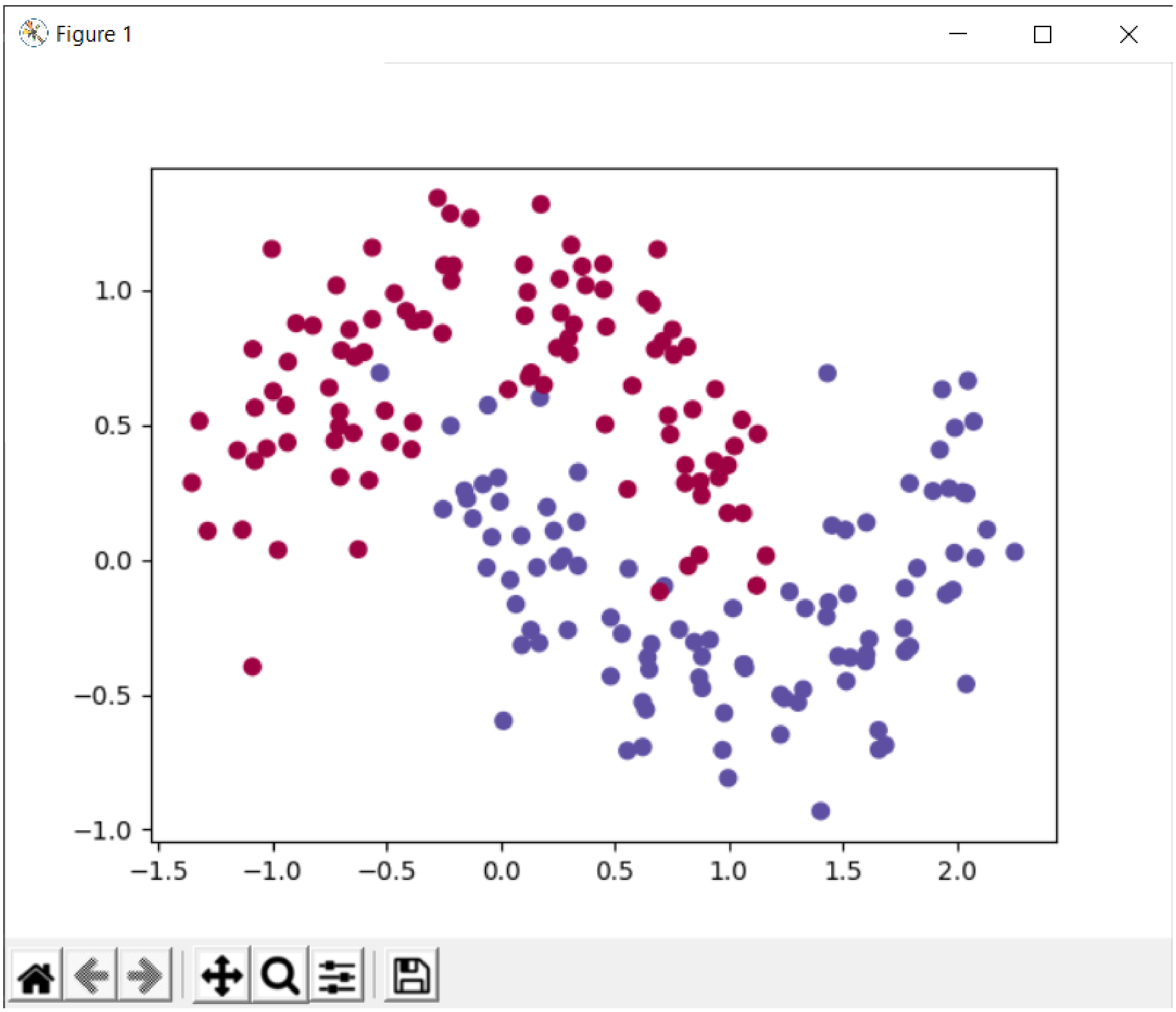
ELEC 576 / COMP 576 – Fall 2021 Assignment 1

Task - 1:

a. **Dataset:** three_layer_neuralnetwork.py



b. **Activation Function:**
Tanh function
tanh = 1−e^−2x/ 1+e^−2x

dtanh(x) /dx = 2e−2x /1+e−2x + ((1−e)∗2 −2x e−2x)/ (1+e−2x)^ 2 = 4e^−2x/ (1+e−2x) ^2 = 1 − tanh^2(x)

Sigmoid function
dsig(x)/ dx = e^−x / (1+e−x)^ −2 = sig(x)(1 − sig(x))

ReLU function
The derivative of x is, 1 for x > 0 and 0 for x = 0 .

c. Build a Neural Network:
Feed forward :

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

    return None
```

Calculate_loss:

```
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
    y_onehot = OneHotEncoder(sparse=False).fit_transform(y.reshape((-1, 1)))
    data_loss = - np.sum(np.log(self.probs) * y_onehot) / num_examples

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

    return output_loss
```

d. Backward Pass - Backward Propagation

1. Derivation :

loss function :-

$$L = -\frac{1}{N} \sum_{n=1}^{N} y_n \log \hat{y}_n$$

For computing we take.

$$l = -y \log y^{\wedge}$$

$$\frac{\partial l}{\partial w_1} = \sum_{k,l} \frac{\partial l}{\partial y^k} \frac{\partial y^k}{\partial z_2} \frac{\partial z_2}{\partial w_1}$$

for softmax = $\frac{\partial y}{\partial z_2} = \begin{cases} -y^k y^l & k \neq l \\ y^k(1-y^k) & k = l \end{cases}$

$$\frac{\partial l}{\partial z_2} = \sum_k \frac{\partial l}{\partial y^k} \frac{\partial y^k}{\partial z_2} = y^l - y^l.$$

$$\boxed{\therefore \frac{\partial l}{\partial w_2} = (y^{\wedge} - y) a_1^T \quad \therefore \frac{\partial l}{\partial w_2^{ij}} = (y^i - y^{i}) a_1^j}$$

with gradient b_2 $\boxed{\frac{\partial l}{\partial b_2} = y^{\wedge} - y}$ $\Rightarrow \frac{\partial l}{\partial b_2} = (y^{\wedge} - y)$

For w_1

$$\frac{\partial l}{\partial w_1} = \sum_{l,k,h} \frac{\partial l}{\partial z_2} \frac{\partial z_2}{\partial a_1^k} \frac{\partial a_1^k}{\partial z_1^h} \frac{\partial z_1^h}{\partial w_2} \Rightarrow \sum_{l,k,h} (y - y^l) w_2 \frac{\partial a_1}{\partial z_1^h} \frac{\partial z_1^h}{\partial w_1}$$

$$\frac{\partial a_1}{\partial z_1} = \begin{cases} a_1^h(1-a_1^h) & k = h \\ 0 & k \neq h \end{cases}$$

$$\boxed{\frac{\partial l}{\partial w_2} = \sum_l (y - y^l) w_2^{li} a_1^i (1 - a_1^i) x_j}$$

$$w_2^T (y^{\wedge} - y) .* a_1 .* (1 - a_1) x^T$$

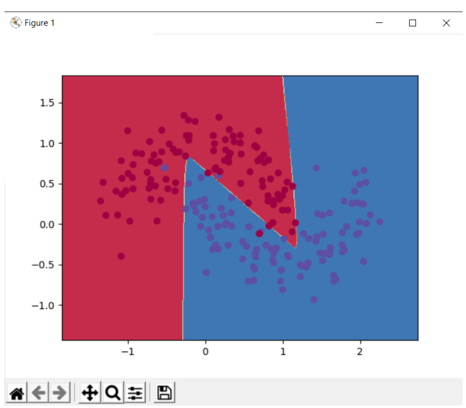
$$\boxed{b_1 = w_2^T (y^{\wedge} - y) .* a_1 .* (1 - a_1)}$$

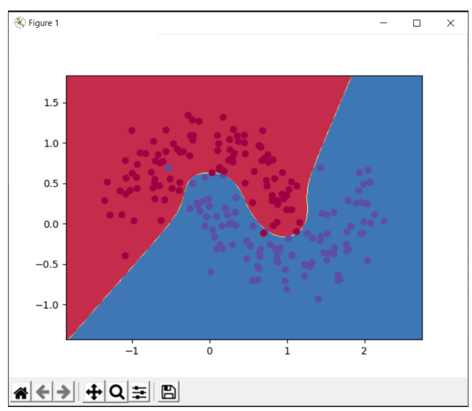
2. **Implementation:**

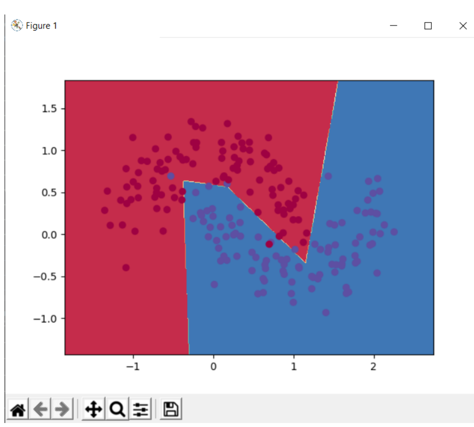
```python
def backprop(self, X, y):
    '''
    backprop run backpropagation to compute the gradients used to update the parameters in the backward step
    :param X: input data
    :param y: given labels
    :return: dL/dW1, dL/b1, dL/dW2, dL/db2
    '''

    # IMPLEMENT YOUR BACKPROP HERE

    num_examples = len(X)
    delta3 = self.probs
    delta3[range(num_examples), y] -= 1
    dW2 = (self.a1.T).dot(delta3)  # dL/dW2
    db2 = np.sum(delta3, axis=0, keepdims=True)  # dL/db2
    delta2 = delta3.dot(self.W2.T) * (self.diff_actFun(self.a1, type=self.actFun_type))
    dW1 = np.dot(X.T, delta2)  # dL/dW1
    db1 = np.sum(delta2, axis=0)  # dL/db1
    return dW1, dW2, db1, db2
```

e. Time to have fun - Training

1. Plots for Tanh, sigmoid,relu:

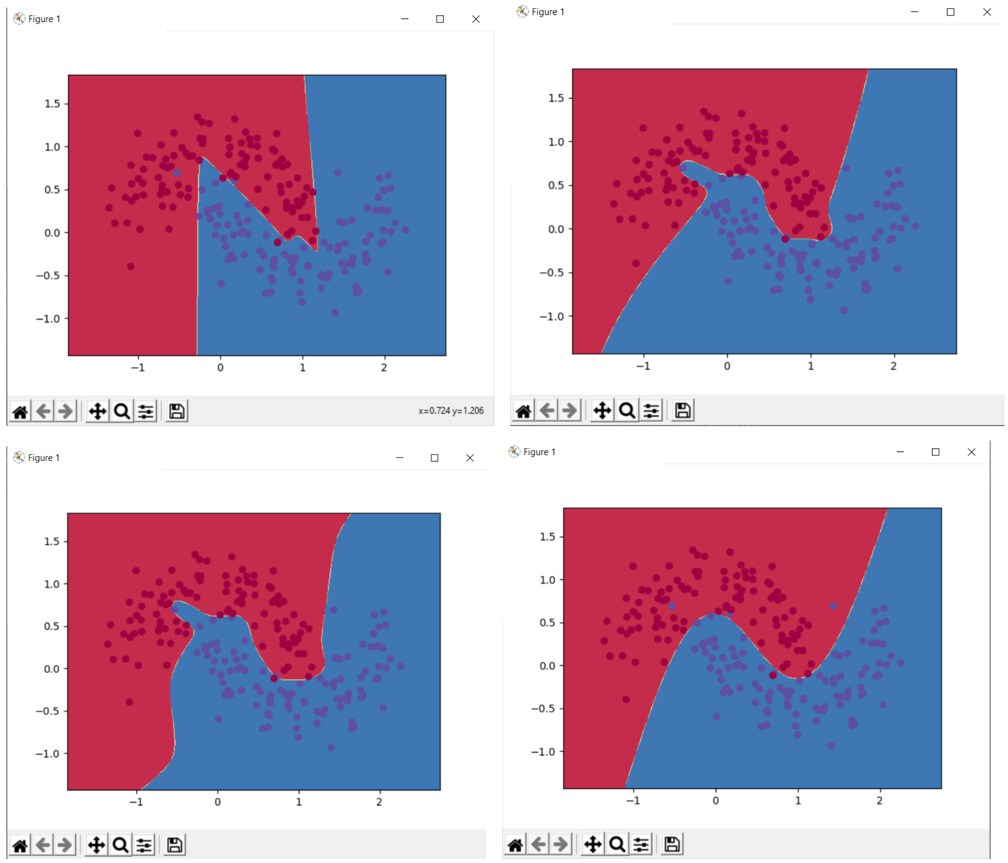






Here we can observe that the sigmoid function has a better fitting than other Tanh and relu functions.

2. Tanh for different hidden layers:



The above 3 figures are plots of Tanh activation function for hidden layers = 5, 10, 20, 6.

We can observe that for hidden layers = 6 the plot is smooth and fits perfectly, while others are under fitted or overfitted.

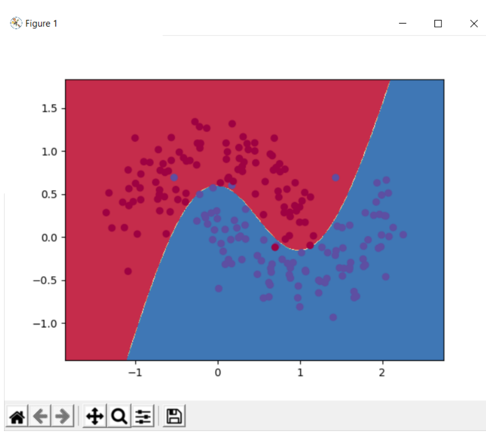
f. Deep Neural Network:

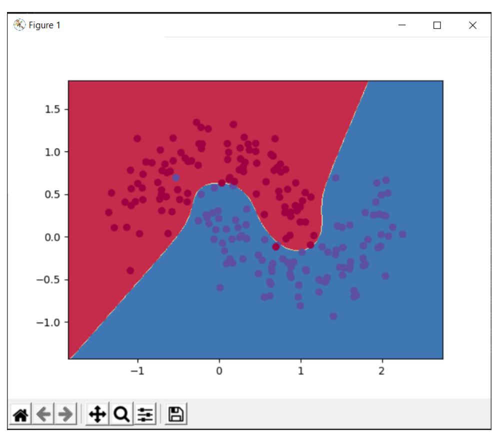
<u>Layer Class:</u>

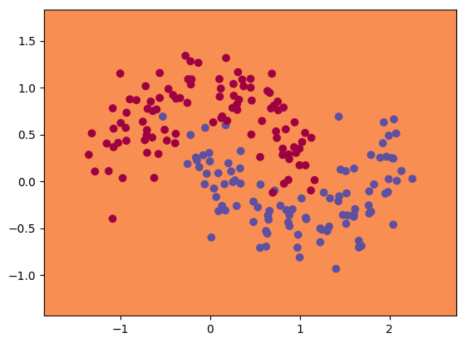
```
import numpy as np
from configs import configration

config = configration()

class Layer():
```

```
def __init__(self, layer_id, actFun):

    self.actFun = actFun
    self.layer_id = layer_id



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

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

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

    return self.a

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

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

        return dW, db
```

<u>Neuralnetwork Class:</u>

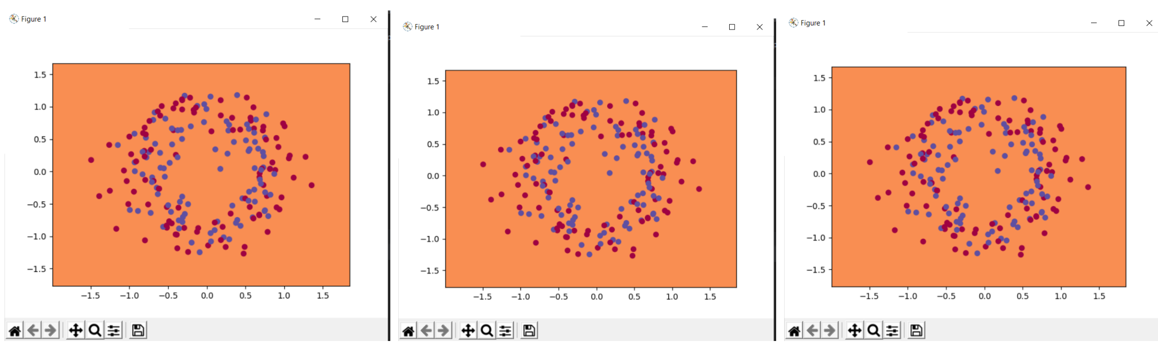






This is for the Make_moons dataset .
Here we can observe that while the tanh and sigmoid act_fun are fine but the relu function is filled with an orange background, this is due to vanishing gradients and strong regularization.

- This is for make_circles dataset



As we can observe for all act_fun i.e. tanh, sigmoid and relu, due to densely packed and nodes being over each other nodes the gradients are vanishing and are very strongly regularized therefore the orange background.

<u>Task - 2 : Deep Convolutional Networks with MNIST</u>

a. <u>Build and train Networks:</u>

- <u>Functions:</u>

1. Weight function:

```python
def weight_variable(shape, initializer=None):
    '''
    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
```

2. Bias function:

```python
def bias_variable(shape, initializer=None):
    '''
    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
```

3. Conv2d function:

```python
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
```

4. Max_pool_2x2:

```python
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
```

- <u>Build your network:</u>

first convolutional layer

W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)

```
h_pool1 = max_pool_2x2(h_conv1)

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

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

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

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

- <u>Set up Training:</u>

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

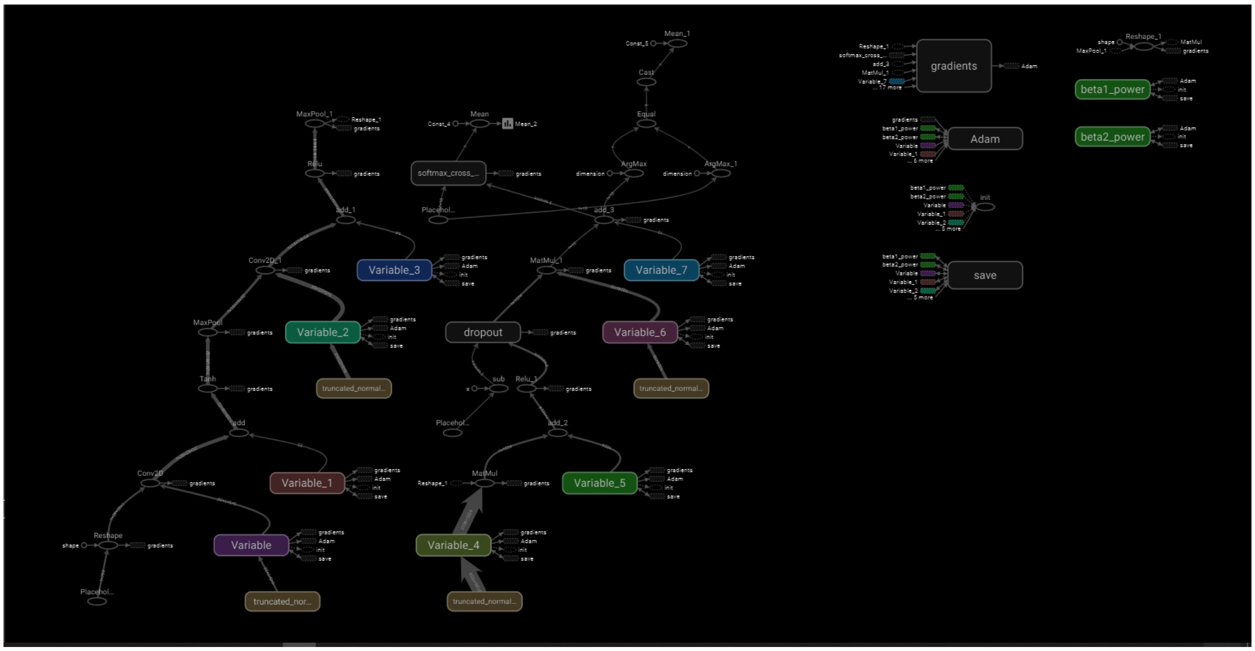
- <u>Run Training:</u>

```
step 0, training accuracy 0.08
step 100, training accuracy 0.04
step 200, training accuracy 0.34
step 300, training accuracy 0.22
step 400, training accuracy 0.64
step 500, training accuracy 0.22
step 600, training accuracy 0.4
step 700, training accuracy 0.62
step 800, training accuracy 0.6
```

step 900, training accuracy 0.6
step 1000, training accuracy 0.64
step 1100, training accuracy 0.66
step 1200, training accuracy 0.72
step 1300, training accuracy 0.72
step 1400, training accuracy 0.72
step 1500, training accuracy 0.74
step 1600, training accuracy 0.7
step 1700, training accuracy 0.84
step 1800, training accuracy 0.82
step 1900, training accuracy 0.78
step 2000, training accuracy 0.7
step 2100, training accuracy 0.82
step 2200, training accuracy 0.84
step 2300, training accuracy 0.74
step 2400, training accuracy 0.84
step 2500, training accuracy 0.86
step 2600, training accuracy 0.76
step 2700, training accuracy 0.78
step 2800, training accuracy 0.7
step 2900, training accuracy 0.82
step 3000, training accuracy 0.72
step 3100, training accuracy 0.86
step 3200, training accuracy 0.78
step 3300, training accuracy 0.92
step 3400, training accuracy 0.78
step 3500, training accuracy 0.84
step 3600, training accuracy 0.8
step 3700, training accuracy 0.86
step 3800, training accuracy 0.86
step 3900, training accuracy 0.82
step 4000, training accuracy 0.86
step 4100, training accuracy 0.8
step 4200, training accuracy 0.86
step 4300, training accuracy 0.9
step 4400, training accuracy 0.82
step 4500, training accuracy 0.86
step 4600, training accuracy 0.82
step 4700, training accuracy 0.86
step 4800, training accuracy 0.88
step 4900, training accuracy 0.88
step 5000, training accuracy 0.92
step 5100, training accuracy 0.94
step 5200, training accuracy 0.94

step 5300, training accuracy 0.92
step 5400, training accuracy 0.88
test accuracy 0.8886
The training takes 383.276304 seconds to finish

- Visualise Training:



b. More on Visualizing:

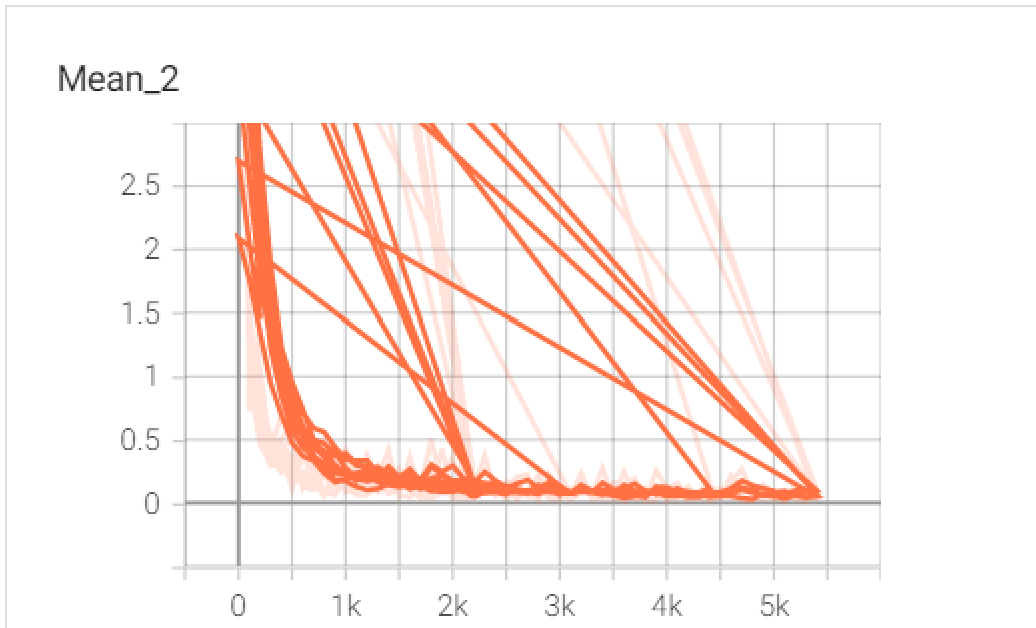
By using ‘tensorboard --logdir=./results/’ we will get the scalar, histogram and graphs for all the summaries we plot for.

```
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)
```

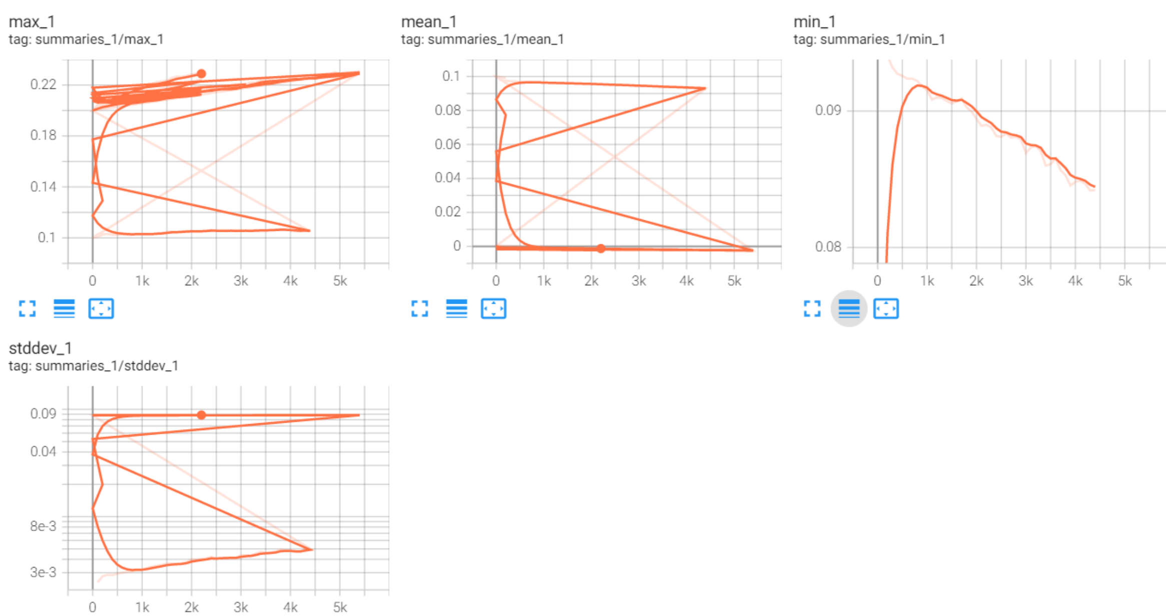
```
variable_summaries(W_conv1)
variable_summaries(W_conv2)
variable_summaries(W_fc1)
variable_summaries(W_fc2)
variable_summaries(b_fc2)
variable_summaries(b_fc1)
variable_summaries(b_conv2)
variable_summaries(b_conv1)
variable_summaries(conv2d(h_pool1, W_conv2) + b_conv2)
variable_summaries(conv2d(x_image,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

The above are the summaries for all the convolution layers, softmax, dropout and densely packed layers.

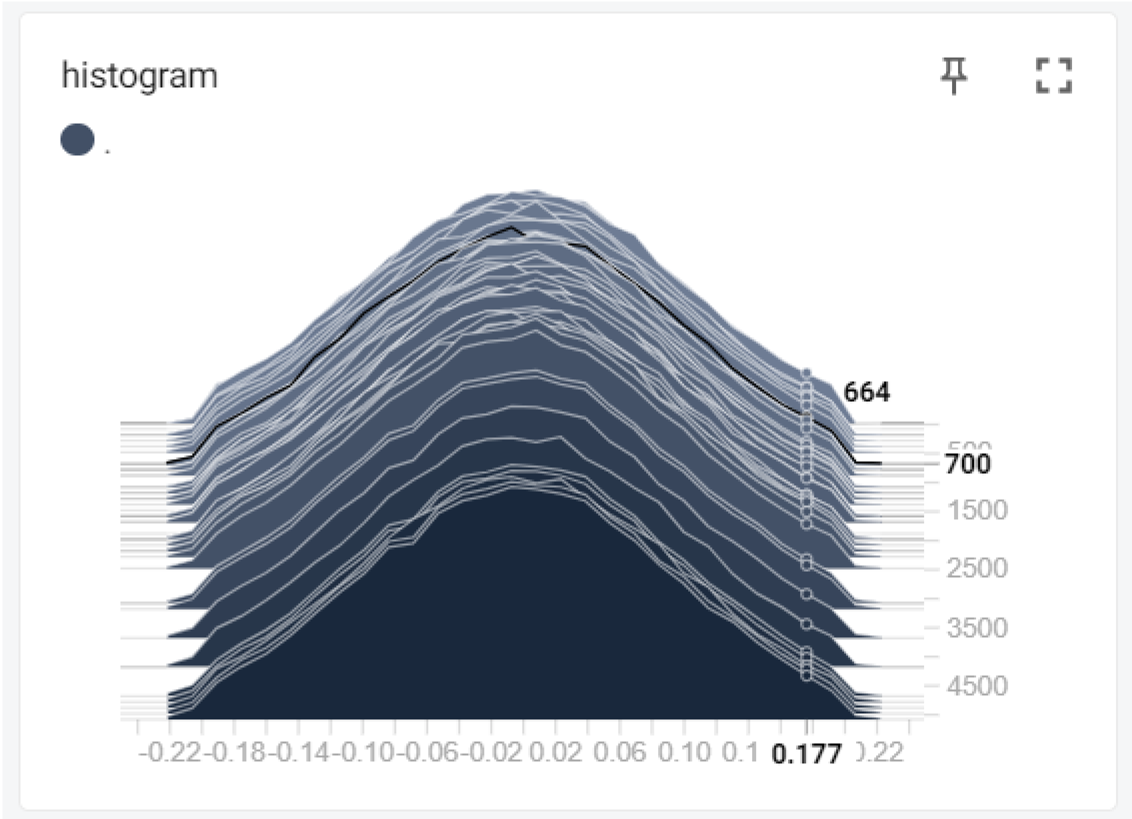
The loss path is given below.



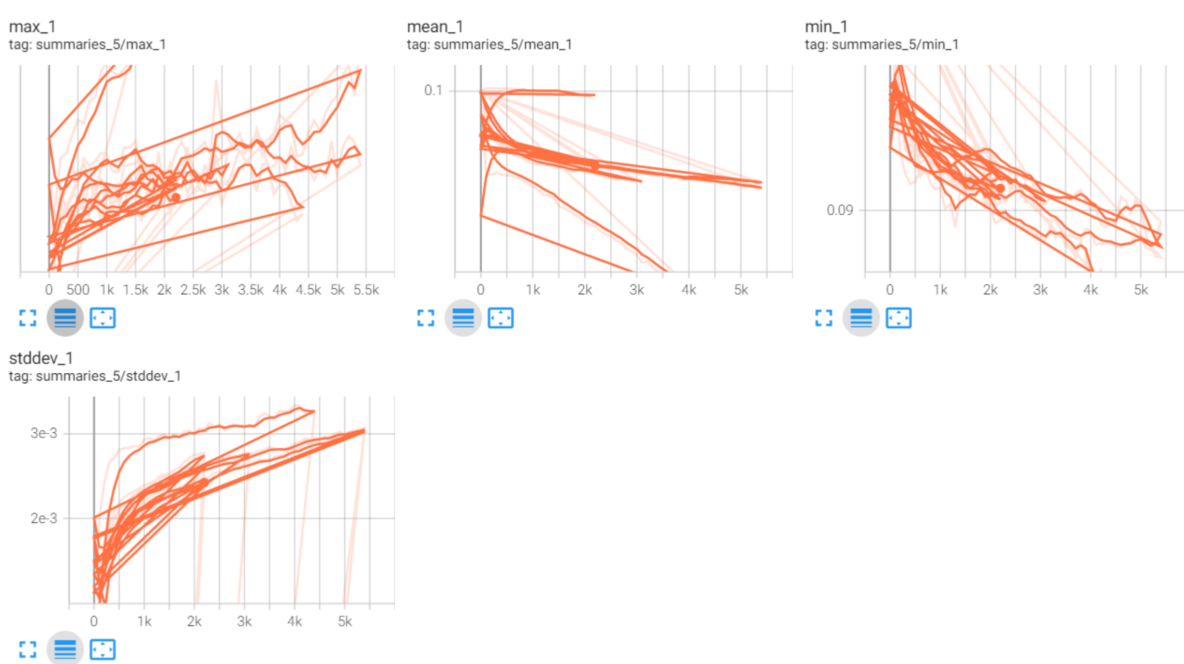
For weight W_conv1:



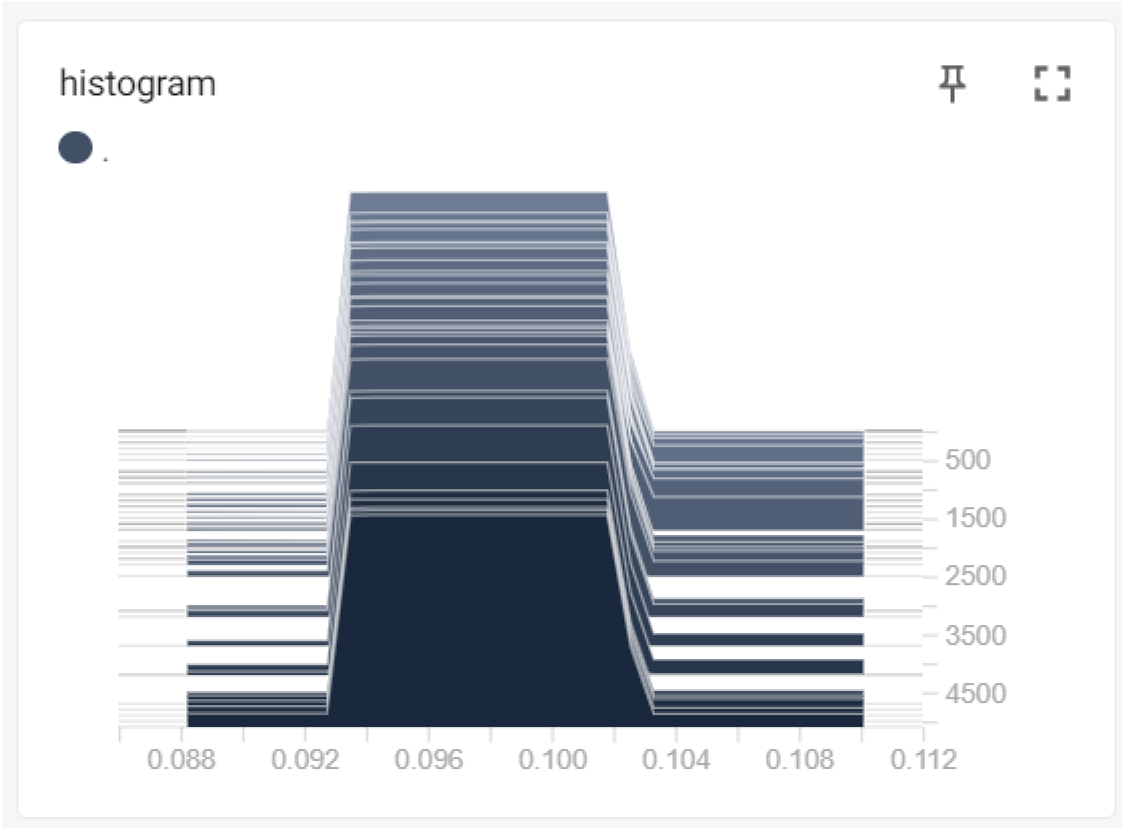
Histogram of weight:



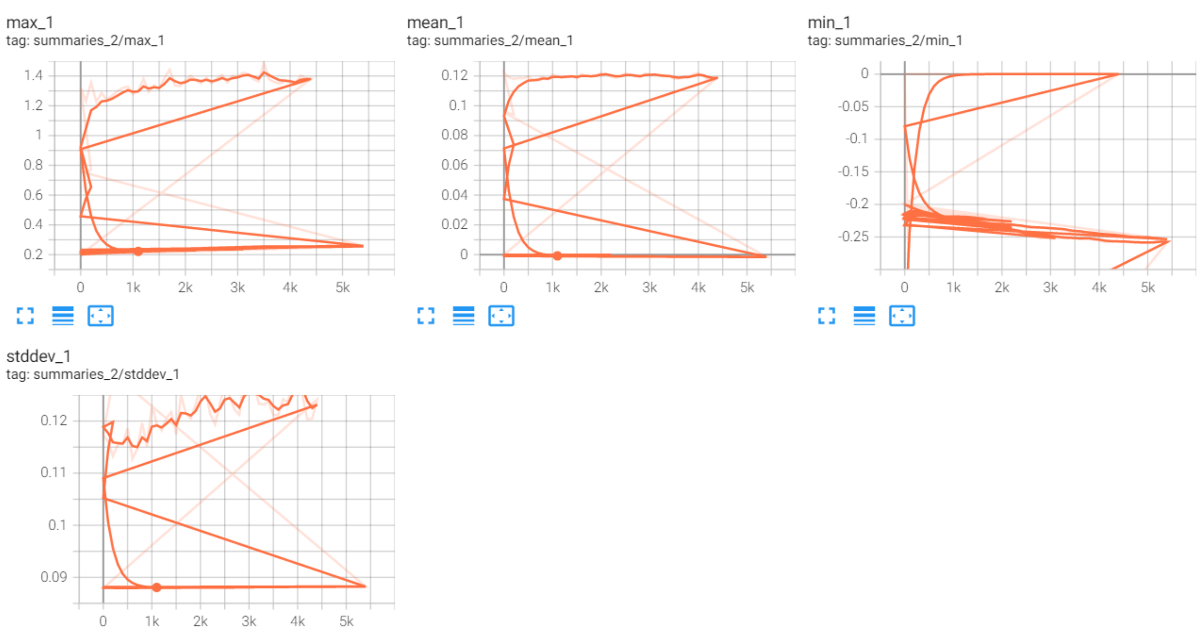
For Bias b_fc1:



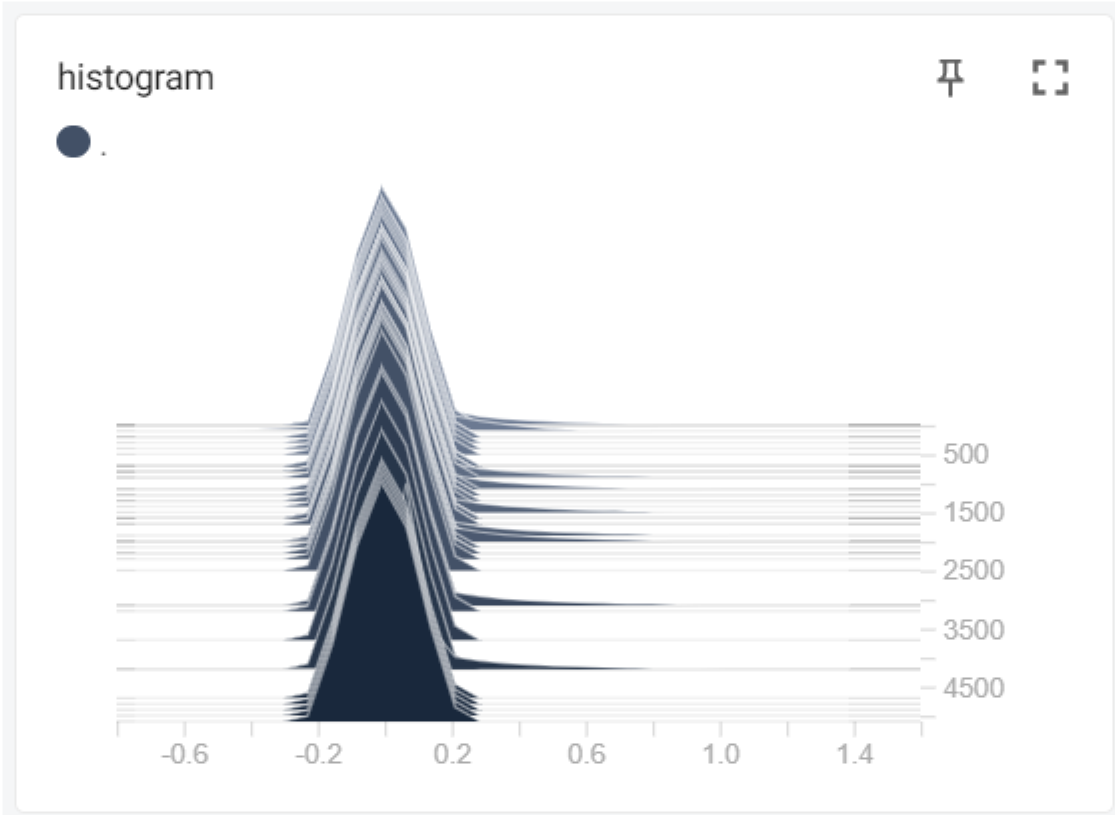
Histogram for Bias:



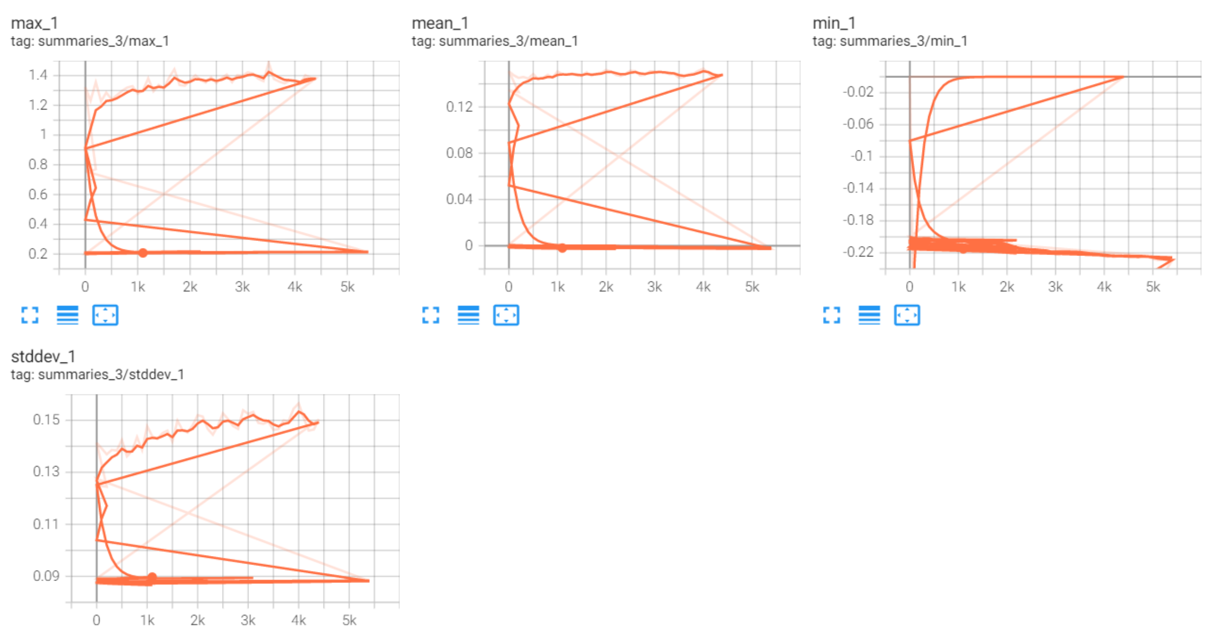
For W_conv2:



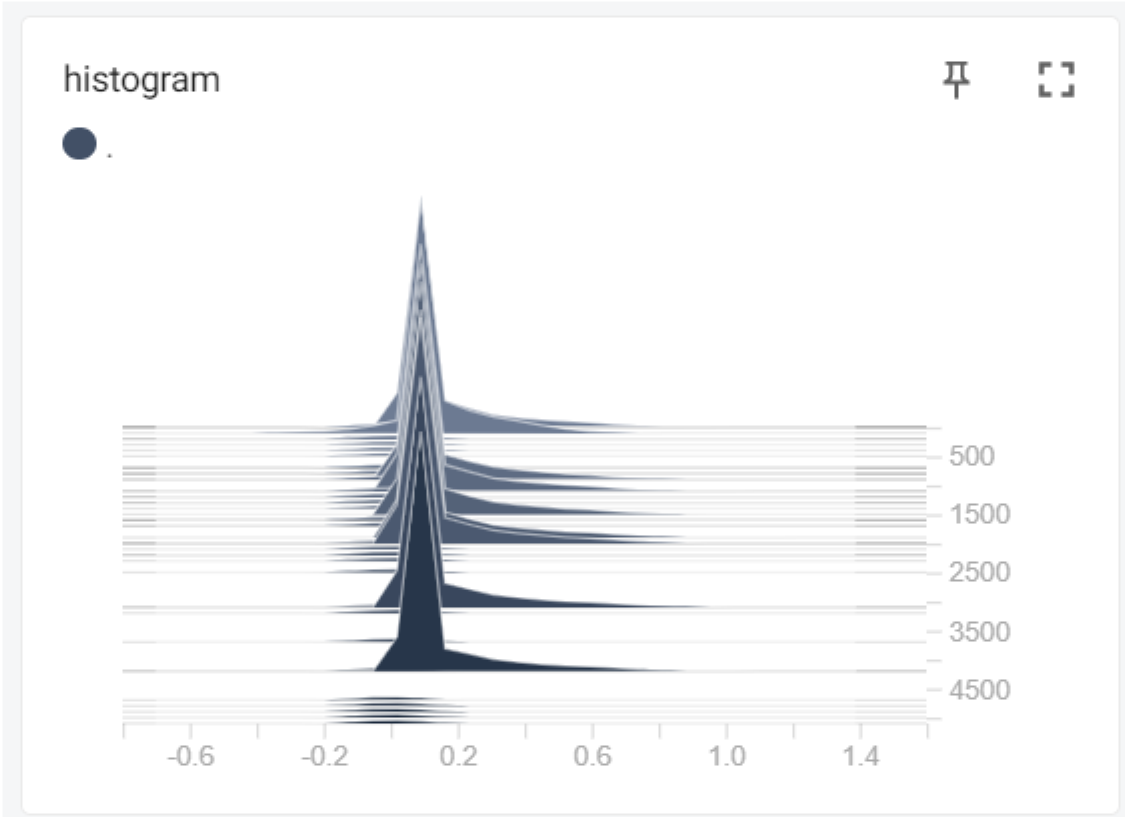
Histogram:



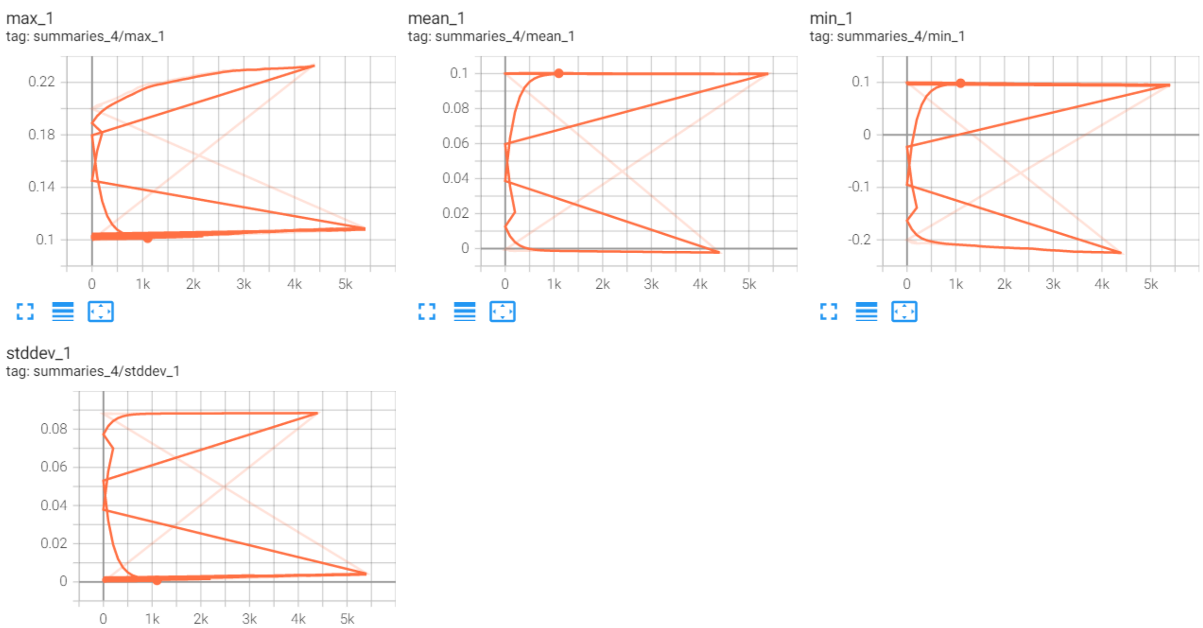
For W_fc1:



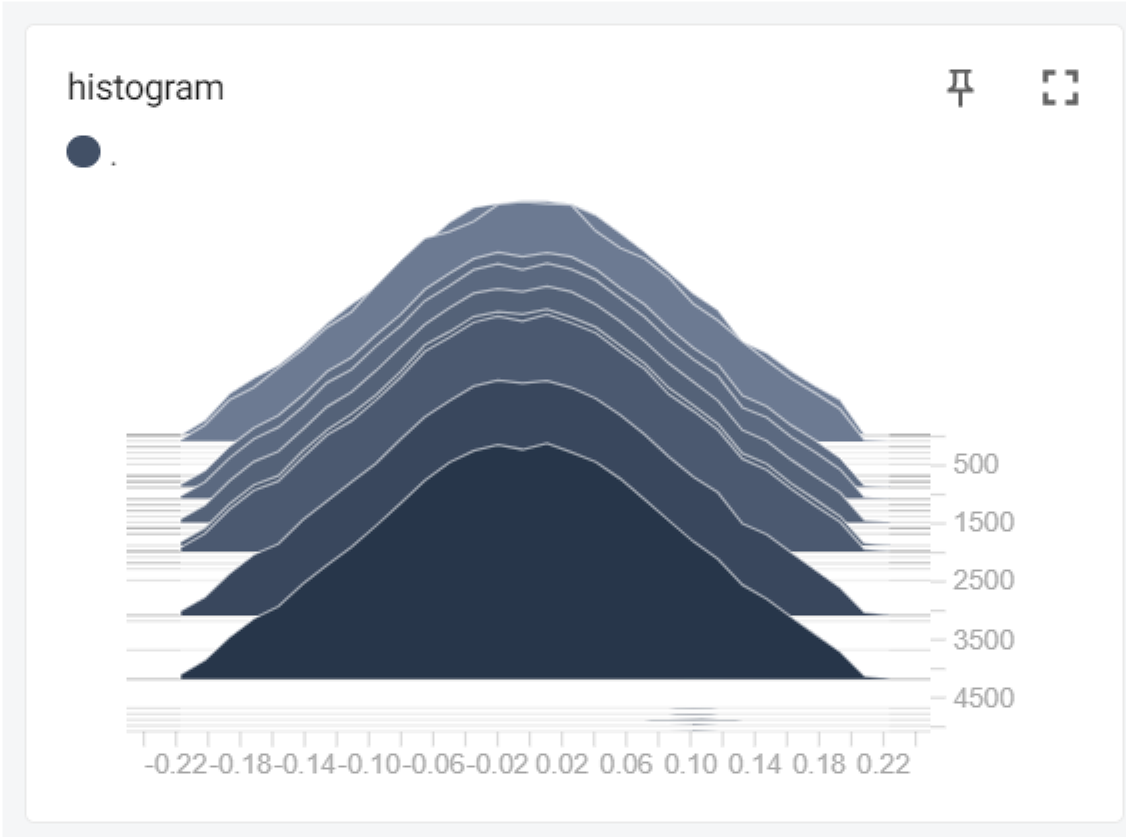
Histogram:



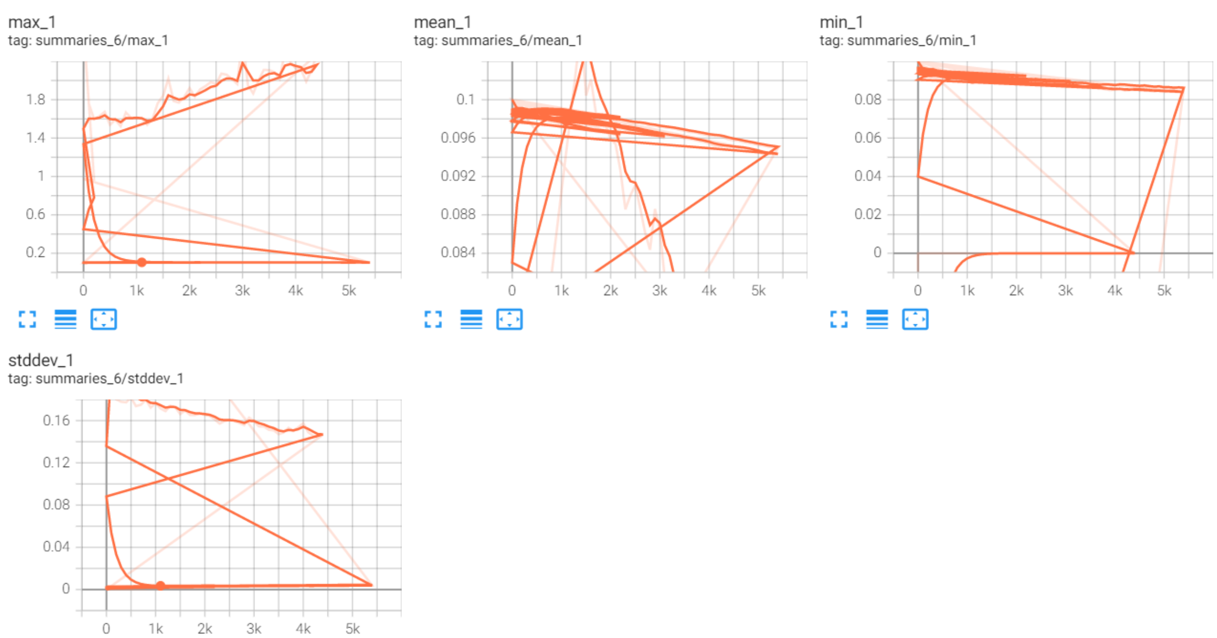
For W_fc2:



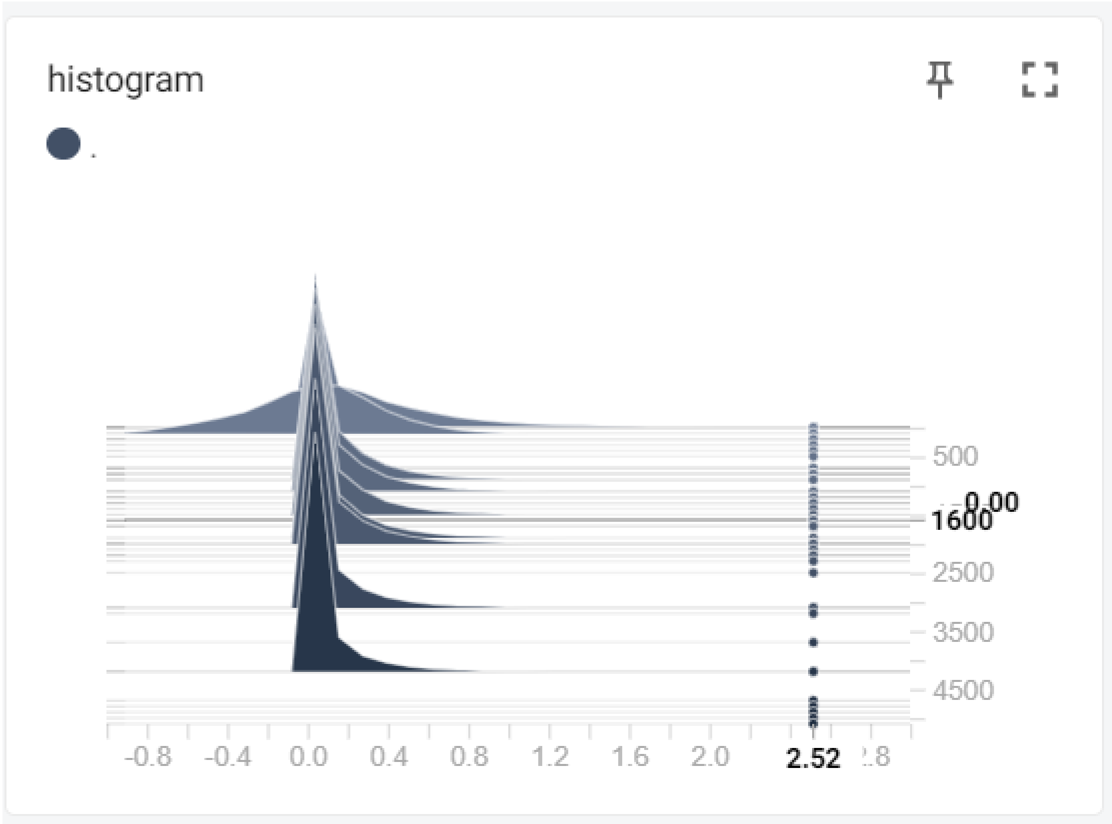
Histogram:



For b_fc2:

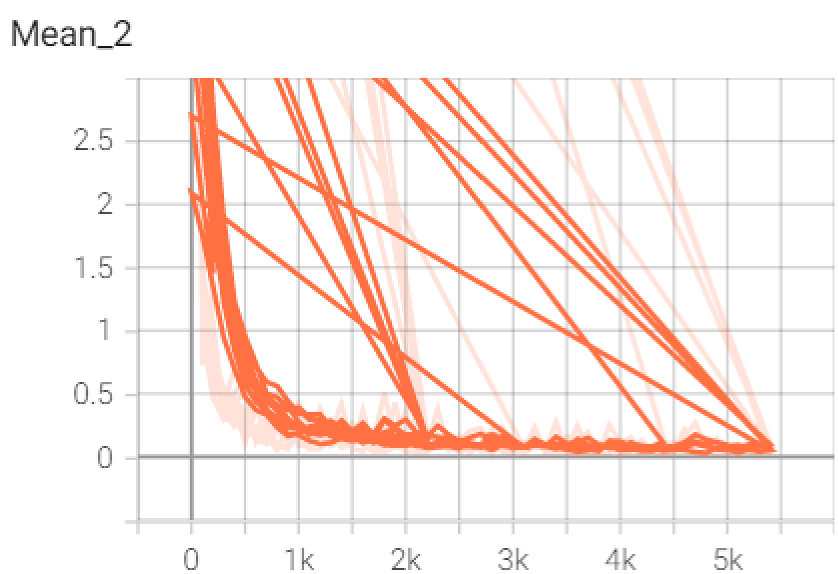


Histogram:



Like the following above images we will get plots of summaries for all the variables we get.

The final training loss for all test cases :



c. Time for more fun:

For Adam optimiser:

step 100, training accuracy 0.82
step 200, training accuracy 0.92
step 300, training accuracy 0.94
step 400, training accuracy 0.94
step 500, training accuracy 0.96
step 600, training accuracy 0.98
step 700, training accuracy 0.94
step 800, training accuracy 0.96
step 900, training accuracy 0.92
step 1000, training accuracy 0.96
step 1100, training accuracy 1
step 1200, training accuracy 0.96
step 1300, training accuracy 0.9
step 1400, training accuracy 0.98
step 1500, training accuracy 1
step 1600, training accuracy 0.96
step 1700, training accuracy 0.98
step 1800, training accuracy 0.98
step 1900, training accuracy 1
step 2000, training accuracy 0.96
step 2100, training accuracy 0.98
step 2200, training accuracy 0.98
step 2300, training accuracy 1
step 2400, training accuracy 0.96
step 2500, training accuracy 0.96
step 2600, training accuracy 0.98
step 2700, training accuracy 0.94
step 2800, training accuracy 0.98
step 2900, training accuracy 0.98
step 3000, training accuracy 0.98
step 3100, training accuracy 0.98
step 3200, training accuracy 0.94
step 3300, training accuracy 0.96
step 3400, training accuracy 0.98
step 3500, training accuracy 0.98
step 3600, training accuracy 1
step 3700, training accuracy 0.98
step 3800, training accuracy 1
step 3900, training accuracy 1
step 4000, training accuracy 1
step 4100, training accuracy 1
step 4200, training accuracy 0.94
step 4300, training accuracy 1

step 4400, training accuracy 0.98
step 4500, training accuracy 0.98
step 4600, training accuracy 1
step 4700, training accuracy 1
step 4800, training accuracy 1
step 4900, training accuracy 0.98
step 5000, training accuracy 0.98
step 5100, training accuracy 1
step 5200, training accuracy 1
step 5300, training accuracy 1
step 5400, training accuracy 1
test accuracy 0.9889

For Gradient Optimiser:

step 0, training accuracy 0.08
step 100, training accuracy 0.04
step 200, training accuracy 0.34
step 300, training accuracy 0.22
step 400, training accuracy 0.64
step 500, training accuracy 0.22
step 600, training accuracy 0.4
step 700, training accuracy 0.62
step 800, training accuracy 0.6
step 900, training accuracy 0.6
step 1000, training accuracy 0.64
step 1100, training accuracy 0.66
step 1200, training accuracy 0.72
step 1300, training accuracy 0.72
step 1400, training accuracy 0.72
step 1500, training accuracy 0.74
step 1600, training accuracy 0.7
step 1700, training accuracy 0.84
step 1800, training accuracy 0.82
step 1900, training accuracy 0.78
step 2000, training accuracy 0.7
step 2100, training accuracy 0.82
step 2200, training accuracy 0.84
step 2300, training accuracy 0.74
step 2400, training accuracy 0.84
step 2500, training accuracy 0.86
step 2600, training accuracy 0.76
step 2700, training accuracy 0.78
step 2800, training accuracy 0.7
step 2900, training accuracy 0.82

step 3000, training accuracy 0.72
step 3100, training accuracy 0.86
step 3200, training accuracy 0.78
step 3300, training accuracy 0.92
step 3400, training accuracy 0.78
step 3500, training accuracy 0.84
step 3600, training accuracy 0.8
step 3700, training accuracy 0.86
step 3800, training accuracy 0.86
step 3900, training accuracy 0.82
step 4000, training accuracy 0.86
step 4100, training accuracy 0.8
step 4200, training accuracy 0.86
step 4300, training accuracy 0.9
step 4400, training accuracy 0.82
step 4500, training accuracy 0.86
step 4600, training accuracy 0.82
step 4700, training accuracy 0.86
step 4800, training accuracy 0.88
step 4900, training accuracy 0.88
step 5000, training accuracy 0.92
step 5100, training accuracy 0.94
step 5200, training accuracy 0.94
step 5300, training accuracy 0.92
step 5400, training accuracy 0.88
test accuracy 0.8886

For Tanh:
step 0, training accuracy 0.06
step 100, training accuracy 0.4
step 200, training accuracy 0.7
step 300, training accuracy 0.86
step 400, training accuracy 0.84
step 500, training accuracy 0.94
step 600, training accuracy 0.94
step 700, training accuracy 0.98
step 800, training accuracy 0.94
step 900, training accuracy 0.96
step 1000, training accuracy 0.96
step 1100, training accuracy 0.9
step 1200, training accuracy 0.98
step 1300, training accuracy 0.98
step 1400, training accuracy 0.94
step 1500, training accuracy 0.98

step 1600, training accuracy 0.96
step 1700, training accuracy 0.98
step 1800, training accuracy 0.94
step 1900, training accuracy 0.98
step 2000, training accuracy 1
step 2100, training accuracy 0.94
step 2200, training accuracy 0.96
step 2300, training accuracy 1
step 2400, training accuracy 0.96
step 2500, training accuracy 0.92
step 2600, training accuracy 0.96
step 2700, training accuracy 1
step 2800, training accuracy 0.98
step 2900, training accuracy 1
step 3000, training accuracy 0.96
step 3100, training accuracy 1
step 3200, training accuracy 0.98
step 3300, training accuracy 0.98
step 3400, training accuracy 0.96
step 3500, training accuracy 0.96
step 3600, training accuracy 1
step 3700, training accuracy 0.98
step 3800, training accuracy 1
step 3900, training accuracy 0.98
step 4000, training accuracy 0.98
step 4100, training accuracy 0.98
step 4200, training accuracy 1
step 4300, training accuracy 0.96
step 4400, training accuracy 1
step 4500, training accuracy 1
step 4600, training accuracy 0.98
step 4700, training accuracy 1
step 4800, training accuracy 1
step 4900, training accuracy 1
step 5000, training accuracy 0.98
step 5100, training accuracy 0.98
step 5200, training accuracy 0.96
step 5300, training accuracy 1
step 5400, training accuracy 0.98
test accuracy 0.9846

From the above results we can observe that Tanh nonlinearity, and Adam optimiser have higher test accuracy than gradient optimiser and relu.

The below is the test accuracy plot: for Gradient optimiser, Tanh and xavier initilizer.