RICE UNIVERSITY, DEPARTMENT OF COMPUTER SCIENCE

Assignment 1, COMP 576

Chen Zeng(cz39)

September 21, 2018

1.a

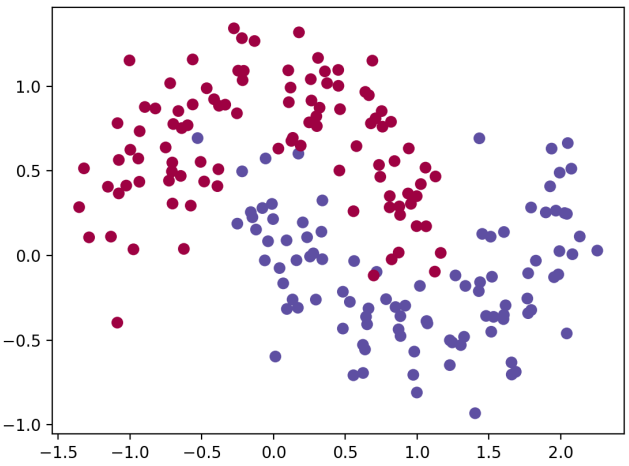

Figure 0.1: generate_data

1.b.1

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$sigmoid(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$

$$relu(x) = max(0, x)$$

```
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

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

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

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

1.b.3

```
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

```
self.z1 = np.dot(self.W1, X) + self.b1
self.a1 = actFun(self.z1)
self.z2 = np.dot(self.W2, self.a1) + self.b2
```

1.c.2

```
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)
```

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

```
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)
```

1.e.2: change type

Tanh

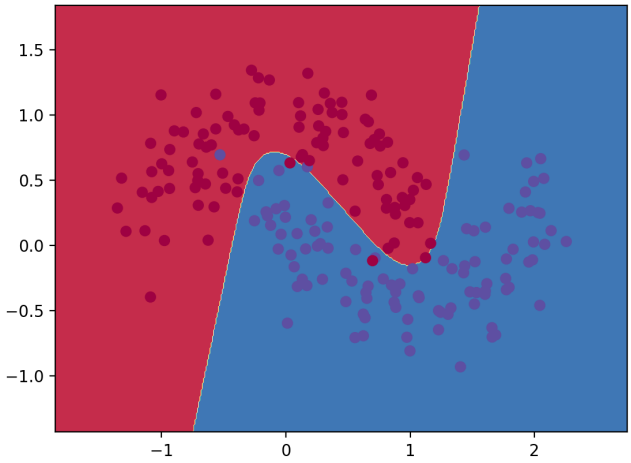

Figure 0.2: tanh

```
Loss after iteration 0: 0.432387
Loss after iteration 1000: 0.068947
Loss after iteration 2000: 0.069197
Loss after iteration 3000: 0.071218
Loss after iteration 4000: 0.071253
Loss after iteration 5000: 0.071278
Loss after iteration 6000: 0.071293
Loss after iteration 7000: 0.071303
Loss after iteration 8000: 0.071308
Loss after iteration 9000: 0.071312
Loss after iteration 10000: 0.071314
Loss after iteration 11000: 0.071315
Loss after iteration 12000: 0.071315
Loss after iteration 13000: 0.071316
Loss after iteration 14000: 0.071316
Loss after iteration 15000: 0.071316
Loss after iteration 16000: 0.071316
Loss after iteration 17000: 0.071316
Loss after iteration 18000: 0.071316
Loss after iteration 19000: 0.071316
```

Sigmoid

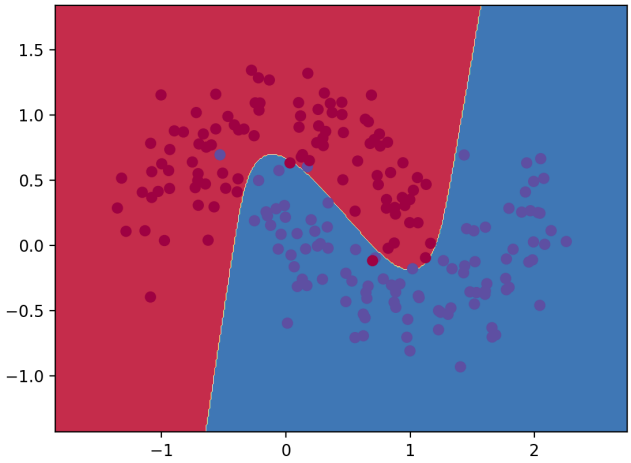

Figure 0.3: sigmoid

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

ReLu

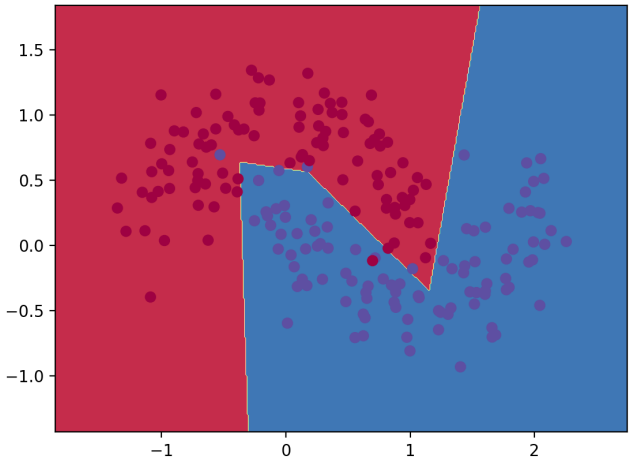

Figure 0.4: relu

```
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
```

Observation and Explain: ReLu performs better than Tanh and Sigmoid, and the later two perform very similar. The reason is that the ReLu can make some of the output to be zero and this can lead to Scale-invariant. (reference: https://en.wikipedia.org/wiki/Rectifier_(neural_networks))

1.e.2: change nn_hidden_dim

type=tanh, nn_hidden_dim=1

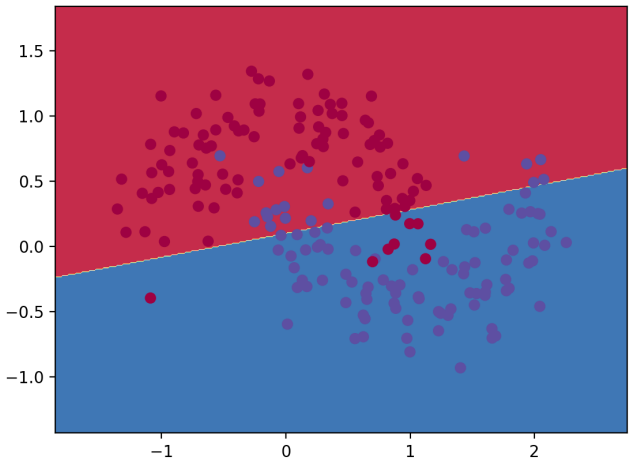

Figure 0.5: nn_hidden_dim=1

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

type=tanh, nn_hidden_dim=2

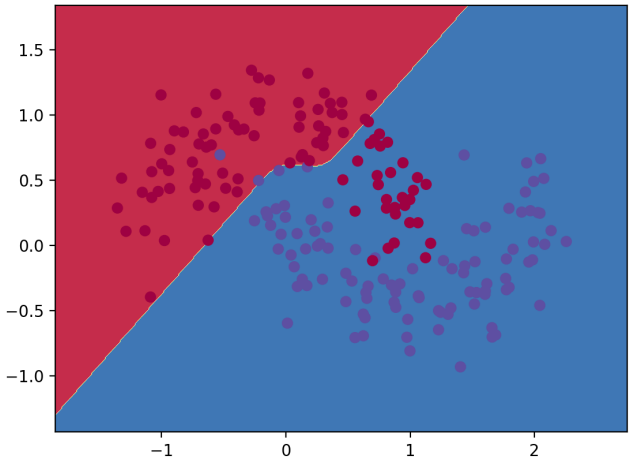

Figure 0.6: nn_hidden_dim=2

```
Loss after iteration 0: 0.546544
Loss after iteration 1000: 0.323354
Loss after iteration 2000: 0.320264
Loss after iteration 3000: 0.320643
Loss after iteration 4000: 0.322727
Loss after iteration 5000: 0.326986
Loss after iteration 6000: 0.275747
Loss after iteration 7000: 0.290241
Loss after iteration 8000: 0.322264
Loss after iteration 9000: 0.325693
Loss after iteration 10000: 0.319581
Loss after iteration 11000: 0.318402
Loss after iteration 12000: 0.324786
Loss after iteration 13000: 0.328108
Loss after iteration 14000: 0.233670
Loss after iteration 15000: 0.324291
Loss after iteration 16000: 0.301067
Loss after iteration 17000: 0.287431
Loss after iteration 18000: 0.319567
Loss after iteration 19000: 0.323060
```

type=tanh, nn_hidden_dim=3

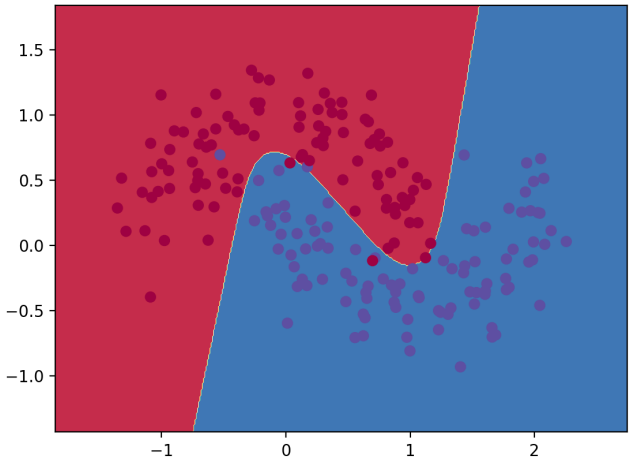

Figure 0.7: nn_hidden_dim=3

```
Loss after iteration 0: 0.432387
Loss after iteration 1000: 0.068947
Loss after iteration 2000: 0.069197
Loss after iteration 3000: 0.071218
Loss after iteration 4000: 0.071253
Loss after iteration 5000: 0.071278
Loss after iteration 6000: 0.071293
Loss after iteration 7000: 0.071303
Loss after iteration 8000: 0.071308
Loss after iteration 9000: 0.071312
Loss after iteration 10000: 0.071314
Loss after iteration 11000: 0.071315
Loss after iteration 12000: 0.071315
Loss after iteration 13000: 0.071316
Loss after iteration 14000: 0.071316
Loss after iteration 15000: 0.071316
Loss after iteration 16000: 0.071316
Loss after iteration 17000: 0.071316
Loss after iteration 18000: 0.071316
Loss after iteration 19000: 0.071316
```

type=tanh, nn_hidden_dim=4

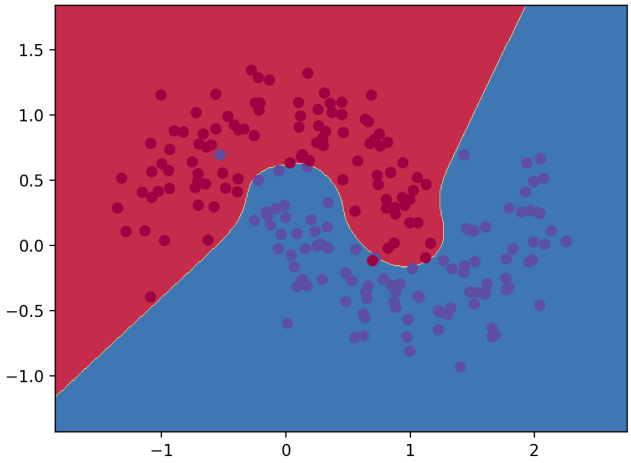

Figure 0.8: nn_hidden_dim=4

```
Loss after iteration 0: 0.466461
Loss after iteration 1000: 0.066685
Loss after iteration 2000: 0.060119
Loss after iteration 3000: 0.057102
Loss after iteration 4000: 0.055832
Loss after iteration 5000: 0.055171
Loss after iteration 6000: 0.054800
Loss after iteration 7000: 0.050578
Loss after iteration 8000: 0.050896
Loss after iteration 9000: 0.050591
Loss after iteration 10000: 0.052380
Loss after iteration 11000: 0.056980
Loss after iteration 12000: 0.050620
Loss after iteration 13000: 0.051948
Loss after iteration 14000: 0.064988
Loss after iteration 15000: 0.051433
Loss after iteration 16000: 0.050119
Loss after iteration 17000: 0.050264
Loss after iteration 18000: 0.071774
Loss after iteration 19000: 0.050135
```

type=tanh, nn_hidden_dim=5

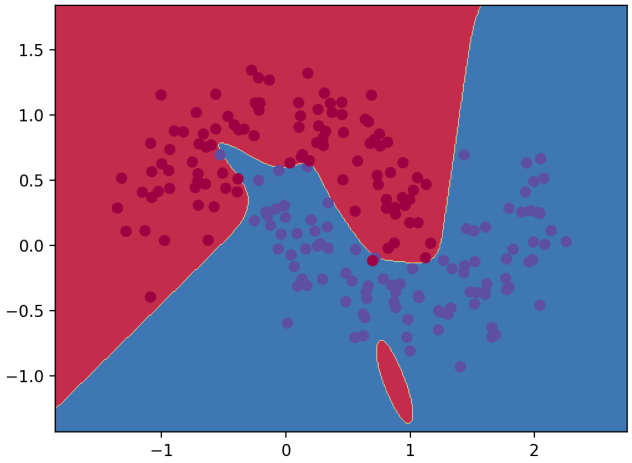

Figure 0.9: nn_hidden_dim=5

```
Loss after iteration 0: 0.613897
Loss after iteration 1000: 0.053880
Loss after iteration 2000: 0.044053
Loss after iteration 3000: 0.041367
Loss after iteration 4000: 0.040268
Loss after iteration 5000: 0.039732
Loss after iteration 6000: 0.039444
Loss after iteration 7000: 0.039280
Loss after iteration 8000: 0.039185
Loss after iteration 9000: 0.039128
Loss after iteration 10000: 0.039093
Loss after iteration 11000: 0.039072
Loss after iteration 12000: 0.039059
Loss after iteration 13000: 0.039050
Loss after iteration 14000: 0.039045
Loss after iteration 15000: 0.039042
Loss after iteration 16000: 0.039040
Loss after iteration 17000: 0.039038
Loss after iteration 18000: 0.039038
Loss after iteration 19000: 0.039037
```

Observation and Explain: When nn_hidden_dim=3, the performance is the best. Because when the value is too small, the model will unfit the data. And when the value is too big, the model will overfit the data.

1.f DeepNeuralNetwork implementation

In my implementation, I only implement the class of DeepNeuralNetwork and don't implement the class of Layer as mentioned in instruction. We can call the DeepNeuralNetwork like this:

```
model = DeepNeuralNetwork(nn_dims=[2, 3, 2, 4, 5], actFun_type='tanh')
```

It means the model is constructed by 5 layers with 2, 3, 2, 4, and 5 units on each layer from input to output. And the active function is tanh.
Here are the detailed implementation on key functions:
function feedforward:

```
    def feedforward(self, X, actFun):
        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):
        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)
        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):
    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):
    # 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 for using the whole dataset, not 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:

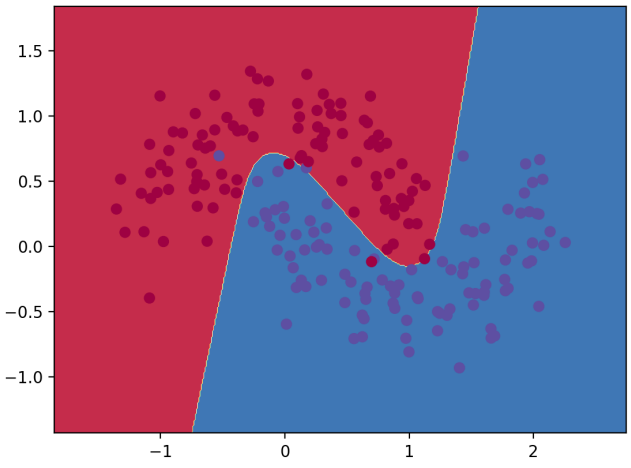

Figure 0.10: number of layers = 3

When we set the layer as:

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

The result is:

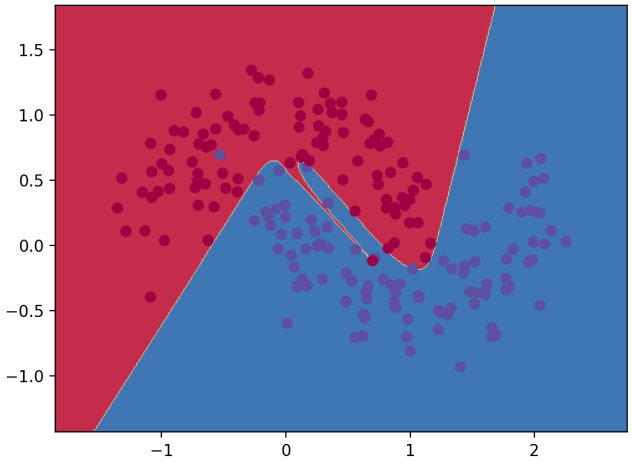

Figure 0.11: number of layers = 4

When we set the layer as:

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

The result is:

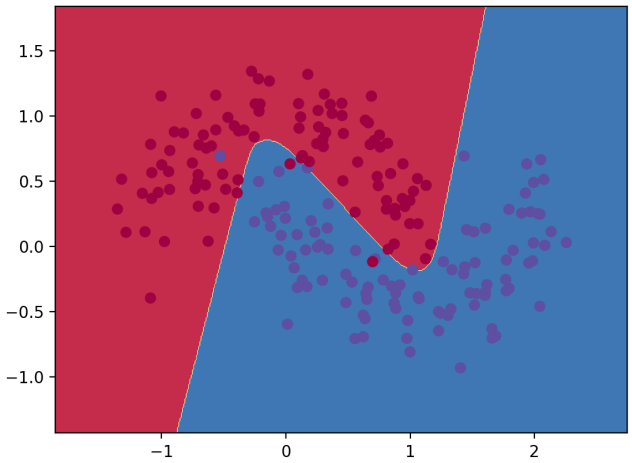

Figure 0.12: number of layers = 5

Observation and Explain: When the number of layers is 3 or 5, the performance is the better than the number of layers to be 4. The the layers of 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)
```

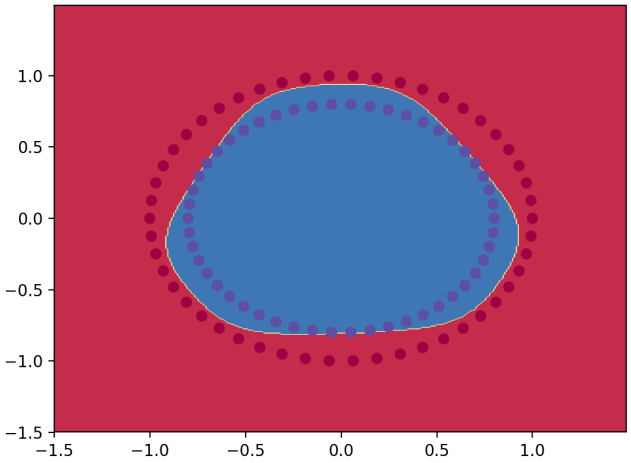

Figure 0.13: dataset=make_circles

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

2.a.2

Reference from the official GitHub repo of tensorflow:
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/mnist_deep.py

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

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

def conv2d(x, W):
```

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

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

2.a.3

Reference from the official GitHub repo of tensorflow:
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/mnist_deep.py

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

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

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

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

    # densely connected layer
    W_fc1 = weight_variable([7 * 7 * 64, 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
```

2.a.4

Reference from the official GitHub repo of tensorflow:
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/mnist_deep.py

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

2.a.5

To run this codes, multiple parts of codes are modified.
First of all, following the list here: https://stackoverflow.com/questions/41066244/tensorflow-module-object-has-no-attribute-scalar-summary. Many of the module should be renamed following the most updated instructions.
Secondly, the way to load the mnist dataset need to be modified.

```
import argparse
FLAGS = None
parser = argparse.ArgumentParser()
parser.add_argument('--data_dir', type=str,
                    default='/tmp/tensorflow/mnist/input_data',
                    help='Directory for storing input data')
FLAGS, unparsed = parser.parse_known_args()
mnist = input_data.read_data_sets(FLAGS.data_dir)
```

Here is the final output results:

```
step 0, training accuracy 0.06
step 100, training accuracy 0.9
step 200, training accuracy 0.86
step 300, training accuracy 0.86
step 400, training accuracy 0.94
step 500, training accuracy 0.94
step 600, training accuracy 0.94
step 700, training accuracy 0.94
step 800, training accuracy 1
step 900, training accuracy 0.94
step 1000, training accuracy 0.98
step 1100, training accuracy 1
```

```
step 1200, training accuracy 0.98
step 1300, training accuracy 1
step 1400, training accuracy 1
step 1500, training accuracy 1
step 1600, training accuracy 0.98
step 1700, training accuracy 0.98
step 1800, training accuracy 1
step 1900, training accuracy 0.98
step 2000, training accuracy 1
step 2100, training accuracy 0.96
step 2200, training accuracy 1
step 2300, training accuracy 0.98
step 2400, training accuracy 0.98
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 1
step 3000, training accuracy 0.96
step 3100, training accuracy 1
step 3200, training accuracy 0.98
step 3300, training accuracy 1
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 0.98
step 4000, training accuracy 0.96
step 4100, training accuracy 1
step 4200, training accuracy 0.98
step 4300, training accuracy 0.98
step 4400, training accuracy 1
step 4500, training accuracy 0.94
step 4600, training accuracy 0.96
step 4700, training accuracy 0.98
step 4800, training accuracy 1
step 4900, training accuracy 1
step 5000, training accuracy 1
step 5100, training accuracy 0.98
step 5200, training accuracy 1
step 5300, training accuracy 1
step 5400, training accuracy 0.98
test accuracy 0.9869
```

```
The training takes 593.941379 second to finish
```

2.a.6

To run the TensorBoard, I slightly modify the command comparing with the instructions:

```
tensorboard --logdir=results/ --host localhost --port 8088
```

Then we can open the console at:

```
http://localhost:8088
```

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

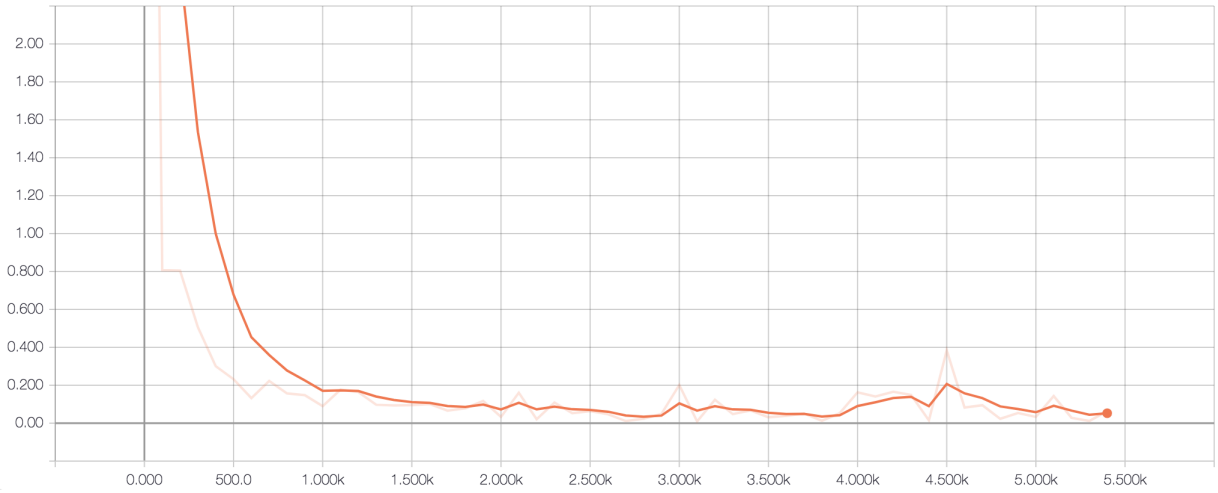

Figure 0.14: scalars

The figure of graphs:

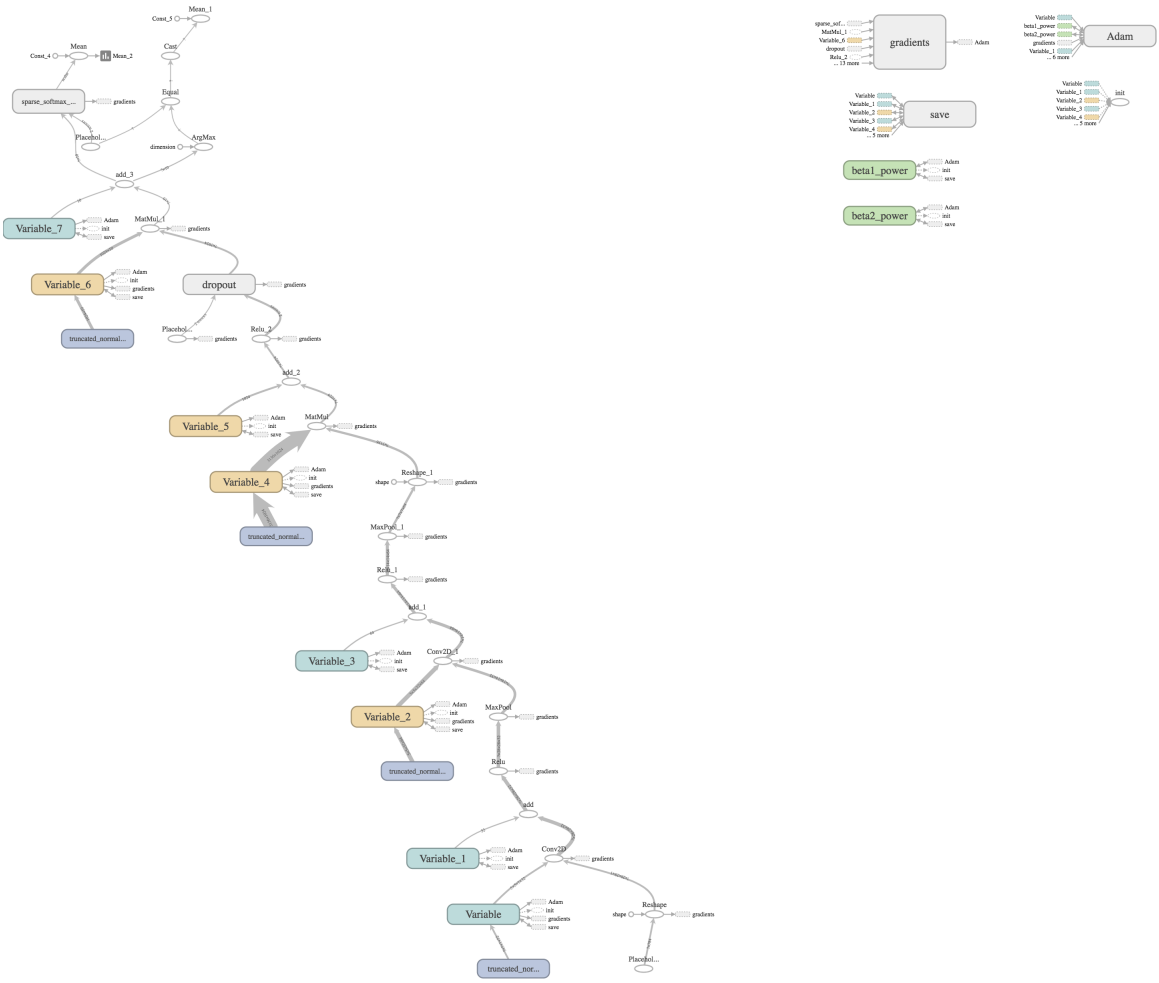

Figure 0.15: graphs

The figure of projector:

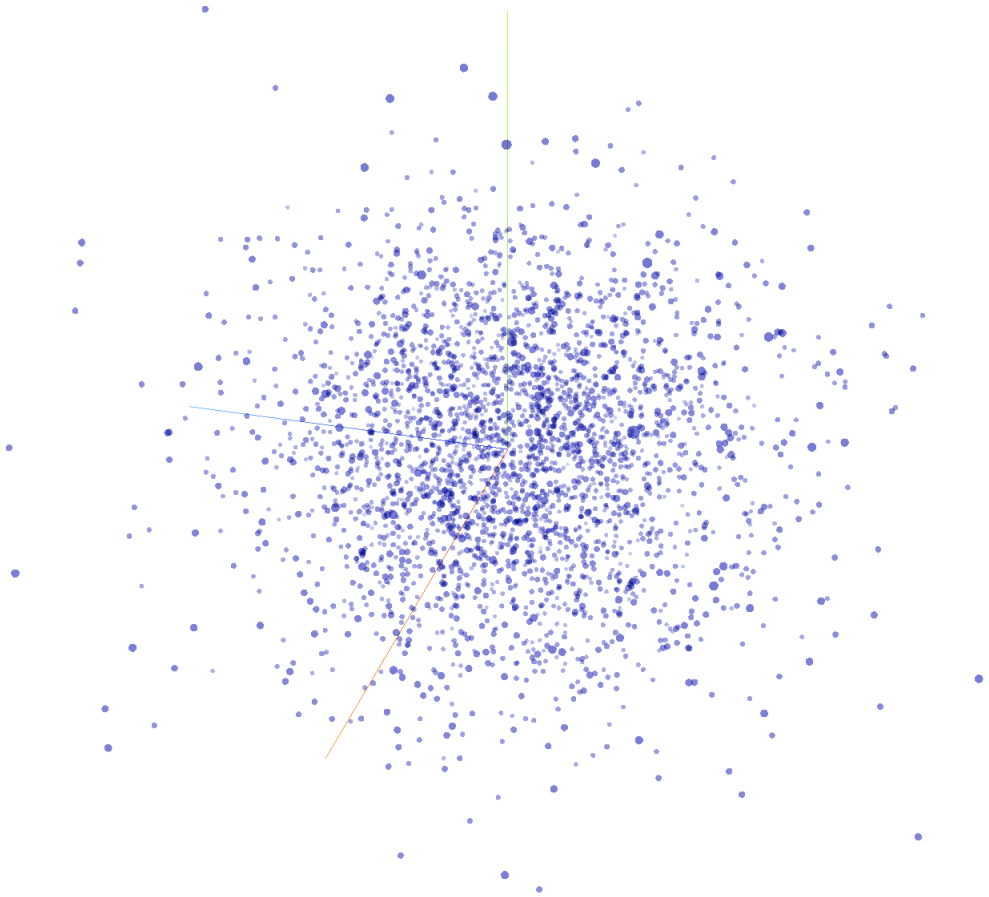

Figure 0.16: projector

2.b

Write function variable_summaries for plotting variables:

```
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)
```

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:

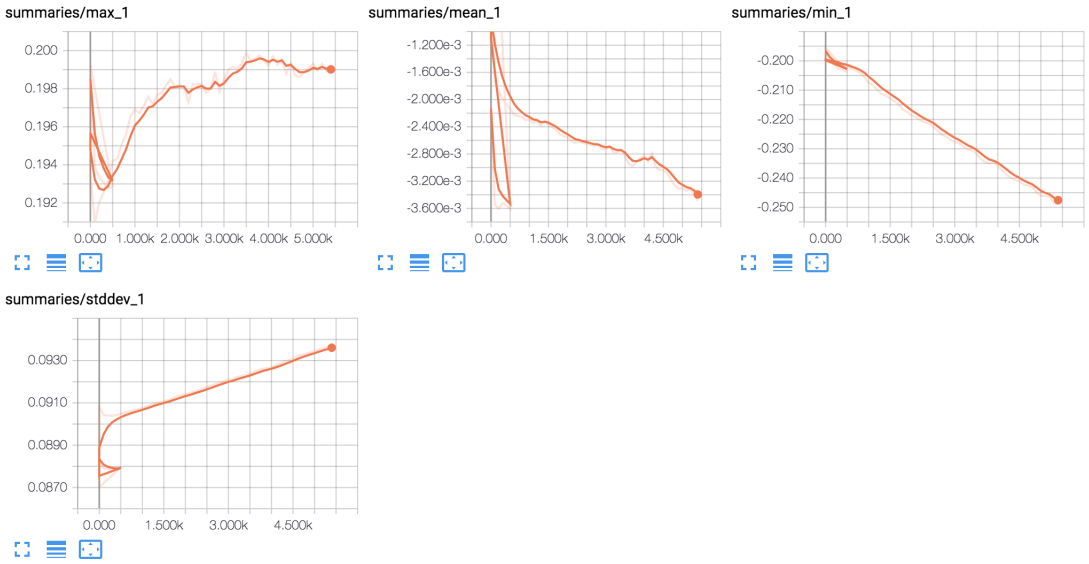

Figure 0.17: W_conv1

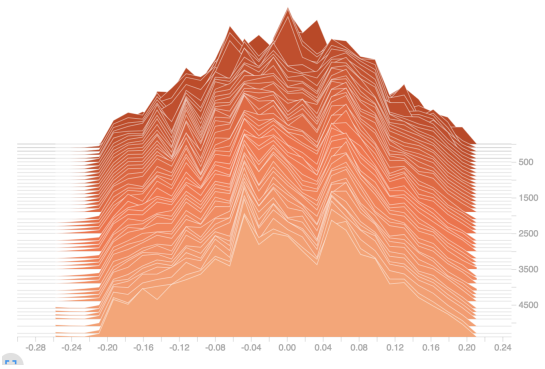

Figure 0.18: W_conv1_histogram

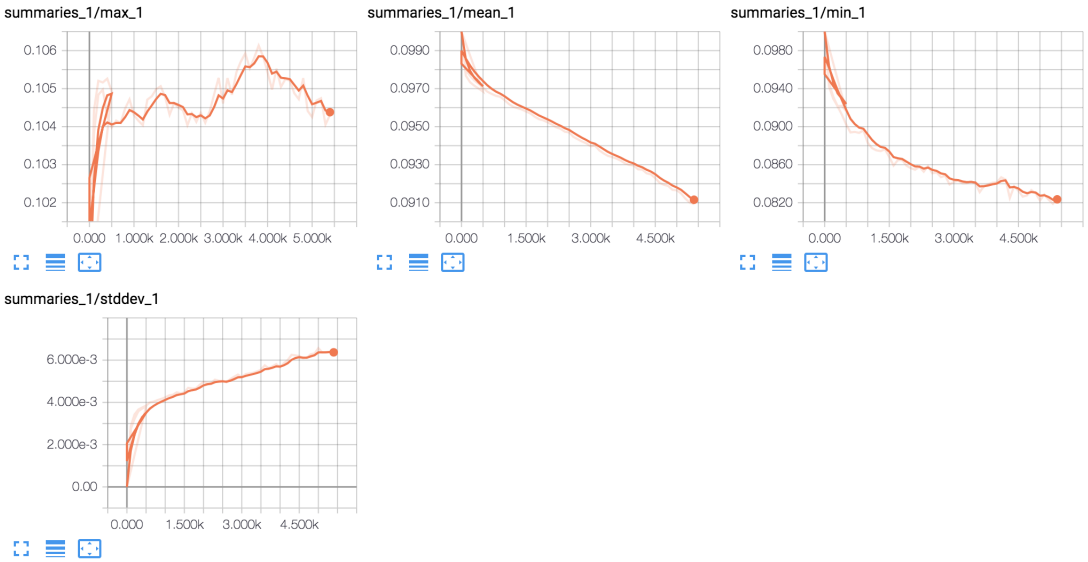

Figure 0.19: b_conv1

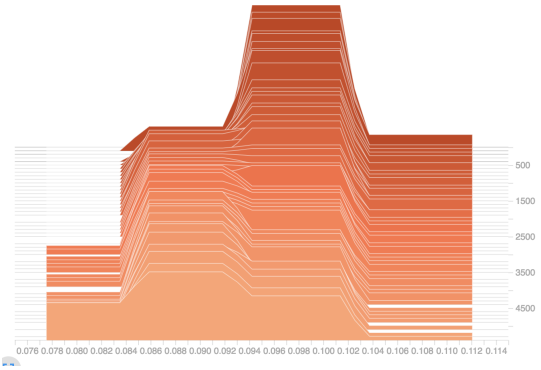

Figure 0.20: b_conv1_histogram

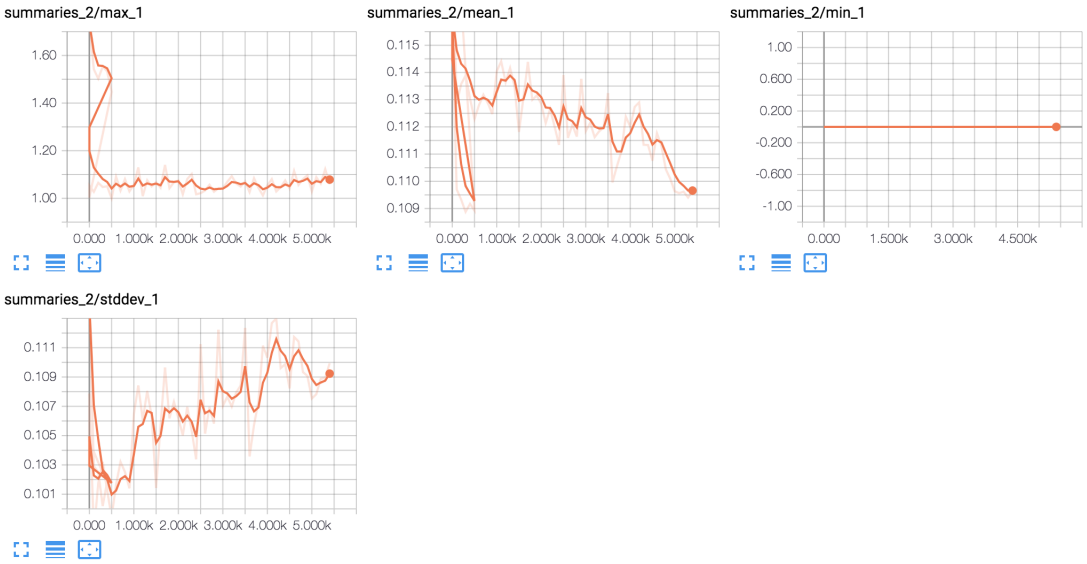

Figure 0.21: h_conv1

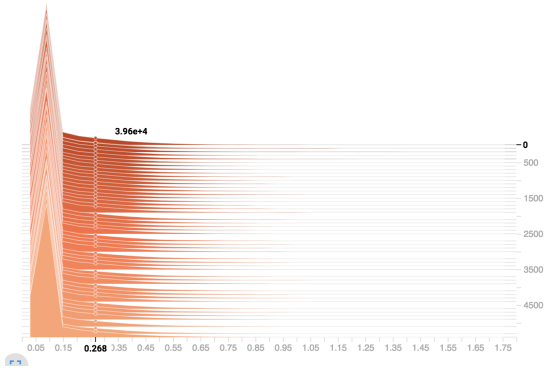

Figure 0.22: h_conv1_histogram

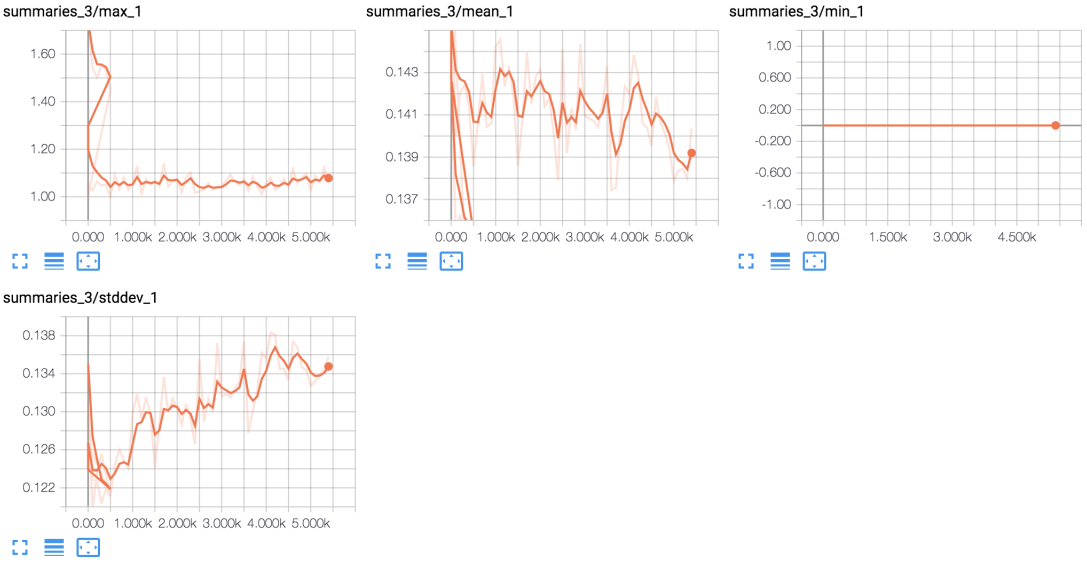

Figure 0.23: h_pool1

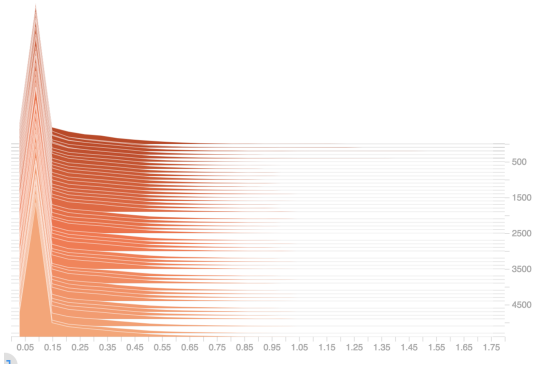

Figure 0.24: h_pool1_histogram

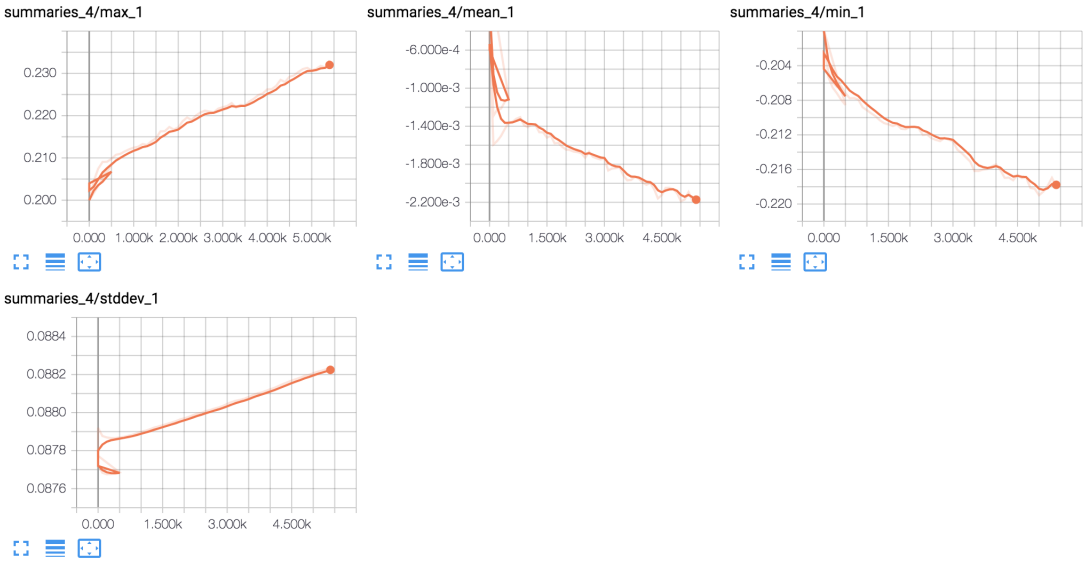

Figure 0.25: W_conv2

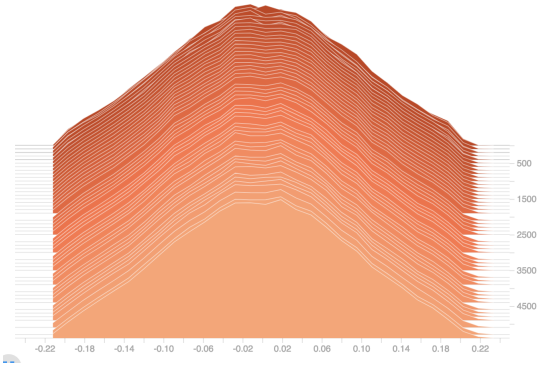

Figure 0.26: W_conv2_histogram

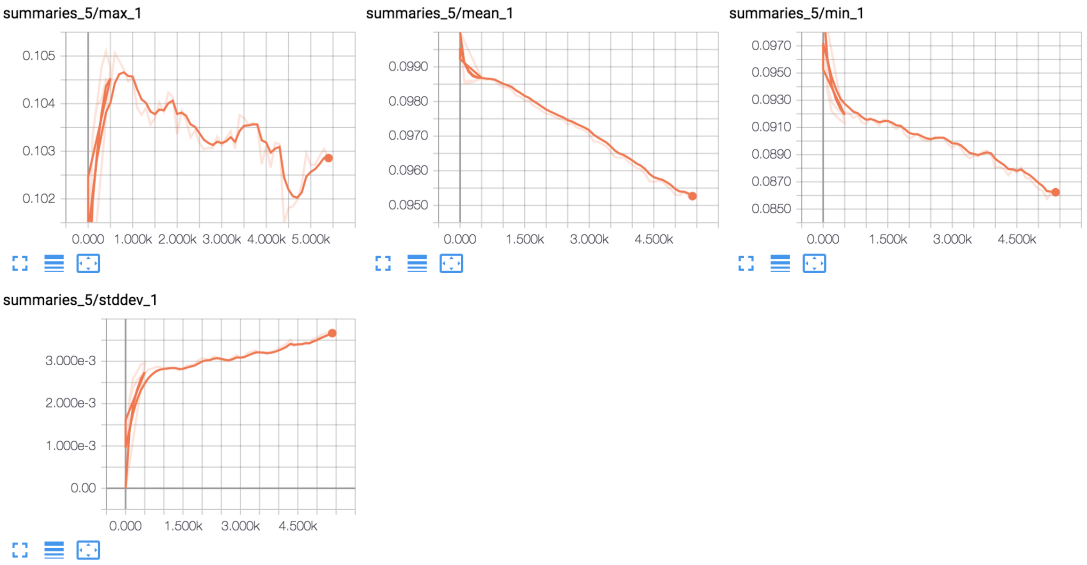

Figure 0.27: b_conv2

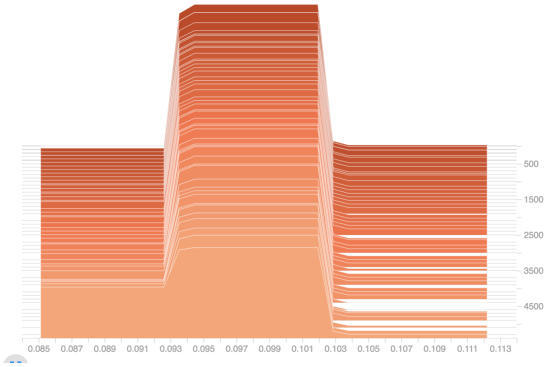

Figure 0.28: b_conv2_histogram

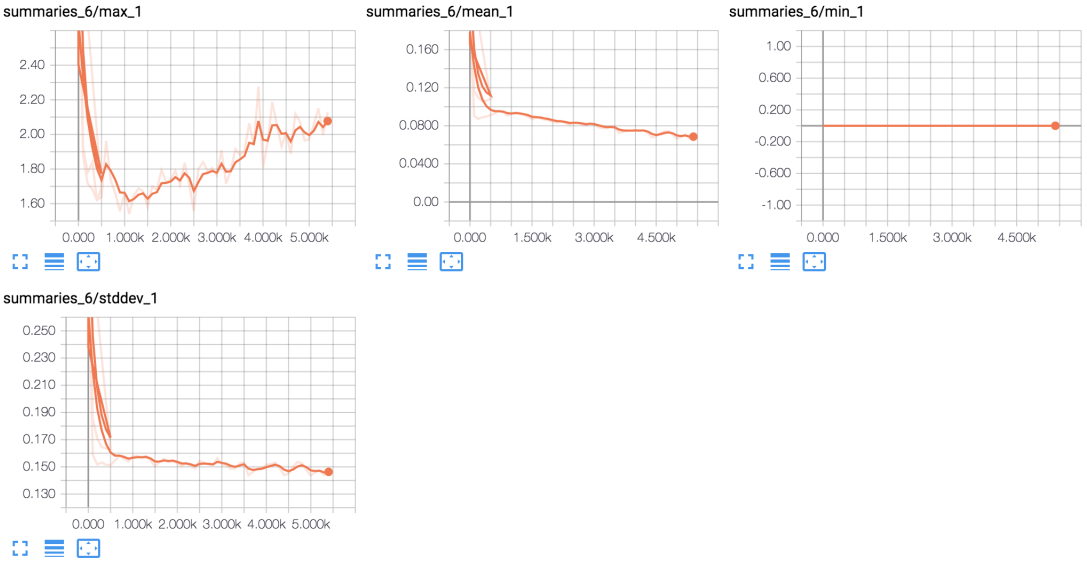

Figure 0.29: h_conv2

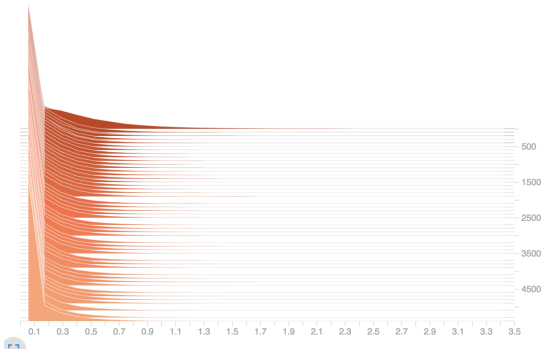

Figure 0.30: h_conv2_histogram

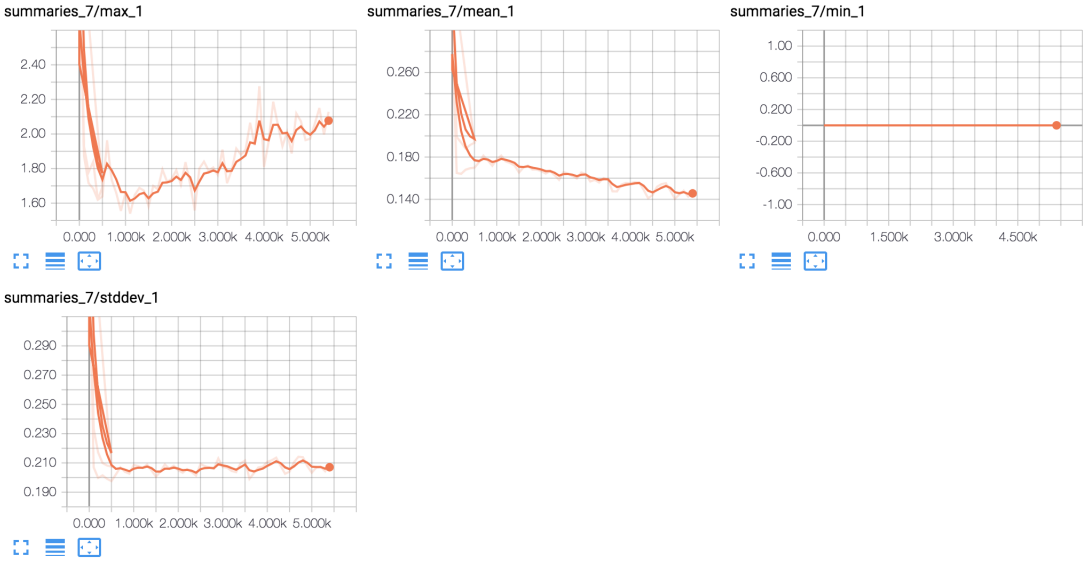

Figure 0.31: h_pool2

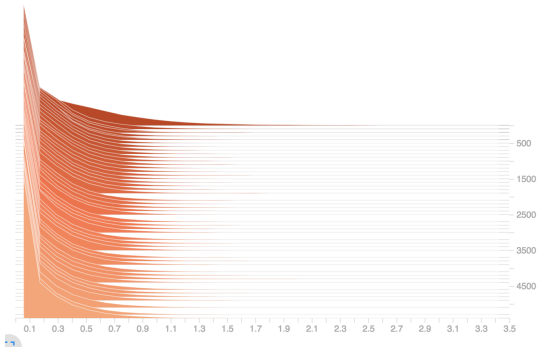

Figure 0.32: h_pool2_histogram

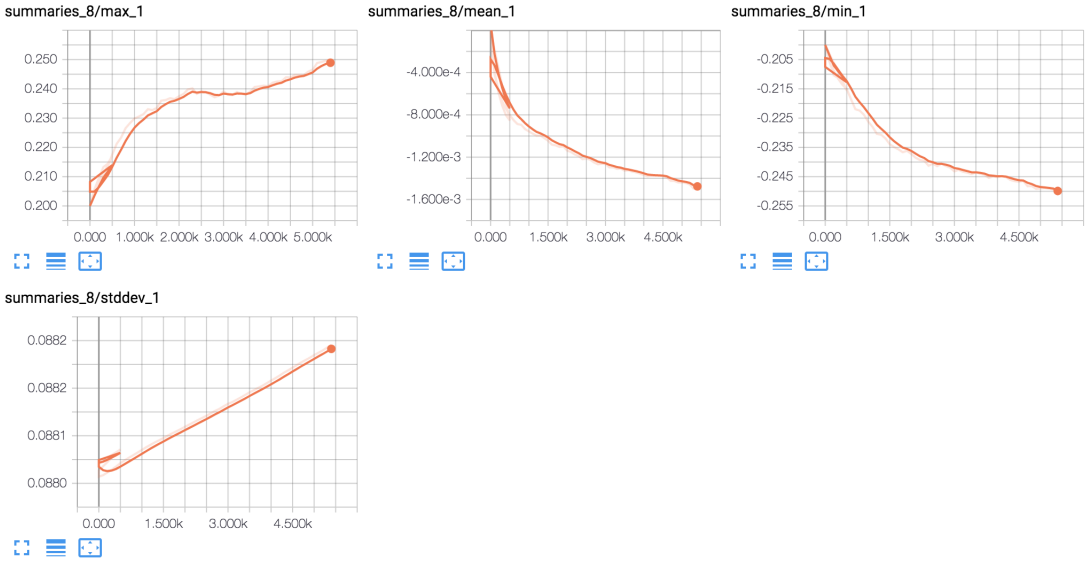

Figure 0.33: W_fc1

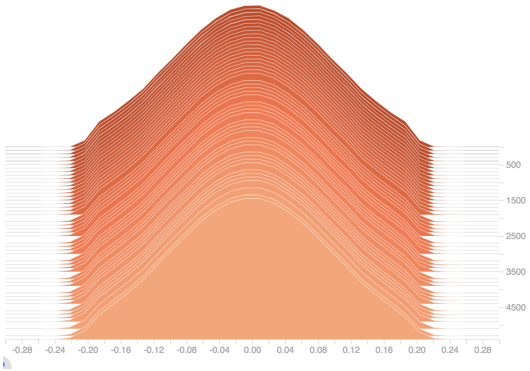

Figure 0.34: W_fc1_histogram

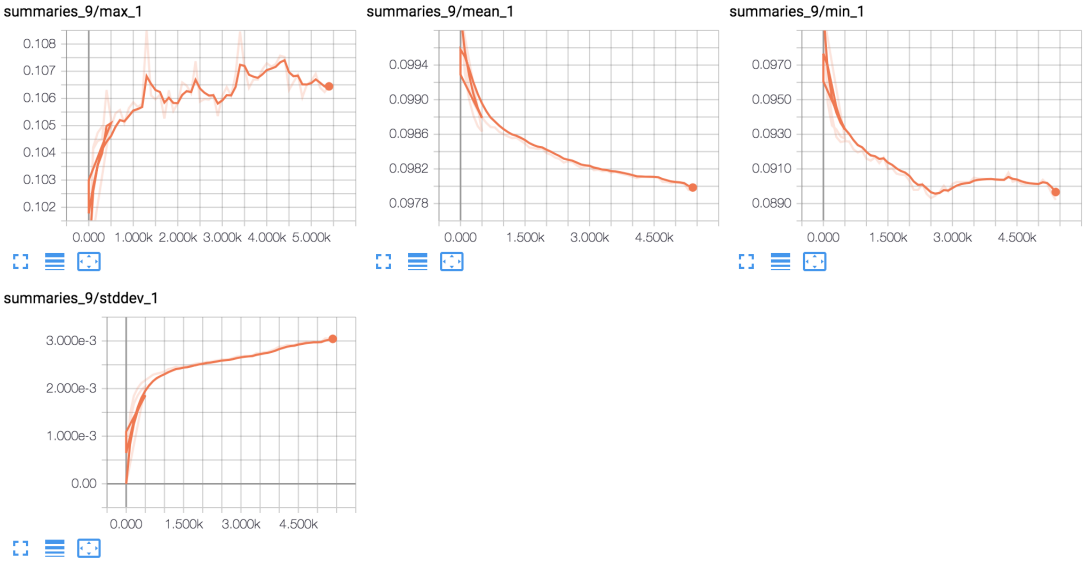

Figure 0.35: b_fc1

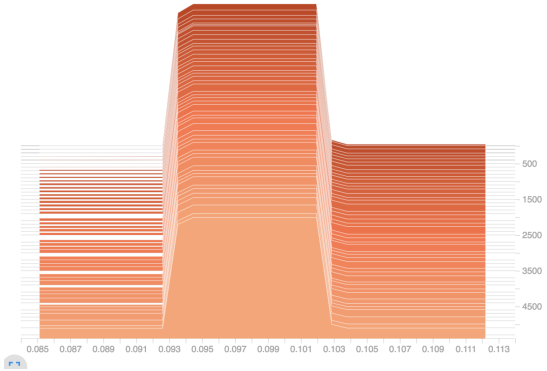

Figure 0.36: b_fc1_histogram

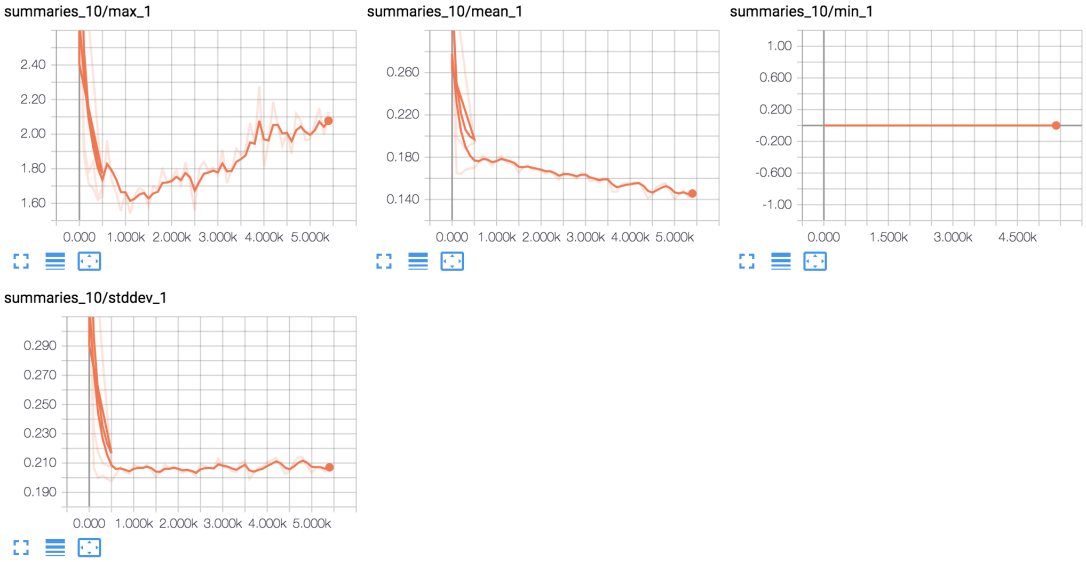

Figure 0.37: h_pool2_flat

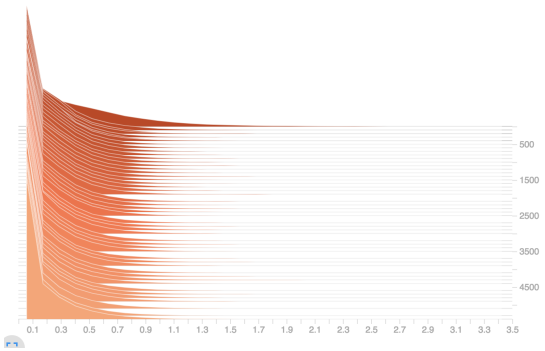

Figure 0.38: h_pool2_flat_histogram

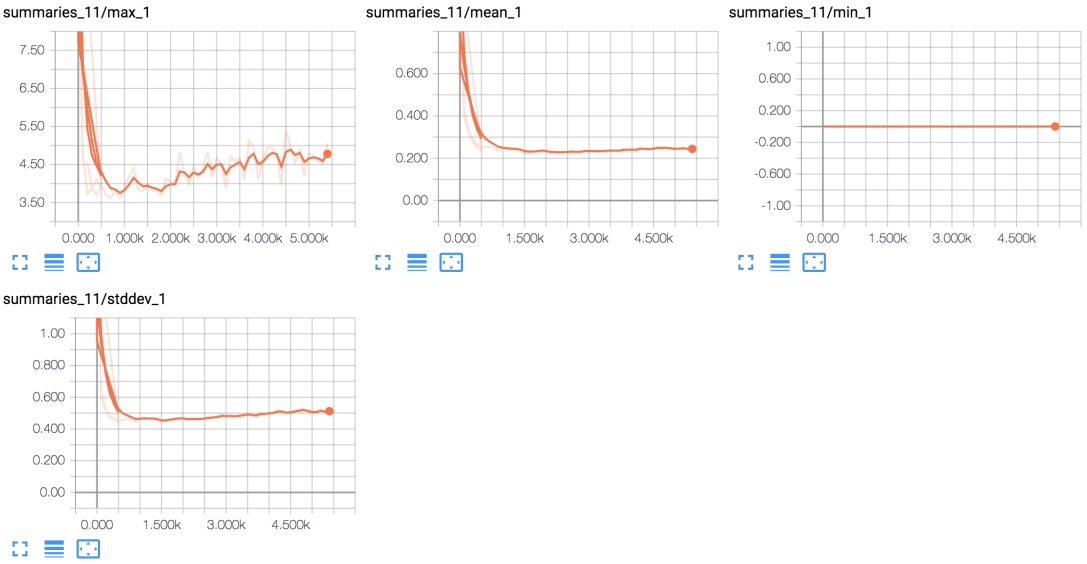

Figure 0.39: h_fc1

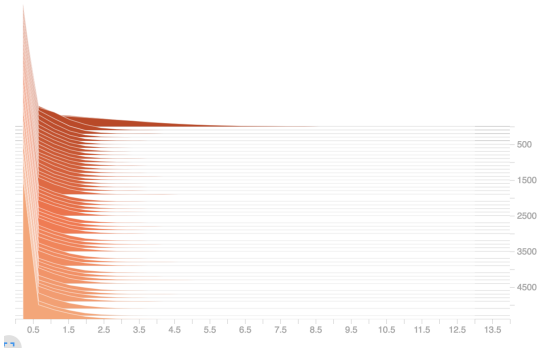

Figure 0.40: h_fc1_histogram

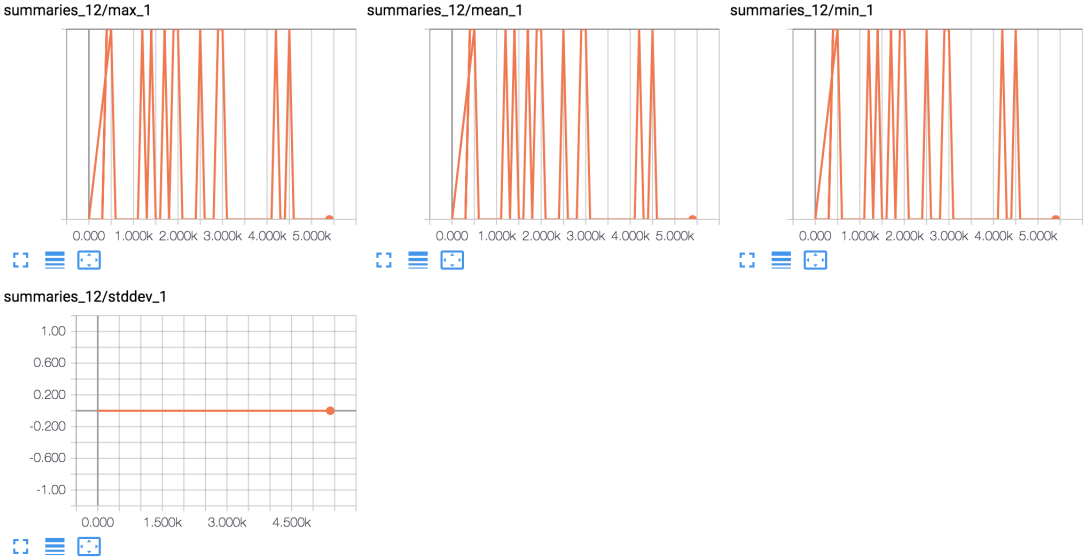

Figure 0.41: keep_prob

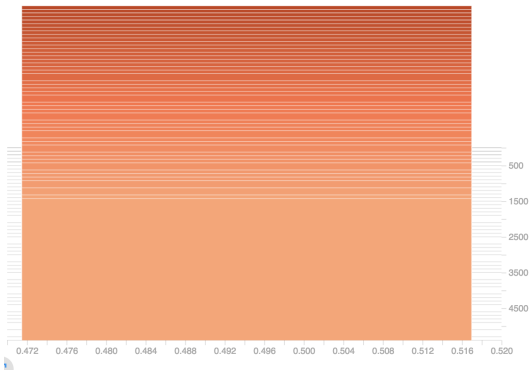

Figure 0.42: keep_prob_histogram

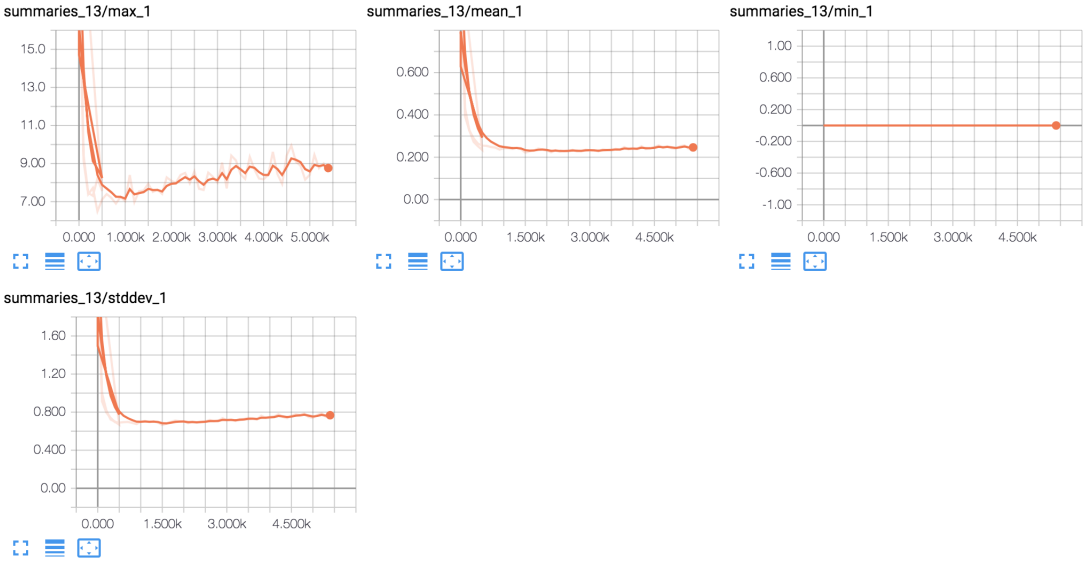

Figure 0.43: h_fc1_drop

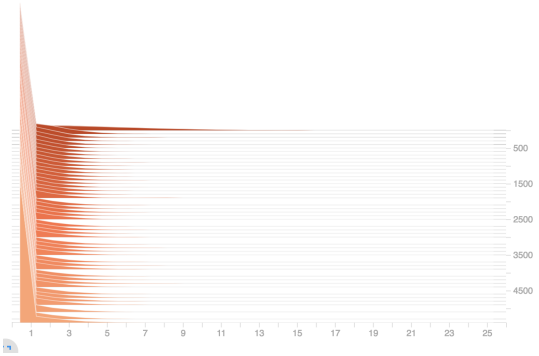

Figure 0.44: h_fc1_drop_histogram

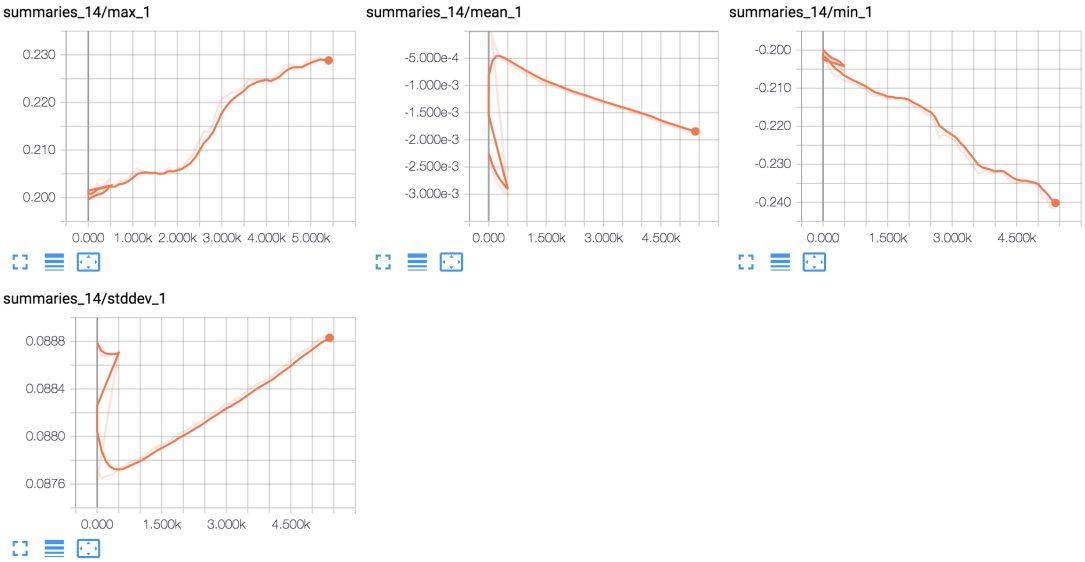

Figure 0.45: W_fc2

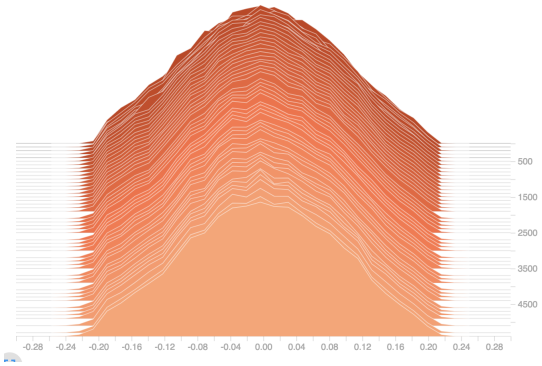

Figure 0.46: W_fc2_histogram

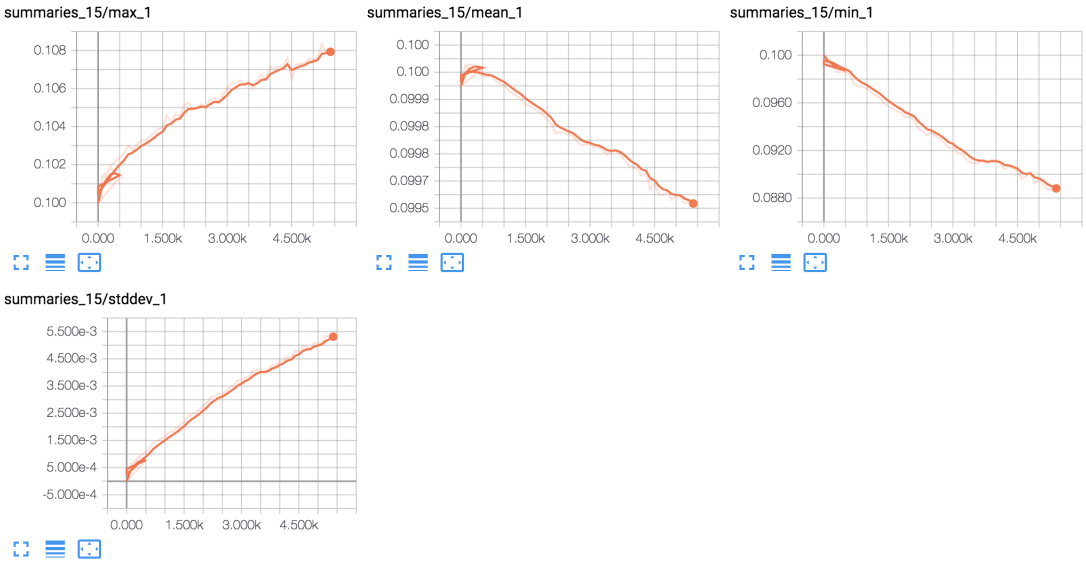

Figure 0.47: b_fc2

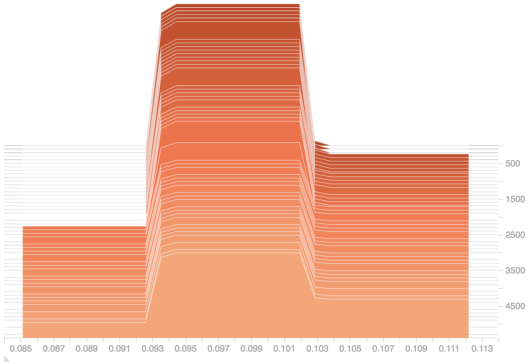

Figure 0.48: b_fc2_histogram

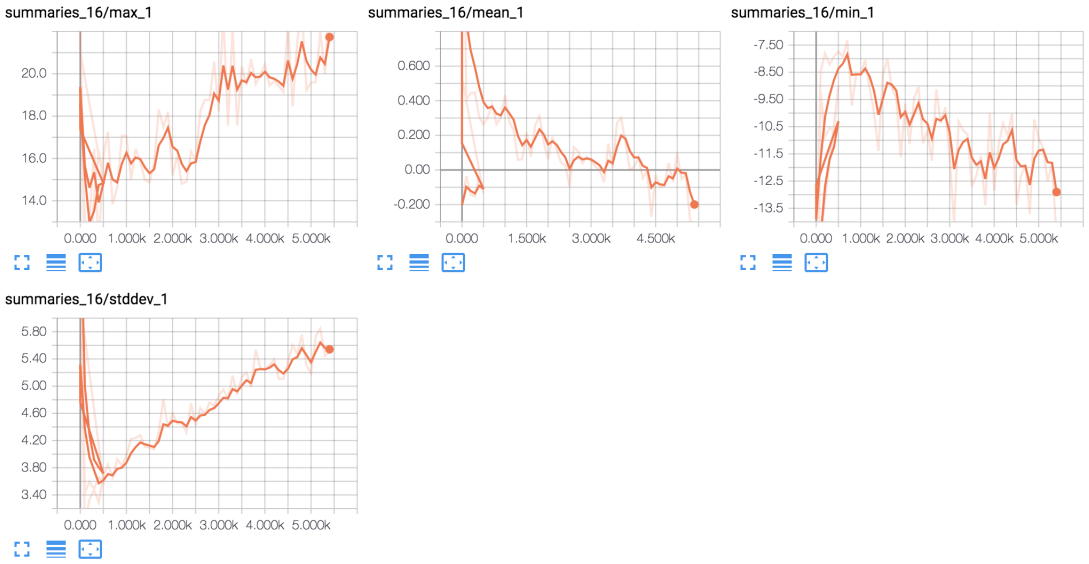

Figure 0.49: y_conv

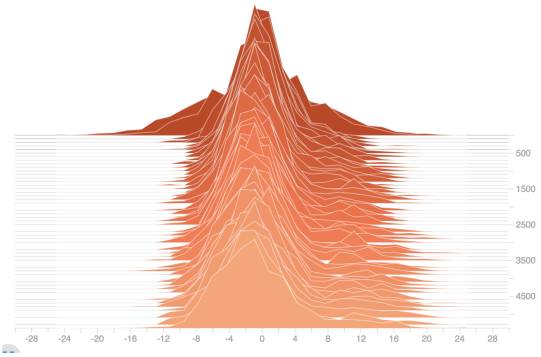

Figure 0.50: y_conv_histogram

2.c

I change the relu to tanh and do the following modifications:

```
# h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_conv1 = tf.nn.tanh(conv2d(x_image, W_conv1) + b_conv1)

# h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_conv2 = tf.nn.tanh(conv2d(h_pool1, W_conv2) + b_conv2)

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

Here is the final output results:

```
step 0, training accuracy 0.12
step 100, training accuracy 0.84
step 200, training accuracy 0.88
step 300, training accuracy 0.96
step 400, training accuracy 0.9
step 500, training accuracy 0.9
step 600, training accuracy 0.9
step 700, training accuracy 0.94
step 800, training accuracy 0.96
step 900, training accuracy 0.86
step 1000, training accuracy 0.98
step 1100, training accuracy 0.94
step 1200, training accuracy 0.94
step 1300, training accuracy 0.98
step 1400, training accuracy 0.96
step 1500, training accuracy 1
step 1600, training accuracy 0.96
step 1700, training accuracy 0.94
step 1800, training accuracy 0.96
step 1900, training accuracy 0.98
step 2000, training accuracy 0.98
step 2100, training accuracy 1
step 2200, training accuracy 1
step 2300, training accuracy 1
step 2400, training accuracy 0.98
step 2500, training accuracy 1
step 2600, training accuracy 0.98
step 2700, training accuracy 0.96
step 2800, training accuracy 0.94
step 2900, training accuracy 0.96
step 3000, training accuracy 0.98
step 3100, training accuracy 0.94
step 3200, training accuracy 0.98
step 3300, training accuracy 1
step 3400, training accuracy 1
step 3500, training accuracy 1
step 3600, training accuracy 1
step 3700, training accuracy 1
step 3800, training accuracy 1
step 3900, training accuracy 0.96
step 4000, training accuracy 0.98
step 4100, training accuracy 0.98
```

```
step 4200, training accuracy 0.96
step 4300, training accuracy 0.94
step 4400, training accuracy 0.94
step 4500, training accuracy 1
step 4600, training accuracy 0.98
step 4700, training accuracy 1
step 4800, training accuracy 1
step 4900, training accuracy 1
step 5000, training accuracy 1
step 5100, training accuracy 1
step 5200, training accuracy 1
step 5300, training accuracy 1
step 5400, training accuracy 1
test accuracy 0.983
The training takes 604.964856 second to finish
```

The figure of scalars:

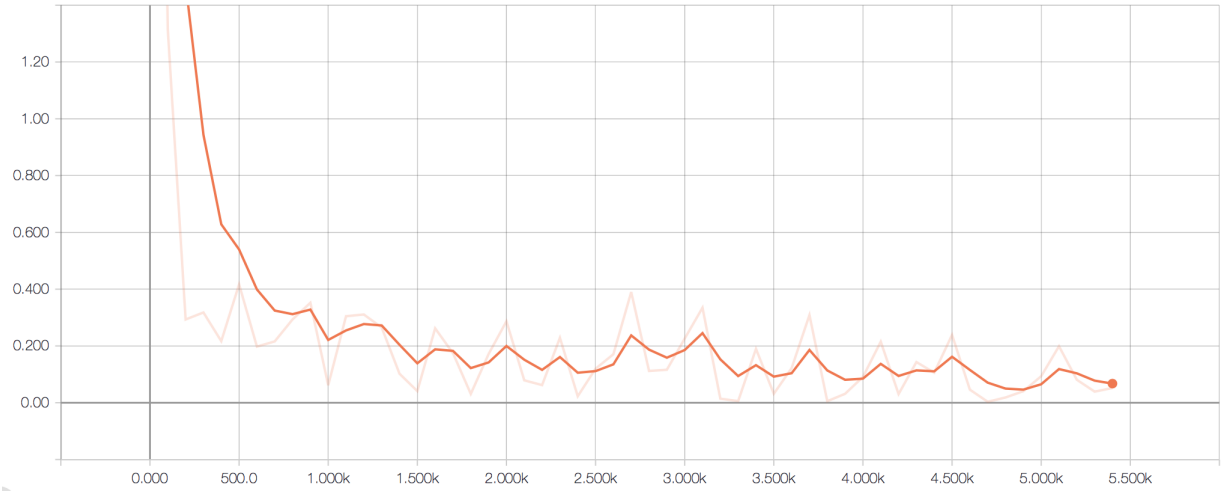

Figure 0.51: scalars

The figure of graphs:

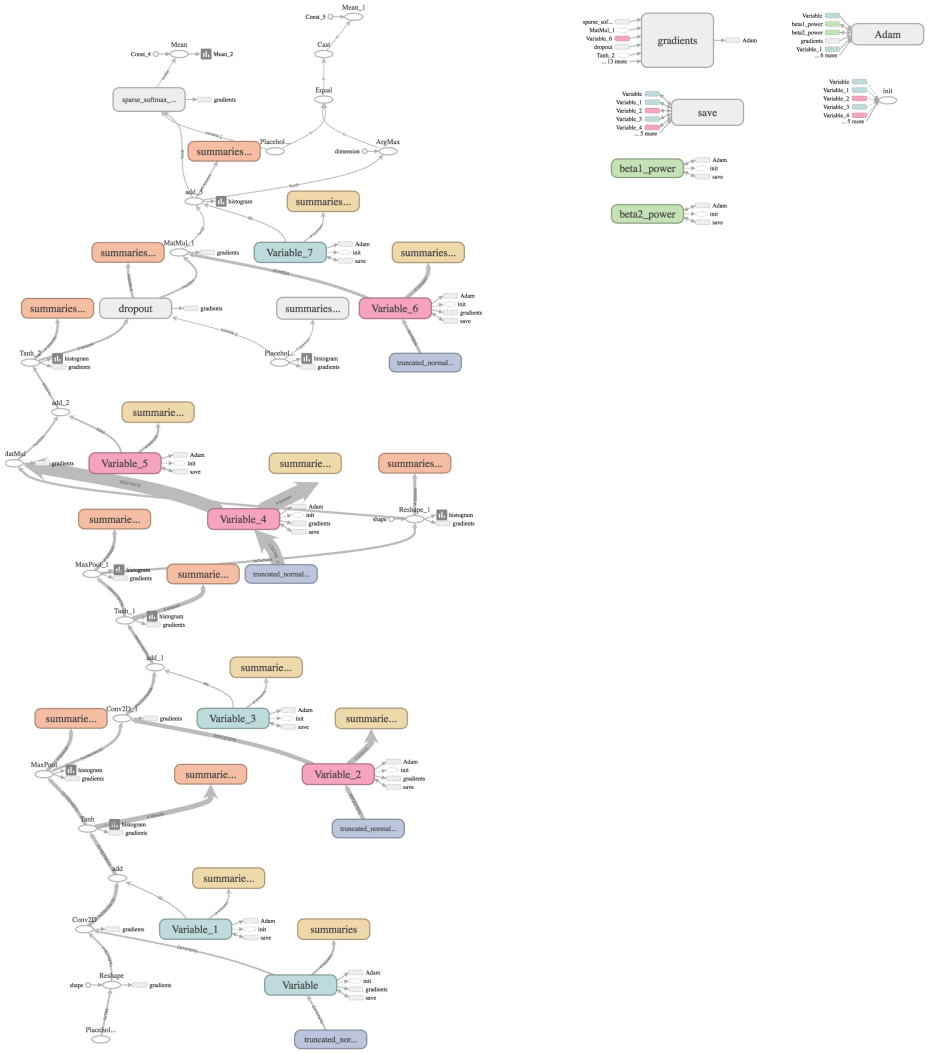

Figure 0.52: graphs

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:

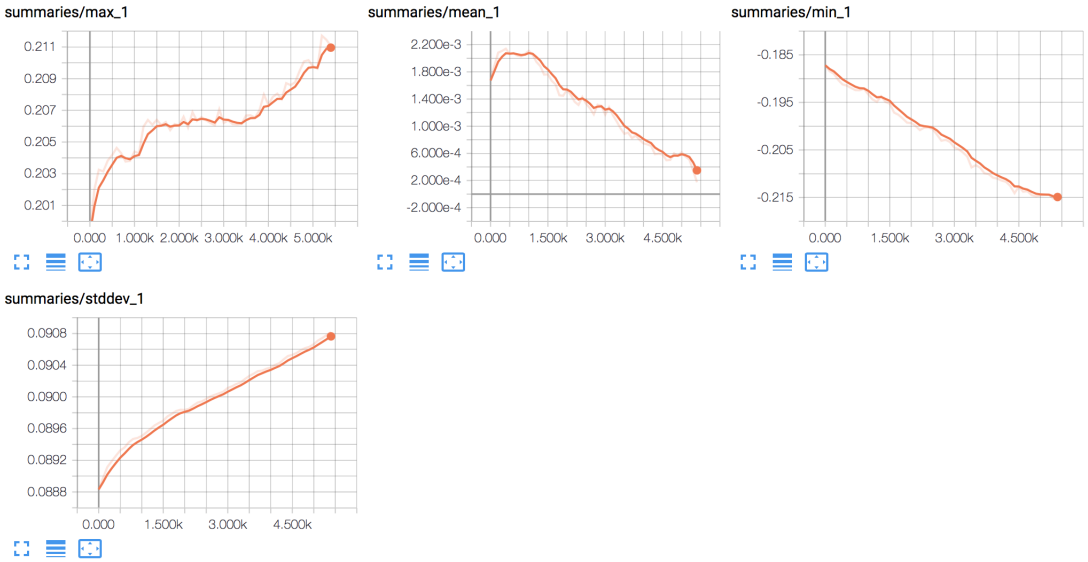

Figure 0.53: W_conv1

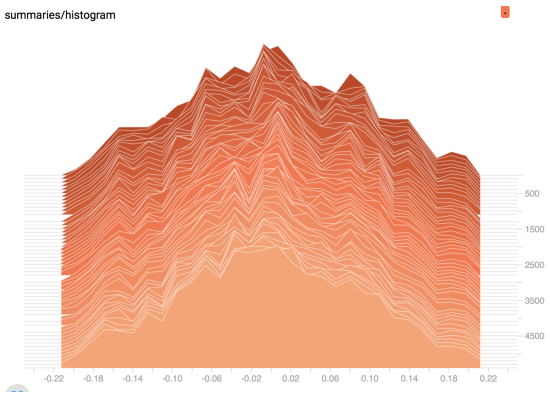

Figure 0.54: W_conv1_histogram

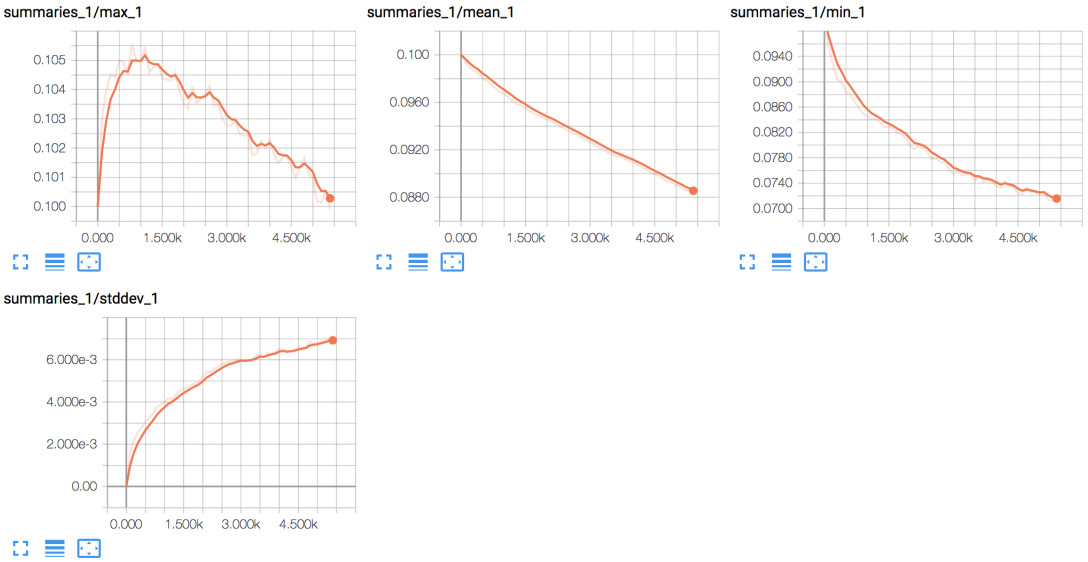

Figure 0.55: b_conv1

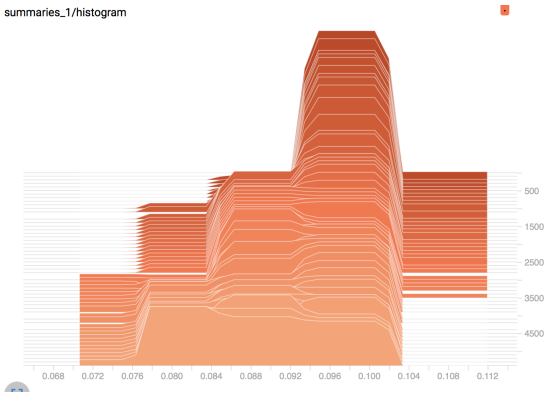

Figure 0.56: b_conv1_histogram

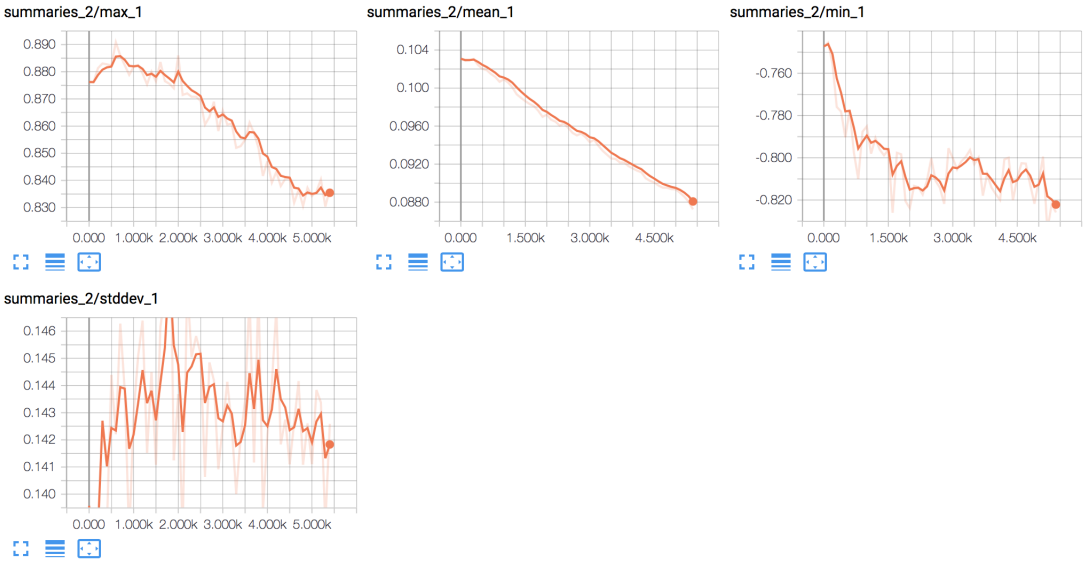

Figure 0.57: h_conv1

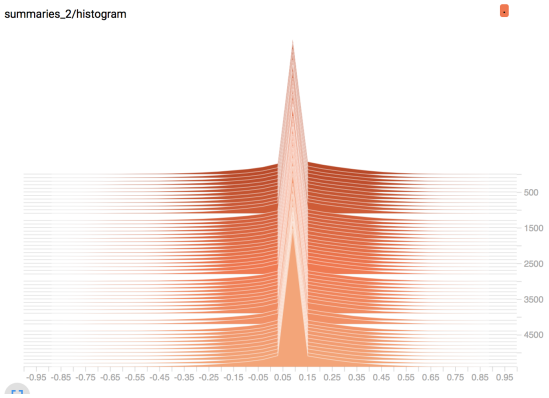

Figure 0.58: h_conv1_histogram

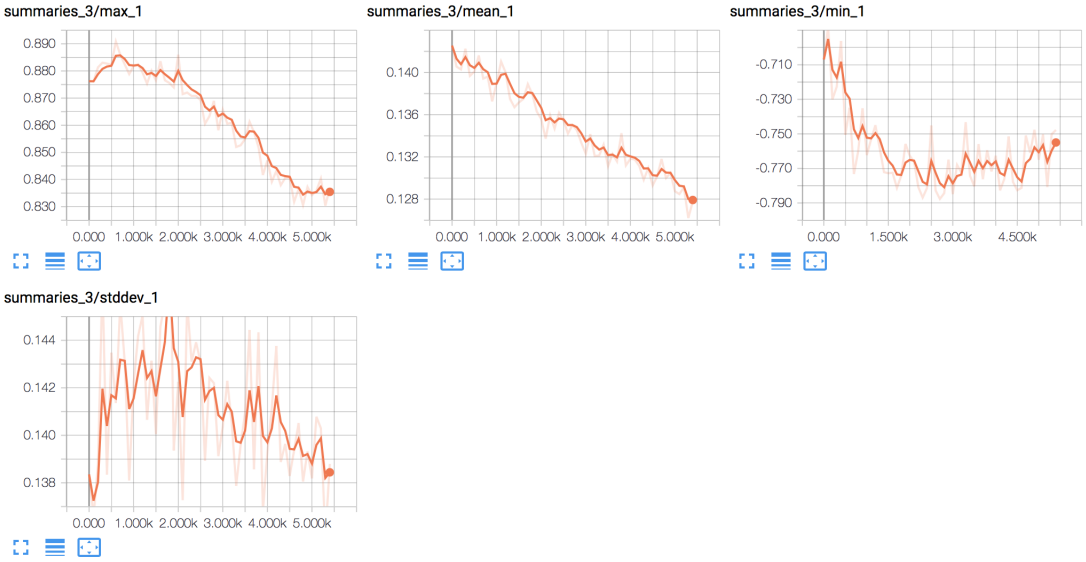

Figure 0.59: h_pool1

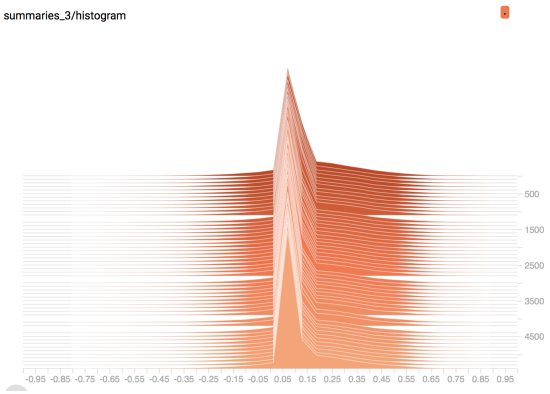

Figure 0.60: h_pool1_histogram

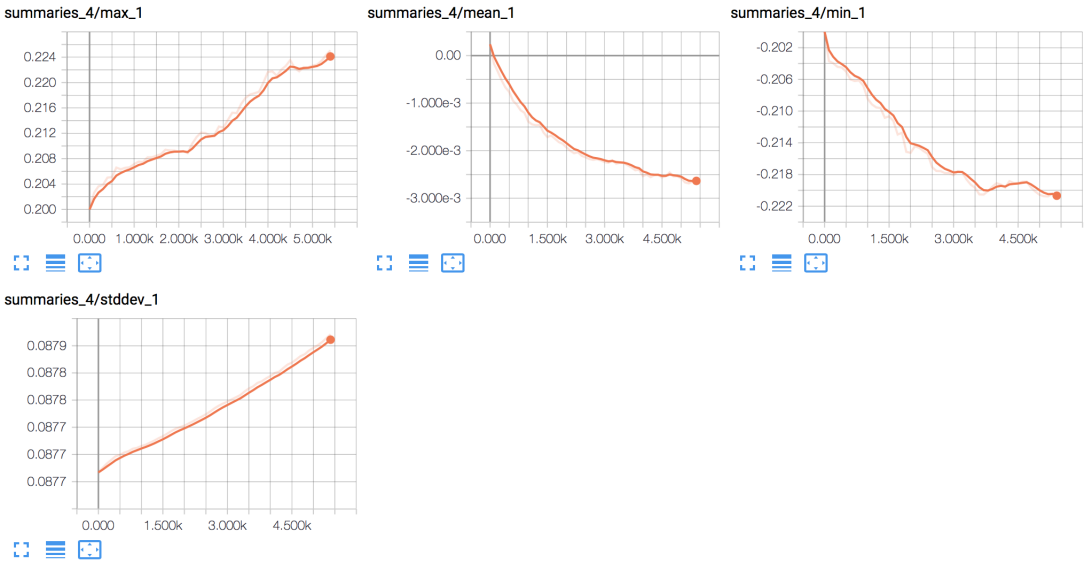

Figure 0.61: W_conv2

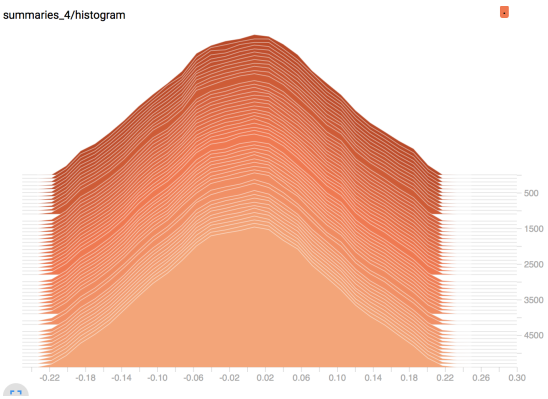

Figure 0.62: W_conv2_histogram

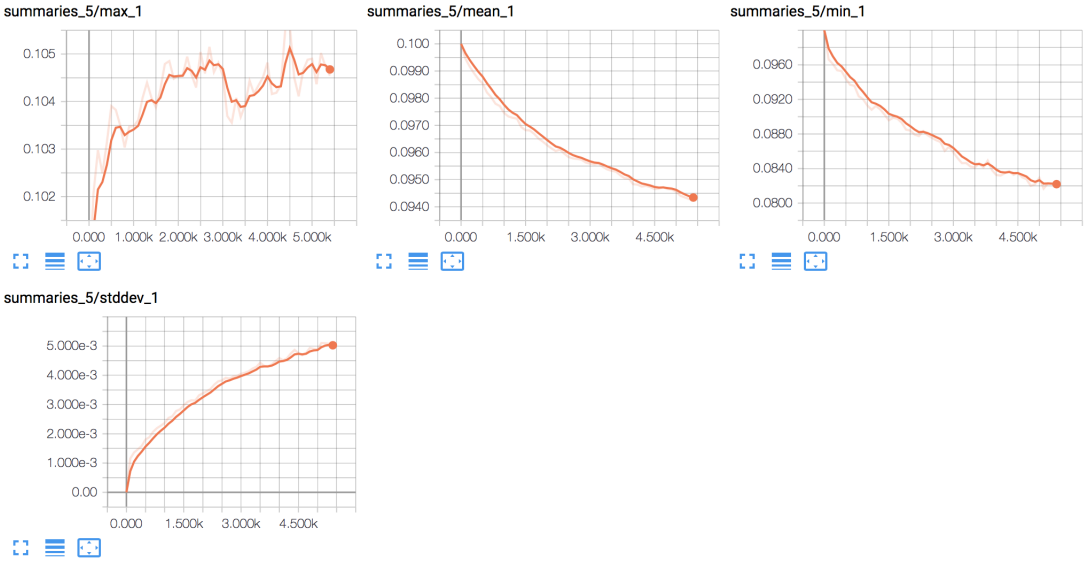

Figure 0.63: b_conv2

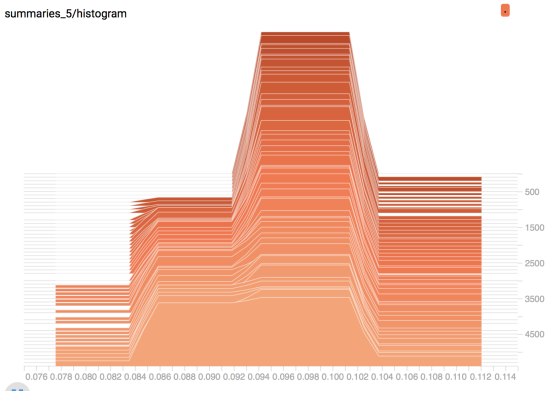

Figure 0.64: b_conv2_histogram

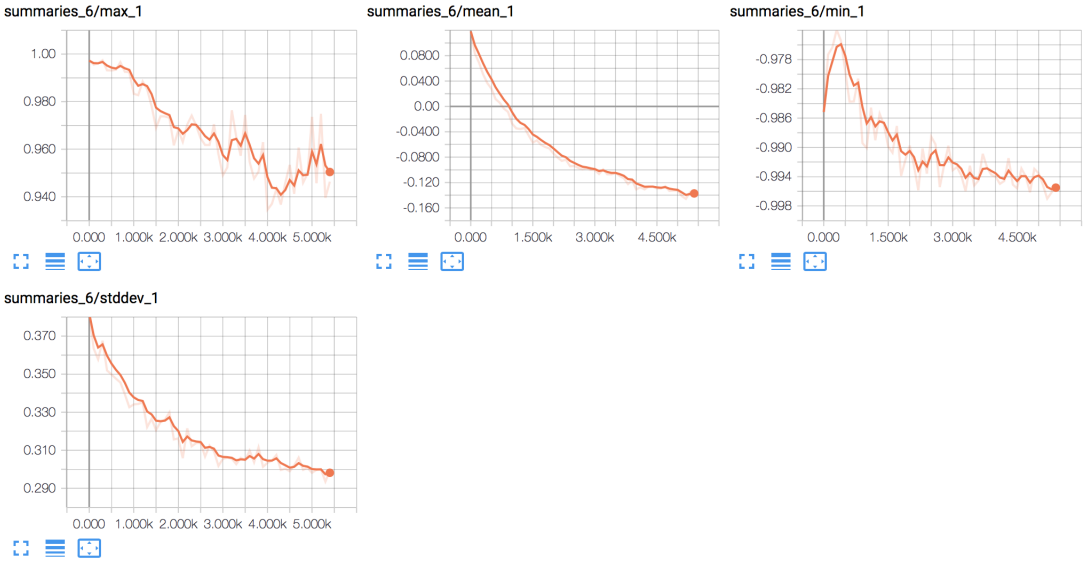

Figure 0.65: h_conv2

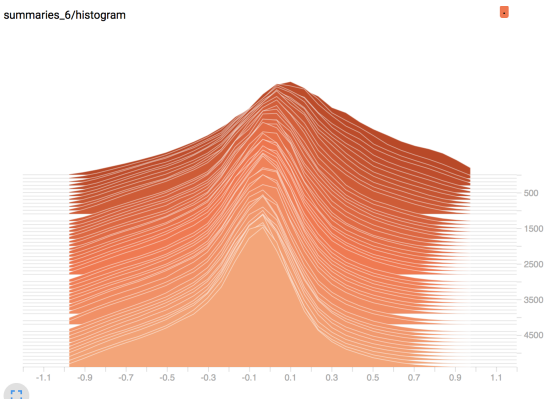

Figure 0.66: h_conv2_histogram

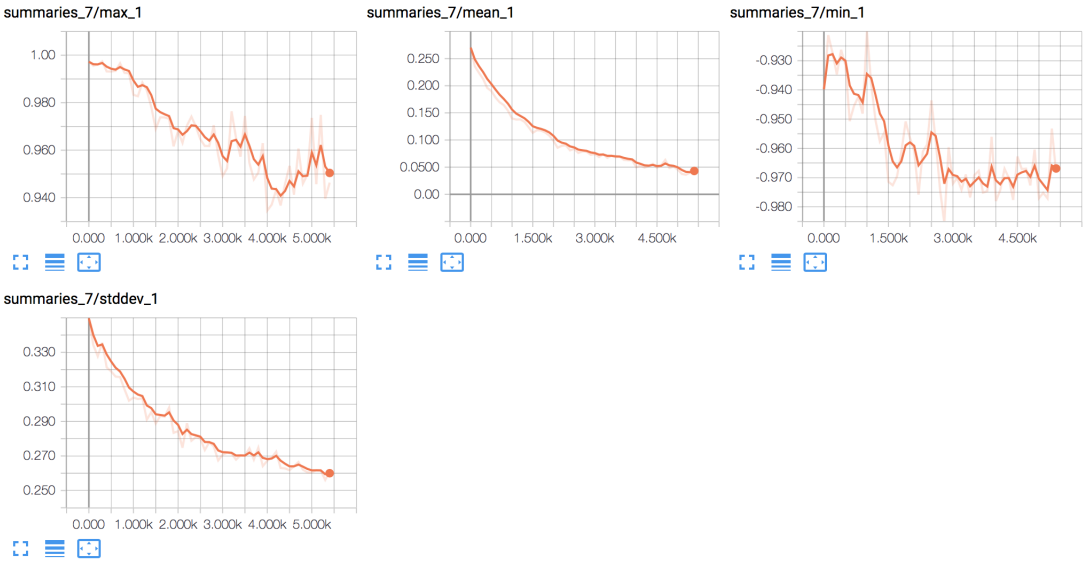

Figure 0.67: h_pool2

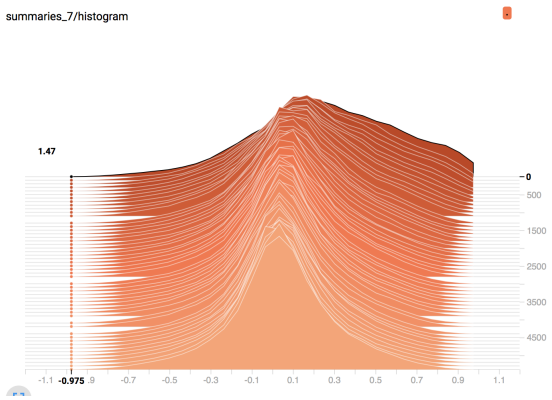

Figure 0.68: h_pool2_histogram

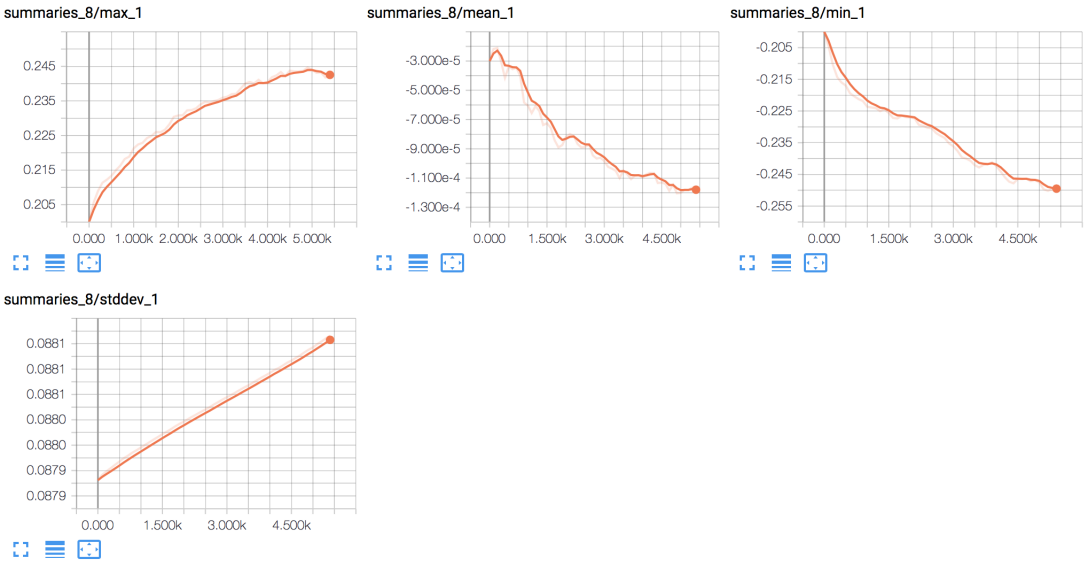

Figure 0.69: W_fc1

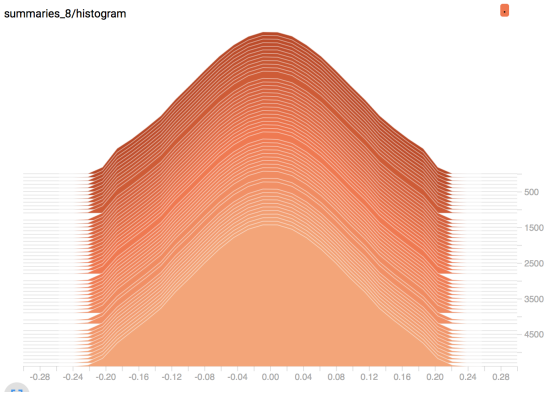

Figure 0.70: W_fc1_histogram

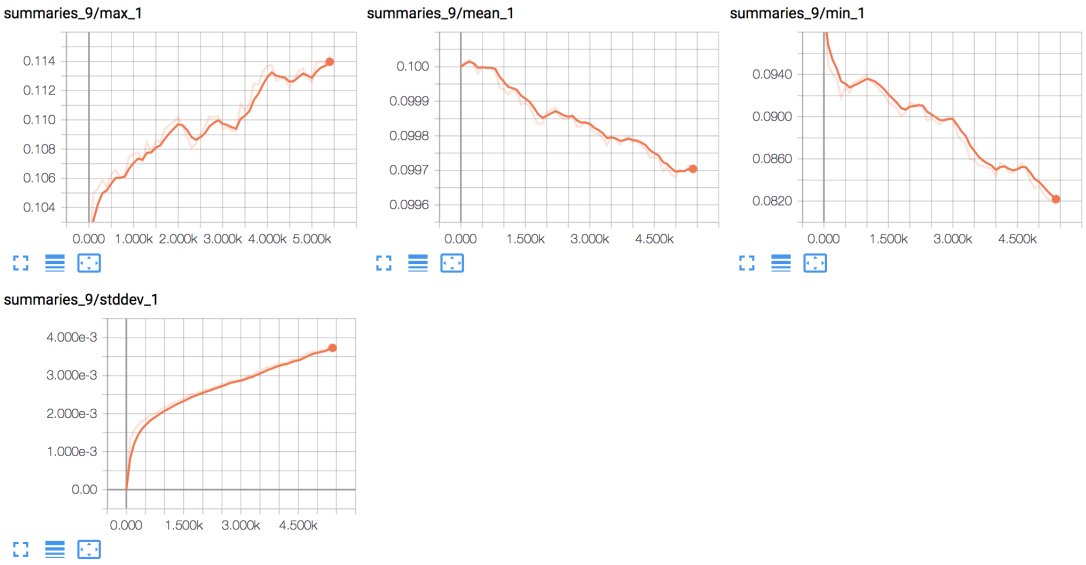

Figure 0.71: b_fc1

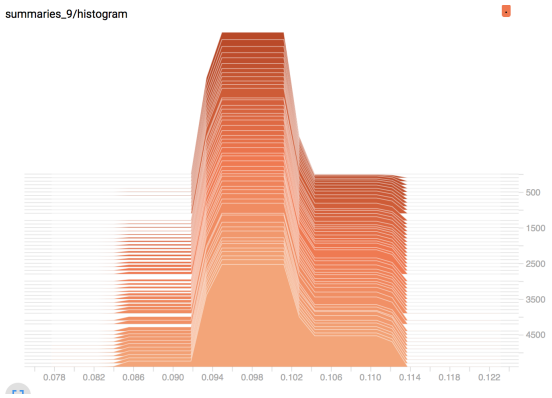

Figure 0.72: b_fc1_histogram

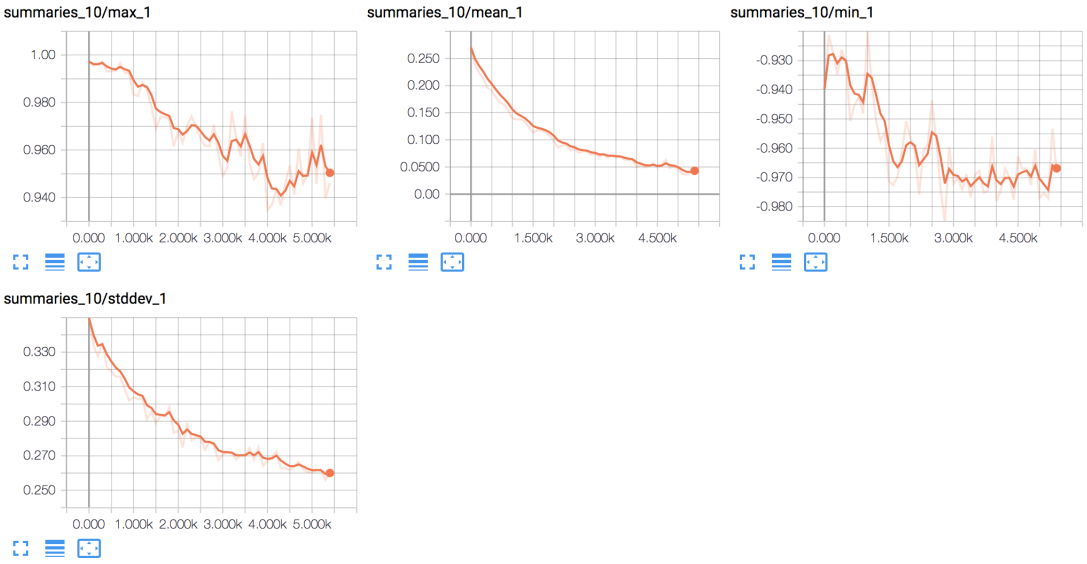

Figure 0.73: h_pool2_flat

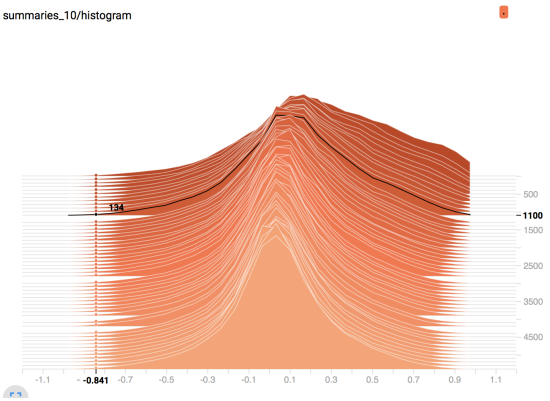

Figure 0.74: h_pool2_flat_histogram

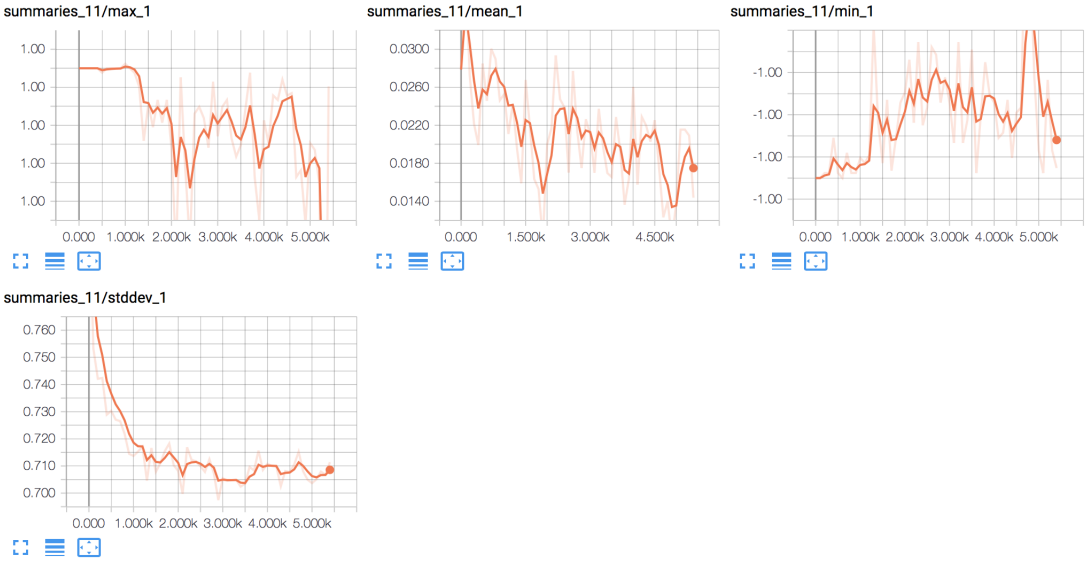

Figure 0.75: h_fc1

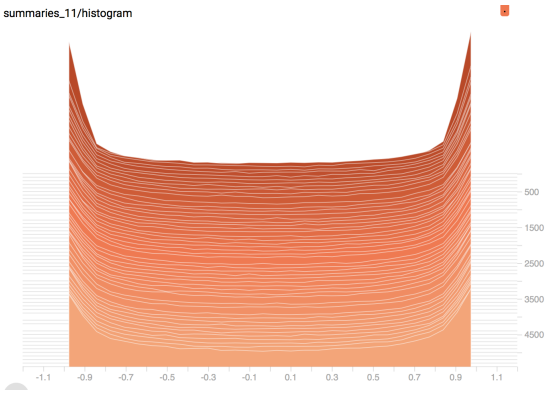

Figure 0.76: h_fc1_histogram

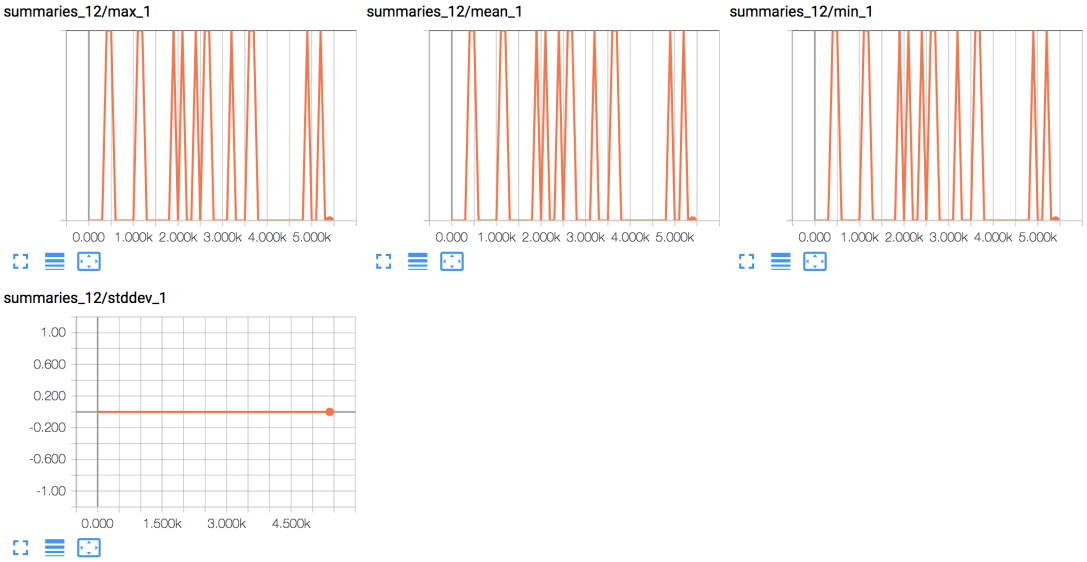

Figure 0.77: keep_prob

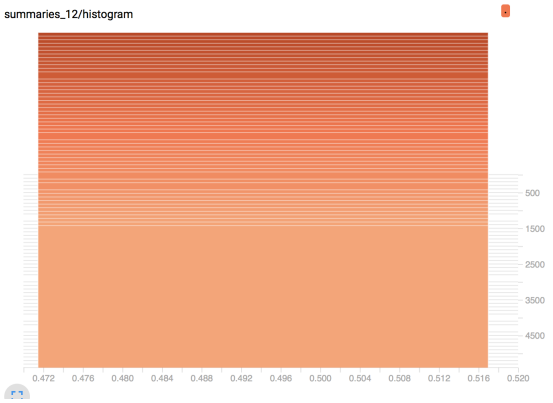

Figure 0.78: keep_prob_histogram

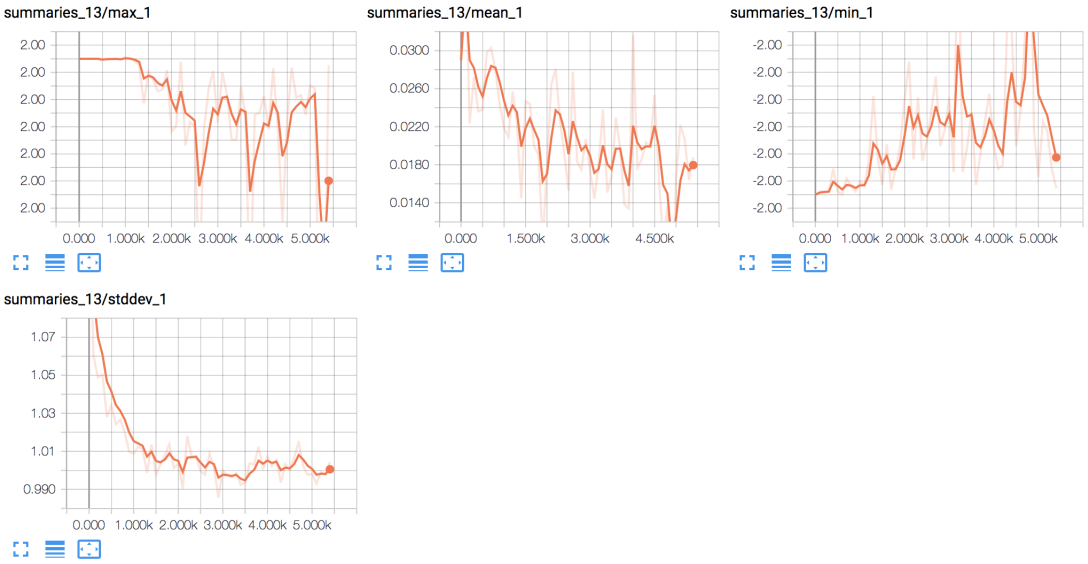

Figure 0.79: h_fc1_drop

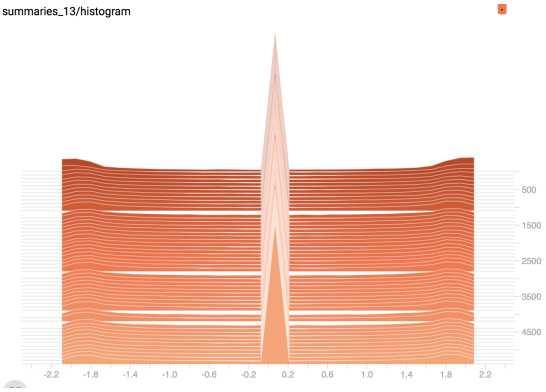

Figure 0.80: h_fc1_drop_histogram

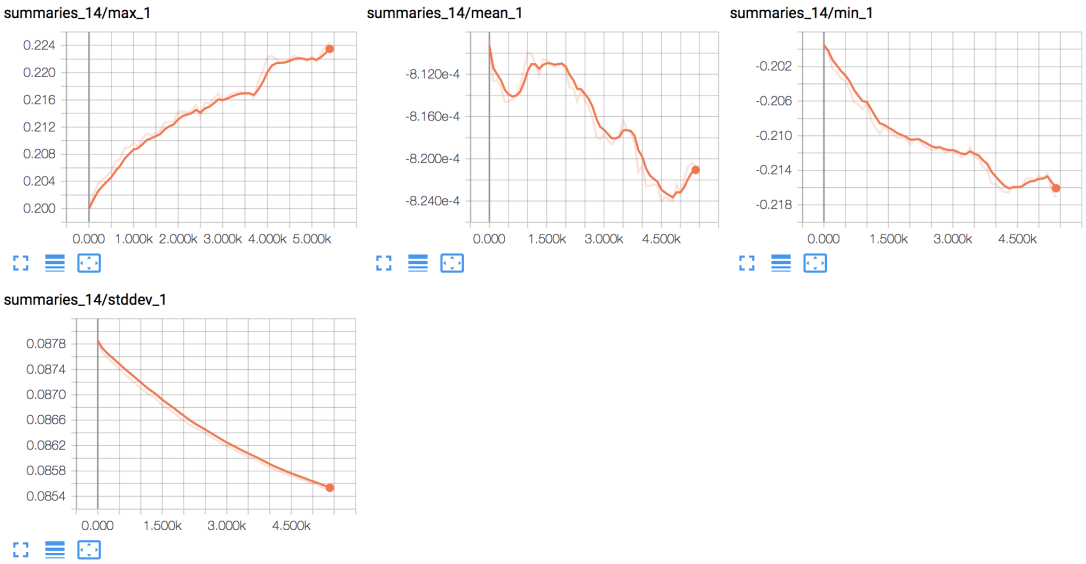

Figure 0.81: W_fc2

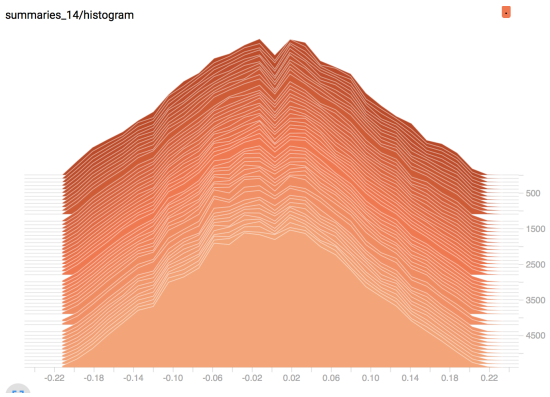

Figure 0.82: W_fc2_histogram

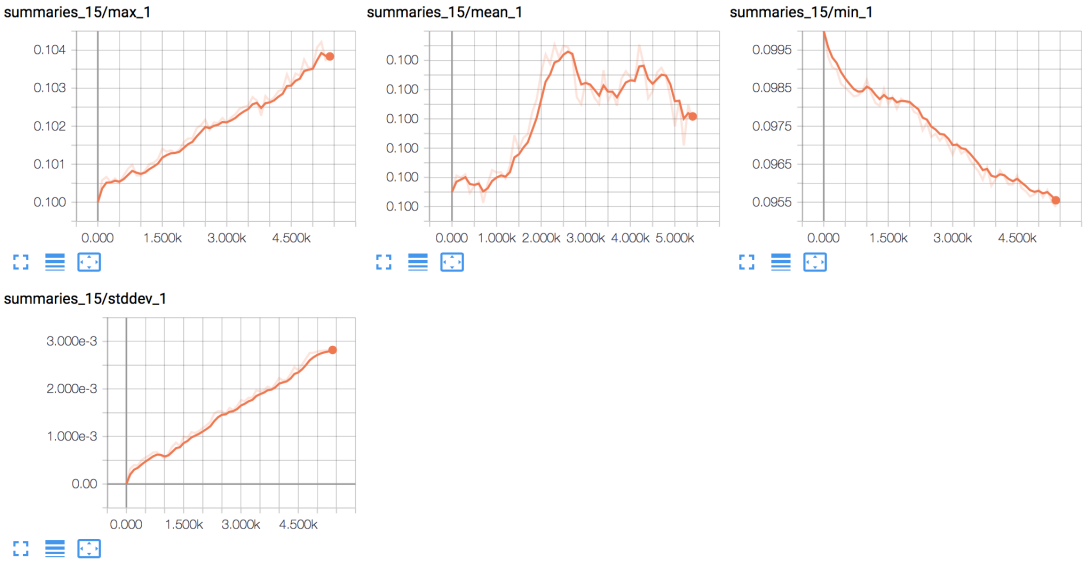

Figure 0.83: b_fc2

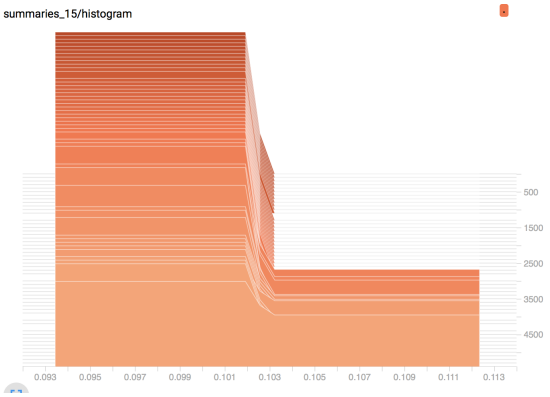

Figure 0.84: b_fc2_histogram

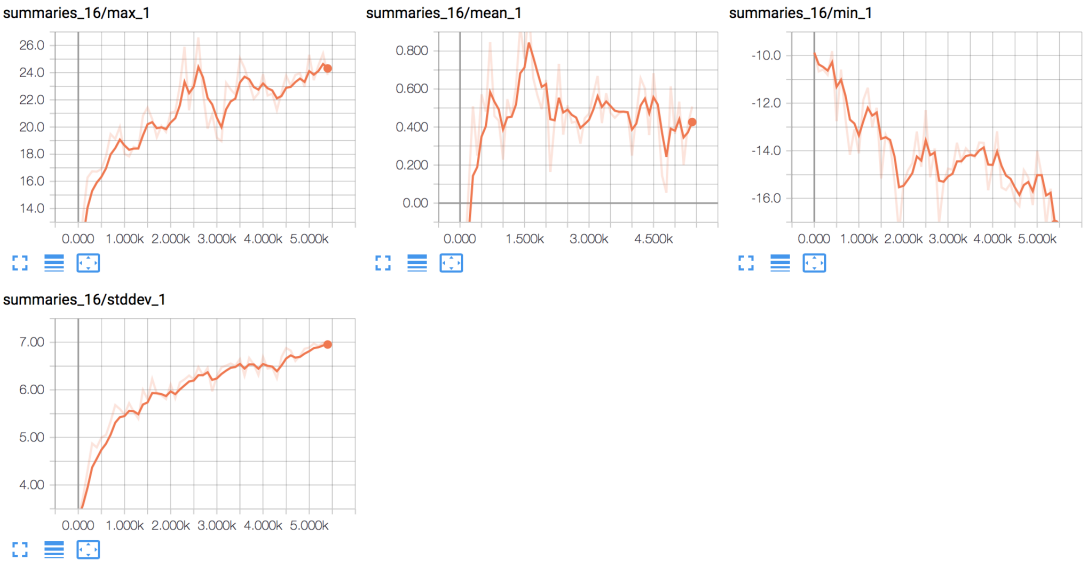

Figure 0.85: y_conv

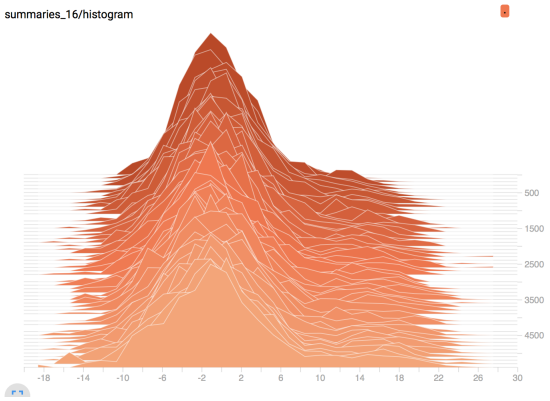

Figure 0.86: y_conv_histogram