Assignment1, COMP 576

Yikun Li (yl212)

September 29, 2021

Task 1.a

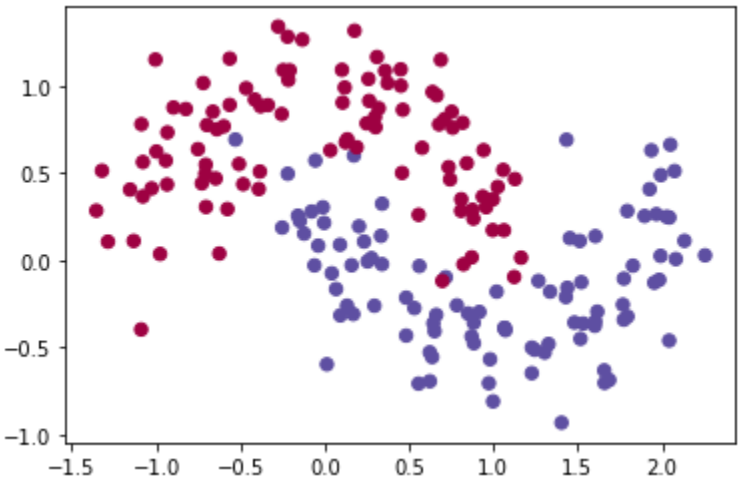


Figure 1_1: generate_data

1.b.1

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

    # YOU IMPLMENT YOUR actFun HERE
    if type == "tanh":
        return np.tanh(z)
    elif type == "sigmoid":
        return 1 / (1 + np.exp(-z))
    elif type == "relu":
        return np.maximum(0, z)
    else:
        return None
```

1.b.2

Derive the derivatives of Tanh, Sigmoid and ReLU:

$$\frac{d}{dx}tanh(x) = 1 - tanh^2(x)$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$\frac{dx}{x}relu(x) = x > 0?1:0$$

1.b.3

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

    # YOU IMPLEMENT YOUR diff_actFun HERE
    if type == 'tanh':
        return 1 - np.square(np.tanh(z))
    elif type == 'sigmoid':
        tmp = 1 / (1 + np.exp(-z))
        return tmp * (1 - tmp)
    elif type == 'relu':
        return np.where(z > 0, 1, 0)
    else:
        return None
```

1.c.1

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

    # YOU IMPLEMENT YOUR feedforward HERE

    self.z1 = np.dot(X, self.W1) + self.b1
    self.a1 = actFun(self.z1)
    self.z2 = np.dot(self.a1, self.W2) + self.b2
    exp_scores = np.exp(self.z2)
    self.probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    return None
```

1.c.2

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

    # YOU IMPLEMENT YOUR CALCULATION OF THE LOSS HERE

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

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

1.d.1

$$\Delta_3 = \frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2}$$

$$\frac{\partial L}{\partial W_2} = a_1^T \Delta_3$$

$$\frac{\partial L}{\partial b_2} = \sum \Delta_3$$

$$\Delta_2 = diff * \Delta_3 W_2^T$$

$$\frac{\partial L}{\partial W_1} = X^T \Delta_2$$

$$\frac{\partial L}{\partial b_2} = \sum \Delta_2$$

1.d.2

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

1.e.1: Change Type

Tanh:

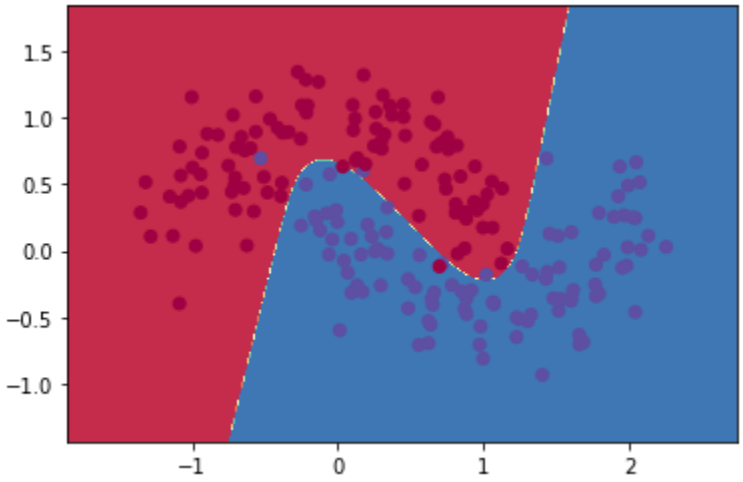


Figure 1_2: Tanh

Sigmoid:

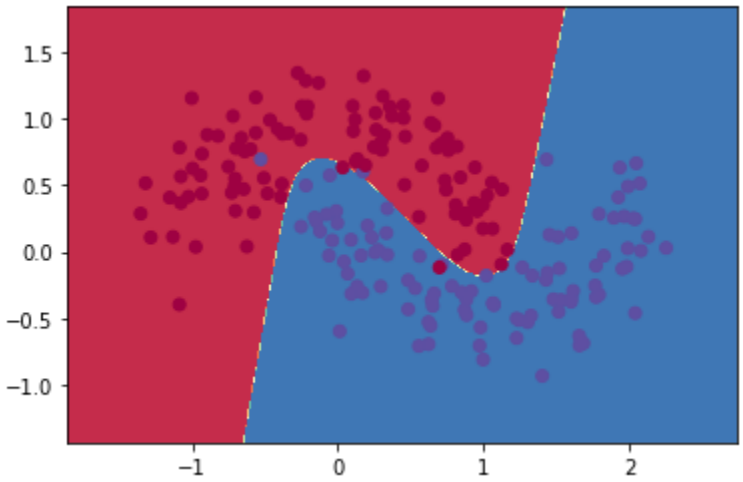


Figure 1_3: Sigmoid

ReLu:

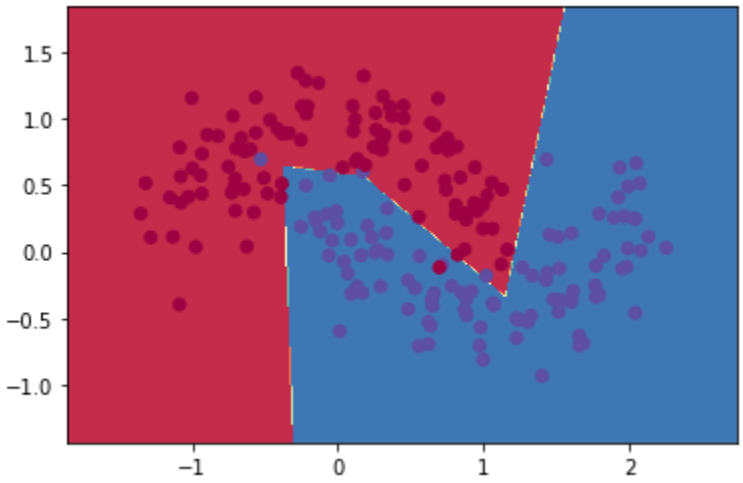


Figure 1_4: ReLu

Observation and explanation: ReLu performs better than Tanh and Sigmoid, the latter two performing very similarly. The reason for this situation is that the ReLu can make some of the output to be zero which can lead to Scale-invariant.

1.e.2: Change nn_hidden_dim

Type = Tanh, nn_hidden_dim = 1

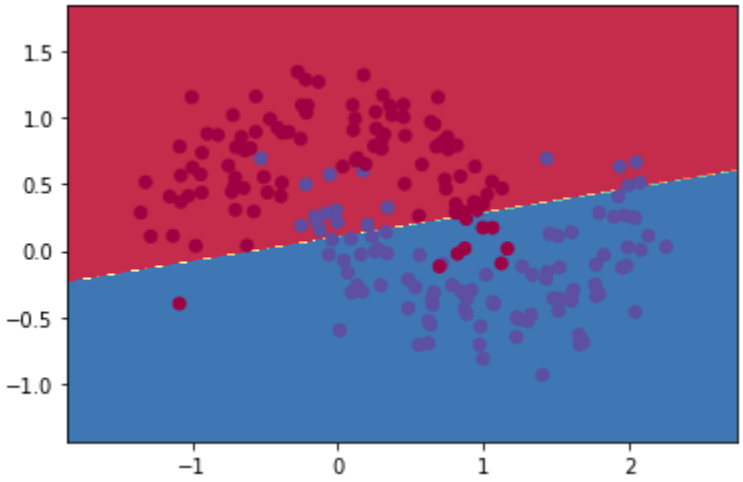


Figure 1_5: nn_hidden_dim = 1

Type = Tanh, nn_hidden_dim = 3

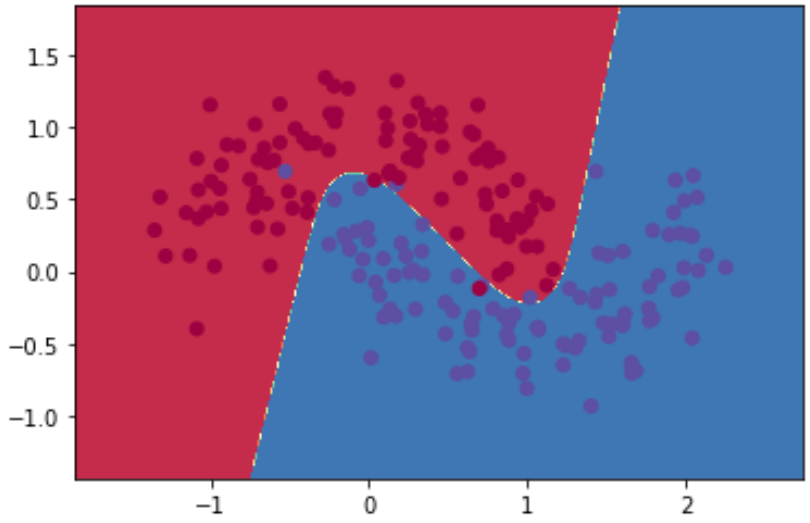


Figure 1_6: nn_hidden_dim = 3

Type = Tanh, nn_hidden_dim = 5

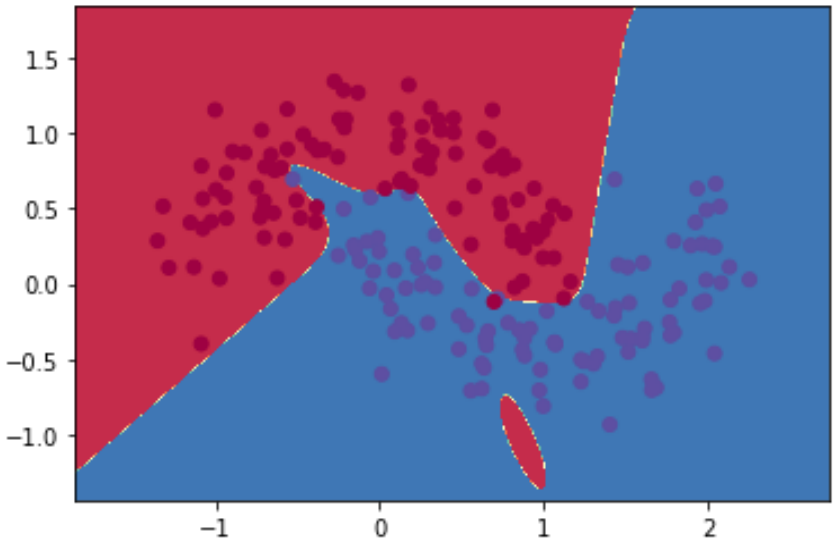


Figure 1_7: nn_hidden_dim = 5

Type = Tanh, nn_hidden_dim = 7

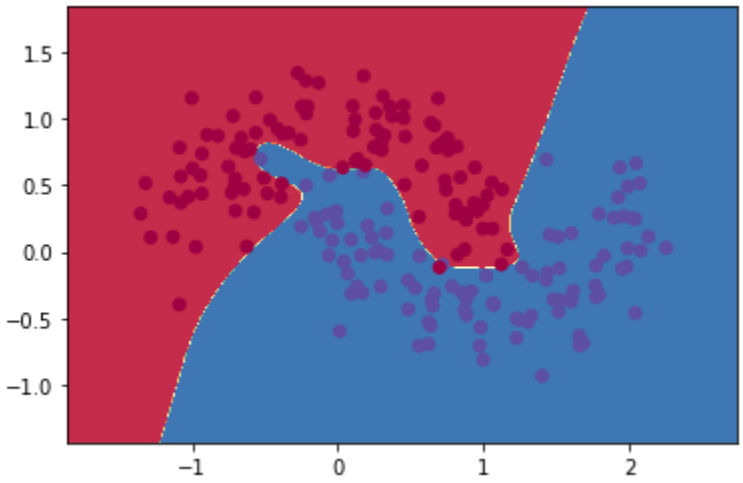


Figure 1_8: nn_hidden_dim = 7

Type = Tanh, nn_hidden_dim = 9

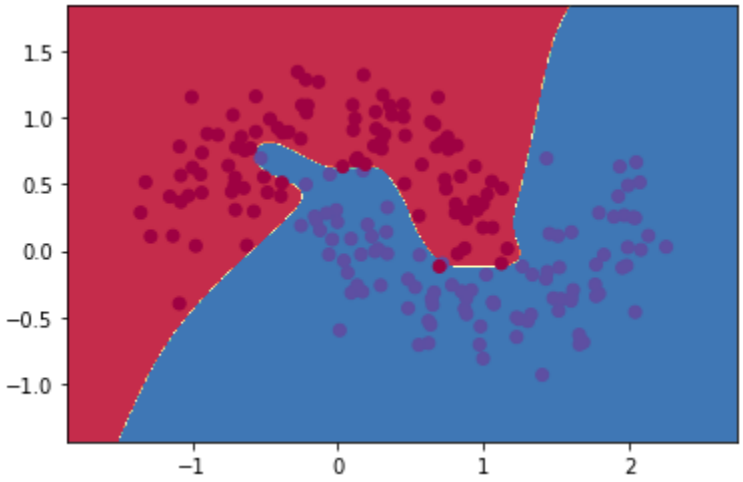


Figure 1_9: nn_hidden_dim = 9

Observation: When the nn_hidden_dim = 3, the performance is the best. Because if the value is too small, the model will unfit the data; if the value is too big, the model will overfit the data.

1.f : DeepNeuralNetwork implementation

In my implementation, call the DeepNeuralNetwork this:

```
model = DeepNeuralNetwork(nn_dims=[2, 3, 2], actFun_type='tanh')
```

Which means the model is constructed by 3 layers with 2, 3, 2 units on each layer from input to output, and the active function is tanh.

Here are some detailed implementation on key functions:

Function feedforward:

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

Function calculate_loss:

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

    # YOU IMPLEMENT YOUR CALCULATION OF THE LOSS HERE

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

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

Function backprop:

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

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

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

    return dW, db
```

Function fit_model:

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

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

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

        # Optionally print the loss.
        # This is expensive because it uses the whole dataset, so we don't want to do it too often.
        if print_loss and i % 1000 == 0:
            print("Loss after iteration %i: %f" % (i, self.calculate_loss(X, y)))
```

1.f: Change number of layers

In this section, let's keep the dataset to be make_moons and the active function to be tanh. The change part is the number of layers and each time we add a three-unit layer in the network. Let's observe the difference on the boundary of images.

When we set the layer as:

```
model = DeepNeuralNetwork(nn_dims=[2, 3, 2], actFun_type='tanh')
```

The result is:

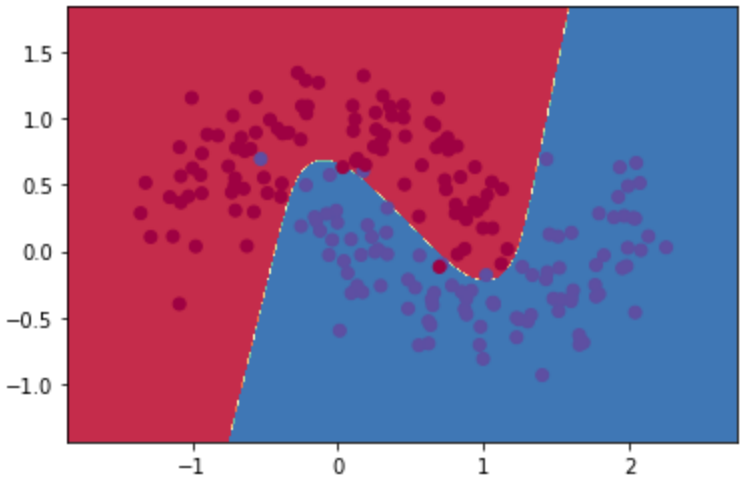


Figure 1_10: nn_hidden_dim = 3

When we set the layer as:

```
model = DeepNeuralNetwork(nn_dims=[2, 3, 3, 2], actFun_type='tanh')
```

The result is:

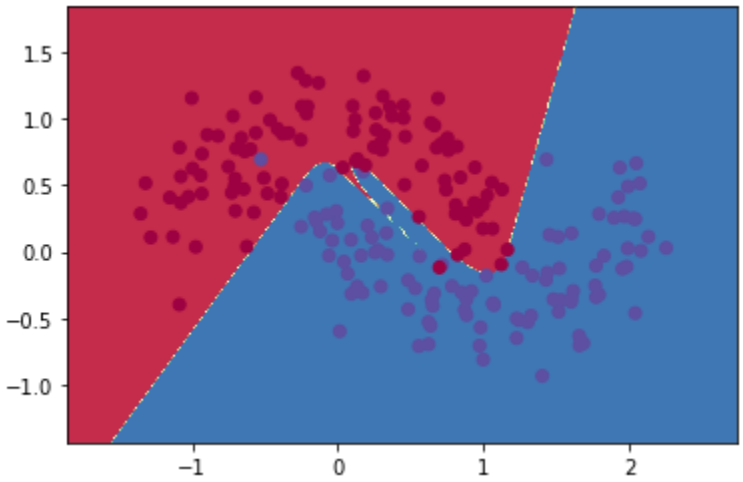


Figure 1_11: nn_hidden_dim = 4

When we set the layer as:

```
model = DeepNeuralNetwork(nn_dims=[2, 3, 3, 3, 2], actFun_type='tanh')
```

The result is:

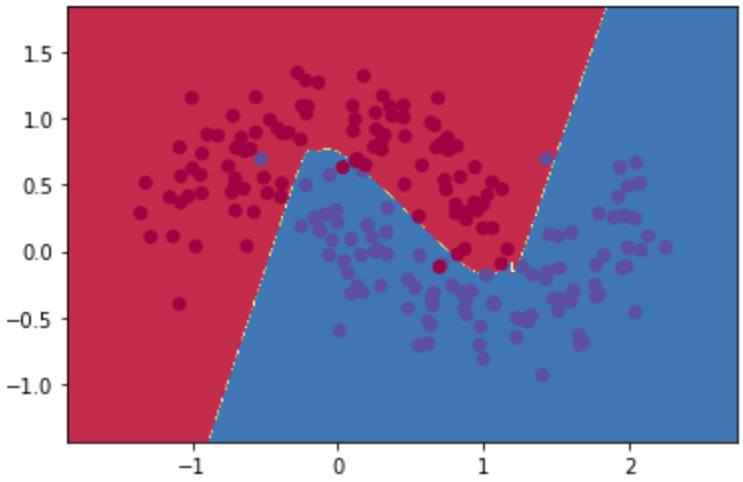


Figure 1_12: nn_hidden_dim = 5

Observation: When the number of layers is 3 or 5, the performance is better than the number of layers to be 4. The layers of the network go deeper, the performance may be better, although this is not a must.

1.f: Change dataset

When the dataset is changed to be make_circles:

```
X, y = datasets.make_circles(
    n_samples=100, shuffle=True, noise=None, random_state=None, factor=0.8)
```

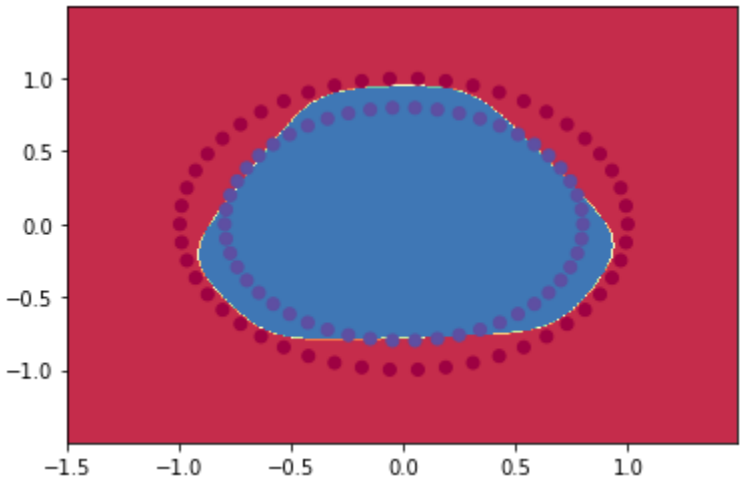


Figure 1_13: dataset=make_circles

This dataset is made up of data points in circles and are classified into two classes, with each of them making up a circle. As the boundary shows in the above image, our three-layer network can classify it well.

Task 2.a.2

Reference from the tutorial GitHub repo of tensorflow:
https://gist.github.com/saitodev/c4c7a8c83f5aa4a00e93084dd3f848c5

```python
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)
    return tf.Variable(initial)

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)
    return tf.Variable(initial)
```

```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
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

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

2.a.3

Reference from the tutorial GitHub repo of tensorflow:
https://gist.github.com/saitodev/c4c7a8c83f5aa4a00e93084dd3f848c5

```
# 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_')

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

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

# 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)
variable_summaries(W_conv2)
variable_summaries(b_conv2)
variable_summaries(h_conv2)
variable_summaries(h_pool2)

# 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)
variable_summaries(W_fc1)
variable_summaries(b_fc1)
variable_summaries(h_pool2_flat)
variable_summaries(h_fc1)
```

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

# 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_conv')
variable_summaries(W_fc2)
variable_summaries(b_fc2)
variable_summaries(y_conv)
```

2.a.4

Reference from the tutorial GitHub repo of tensorflow:
https://gist.github.com/saitodev/c4c7a8c83f5aa4a00e93084dd3f848c5

```
# 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')
```

2.a.5

```python
# 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)

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

# 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))
```

Here is the final output results:

step 0, training accuracy 0.08
step 100, training accuracy 0.9
step 200, training accuracy 0.96
step 300, training accuracy 0.96
step 400, training accuracy 0.84
step 500, training accuracy 0.92
step 600, training accuracy 0.94
step 700, training accuracy 0.9
step 800, training accuracy 0.94
step 900, training accuracy 0.98
step 1000, training accuracy 0.98
step 1100, training accuracy 0.96
step 1200, training accuracy 0.96

step 1300, training accuracy 1
step 1400, training accuracy 0.96
step 1500, training accuracy 0.98
step 1600, training accuracy 0.96
step 1700, training accuracy 0.96
step 1800, training accuracy 0.98
step 1900, training accuracy 0.96
step 2000, training accuracy 0.98
step 2100, training accuracy 0.98
step 2200, training accuracy 1
step 2300, training accuracy 1
step 2400, training accuracy 0.94
step 2500, training accuracy 0.96
step 2600, training accuracy 1
step 2700, training accuracy 0.98
step 2800, training accuracy 1
step 2900, training accuracy 0.96
step 3000, training accuracy 1
step 3100, training accuracy 1
step 3200, training accuracy 0.98
step 3300, training accuracy 0.96
step 3400, training accuracy 1
step 3500, training accuracy 0.98
step 3600, training accuracy 1
step 3700, training accuracy 1
step 3800, training accuracy 1
step 3900, training accuracy 0.96
step 4000, training accuracy 1
step 4100, training accuracy 1
step 4200, training accuracy 1
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 0.98
step 4900, training accuracy 1
step 5000, training accuracy 0.98
step 5100, training accuracy 0.96
step 5200, training accuracy 0.98

step 5300, training accuracy 0.98
step 5400, training accuracy 1
test accuracy 0.9864
The training takes 812.483530 second to finish

2.a.6

There are three tags in the TensorBoard: scalars, graphs, and projector.
The figure of scalars:

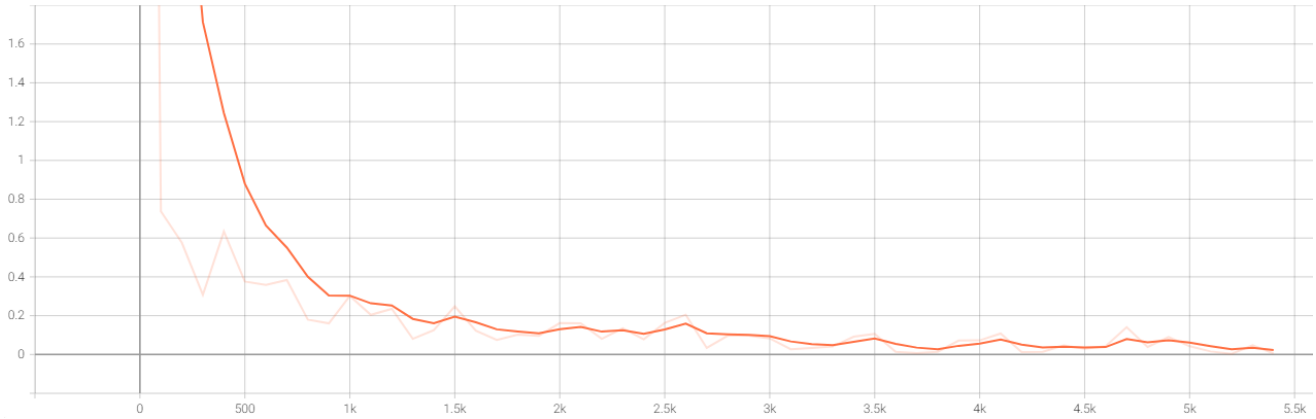


Figure 2_1: scalars

The figure of graphs:

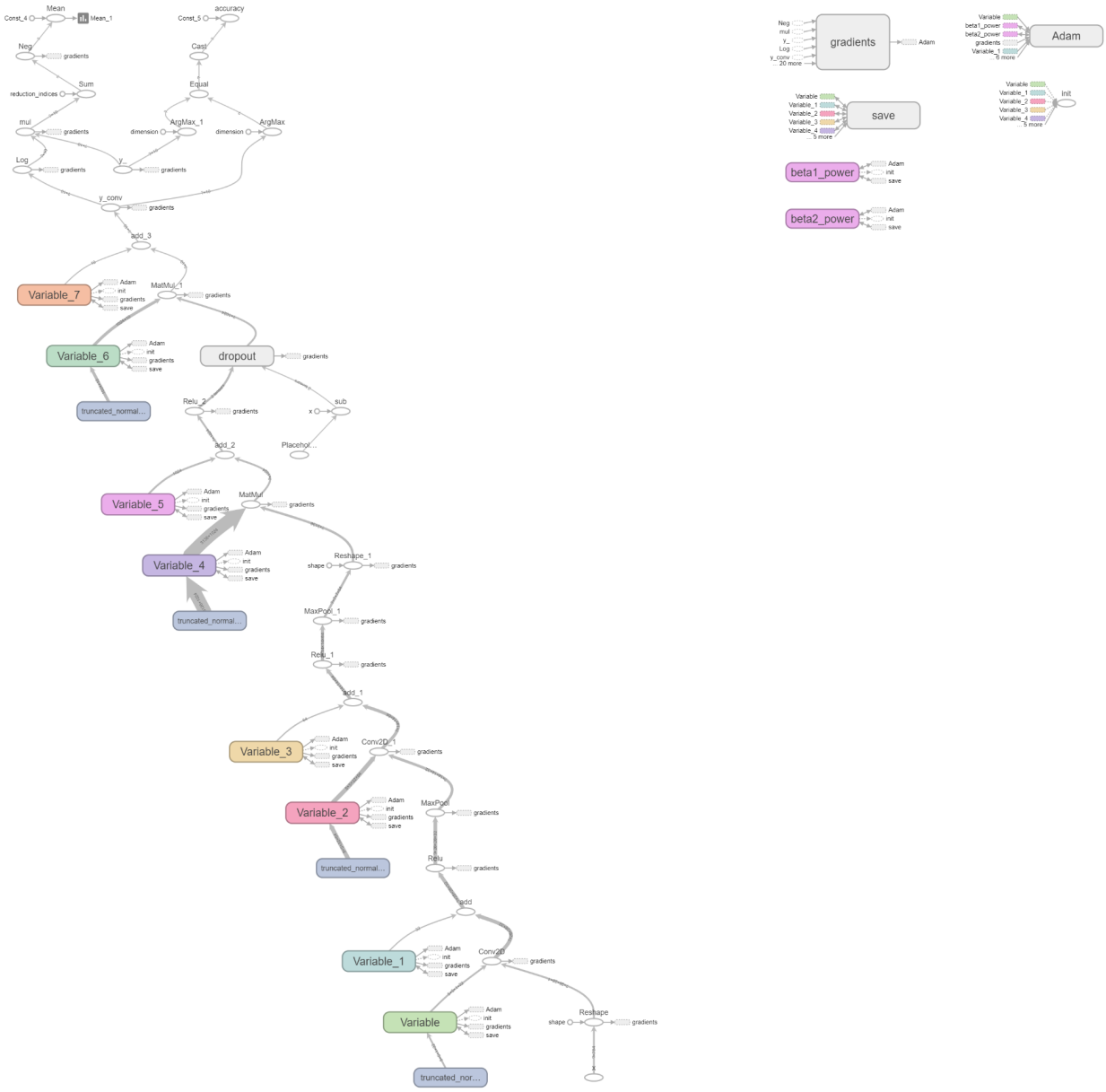


Figure 2_2: graphs

The figure of projector:

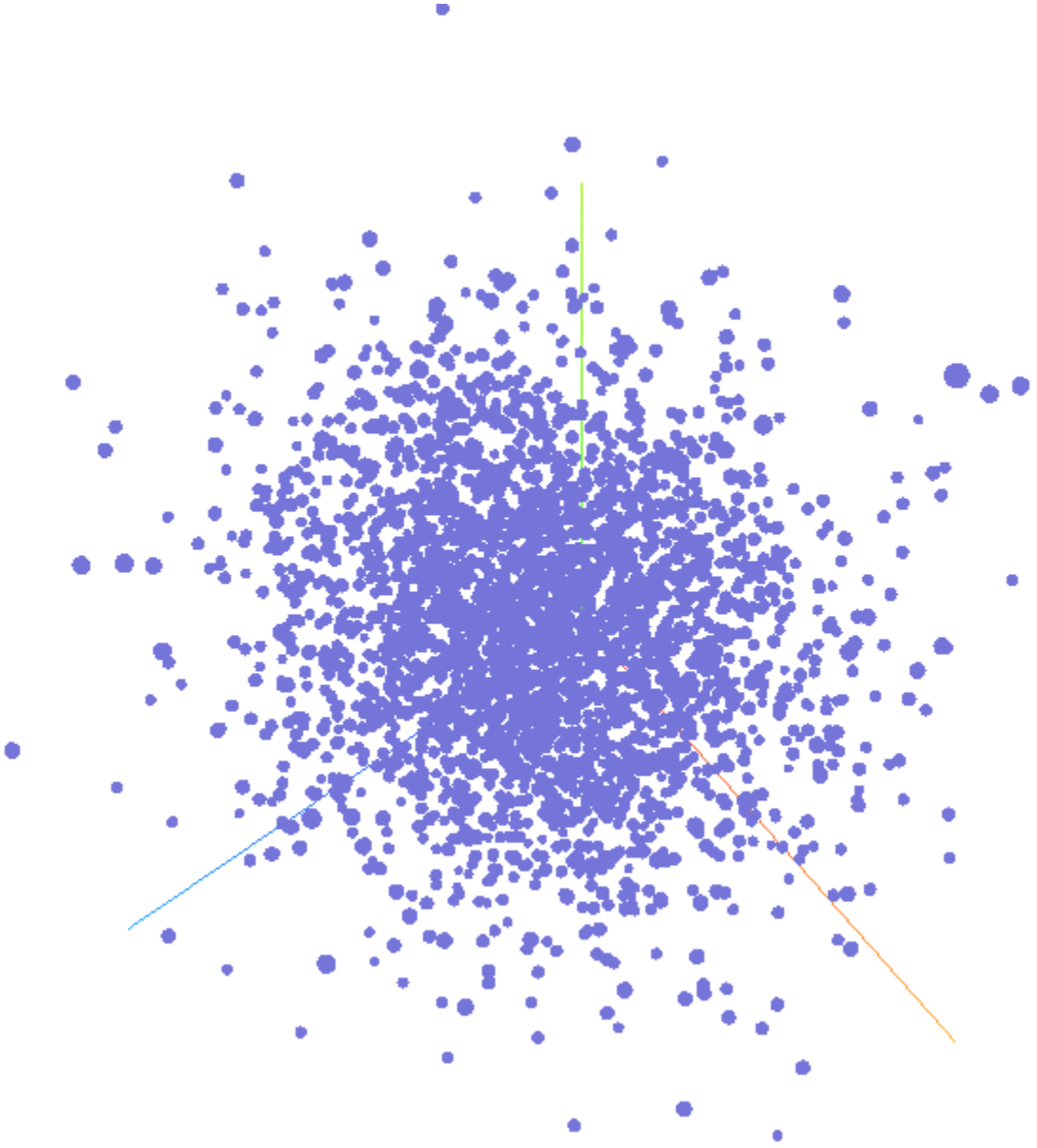


Figure 2_3: projectors

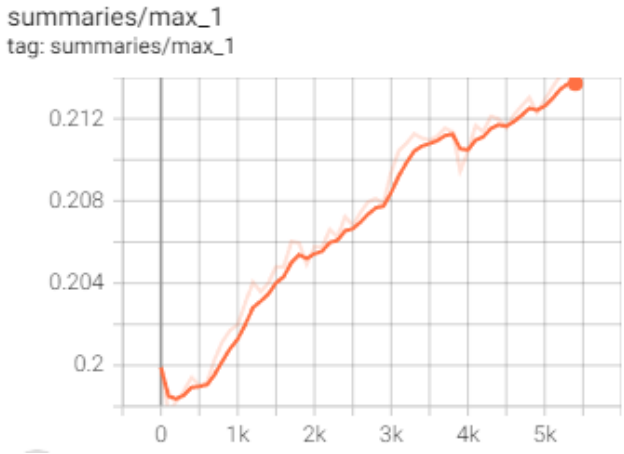
2.b

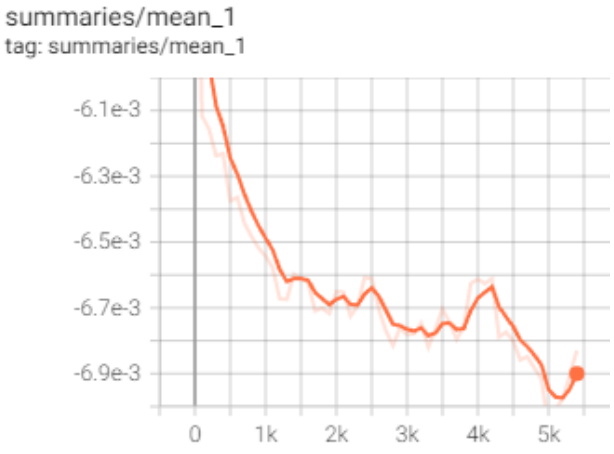
Write function variable_summaries for plotting variables:

```python
def variable_summaries(var):
  """
  Attach a lot of summaries to a Tensor (for TensorBoard visualization).
  """
  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)
```

To use this function, we run it on W_conv1, b_conv1, h_conv1, h_pool1, W_conv2, b_conv2, h_conv2, h_pool2, W_fc1, b_fc1, h_pool2_flat, h_fc1, keep_prob, h_fc1_drop, W_fc2, b_fc2, and y_conv.

Here are the results:





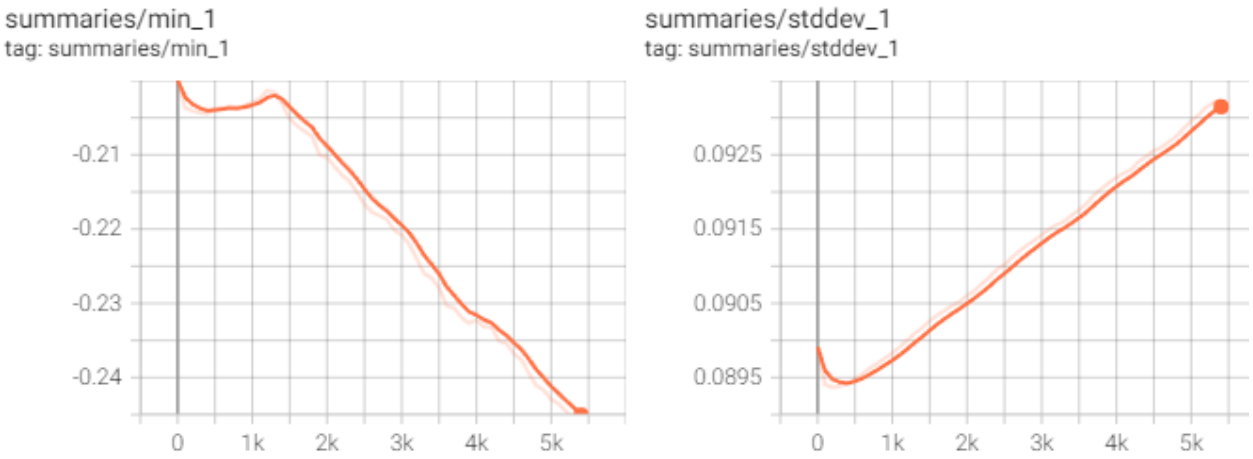


Figure 2_4: W_conv1

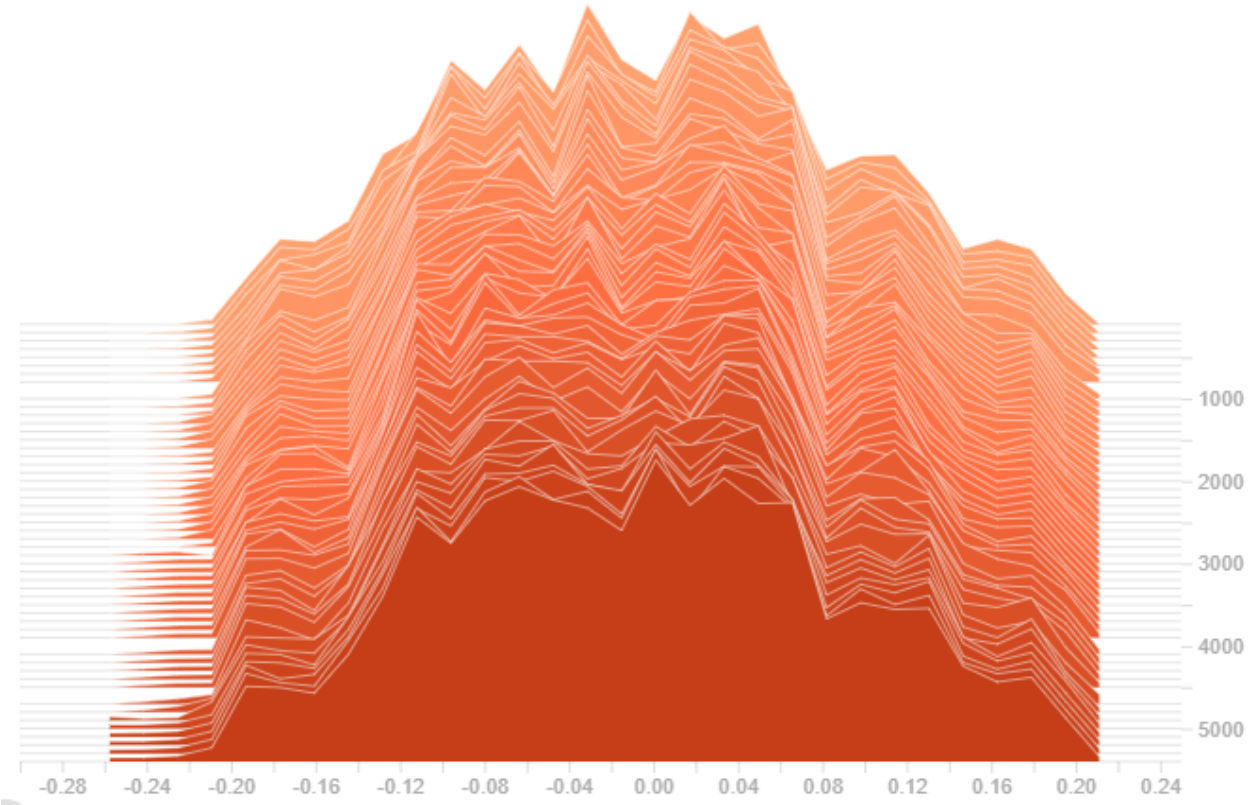


Figure 2_5: W_conv1_histogram

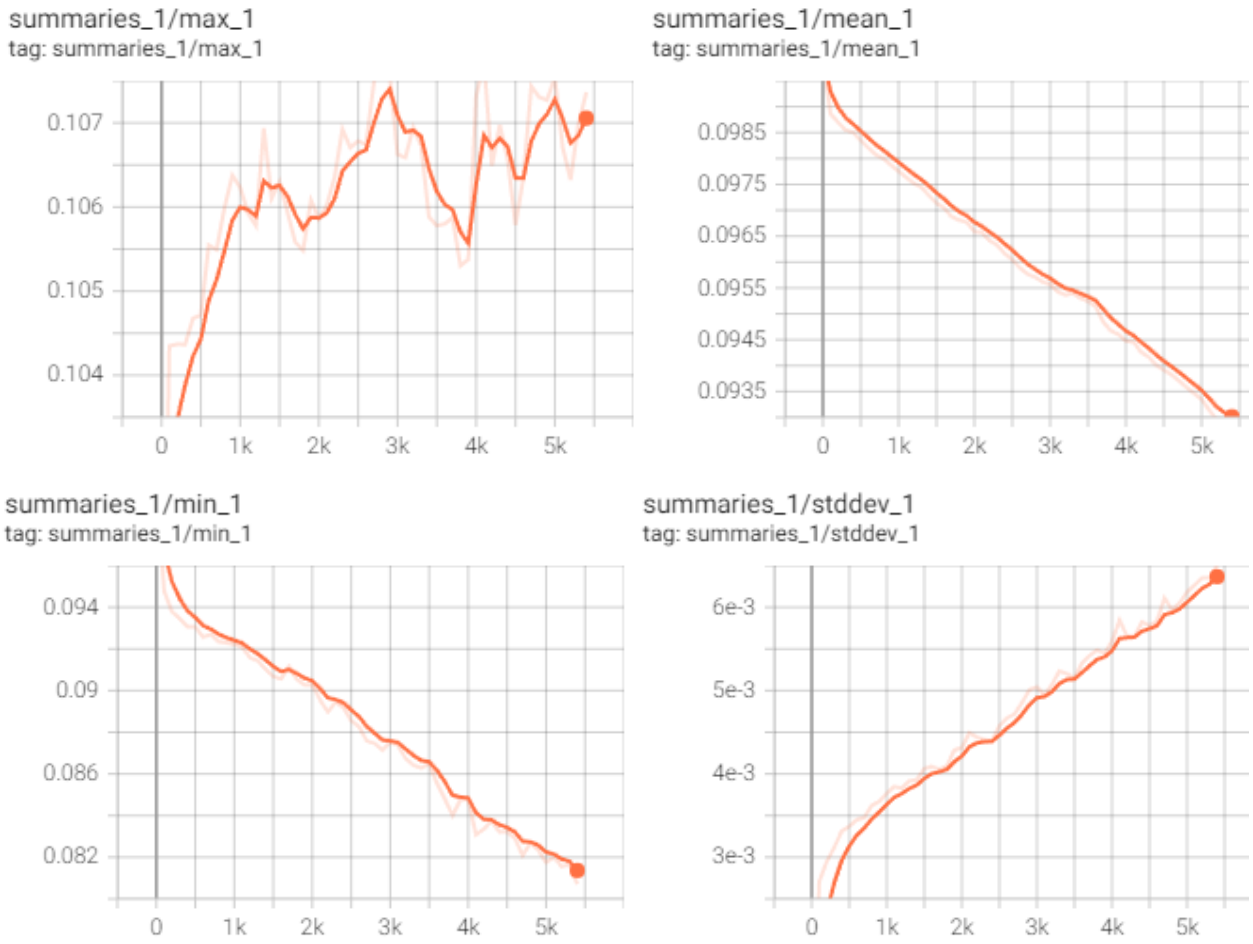


Figure 2_6: b_conv1

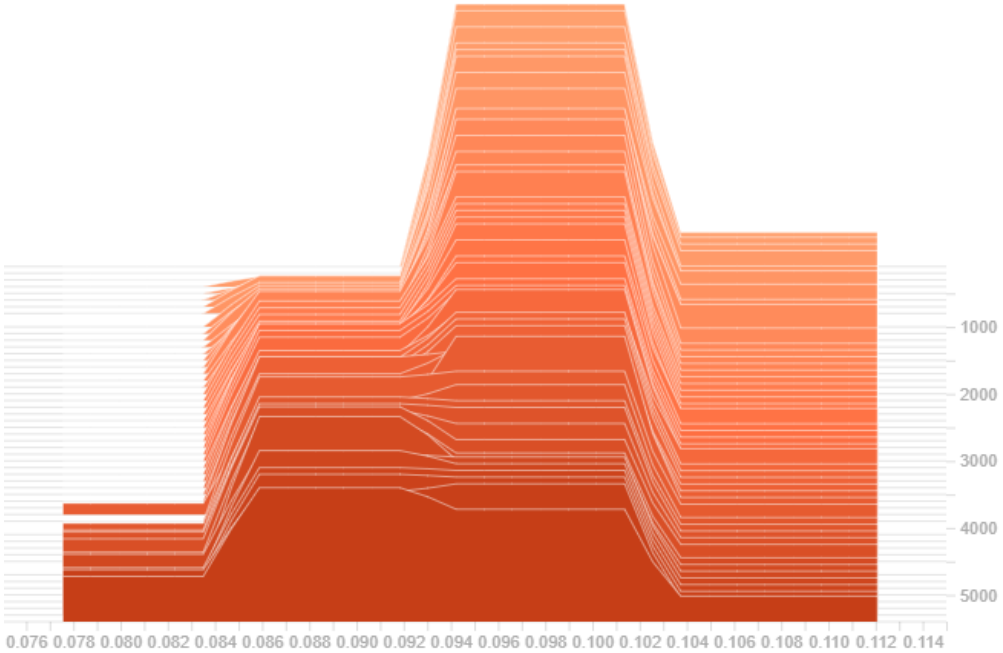


Figure 2_7: b_conv1_histogram

Figure 2_8: h_conv1

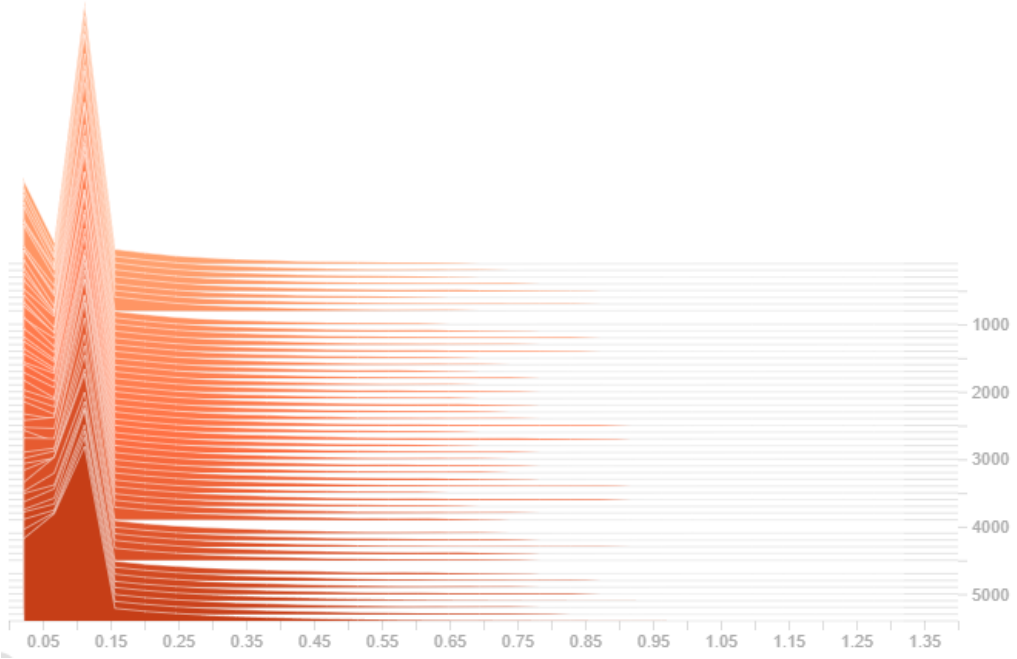


Figure 2_9: h_conv1_histogram

Figure 2_10: h_pool1

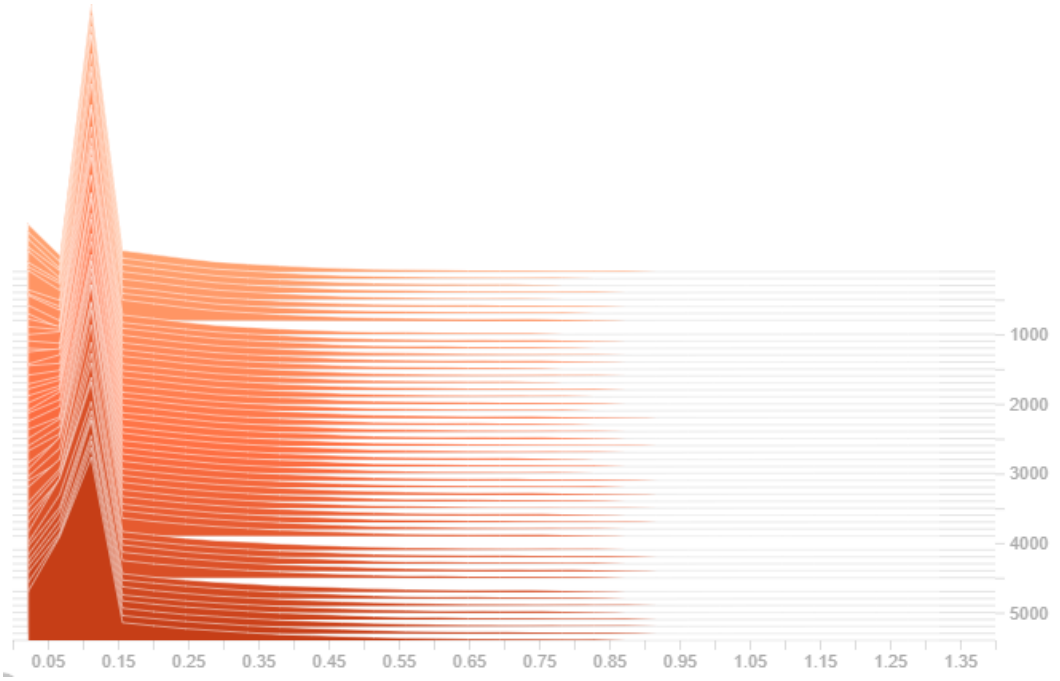


Figure 2_11: h_pool1_histogram

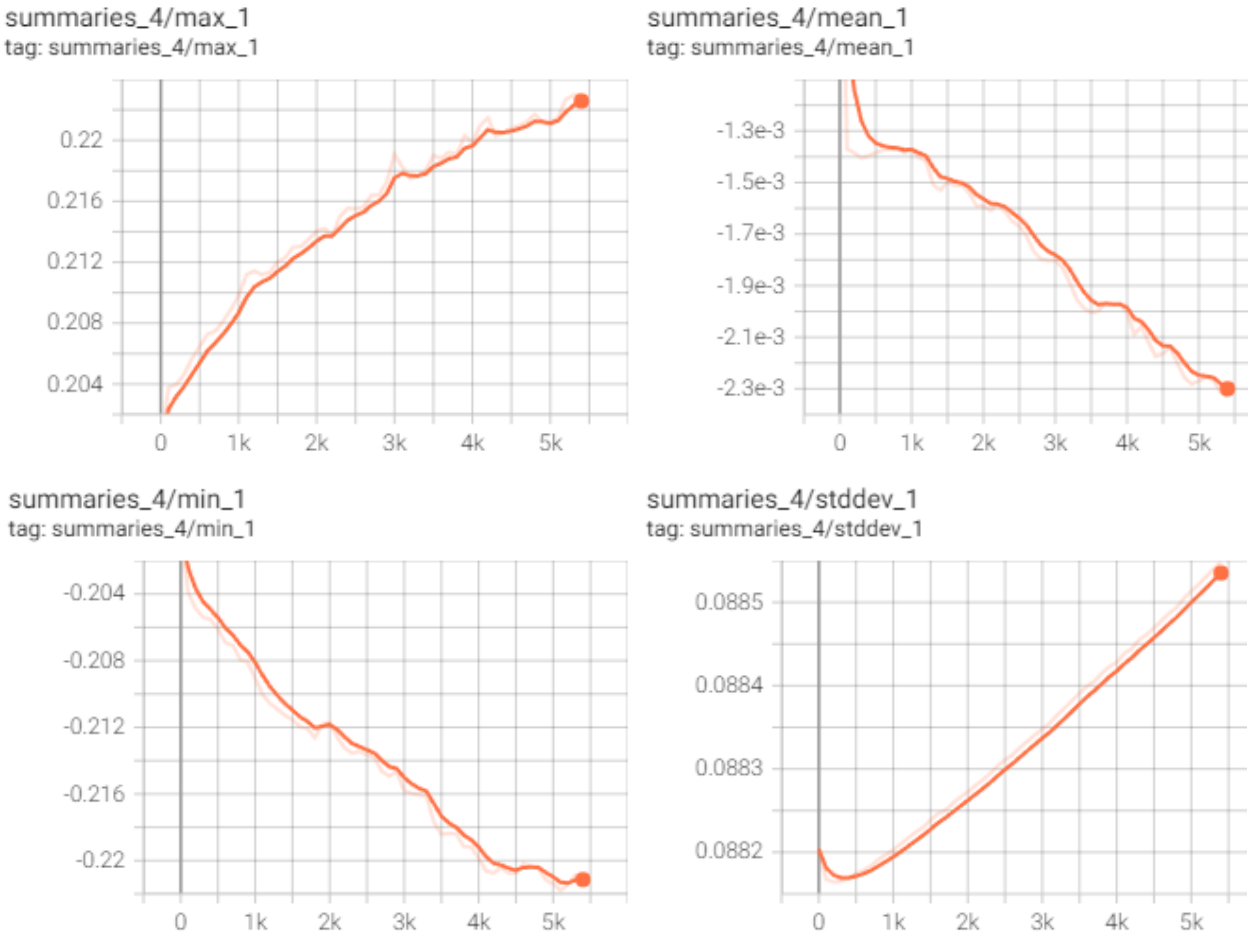


Figure 2_12: W_conv2

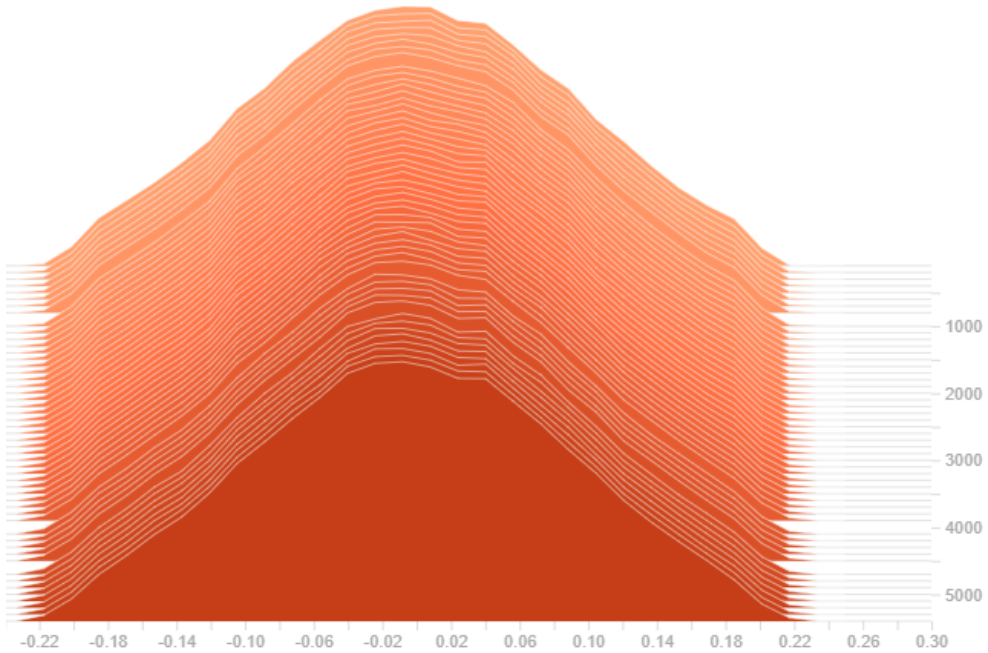


Figure 2_13: W_conv2_histogram

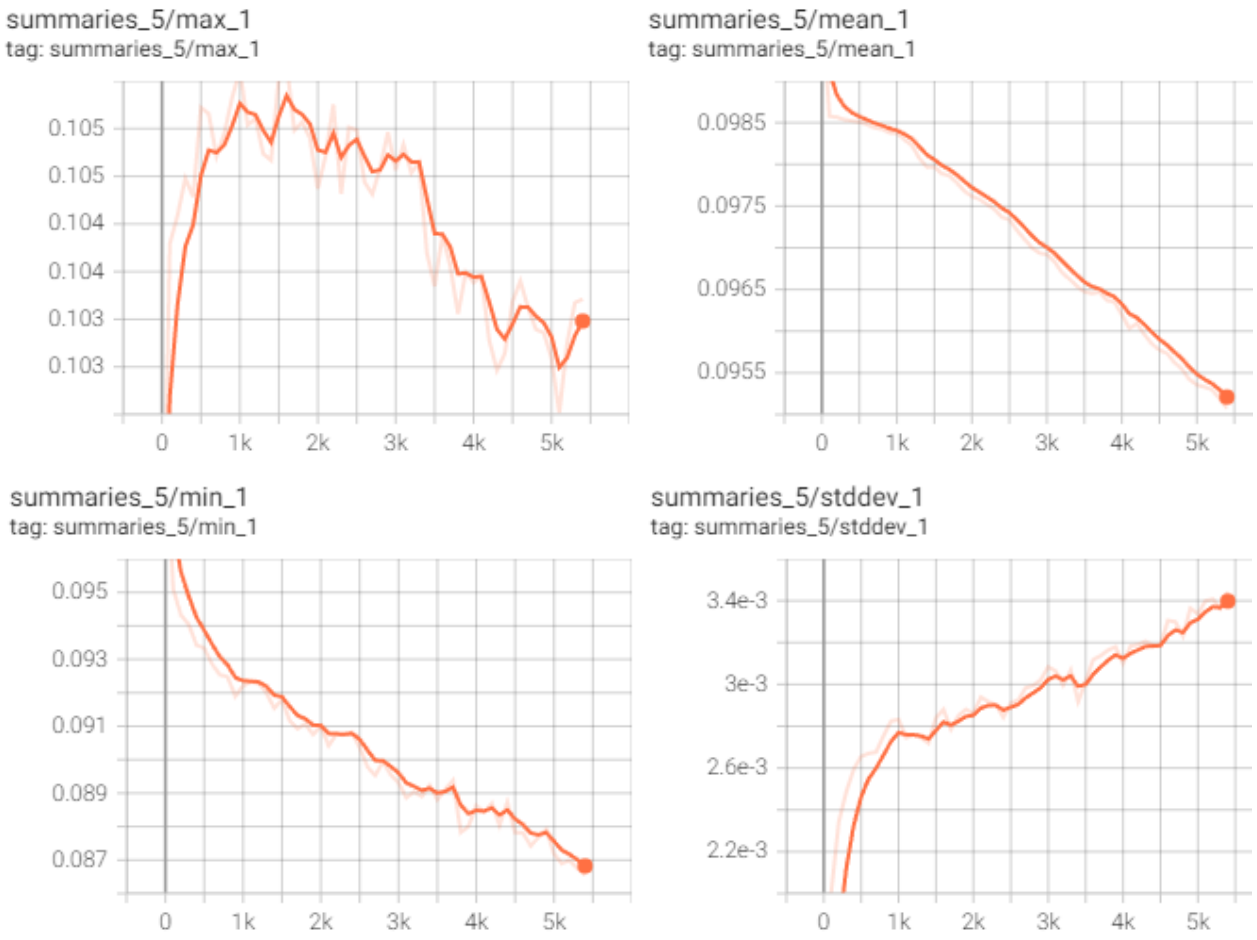


Figure 2_14: b_conv2

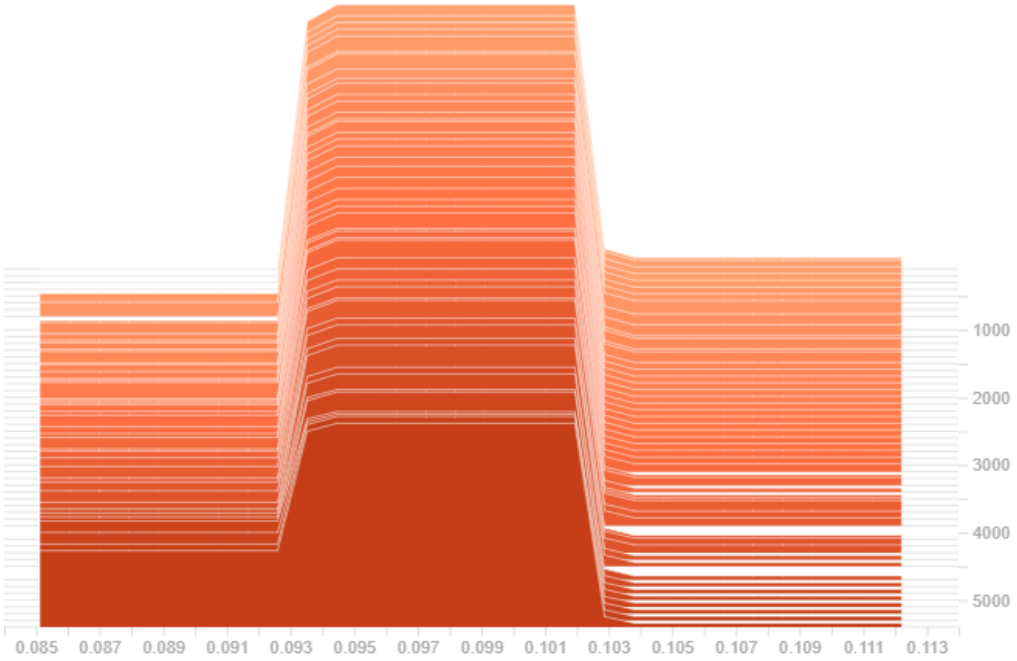


Figure 2_15: b_conv2_histogram

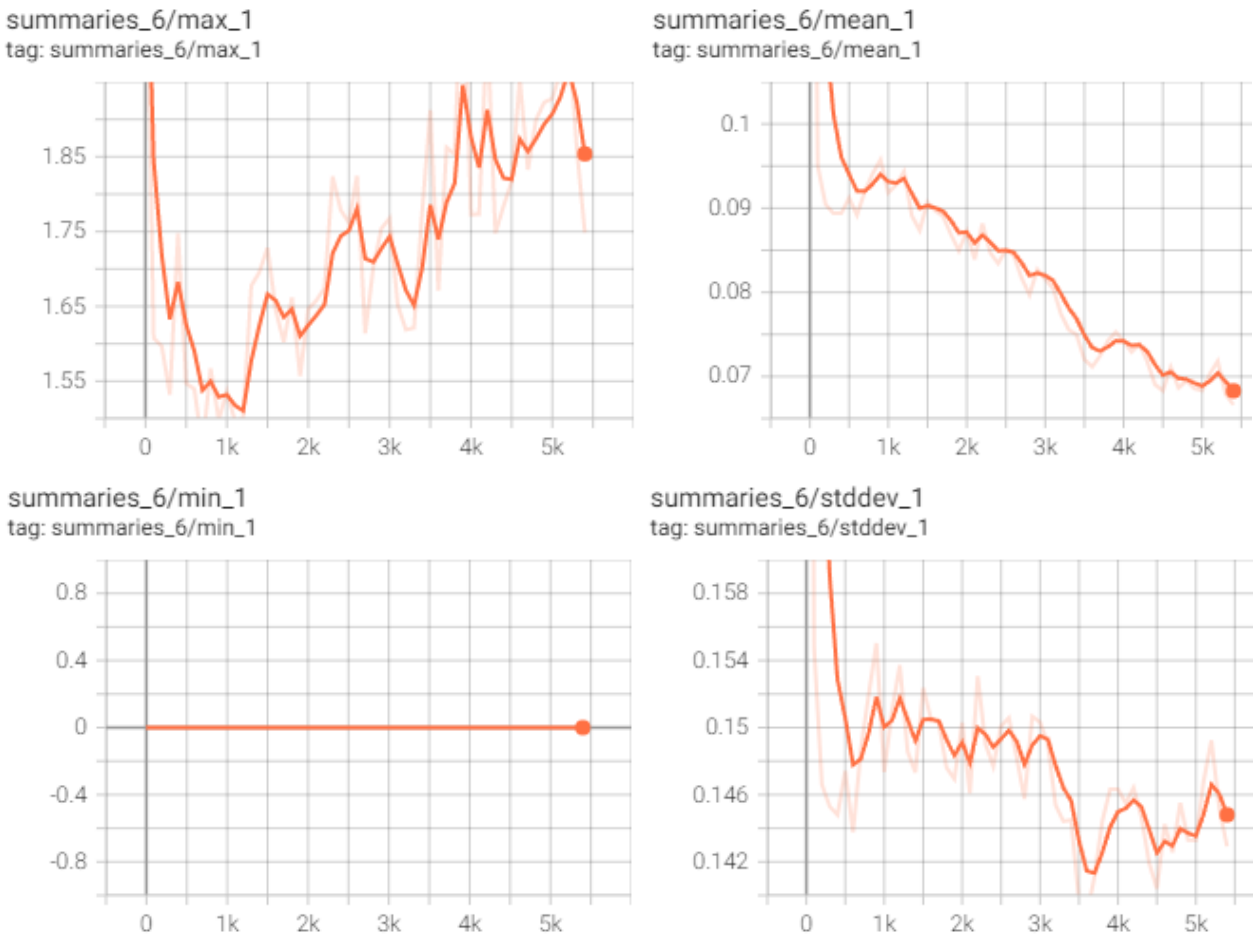


Figure 2_16: h_conv2

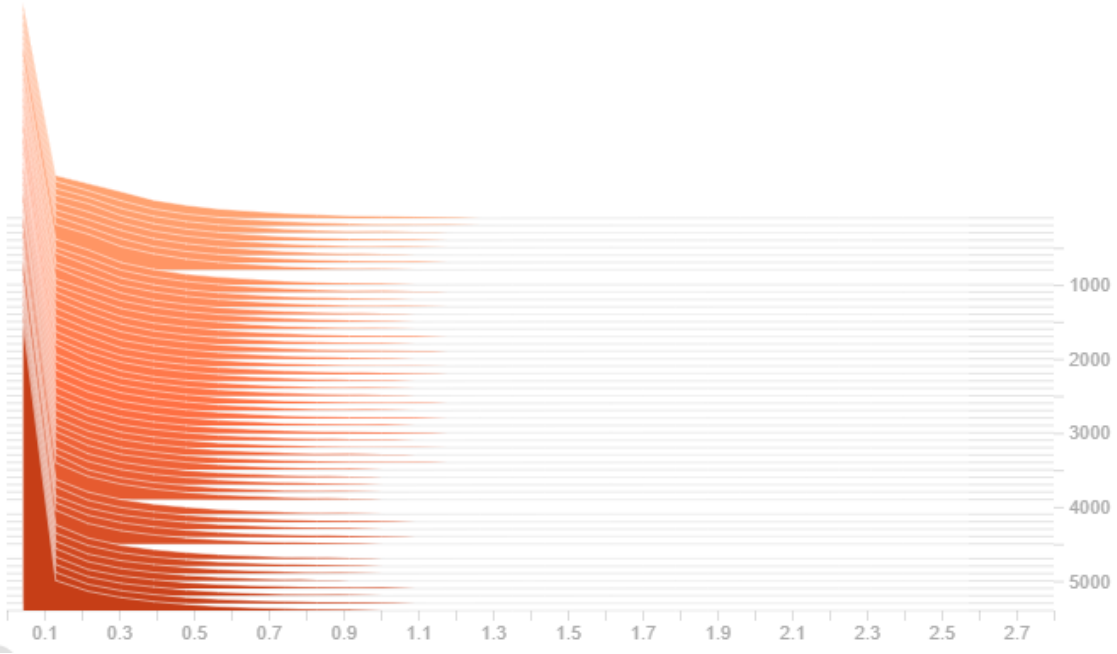


Figure 2_17: h_conv2_histogram

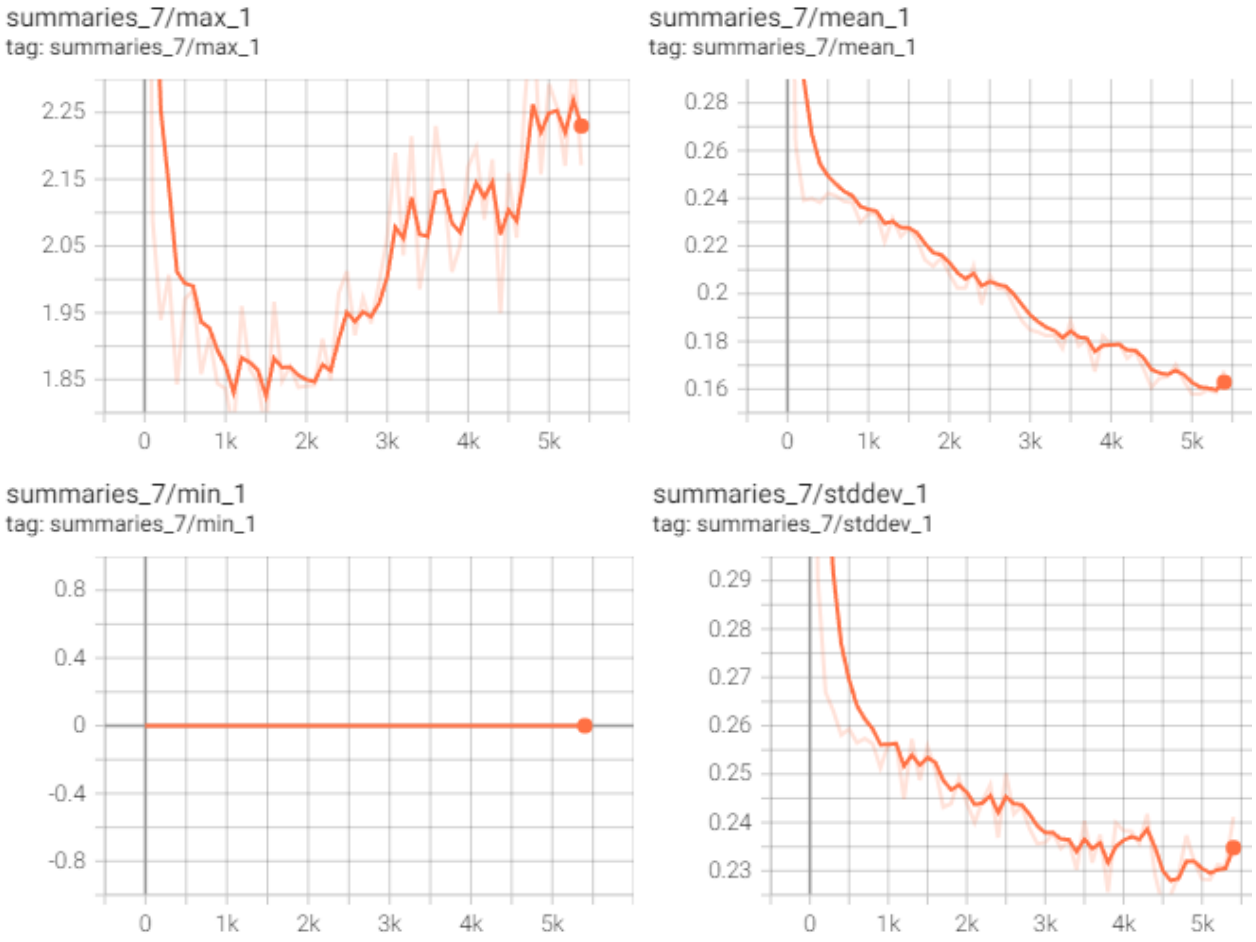


Figure 2_18: h_pool2

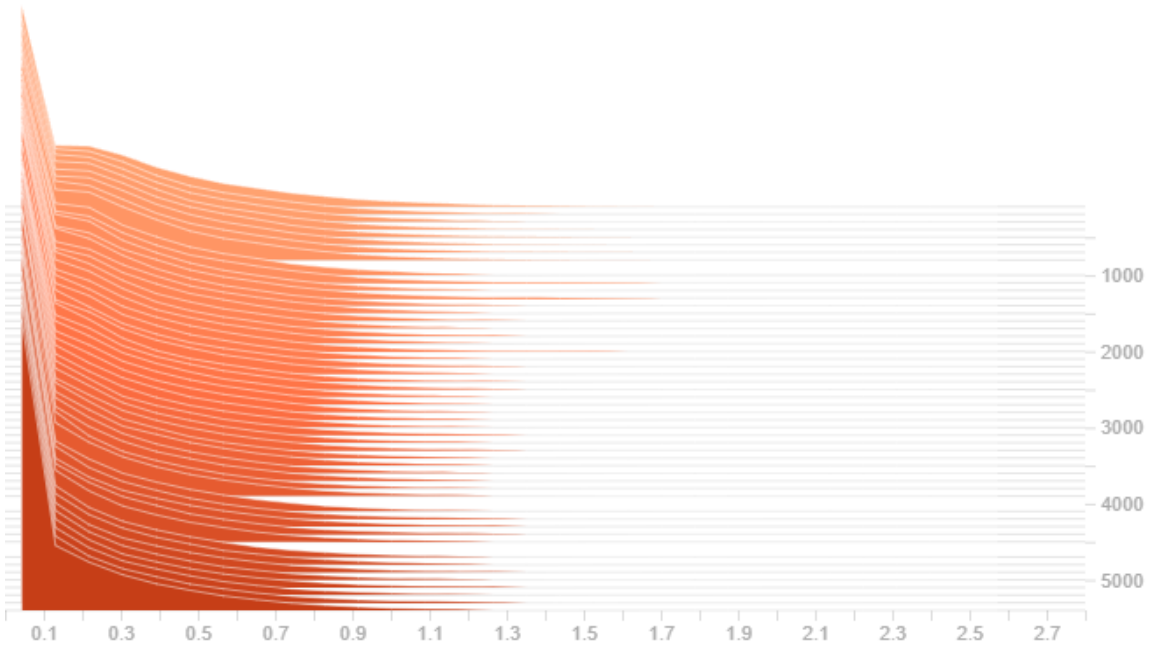


Figure 2_19: h_pool2_histogram

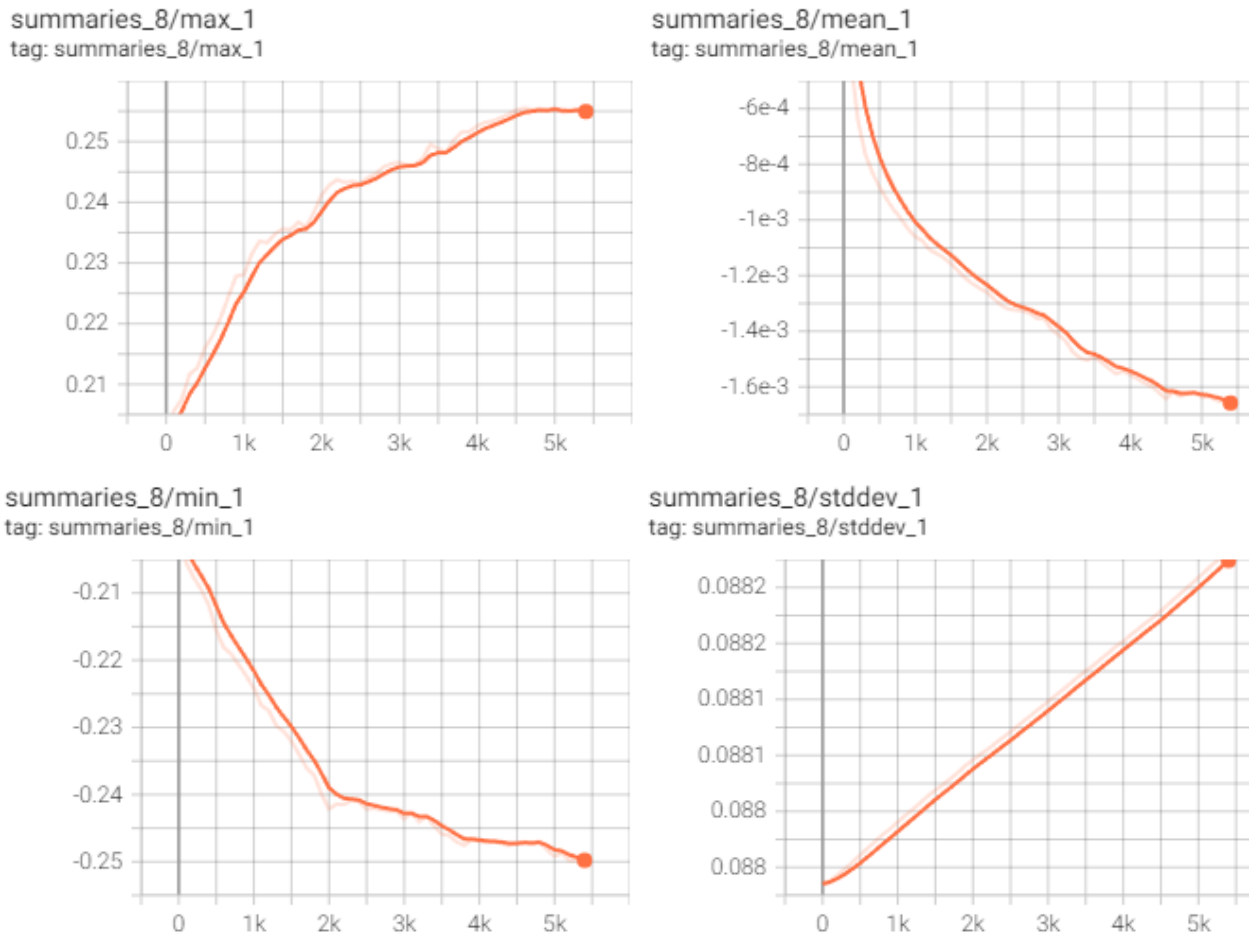


Figure 2_20: W_fc1

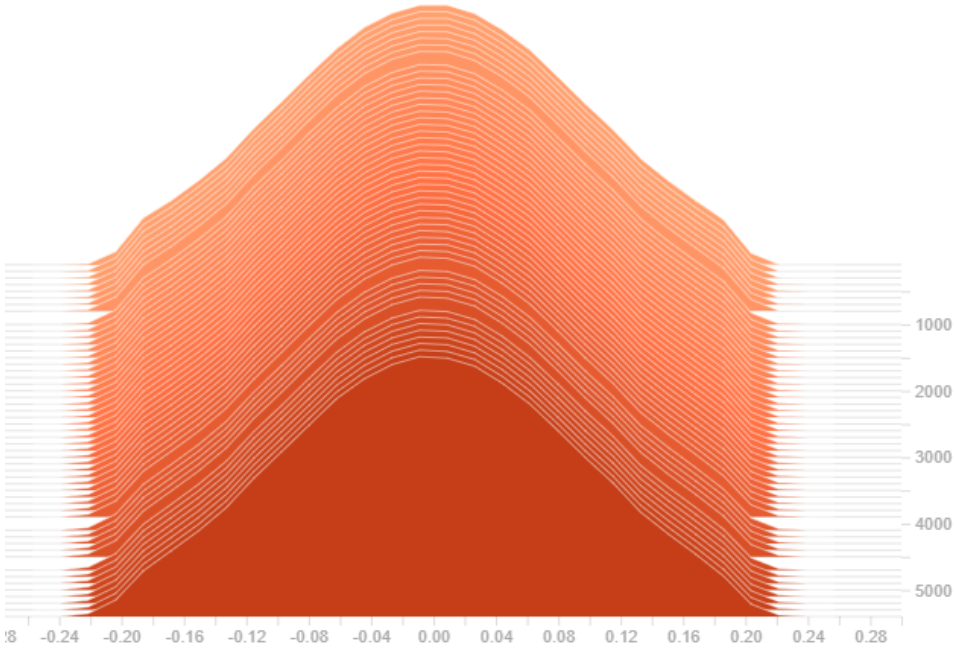


Figure 2_21: W_fc1_histogram

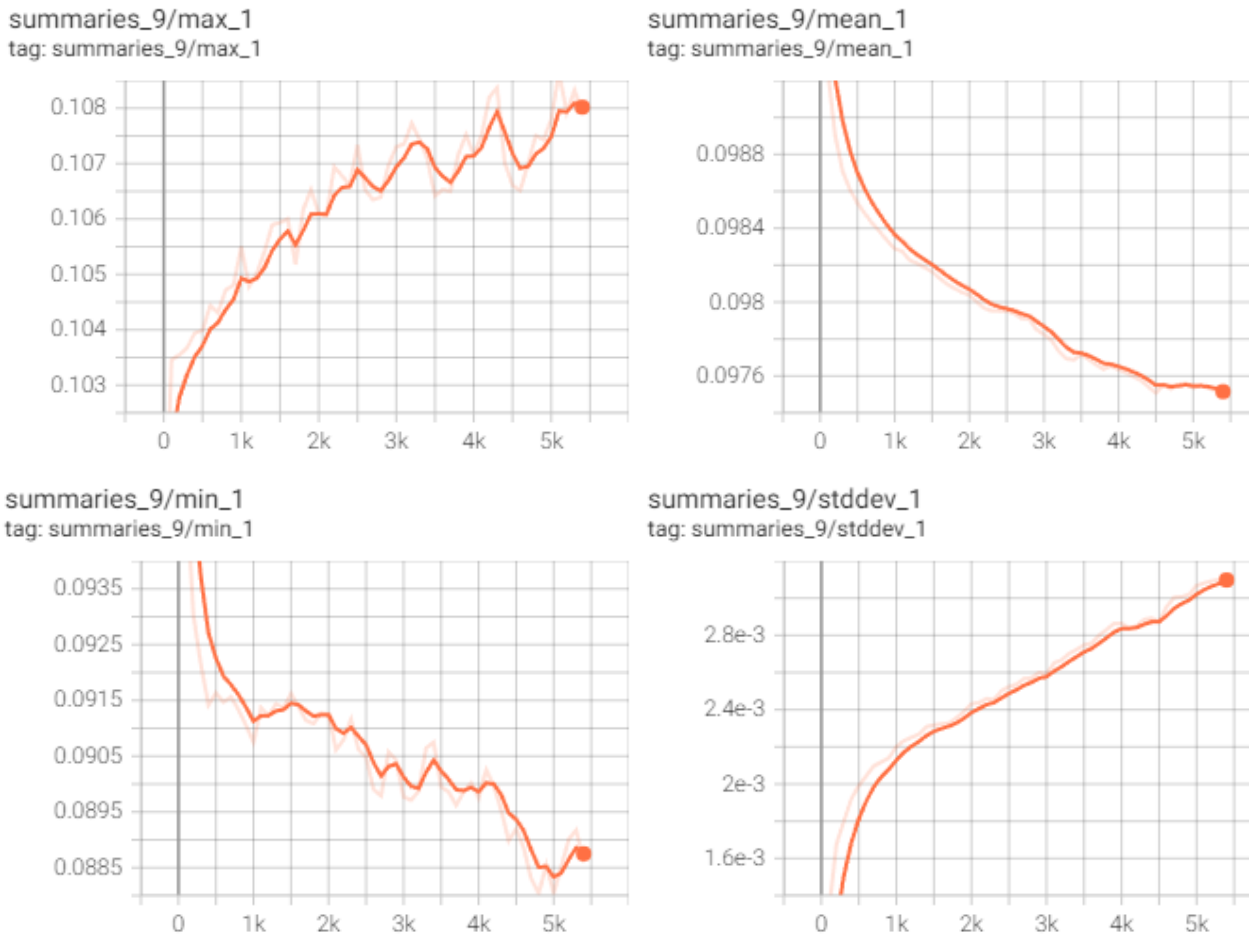


Figure 2_22: b_fc1

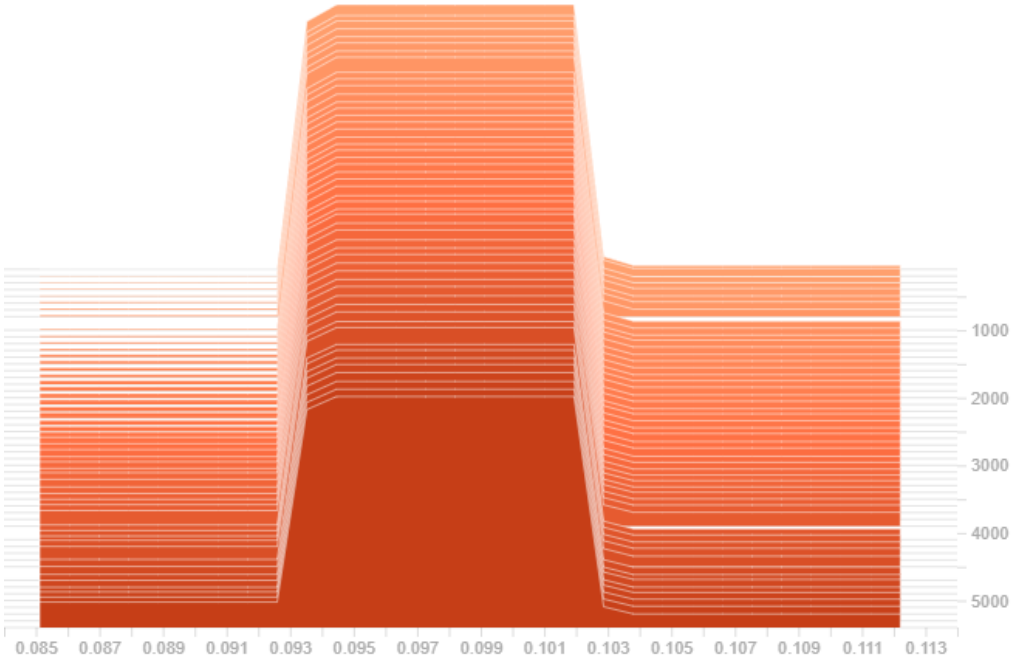


Figure 2_23: b_fc1_histogram

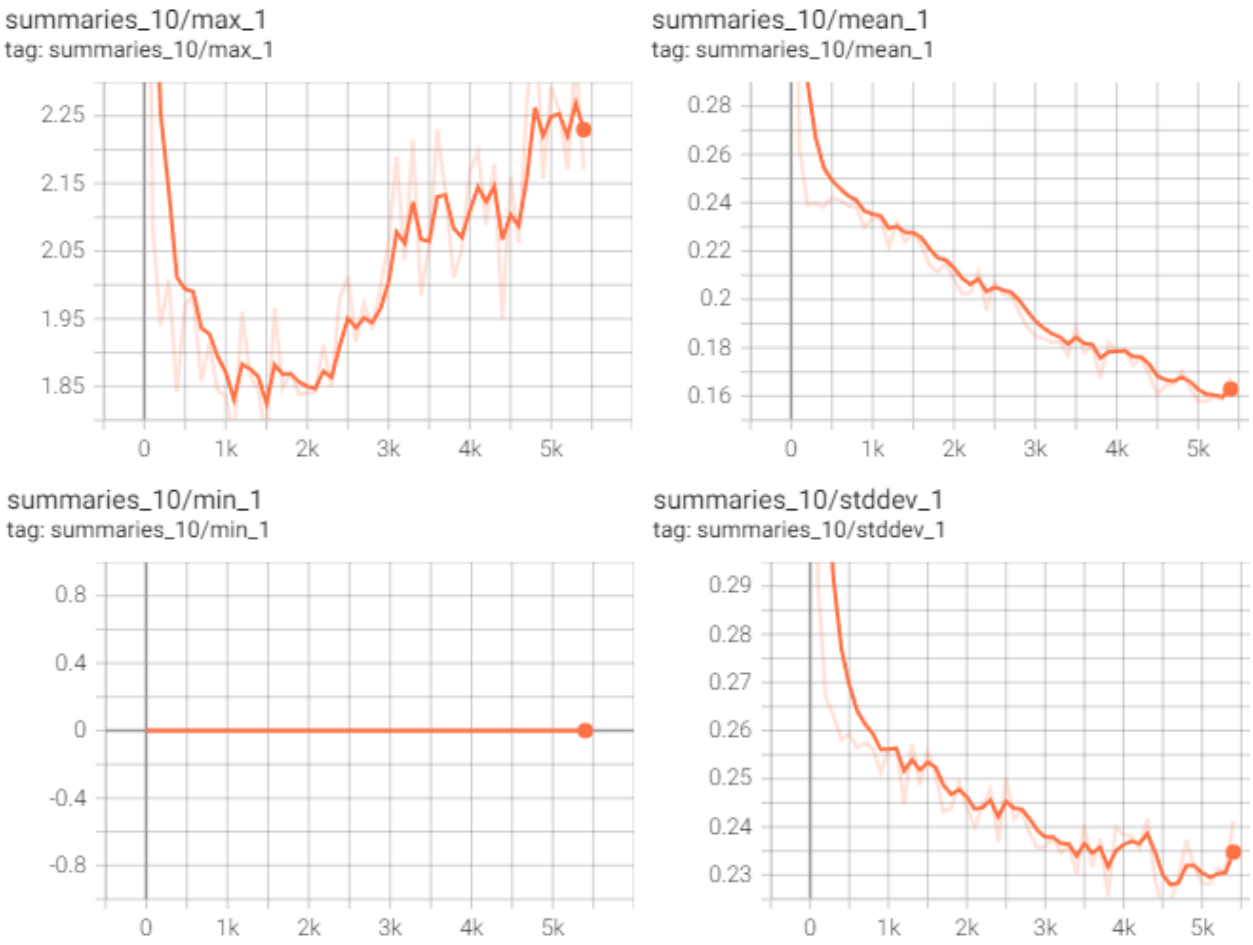


Figure 2_24: h_pool2_flat

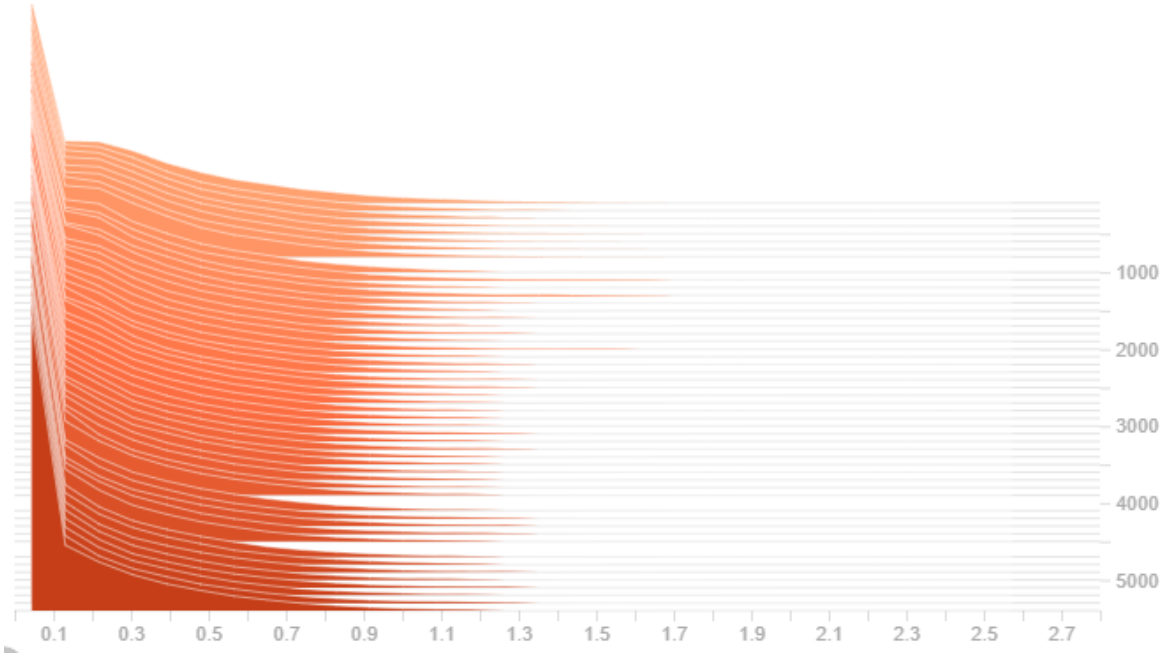


Figure 2_25: h_pool2_flat_histogram

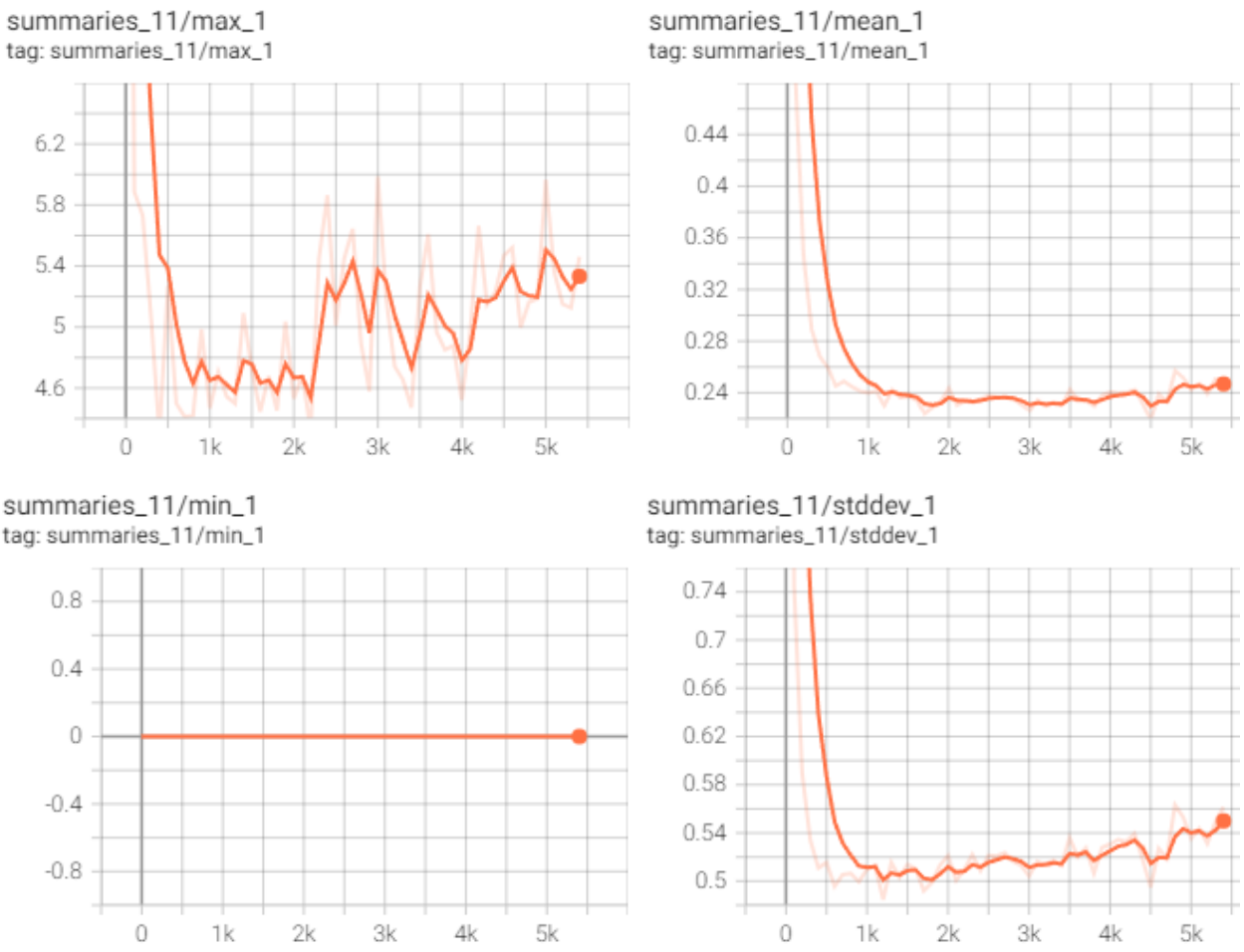


Figure 2_26: h_fc1

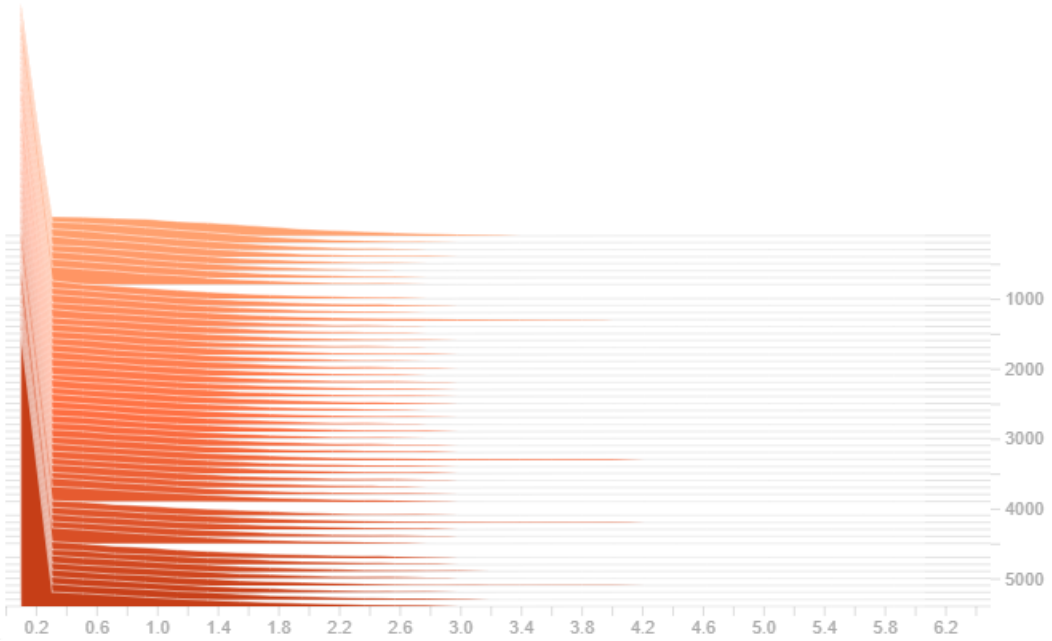


Figure 2_27: h_fc1_histogram

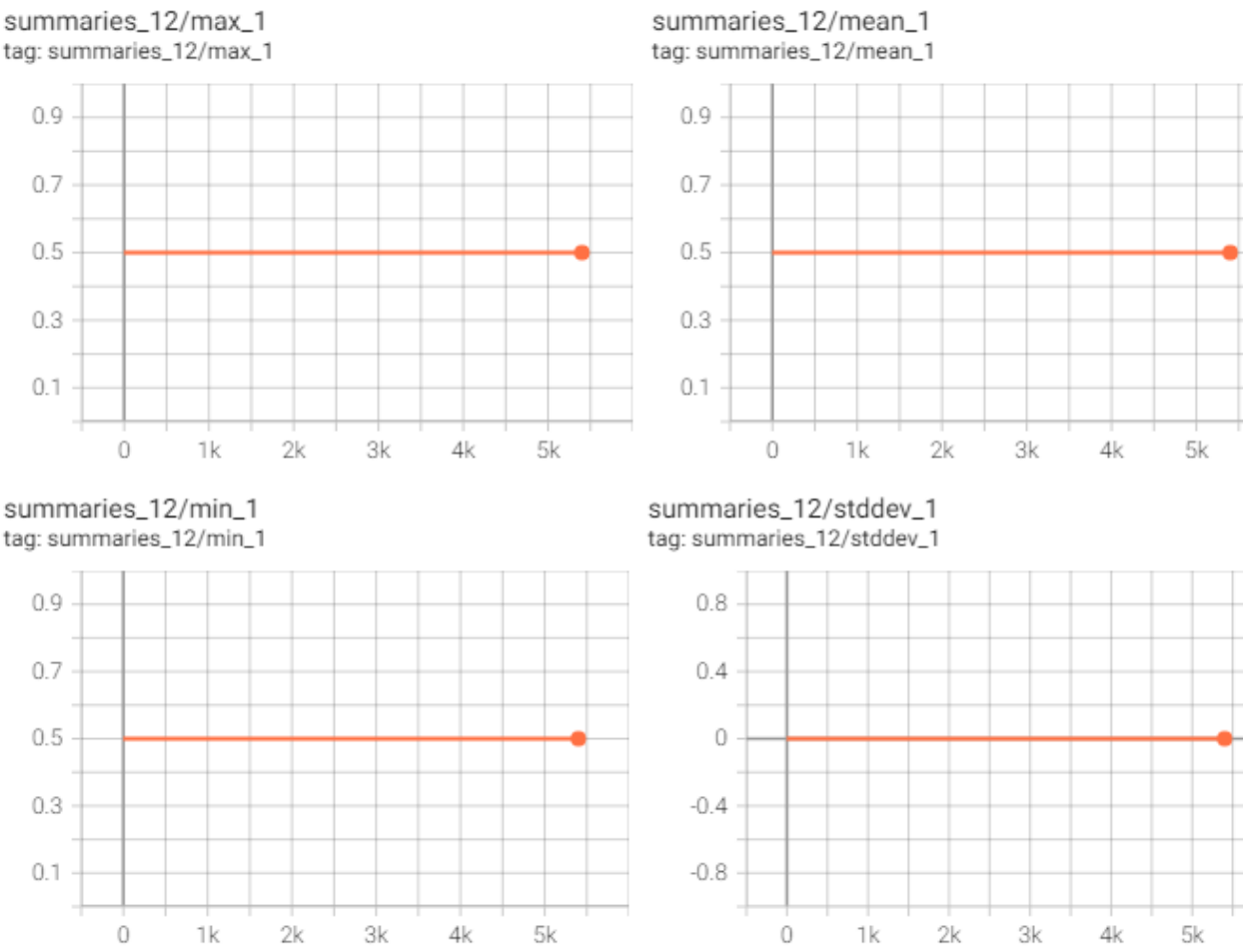


Figure 2_28: keep_prob

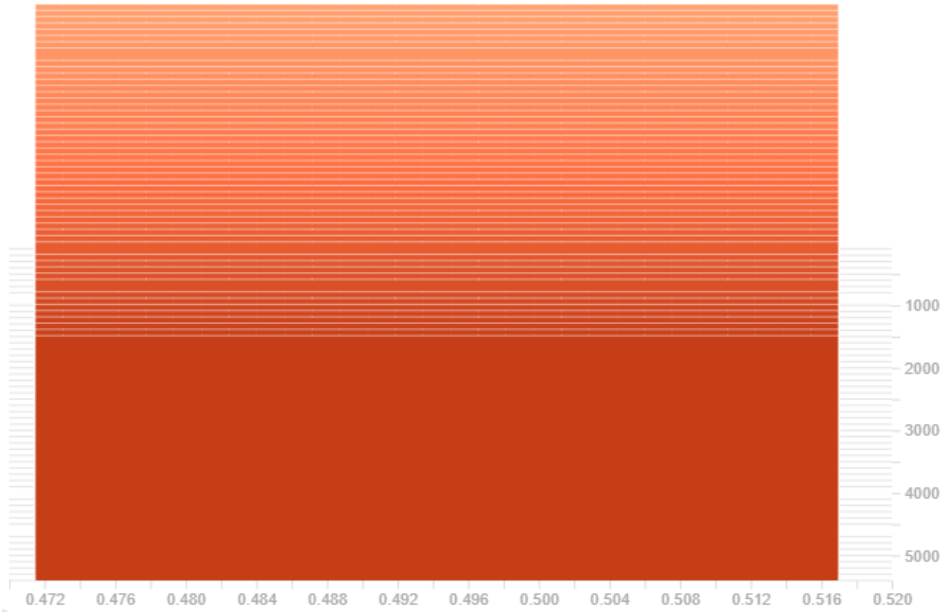


Figure 2_29: keep_prob_histogram

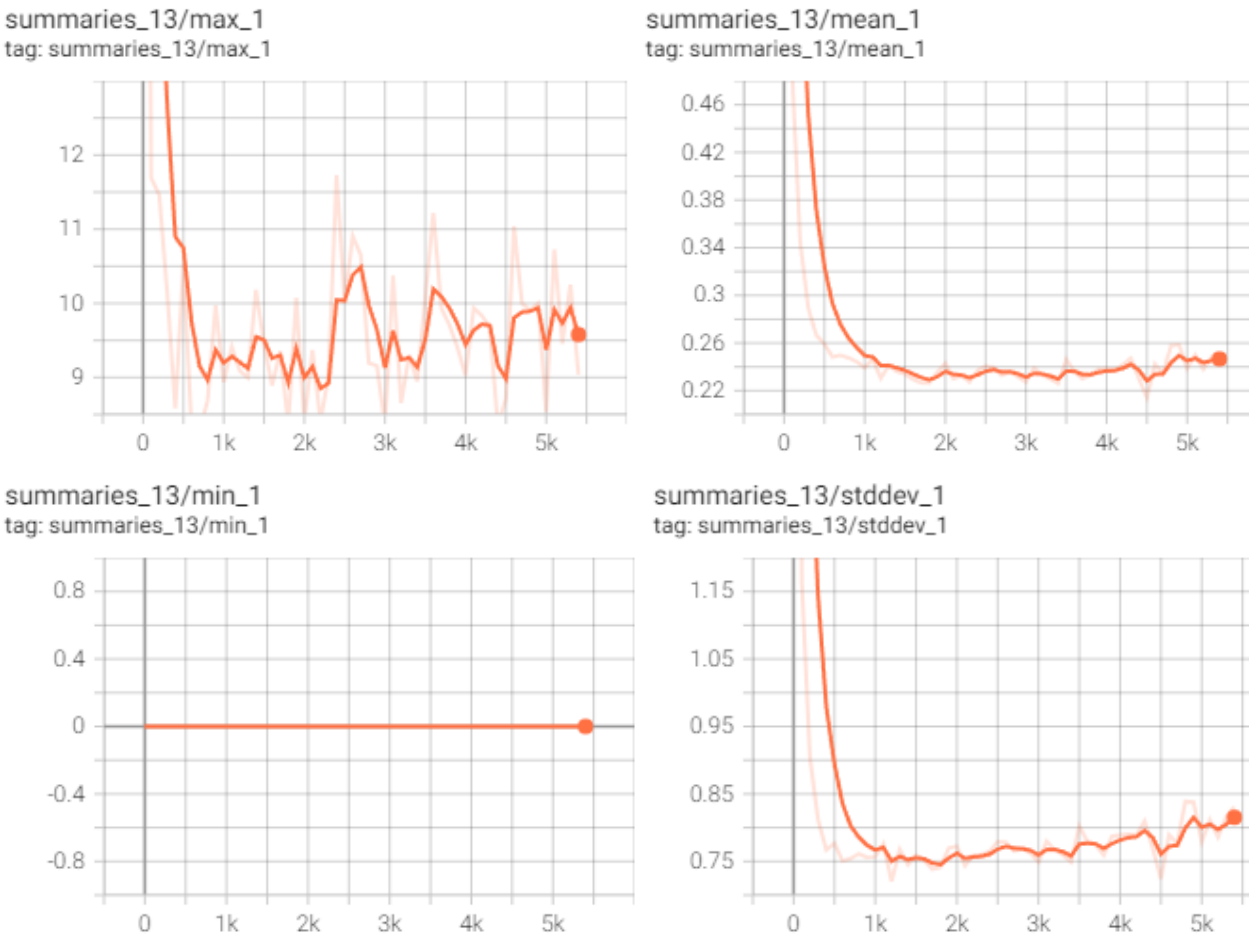


Figure 2_30: h_fc1_drop

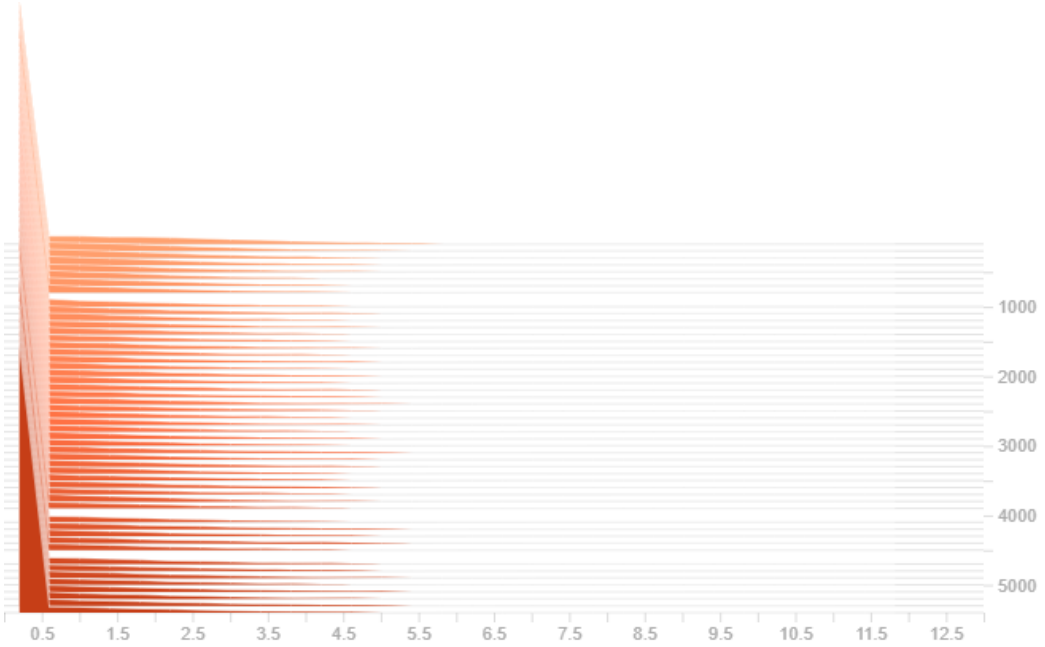


Figure 2_31: h_fc1_drop_histogram

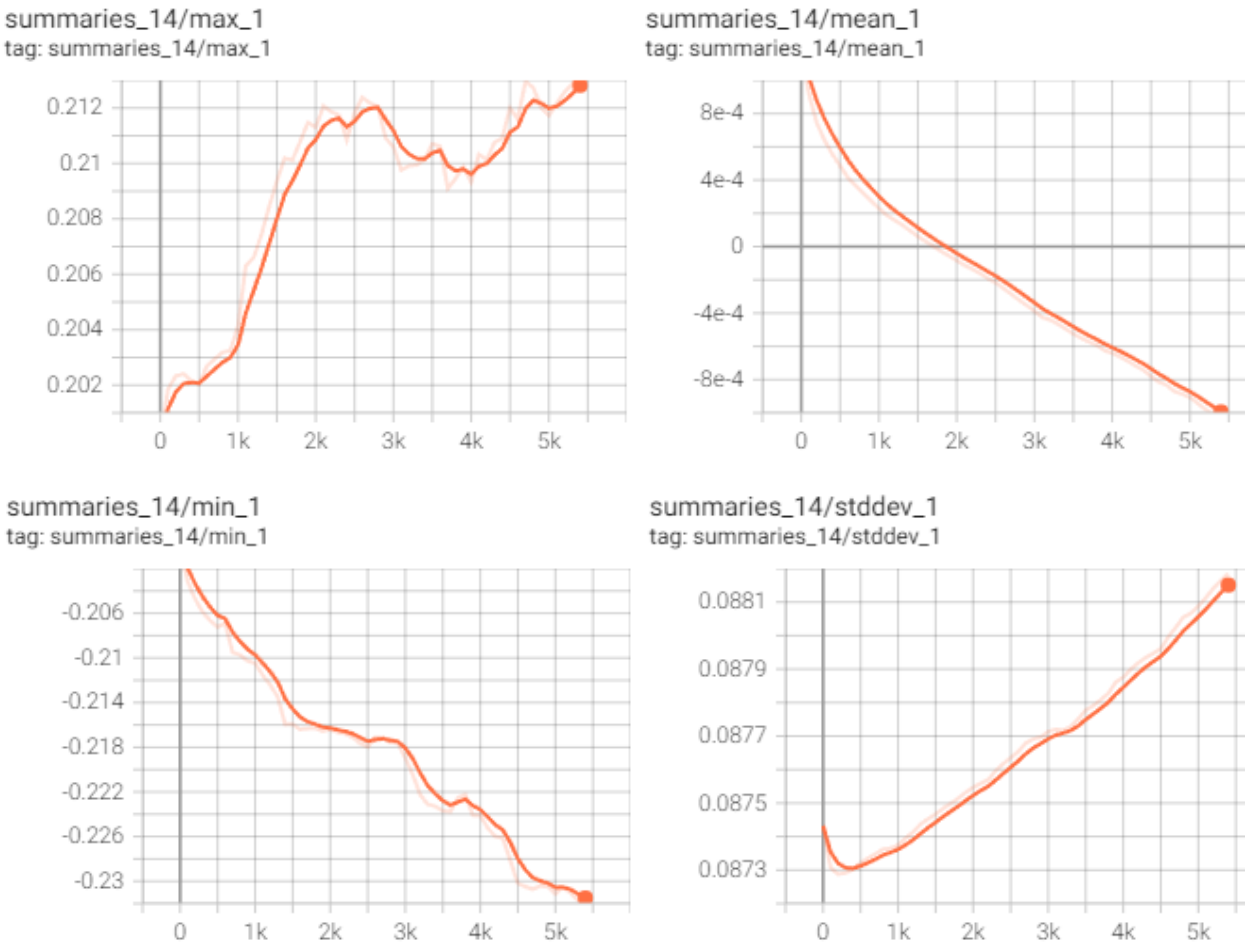


Figure 2_32: W_fc2

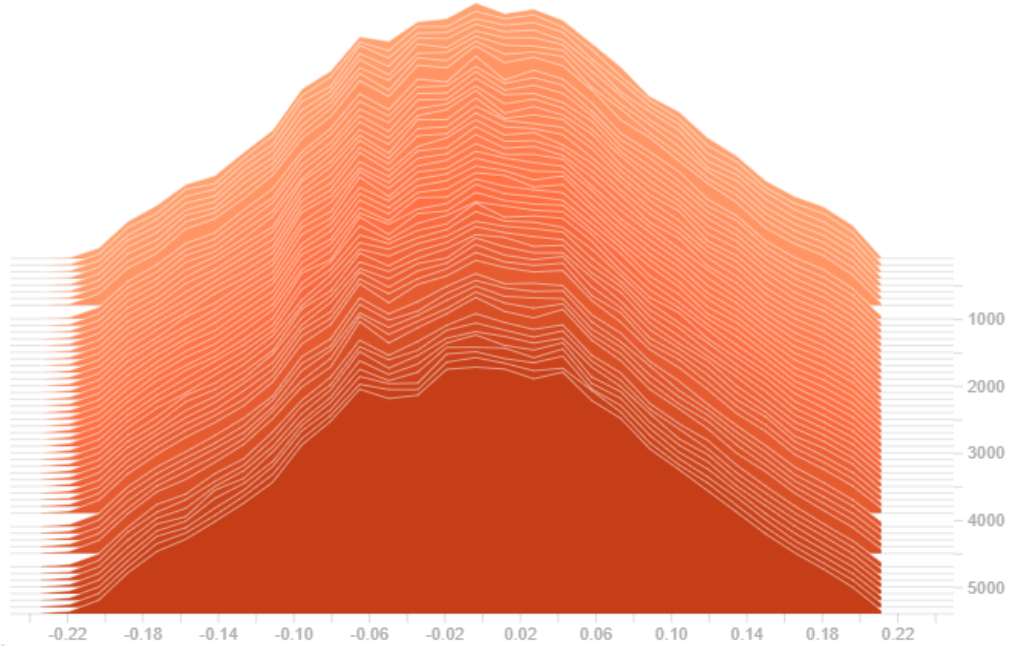


Figure 2_33: W_fc2_histogram

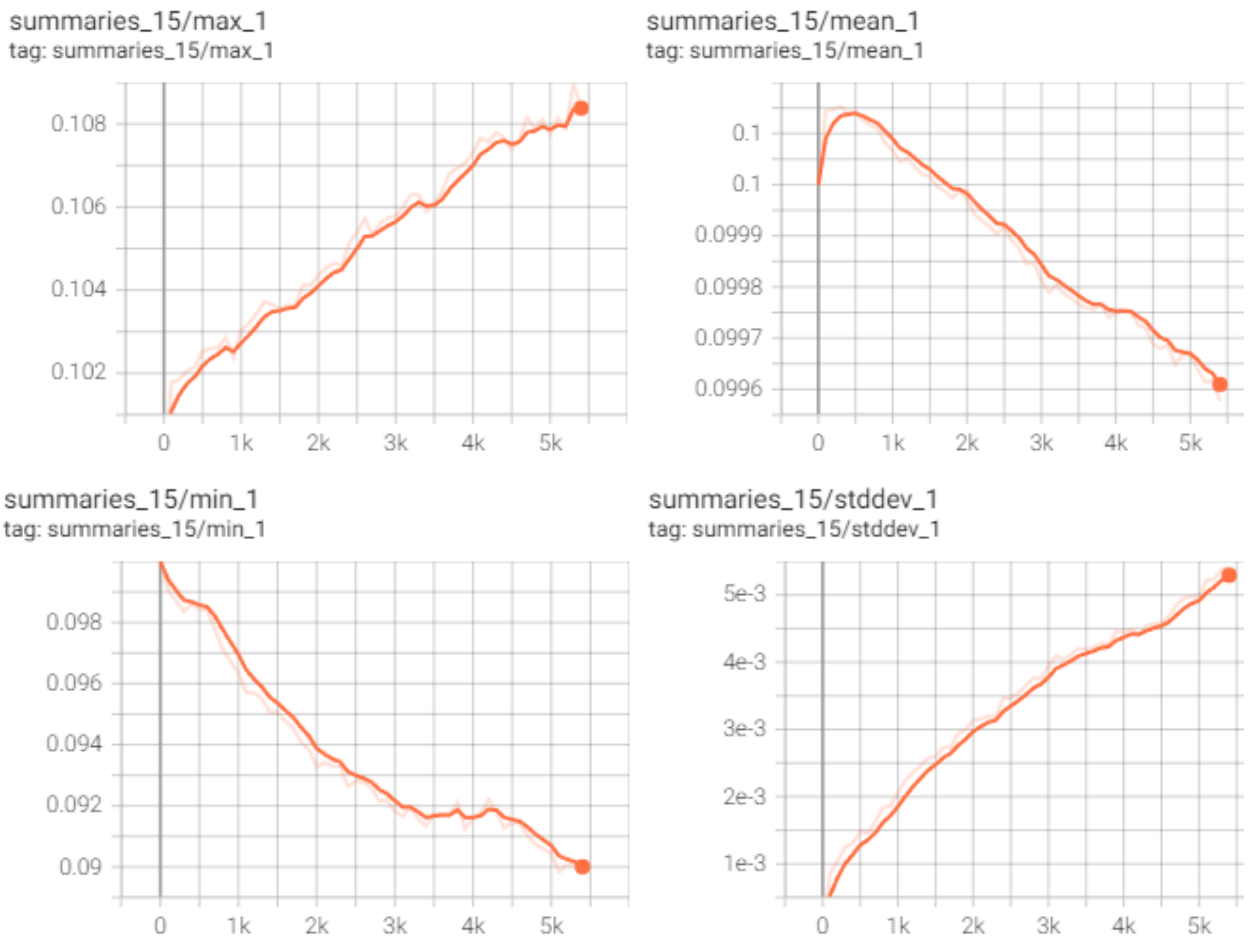


Figure 2_34: b_fc2

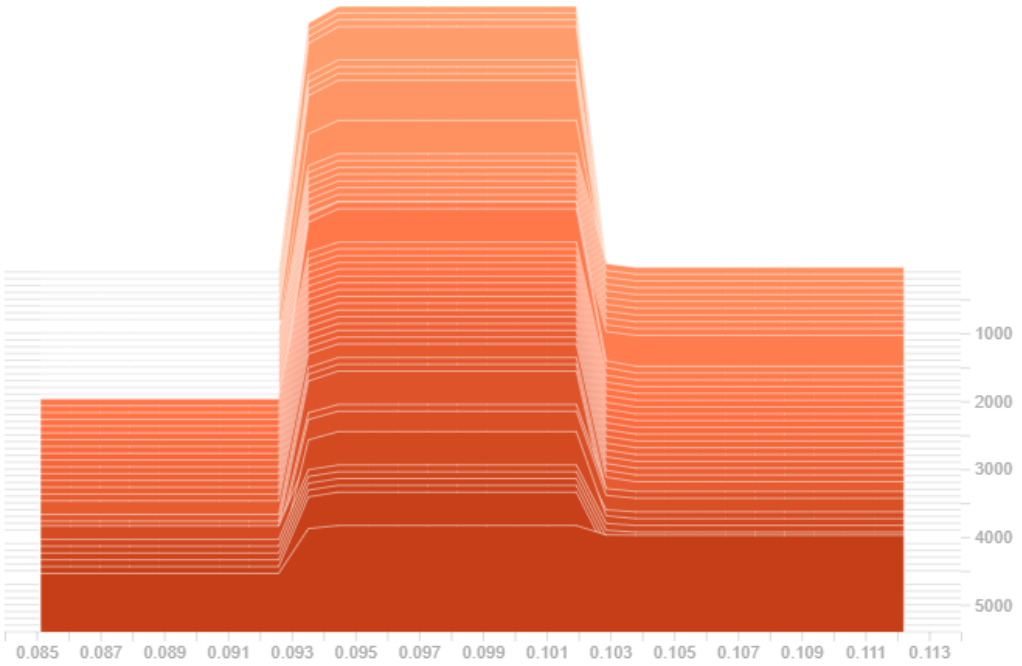


Figure 2_35: b_fc2_histogram

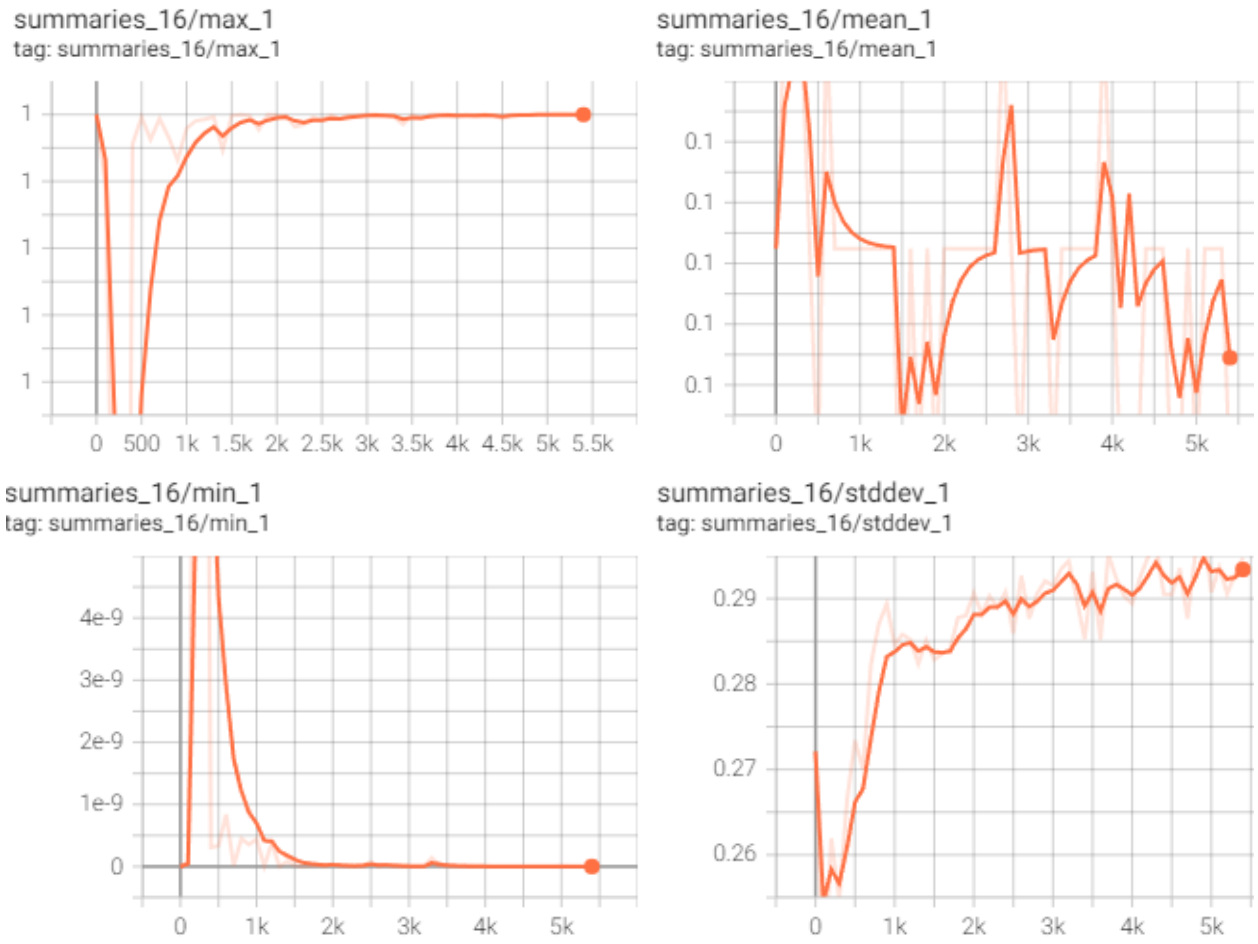


Figure 2_36: y_conv

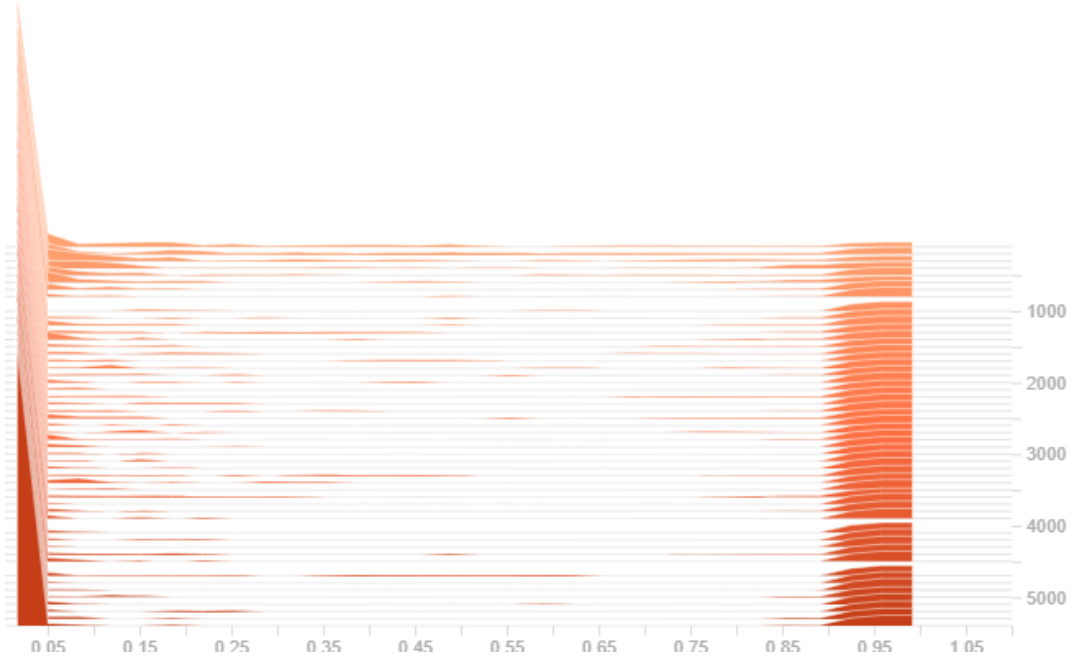


Figure 2_37: y_conv_histogram

2.c

I change the relu to tanh for h_conv1, h_conv2, h_fc1.

Here is the final output results:

step 0, training accuracy 0.06
step 100, training accuracy 0.86
step 200, training accuracy 0.84
step 300, training accuracy 0.86
step 400, training accuracy 0.88
step 500, training accuracy 0.92
step 600, training accuracy 0.96
step 700, training accuracy 0.96
step 800, training accuracy 0.98
step 900, training accuracy 0.92
step 1000, training accuracy 0.92
step 1100, training accuracy 0.96
step 1200, training accuracy 0.98
step 1300, training accuracy 0.92
step 1400, training accuracy 0.94
step 1500, training accuracy 0.92
step 1600, training accuracy 0.94
step 1700, training accuracy 0.96
step 1800, training accuracy 0.96
step 1900, training accuracy 1
step 2000, training accuracy 0.98
step 2100, training accuracy 0.96
step 2200, training accuracy 0.96
step 2300, training accuracy 0.96
step 2400, training accuracy 1
step 2500, training accuracy 0.98
step 2600, training accuracy 0.98
step 2700, training accuracy 0.96
step 2800, training accuracy 0.94
step 2900, training accuracy 0.98
step 3000, training accuracy 0.94
step 3100, training accuracy 1
step 3200, training accuracy 1
step 3300, training accuracy 1

step 3400, training accuracy 1
step 3500, training accuracy 1
step 3600, training accuracy 1
step 3700, training accuracy 1
step 3800, training accuracy 0.96
step 3900, training accuracy 1
step 4000, training accuracy 0.98
step 4100, training accuracy 0.98
step 4200, training accuracy 1
step 4300, training accuracy 0.98
step 4400, training accuracy 0.98
step 4500, training accuracy 0.98
step 4600, training accuracy 0.98
step 4700, training accuracy 0.98
step 4800, training accuracy 0.98
step 4900, training accuracy 1
step 5000, training accuracy 1
step 5100, training accuracy 1
step 5200, training accuracy 0.98
step 5300, training accuracy 1
step 5400, training accuracy 1
test accuracy 0.9835
The training takes 879.099484 second to finish

The figure of scalars:

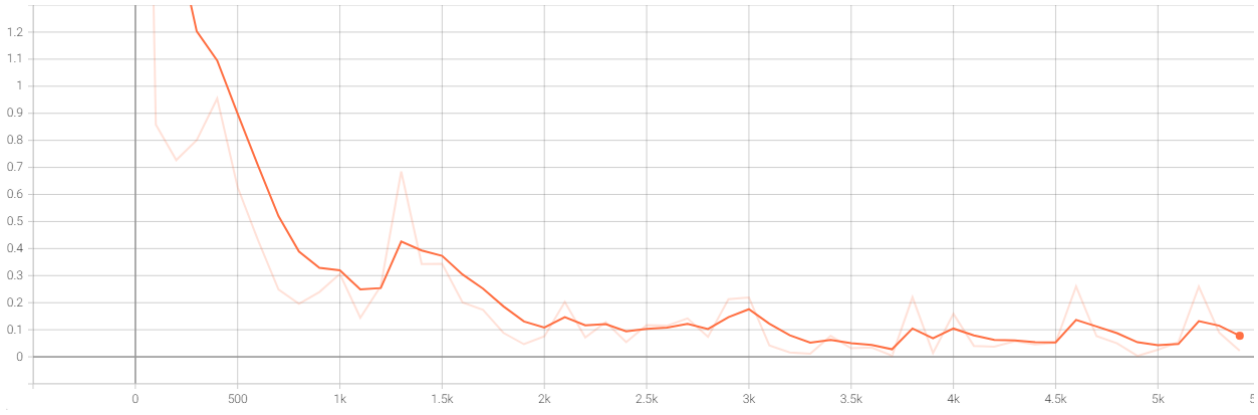


Figure 2_38: scalars

To use the function variable_summaries, we run it on W_conv1, b_conv1, h_conv1, h_pool1, W_conv2, b_conv2, h_conv2, h_pool2, W_fc1, b_fc1, h_pool2_flat, h_fc1, keep_prob, h_fc1_drop, W_fc2, b_fc2, and y_conv.

Here are the results:

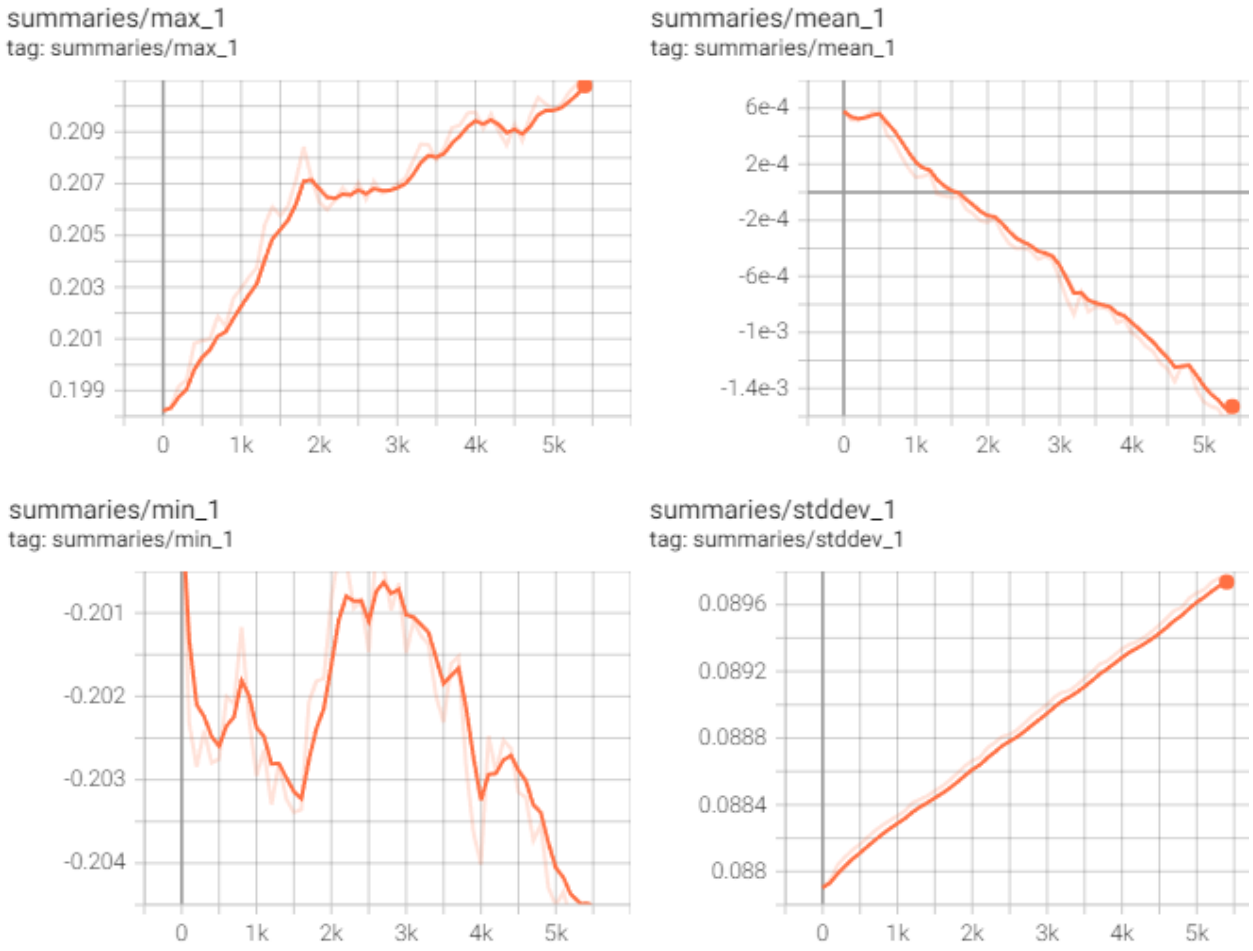


Figure 2_39: W_conv1

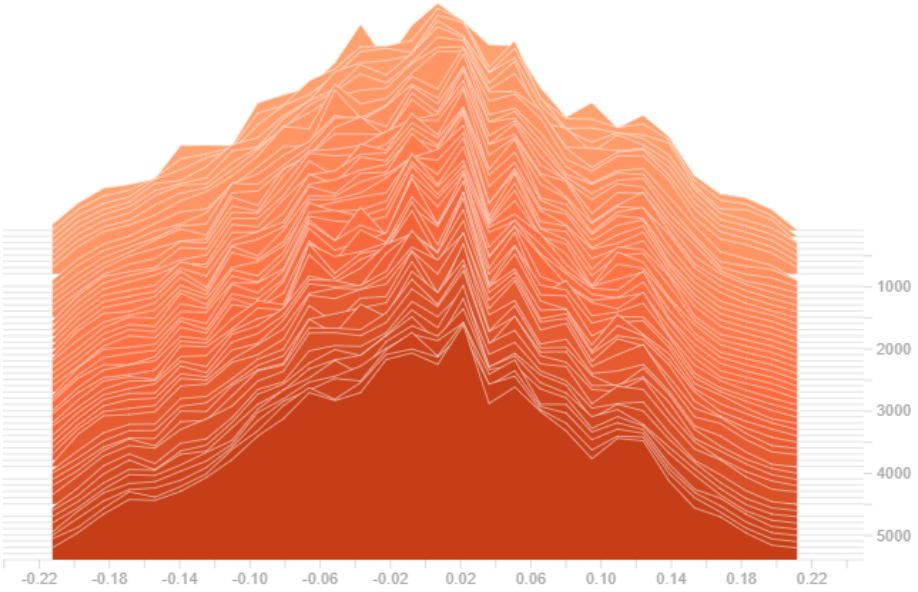


Figure 2_40: W_conv1_histogram

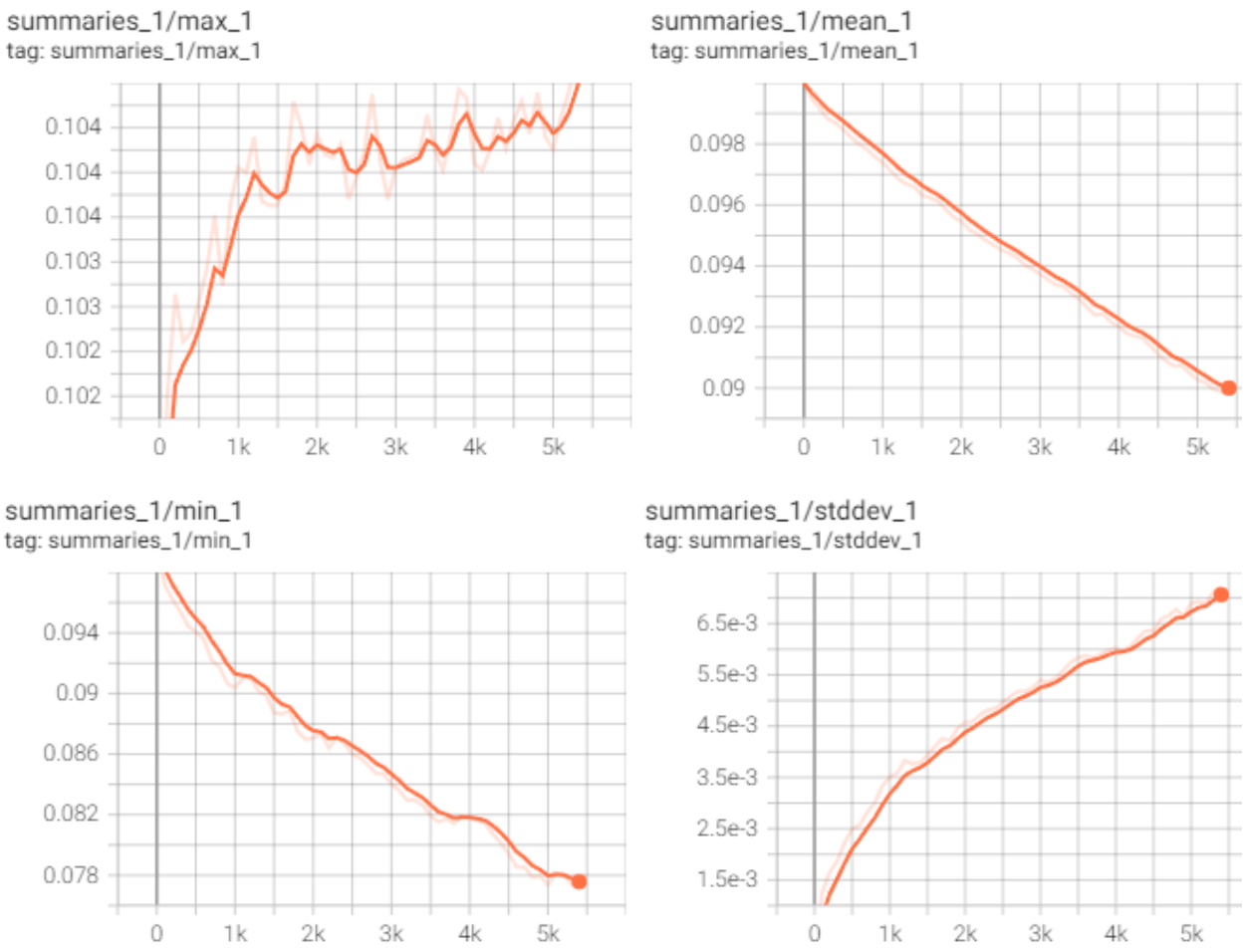


Figure 2_41: b_conv1

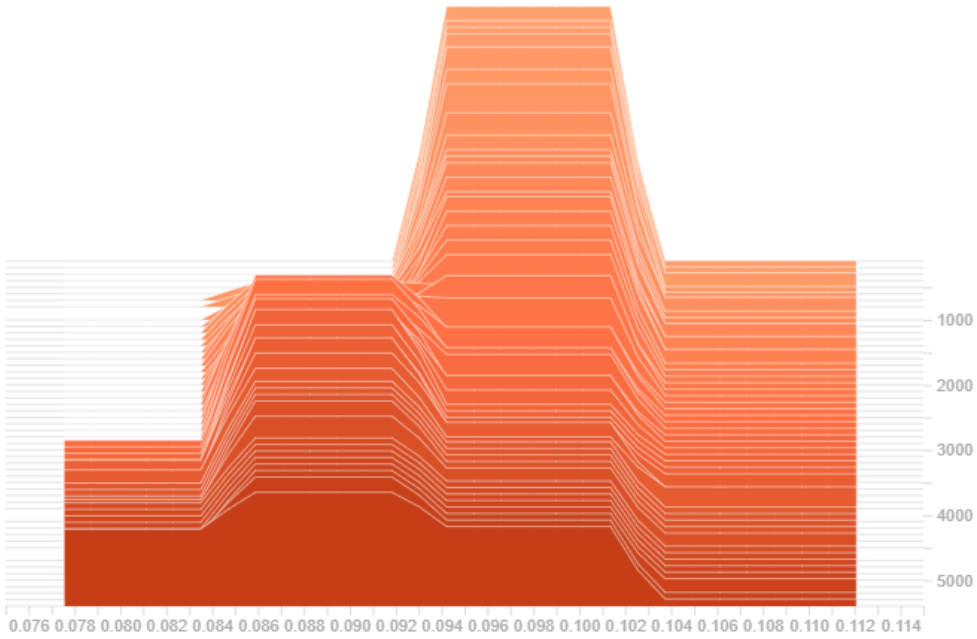


Figure 2_42: b_conv1_histogram

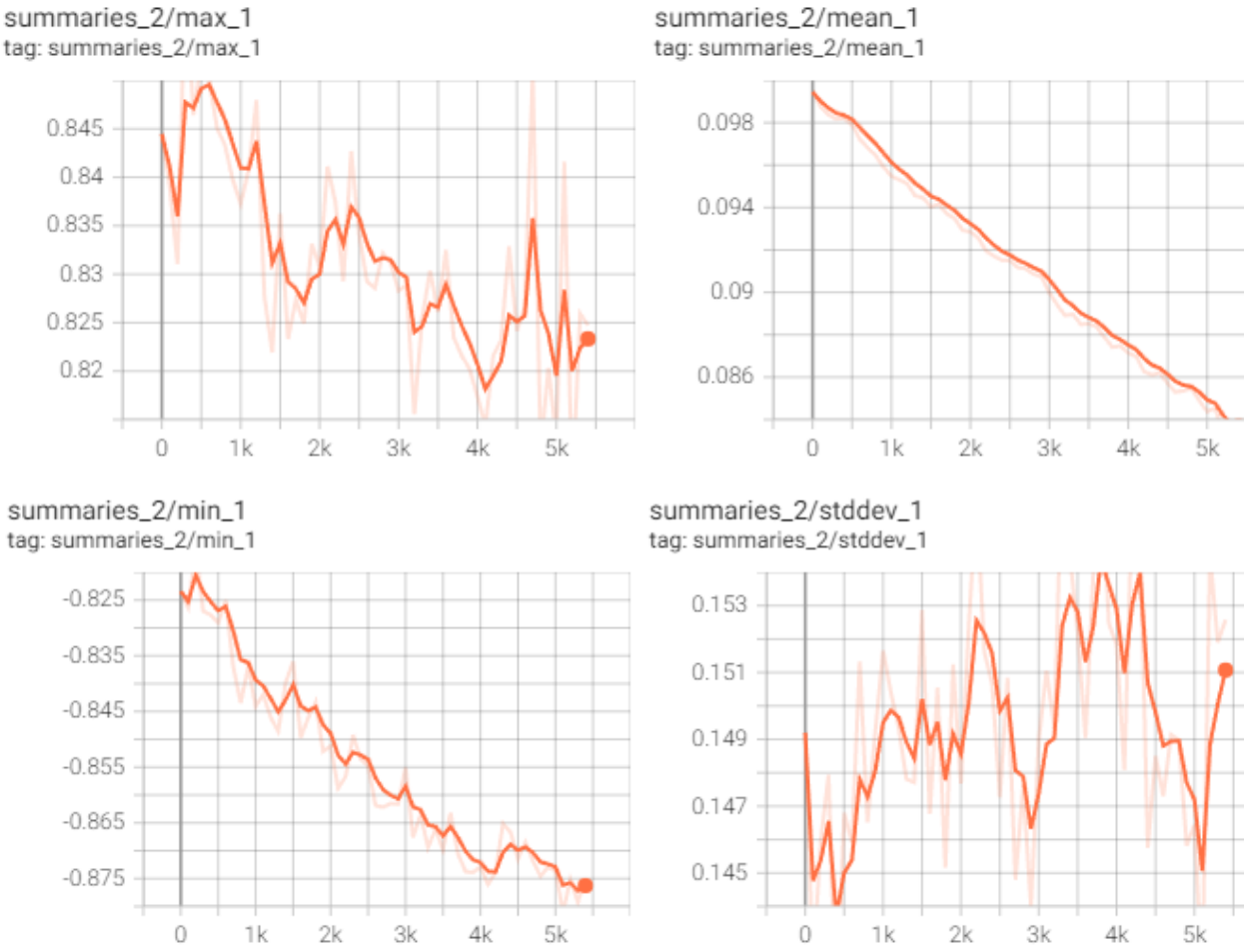


Figure 2_43: h_conv1

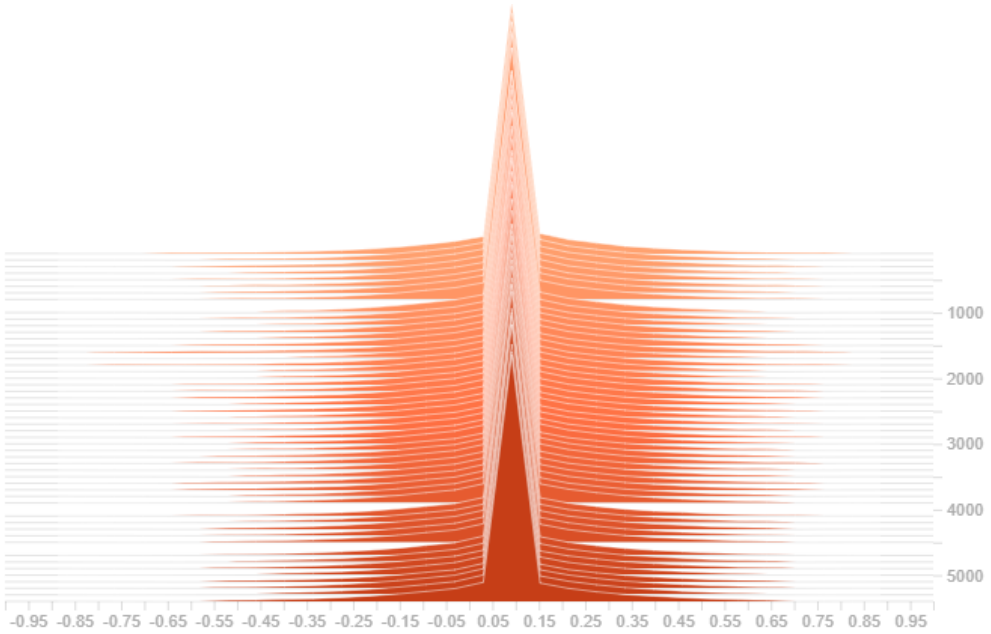


Figure 2_44: h_conv1_histogram

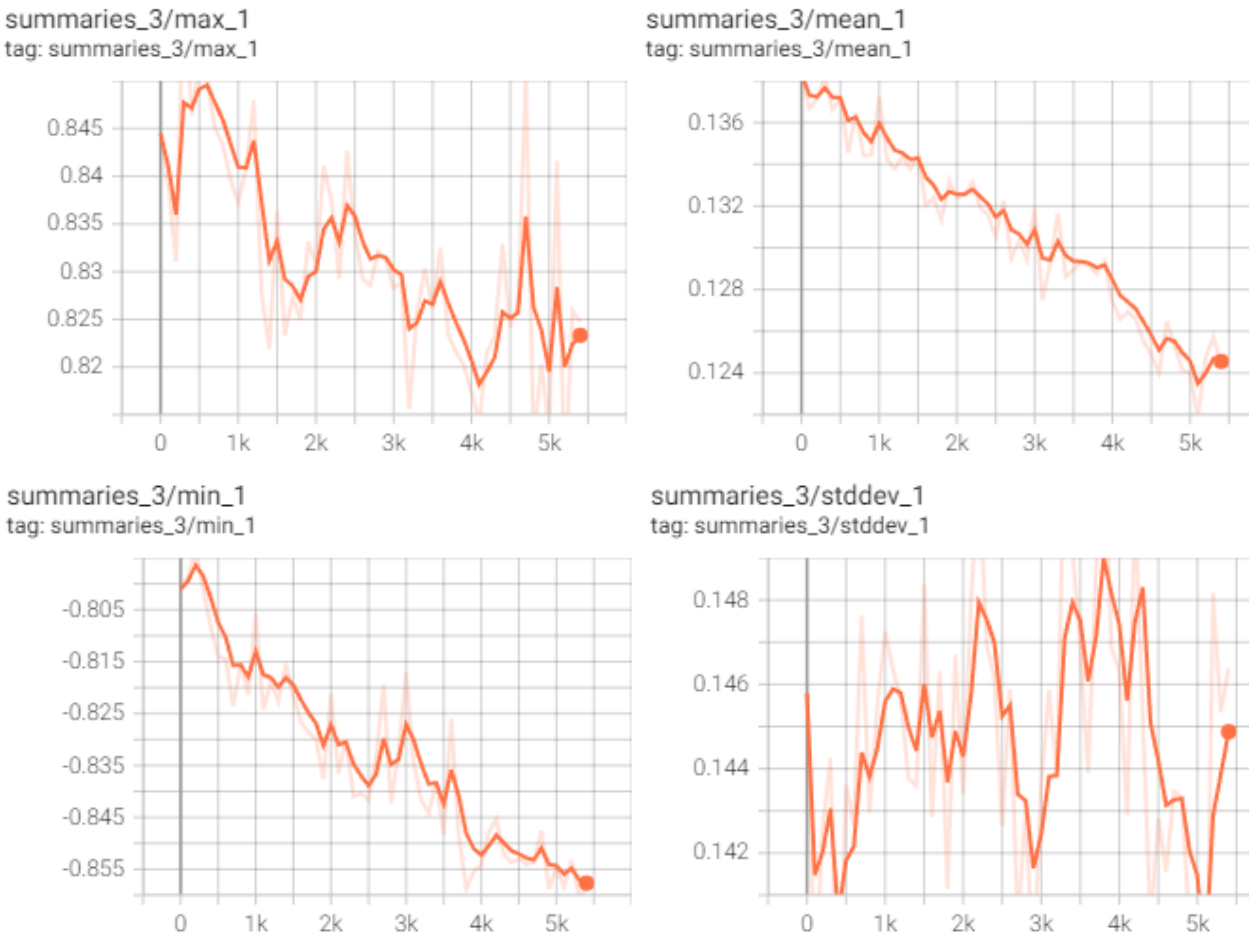


Figure 2_45: h_pool1

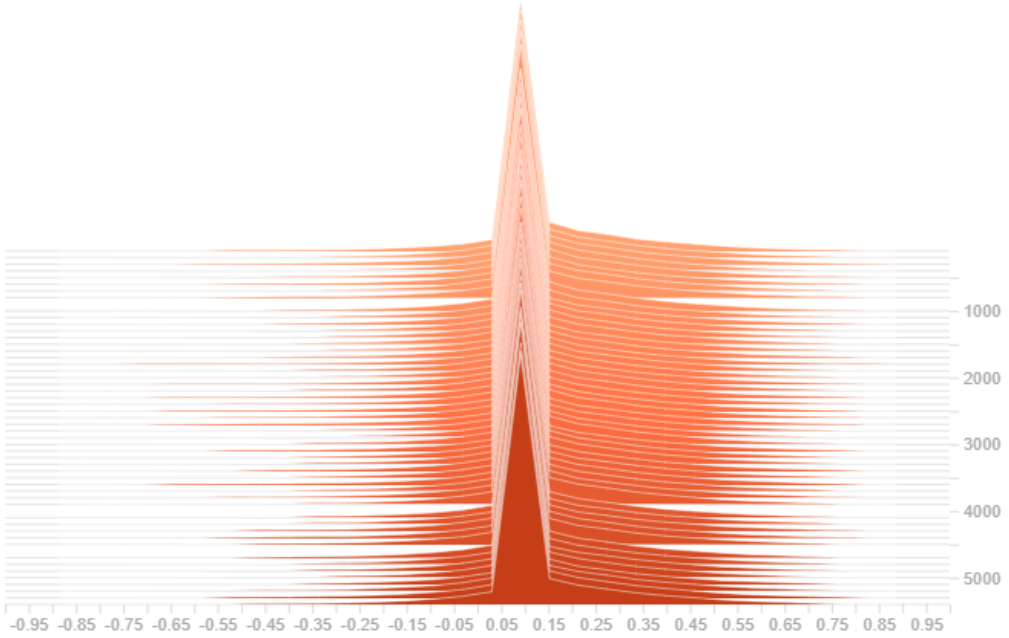


Figure 2_46: h_pool1_histogram

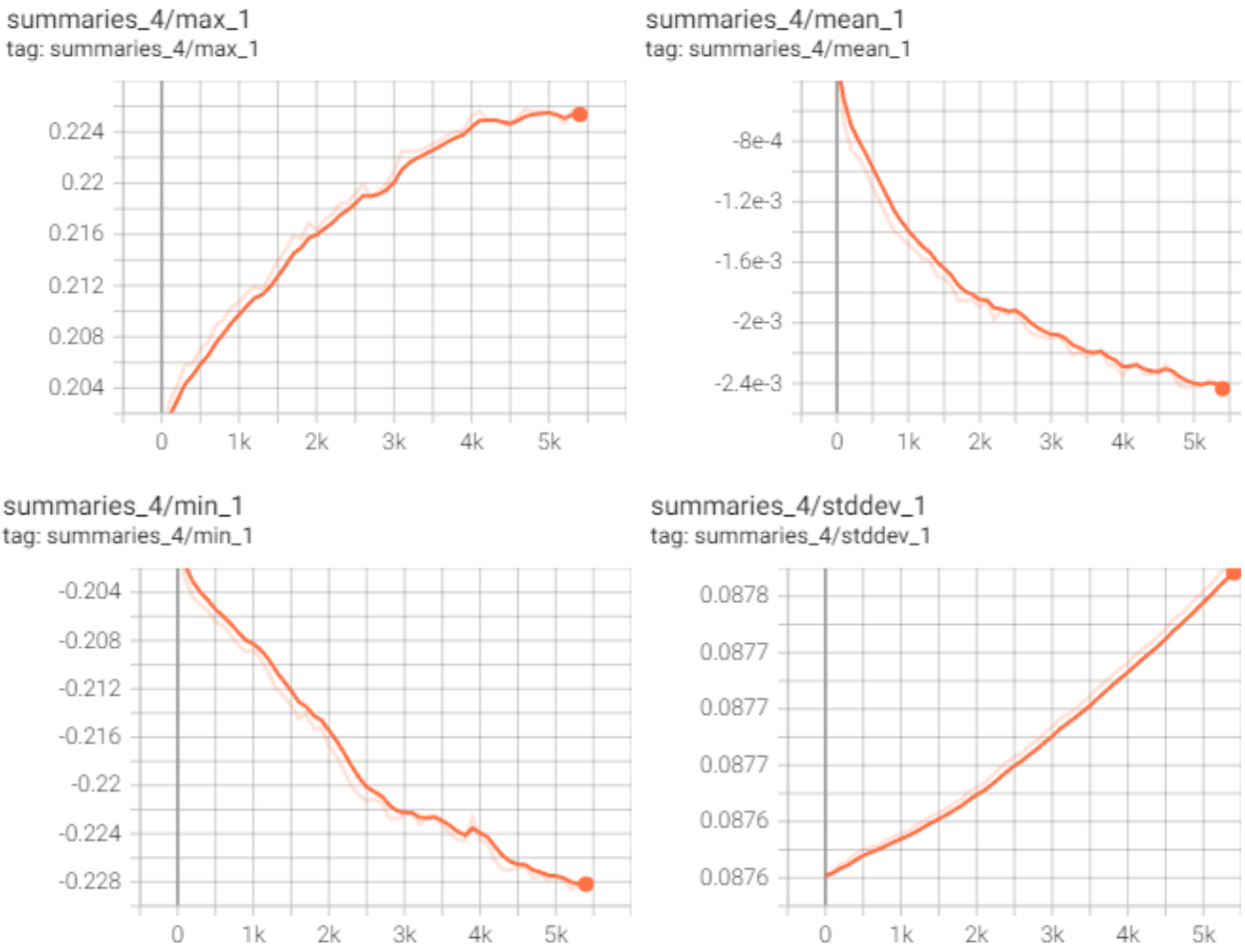


Figure 2_47: W_conv2

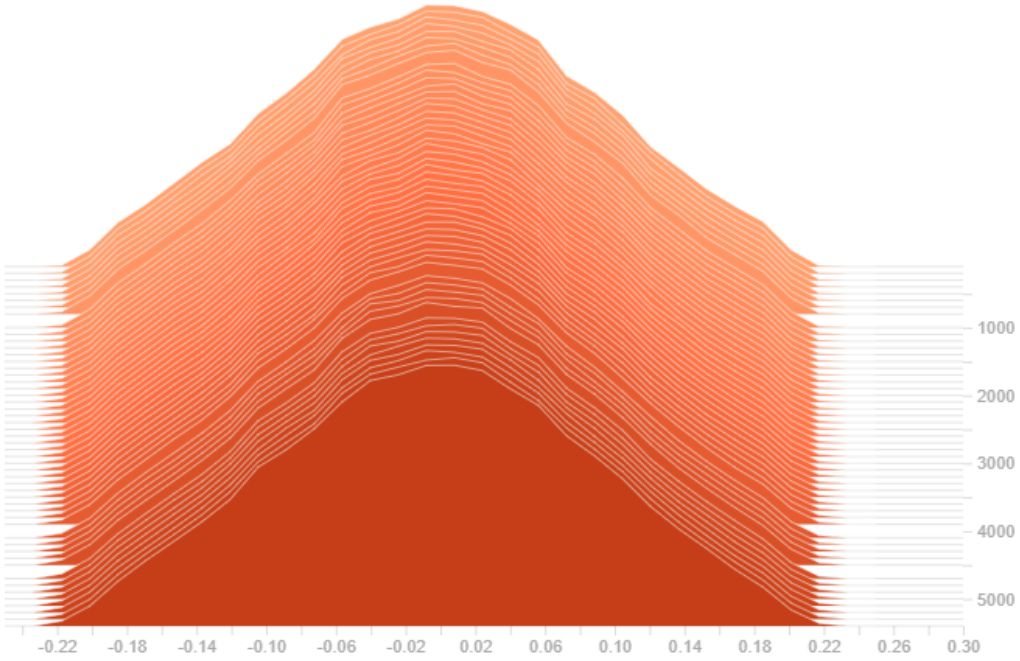


Figure 2_48: W_conv2_histogram

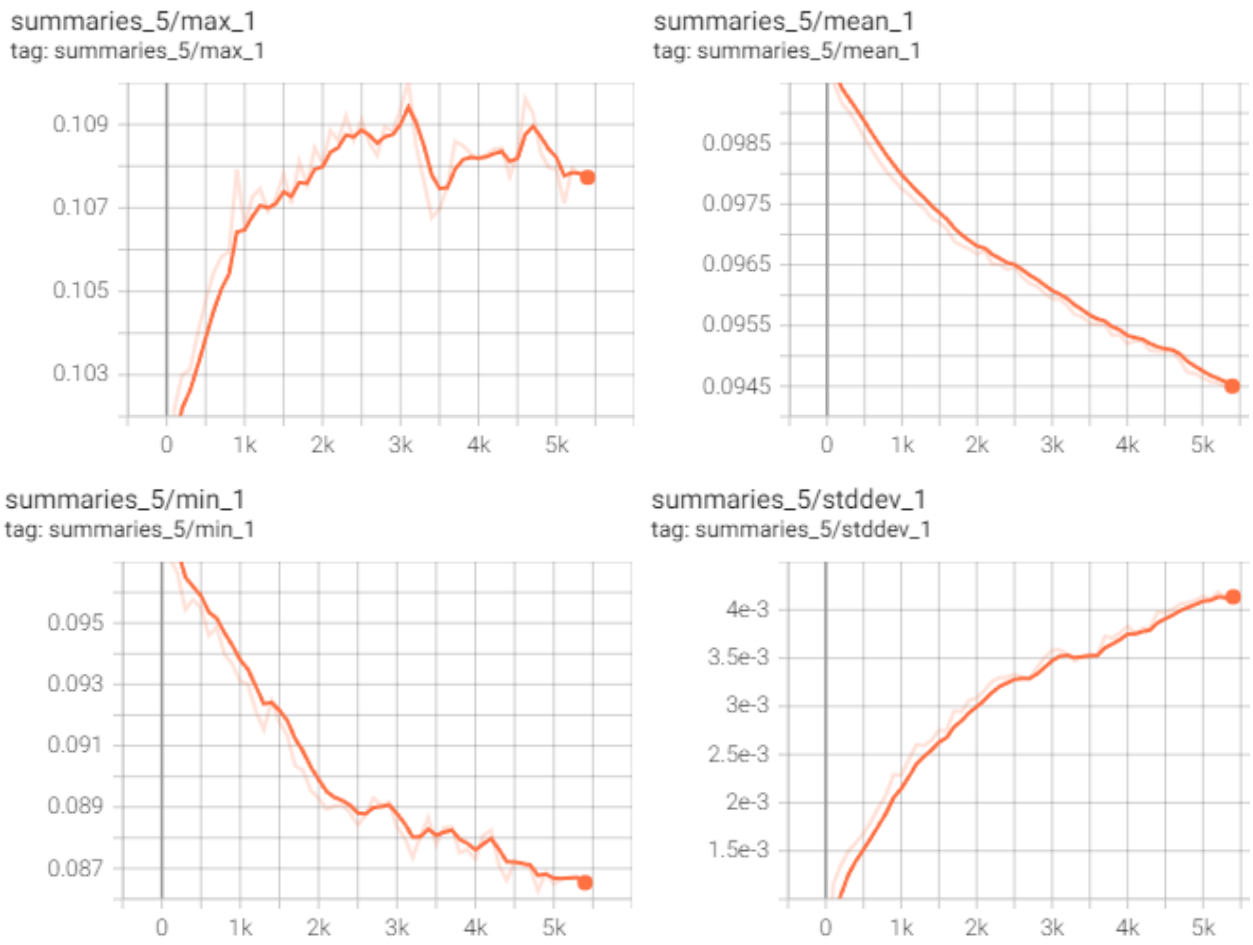


Figure 2_49: b_conv2

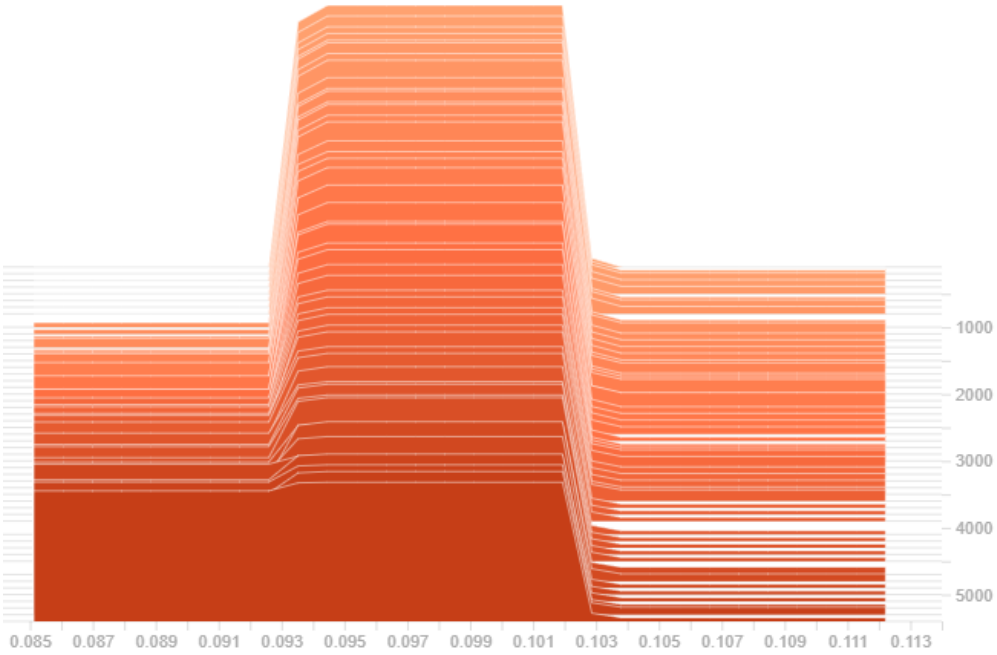


Figure 2_50: b_conv2_histogram