ELEC 576

Date Due: 10/5/2022

Name: Saif Khan

HW 1

Problem 1:

Part A:

- The code below is used to generate the make moons dataset:

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

    model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3 , nn_output_dim=2,
actFun_type='tanh')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)
```

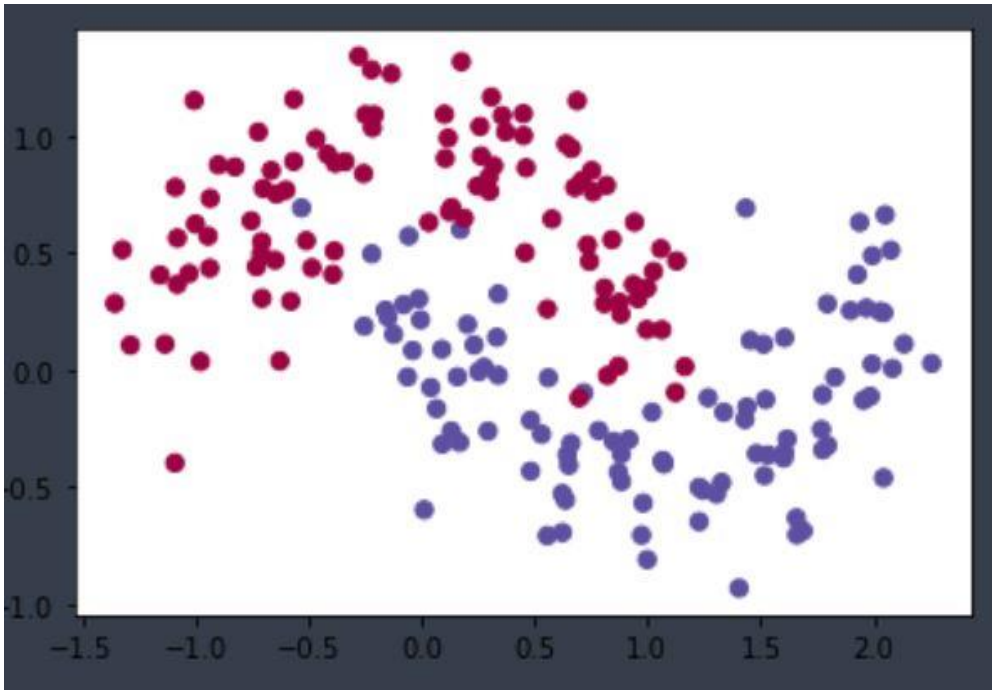


Figure 1: Make moons dataset figure.

Part B:

- The function below computes the activation function in ***actFun(self , z , type)***:

```
def actFun(self, z, type):

    if type == 'tanh':
        z = (2/(1+np.exp(-2*z)))-1
        #return np.tanh(z)
        return z
    elif type == 'sigmoid':
        z = 1/(1+np.exp(-1*z))
    elif type == 'relu':
      return np.maximum(0, z)

    else:
        print("Incorrect function detected,\
              please enter one of the following functions:\
              ReLU, Sigmoid or Tanh")
    return z
```

- The function below computes the activation function in ***diff_actFun(self , z , type)***:

```
    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
        '''

        if type == 'tanh':
            z = (1/np.square(np.cosh(z))) #formula created using
tanh^2+sech^2=1
            #return 1 - np.square(np.tanh(z))
            #f = (2/(1+np.exp(-2*z)))-1
            #z = 1 - np.square(f)
            return z
        elif type == 'sigmoid':
            z = np.exp(-z)/(1 + 2*np.exp(-z) + np.exp(-2*z))
        elif type == 'relu':
          return np.where(z > 0, 1, 0)
```

- Derivative of Sigmoid is function is:

$$\frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \frac{e^{-x}}{1+2e^{-x}+e^{-2x}}$$

- Derivative of Tanh is function is:

$$\frac{d}{dx}\left(\frac{2}{1+e^{-x}}\right) = 1/\cosh\,\hat{}\,2(x)$$

- Derivative of ReLU is function is:

$$f(x) = 0\ for\ x < 0\ , x\ for\ x > or\ equal\ to\ 0$$

$$\frac{d}{dx}(f(x)) = 0\ for\ x < 0\ , 1\ for\ x > or\ equal\ to\ 0$$

Part C:

Part C1:

- The feedforward function is created using 4 formulas:
  - z1 → self.z1
  - a1 → self.a1
  - z2 → self.z2
  - a2 → self.prob

```
    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
```

```
        #Version 1 of a2 = ^y = softmax(z2)
        #self.probs = np.exp(self.z2)/ np.sum(np.exp(self.z2),axis = 1,
keepdims = True)

        #version 2 of a2 = ^y = softmax(z2)
        A = np.exp(self.z2 - np.max(self.z2))
        B = np.sum(A,axis = 1, keepdims = True)
        result = A / B
        self.probs = result

        return None
```

Part C2:

- The function below calculates the loss function using the given **input data X** and **data labels Y** :

```
    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
        yhat = self.probs #This is the same equation found feedforward function
        data_loss = np.sum(-np.log(yhat[range(num_examples), y]))



        # 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
```

Part D:

Part D1:

$$Delta3 = \frac{dL}{dz2} = \frac{dL}{dyhat} * \frac{dyhat}{dz2}$$

$$Delta\ 2 = differentiation\ [Delta3 * Transpose(W2)]$$

$$\frac{dL}{dW1} = Transpose(X) * Delta2$$

$$\frac{dL}{db1} = \sum Delta2$$

$$\frac{dL}{dW2} = Transpose(a1) * Delta3$$

$$\frac{dL}{db2} = \sum Delta3$$

Part D2:

- The code below implements the backpropagation:

```
    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 = (self.a1.T).dot(delta3)
        db2 = np.sum(delta3, axis=0, keepdims=True)
        delta2 = delta3.dot(self.W2.T) * self.diff_actFun(self.z1,
self.actFun_type)
        dW1 = np.dot(X.T, delta2)
        db1 = np.sum(delta2, axis=0)

        return dW1, dW2, db1, db2
```

Part E:

Part E1:

- The code below is using **tanh** function and the results are shown after the code below:

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

model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3 ,
nn_output_dim=2, actFun_type='tanh')
model.fit_model(X,y)
model.visualize_decision_boundary(X,y)
```

```
Loss after iteration 0: 0.432387
Loss after iteration 1000: 0.068947
Loss after iteration 2000: 0.068898
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
```

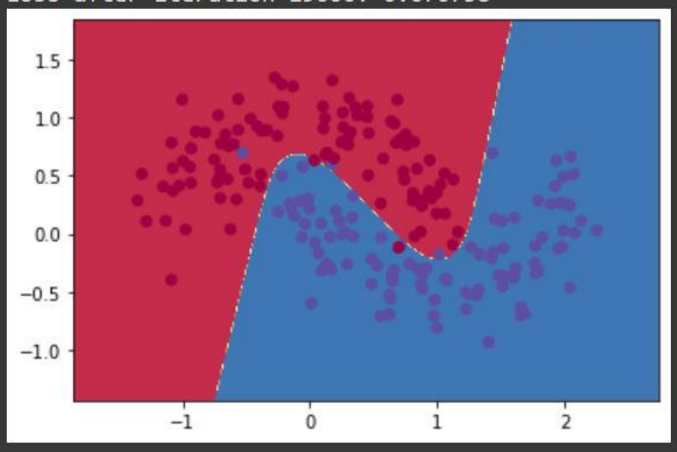


Figure 2: Training Results using **tanh** activation function.

- The code below is using **Sigmoid** function and the results are shown after the code below:

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

model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3 ,
nn_output_dim=2, actFun_type='sigmoid')
model.fit_model(X,y)
model.visualize_decision_boundary(X,y)
```

```
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
```

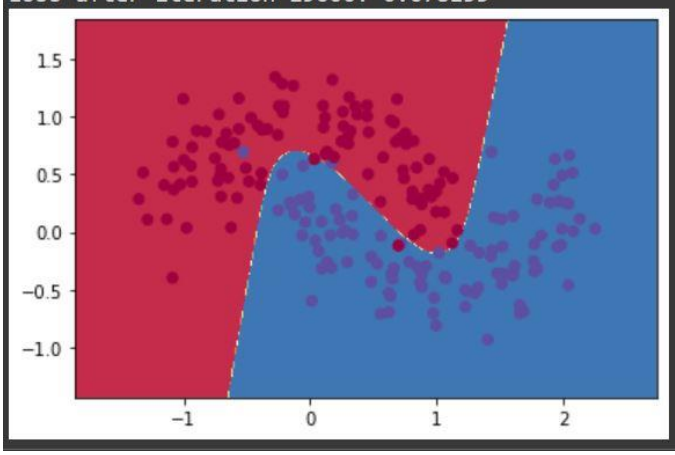


Figure 3: Training Results using **sigmoid** activation function.

- The code below is using **ReLU** function and the results are shown after the code below:

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

model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3 ,
nn_output_dim=2, actFun_type='relu')
model.fit_model(X,y)
model.visualize_decision_boundary(X,y)
```

```
Loss after iteration 0: 0.560274
Loss after iteration 1000: 0.072179
Loss after iteration 2000: 0.071301
Loss after iteration 3000: 0.071159
Loss after iteration 4000: 0.071190
Loss after iteration 5000: 0.071136
Loss after iteration 6000: 0.071276
Loss after iteration 7000: 0.071090
Loss after iteration 8000: 0.071265
Loss after iteration 9000: 0.071084
Loss after iteration 10000: 0.071090
Loss after iteration 11000: 0.071087
Loss after iteration 12000: 0.071086
Loss after iteration 13000: 0.071069
Loss after iteration 14000: 0.071114
Loss after iteration 15000: 0.071074
Loss after iteration 16000: 0.071113
Loss after iteration 17000: 0.071071
Loss after iteration 18000: 0.071090
Loss after iteration 19000: 0.071219
```

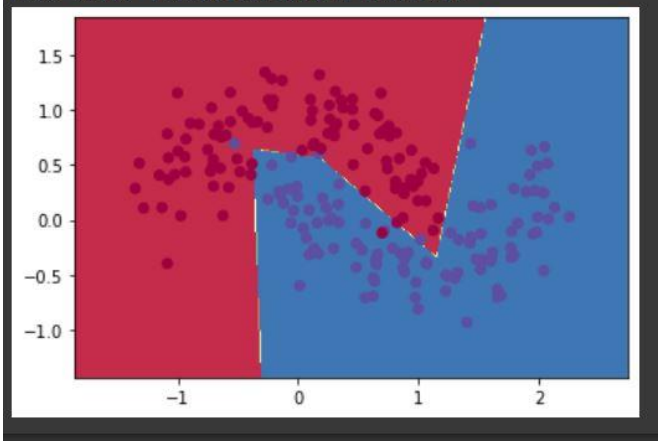


Figure 4: Training Results using **ReLU** activation function.

Part E2:

- Figures 6 to 19 shows training of the 3-layer neural network done using **tanh** activation function with various hidden number of layers.

- Increasing the hidden number of layers to 10 as shown in figure 6 our loss is 3.05% which is a significant decrease in loss compared to the 7.07% shown in figure 2.

- Increasing the hidden number of layers from 10 till 32 as shown in figures 7 till 13 it can be seen that the training loss plateaus at around 3.08%. If the hidden number of layers is increased from 32 to 50 it can be seen in figures 14 and 15

that the training loss actually increases to 3.1% which tells us that after a certain hidden number of layers is added the training loss effects our training model negatively.

- If we further increase the hidden number of layers from 100 to 400 it can be seen in figures 16 to 19 our loss is gradually increasing which confirms that adding more hidden number of layers after 10 actually effects our model negatively since the training loss increases rather than decreasing.

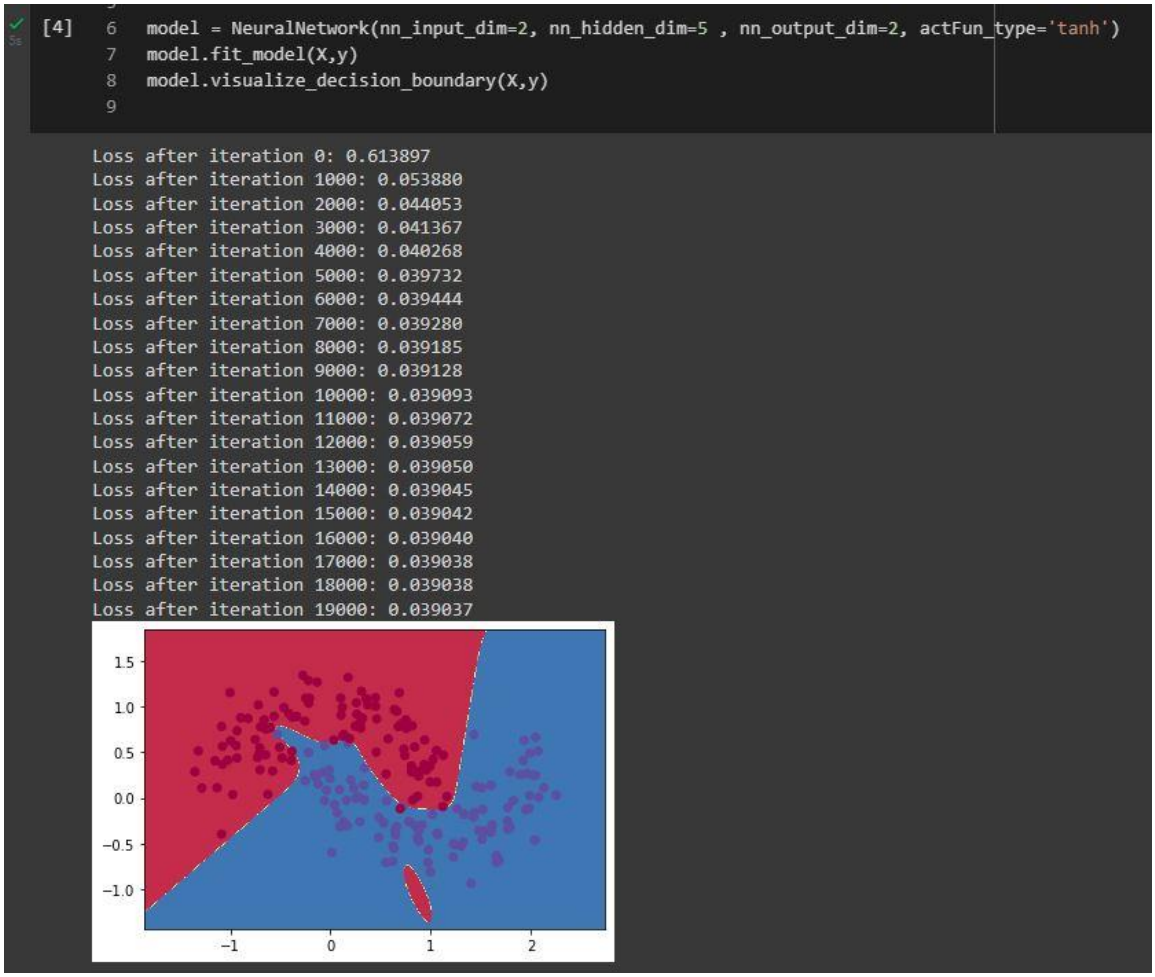


Figure 5: Training Results using **tanh** activation function after setting number of hidden units to **5**.

```
model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=10 , nn_output_dim=2, actFun_type='tanh')
model.fit_model(X,y)
model.visualize_decision_boundary(X,y)
```

```
Loss after iteration 0: 0.634102
Loss after iteration 1000: 0.047377
Loss after iteration 2000: 0.038216
Loss after iteration 3000: 0.035216
Loss after iteration 4000: 0.033740
Loss after iteration 5000: 0.032800
Loss after iteration 6000: 0.032126
Loss after iteration 7000: 0.031624
Loss after iteration 8000: 0.031264
Loss after iteration 9000: 0.031044
Loss after iteration 10000: 0.030910
Loss after iteration 11000: 0.030821
Loss after iteration 12000: 0.030758
Loss after iteration 13000: 0.030711
Loss after iteration 14000: 0.030675
Loss after iteration 15000: 0.030646
Loss after iteration 16000: 0.030622
Loss after iteration 17000: 0.030601
Loss after iteration 18000: 0.030582
Loss after iteration 19000: 0.030565
```

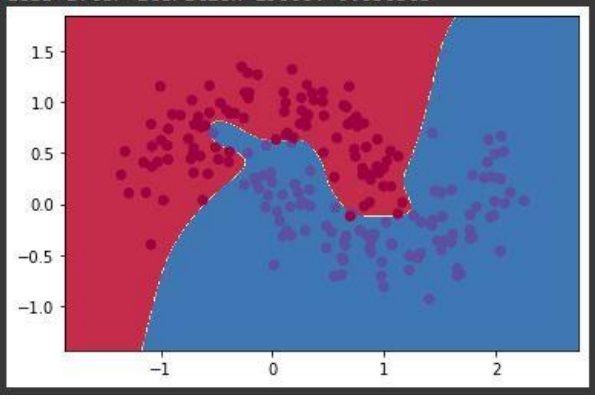


Figure 6: Training Results using **tanh** activation function after setting number of hidden units to **10**.

```
[25]  6  model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=12, nn_output_dim=2, actFun_type='tanh')
      7  model.fit_model(X,y)
      8  model.visualize_decision_boundary(X,y)
      9
```

```
Loss after iteration 0: 0.677208
Loss after iteration 1000: 0.050805
Loss after iteration 2000: 0.043095
Loss after iteration 3000: 0.039945
Loss after iteration 4000: 0.038503
Loss after iteration 5000: 0.037486
Loss after iteration 6000: 0.036667
Loss after iteration 7000: 0.036000
Loss after iteration 8000: 0.035415
Loss after iteration 9000: 0.034876
Loss after iteration 10000: 0.034362
Loss after iteration 11000: 0.033855
Loss after iteration 12000: 0.033323
Loss after iteration 13000: 0.032796
Loss after iteration 14000: 0.032343
Loss after iteration 15000: 0.031943
Loss after iteration 16000: 0.031595
Loss after iteration 17000: 0.031299
Loss after iteration 18000: 0.031059
Loss after iteration 19000: 0.030888
```

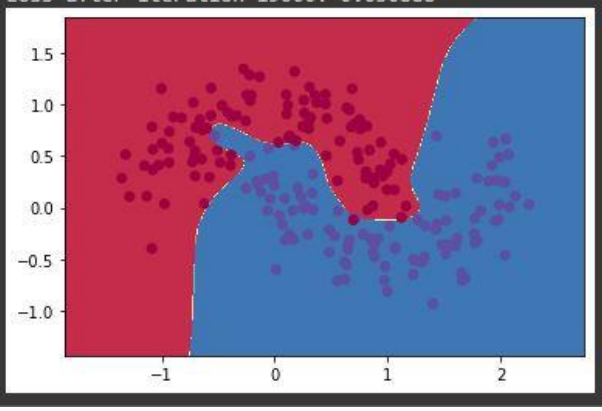


Figure 7: Training Results using **tanh** activation function after setting number of hidden units to **12**.

```
[26]  6  model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=16, nn_output_dim=2, actFun_type='tanh')
      7  model.fit_model(X,y)
      8  model.visualize_decision_boundary(X,y)
      9
```

```
Loss after iteration 0: 0.671185
Loss after iteration 1000: 0.064950
Loss after iteration 2000: 0.041072
Loss after iteration 3000: 0.035949
Loss after iteration 4000: 0.034260
Loss after iteration 5000: 0.033236
Loss after iteration 6000: 0.032480
Loss after iteration 7000: 0.031927
Loss after iteration 8000: 0.031574
Loss after iteration 9000: 0.031353
Loss after iteration 10000: 0.031204
Loss after iteration 11000: 0.031094
Loss after iteration 12000: 0.031003
Loss after iteration 13000: 0.030922
Loss after iteration 14000: 0.030846
Loss after iteration 15000: 0.030772
Loss after iteration 16000: 0.030702
Loss after iteration 17000: 0.030637
Loss after iteration 18000: 0.030580
Loss after iteration 19000: 0.030530
```

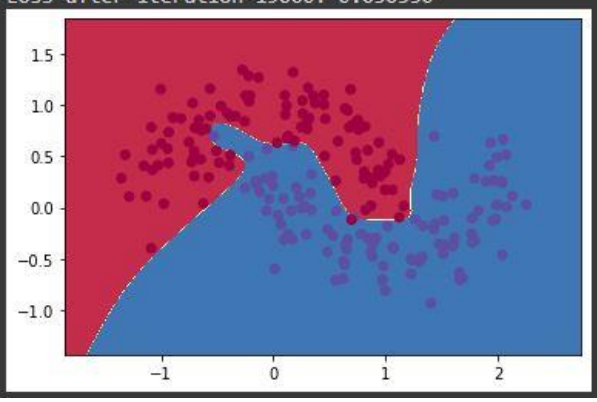


Figure 8: Training Results using **tanh** activation function after setting number of hidden units to **16**.

```
model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=18, nn_output_dim=2, actFun_type='tanh')
model.fit_model(X,y)
model.visualize_decision_boundary(X,y)
```

```
Loss after iteration 0: 0.364404
Loss after iteration 1000: 0.052611
Loss after iteration 2000: 0.043724
Loss after iteration 3000: 0.038608
Loss after iteration 4000: 0.036049
Loss after iteration 5000: 0.034500
Loss after iteration 6000: 0.033374
Loss after iteration 7000: 0.032571
Loss after iteration 8000: 0.032102
Loss after iteration 9000: 0.031800
Loss after iteration 10000: 0.031587
Loss after iteration 11000: 0.031429
Loss after iteration 12000: 0.031308
Loss after iteration 13000: 0.031217
Loss after iteration 14000: 0.031144
Loss after iteration 15000: 0.031083
Loss after iteration 16000: 0.031030
Loss after iteration 17000: 0.030984
Loss after iteration 18000: 0.030943
Loss after iteration 19000: 0.030908
```

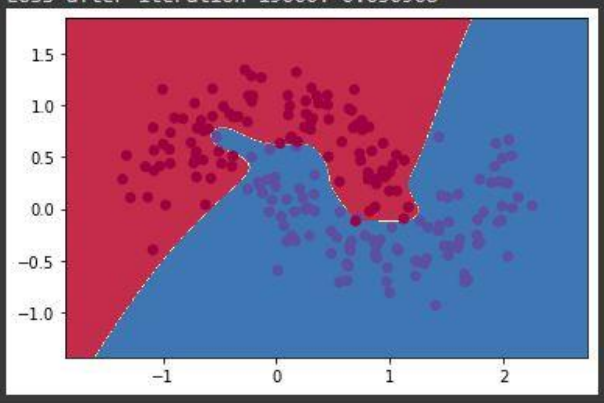


Figure 9: Training Results using **tanh** activation function after setting number of hidden units to **18**.

```
[28]  6  model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=20, nn_output_dim=2, actFun_type='tanh')
      7  model.fit_model(X,y)
      8  model.visualize_decision_boundary(X,y)
      9
```

```
Loss after iteration 0: 0.498538
Loss after iteration 1000: 0.047806
Loss after iteration 2000: 0.039152
Loss after iteration 3000: 0.036454
Loss after iteration 4000: 0.035112
Loss after iteration 5000: 0.034247
Loss after iteration 6000: 0.033601
Loss after iteration 7000: 0.033072
Loss after iteration 8000: 0.032624
Loss after iteration 9000: 0.032252
Loss after iteration 10000: 0.031943
Loss after iteration 11000: 0.031683
Loss after iteration 12000: 0.031463
Loss after iteration 13000: 0.031286
Loss after iteration 14000: 0.031150
Loss after iteration 15000: 0.031039
Loss after iteration 16000: 0.030948
Loss after iteration 17000: 0.030877
Loss after iteration 18000: 0.030821
Loss after iteration 19000: 0.030774
```

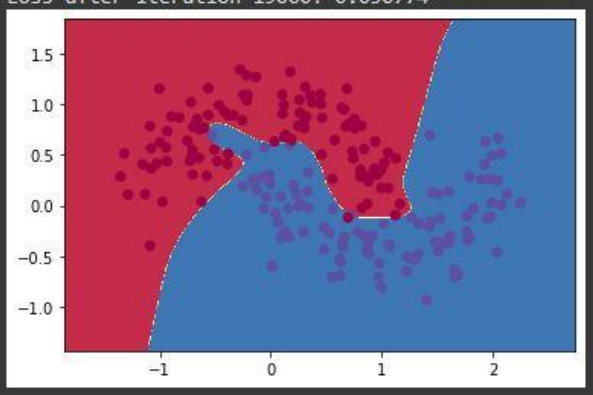


Figure 10: Training Results using **tanh** activation function after setting number of hidden units to **20**.

```
[29]  6  model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=25, nn_output_dim=2, actFun_type='tanh')
      7  model.fit_model(X,y)
      8  model.visualize_decision_boundary(X,y)
      9
```

```
Loss after iteration 0: 1.421211
Loss after iteration 1000: 0.052361
Loss after iteration 2000: 0.042873
Loss after iteration 3000: 0.038661
Loss after iteration 4000: 0.036657
Loss after iteration 5000: 0.035436
Loss after iteration 6000: 0.034610
Loss after iteration 7000: 0.033989
Loss after iteration 8000: 0.033462
Loss after iteration 9000: 0.032994
Loss after iteration 10000: 0.032576
Loss after iteration 11000: 0.032205
Loss after iteration 12000: 0.031881
Loss after iteration 13000: 0.031603
Loss after iteration 14000: 0.031374
Loss after iteration 15000: 0.031191
Loss after iteration 16000: 0.031044
Loss after iteration 17000: 0.030925
Loss after iteration 18000: 0.030827
Loss after iteration 19000: 0.030746
```

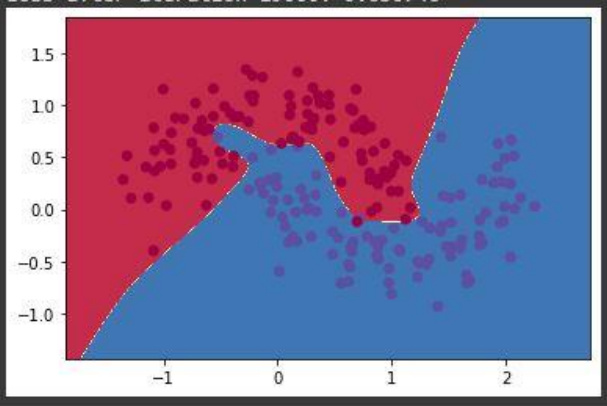


Figure 11: Training Results using **tanh** activation function after setting number of hidden units to **25**.

```
model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=30, nn_output_dim=2, actFun_type='tanh')
model.fit_model(X,y)
model.visualize_decision_boundary(X,y)

```

```
Loss after iteration 0: 0.345044
Loss after iteration 1000: 0.057101
Loss after iteration 2000: 0.050564
Loss after iteration 3000: 0.045179
Loss after iteration 4000: 0.040010
Loss after iteration 5000: 0.037333
Loss after iteration 6000: 0.035688
Loss after iteration 7000: 0.034540
Loss after iteration 8000: 0.033705
Loss after iteration 9000: 0.033048
Loss after iteration 10000: 0.032519
Loss after iteration 11000: 0.032088
Loss after iteration 12000: 0.031746
Loss after iteration 13000: 0.031495
Loss after iteration 14000: 0.031308
Loss after iteration 15000: 0.031168
Loss after iteration 16000: 0.031067
Loss after iteration 17000: 0.030991
Loss after iteration 18000: 0.030932
Loss after iteration 19000: 0.030884
```

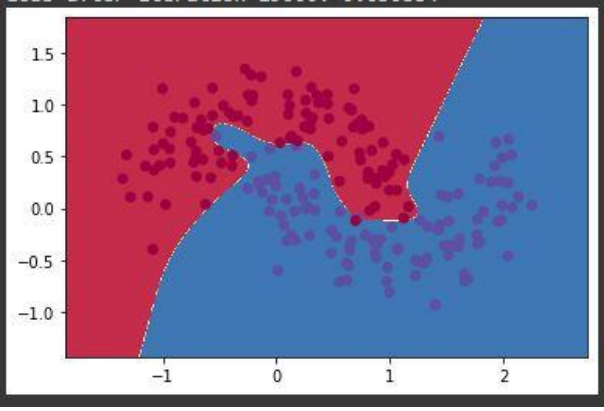


Figure 12: Training Results using **tanh** activation function after setting number of hidden units to **30**.

```
[31]  6  model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=32, nn_output_dim=2, actFun_type='tanh')
      7  model.fit_model(X,y)
      8  model.visualize_decision_boundary(X,y)
      9
```

```
Loss after iteration 0: 1.349165
Loss after iteration 1000: 0.056592
Loss after iteration 2000: 0.050824
Loss after iteration 3000: 0.047725
Loss after iteration 4000: 0.043147
Loss after iteration 5000: 0.039042
Loss after iteration 6000: 0.036523
Loss after iteration 7000: 0.034945
Loss after iteration 8000: 0.033909
Loss after iteration 9000: 0.033184
Loss after iteration 10000: 0.032642
Loss after iteration 11000: 0.032215
Loss after iteration 12000: 0.031870
Loss after iteration 13000: 0.031600
Loss after iteration 14000: 0.031394
Loss after iteration 15000: 0.031237
Loss after iteration 16000: 0.031113
Loss after iteration 17000: 0.031012
Loss after iteration 18000: 0.030930
Loss after iteration 19000: 0.030862
```

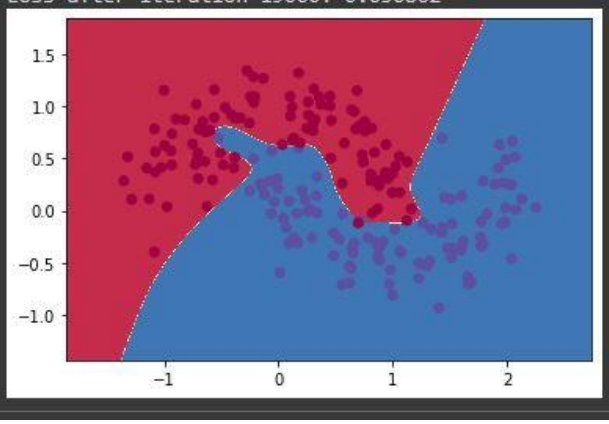


Figure 13: Training Results using **tanh** activation function after setting number of hidden units to **32**.

```
[32]  6  model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=36, nn_output_dim=2, actFun_type='tanh')
      7  model.fit_model(X,y)
      8  model.visualize_decision_boundary(X,y)
      9
```

```
Loss after iteration 0: 0.650120
Loss after iteration 1000: 0.056722
Loss after iteration 2000: 0.050741
Loss after iteration 3000: 0.048086
Loss after iteration 4000: 0.045070
Loss after iteration 5000: 0.041328
Loss after iteration 6000: 0.038093
Loss after iteration 7000: 0.036041
Loss after iteration 8000: 0.034681
Loss after iteration 9000: 0.033730
Loss after iteration 10000: 0.033055
Loss after iteration 11000: 0.032574
Loss after iteration 12000: 0.032215
Loss after iteration 13000: 0.031939
Loss after iteration 14000: 0.031722
Loss after iteration 15000: 0.031547
Loss after iteration 16000: 0.031400
Loss after iteration 17000: 0.031274
Loss after iteration 18000: 0.031166
Loss after iteration 19000: 0.031075
```

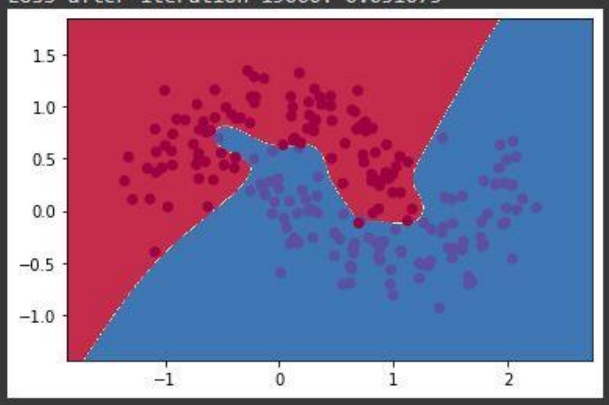


Figure 14: Training Results using **tanh** activation function after setting number of hidden units to **36**.

```
[33]
5
6  model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=50, nn_output_dim=2, actFun_type='tanh')
7  model.fit_model(X,y)
8  model.visualize_decision_boundary(X,y)
9
```

```
Loss after iteration 0: 1.056085
Loss after iteration 1000: 0.056989
Loss after iteration 2000: 0.049367
Loss after iteration 3000: 0.042195
Loss after iteration 4000: 0.037284
Loss after iteration 5000: 0.034583
Loss after iteration 6000: 0.033298
Loss after iteration 7000: 0.032611
Loss after iteration 8000: 0.032192
Loss after iteration 9000: 0.031917
Loss after iteration 10000: 0.031724
Loss after iteration 11000: 0.031581
Loss after iteration 12000: 0.031468
Loss after iteration 13000: 0.031375
Loss after iteration 14000: 0.031296
Loss after iteration 15000: 0.031227
Loss after iteration 16000: 0.031167
Loss after iteration 17000: 0.031115
Loss after iteration 18000: 0.031069
Loss after iteration 19000: 0.031028
```

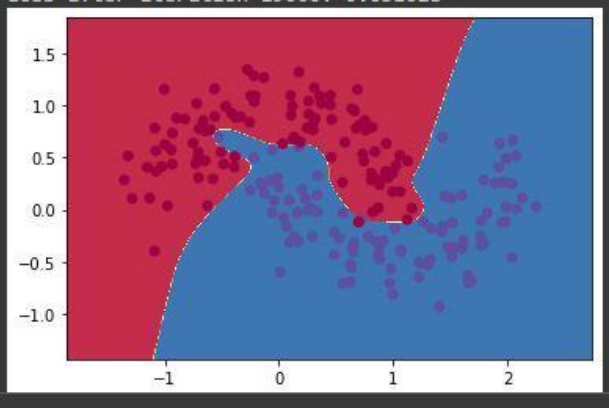


Figure 15: Training Results using **tanh** activation function after setting number of hidden units to **50**.

```
model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=100, nn_output_dim=2, actFun_type='tanh')
model.fit_model(X,y)
model.visualize_decision_boundary(X,y)

Loss after iteration 0: 3.337639
Loss after iteration 1000: 0.055917
Loss after iteration 2000: 0.051549
Loss after iteration 3000: 0.049034
Loss after iteration 4000: 0.044996
Loss after iteration 5000: 0.041430
Loss after iteration 6000: 0.039521
Loss after iteration 7000: 0.038245
Loss after iteration 8000: 0.037321
Loss after iteration 9000: 0.036611
Loss after iteration 10000: 0.036011
Loss after iteration 11000: 0.035453
Loss after iteration 12000: 0.034989
Loss after iteration 13000: 0.034534
Loss after iteration 14000: 0.034084
Loss after iteration 15000: 0.033670
Loss after iteration 16000: 0.033307
Loss after iteration 17000: 0.032995
Loss after iteration 18000: 0.032727
Loss after iteration 19000: 0.032493
```

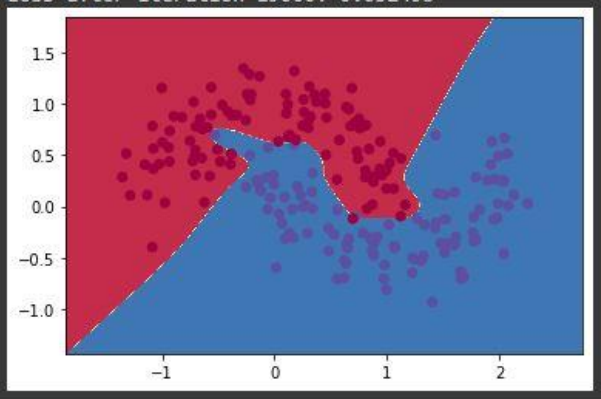


Figure 16: Training Results using **tanh** activation function after setting number of hidden units to **100**.

```
[35]  6  model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=200, nn_output_dim=2, actFun_type='tanh')
      7  model.fit_model(X,y)
      8  model.visualize_decision_boundary(X,y)
      9
```

```
Loss after iteration 0: 5.105435
Loss after iteration 1000: 0.057131
Loss after iteration 2000: 0.052637
Loss after iteration 3000: 0.049063
Loss after iteration 4000: 0.045402
Loss after iteration 5000: 0.042689
Loss after iteration 6000: 0.040019
Loss after iteration 7000: 0.038934
Loss after iteration 8000: 0.038202
Loss after iteration 9000: 0.037683
Loss after iteration 10000: 0.037293
Loss after iteration 11000: 0.036985
Loss after iteration 12000: 0.036732
Loss after iteration 13000: 0.036520
Loss after iteration 14000: 0.036340
Loss after iteration 15000: 0.036186
Loss after iteration 16000: 0.036052
Loss after iteration 17000: 0.035935
Loss after iteration 18000: 0.035831
Loss after iteration 19000: 0.035737
```

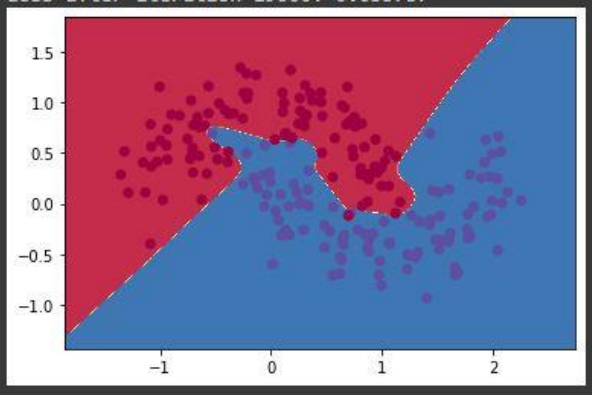


Figure 17: Training Results using **tanh** activation function after setting number of hidden units to **200**.

```
[36]
model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=300, nn_output_dim=2, actFun_type='tanh')
model.fit_model(X,y)
model.visualize_decision_boundary(X,y)
```

```
Loss after iteration 0: 6.761681
Loss after iteration 1000: 0.060224
Loss after iteration 2000: 0.054406
Loss after iteration 3000: 0.051136
Loss after iteration 4000: 0.045940
Loss after iteration 5000: 0.041287
Loss after iteration 6000: 0.038355
Loss after iteration 7000: 0.036679
Loss after iteration 8000: 0.035665
Loss after iteration 9000: 0.035061
Loss after iteration 10000: 0.034640
Loss after iteration 11000: 0.034312
Loss after iteration 12000: 0.034022
Loss after iteration 13000: 0.033738
Loss after iteration 14000: 0.033453
Loss after iteration 15000: 0.033173
Loss after iteration 16000: 0.032914
Loss after iteration 17000: 0.032686
Loss after iteration 18000: 0.032496
Loss after iteration 19000: 0.032340
```

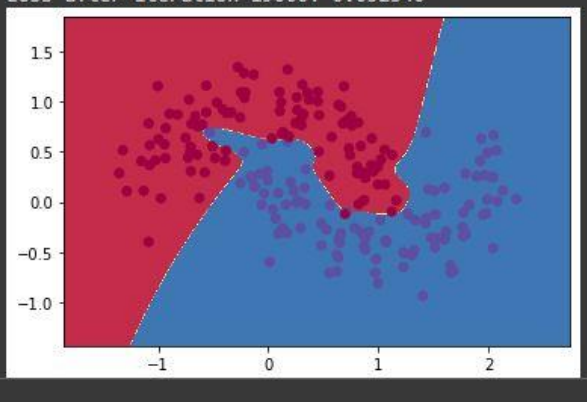


Figure 18: Training Results using **tanh** activation function after setting number of hidden units to **300**.

```
[37]  6  model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=400, nn_output_dim=2, actFun_type='tanh')
      7  model.fit_model(X,y)
      8  model.visualize_decision_boundary(X,y)
      9
```

```
Loss after iteration 0: 8.781713
Loss after iteration 1000: 0.056969
Loss after iteration 2000: 0.046617
Loss after iteration 3000: 0.041765
Loss after iteration 4000: 0.039756
Loss after iteration 5000: 0.038676
Loss after iteration 6000: 0.037889
Loss after iteration 7000: 0.037243
Loss after iteration 8000: 0.036733
Loss after iteration 9000: 0.036258
Loss after iteration 10000: 0.035815
Loss after iteration 11000: 0.035457
Loss after iteration 12000: 0.035185
Loss after iteration 13000: 0.034978
Loss after iteration 14000: 0.034820
Loss after iteration 15000: 0.034698
Loss after iteration 16000: 0.034601
Loss after iteration 17000: 0.034530
Loss after iteration 18000: 0.034470
Loss after iteration 19000: 0.034385
```

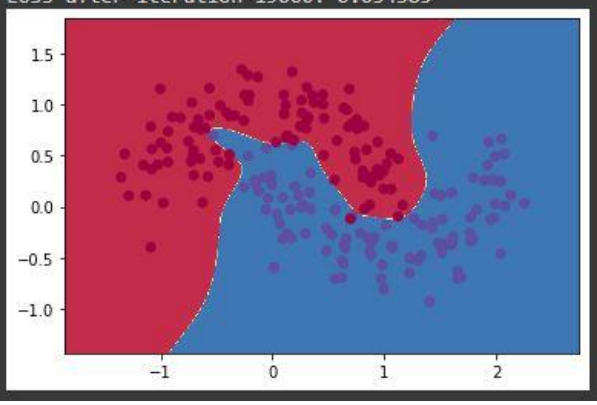


Figure 19: Training Results using **tanh** activation function after setting number of hidden units to **400**.

Part F:

- The code below is the new class called **DeepNeuralNetwork** that inherits **Neural Network**:

```
class DeepNeuralNetwork(object):
    """
    This class builds and trains a deep neural network
    """

    def __init__(self, nn_dims, actFun_type='tanh', reg_lambda=0.01,
seed=0):

        self.nn_dims = nn_dims
        self.actFun_type = actFun_type
        self.reg_lambda = reg_lambda

        # initialize the weights and biases in the network
        self.W = []
        self.b = []

        np.random.seed(seed)
        for i in range(len(nn_dims)-1):
            self.W.append(np.random.randn(self.nn_dims[i],
self.nn_dims[i+1])
                          / np.sqrt(self.nn_dims[i]))
            self.b.append(np.zeros((1, self.nn_dims[i+1])))

    def actFun(self, z, type):

        if type == 'tanh':
            z = (2/(1+np.exp(-2*z)))-1
            #return np.tanh(z)
            return z
        elif type == 'sigmoid':
            z = 1/(1+np.exp(-1*z))
        elif type == 'relu':
          return np.maximum(0, z)

        else:
            print("Incorrect function detected,\
                  please enter one of the following functions:\
                  ReLU, Sigmoid or Tanh")
        return z

    def diff_actFun(self, z, type):


        if type == 'tanh':
            z = (1/np.square(np.cosh(z))) #formula created using
tanh^2+sech^2=1
            #return 1 - np.square(np.tanh(z))
            #f = (2/(1+np.exp(-2*z)))-1
            #z = 1 - np.square(f)
            return z
        elif type == 'sigmoid':
            z = np.exp(-z)/(1 + 2*np.exp(-z) + np.exp(-2*z))
        elif type == 'relu':
          return np.where(z > 0, 1, 0)

        else:
            print("Incorrect function detected,\
                  please enter one of the following functions:\
                  ReLU, Sigmoid or Tanh")
        return z

    def feedforward(self, X, actFun):
```

```
        self.a = []
        self.z = []
        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]))
        C = np.exp(self.z[len(self.z)-1])
        self.probs = C / np.sum(C, axis=1, keepdims=True)
        return None

    def calculate_loss(self, X, y):

        num_examples = len(X)
        self.feedforward(X, lambda x: self.actFun(x,
type=self.actFun_type))
        # Calculating the loss

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

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

    def predict(self, 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):


        num_examples = len(X)
        delta = self.probs
        delta[range(num_examples), y] -= 1

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

        return dW, db
```

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

        # 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)

            # Add regularization terms
            for i in range(len(dW)):
                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]

            # print the loss.
            if print_loss and i % 1000 == 0:
                print("Loss after iteration %i: %f" % (i,
self.calculate_loss(X, y)))

    def visualize_decision_boundary(self, X, y):

        plot_decision_boundary(lambda x: self.predict(x), X, y)
```

- The code below is used to execute training for the **Make-Moons Dataset**:

```
def main():

    # # generate and visualize Make-Moons dataset
    X, y = generate_data()

    #Train the Neural Network Model
    model = DeepNeuralNetwork(nn_dims=[2,10,8], actFun_type='sigmoid')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

- The code below is used to execute training for the **Make-Circles Dataset**:

```
def main():

    # # generate and visualize Make-Circles dataset
    X, y = datasets.make_circles(n_samples=100, shuffle=True, noise=None,
random_state=None, factor=0.8)

    #Train the Neural Network Model
    model = DeepNeuralNetwork(nn_dims=[2,10,8], actFun_type='sigmoid')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

- Figures 20 to 28 show the **Make-Moons Dataset** using 3, 4 and 5 layer network. From figures 20 to 22 it can be seen that using relu activation function with 4 and 5 layer network makes a significant loss reduction and better data fit as compared to a 3 layer network. The sigmoid activation function in figures 23 to 25 don't show much of an improvement unless the hidden number of layers are increased significantly such as 100. The tanh function on the other hand shows much improvement and better fit even with 4 and 5 layer compared to relu and sigmoid functions and lower number of hidden layers as shown in figures 26 to 28.

```
[75] 377     # # generate and visualize Make-Moons dataset
     378     X, y = generate_data()
     379
     380     model = DeepNeuralNetwork(nn_dims=[2, 10, 2], actFun_type='relu')
     381     model.fit_model(X,y)
     382     model.visualize_decision_boundary(X,y)
     383
     384 if __name__ == "__main__":
     385     main()
```

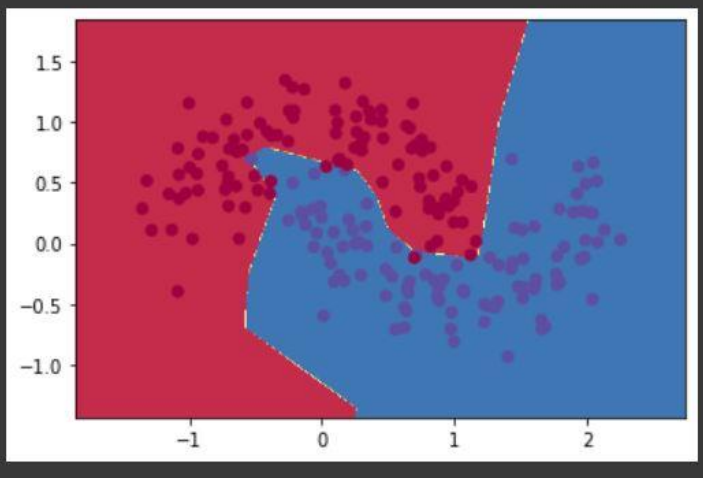


Figure 20: Training Results using **relu** activation function using 3 layer neural network.

```
375 def main():
376
377     # # generate and visualize Make-Moons dataset
378     X, y = generate_data()
379     # # generate and visualize Make-Circles dataset
380     #X, y = datasets.make_circles(n_samples=100, shuffle=True, noise=None, random_state=None, factor=0.8)
381     model = DeepNeuralNetwork(nn_dims=[2, 10, 4, 5], actFun_type='relu')
382     model.fit_model(X,y)
383     model.visualize_decision_boundary(X,y)
384
385 if __name__ == "__main__":
386     main()
```

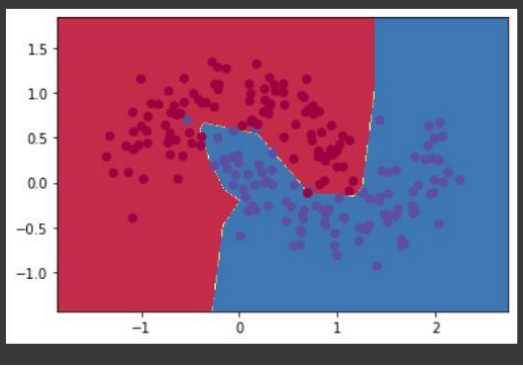


Figure 21: Training Results using **relu** activation function using 4 layer neural network.

```
381     model = DeepNeuralNetwork(nn_dims=[2, 10, 4, 8,8], actFun_type='relu')
382     model.fit_model(X,y)
383     model.visualize_decision_boundary(X,y)
384
385 if __name__ == "__main__":
386     main()
```

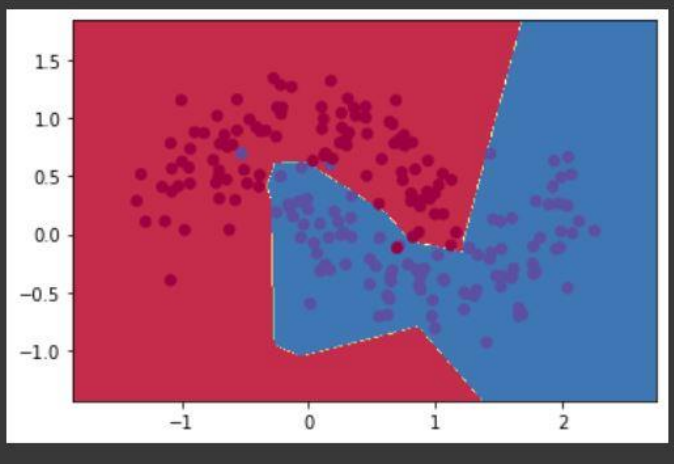


Figure 22: Training Results using **relu** activation function using 5 layer neural network.

```python
def main():

    # # generate and visualize Make-Moons dataset
    X, y = generate_data()

    model = DeepNeuralNetwork(nn_dims=[2, 4, 2], actFun_type='sigmoid')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

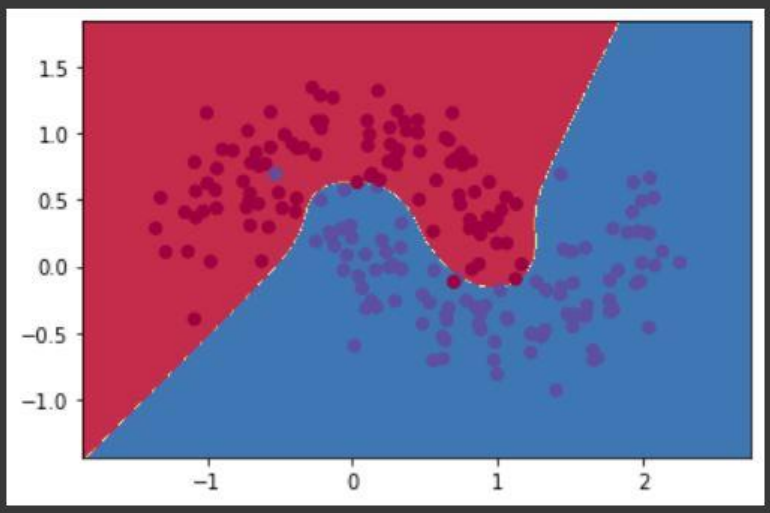


Figure 23: Training Results using **sigmoid** activation function using 3 layer neural network.

```
381     model = DeepNeuralNetwork(nn_dims=[2, 20, 8, 8], actFun_type='sigmoid')
382     model.fit_model(X,y)
383     model.visualize_decision_boundary(X,y)
384
385 if __name__ == "__main__":
386     main()
```

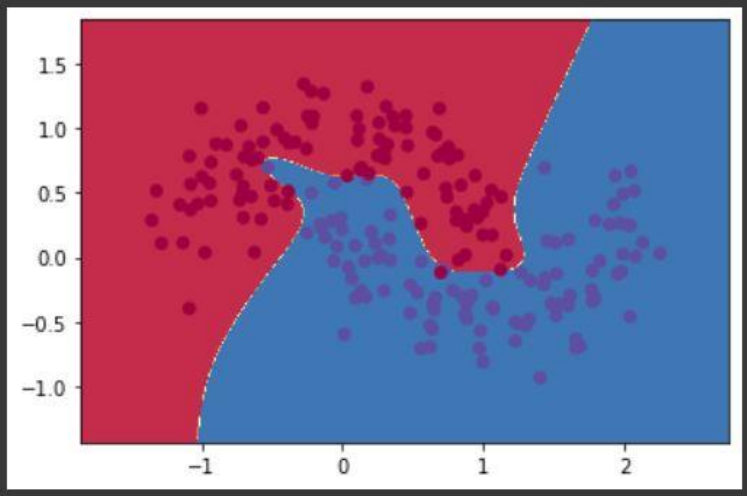


Figure 24: Training Results using **sigmoid** activation function using 4-layer neural network.

```
381     model = DeepNeuralNetwork(nn_dims=[2, 100, 8, 8, 10], actFun_type='sigmoid')
382     model.fit_model(X,y)
383     model.visualize_decision_boundary(X,y)
384
385 if __name__ == "__main__":
386     main()
```

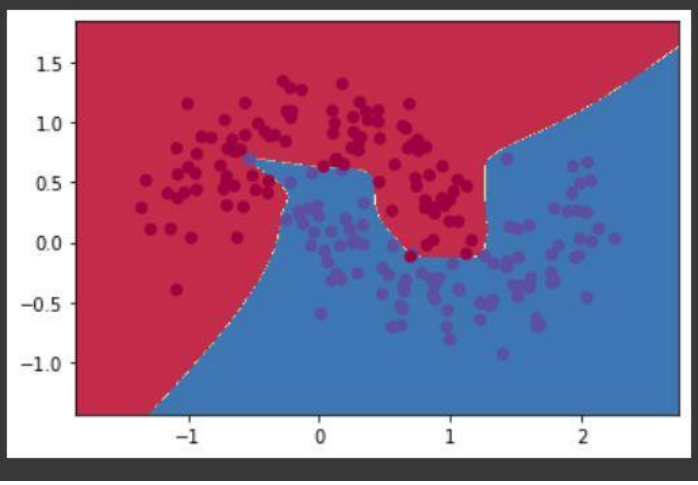


Figure 25: Training Results using **sigmoid** activation function using 5-layer neural network.

```
379     # # generate and visualize Make-Moons dataset
380     X, y = generate_data()
381
382     model = DeepNeuralNetwork(nn_dims=[2, 10, 2], actFun_type='tanh')
383     model.fit_model(X,y)
384     model.visualize_decision_boundary(X,y)
385
386 if __name__ == "__main__":
387     main()
```

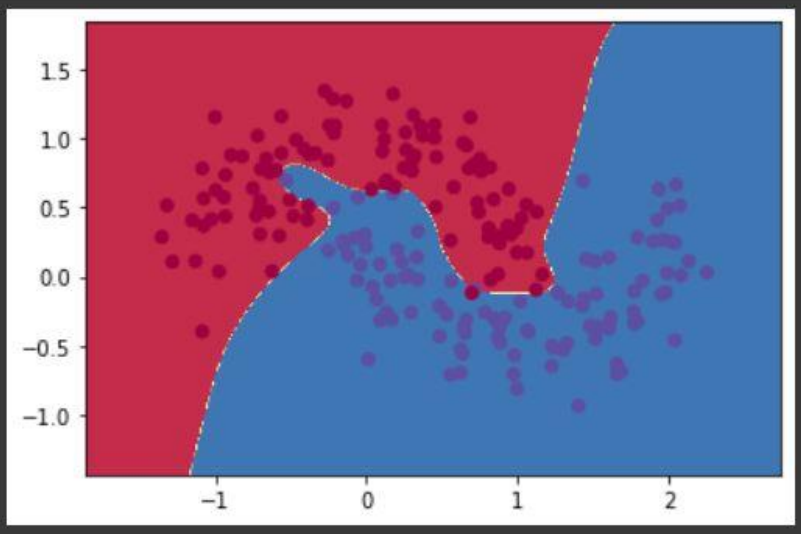


Figure 26: Training Results using **tanh** activation function using 3-layer neural network.

```
    # # generate and visualize Make-Moons dataset
    X, y = generate_data()

    #model = DeepNeuralNetwork(nn_dims=[2, 100, 2], actFun_type='sigmoid')
    model = DeepNeuralNetwork(nn_dims=[2, 4, 4, 4], actFun_type='tanh')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

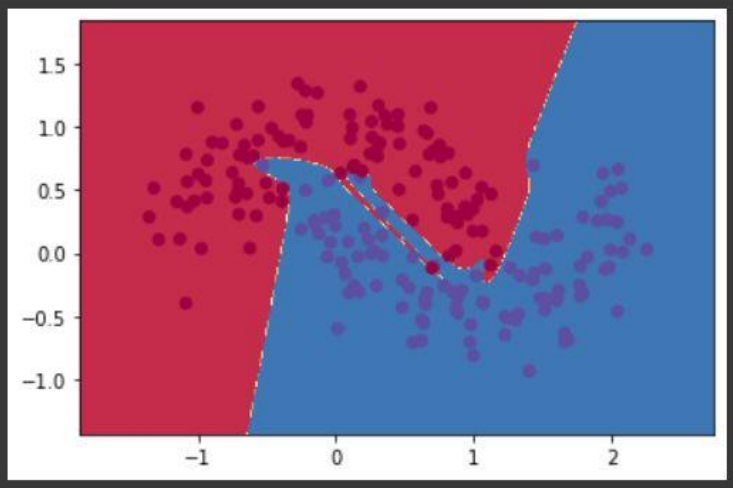


Figure 27: Training Results using **tanh** activation function using 4-layer neural network.

```python
def main():

    # # generate and visualize Make-Moons dataset
    X, y = generate_data()

    #model = DeepNeuralNetwork(nn_dims=[2, 100, 2], actFun_type='sigmoid')
    model = DeepNeuralNetwork(nn_dims=[2, 10, 10, 10,10], actFun_type='tanh')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

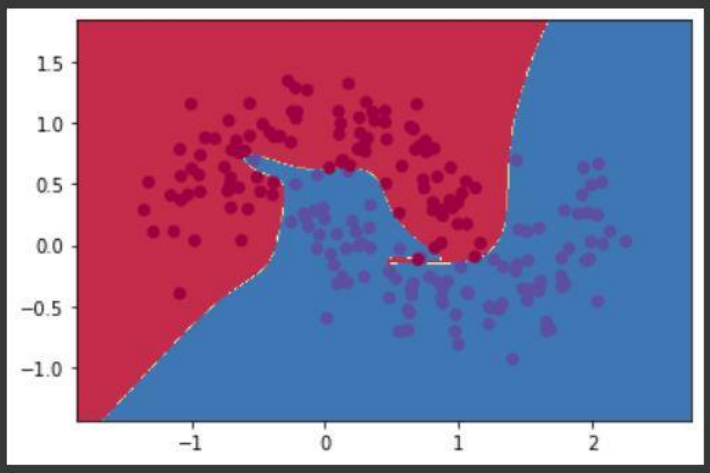


Figure 28: Training Results using **tanh** activation function using 5-layer neural network.

- Figures 29 to 34 show the **Make-Circles Dataset** using 3, 4 and 5 layer network. From figures 29 and 30 it can be seen that using relu activation function with 4 layer network has better data fit but doesn't do well when using a 5 layer network. The same behavior type can be observed with sigmoid activation function in figures 31 and 32 , with a 4 layer network the sigmoid function operates well but it doesn't do well with a 5 layer network. Figures 33 and 34 show the tanh function where using both 4 and 5 layer network not much improvement can be seen even when the hidden number of layers are increased to around 100 which is the same behavior also observed when tanh was used in a 3 layer network.

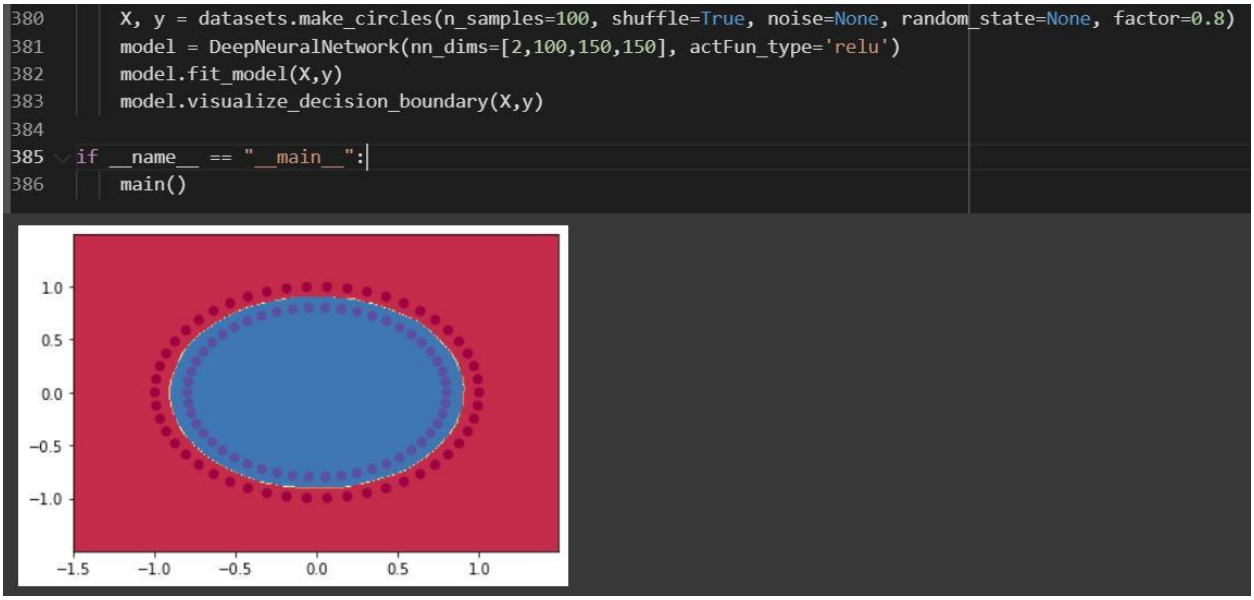


Figure 29: Training Results using **relu** activation function using 4-layer neural network.

```
    X, y = datasets.make_circles(n_samples=100, shuffle=True, noise=None, random_state=None, factor=0.8)
    model = DeepNeuralNetwork(nn_dims=[2,10,8,8,5], actFun_type='relu')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

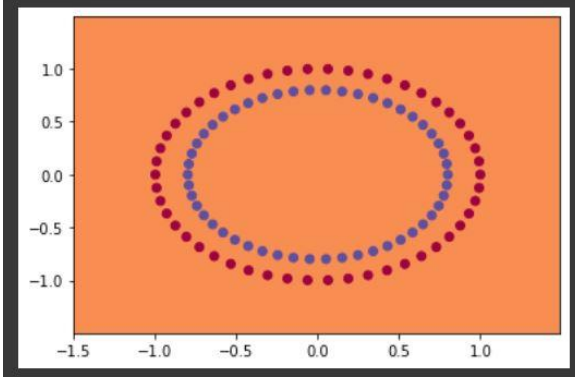


Figure 30: Training Results using **relu** activation function using 5-layer neural network.

```
    X, y = datasets.make_circles(n_samples=100, shuffle=True, noise=None, random_state=None, factor=0.8)
    model = DeepNeuralNetwork(nn_dims=[2,10,8,8], actFun_type='sigmoid')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

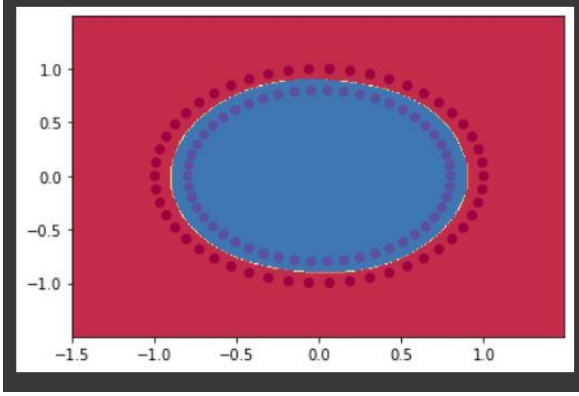


Figure 31: Training Results using **sigmoid** activation function using 4-layer neural network.

```
    X, y = datasets.make_circles(n_samples=100, shuffle=True, noise=None, random_state=None, factor=0.8)
    model = DeepNeuralNetwork(nn_dims=[2,10,8,8,5], actFun_type='sigmoid')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

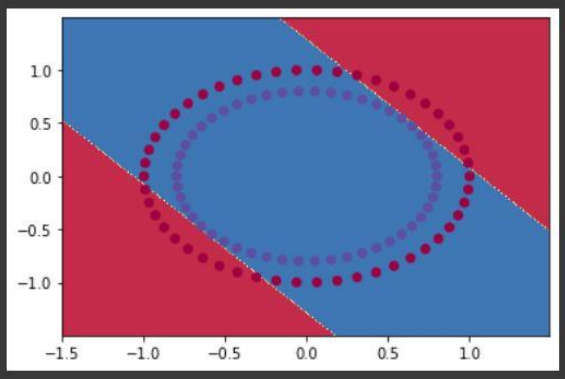


Figure 32: Training Results using **sigmoid** activation function using 5-layer neural network.

```
    X, y = datasets.make_circles(n_samples=100, shuffle=True, noise=None, random_state=None, factor=0.8)
    model = DeepNeuralNetwork(nn_dims=[2,100,20,20], actFun_type='tanh')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

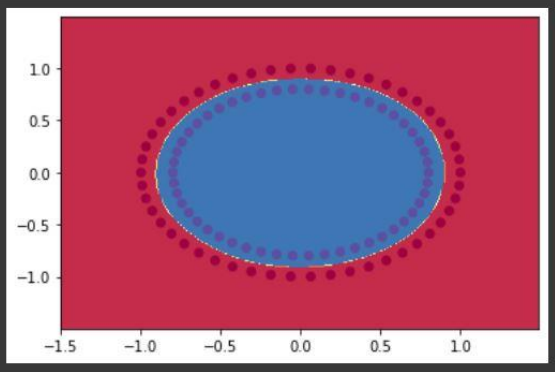


Figure 33: Training Results using **tanh** activation function using 4-layer neural network.

```
    X, y = datasets.make_circles(n_samples=100, shuffle=True, noise=None, random_state=None, factor=0.8)
    model = DeepNeuralNetwork(nn_dims=[2,10,8,8,8], actFun_type='tanh')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

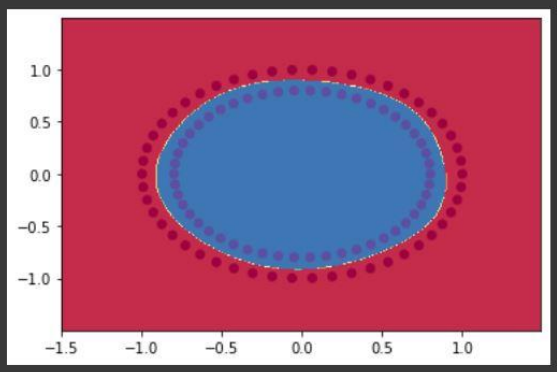


Figure 34: Training Results using **tanh** activation function using 5-layer neural network.

Problem 2:

Problem 2 Part A2:

- The code for ***weight_variable (shape)*** is shown below:

```
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
```

- The code for ***bias_variable (shape)*** is shown below:

```
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
```

- Th code for ***conv2d (x,W)*** is shown below:

```
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
```

- Th code for ***max_pool_2x2(x)*** is shown below:

```
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
```

Problem 2 Part A3:

- Code for input data and input labels:

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

    # Store Accuracies
    val_accuracy_ = tf.placeholder(tf.float32, shape=())
    test_accuracy_ = tf.placeholder(tf.float32, shape=())

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

- Code for First Convolution layer:

```
# first convolutional layer
    W_conv1 = weight_variable([5, 5, 1, 32])
    b_conv1 = bias_variable([32])
    input_1 = conv2d(x_image, W_conv1) + b_conv1
    h_conv1 = tf.nn.relu(input_1)
    h_pool1 = max_pool_2x2(h_conv1)

    # second convolutional layer
    W_conv2 = weight_variable([5, 5, 32, 64])
    b_conv2 = bias_variable([64])
    input_2 = conv2d(h_pool1, W_conv2) + b_conv2
    h_conv2 = tf.nn.relu(input_2)
    h_pool2 = max_pool_2x2(h_conv2)
```

- Code for densely connected layer:

```
# 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])
    input_fc1 = tf.matmul(h_pool2_flat, W_fc1) + b_fc1
    h_fc1 = tf.nn.relu(input_fc1)
```

- Code for Dropout Layer:

```
    # dropout
    keep_prob = tf.placeholder(tf.float32)
```

```
    h_fc1_drop = h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

- Code for Softmax:

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

Problem 2 Part A4:

- Code for Training setup:

```
    # setup training
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv),
reduction_indices=[1]))
    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),
name='accuracy')

    # Add a scalar summary for the snapshot loss.
    tf.summary.scalar(cross_entropy.op.name, cross_entropy)
    # Build the summary operation based on the TF collection of Summaries.
    # Layer-1
    stats_summary("W_conv1", W_conv1)
    stats_summary("b_conv1", b_conv1)
    stats_summary("input_1", input_1)
    stats_summary("h_conv1", h_conv1)
    stats_summary(" h_pool1", h_pool1)
    # Layer-2
    stats_summary("W_conv2", W_conv2)
    stats_summary("b_conv2", b_conv2)
    stats_summary("input_2", input_2)
    stats_summary("h_conv2", h_conv2)
    stats_summary("h_pool2", h_pool2)
    # densely connected layer
    stats_summary("W_fc1", W_fc1)
    stats_summary("b_fc1", b_fc1)
    stats_summary("h_pool2_flat", h_pool2_flat)
    stats_summary("input_fc1", input_fc1)
    stats_summary("h_fc1", h_fc1)
    # Output layer - Softmax
    stats_summary("W_fc2", W_fc2)
    stats_summary("b_fc2", b_fc2)
    stats_summary("y_conv ", y_conv)

    summary_op = tf.summary.merge_all()
    summary_op_test = tf.summary.scalar('test_accuracy', test_accuracy_)
    summary_op_val = tf.summary.scalar('val_accuracy', val_accuracy_)

    # Add the variable initializer Op.
    init = tf.initialize_all_variables()
```

```
    # Create a saver for writing training checkpoints.
    saver = tf.train.Saver()

    # Instantiate a SummaryWriter to output summaries and the Graph.
    summary_writer = tf.summary.FileWriter(result_dir, sess.graph)
    # Val test
    summary_writer_test = tf.summary.FileWriter(test_dir, sess.graph)
    summary_writer_val = tf.summary.FileWriter(val_dir, sess.graph)

    # Run the Op to initialize the variables.
    sess.run(init)
```

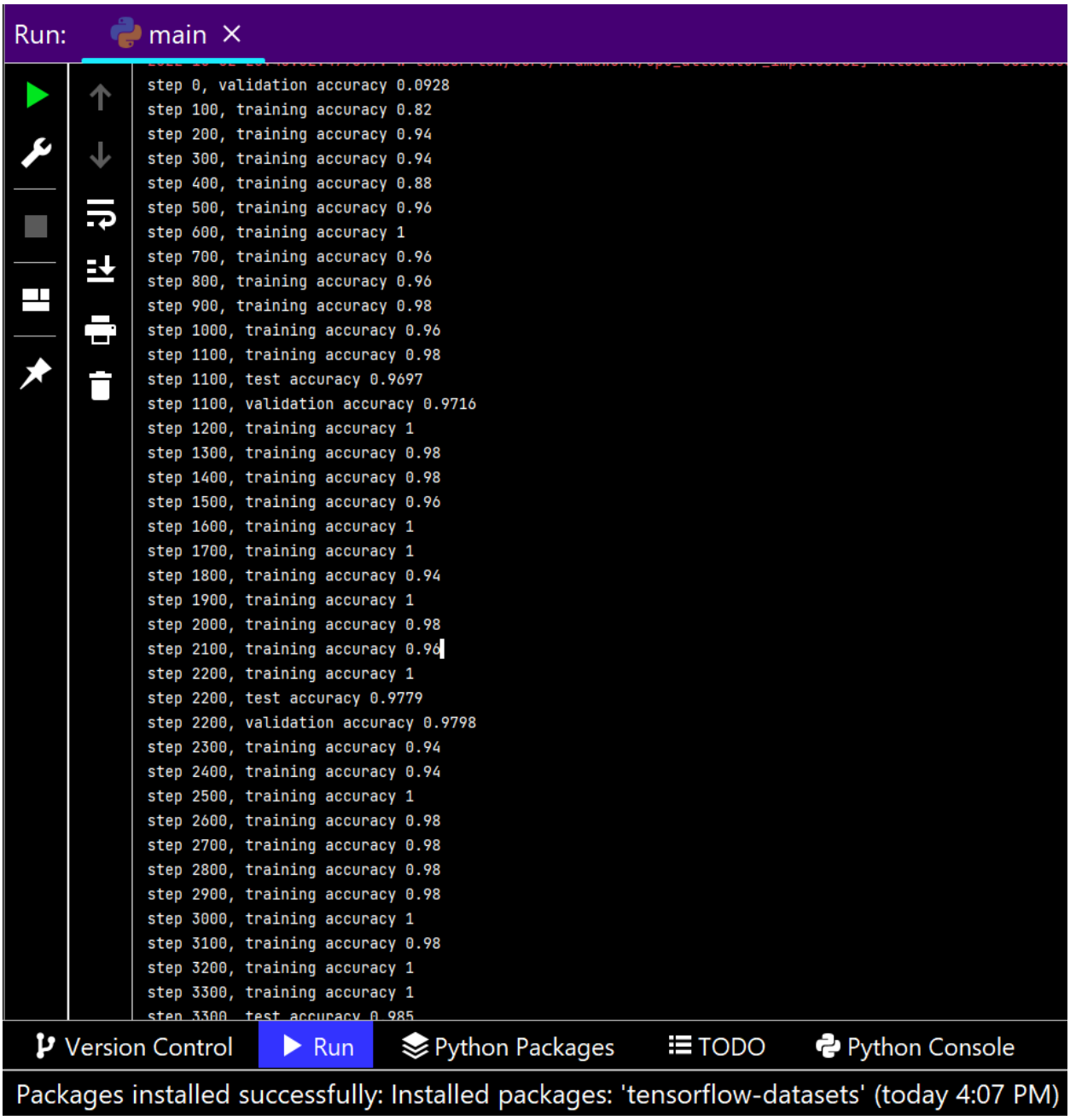
Problem 2 Part A5:

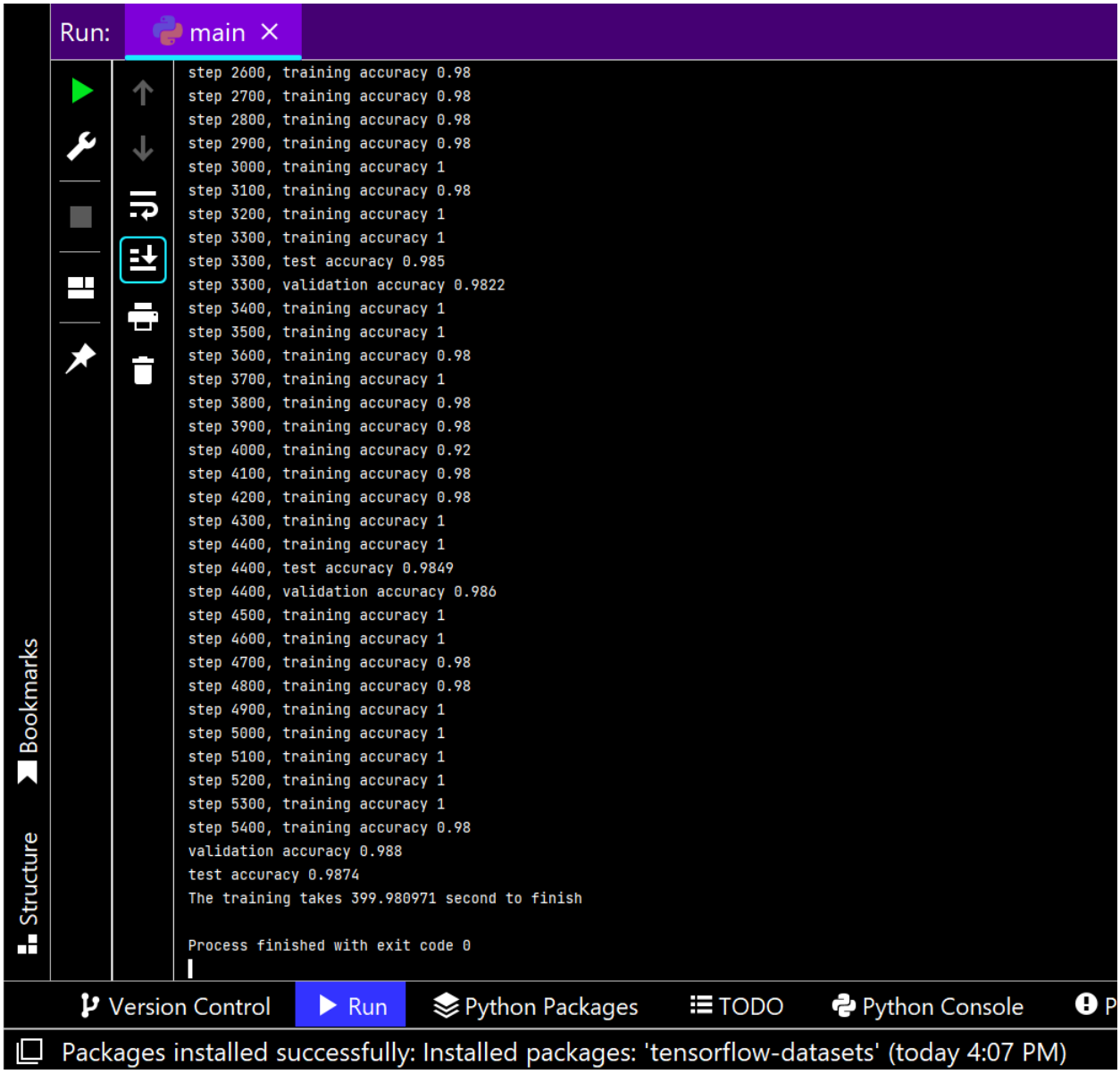
- Code running the training:

```
# run the training
    for i in range(max_step):
        batch = mnist.train.next_batch(50)  # make the data batch, which is
used in the training iteration.
        # the batch size is 50
        if i % 100 == 0:
            # output the training accuracy every 100 iterations
            train_accuracy = accuracy.eval(feed_dict={
                x: batch[0], y_: batch[1], keep_prob: 1.0})
            print("step %d, training accuracy %g" % (i, train_accuracy))

            # Update the events file which is used to monitor the training (in
this case,
            # only the training loss is monitored)
            summary_str = sess.run(summary_op, feed_dict={x: batch[0], y_:
batch[1], keep_prob: 0.5})
            summary_writer.add_summary(summary_str, i)
            summary_writer.flush()

        # save the checkpoints every 1100 iterations
        if i % 1100 == 0 or i == max_step:
            checkpoint_file = os.path.join(result_dir, 'checkpoint')
            saver.save(sess, checkpoint_file, global_step=i)

            feed_dict_test = {x: mnist.test.images, y_: mnist.test.labels,
keep_prob: 1.0}
            test_accuracy = accuracy.eval(feed_dict=feed_dict_test)
            print('step %d, test accuracy %g' % (i, test_accuracy))
            summary_str_test = sess.run(summary_op_test,
feed_dict={test_accuracy_: test_accuracy})
            summary_writer_test.add_summary(summary_str_test, i)
            summary_writer_test.flush()

            feed_dict_val = {x: mnist.validation.images, y_:
mnist.validation.labels, keep_prob: 1.0}
            val_accuracy = accuracy.eval(feed_dict=feed_dict_val)
            print('step %d, validation accuracy %g' % (i, val_accuracy))
            summary_str_val = sess.run(summary_op_val,
feed_dict={val_accuracy_: val_accuracy})
            summary_writer_val.add_summary(summary_str_val, i)
```

```
            summary_writer_val.flush()

        train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
# run one train_step
```

- Below shows the results of the training:



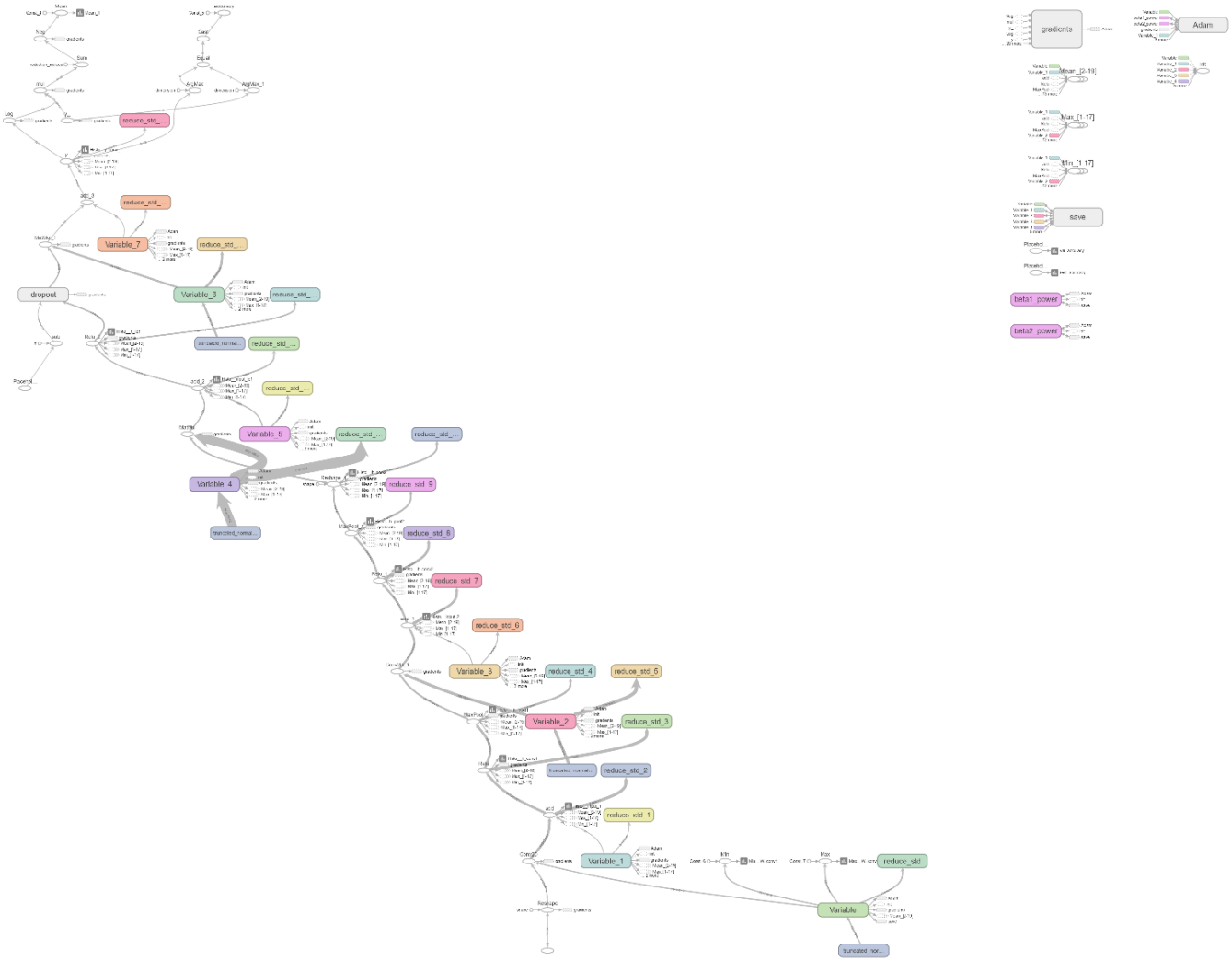


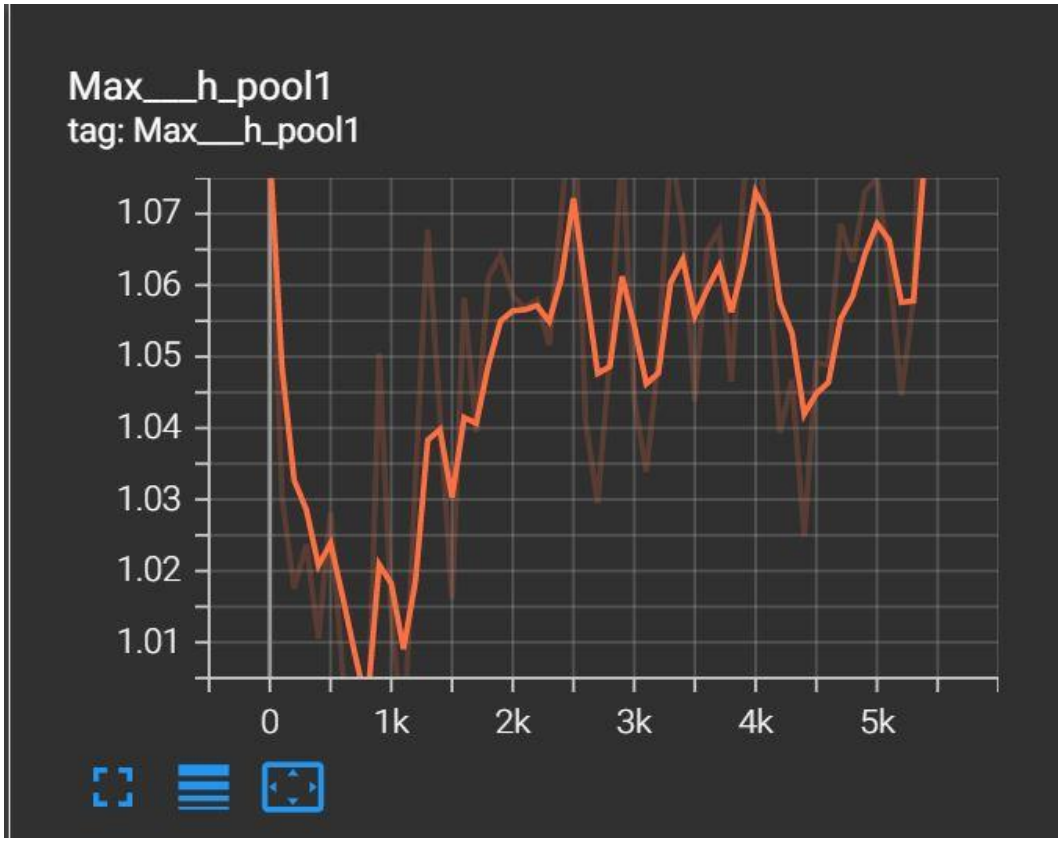
- Based on the figure above it can be seen that the training was completed after 5500 steps at **98.74% test accuracy** and it took 400 seconds to complete the training.

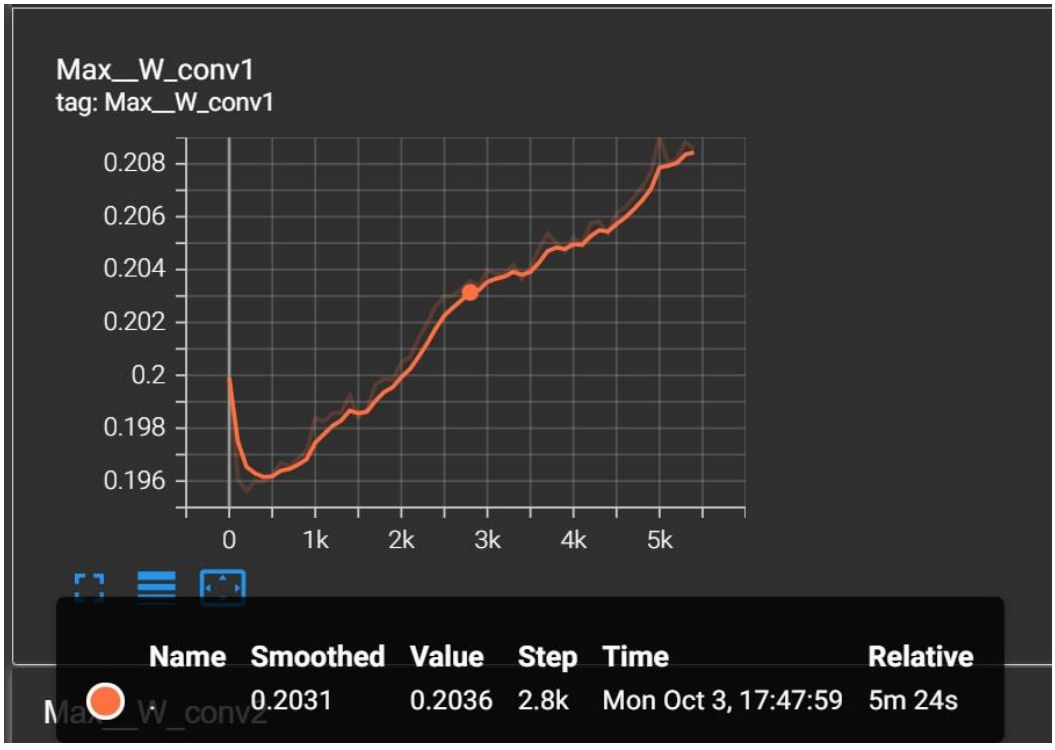
Problem 2 Part A6:

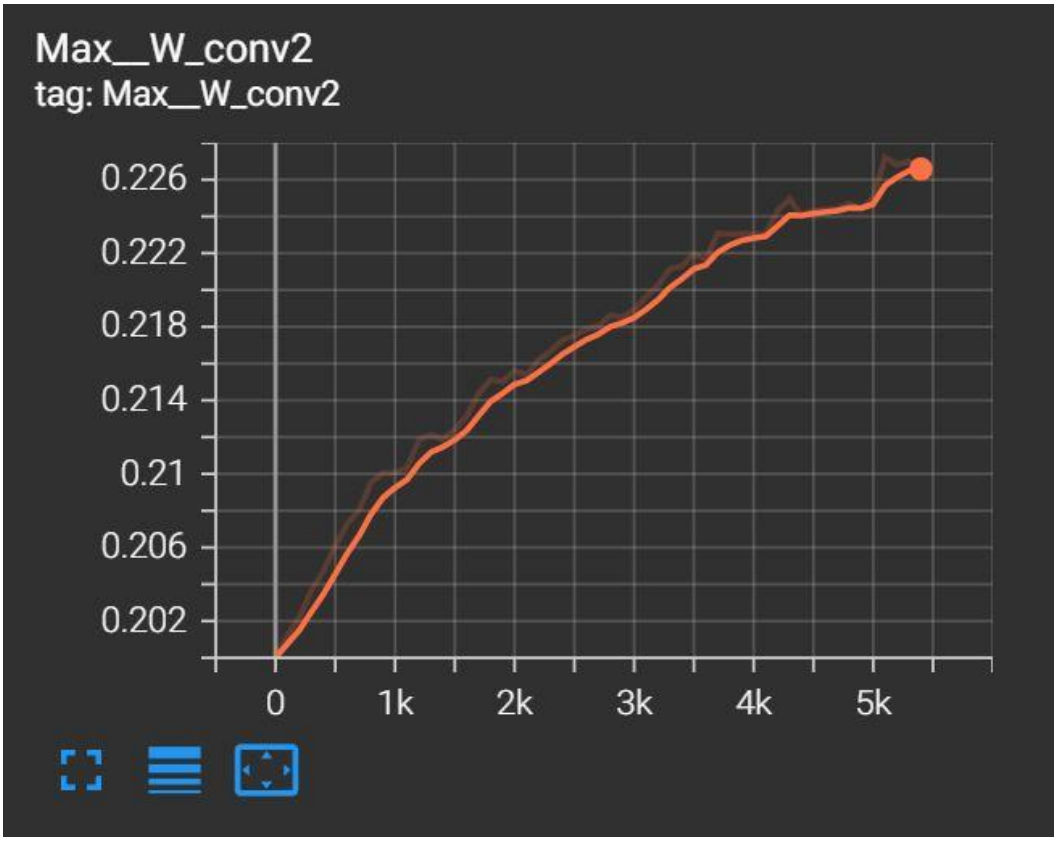
- The images below are results from training using **tensorboard**. The command used to run tensorboard and then accessed at http://localhost:6006 :

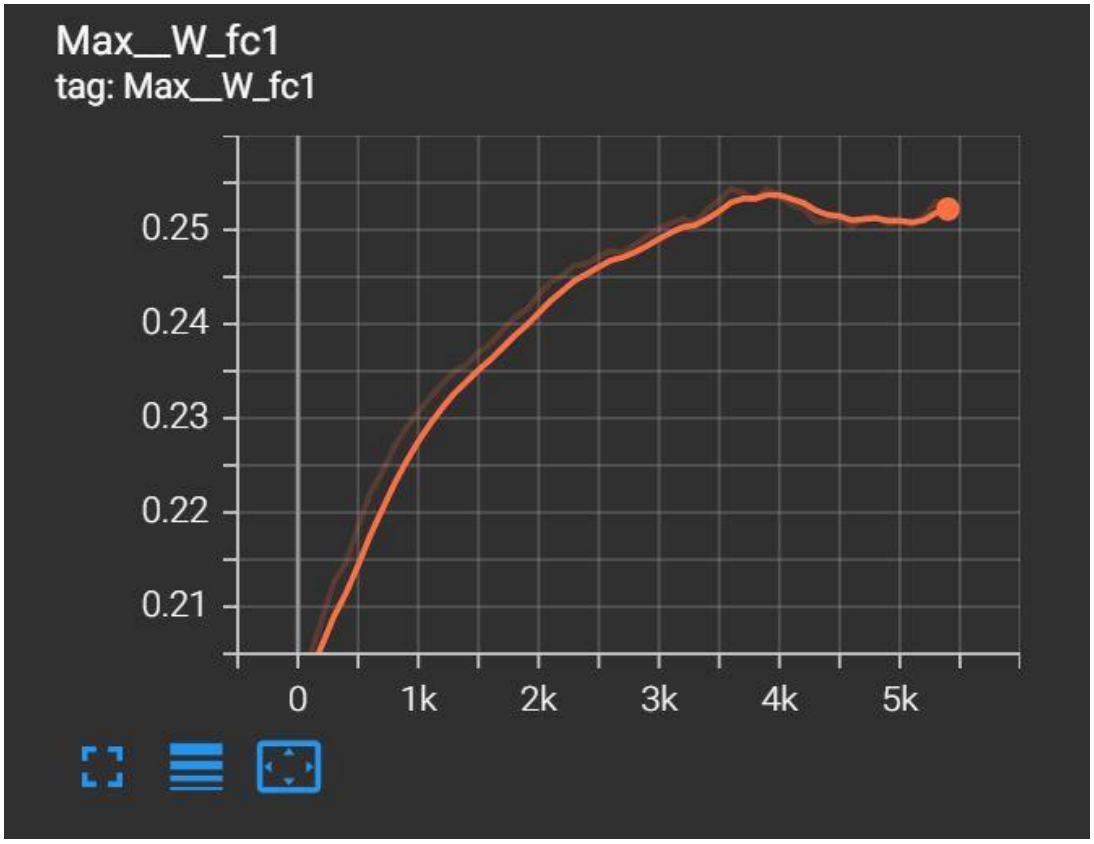
```
tensorboard --logdir=results --bind_all
```

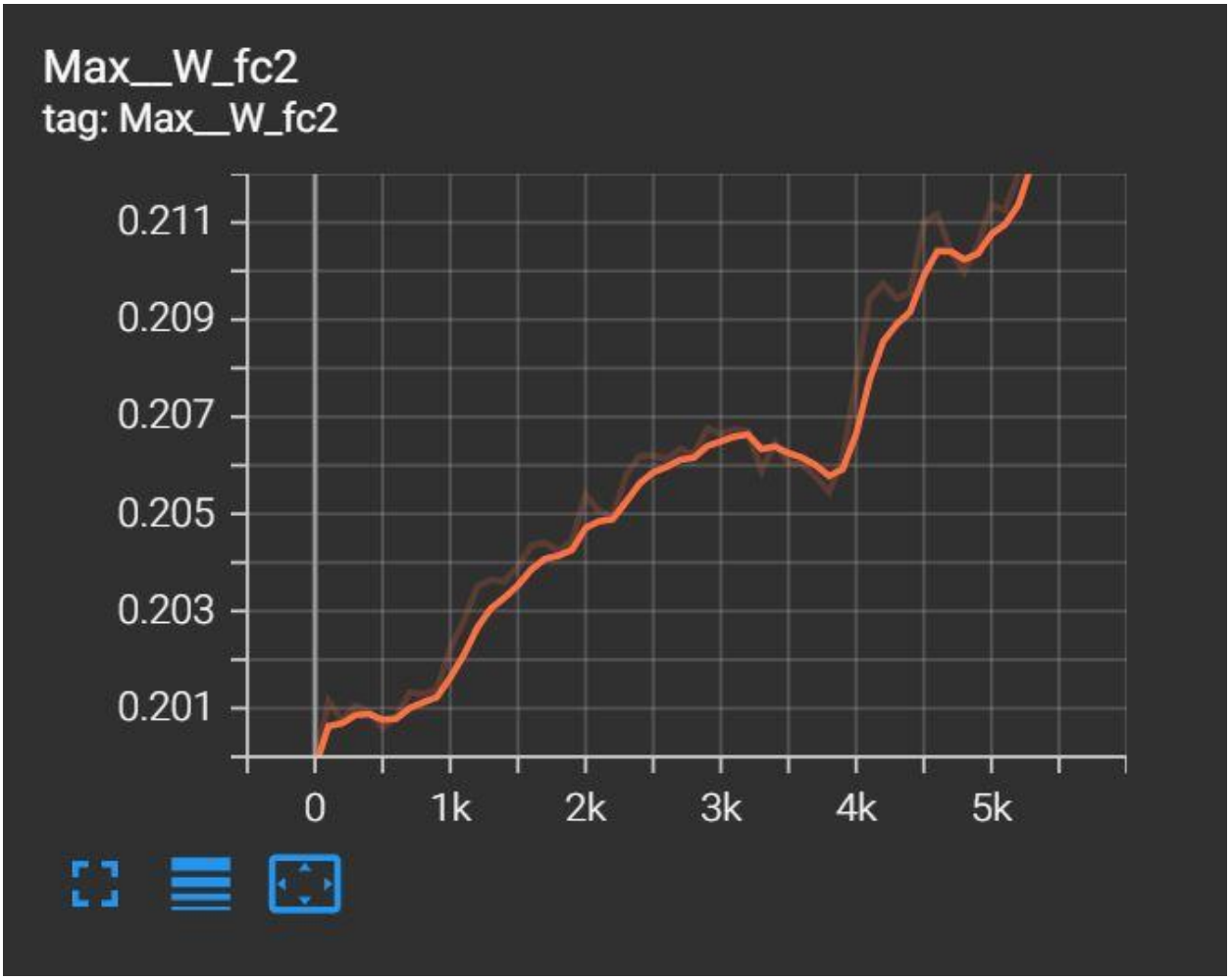


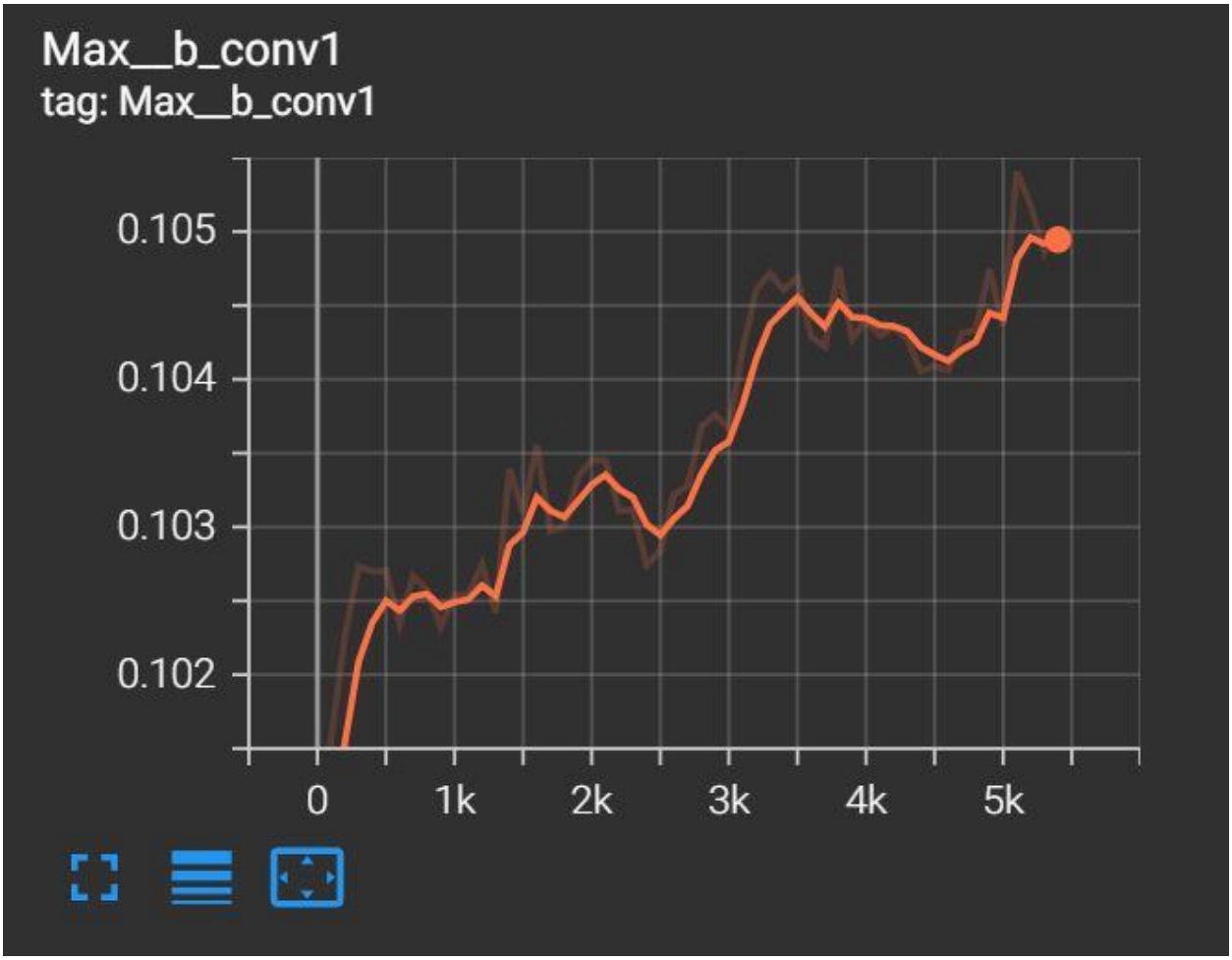

Max___h_pool1
tag: Max___h_pool1
1.07
1.06
1.05
1.04
1.03
1.02
1.01
0
1k
2k
3k
4k
5k

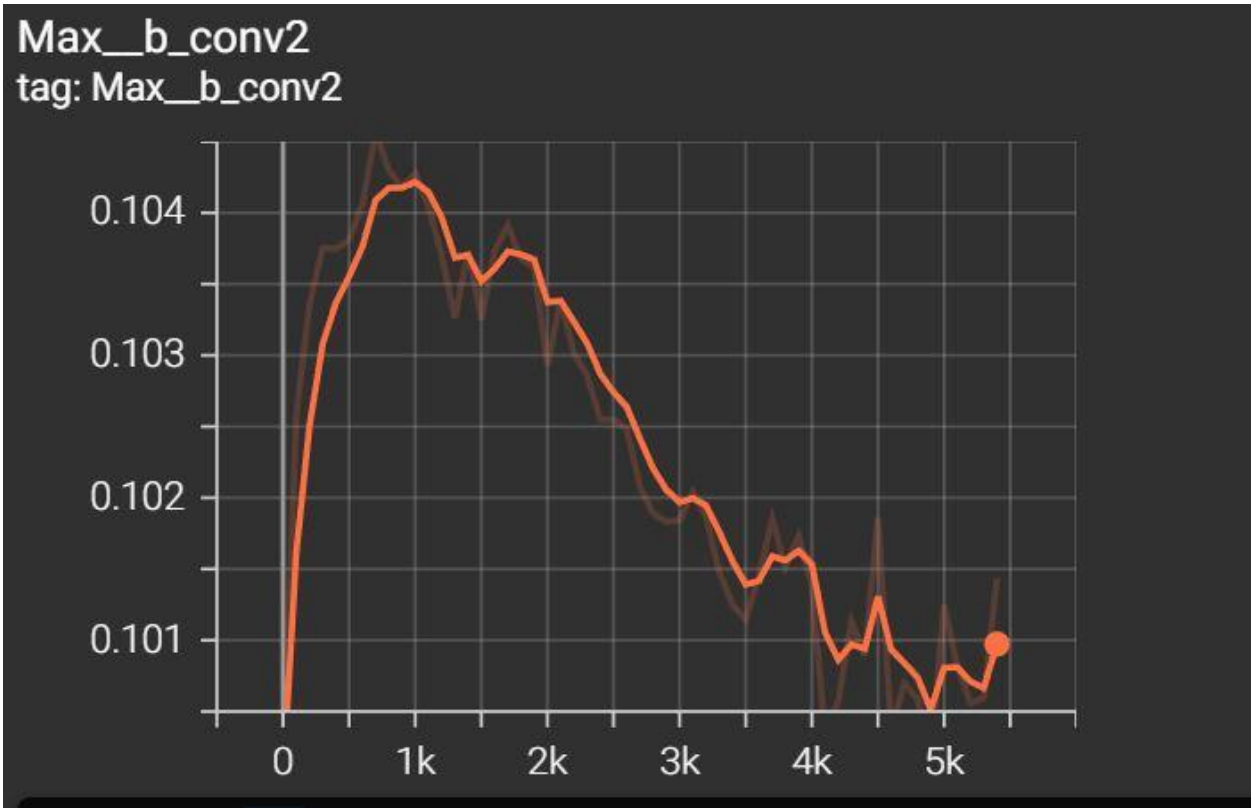

Max__W_conv1
tag: Max__W_conv1
0.208
0.206
0.204
0.202
0.2
0.198
0.196
0
1k
2k
3k
4k
5k
Name Smoothed Value Step Time Relative
0.2031 0.2036 2.8k Mon Oct 3, 17:47:59 5m 24s

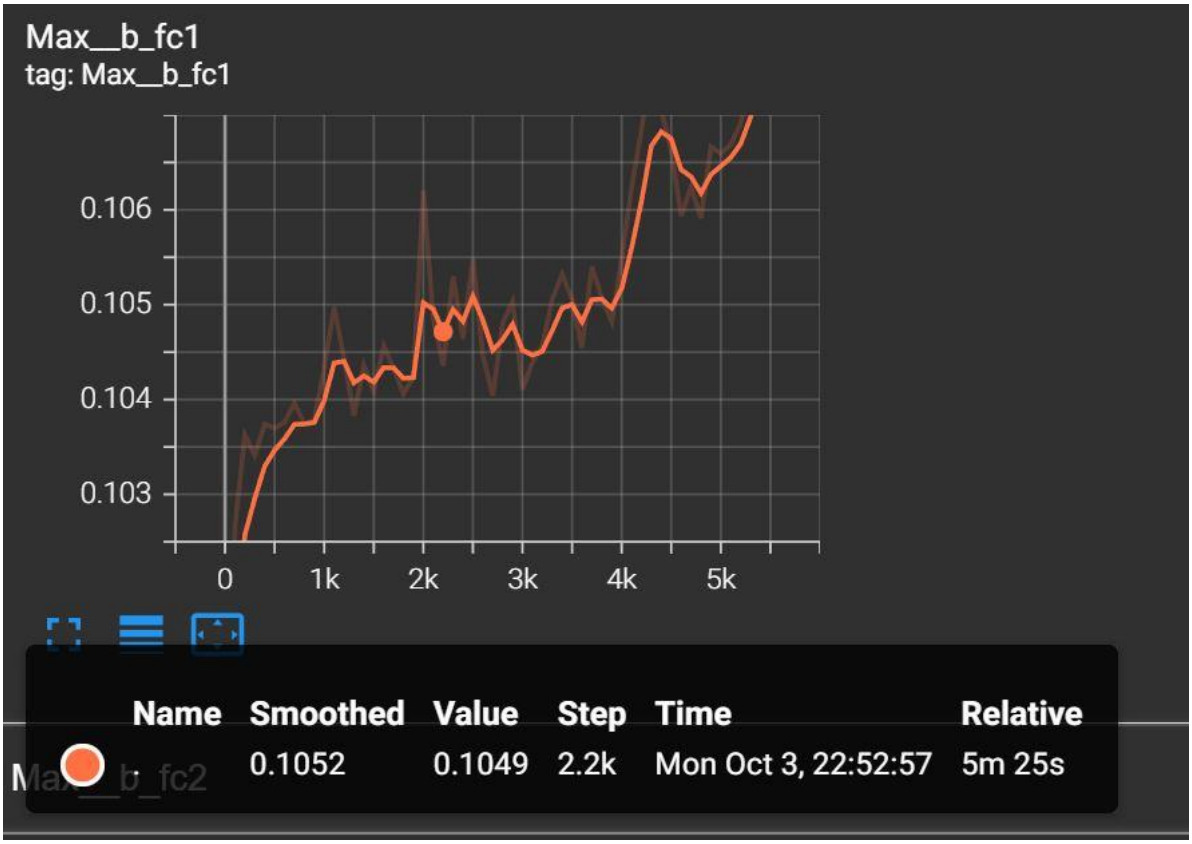

Max__W_conv2
tag: Max__W_conv2
0.226
0.222
0.218
0.214
0.21
0.206
0.202
0
1k
2k
3k
4k
5k

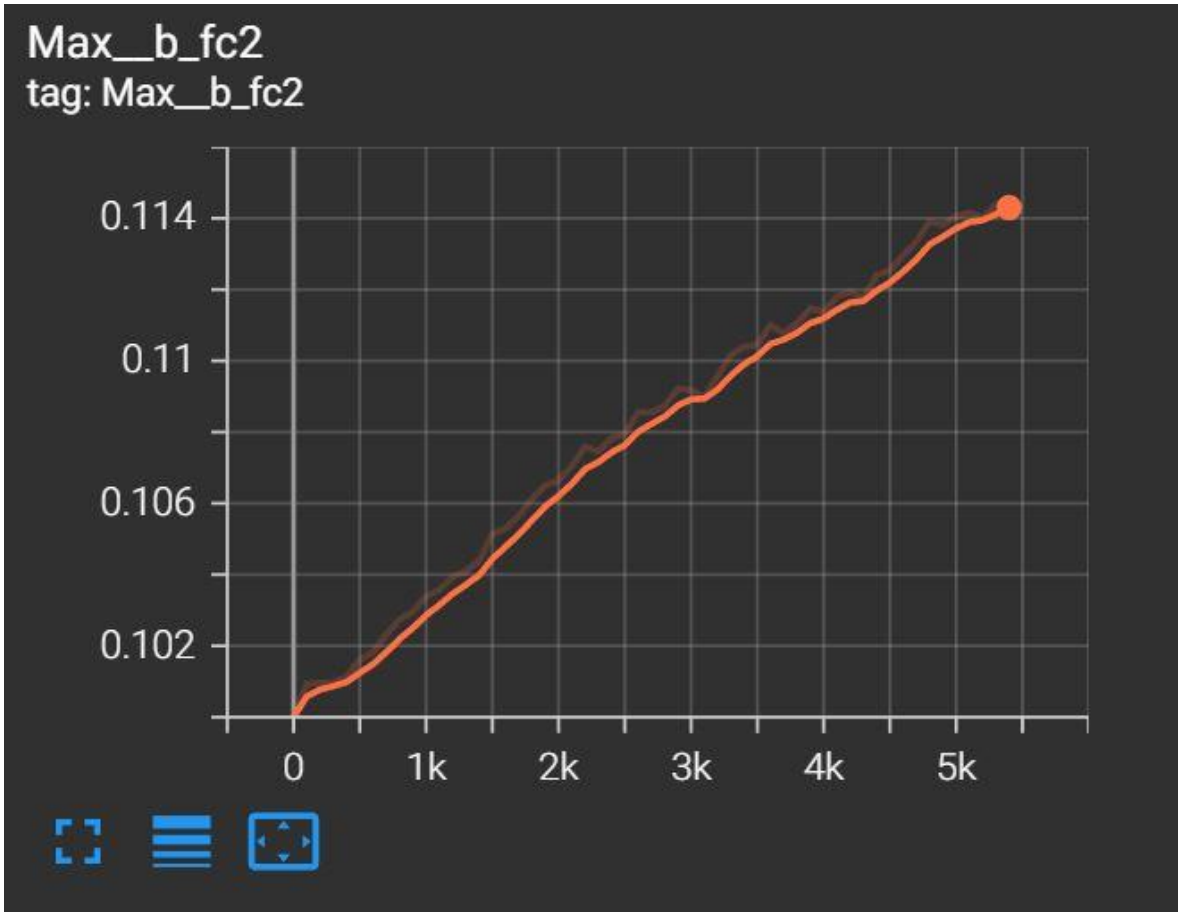

Max__W_fc1
tag: Max__W_fc1
0.25
0.24
0.23
0.22
0.21
0
1k
2k
3k
4k
5k

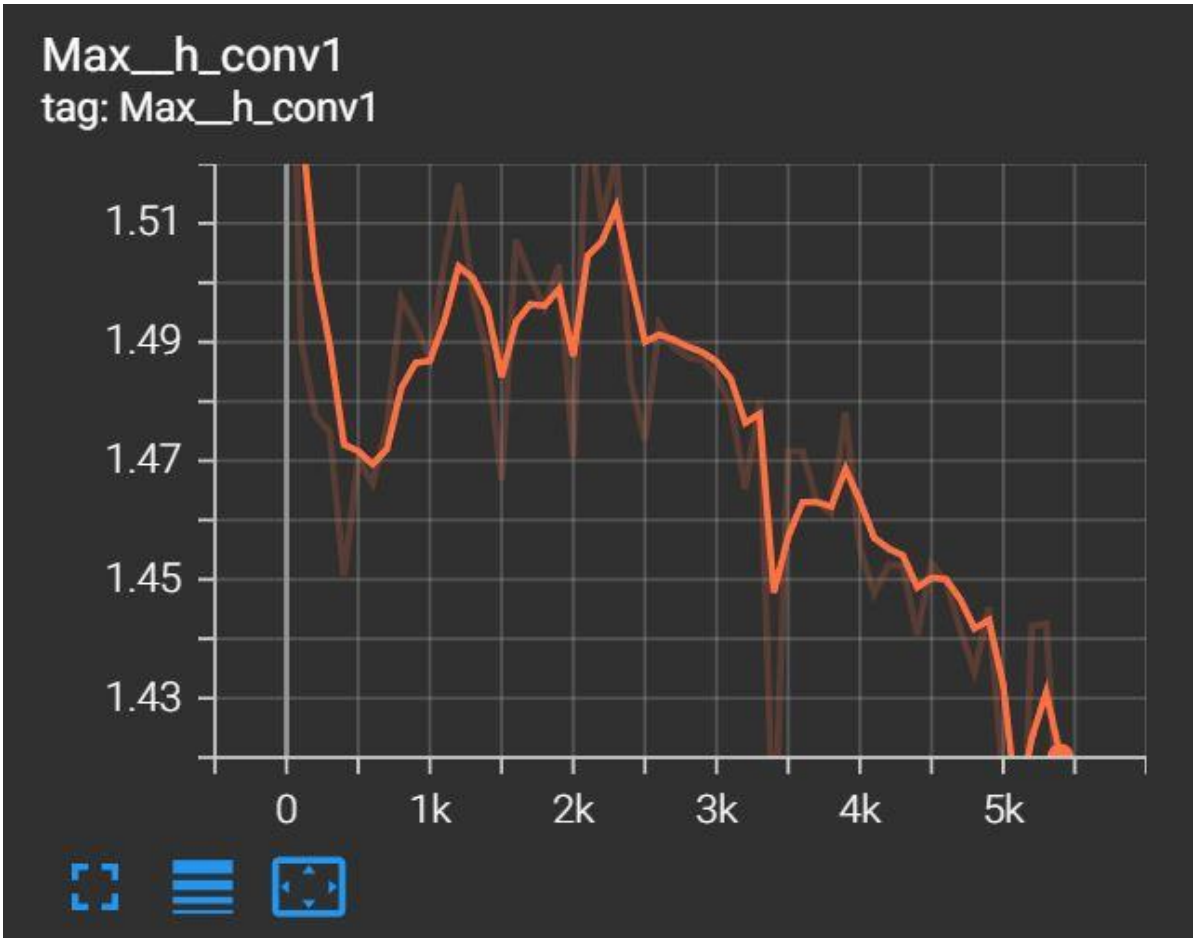

Max__W_fc2
tag: Max__W_fc2
0.211
0.209
0.207
0.205
0.203
0.201
0
1k
2k
3k
4k
5k

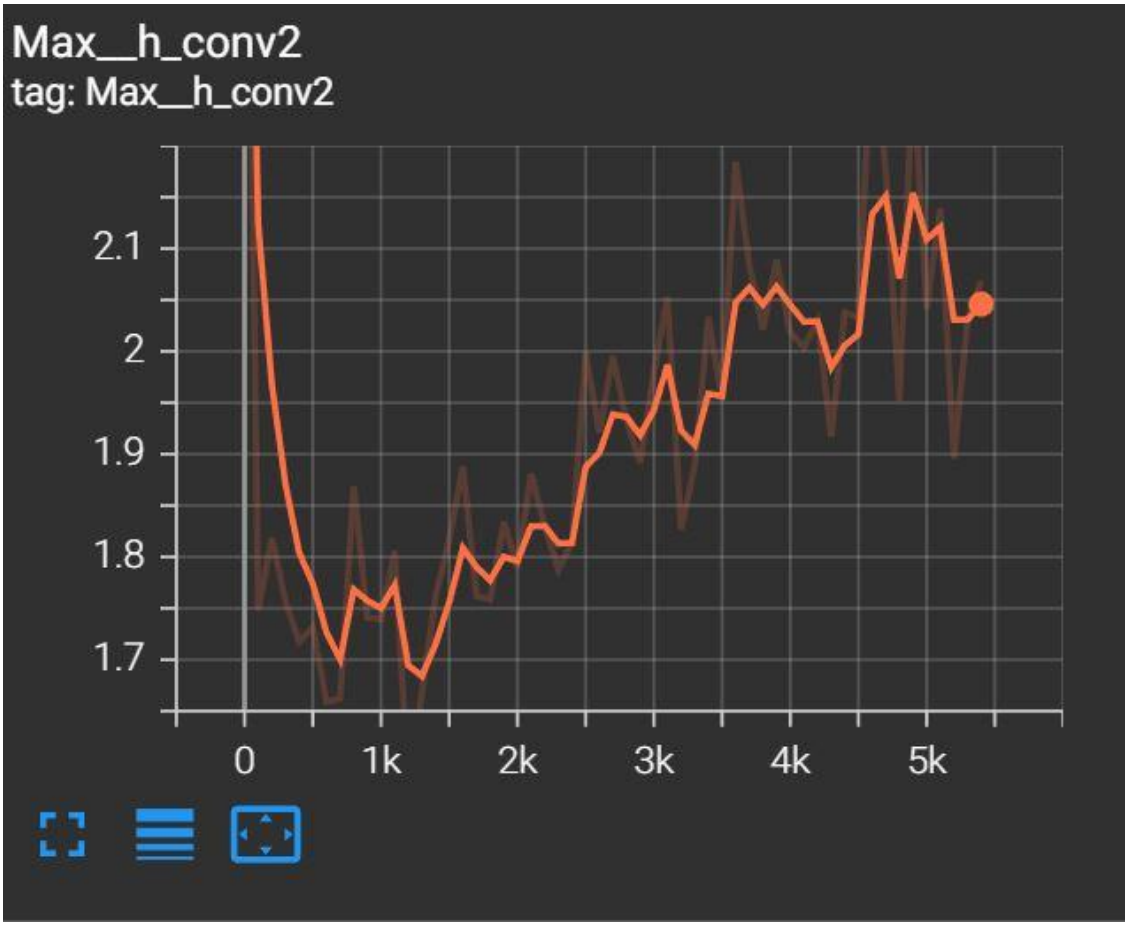

Max__b_conv1
tag: Max__b_conv1
0.105
0.104
0.103
0.102
0
1k
2k
3k
4k
5k

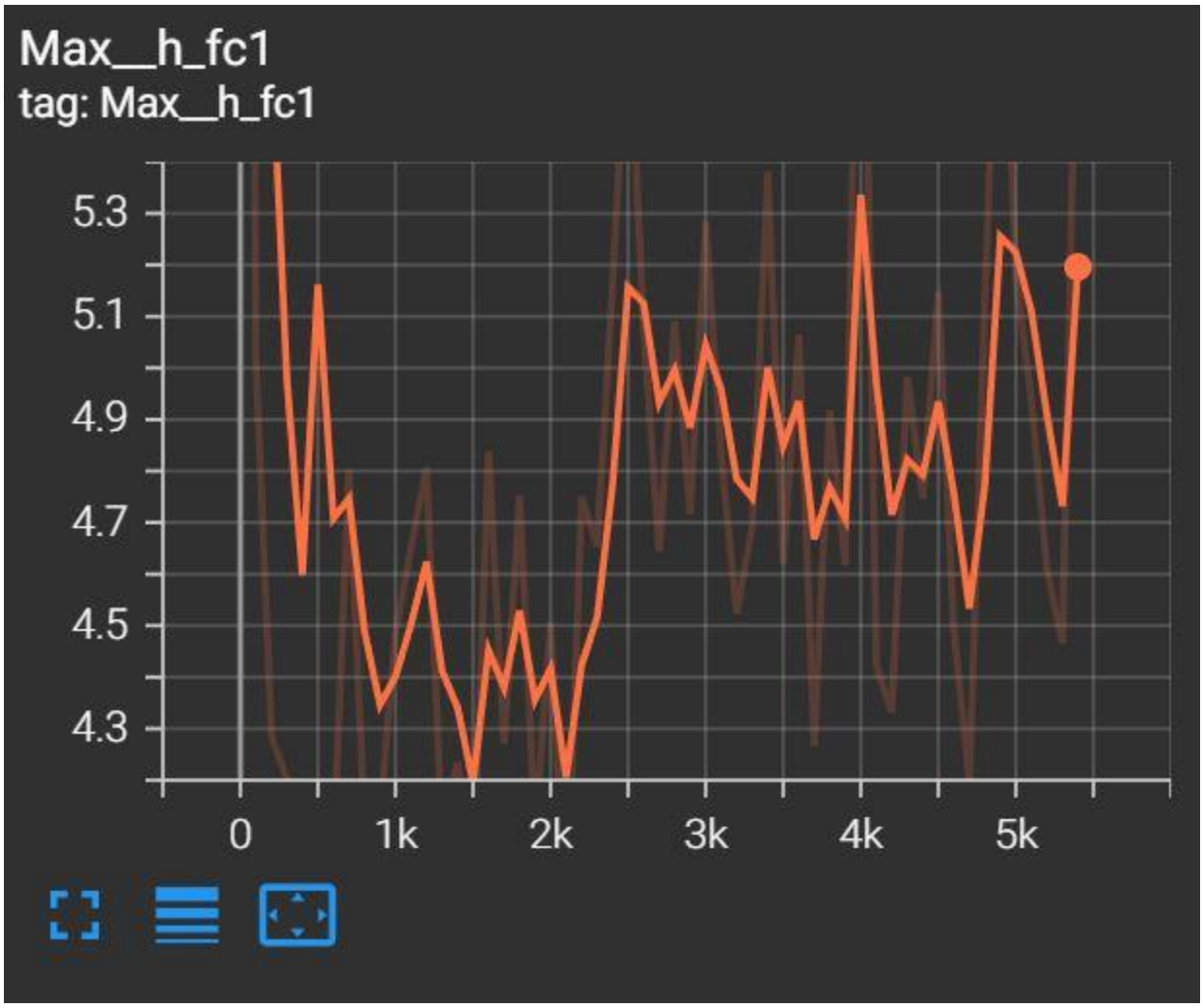

Max__b_conv2
tag: Max__b_conv2
0.104
0.103
0.102
0.101
0
1k
2k
3k
4k
5k

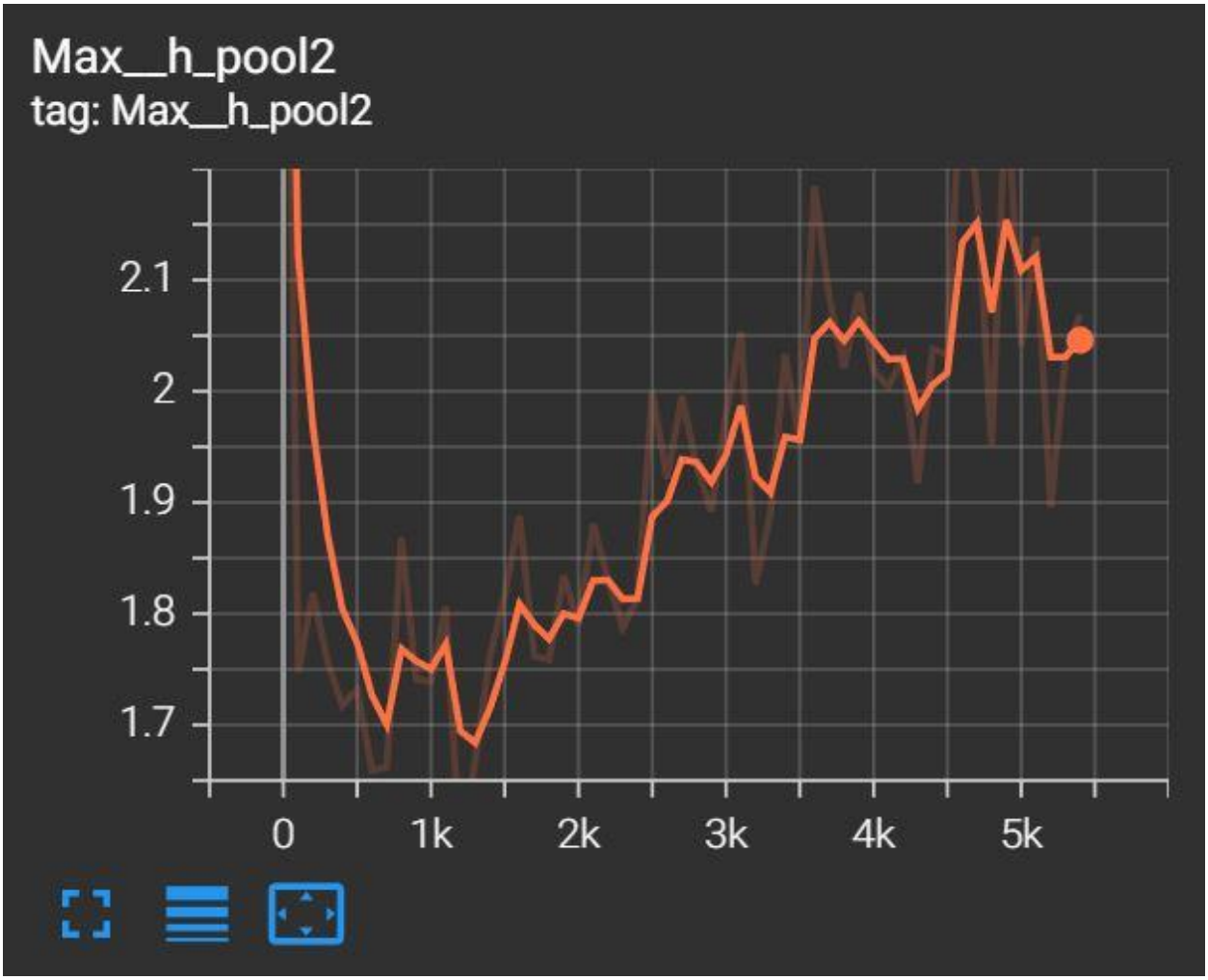

Max__b_fc1
tag: Max__b_fc1
0.106
0.105
0.104
0.103
0
1k
2k
3k
4k
5k
Name Smoothed Value Step Time Relative
0.1052 0.1049 2.2k Mon Oct 3, 22:52:57 5m 25s

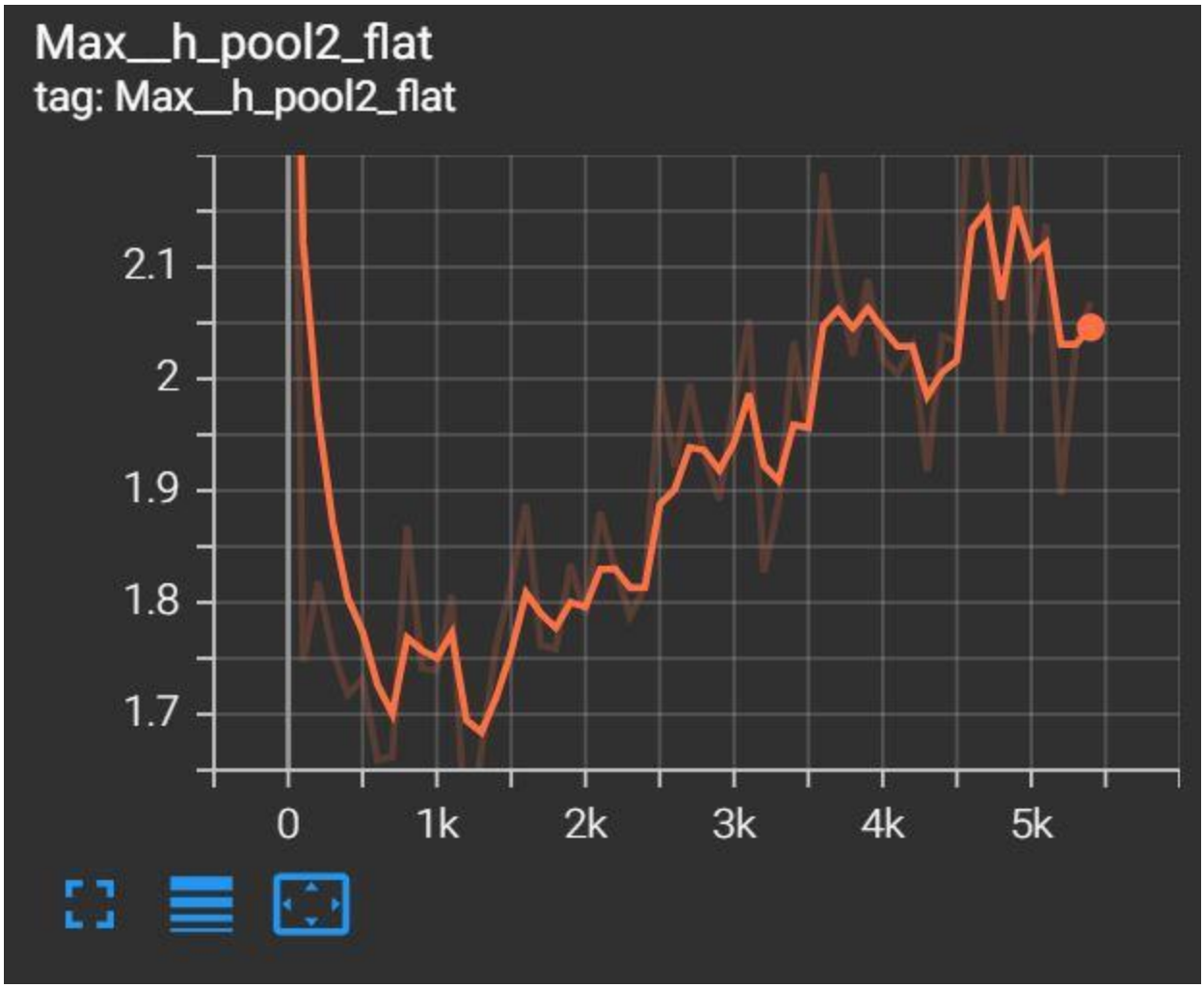

Max__b_fc2
tag: Max__b_fc2
0.114
0.11
0.106
0.102
0
1k
2k
3k
4k
5k


Max__h_conv1
tag: Max__h_conv1
1.51
1.49
1.47
1.45
1.43
0
1k
2k
3k
4k
5k

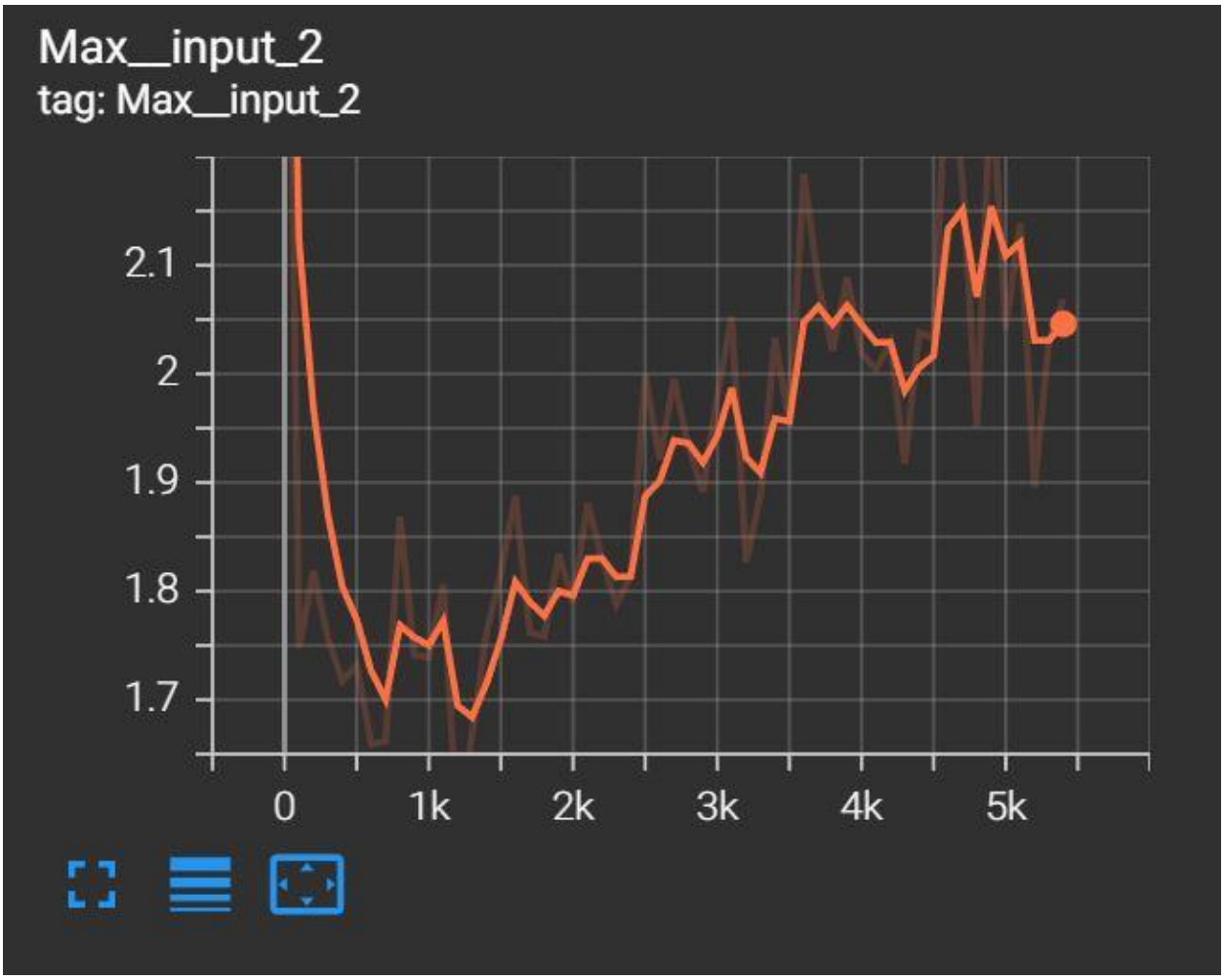

Max__h_conv2
tag: Max__h_conv2
2.1
2
1.9
1.8
1.7
0
1k
2k
3k
4k
5k

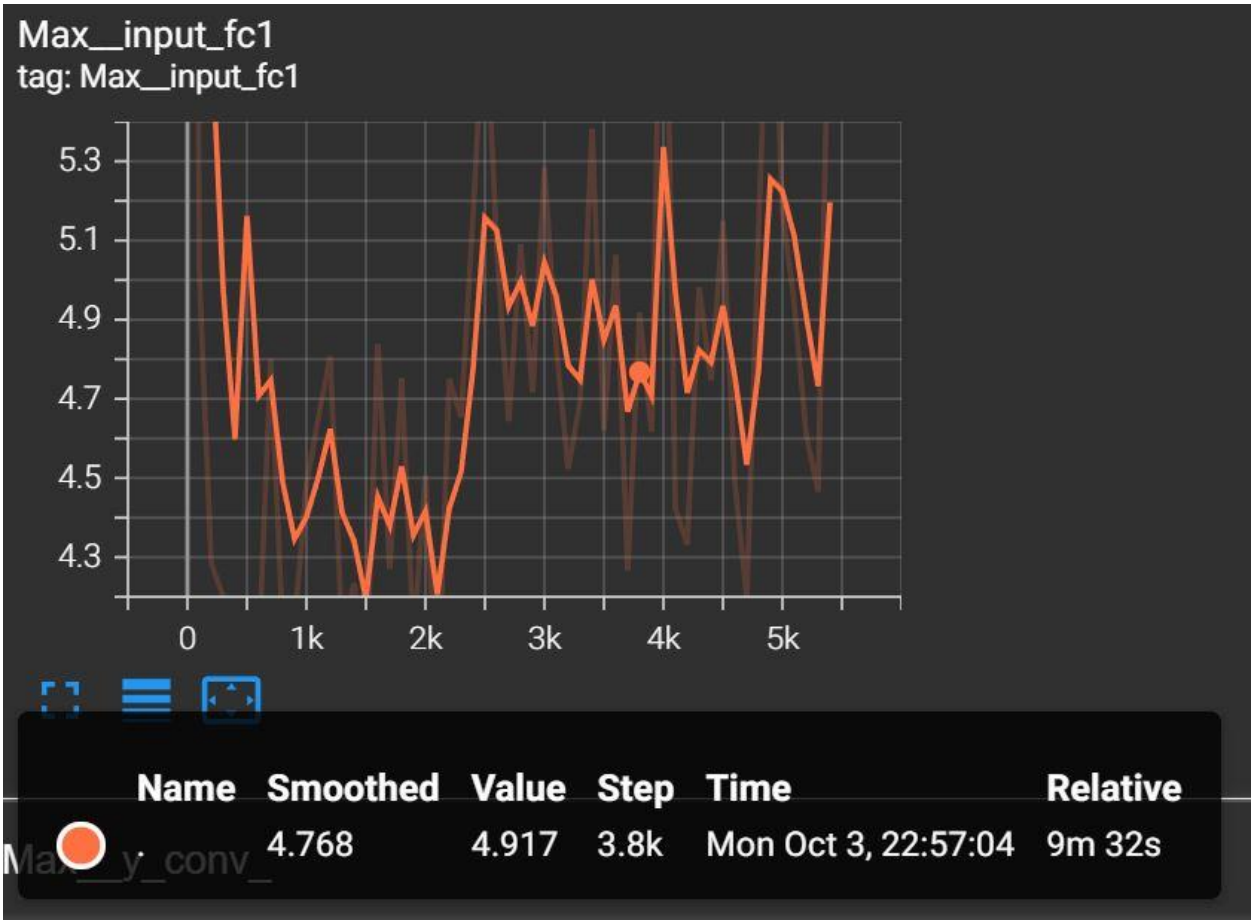

Max__h_fc1
tag: Max__h_fc1
5.3
5.1
4.9
4.7
4.5
4.3
0
1k
2k
3k
4k
5k

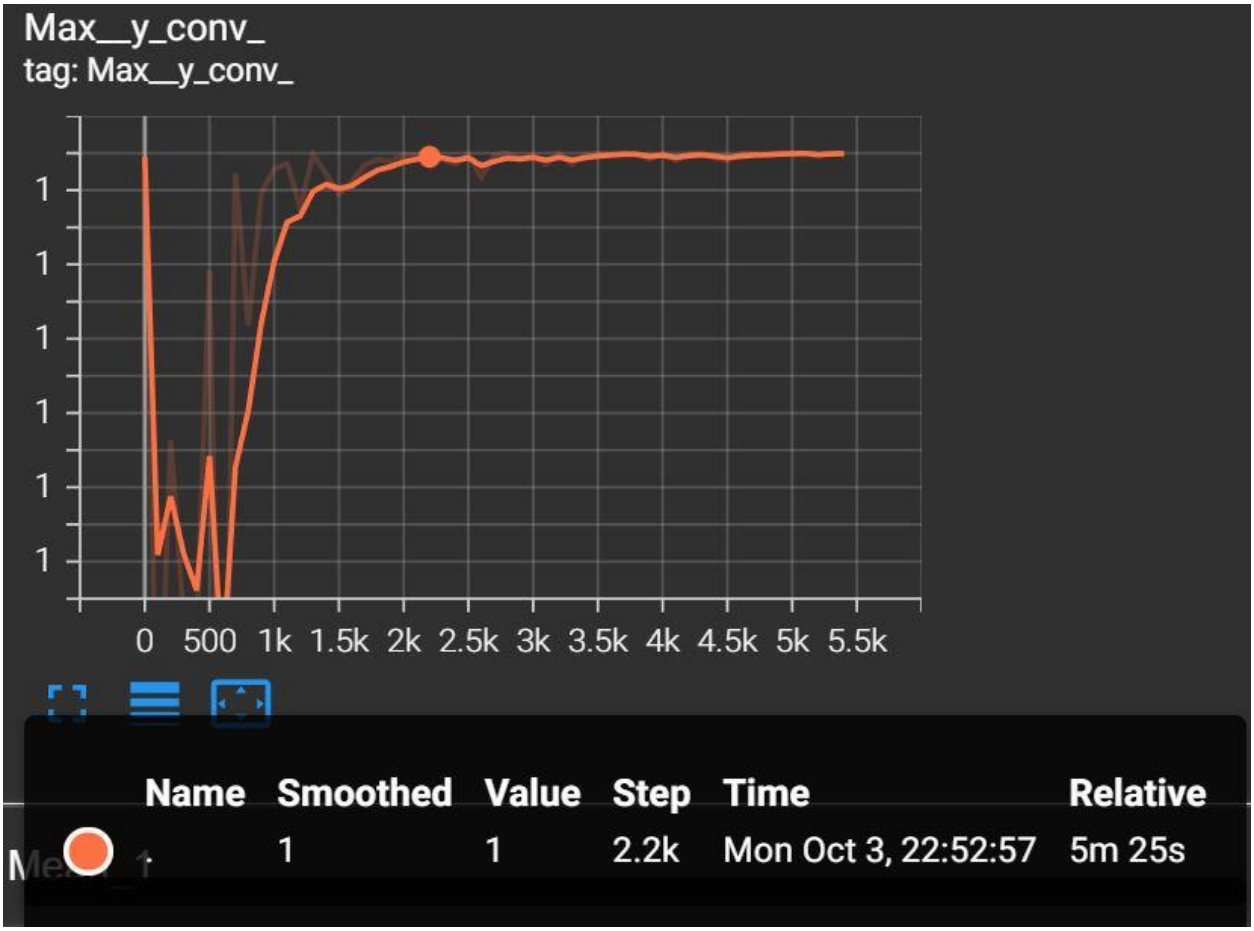

Max__h_pool2
tag: Max__h_pool2
2.1
2
1.9
1.8
1.7
0
1k
2k
3k
4k
5k

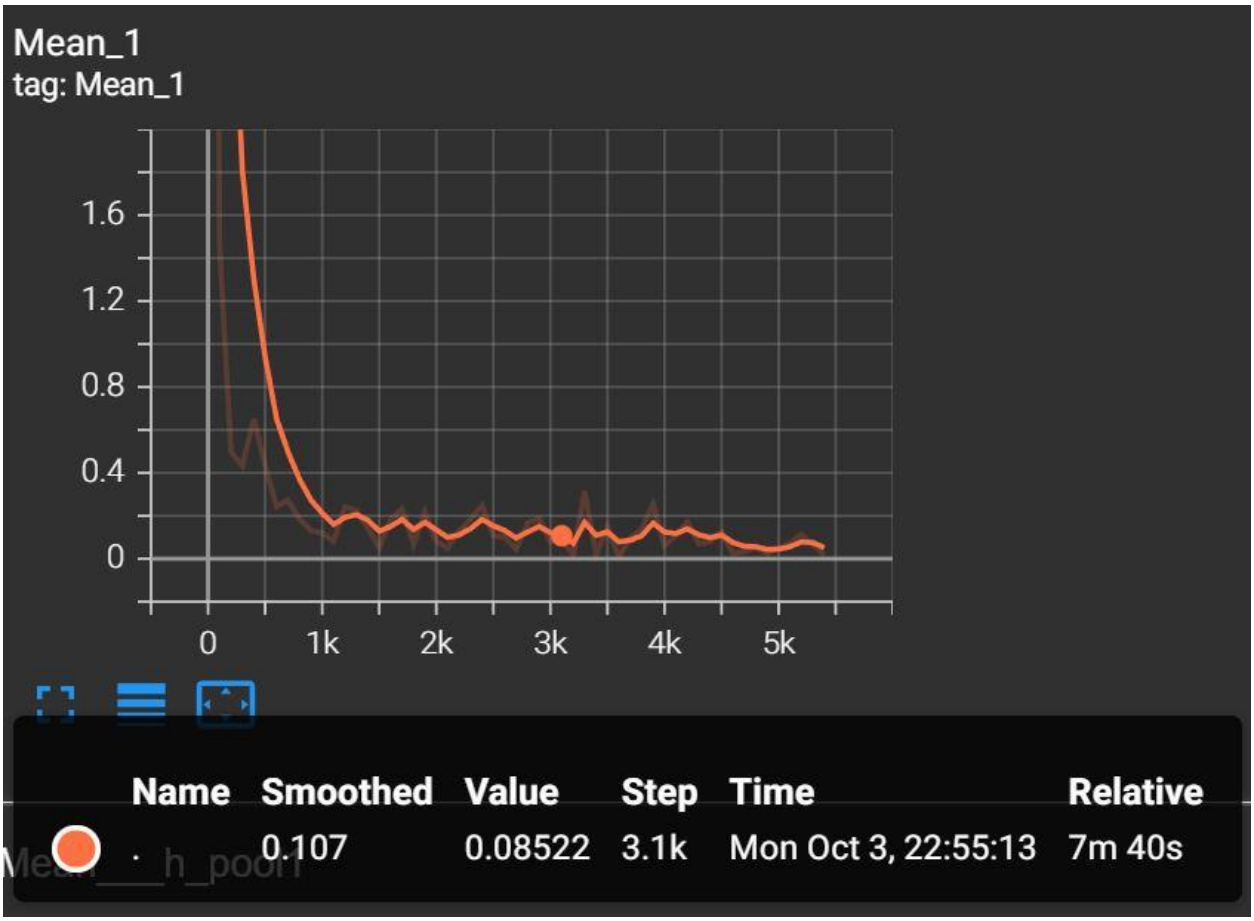

Max__h_pool2_flat
tag: Max__h_pool2_flat
2.1
2
1.9
1.8
1.7
0
1k
2k
3k
4k
5k

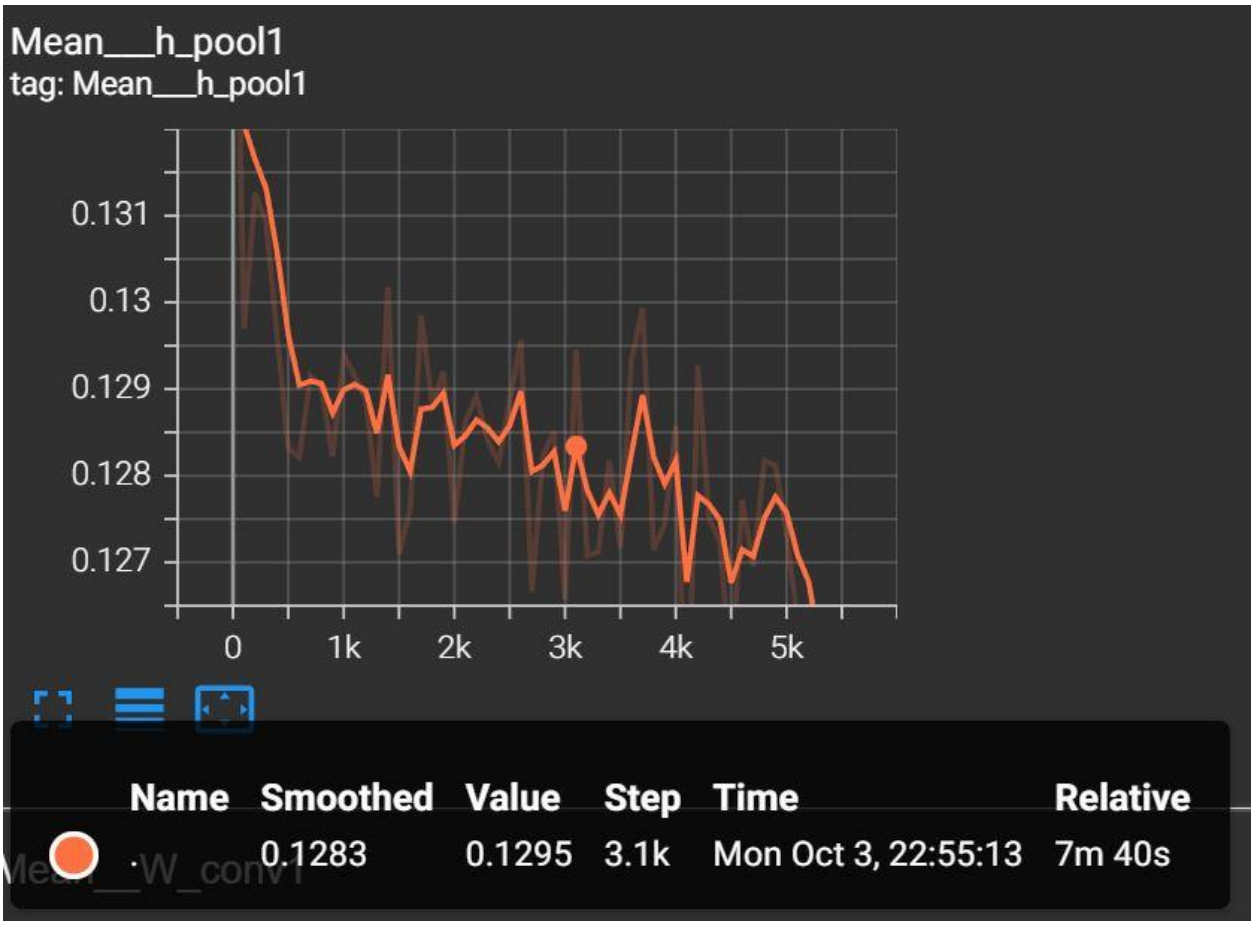

Max__input_1
tag: Max__input_1
1.51
1.49
1.47
1.45
1.43
0
1k
2k
3k
4k
5k
Name Smoothed Value Step Time Relative
1.49 1.474 2.5k Mon Oct 3, 22:53:49 6m 17s

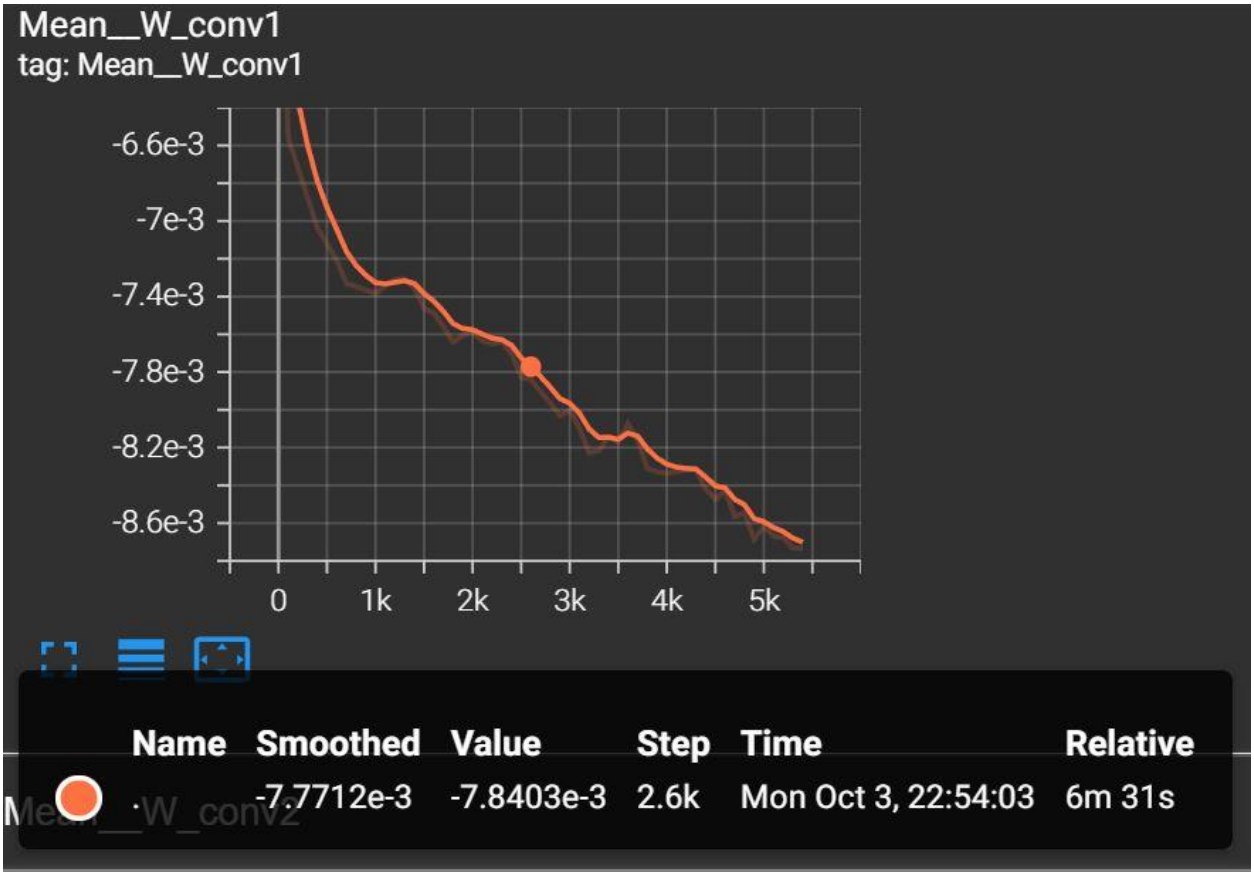

Max__input_2
tag: Max__input_2
2.1
2
1.9
1.8
1.7
0
1k
2k
3k
4k
5k

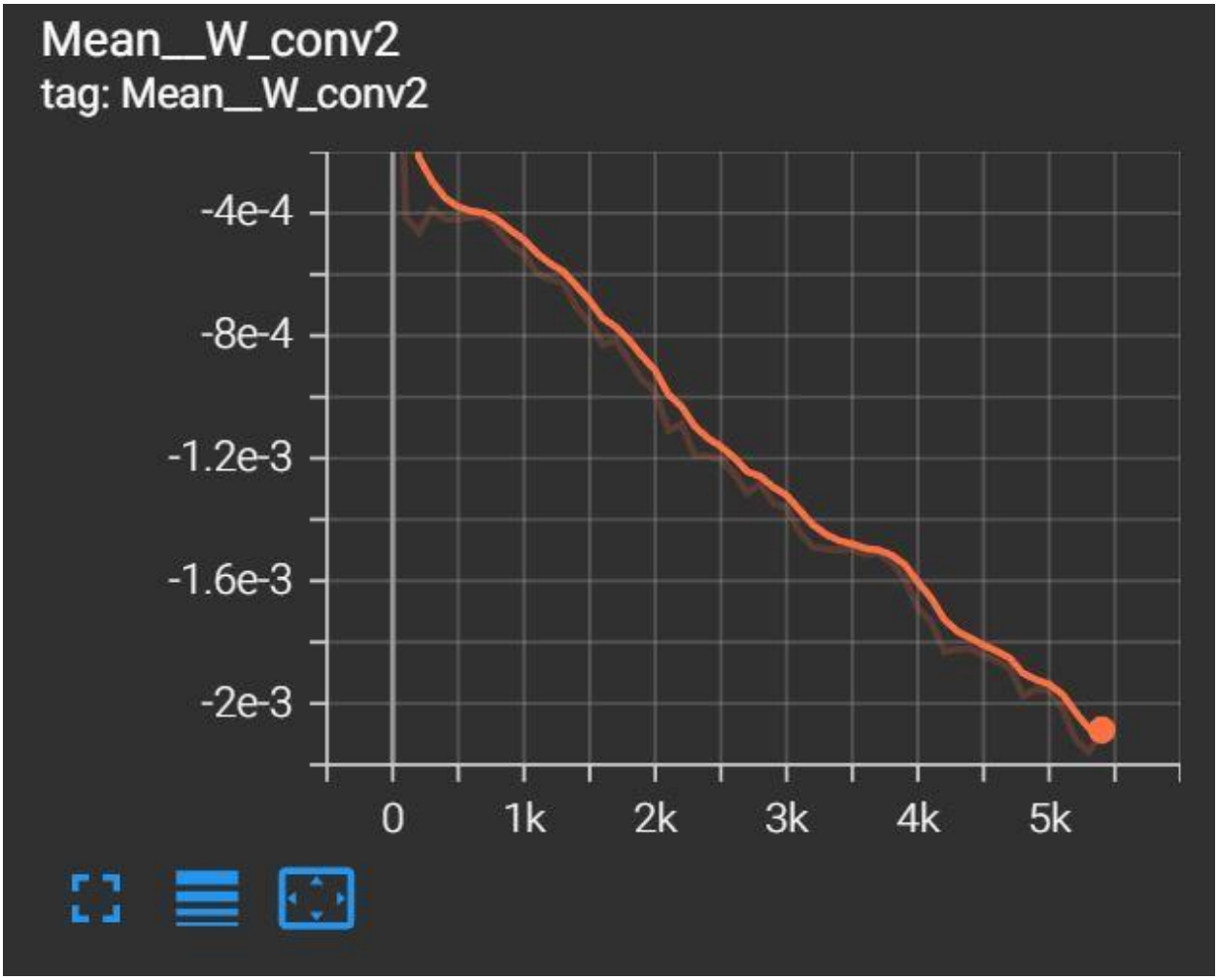

Max__input_fc1
tag: Max__input_fc1
5.3
5.1
4.9
4.7
4.5
4.3
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
4.768
4.917
3.8k
Mon Oct 3, 22:57:04
9m 32s

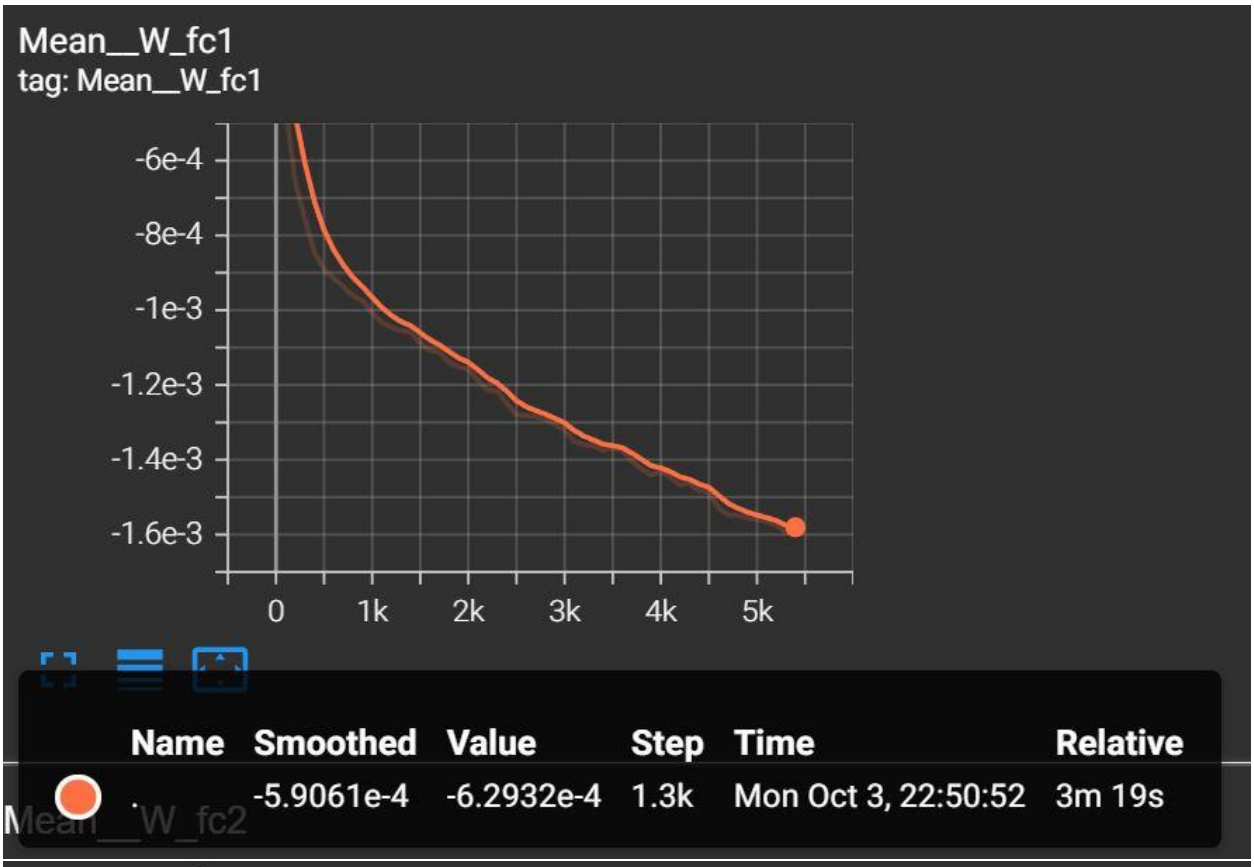

Max__y_conv_
tag: Max__y_conv_
1
1
1
1
1
1
0
500
1k
1.5k
2k
2.5k
3k
3.5k
4k
4.5k
5k
5.5k
Name
Smoothed
Value
Step
Time
Relative
1
1
2.2k
Mon Oct 3, 22:52:57
5m 25s

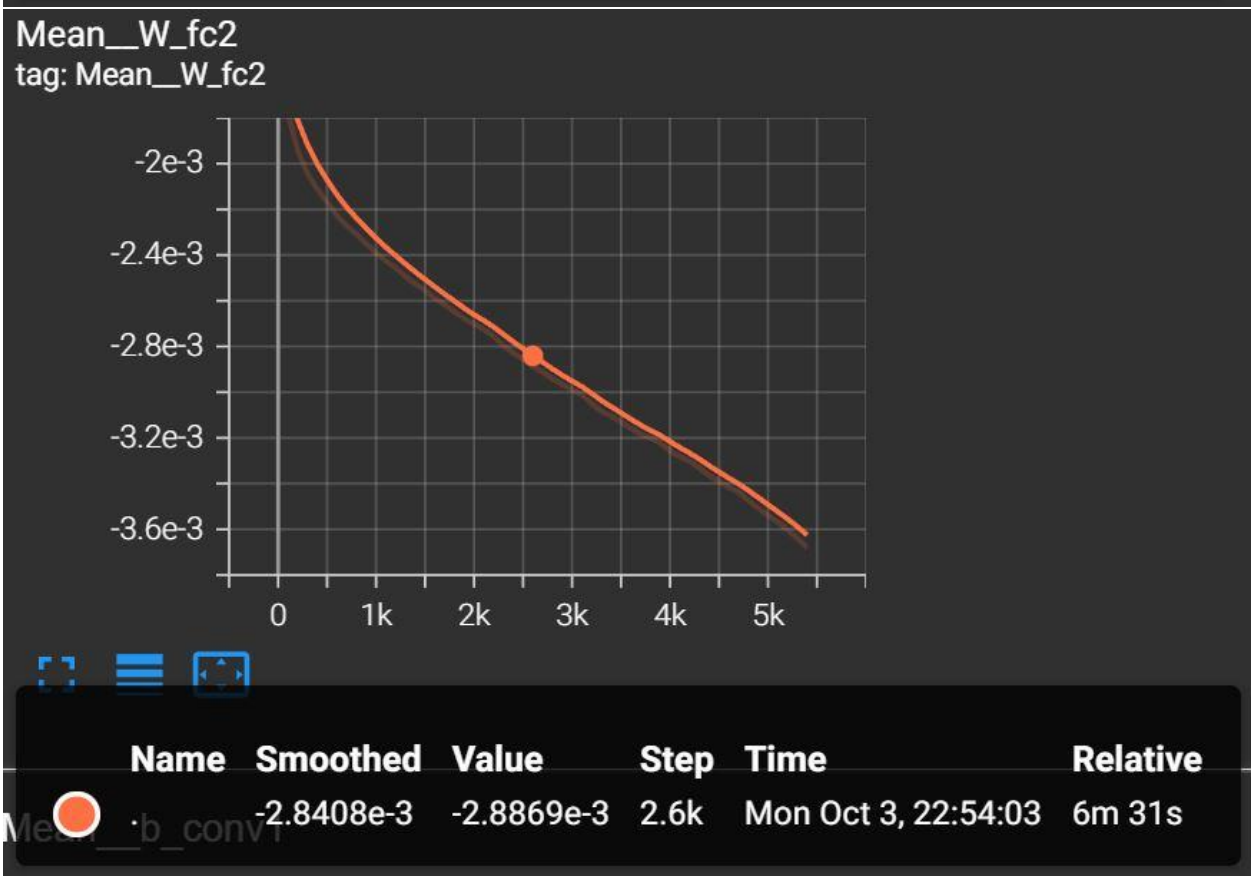

Mean_1
tag: Mean_1
1.6
1.2
0.8
0.4
0
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
0.107
0.08522
3.1k
Mon Oct 3, 22:55:13
7m 40s

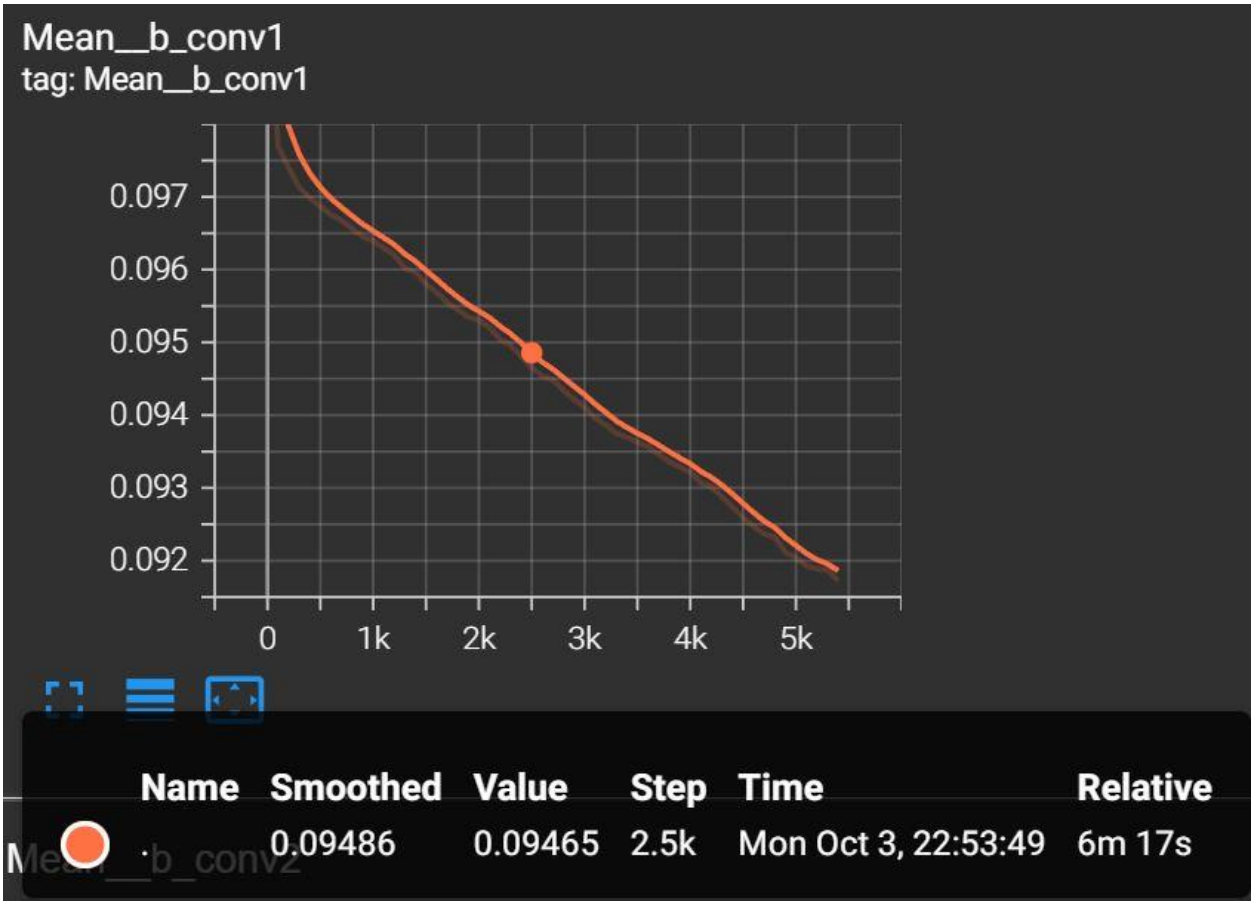

Mean___h_pool1
tag: Mean___h_pool1
0.131
0.13
0.129
0.128
0.127
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
0.1283
0.1295
3.1k
Mon Oct 3, 22:55:13
7m 40s

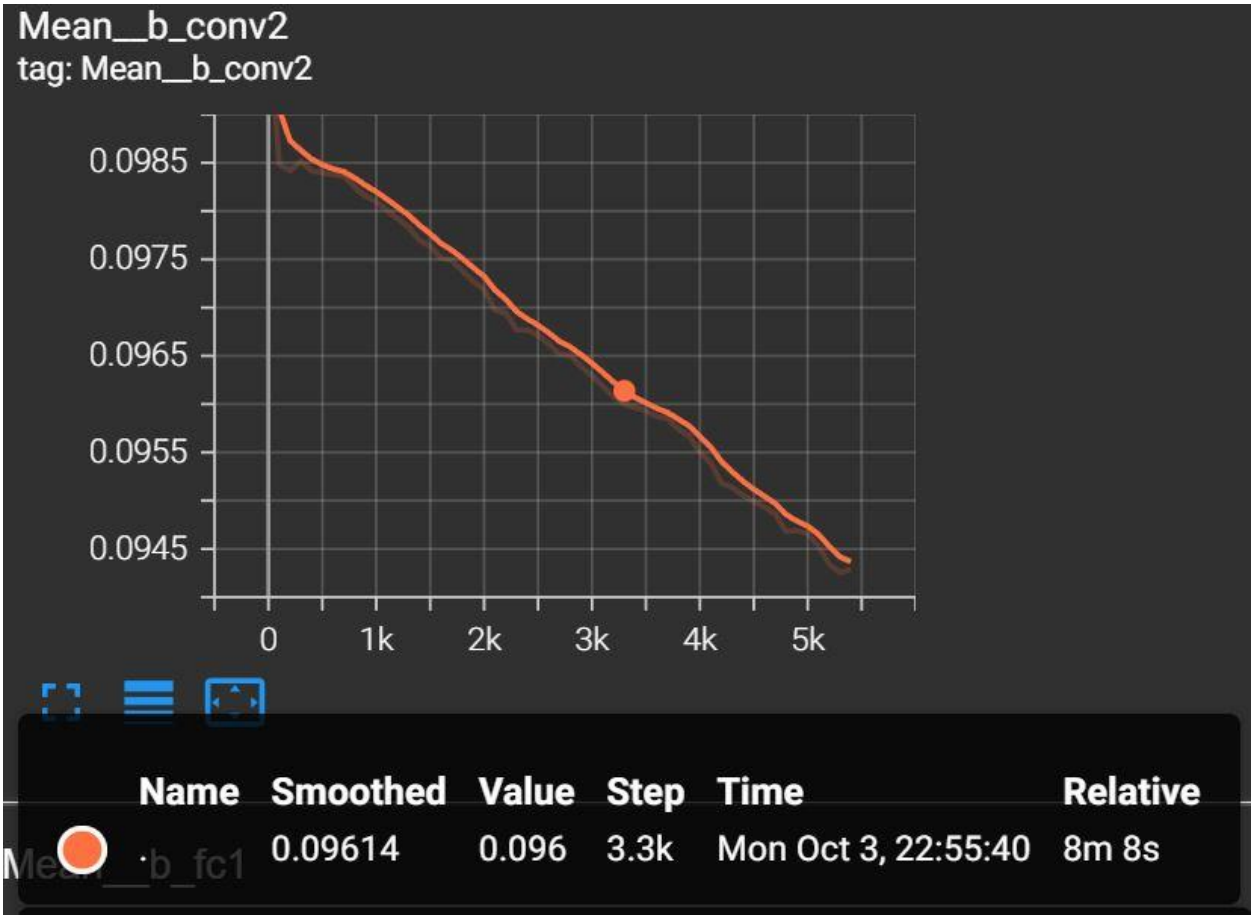

Mean__W_conv1
tag: Mean__W_conv1
-6.6e-3
-7e-3
-7.4e-3
-7.8e-3
-8.2e-3
-8.6e-3
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
-7.7712e-3
-7.8403e-3
2.6k
Mon Oct 3, 22:54:03
6m 31s

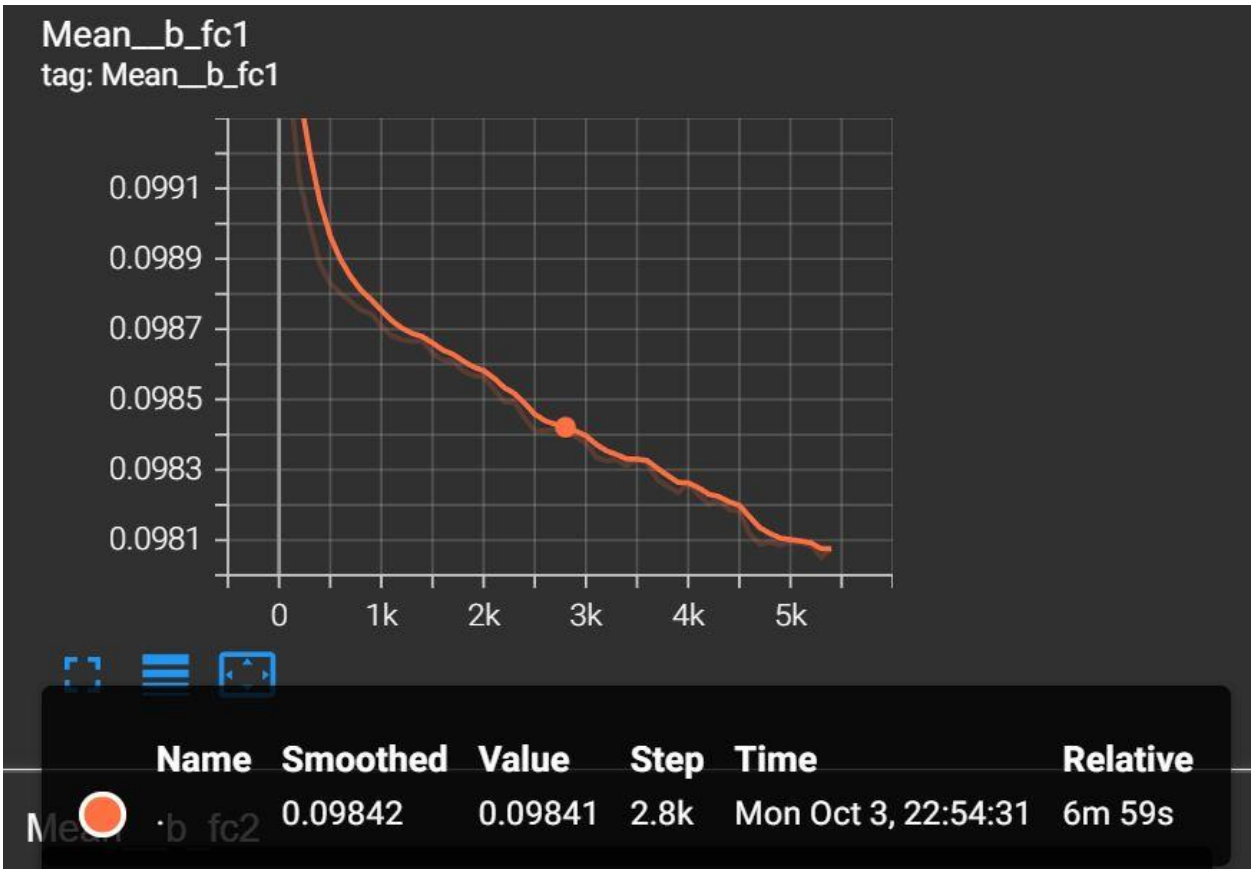

Mean__W_conv2
tag: Mean__W_conv2
-4e-4
-8e-4
-1.2e-3
-1.6e-3
-2e-3
0
1k
2k
3k
4k
5k

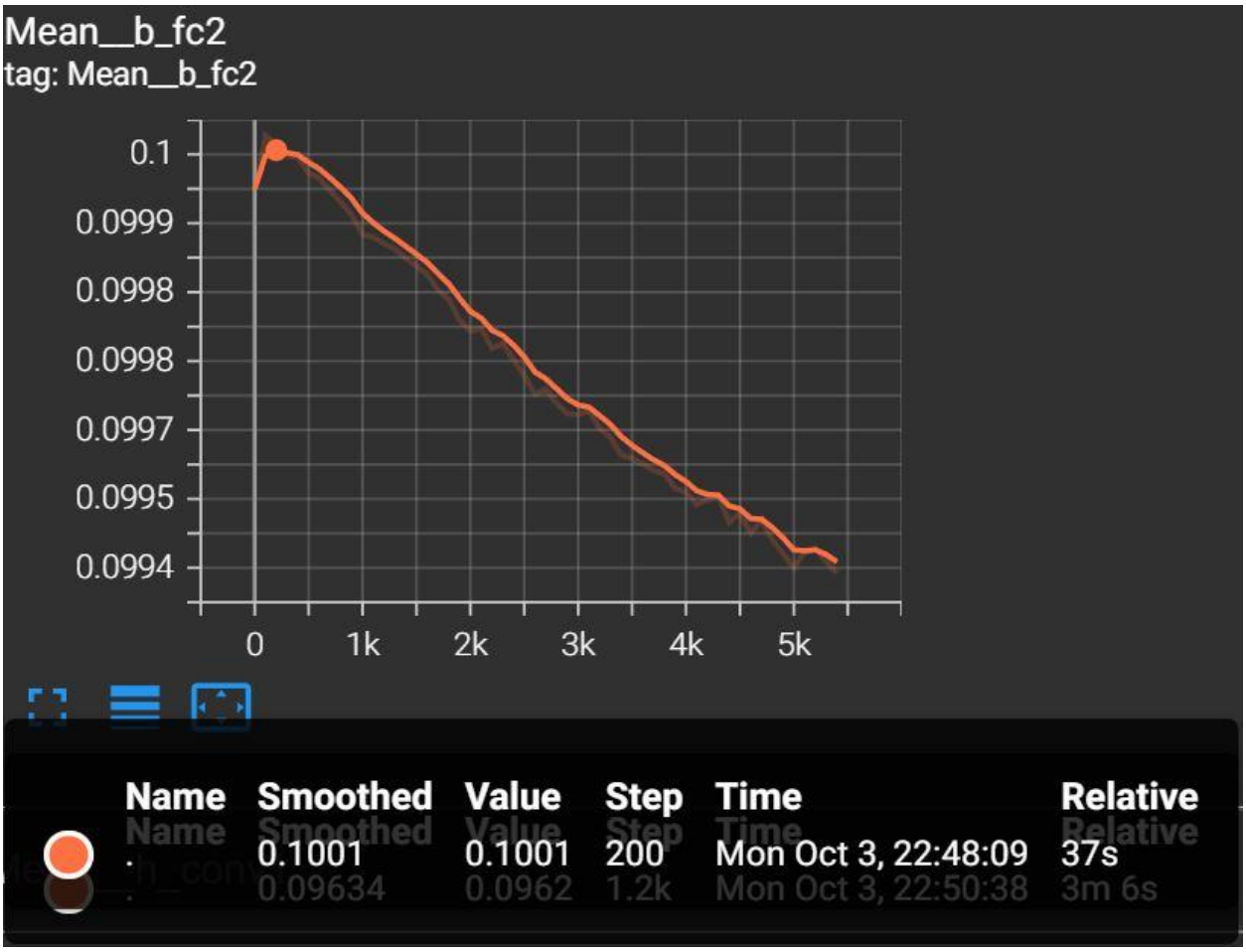

Mean__W_fc1
tag: Mean__W_fc1
-6e-4
-8e-4
-1e-3
-1.2e-3
-1.4e-3
-1.6e-3
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
-5.9061e-4
-6.2932e-4
1.3k
Mon Oct 3, 22:50:52
3m 19s

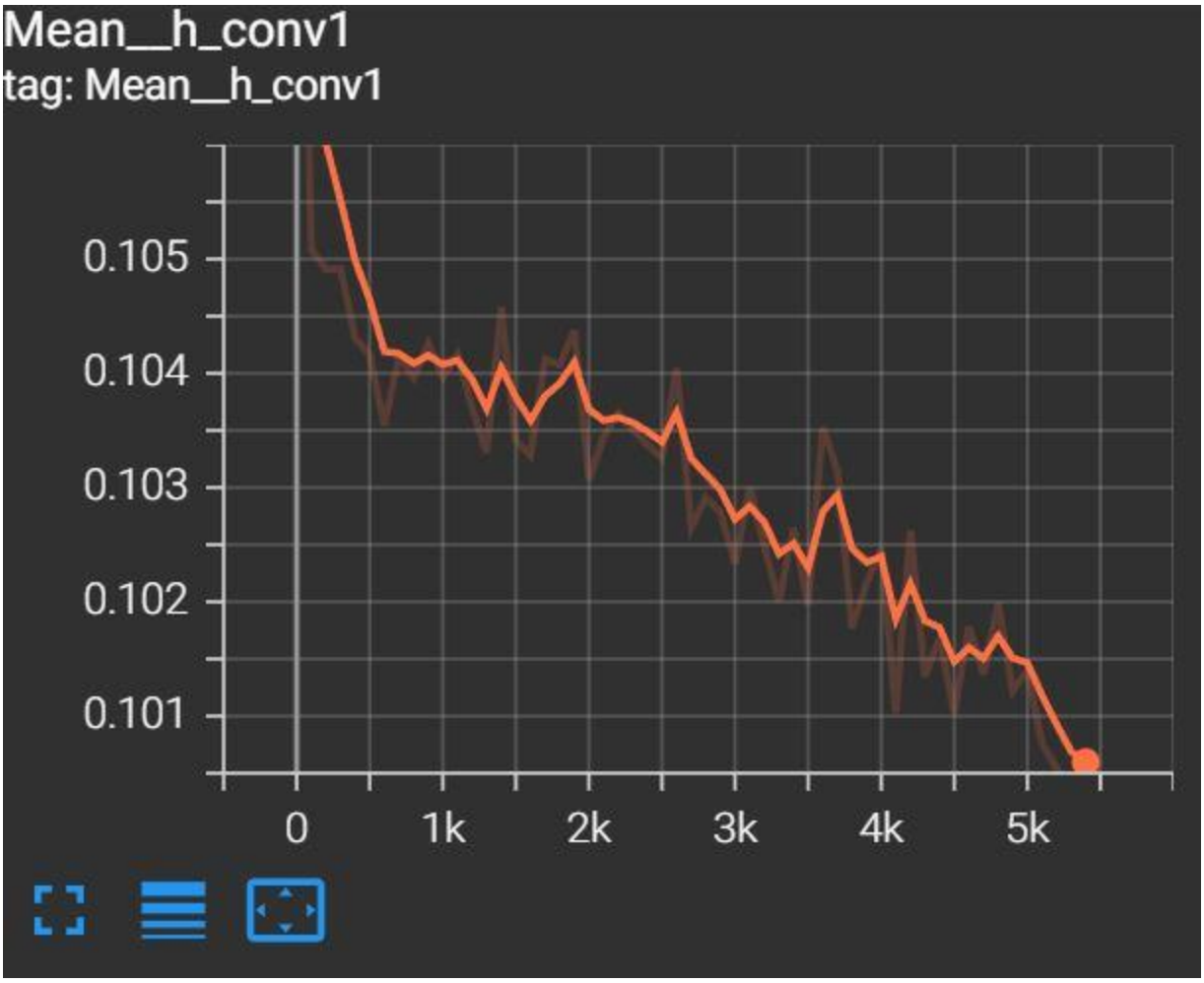

Mean__W_fc2
tag: Mean__W_fc2
-2e-3
-2.4e-3
-2.8e-3
-3.2e-3
-3.6e-3
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
-2.8408e-3
-2.8869e-3
2.6k
Mon Oct 3, 22:54:03
6m 31s

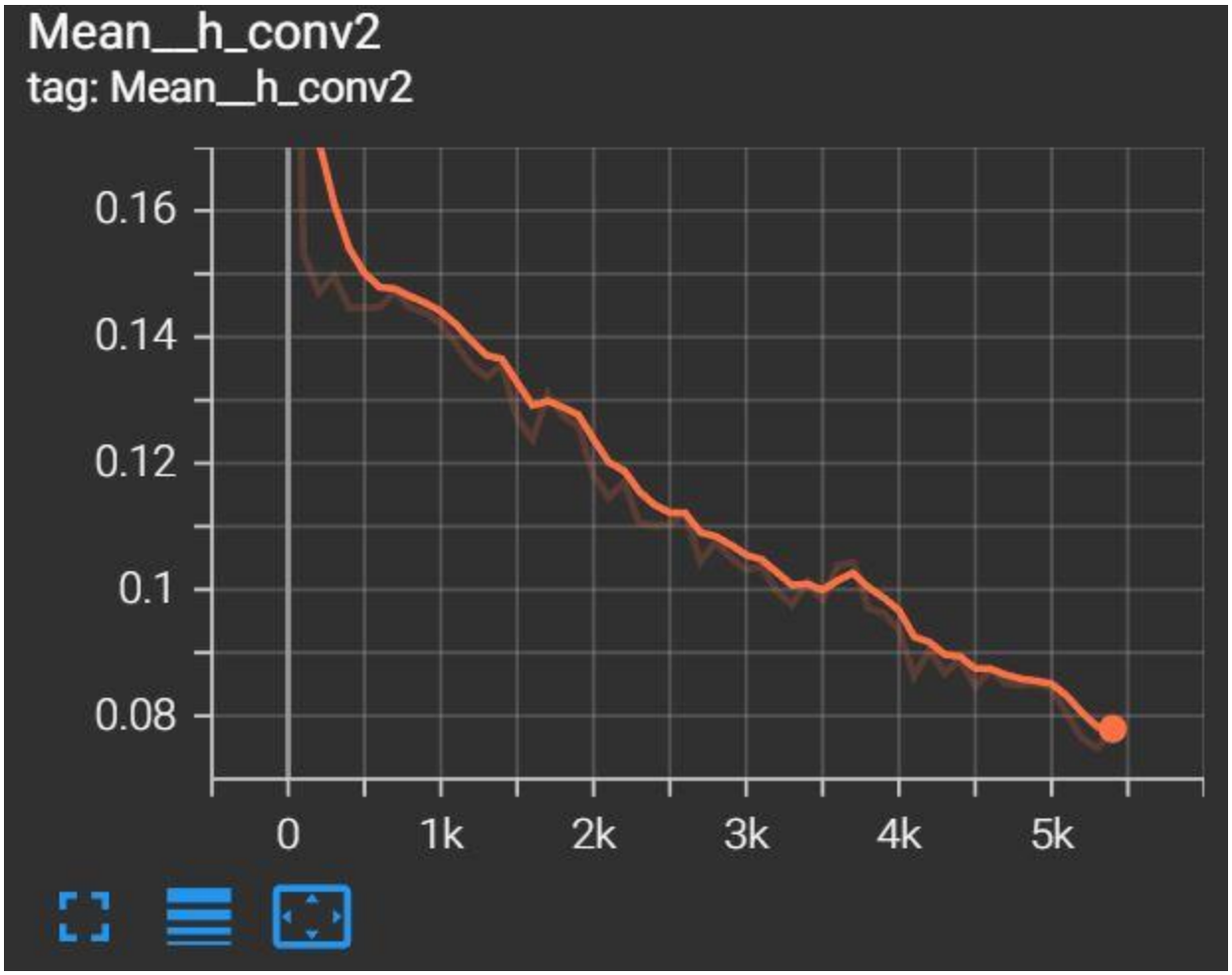

Mean__b_conv1
tag: Mean__b_conv1
0.097
0.096
0.095
0.094
0.093
0.092
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
0.09486
0.09465
2.5k
Mon Oct 3, 22:53:49
6m 17s

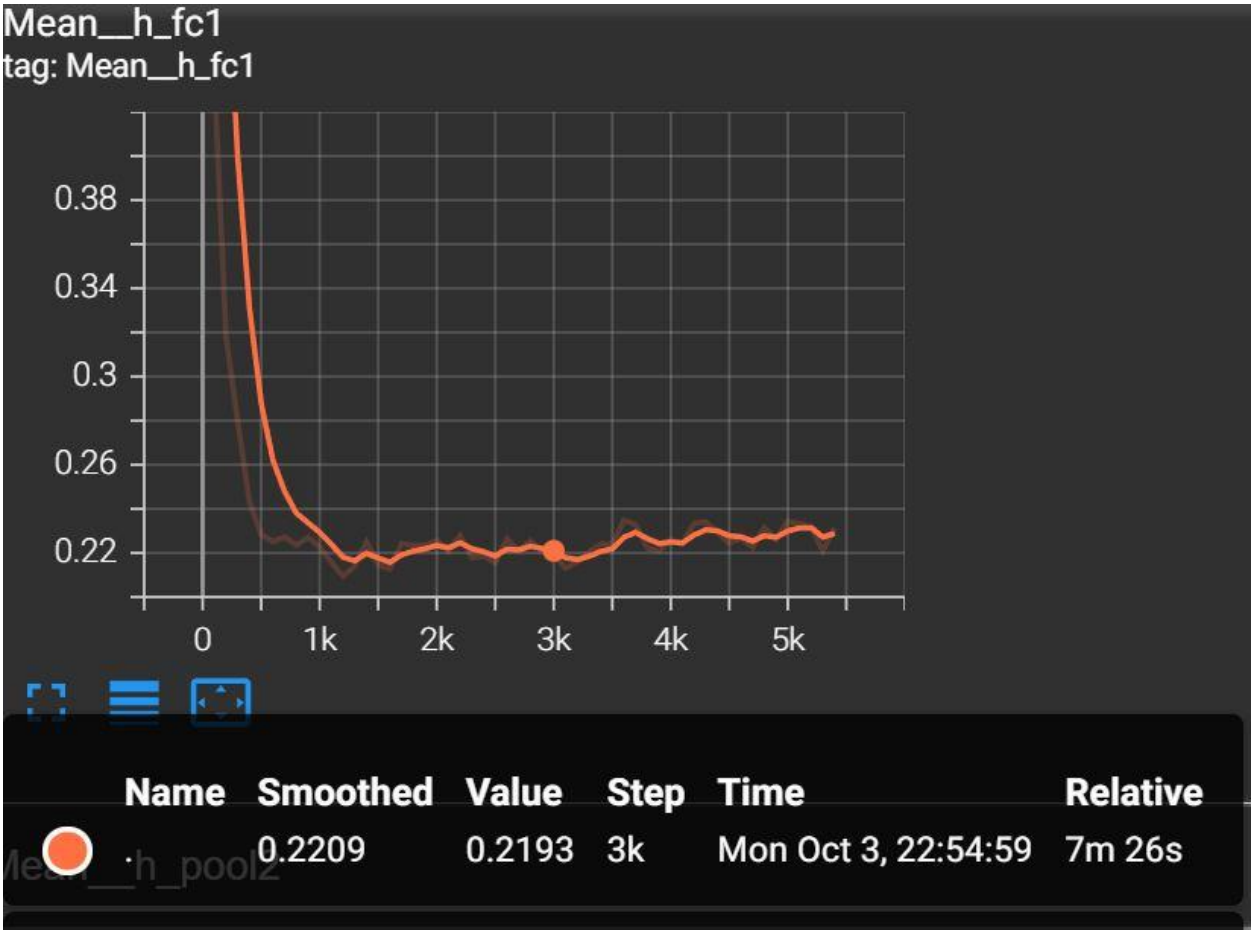

Mean__b_conv2
tag: Mean__b_conv2
0.0985
0.0975
0.0965
0.0955
0.0945
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
·b_fc1
0.09614
0.096
3.3k
Mon Oct 3, 22:55:40
8m 8s

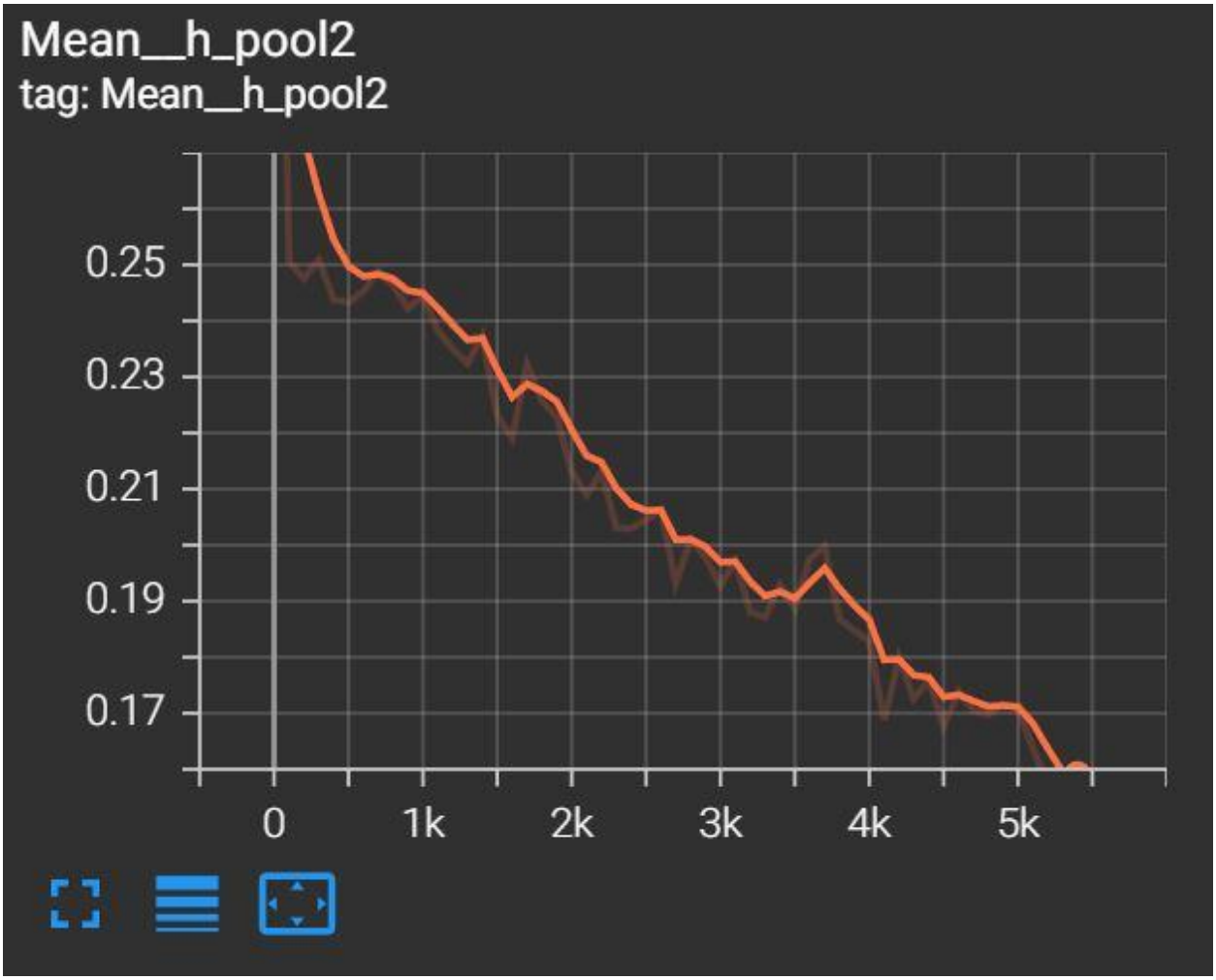

Mean__b_fc1
tag: Mean__b_fc1
0.0991
0.0989
0.0987
0.0985
0.0983
0.0981
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
0.09842
0.09841
2.8k
Mon Oct 3, 22:54:31
6m 59s

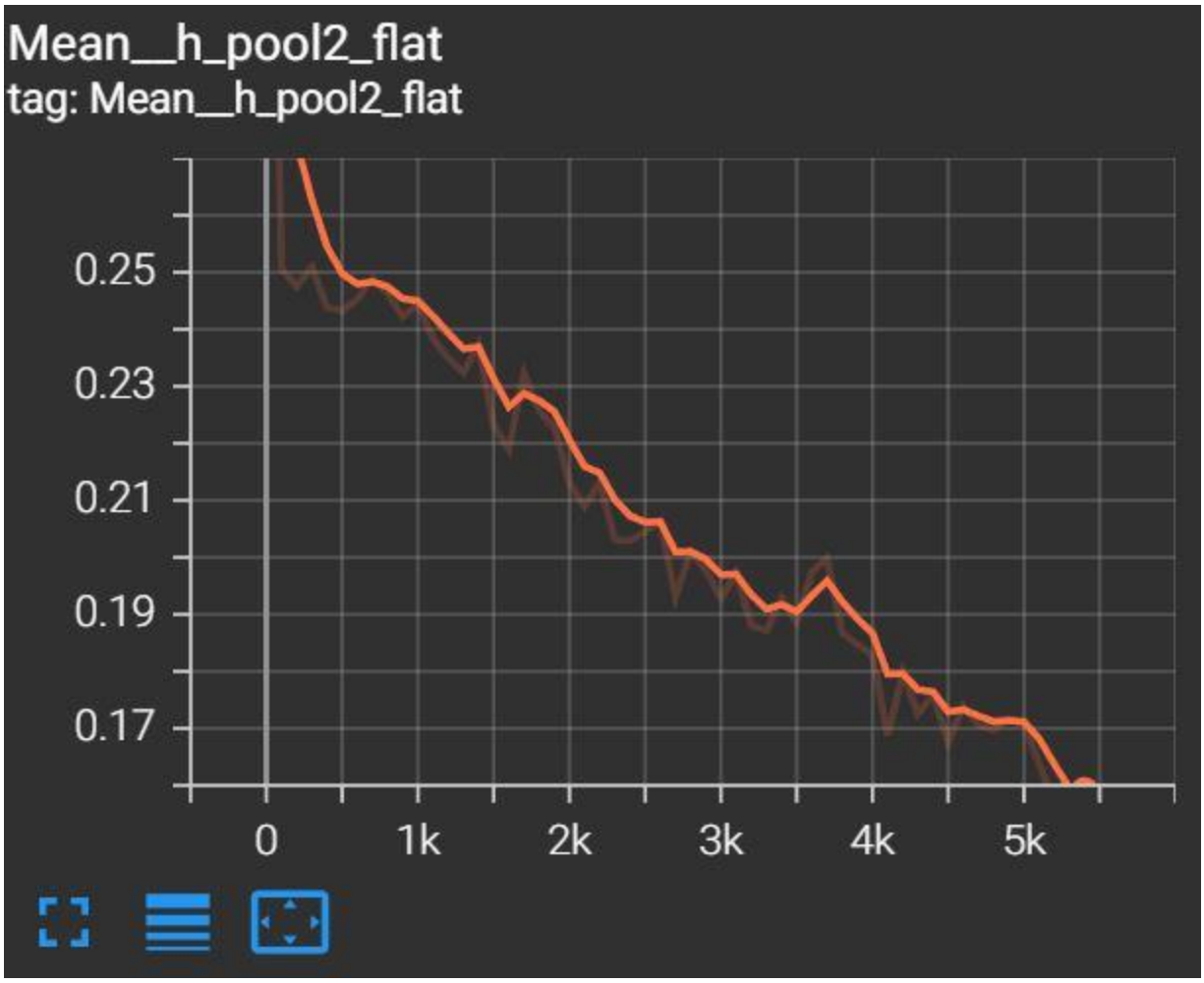

Mean__b_fc2
tag: Mean__b_fc2
0.1
0.0999
0.0998
0.0998
0.0997
0.0995
0.0994
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
0.1001
0.1001
200
Mon Oct 3, 22:48:09
37s

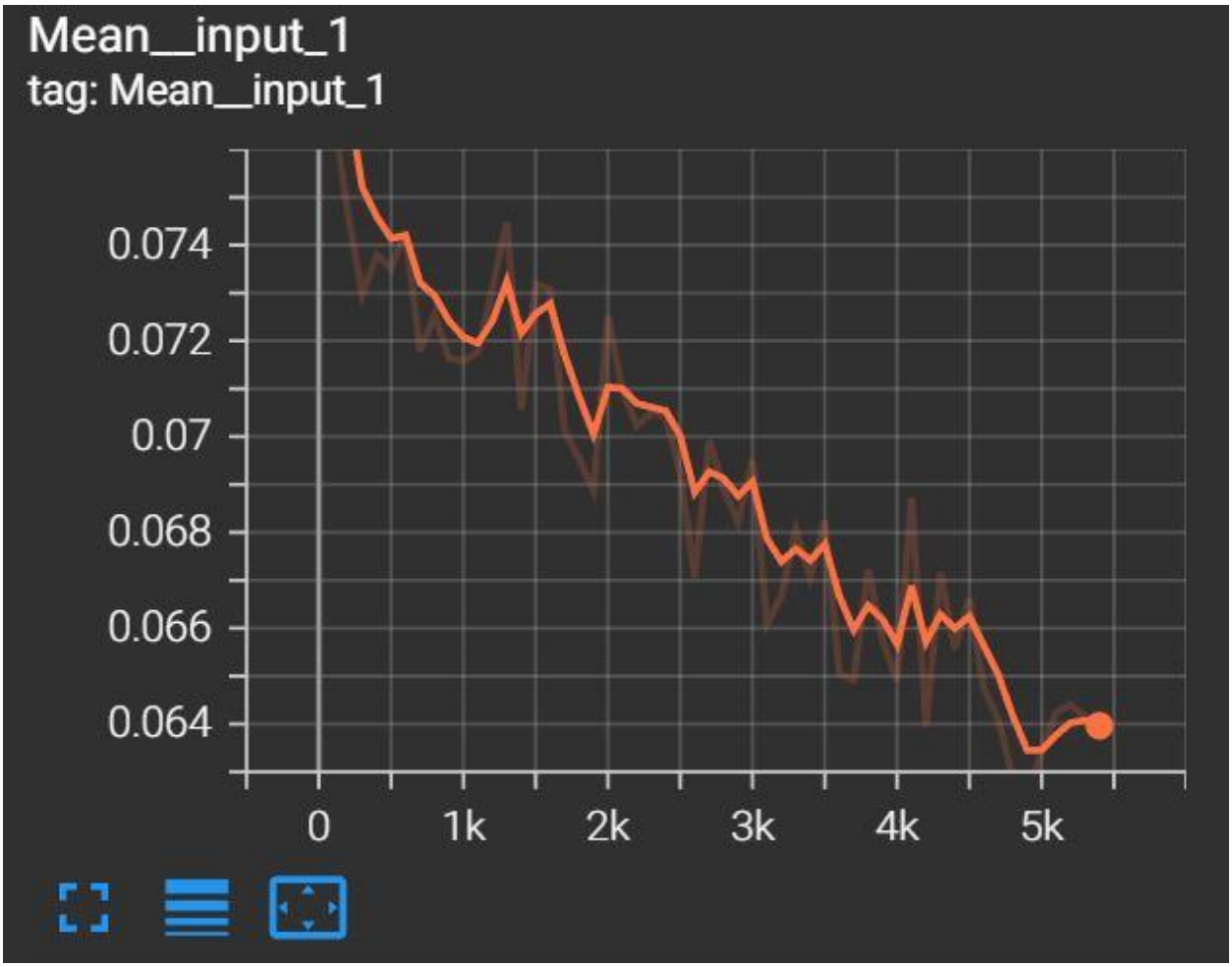

Mean__h_conv1
tag: Mean__h_conv1
0.105
0.104
0.103
0.102
0.101
0
1k
2k
3k
4k
5k

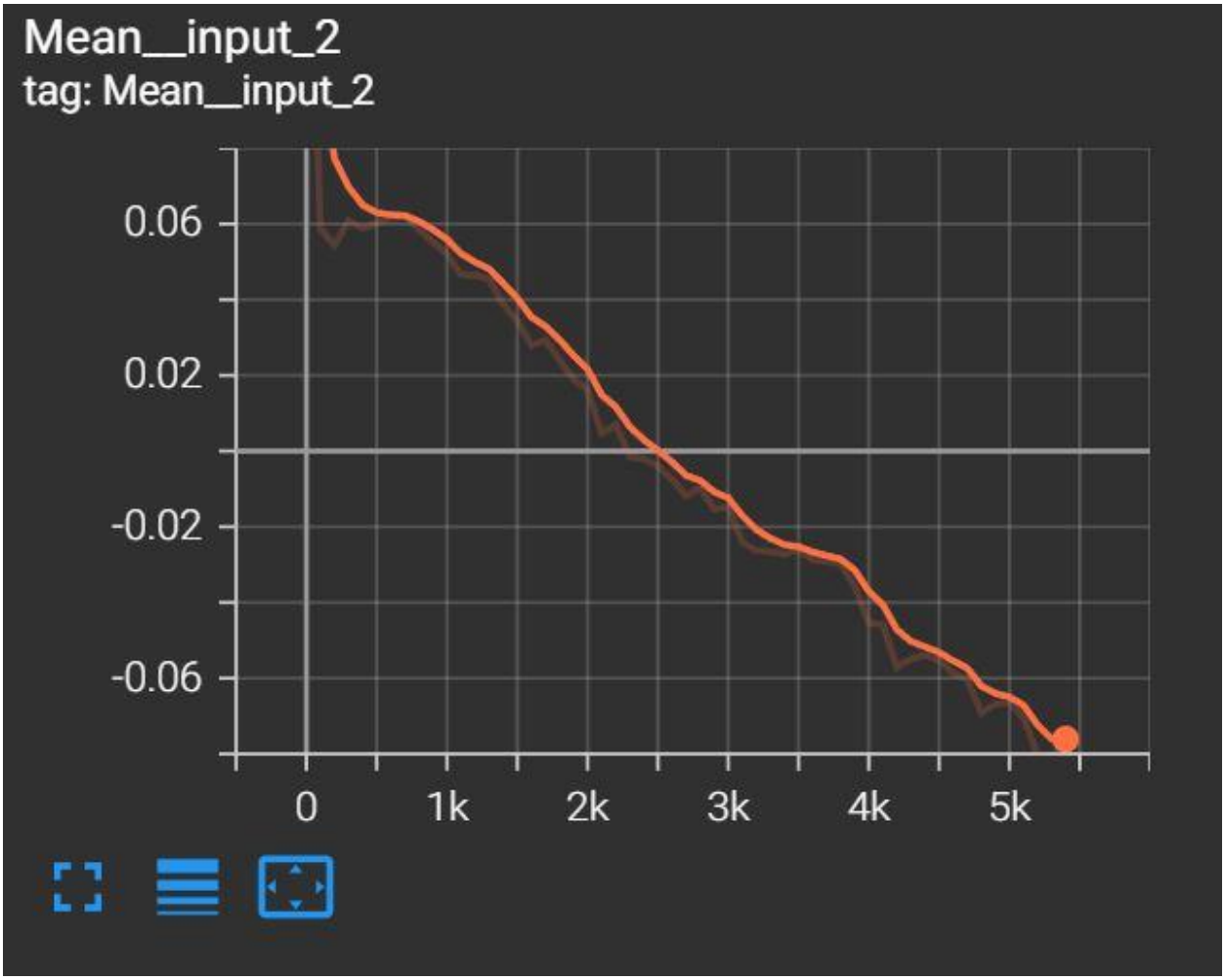

Mean__h_conv2
tag: Mean__h_conv2
0.16
0.14
0.12
0.1
0.08
0
1k
2k
3k
4k
5k

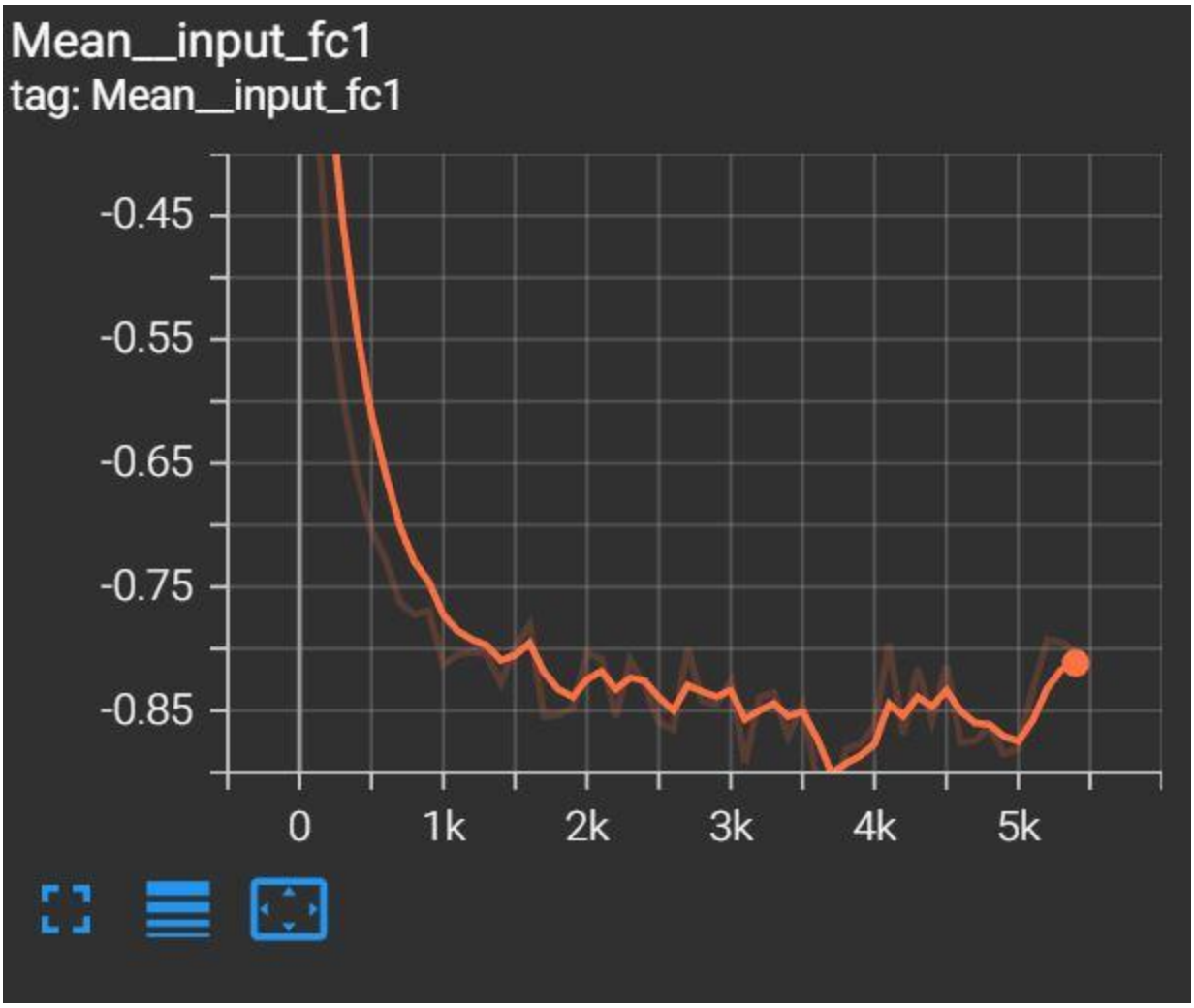

Mean__h_fc1
tag: Mean__h_fc1
0.38
0.34
0.3
0.26
0.22
0
1k
2k
3k
4k
5k
Name
Smoothed
Value
Step
Time
Relative
0.2209
0.2193
3k
Mon Oct 3, 22:54:59
7m 26s

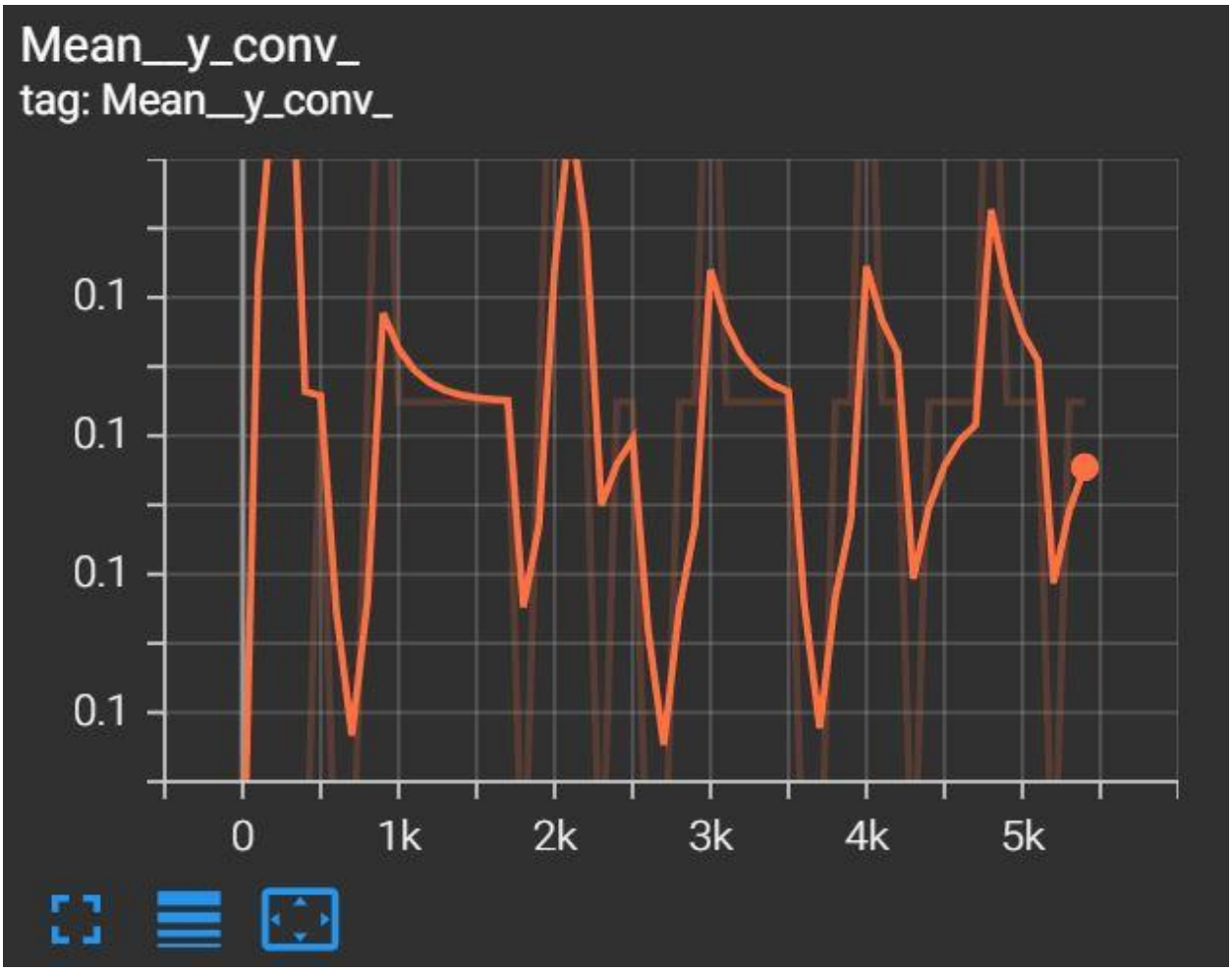

Mean__h_pool2
tag: Mean__h_pool2
0.25
0.23
0.21
0.19
0.17
0
1k
2k
3k
4k
5k

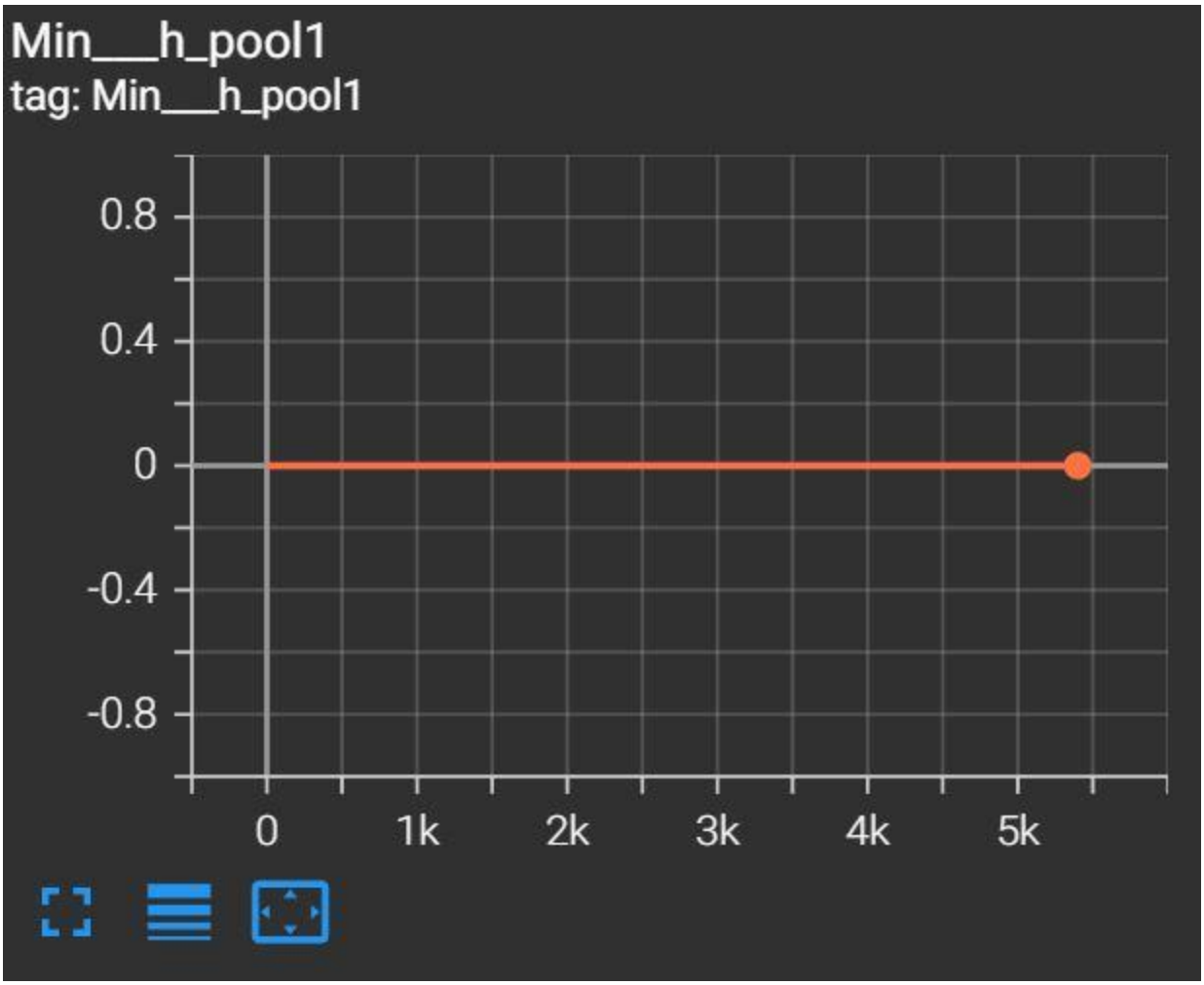

Mean__h_pool2_flat
tag: Mean__h_pool2_flat
0.25
0.23
0.21
0.19
0.17
0
1k
2k
3k
4k
5k

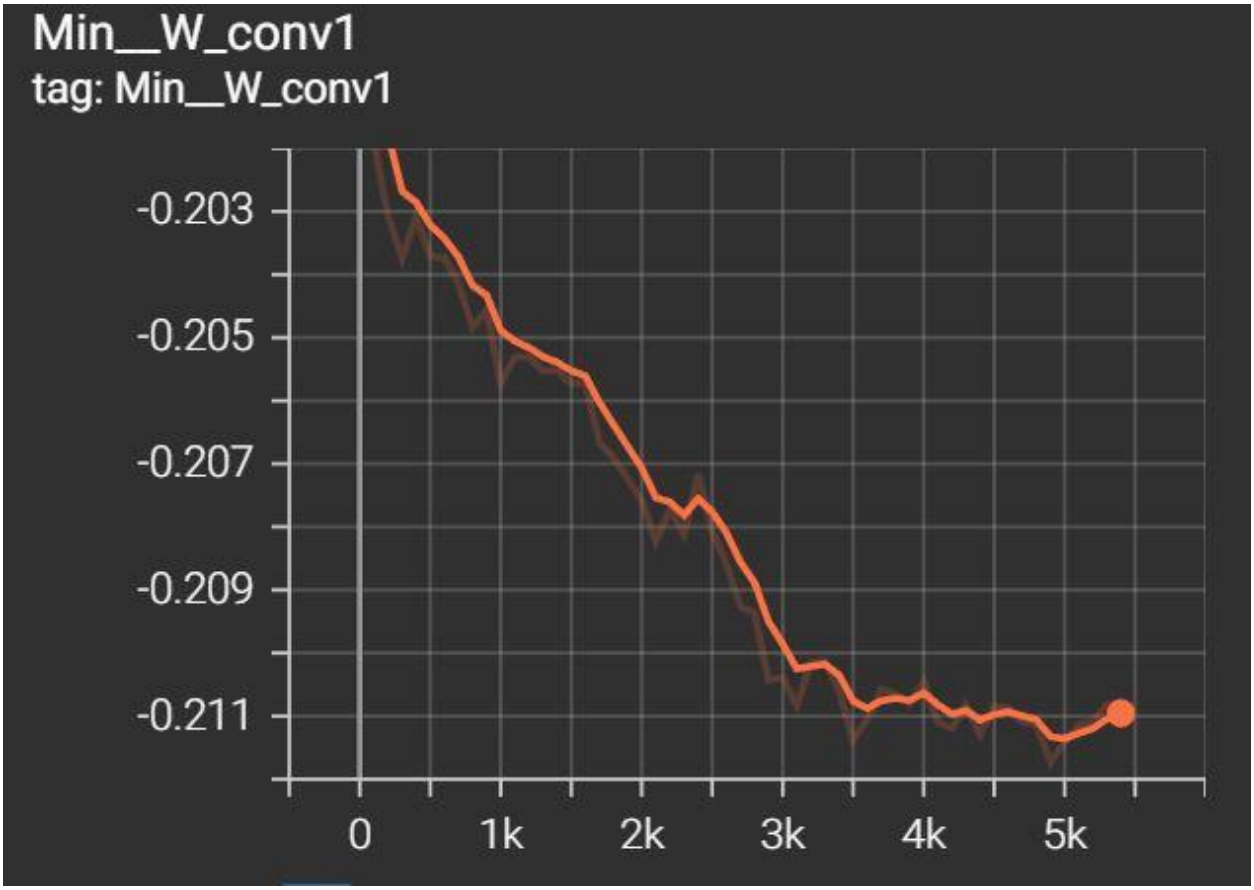

Mean__input_1
tag: Mean__input_1
0.074
0.072
0.07
0.068
0.066
0.064
0
1k
2k
3k
4k
5k


Mean__input_2
tag: Mean__input_2
0.06
0.02
-0.02
-0.06
0
1k
2k
3k
4k
5k

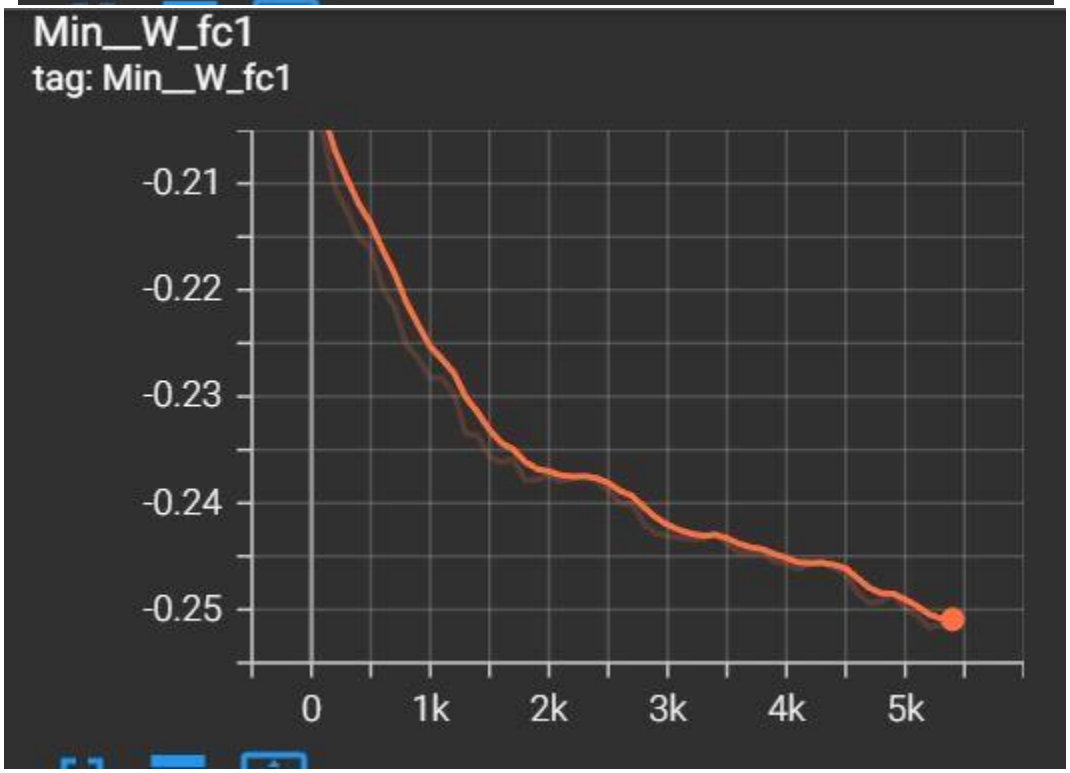

Mean__input_fc1
tag: Mean__input_fc1
-0.45
-0.55
-0.65
-0.75
-0.85
0
1k
2k
3k
4k
5k

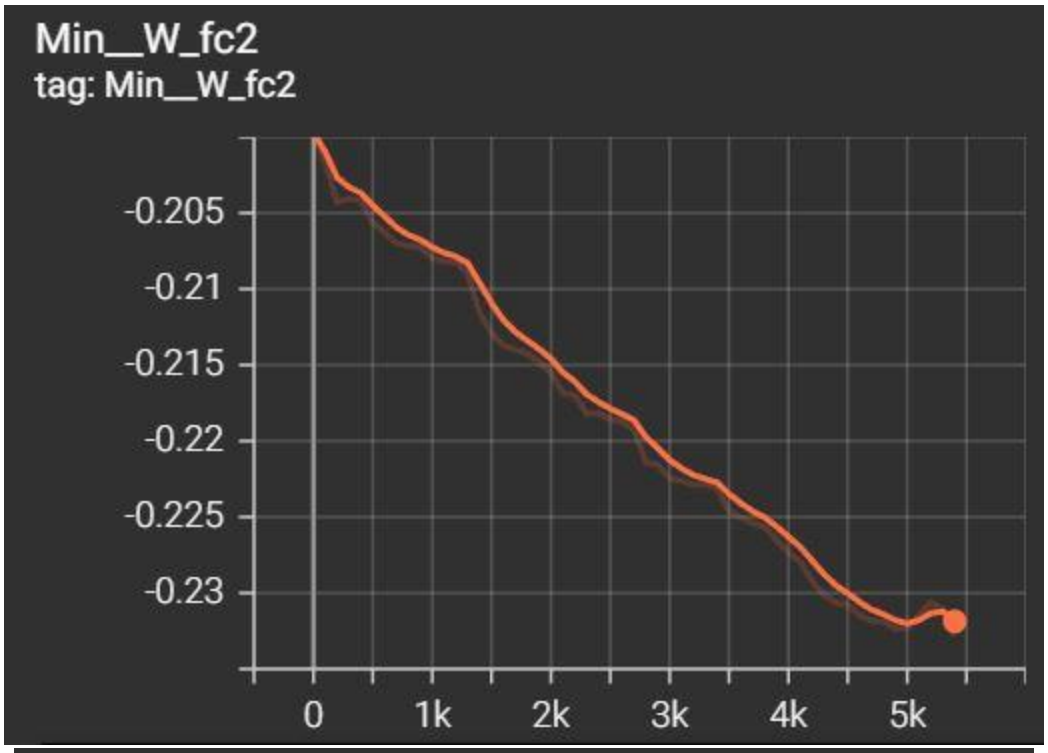

Mean__y_conv_
tag: Mean__y_conv_
0.1
0.1
0.1
0.1
0
1k
2k
3k
4k
5k

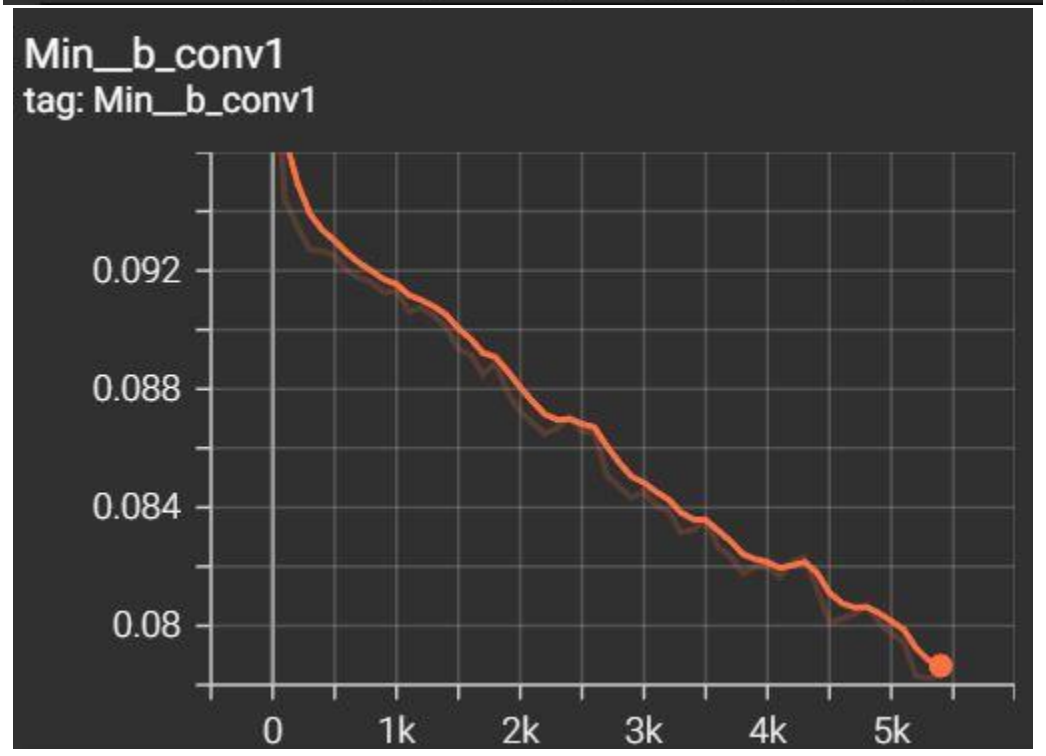

Min___h_pool1
tag: Min___h_pool1
0.8
0.4
0
-0.4
-0.8
0
1k
2k
3k
4k
5k

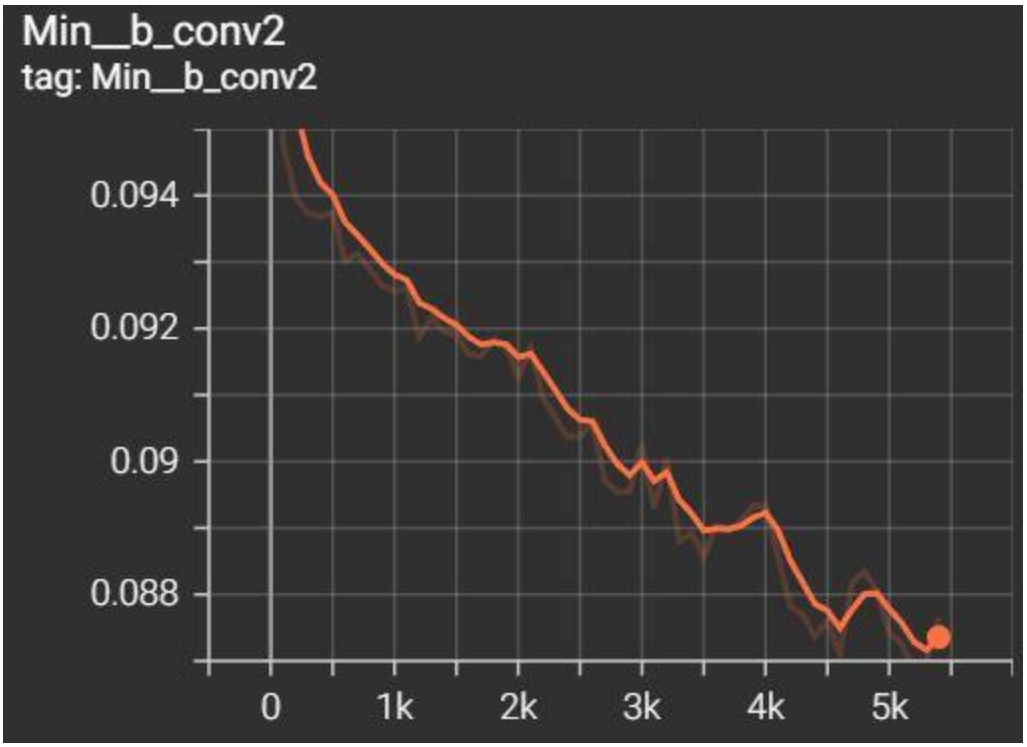

Min__W_conv1
tag: Min__W_conv1
-0.203
-0.205
-0.207
-0.209
-0.211
0
1k
2k
3k
4k
5k

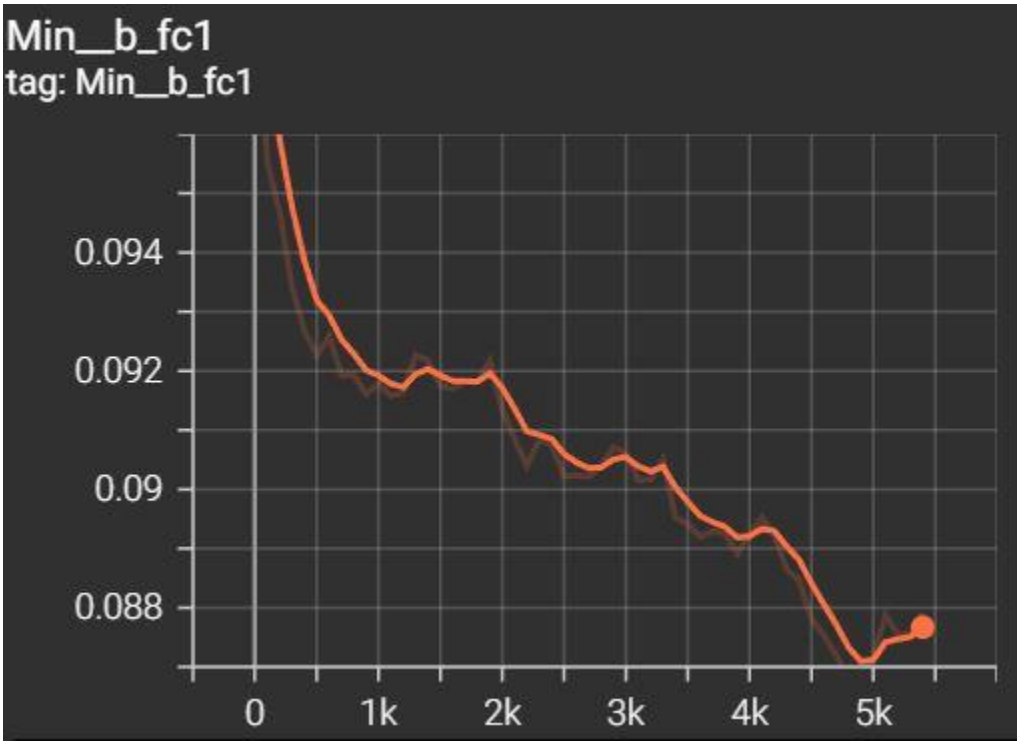

Min__W_conv2
tag: Min__W_conv2
-0.204
-0.208
-0.212
-0.216
-0.22
-0.224
0
1k
2k
3k
4k
5k

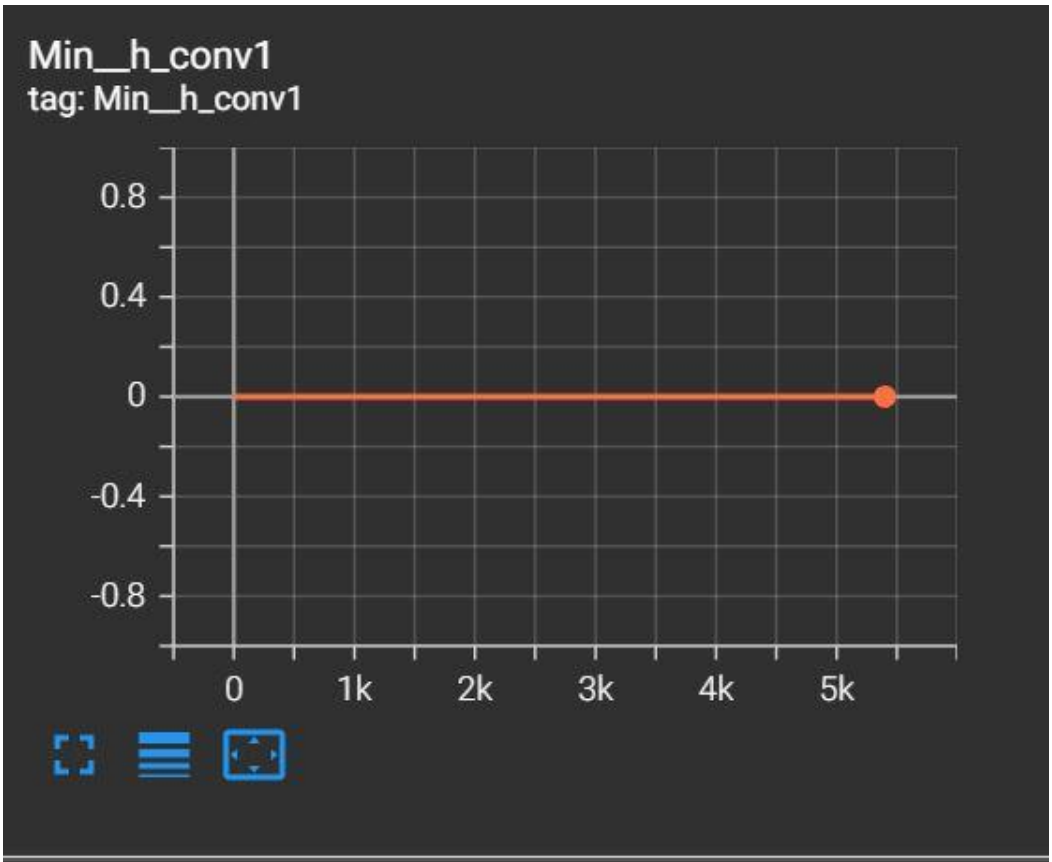

Min__W_fc1
tag: Min__W_fc1
-0.21
-0.22
-0.23
-0.24
-0.25
0
1k
2k
3k
4k
5k


Min__W_fc2
tag: Min__W_fc2
-0.205
-0.21
-0.215
-0.22
-0.225
-0.23
0
1k
2k
3k
4k
5k

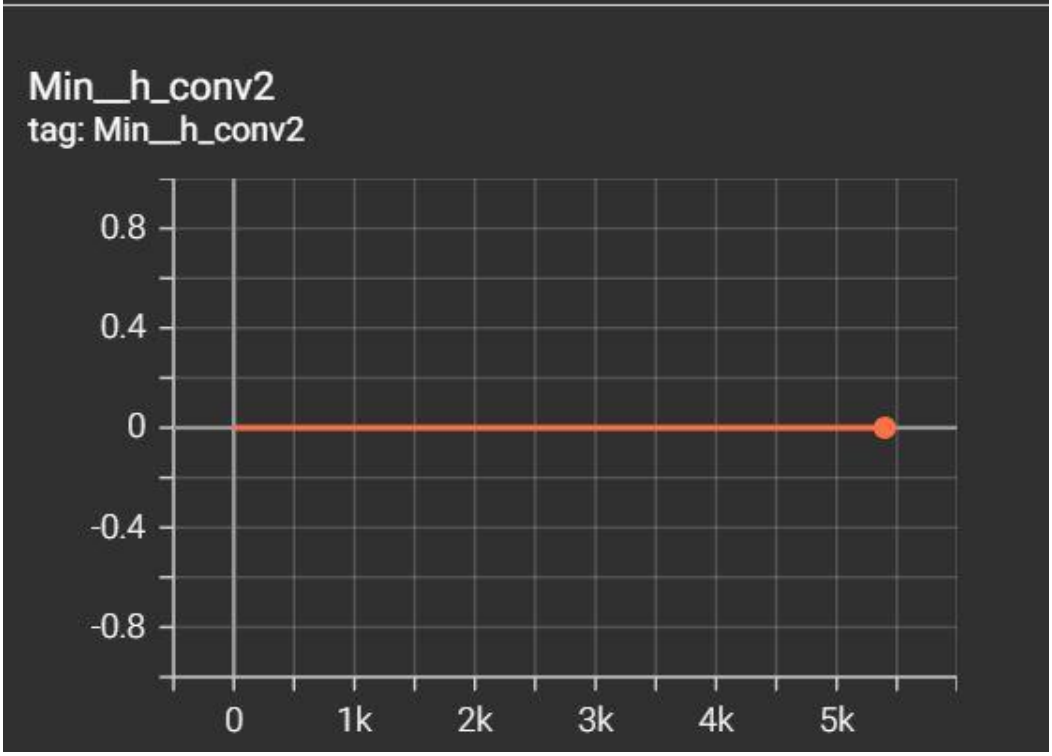

Min__b_conv1
tag: Min__b_conv1
0.092
0.088
0.084
0.08
0
1k
2k
3k
4k
5k

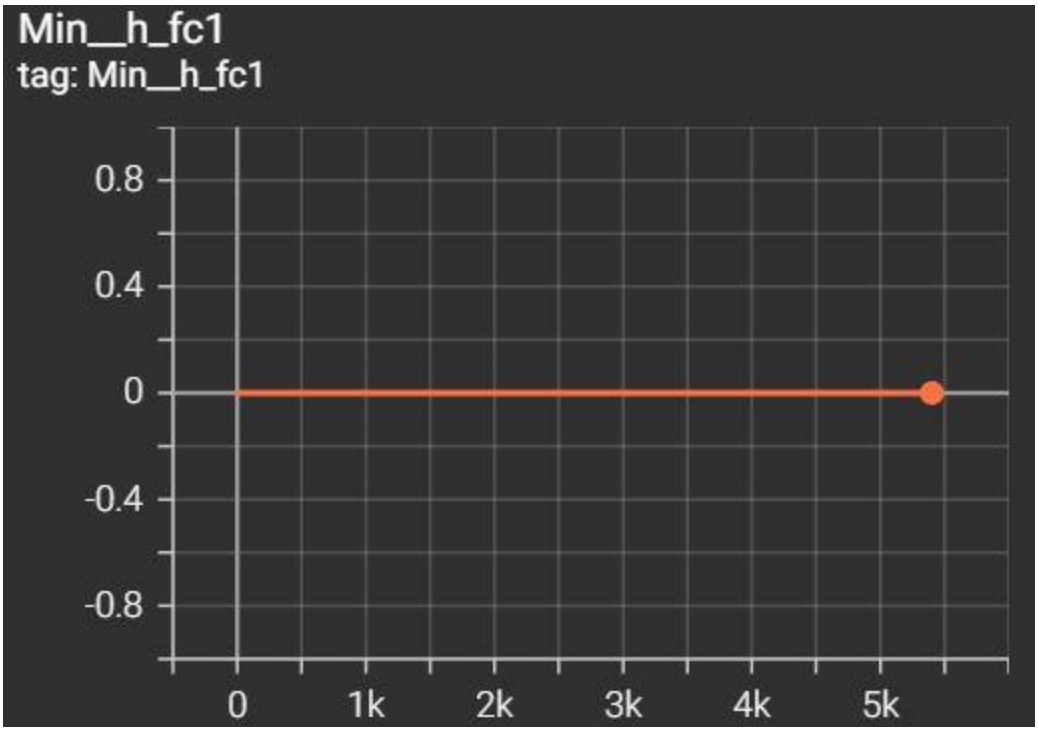

Min__b_conv2
tag: Min__b_conv2
0.094
0.092
0.09
0.088
0
1k
2k
3k
4k
5k

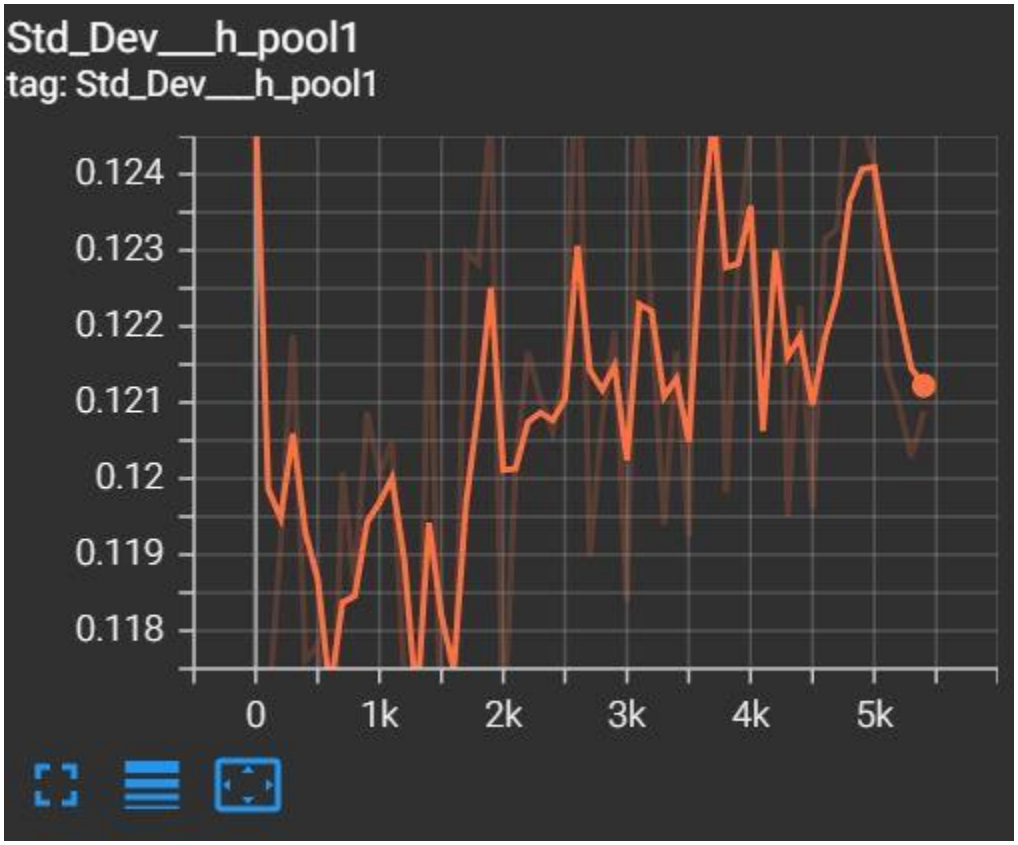

Min__b_fc1
tag: Min__b_fc1
0.094
0.092
0.09
0.088
0
1k
2k
3k
4k
5k

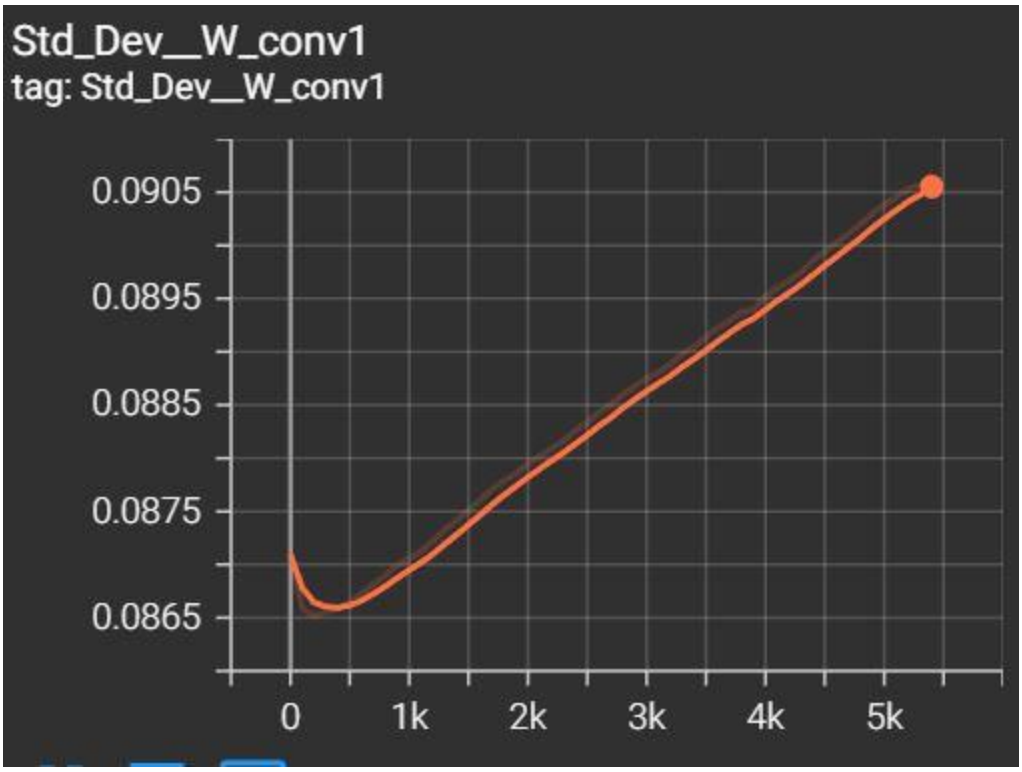

Min__h_conv1
tag: Min__h_conv1
0.8
0.4
0
-0.4
-0.8
0
1k
2k
3k
4k
5k

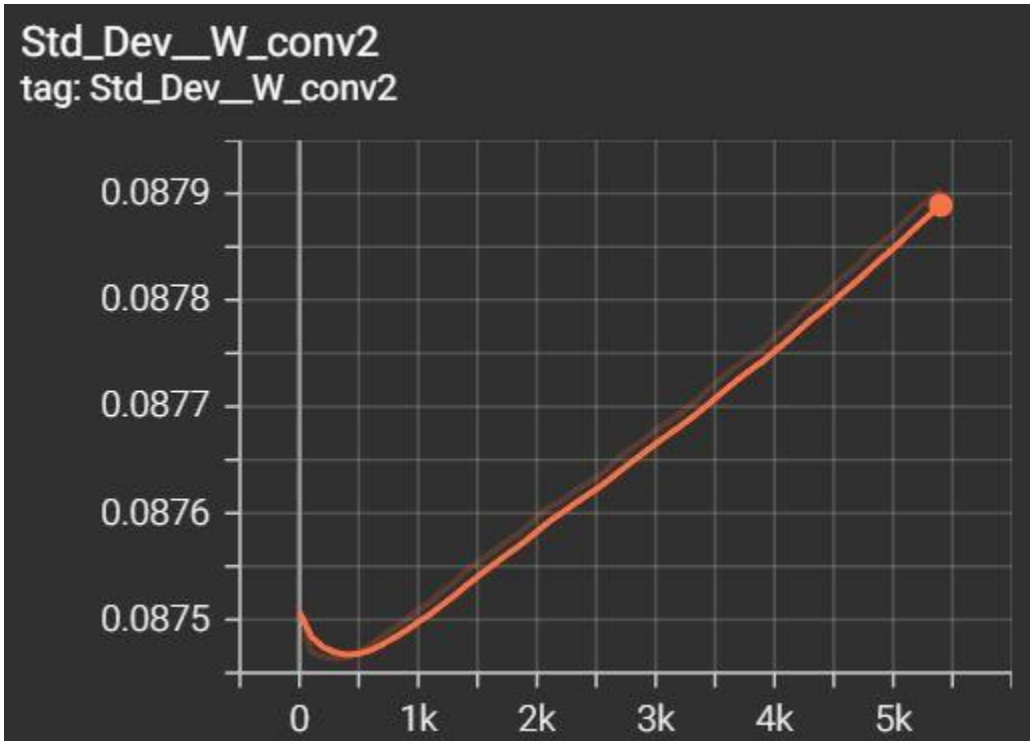
Min__h_conv2

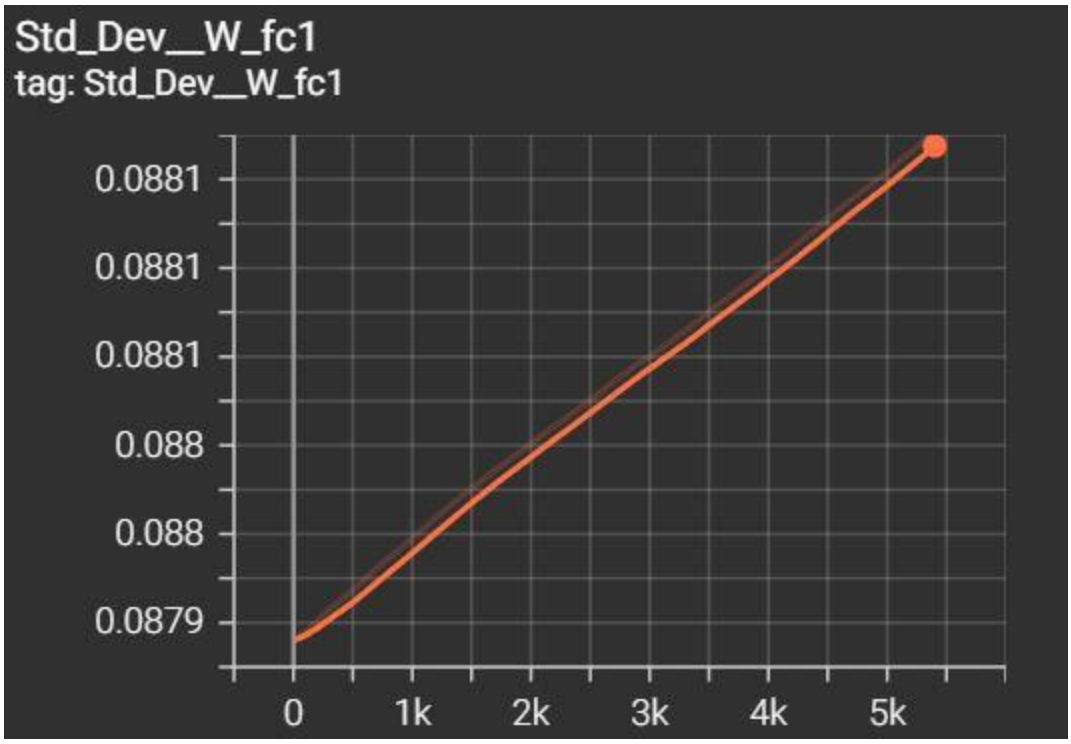

Min__h_conv2
tag: Min__h_conv2
0.8
0.4
0
-0.4
-0.8
0
1k
2k
3k
4k
5k

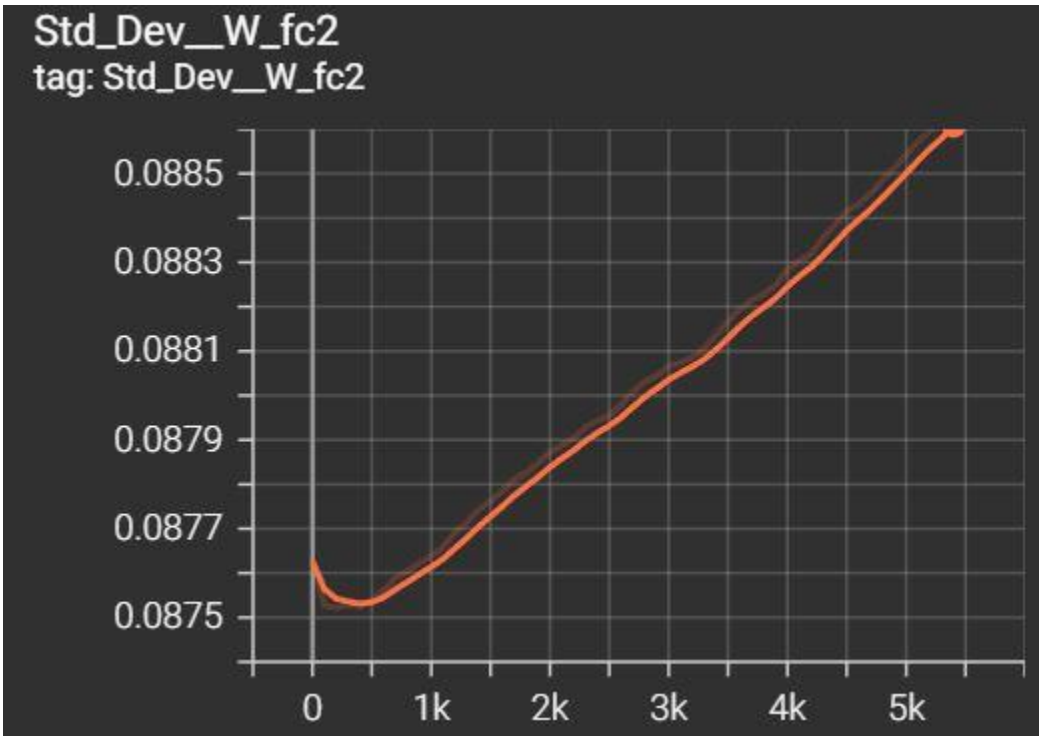

Min__h_fc1
tag: Min__h_fc1
0.8
0.4
0
-0.4
-0.8
0
1k
2k
3k
4k
5k

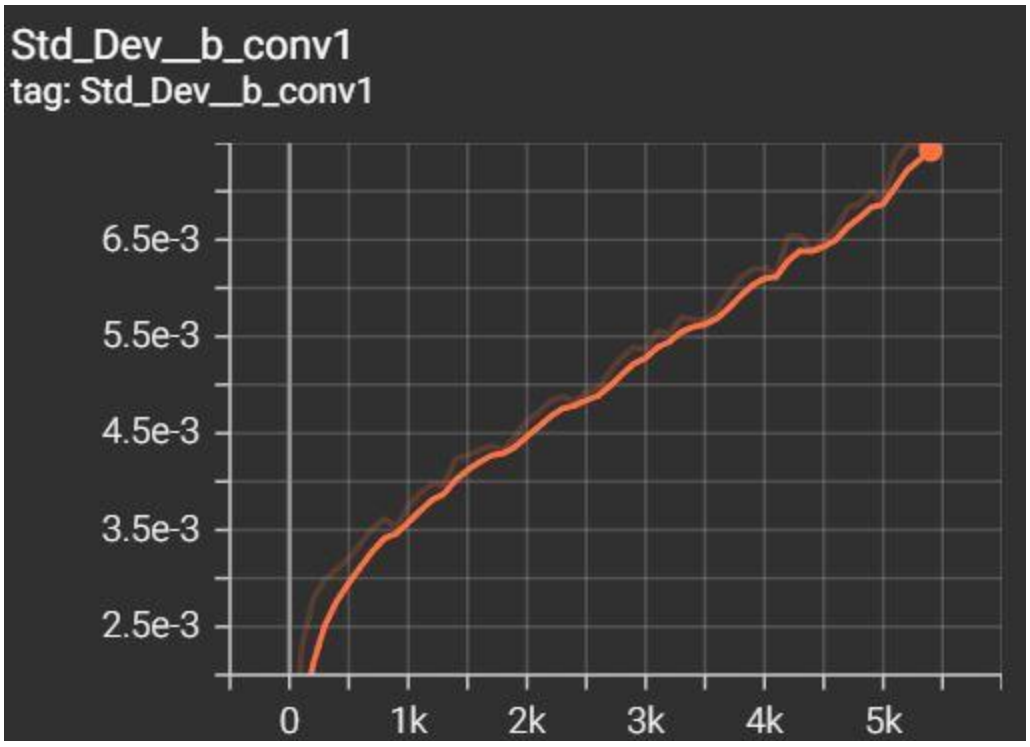

Std_Dev___h_pool1
tag: Std_Dev___h_pool1
0.124
0.123
0.122
0.121
0.12
0.119
0.118
0
1k
2k
3k
4k
5k

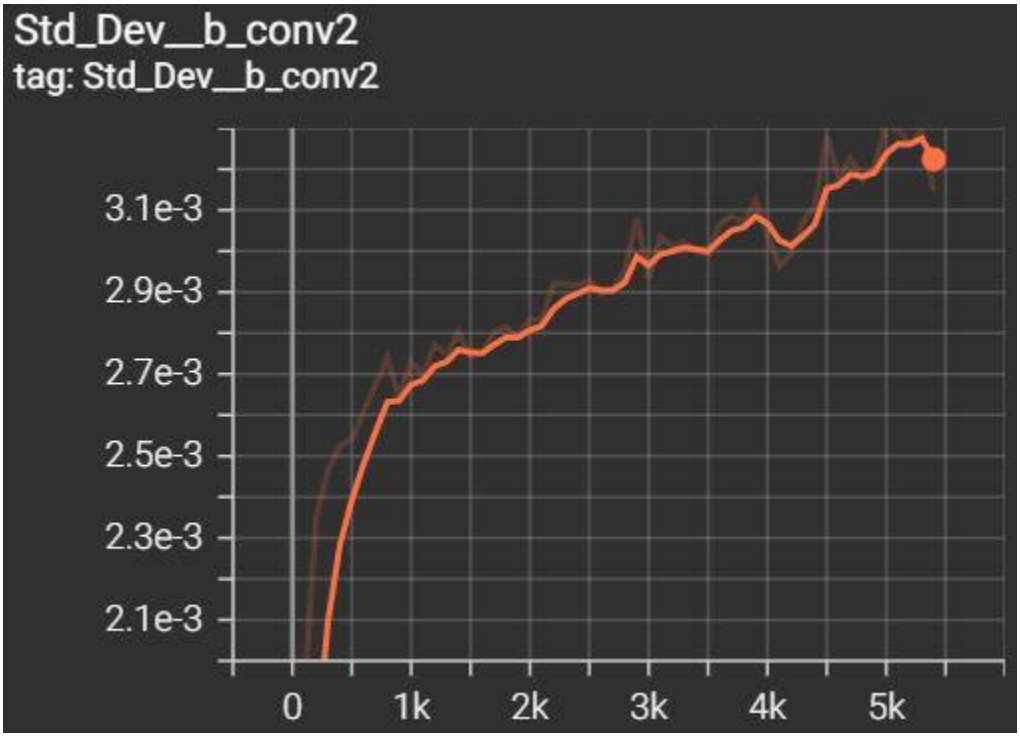

Std_Dev__W_conv1
tag: Std_Dev__W_conv1
0.0905
0.0895
0.0885
0.0875
0.0865
0
1k
2k
3k
4k
5k

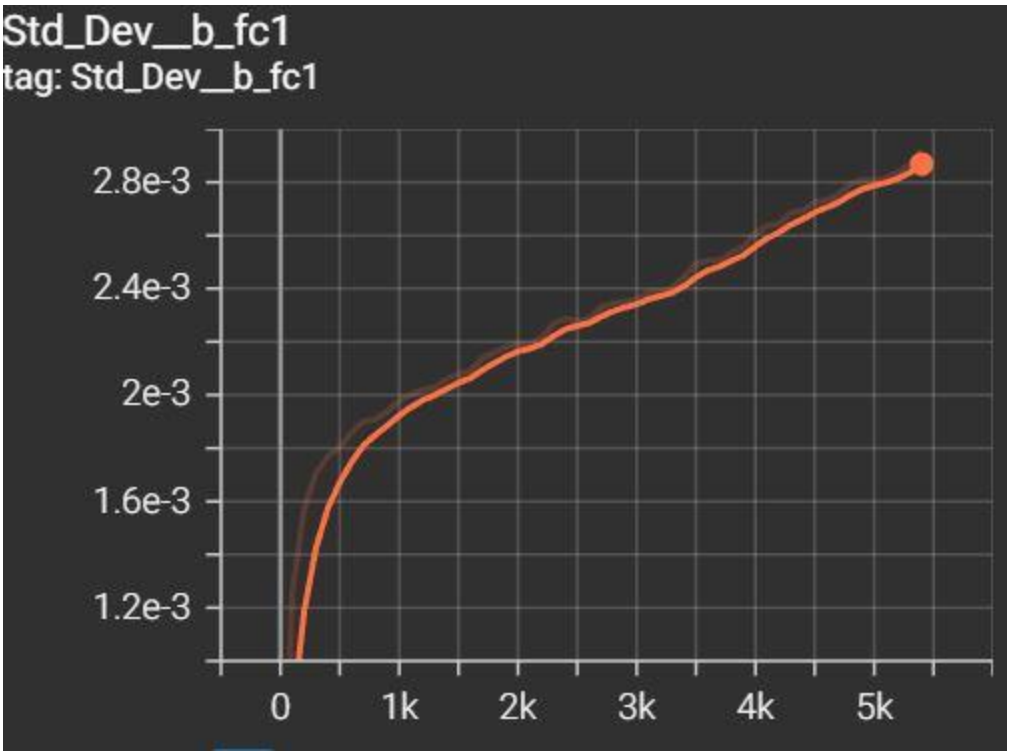

Std_Dev__W_conv2
tag: Std_Dev__W_conv2
0.0879
0.0878
0.0877
0.0876
0.0875
0
1k
2k
3k
4k
5k

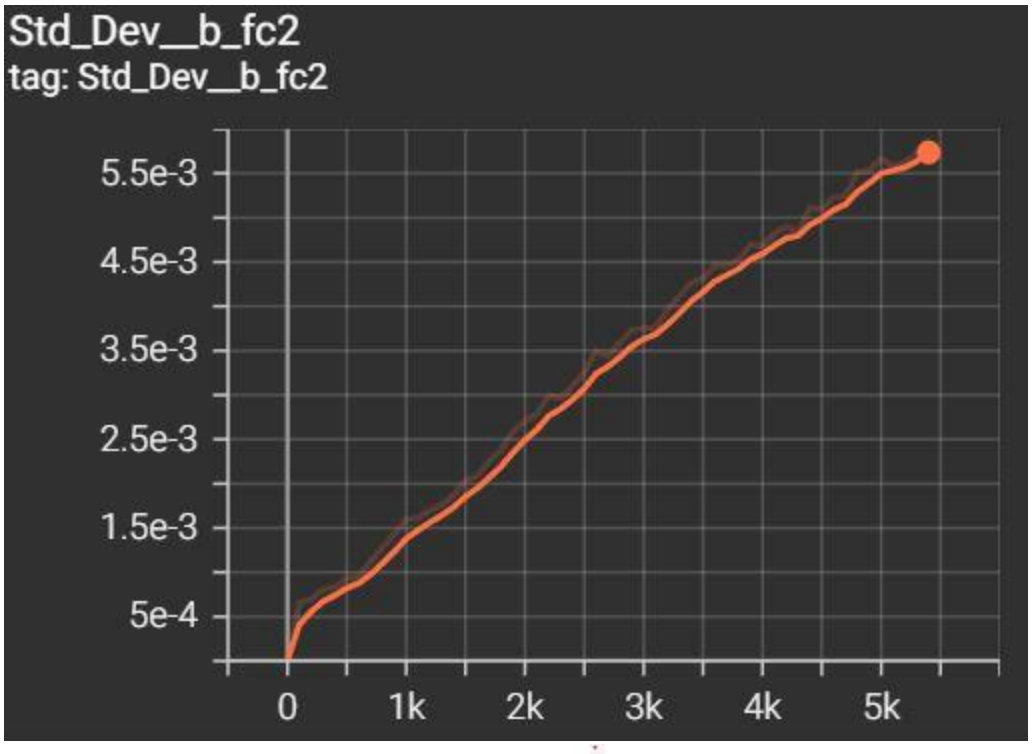

Std_Dev__W_fc1
tag: Std_Dev__W_fc1
0.0881
0.0881
0.0881
0.088
0.088
0.0879
0
1k
2k
3k
4k
5k

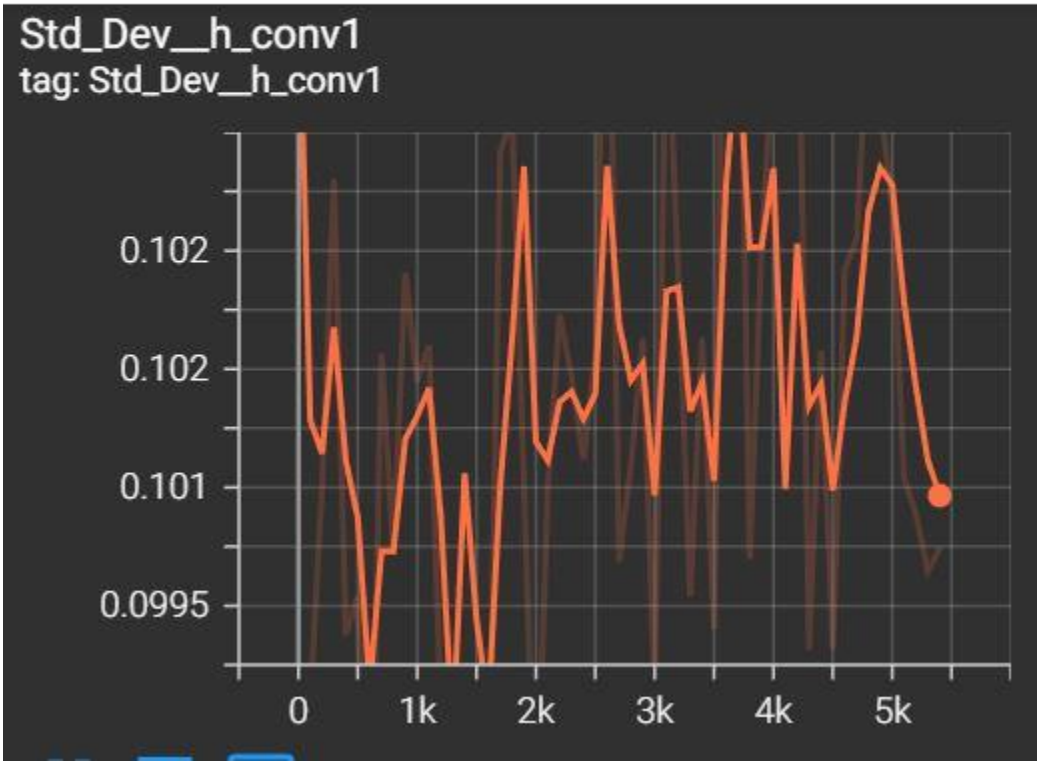

Std_Dev__W_fc2
tag: Std_Dev__W_fc2
0.0885
0.0883
0.0881
0.0879
0.0877
0.0875
0
1k
2k
3k
4k
5k

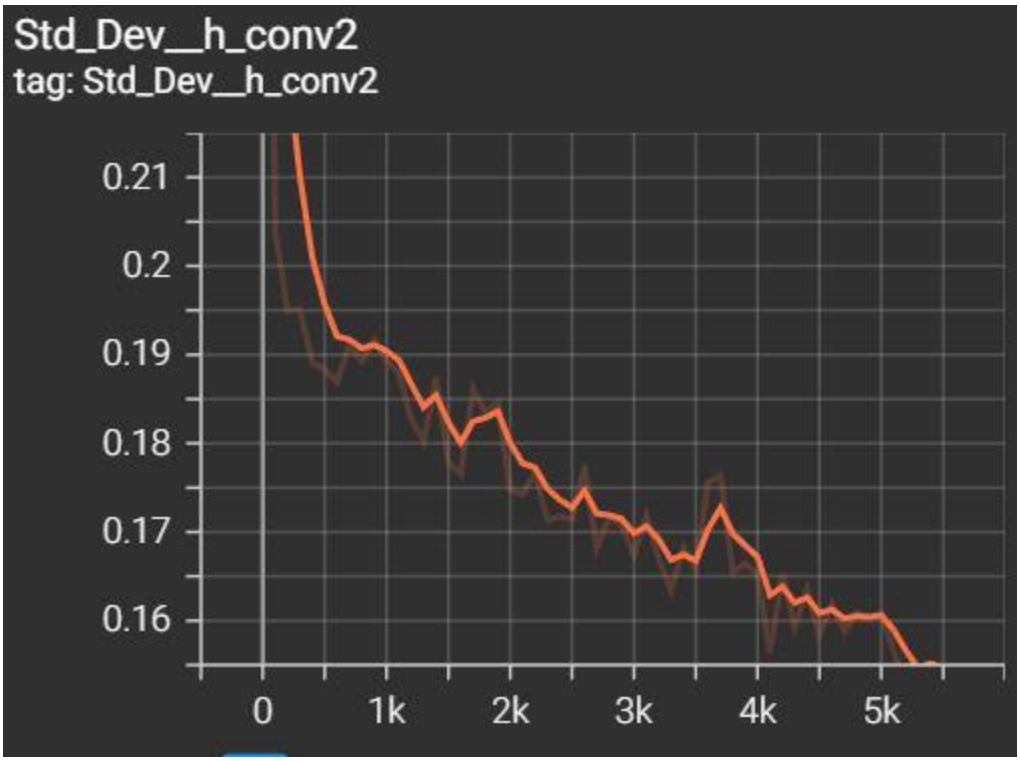

Std_Dev__b_conv1
tag: Std_Dev__b_conv1
6.5e-3
5.5e-3
4.5e-3
3.5e-3
2.5e-3
0
1k
2k
3k
4k
5k

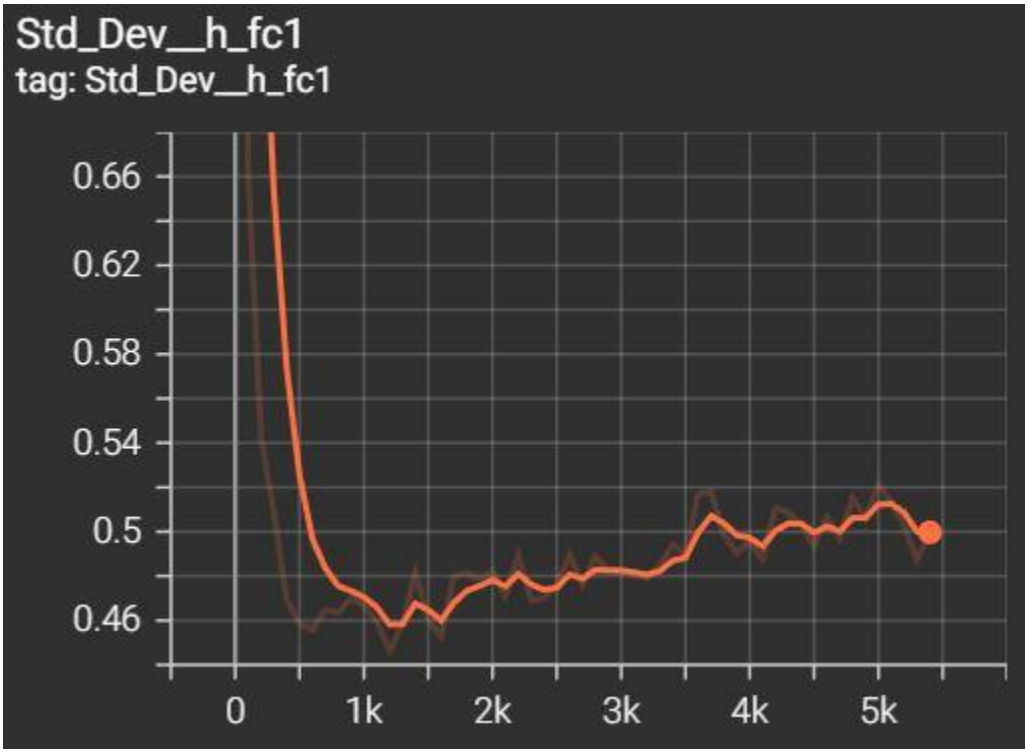

Std_Dev__b_conv2
tag: Std_Dev__b_conv2
3.1e-3
2.9e-3
2.7e-3
2.5e-3
2.3e-3
2.1e-3
0
1k
2k
3k
4k
5k

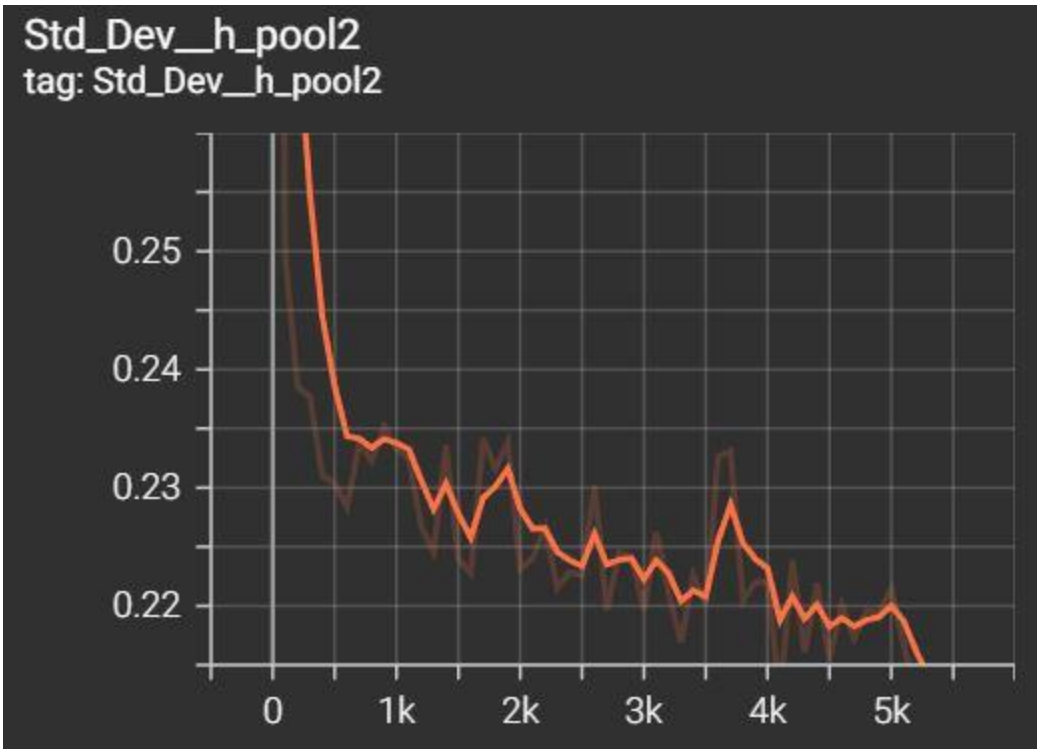

Std_Dev__b_fc1
tag: Std_Dev__b_fc1
2.8e-3
2.4e-3
2e-3
1.6e-3
1.2e-3
0
1k
2k
3k
4k
5k

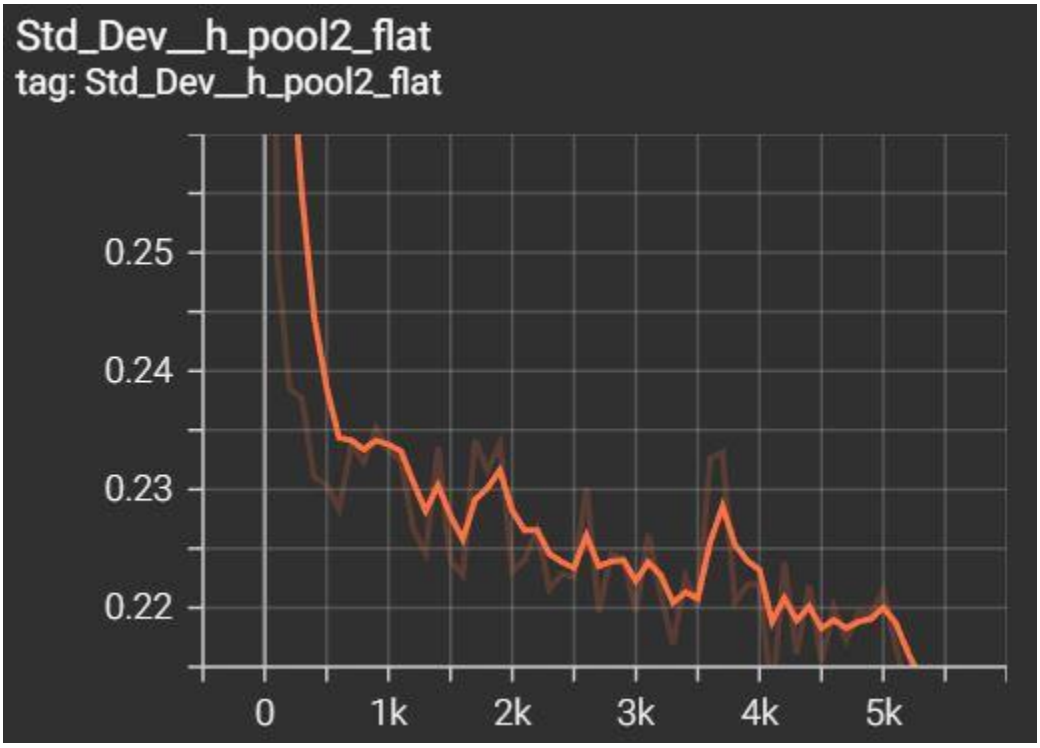

Std_Dev__b_fc2
tag: Std_Dev__b_fc2
5.5e-3
4.5e-3
3.5e-3
2.5e-3
1.5e-3
5e-4
0
1k
2k
3k
4k
5k

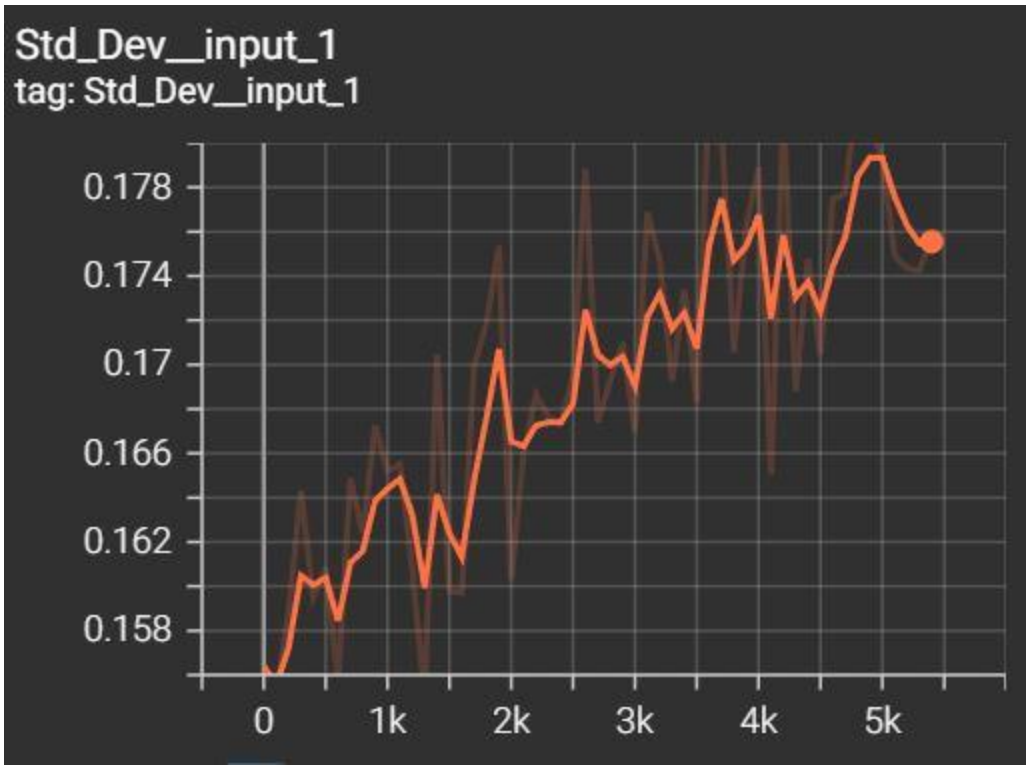

Std_Dev__h_conv1
tag: Std_Dev__h_conv1
0.102
0.102
0.101
0.0995
0
1k
2k
3k
4k
5k

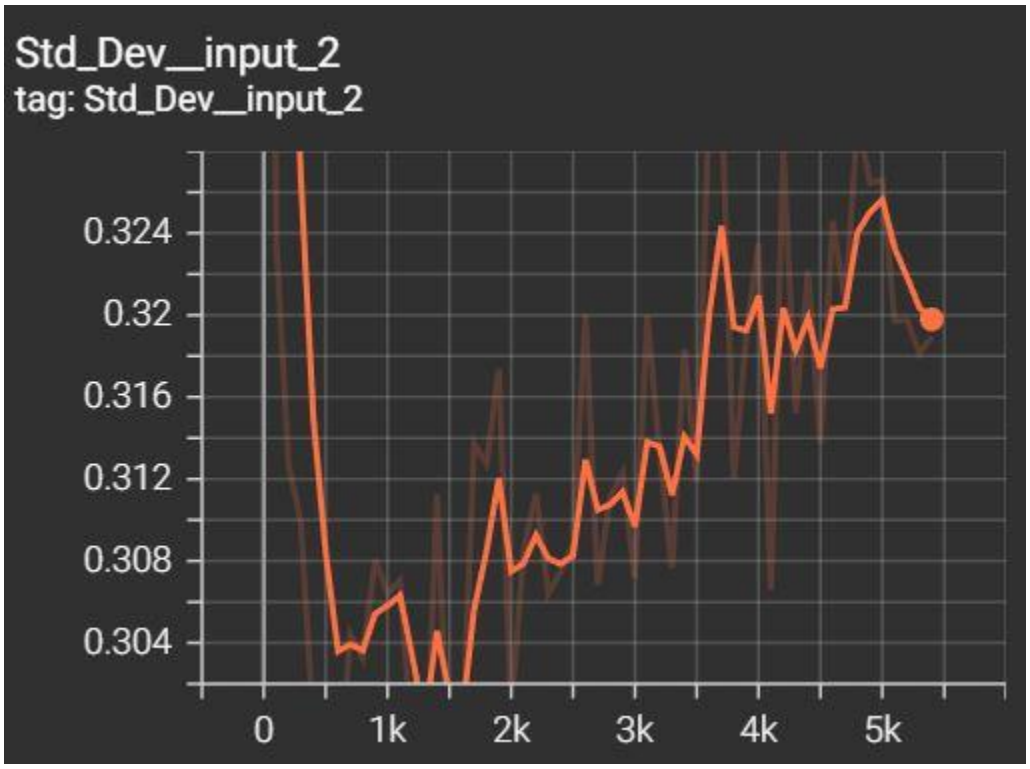

Std_Dev__h_conv2
tag: Std_Dev__h_conv2
0.21
0.2
0.19
0.18
0.17
0.16
0
1k
2k
3k
4k
5k

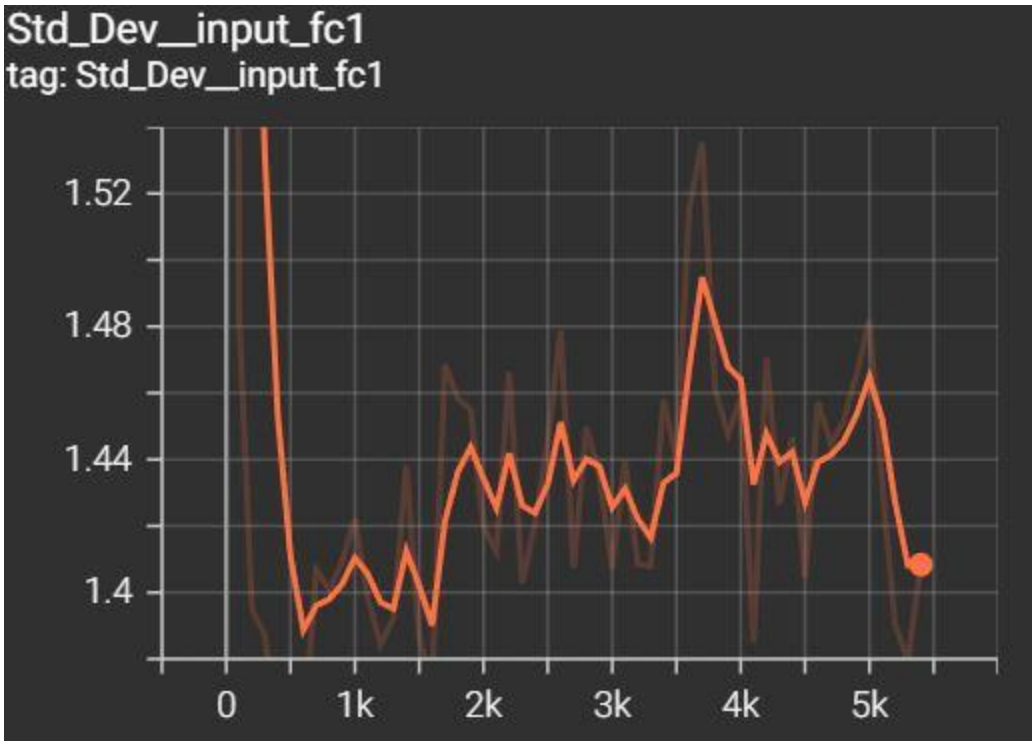

Std_Dev__h_fc1
tag: Std_Dev__h_fc1
0.66
0.62
0.58
0.54
0.5
0.46
0
1k
2k
3k
4k
5k

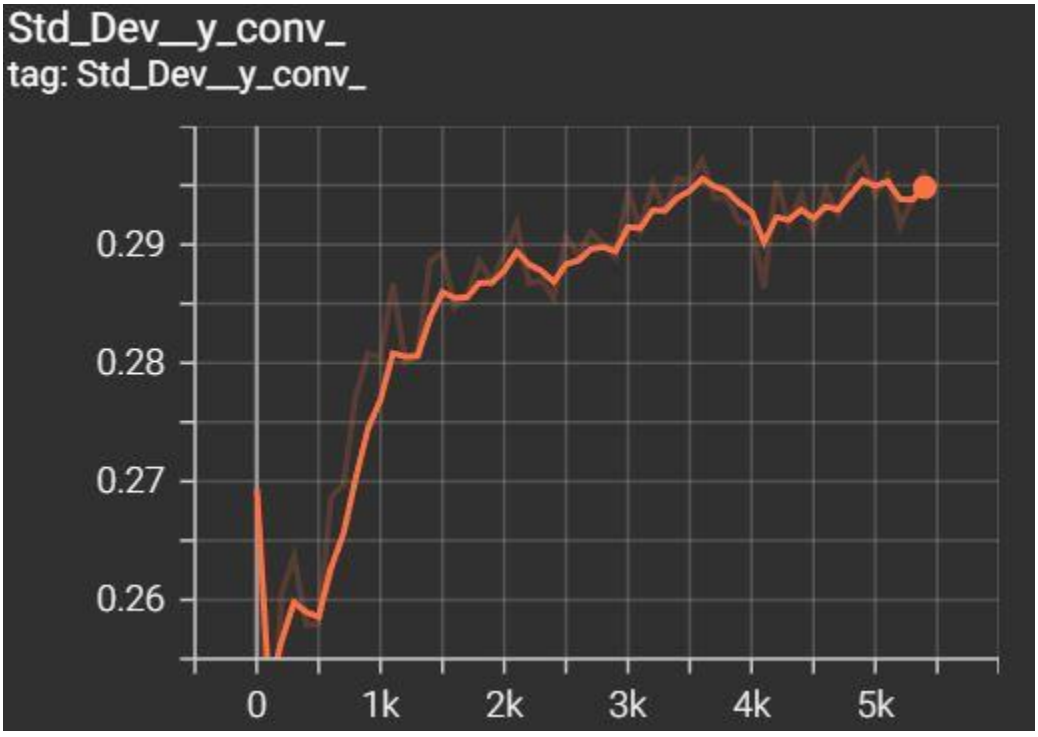

Std_Dev__h_pool2
tag: Std_Dev__h_pool2
0.25
0.24
0.23
0.22
0
1k
2k
3k
4k
5k

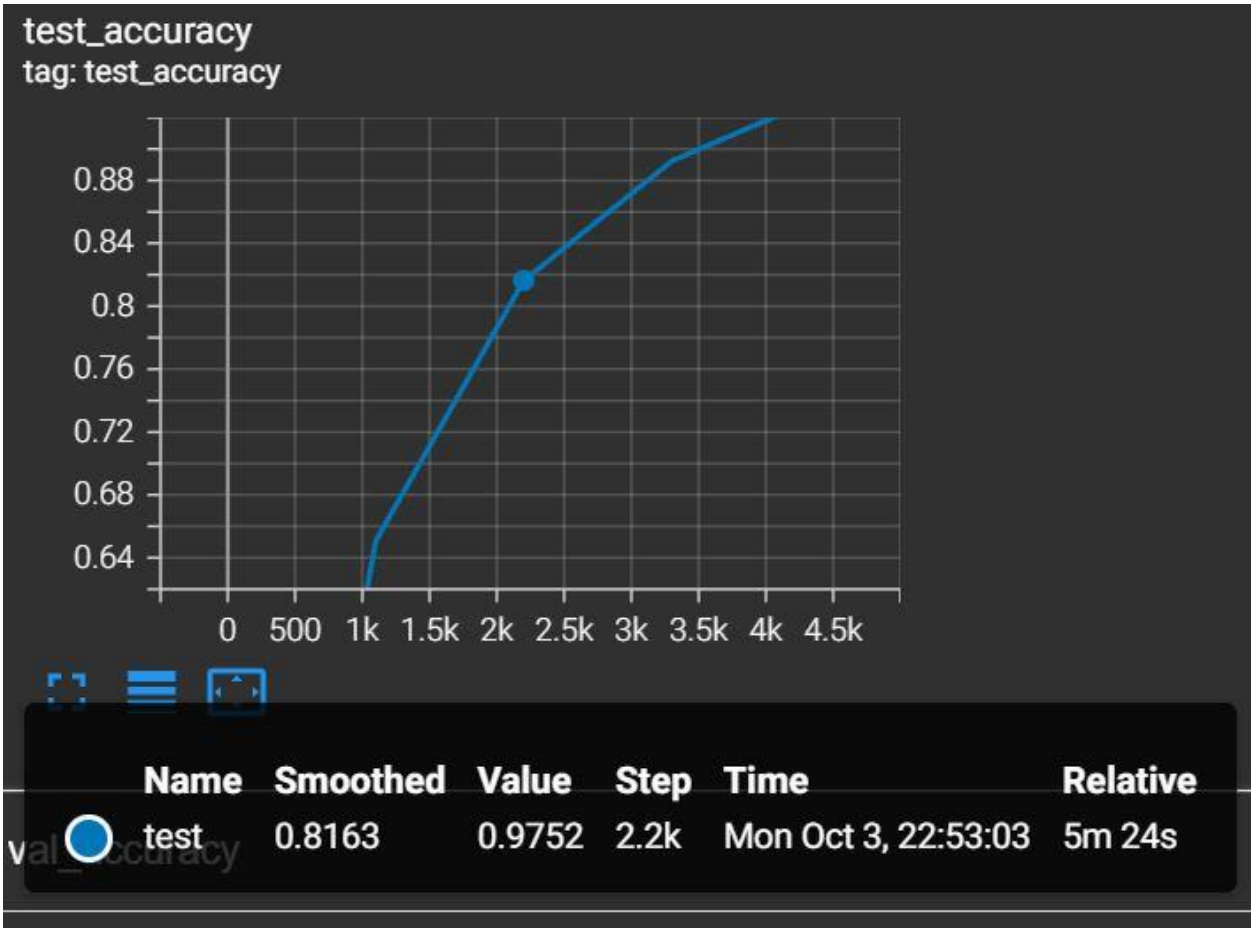

Std_Dev__h_pool2_flat
tag: Std_Dev__h_pool2_flat
0.25
0.24
0.23
0.22
0
1k
2k
3k
4k
5k

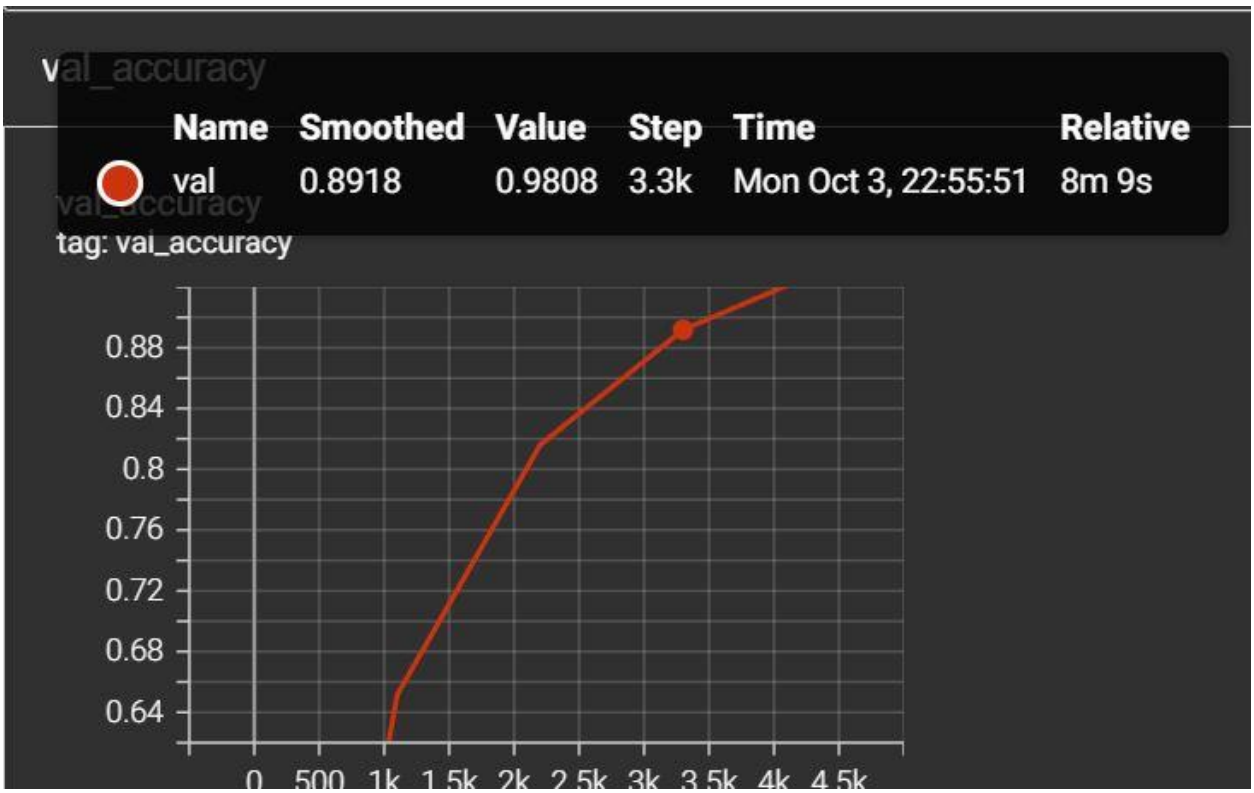

Std_Dev__input_1
tag: Std_Dev__input_1
0.178
0.174
0.17
0.166
0.162
0.158
0
1k
2k
3k
4k
5k


Std_Dev__input_2
tag: Std_Dev__input_2
0.324
0.32
0.316
0.312
0.308
0.304
0
1k
2k
3k
4k
5k


Std_Dev__input_fc1
tag: Std_Dev__input_fc1
1.52
1.48
1.44
1.4
0
1k
2k
3k
4k
5k


Std_Dev__y_conv_
tag: Std_Dev__y_conv_
0.29
0.28
0.27
0.26
0
1k
2k
3k
4k
5k


test_accuracy
tag: test_accuracy
0.88
0.84
0.8
0.76
0.72
0.68
0.64
0
500
1k
1.5k
2k
2.5k
3k
3.5k
4k
4.5k
Name
Smoothed
Value
Step
Time
Relative
test
0.8163
0.9752
2.2k
Mon Oct 3, 22:53:03
5m 24s



Problem 2 Part B:

- The approach for this part is to use the following code shown below to plot more information about the training results:

```
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)
```

Problem 2 Part C:

- The approach for this part is to use gradient descent instead of the Adam training algorithm this can be done by the following steps:
    - Step 1 , convert ReLU to tanh function as shown in the code below:

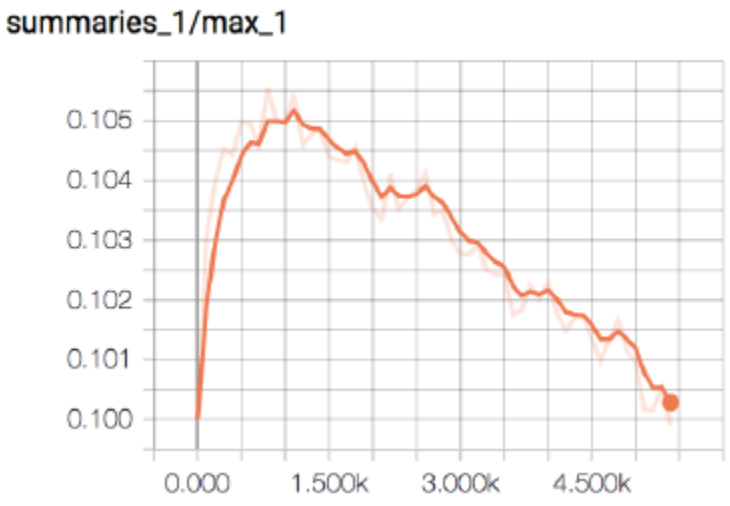
```
    # placeholders for input data and input labeles
    x = tf.placeholder(tf.float32, [None, 784], name='x')
    y_ = tf.placeholder(tf.float32, [None, 10], name='y_')

    # Store Accuracies
    val_accuracy_ = tf.placeholder(tf.float32, shape=())
    test_accuracy_ = tf.placeholder(tf.float32, shape=())

    # 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])
    input_1 = conv2d(x_image, W_conv1) + b_conv1
    h_conv1 = tf.nn.tanh(input_1)
    h_pool1 = max_pool_2x2(h_conv1)

    # second convolutional layer
    W_conv2 = weight_variable([5, 5, 32, 64])
    b_conv2 = bias_variable([64])
    input_2 = conv2d(h_pool1, W_conv2) + b_conv2
    h_conv2 = tf.nn.tanh(input_2)
    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])
    input_fc1 = tf.matmul(h_pool2_flat, W_fc1) + b_fc1
    h_fc1 = tf.nn.tanh(input_fc1)

    # dropout
    keep_prob = tf.placeholder(tf.float32)
    h_fc1_drop = 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.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2, name='y')
```

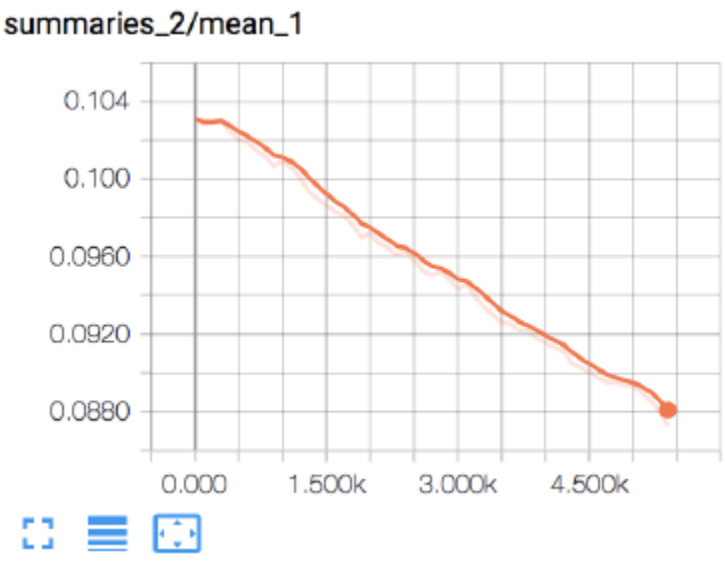
    - Step 2, Change from Adam optimizer to Gradient Descent as shown in the code below:

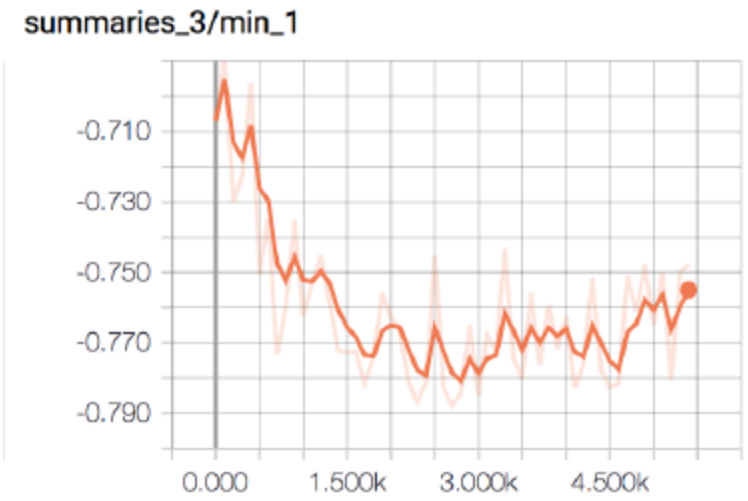
```
    # setup training
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv),
reduction_indices=[1]))
    #train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
    train_step = tf.train.tf.train.GradientDescentOptimizer(1e-
4).minimize(cross_entropy)
```

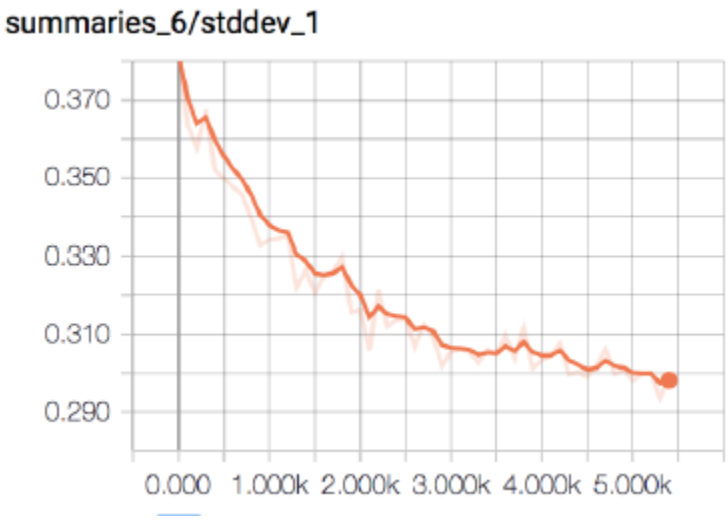
```
    correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32),
name='accuracy')
```

- The results are shown in the figures below:









REFERENCES:

- The code below is the entire code used for Problem 1 of the homework:

```
import numpy as np
from sklearn import datasets, linear_model
import matplotlib.pyplot as plt
from tensorflow.keras.utils import to_categorical
from scipy.special import factorial
def generate_data():
    '''
    generate data
    :return: X: input data, y: given labels
```

```
    '''
    np.random.seed(0)
    X, y = datasets.make_moons(200, noise=0.20)
    return X, y

def plot_decision_boundary(pred_func, X, y):
    '''
    plot the decision boundary
    :param pred_func: function used to predict the label
    :param X: input data
    :param y: given labels
    :return:
    '''
    # Set min and max values and give it some padding
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
h))
    # Predict the function value for the whole gid
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)
    plt.show()

##############################################################################
##########################################
##############################################################################
##########################################
# YOUR ASSSIGMENT STARTS HERE
# FOLLOW THE INSTRUCTION BELOW TO BUILD AND TRAIN A 3-LAYER NEURAL NETWORK
##############################################################################
##########################################
##############################################################################
##########################################
class NeuralNetwork(object):
    """
    This class builds and trains a neural network
    """
    def __init__(self, nn_input_dim, nn_hidden_dim , nn_output_dim,
actFun_type='tanh', reg_lambda=0.01, seed=0):
        '''
        :param nn_input_dim: input dimension
        :param nn_hidden_dim: the number of hidden units
        :param nn_output_dim: output dimension
        :param actFun_type: type of activation function. 3 options: 'tanh',
'sigmoid', 'relu'
        :param reg_lambda: regularization coefficient
        :param seed: random seed
        '''
        self.nn_input_dim = nn_input_dim
        self.nn_hidden_dim = nn_hidden_dim
        self.nn_output_dim = nn_output_dim
        self.actFun_type = actFun_type
        self.reg_lambda = reg_lambda

        # initialize the weights and biases in the network
        np.random.seed(seed)
        self.W1 = np.random.randn(self.nn_input_dim, self.nn_hidden_dim) /
np.sqrt(self.nn_input_dim)
```

```
        self.b1 = np.zeros((1, self.nn_hidden_dim))
        self.W2 = np.random.randn(self.nn_hidden_dim, self.nn_output_dim) /
np.sqrt(self.nn_hidden_dim)
        self.b2 = np.zeros((1, self.nn_output_dim))

    def actFun(self, z, type):

        if type == 'tanh':
            z = (2/(1+np.exp(-2*z)))-1
            #return np.tanh(z)
            return z
        elif type == 'sigmoid':
            z = 1/(1+np.exp(-1*z))
        elif type == 'relu':
          return np.maximum(0, z)

        else:
            print("Incorrect function detected,\
                  please enter one of the following functions:\
                  ReLU, Sigmoid or Tanh")
        return z

    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
        '''

        if type == 'tanh':
            z = (1/np.square(np.cosh(z))) #formula created using
tanh^2+sech^2=1
            #return 1 - np.square(np.tanh(z))
            #f = (2/(1+np.exp(-2*z)))-1
            #z = 1 - np.square(f)
            return z
        elif type == 'sigmoid':
            z = np.exp(-z)/(1 + 2*np.exp(-z) + np.exp(-2*z))
        elif type == 'relu':
          return np.where(z > 0, 1, 0)

        else:
            print("Incorrect function detected,\
                  please enter one of the following functions:\
                  ReLU, Sigmoid or Tanh")
        return z

    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


        #Version 1 of a2 = ^y = softmax(z2)
        #self.probs = np.exp(self.z2)/ np.sum(np.exp(self.z2),axis = 1,
keepdims = True)

        #version 2 of a2 = ^y = softmax(z2)
        A = np.exp(self.z2 - np.max(self.z2))
        B = np.sum(A,axis = 1, keepdims = True)
        result = A / B
        self.probs = result

        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
        yhat = self.probs #This is the same equation found feedforward
function
        data_loss = np.sum(-np.log(yhat[range(num_examples), y]))



        # 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

    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, type=self.actFun_type))
        return np.argmax(self.probs, axis=1)

    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 = (self.a1.T).dot(delta3)
        db2 = np.sum(delta3, axis=0, keepdims=True)
```

```
        delta2 = delta3.dot(self.W2.T) * self.diff_actFun(self.z1,
self.actFun_type)
        dW1 = np.dot(X.T, delta2)
        db1 = np.sum(delta2, axis=0)

        return dW1, dW2, db1, db2

    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
            dW1, dW2, db1, db2 = self.backprop(X, y)

            # Add derivatives of regularization terms (b1 and b2 don't have
regularization terms)
            dW2 += self.reg_lambda * self.W2
            dW1 += self.reg_lambda * self.W1

            # Gradient descent parameter update
            self.W1 += -epsilon * dW1
            self.b1 += -epsilon * db1
            self.W2 += -epsilon * dW2
            self.b2 += -epsilon * db2

            # 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:
                print("Loss after iteration %i: %f" % (i,
self.calculate_loss(X, y)))

    def visualize_decision_boundary(self, X, y):
        '''
        visualize_decision_boundary plot the decision boundary created by
the trained network
        :param X: input data
        :param y: given labels
        :return:
        '''
        plot_decision_boundary(lambda x: self.predict(x), X, y)


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

model = NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3, nn_output_dim=2,
actFun_type='relu')
model.fit_model(X,y)
```

```
model.visualize_decision_boundary(X,y)
```

- The code below is the **DeepNeuralNetwork class** code used for Problem 1 of the homework:

```
import numpy as np
from sklearn import datasets, linear_model
import matplotlib.pyplot as plt
from tensorflow.keras.utils import to_categorical
from scipy.special import factorial
def generate_data():
    '''
    generate data
    :return: X: input data, y: given labels
    '''
    np.random.seed(0)
    X, y = datasets.make_moons(200, noise=0.20)
    return X, y

def plot_decision_boundary(pred_func, X, y):
    '''
    plot the decision boundary
    :param pred_func: function used to predict the label
    :param X: input data
    :param y: given labels
    :return:
    '''
    # Set min and max values and give it some padding
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
h))
    # Predict the function value for the whole gid
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)
    plt.show()

##############################################################################
########################################
##############################################################################
########################################
# YOUR ASSSIGMENT STARTS HERE
# FOLLOW THE INSTRUCTION BELOW TO BUILD AND TRAIN A 3-LAYER NEURAL NETWORK
##############################################################################
########################################
##############################################################################
########################################
class NeuralNetwork(object):
    """
    This class builds and trains a neural network
    """
```

```
    def __init__(self, nn_input_dim, nn_hidden_dim , nn_output_dim,
actFun_type='tanh', reg_lambda=0.01, seed=0):
        '''
        :param nn_input_dim: input dimension
        :param nn_hidden_dim: the number of hidden units
        :param nn_output_dim: output dimension
        :param actFun_type: type of activation function. 3 options: 'tanh',
'sigmoid', 'relu'
        :param reg_lambda: regularization coefficient
        :param seed: random seed
        '''
        self.nn_input_dim = nn_input_dim
        self.nn_hidden_dim = nn_hidden_dim
        self.nn_output_dim = nn_output_dim
        self.actFun_type = actFun_type
        self.reg_lambda = reg_lambda

        # initialize the weights and biases in the network
        np.random.seed(seed)
        self.W1 = np.random.randn(self.nn_input_dim, self.nn_hidden_dim) /
np.sqrt(self.nn_input_dim)
        self.b1 = np.zeros((1, self.nn_hidden_dim))
        self.W2 = np.random.randn(self.nn_hidden_dim, self.nn_output_dim) /
np.sqrt(self.nn_hidden_dim)
        self.b2 = np.zeros((1, self.nn_output_dim))

    def actFun(self, z, type):

        if type == 'tanh':
            z = (2/(1+np.exp(-2*z)))-1
            #return np.tanh(z)
            return z
        elif type == 'sigmoid':
            z = 1/(1+np.exp(-1*z))
        elif type == 'relu':
          return np.maximum(0, z)

        else:
            print("Incorrect function detected,\
                  please enter one of the following functions:\
                  ReLU, Sigmoid or Tanh")
        return z

    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
        '''

        if type == 'tanh':
            z = (1/np.square(np.cosh(z))) #formula created using
tanh^2+sech^2=1
            #return 1 - np.square(np.tanh(z))
            #f = (2/(1+np.exp(-2*z)))-1
            #z = 1 - np.square(f)
            return z
        elif type == 'sigmoid':
            z = np.exp(-z)/(1 + 2*np.exp(-z) + np.exp(-2*z))
        elif type == 'relu':
```

```
            return np.where(z > 0, 1, 0)

        else:
            print("Incorrect function detected,\
                  please enter one of the following functions:\
                  ReLU, Sigmoid or Tanh")
        return z

    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


        #Version 1 of a2 = ^y = softmax(z2)
        #self.probs = np.exp(self.z2)/ np.sum(np.exp(self.z2),axis = 1,
keepdims = True)

        #version 2 of a2 = ^y = softmax(z2)
        A = np.exp(self.z2 - np.max(self.z2))
        B = np.sum(A,axis = 1, keepdims = True)
        result = A / B
        self.probs = result

        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
        yhat = self.probs #This is the same equation found feedforward
function
        data_loss = np.sum(-np.log(yhat[range(num_examples), y]))



        # 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

    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, type=self.actFun_type))
        return np.argmax(self.probs, axis=1)

    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 = (self.a1.T).dot(delta3)
        db2 = np.sum(delta3, axis=0, keepdims=True)
        delta2 = delta3.dot(self.W2.T) * self.diff_actFun(self.z1,
self.actFun_type)
        dW1 = np.dot(X.T, delta2)
        db1 = np.sum(delta2, axis=0)

        return dW1, dW2, db1, db2

    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
            dW1, dW2, db1, db2 = self.backprop(X, y)

            # Add derivatives of regularization terms (b1 and b2 don't have
regularization terms)
            dW2 += self.reg_lambda * self.W2
            dW1 += self.reg_lambda * self.W1

            # Gradient descent parameter update
            self.W1 += -epsilon * dW1
            self.b1 += -epsilon * db1
            self.W2 += -epsilon * dW2
            self.b2 += -epsilon * db2

            # 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:
```

```
                print("Loss after iteration %i: %f" % (i,
self.calculate_loss(X, y)))

    def visualize_decision_boundary(self, X, y):
        '''
        visualize_decision_boundary plot the decision boundary created by
the trained network
        :param X: input data
        :param y: given labels
        :return:
        '''
        plot_decision_boundary(lambda x: self.predict(x), X, y)



class DeepNeuralNetwork(object):
    """
    This class builds and trains a deep neural network
    """

    def __init__(self, nn_dims, actFun_type='tanh', reg_lambda=0.01,
seed=0):

        self.nn_dims = nn_dims
        self.actFun_type = actFun_type
        self.reg_lambda = reg_lambda

        # initialize the weights and biases in the network
        self.W = []
        self.b = []

        np.random.seed(seed)
        for i in range(len(nn_dims)-1):
            self.W.append(np.random.randn(self.nn_dims[i],
self.nn_dims[i+1])
                          / np.sqrt(self.nn_dims[i]))
            self.b.append(np.zeros((1, self.nn_dims[i+1])))

    def actFun(self, z, type):

        if type == 'tanh':
            z = (2/(1+np.exp(-2*z)))-1
            #return np.tanh(z)
            return z
        elif type == 'sigmoid':
            z = 1/(1+np.exp(-1*z))
        elif type == 'relu':
          return np.maximum(0, z)

        else:
            print("Incorrect function detected,\
                  please enter one of the following functions:\
                  ReLU, Sigmoid or Tanh")
        return z

    def diff_actFun(self, z, type):


        if type == 'tanh':
            z = (1/np.square(np.cosh(z))) #formula created using
tanh^2+sech^2=1
            #return 1 - np.square(np.tanh(z))
```

```
            #f = (2/(1+np.exp(-2*z)))-1
            #z = 1 - np.square(f)
            return z
        elif type == 'sigmoid':
            z = np.exp(-z)/(1 + 2*np.exp(-z) + np.exp(-2*z))
        elif type == 'relu':
          return np.where(z > 0, 1, 0)

        else:
            print("Incorrect function detected,\
                  please enter one of the following functions:\
                  ReLU, Sigmoid or Tanh")
        return z

    def feedforward(self, X, actFun):

        self.a = []
        self.z = []
        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]))
        C = np.exp(self.z[len(self.z)-1])
        self.probs = C / np.sum(C, axis=1, keepdims=True)
        return None

    def calculate_loss(self, X, y):

        num_examples = len(X)
        self.feedforward(X, lambda x: self.actFun(x, type=self.actFun_type))
        # Calculating the loss

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

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

    def predict(self, 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):


        num_examples = len(X)
        delta = self.probs
        delta[range(num_examples), y] -= 1

        db = []
        dW = []
        for num in range(len(self.z)):
            i = len(self.z) - num - 1
```

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

        return dW, db

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

        # 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)

            # Add regularization terms
            for i in range(len(dW)):
                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]

            # print the loss.
            if print_loss and i % 1000 == 0:
                print("Loss after iteration %i: %f" % (i,
self.calculate_loss(X, y)))

    def visualize_decision_boundary(self, X, y):

        plot_decision_boundary(lambda x: self.predict(x), X, y)

def main():

    # # generate and visualize Make-Moons dataset
    #X, y = generate_data()
    # # generate and visualize Make-Circles dataset
    X, y = datasets.make_circles(n_samples=100, shuffle=True, noise=None,
random_state=None, factor=0.8)

    #Train the Neural Network Model
    model = DeepNeuralNetwork(nn_dims=[2,10,8,8,5], actFun_type='sigmoid')
    model.fit_model(X,y)
    model.visualize_decision_boundary(X,y)

if __name__ == "__main__":
    main()
```

- The code below is the entire code used for **problem 2** of the homework:

```
import os
import time

import tensorflow as tf

if (tf.__version__.split('.')[0] == '2'):
    import tensorflow.compat.v1 as tf

    tf.disable_v2_behavior()

# Load MNIST dataset
#import input_data


import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

sess = tf.InteractiveSession()


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


def stats_summary(var_name, var):
    ##monitor the statistics (min, max, mean, standard deviation, histogram)
    tf.summary.scalar("Min__" + var_name, tf.reduce_min(var))
    tf.summary.scalar("Max__" + var_name, tf.reduce_max(var))
    tf.summary.scalar("Mean__" + var_name, tf.reduce_mean(var))
    tf.summary.scalar("Std_Dev__" + var_name, tf.math.reduce_std(var))
    tf.summary.histogram("Histo__" + var_name, var)


def main():
    # Specify training parameters
    result_dir = './results/'  # directory where the results from the
training are saved
    max_step = 5500  # the maximum iterations. After max_step iterations,
the training will stop no matter what
    test_dir = result_dir + 'test/'
    val_dir = result_dir + 'val/'

    start_time = time.time()  # start timing

    # FILL IN THE CODE BELOW TO BUILD YOUR NETWORK

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

    # Store Accuracies
    val_accuracy_ = tf.placeholder(tf.float32, shape=())
    test_accuracy_ = tf.placeholder(tf.float32, shape=())

    # 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])
    input_1 = conv2d(x_image, W_conv1) + b_conv1
    h_conv1 = tf.nn.relu(input_1)
    h_pool1 = max_pool_2x2(h_conv1)

    # second convolutional layer
    W_conv2 = weight_variable([5, 5, 32, 64])
    b_conv2 = bias_variable([64])
    input_2 = conv2d(h_pool1, W_conv2) + b_conv2
    h_conv2 = tf.nn.relu(input_2)
    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])
    input_fc1 = tf.matmul(h_pool2_flat, W_fc1) + b_fc1
    h_fc1 = tf.nn.relu(input_fc1)

    # dropout
    keep_prob = tf.placeholder(tf.float32)
    h_fc1_drop = 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.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2, name='y')

    # FILL IN THE FOLLOWING CODE TO SET UP THE TRAINING

    # setup training
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv),
reduction_indices=[1]))
    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),
name='accuracy')

    # Add a scalar summary for the snapshot loss.
    tf.summary.scalar(cross_entropy.op.name, cross_entropy)
    # Build the summary operation based on the TF collection of Summaries.
    # Layer-1
    stats_summary("W_conv1", W_conv1)
    stats_summary("b_conv1", b_conv1)
    stats_summary("input_1", input_1)
    stats_summary("h_conv1", h_conv1)
    stats_summary(" h_pool1", h_pool1)
    # Layer-2
    stats_summary("W_conv2", W_conv2)
    stats_summary("b_conv2", b_conv2)
    stats_summary("input_2", input_2)
    stats_summary("h_conv2", h_conv2)
    stats_summary("h_pool2", h_pool2)
    # densely connected layer
    stats_summary("W_fc1", W_fc1)
    stats_summary("b_fc1", b_fc1)
    stats_summary("h_pool2_flat", h_pool2_flat)
    stats_summary("input_fc1", input_fc1)
    stats_summary("h_fc1", h_fc1)
    # Output layer - Softmax
    stats_summary("W_fc2", W_fc2)
    stats_summary("b_fc2", b_fc2)
    stats_summary("y_conv ", y_conv)
```

```
    summary_op = tf.summary.merge_all()
    summary_op_test = tf.summary.scalar('test_accuracy', test_accuracy_)
    summary_op_val = tf.summary.scalar('val_accuracy', val_accuracy_)

    # Add the variable initializer Op.
    init = tf.initialize_all_variables()

    # Create a saver for writing training checkpoints.
    saver = tf.train.Saver()

    # Instantiate a SummaryWriter to output summaries and the Graph.
    summary_writer = tf.summary.FileWriter(result_dir, sess.graph)
    # Val test
    summary_writer_test = tf.summary.FileWriter(test_dir, sess.graph)
    summary_writer_val = tf.summary.FileWriter(val_dir, sess.graph)

    # Run the Op to initialize the variables.
    sess.run(init)

    # run the training
    for i in range(max_step):
        batch = mnist.train.next_batch(50)  # make the data batch, which is
used in the training iteration.
        # the batch size is 50
        if i % 100 == 0:
            # output the training accuracy every 100 iterations
            train_accuracy = accuracy.eval(feed_dict={
                x: batch[0], y_: batch[1], keep_prob: 1.0})
            print("step %d, training accuracy %g" % (i, train_accuracy))

            # Update the events file which is used to monitor the training
(in this case,
            # only the training loss is monitored)
            summary_str = sess.run(summary_op, feed_dict={x: batch[0], y_:
batch[1], keep_prob: 0.5})
            summary_writer.add_summary(summary_str, i)
            summary_writer.flush()

        # save the checkpoints every 1100 iterations
        if i % 1100 == 0 or i == max_step:
            checkpoint_file = os.path.join(result_dir, 'checkpoint')
            saver.save(sess, checkpoint_file, global_step=i)

            feed_dict_test = {x: mnist.test.images, y_: mnist.test.labels,
keep_prob: 1.0}
            test_accuracy = accuracy.eval(feed_dict=feed_dict_test)
            print('step %d, test accuracy %g' % (i, test_accuracy))
            summary_str_test = sess.run(summary_op_test,
feed_dict={test_accuracy_: test_accuracy})
            summary_writer_test.add_summary(summary_str_test, i)
            summary_writer_test.flush()

            feed_dict_val = {x: mnist.validation.images, y_:
mnist.validation.labels, keep_prob: 1.0}
            val_accuracy = accuracy.eval(feed_dict=feed_dict_val)
            print('step %d, validation accuracy %g' % (i, val_accuracy))
            summary_str_val = sess.run(summary_op_val,
feed_dict={val_accuracy_: val_accuracy})
            summary_writer_val.add_summary(summary_str_val, i)
            summary_writer_val.flush()
```

```
        train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob:
0.5})  # run one train_step

    # print validation error
    print('validation accuracy %g' % accuracy.eval(feed_dict={
        x: mnist.validation.images, y_: mnist.validation.labels, keep_prob:
1.0}))

    # print test error
    print("test accuracy %g" % accuracy.eval(feed_dict={
        x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))

    stop_time = time.time()
    print('The training takes %f second to finish' % (stop_time -
start_time))


if __name__ == "__main__":
    main()
```