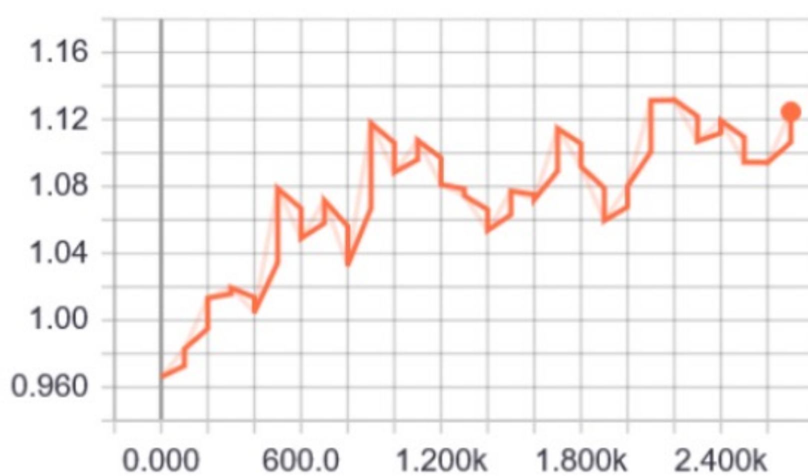
- ConvLayer1 mean for biases



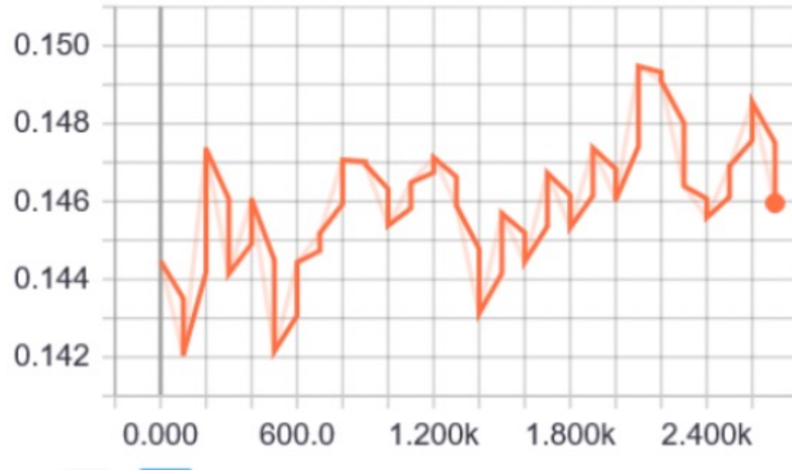
- ConvLayer1 stddev for biases



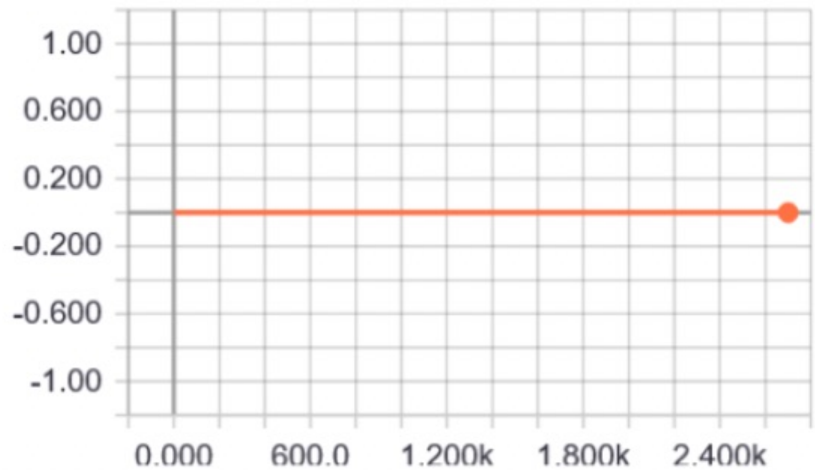
- ConvLayer1 max for max_pool



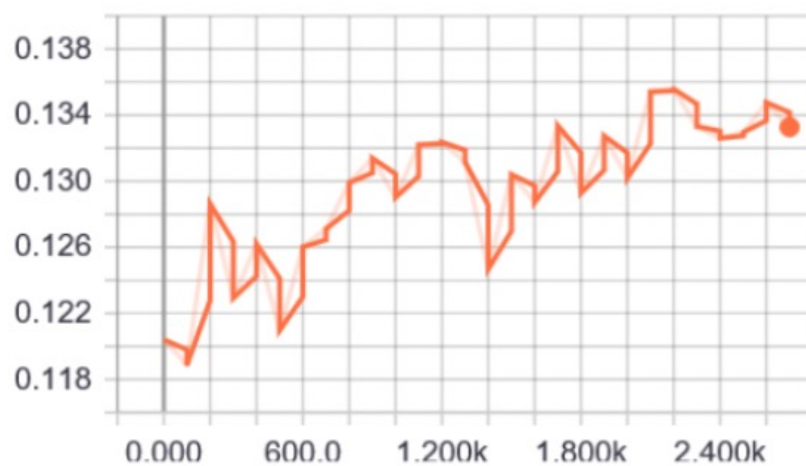
- ConvLayer1 min for max_pool



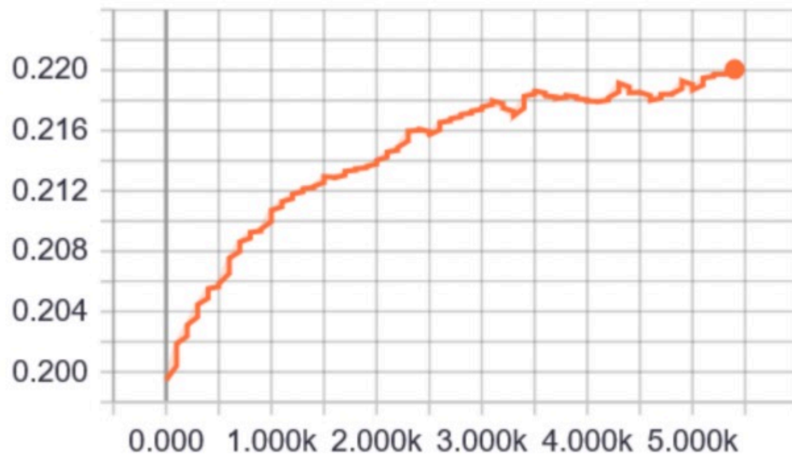
- ConvLayer1 mean for max_pool



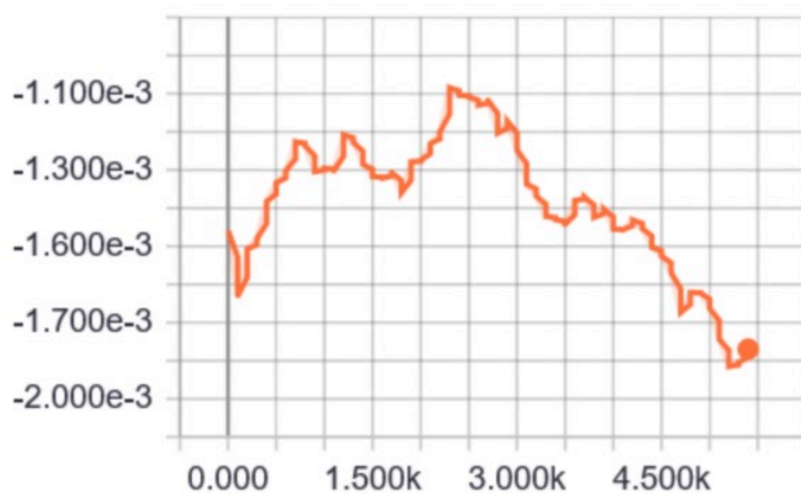
- ConvLayer1 stddev for max_pool



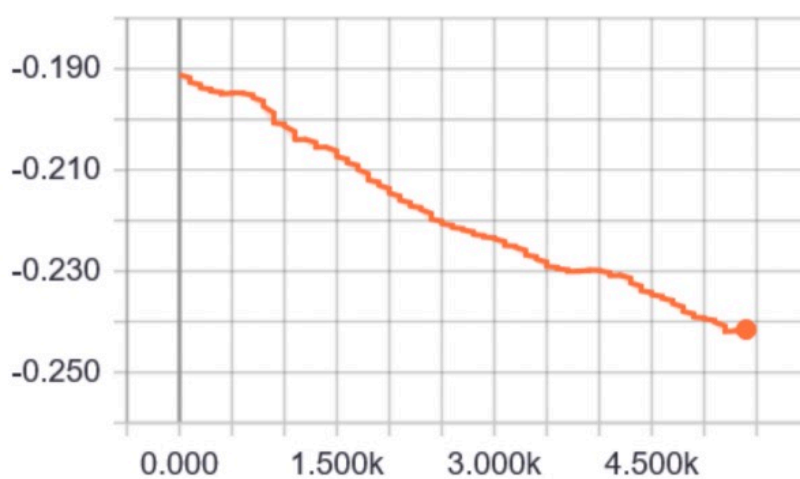
- ConvLayer1 max for weights



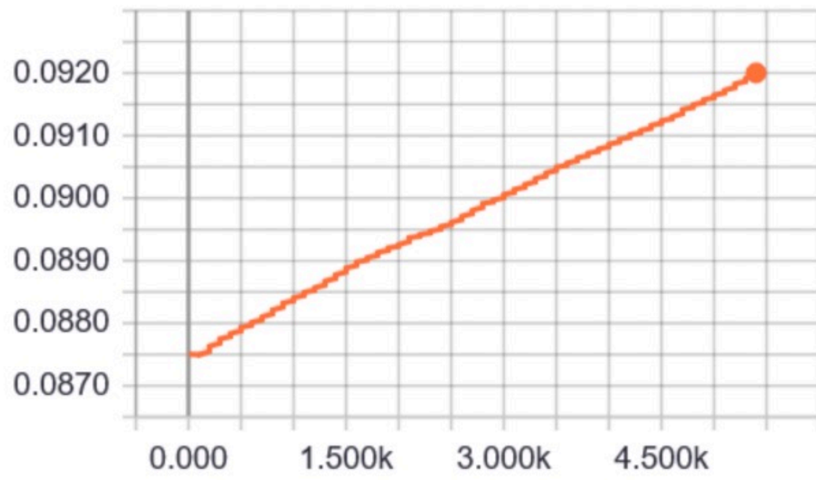
- ConvLayer1 min for weights



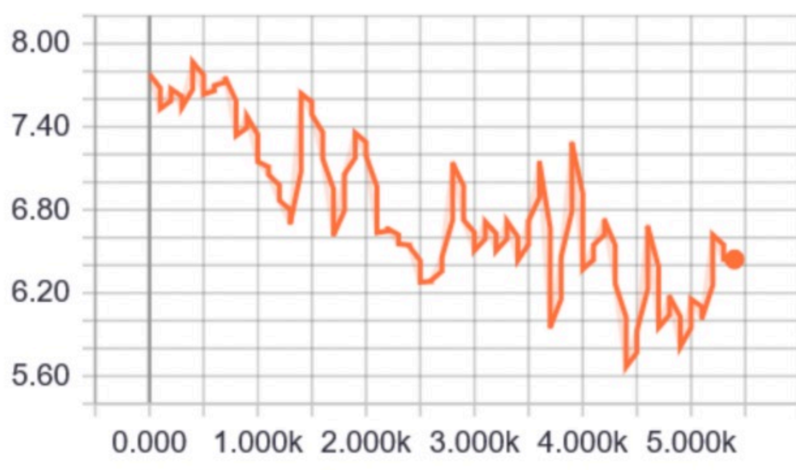
- ConvLayer1 mean for weights



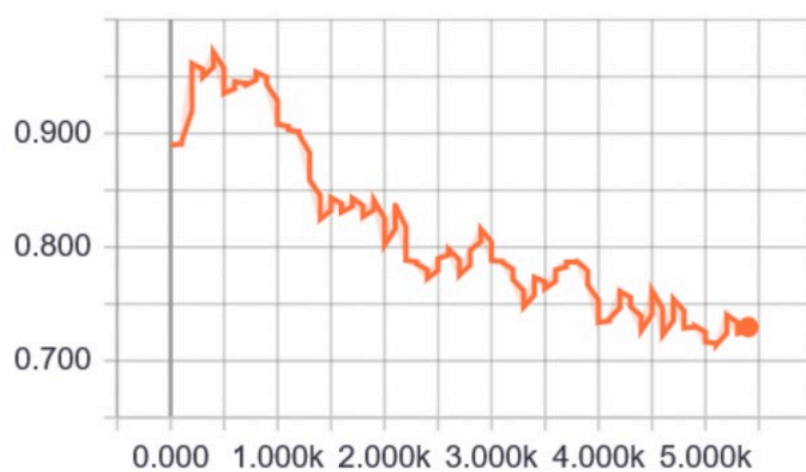
- ConvLayer1 stddev for weights



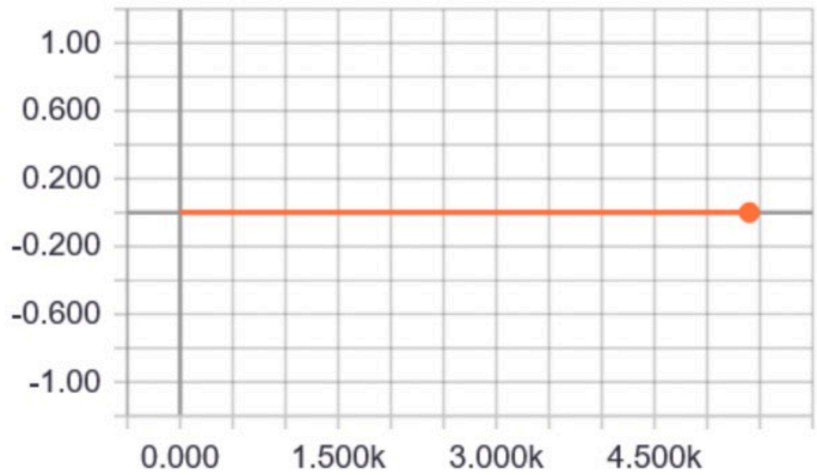
- DenseLayer max for Wx_plust_b



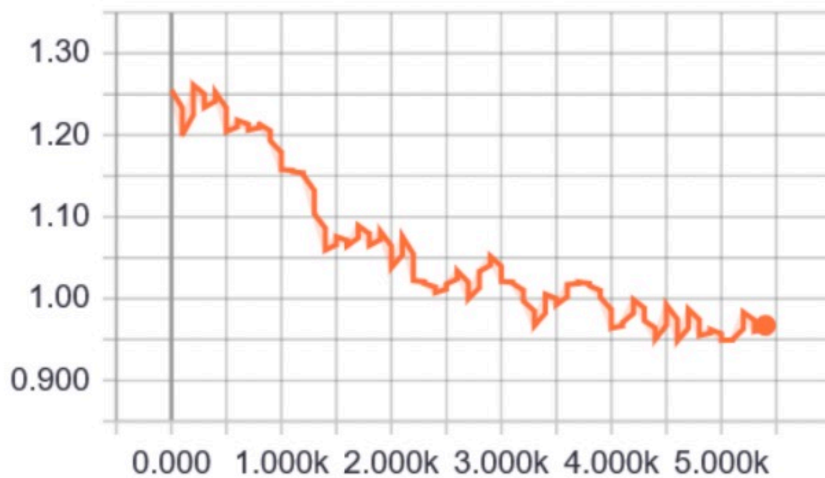
- DenseLayer min for Wx_plust_b



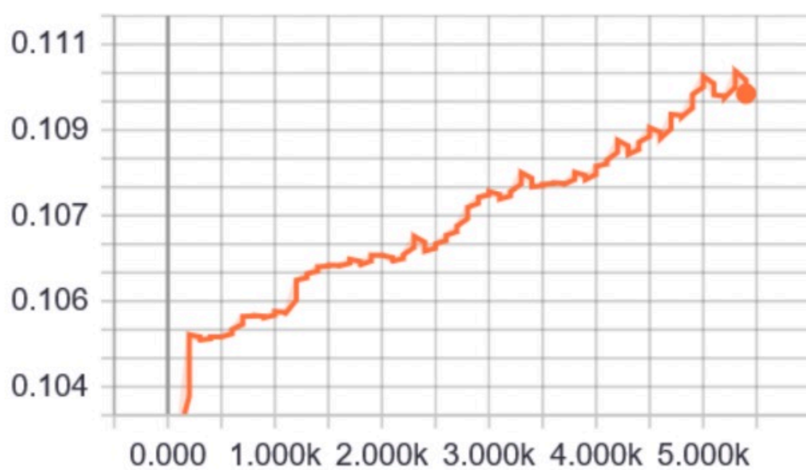
- DenseLayer mean for Wx_plust_b



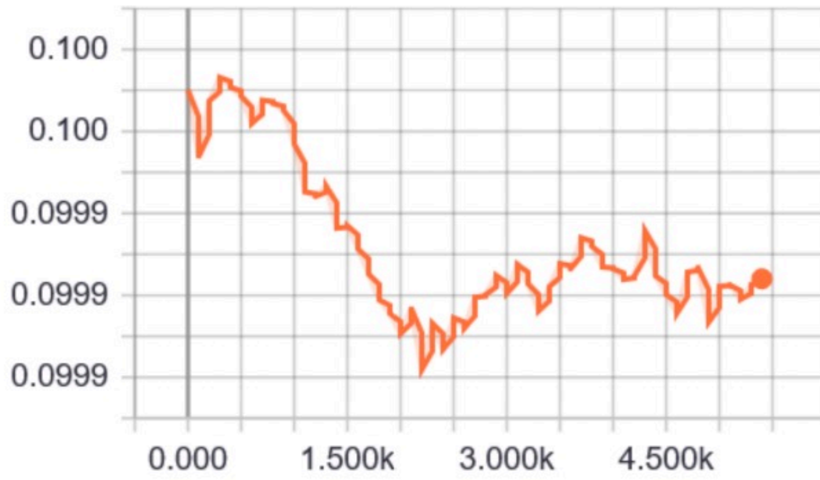
- DenseLayer stddev for Wx_plust_b



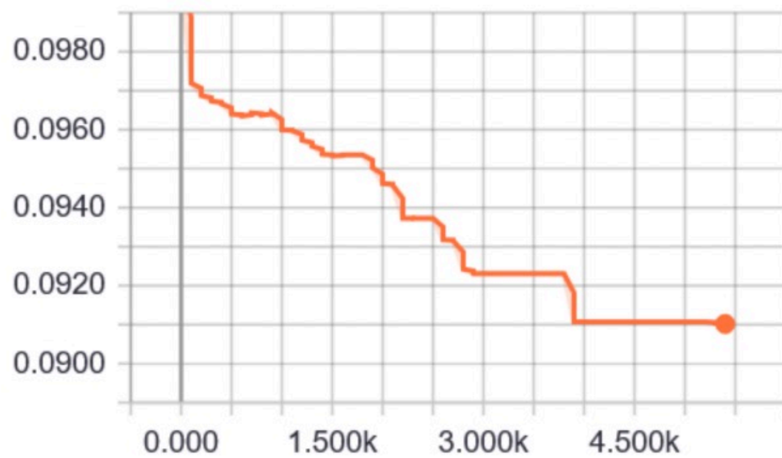
- DenseLayer max for biases



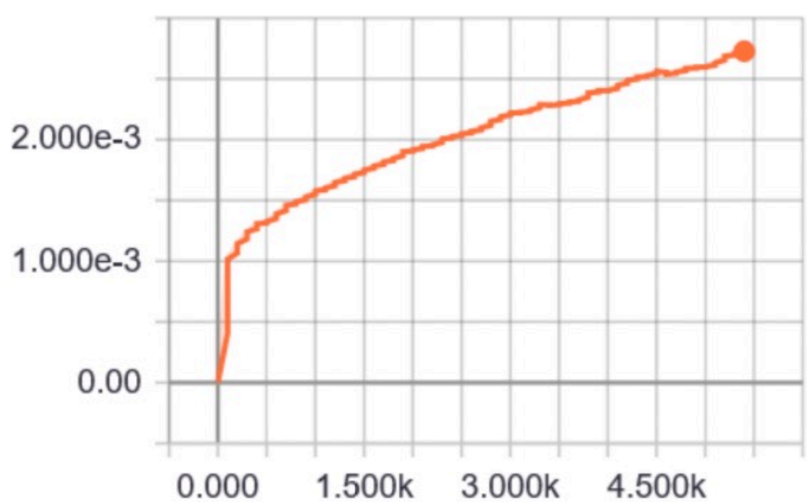
- DenseLayer min for biases



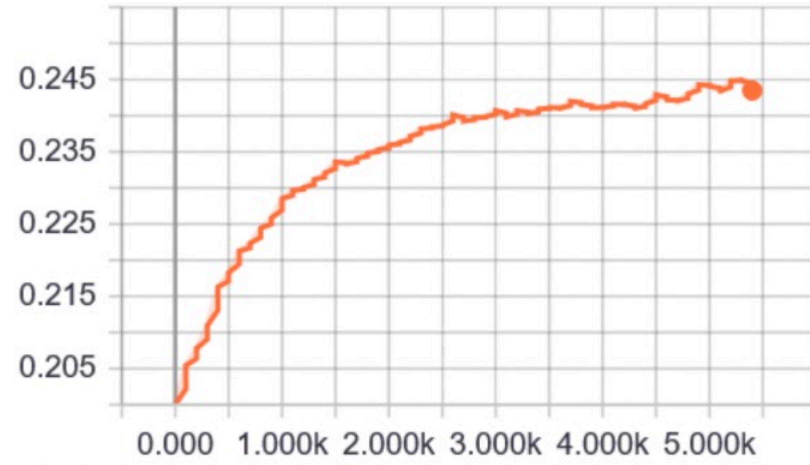
- DenseLayer mean for biases



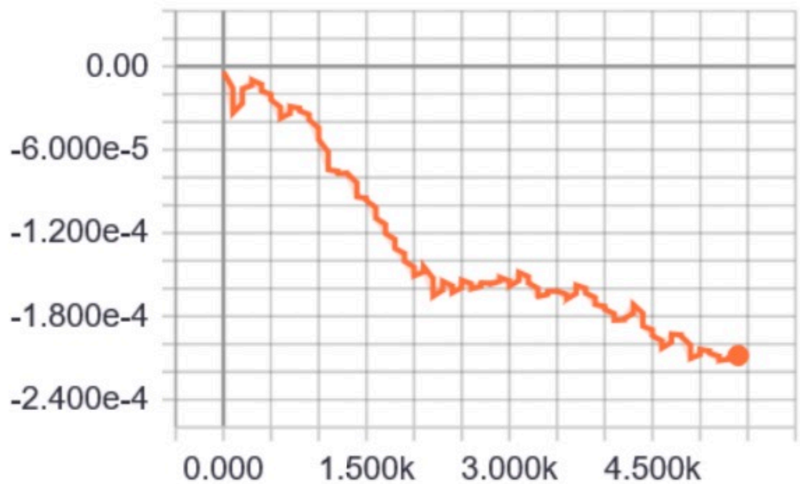
- DenseLayer stddev for biases



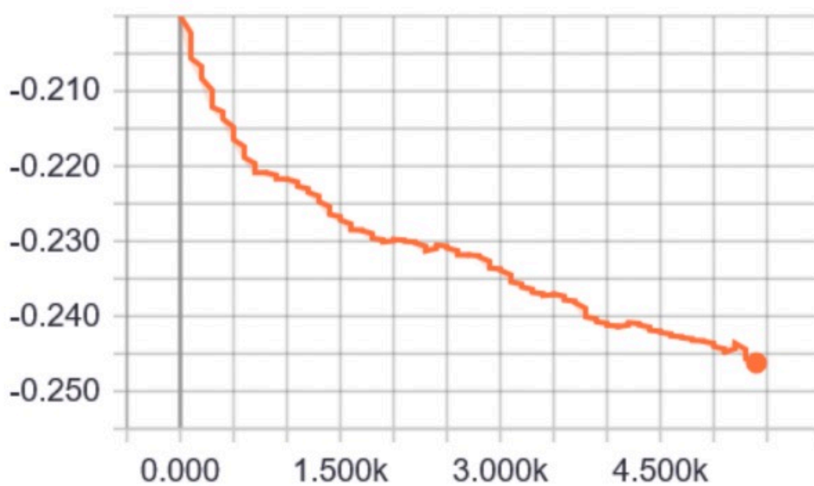
- DenseLayer max for weights



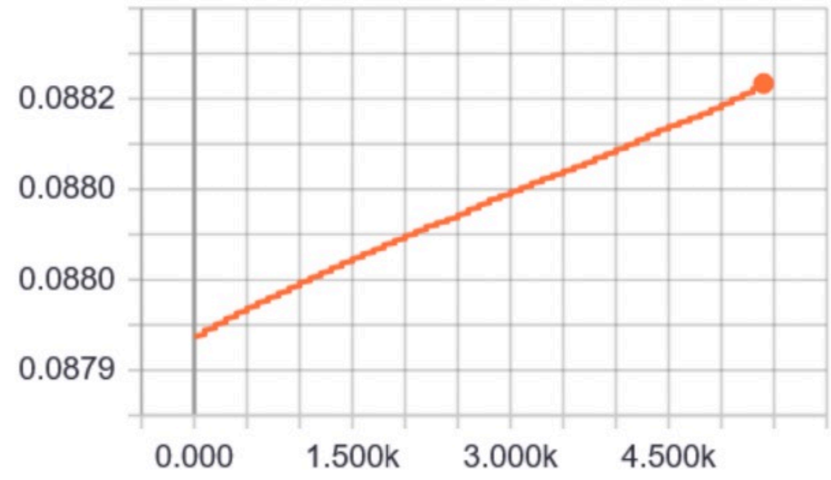
- DenseLayer min for weights



- DenseLayer mean for weights



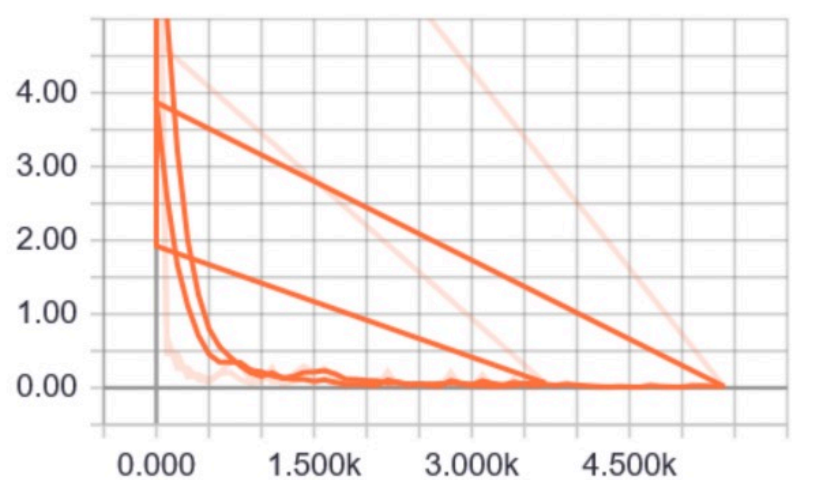
- DenseLayer stddev for weights



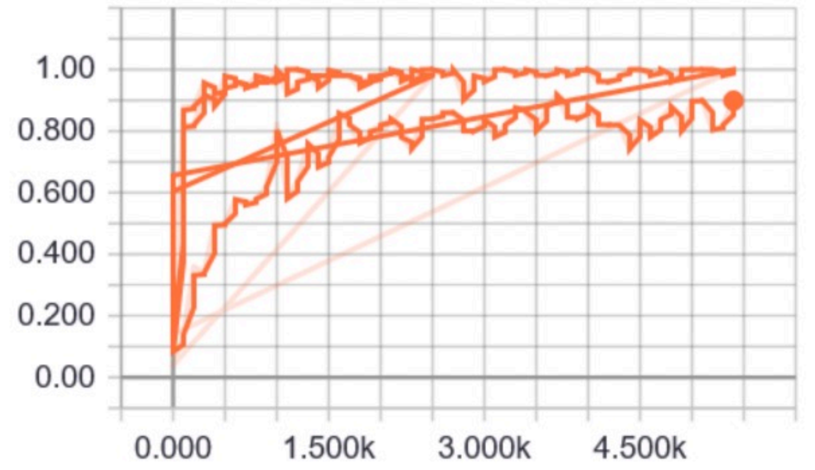
c) Time for More Fun!!! As you have noticed, I use ReLU non-linearity, random initialization, and Adam train￾ing algorithm in dcn mnist.py. In this section, run the network training with different non￾linearities (tanh, sigmoid leaky-ReLU, MaxOut,...), initialization techniques (Xavier...) and training algorithms (SGD, Momentum-based Methods, Adagrad..). Make sure you still monitor the terms specified in part (b). Include the figures generated by TensorBoard and describe what you observe. Again, be curious and creative! You are encouraged to work in groups, but you need to submit separate reports.

Training with Tanh activation function and momentum optimizer. And the final accuracy of 0.9125.

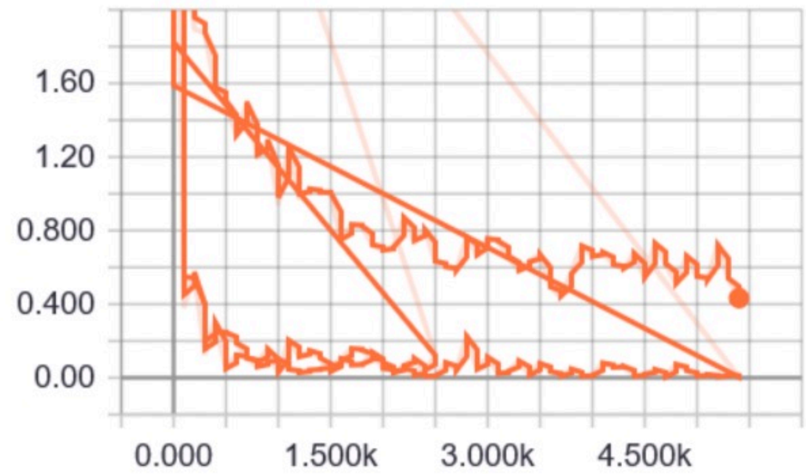
- Mean of Tanh Model



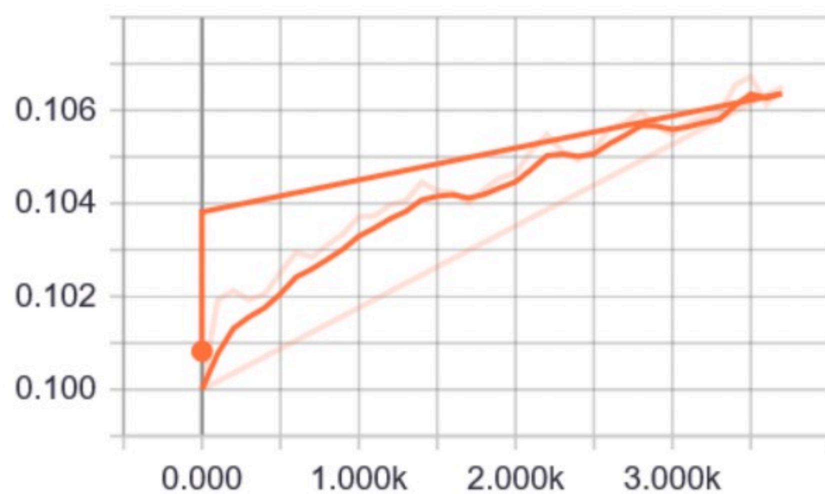
- Accuracy of Tanh Model



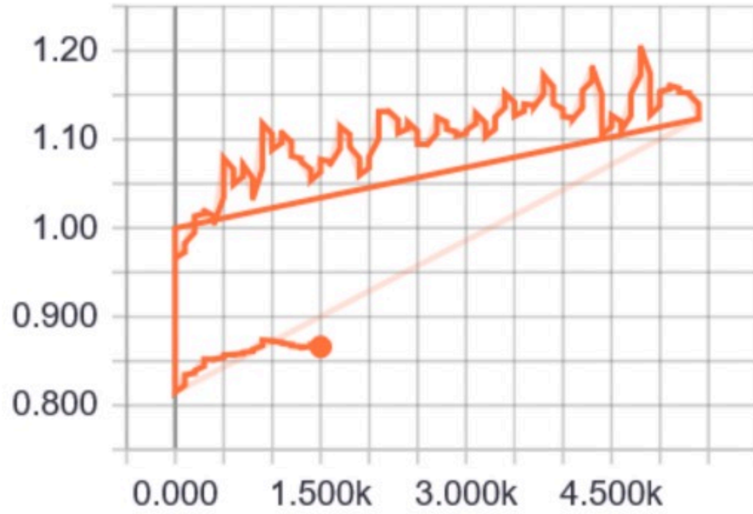
- Cross_entropy of Tanh Model



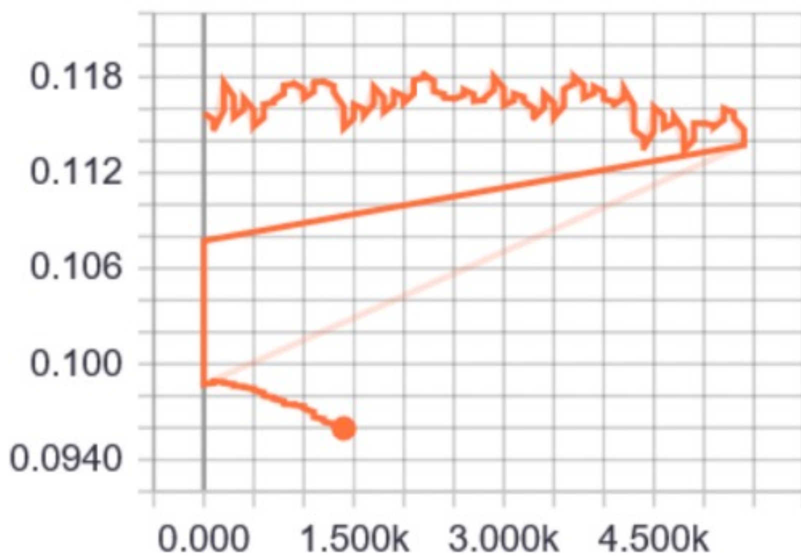
- Summaries of Tanh Model



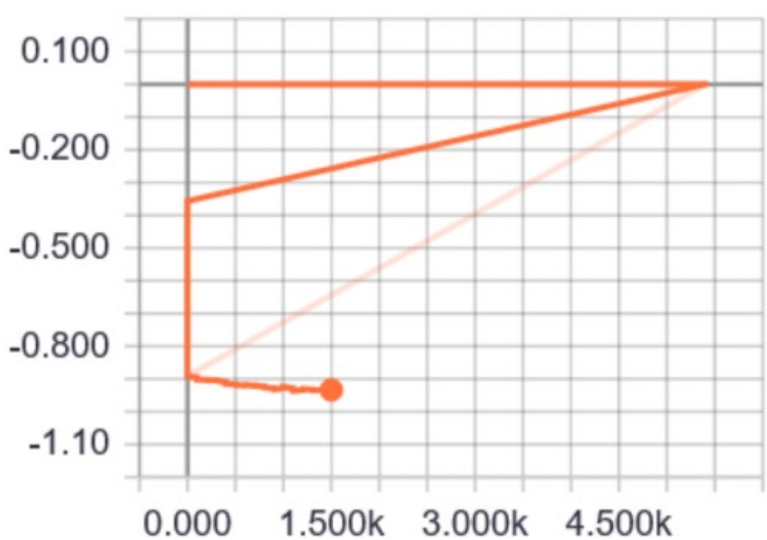
- ConvLayer max of Tanh Model



- ConvLayer mean of Tanh Model



- ConvLayer min of Tanh Model



- ConvLayer stddev of Tanh Model