ELEC 576: Assignment 1

Xiaoqian Chen

October 8, 2019

Contents

1 Artificial Neural Network Classifier

1.1 Architecture

Suppose we have a fixed training set (x(1),y(1)),...,(x(N),y(N)) of N training examples. We have a set $\{X_i\}_{i=1}^N \subset \mathbb{R}^{n_1}$ of $N \in \mathbb{N}$ *data points*, each with $n_1 \in \mathbb{N}$ *dimensions*, and that we have *classified* each such point as belonging to one of $n_C \in \mathbb{N}$ *classes*. Let $\{y_i\}_{i=1}^N \subset \mathscr{C}^{n_L}$ and note that the information that x_i belongs to a particular class is captured by the binary vector$(0,1)$ or $(1,0)$. In the Assignment1, $n_1 = 2$ and $n_C = 2$.

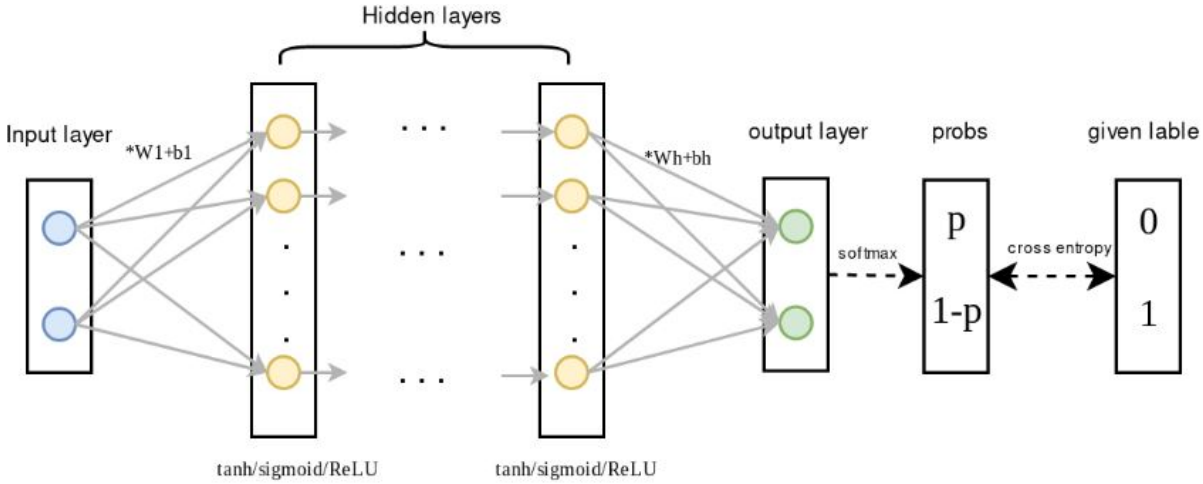


Figure 1: Architecture for Artificial Neural Network Classifier

Multilayer Perception Architecture for an Classifier is a n layers fully-connected network, having an input layer, an output layer and one hidden layers connecting them. Neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. The dimension hidden layer $N_h \geq 1$, the three layers neural network($N_h = 1$) has the simplest architecture. Here, in our assignment1, the dimension of the output layer is 2.

Every "neuron" is a computational unit that takes as input $x = x_1, \ldots, x_k$ (and k +1 intercept term), and outputs $h_W, b(x) = \varphi(\sum Wx + b)$, where $\varphi : R \to R$ is called the activation function. The activation function is applied for all neurons in the hidden layer, such as tanh, sigmoid and ReLU.

$$\varphi(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0, \end{cases} \qquad \text{(ReLU)}$$

$$\varphi(x) = \frac{1}{1+e^{-x}}, \qquad \text{(sigmoid)}$$

$$\varphi(x) = \tanh(x). \qquad \text{(hyperbolic tangent)}$$

We call this step forward propagation. More generally, recalling that we also use a(1)=x to also denote the values from the input layer, then given layer (l)'s activations $a^{(l)}$, we can compute layer (l+1)'s activations $a^{(l+1)}$ as:

$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = \varphi(z^{(l+1)})$$

The softmax function applied to the output from the output layer, which lead to out final output. Assume we have $Out_k = (y_1, y_2)$ from the output layer.There is a probability vector as output of softmax function, such that $\hat{y}_k = (p_1, p_2)$, and $p_1 + p_2 = 1$.

$$\hat{y}_k = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} \frac{e^{y_1}}{e^{y_1}+e^{y_2}} \\ \frac{e^{y_2}}{e^{y_1}+e^{y_2}} \end{bmatrix}$$

The final classification of X_k is computed from $\hat{y}_k$ by identifying the class have the biggest probabilities.

$$\text{Prediction of } X_k = argmax(\hat{y}_k) \tag{1.1}$$

For example, assuming the final output is $\hat{y}_k = (0.2, 0.8)$, the classification label will be (0,1), since the second probability $0.8 > 0.2$.

The goal of the MLP classifier is to obtain output probability vector $\hat{y}_k$ very close to the binary vector encoding the true class of the input vector, which is one of $\{(1,0),(0,1)\}$. The cross entropy is used to measure the performance quality. Given a training set of N examples, we then define the overall LOSS function to be:

$$\mathbb{LOSS}(W,b) = \frac{1}{N}\sum_{i=1}^{N} -y_i log(\hat{y}_i) + \frac{\lambda}{2}||W||^2$$

The first term in the definition of LOSS function is an average cross entropy error term and the second term is a regularization term. Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent. The regularization coefficient λ controls the relative importance of the two terms.

1.2 Back-Propagation

The learning algorithm is use gradient descent to minimize LOSS function $\mathbb{LOSS}(W,b)$ to train weights W, b. We initialize W, b with small random numbers. One iteration of gradient descent updates the parameters W, b:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial \mathbb{LOSS}(W,b)}{\partial W_{ij}^{(l)}}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial \mathbb{LOSS}(W,b)}{\partial b_i^{(l)}}$$

where α is the learning rate, is a time decay parameter. The key step is computing the partial derivatives above by the back-propagation algorithm. The algorithm can then be written:

1. Perform a feed-forward pass, computing the activations for hidden layers $L_2, \ldots, L_{n_L-1}$, up to the output layer L_{n_L}, using the equations defining the forward propagation steps.
2. For the output layerL_{n_L} , set
$$\delta^{(n_L)} = (a^{(n_L)} - y) \cdot \varphi'(z^{(n_L)})$$
(Remark: $a^{(n_L)} - y$ is the derivative of loss function back to the output layer$z^{(n_L)}$ with chain rule .)
3. For layers$l = L_{n_L-1}, \ldots, L_2$, set
$$\delta^{(l)} = (W^{(l)}\delta^{(l+1)}) \cdot \varphi'(z^{(l)})$$

4. Compute the desired partial derivatives, which are given as:

$$\frac{\nabla \mathbb{LOSS}(W,b)}{\nabla W^{(l)}} = \delta^{(l+1)} a^{(l)T}$$

$$\frac{\nabla \mathbb{LOSS}(W,b)}{\nabla b^{(l)}} = \delta^{(l+1)}$$

To train our neural network, we can now repeatedly take steps of gradient descent to reduce our cost function.

The derivative of the activation function are given by:

$$\varphi'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0, \end{cases} \qquad \text{(ReLU)}$$

$$\varphi'(x) = \varphi(x)(1 - \varphi(x)), \qquad \text{(sigmoid)}$$

$$\varphi'(x) = 1 - \varphi(x)^2. \qquad \text{(hyperbolic tangent)}$$

1.3 Experiments of 3-layer NN with varied activation functions and number of hidden units

Each picture in the next few pages is a visual representation of 1000 distinct elements of some square $[a,b] \times [c,d] \subset \mathbb{R}^2$, each belonging to one of two classes (red or blue), where 0.000, 0.025, and 0.075 represent different levels of *noise* (0.000 represents no noise).

In our implement, the large regularization coefficient $\lambda = 0.1, 0.001$ fails , so that we set $\lambda = 0$.

We implement 3-layer neural network in Python as a class that inherits all of its fields and methods from the class that we implemented for n-layers neural networks: the following is the entire code for this class.

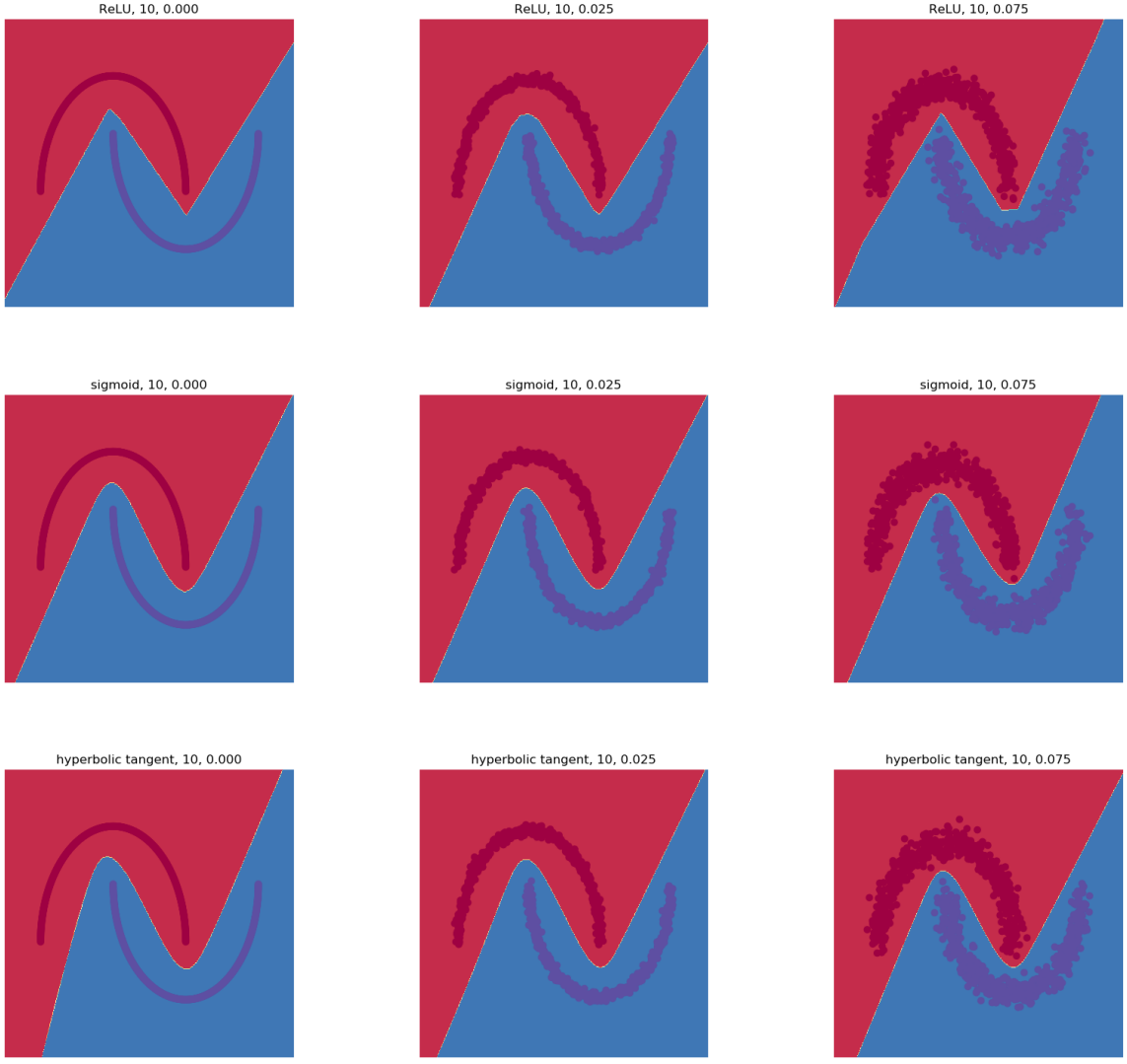
```
class ThreeLayerNeuralNetwork(DeepNeuralNetwork):
    def __init__(self,
                 input_dim = 2,
                 hidden_dim = 10,
                 output_dim = 2,
                 activation_type = 'tanh',
                 regularization = 0,
                 random_seed = None):
        super().__init__([input_dim,
                          hidden_dim,
                          output_dim],
                          activation_type,
                          regularization,
                          random_seed)
```

In the following subsection,we train the network using different activation functions (Tanh, Sigmoid and ReLU) and increase the number of hidden units, ranging from 10 to 30. Each picture in the next few pages is of the *decision boundary* of a trained neural network of 3 layers with $n_1 = 2$, $n_2 \in \{10, 20, 30\}$, $n_3 = 2$.

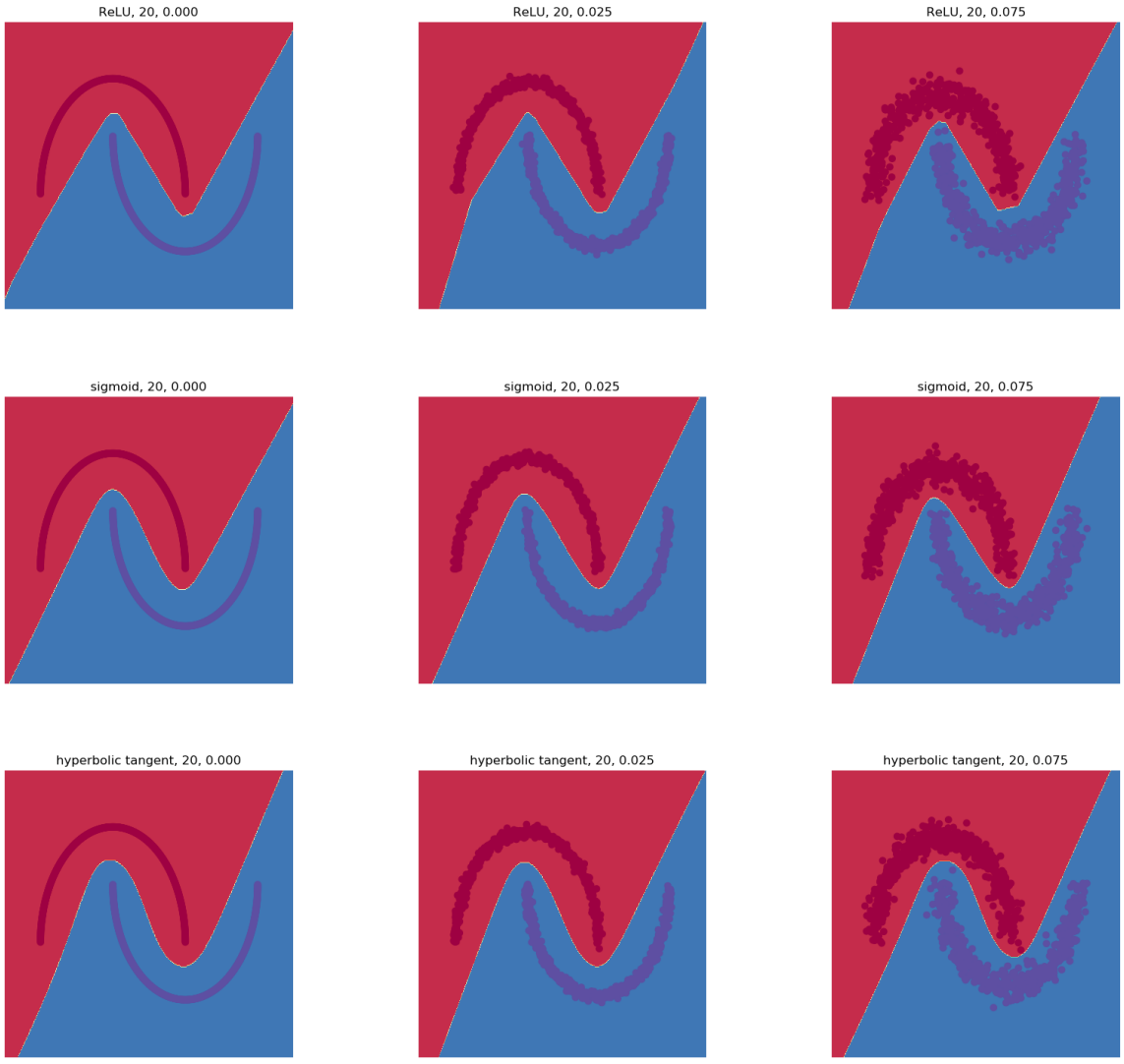
Expecting form the moons data set, 3- layers NN also performs well in distinguishing two Concentric circles.

The implement result indict that all activation functions works well. In details, we have to be careful in the relatively Small learning rate coopering with sigmoid function. In addition, the more units in the hidden layer means more flexibility in the decision boundary.

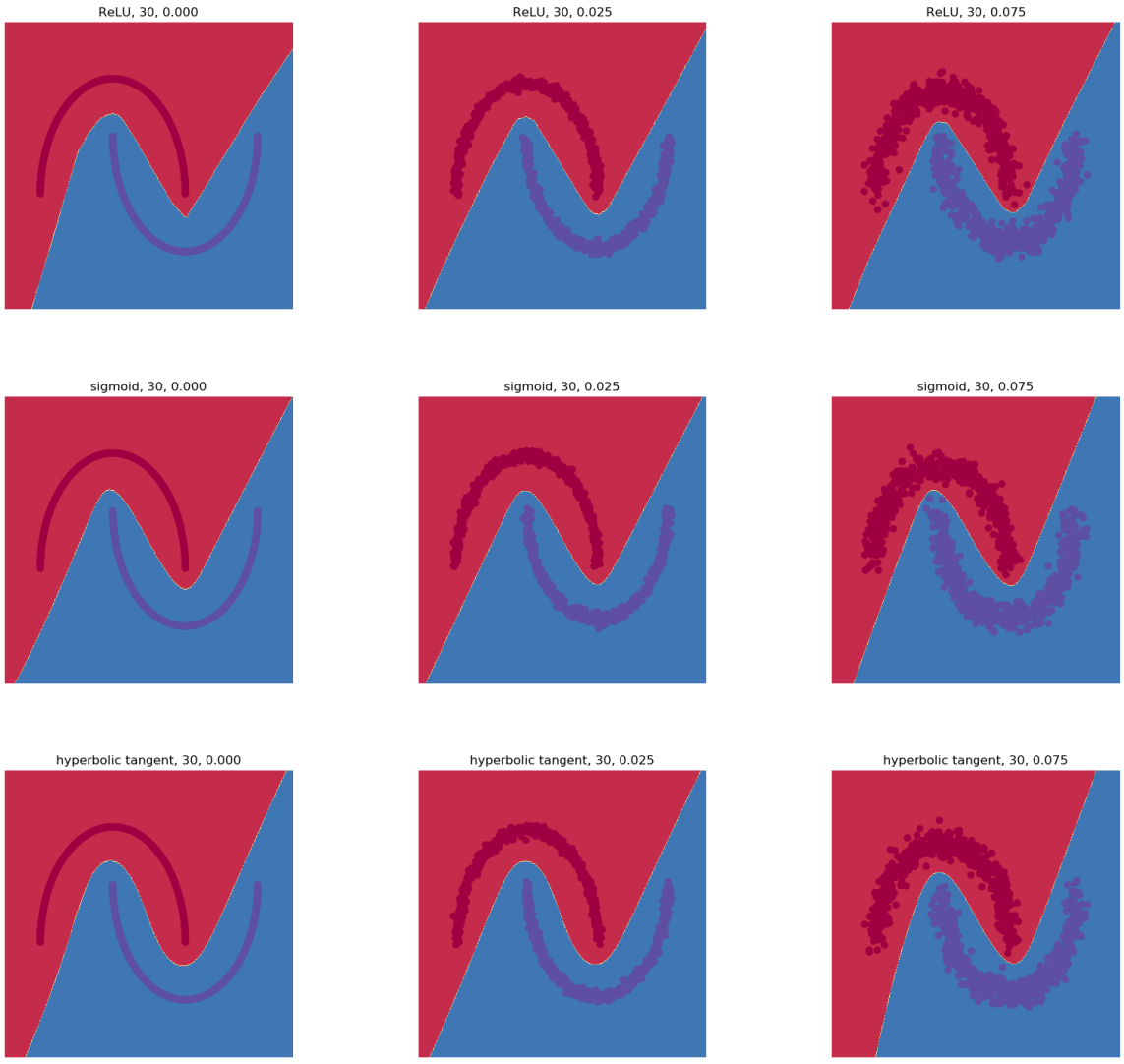
1.3.1 $n_2 = 10$, make_moons Data Set



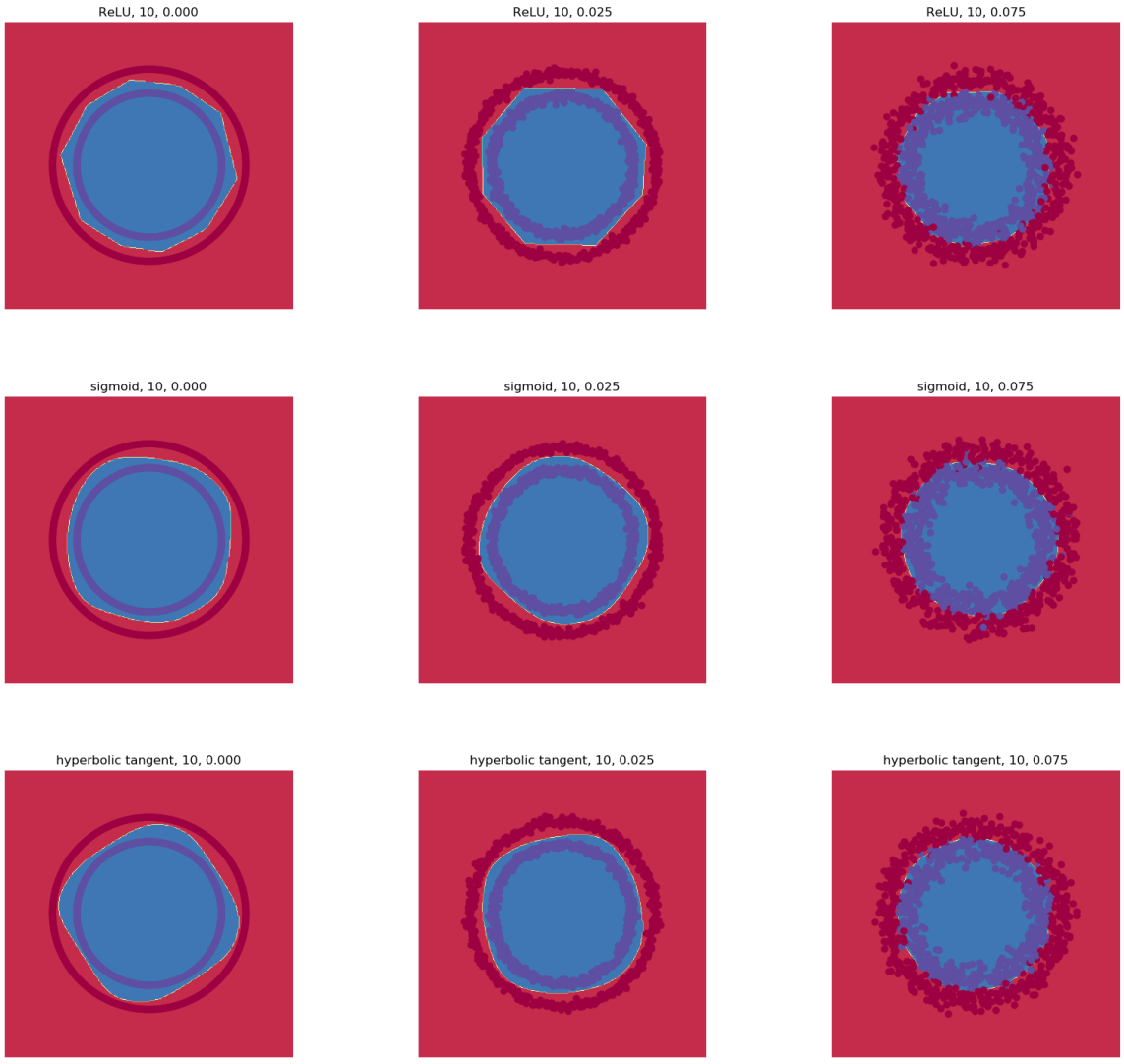
1.3.2 $n_2 = 20$, make_moons Data Set



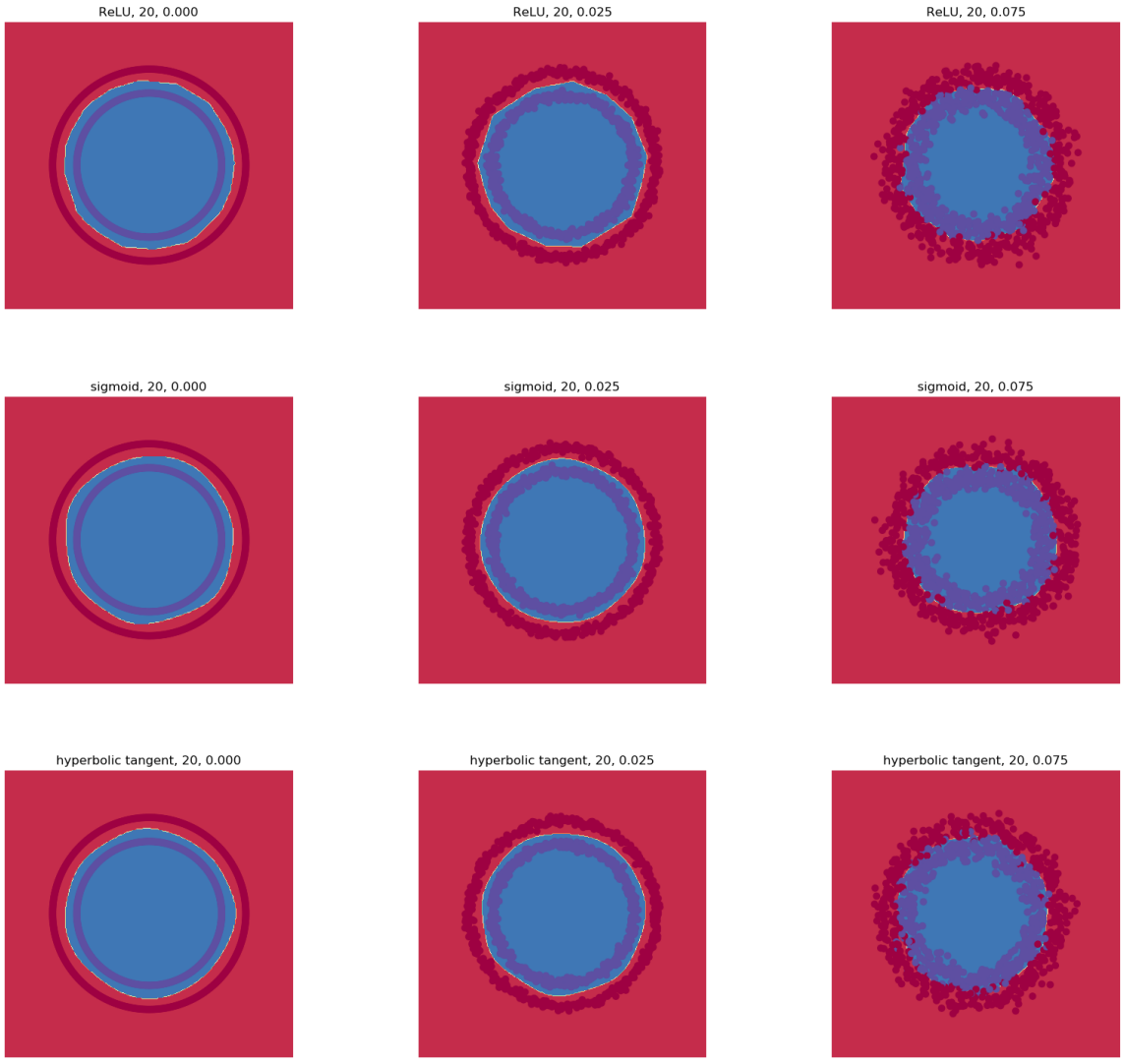
1.3.3 $n_2 = 30$, make_moons Data Set



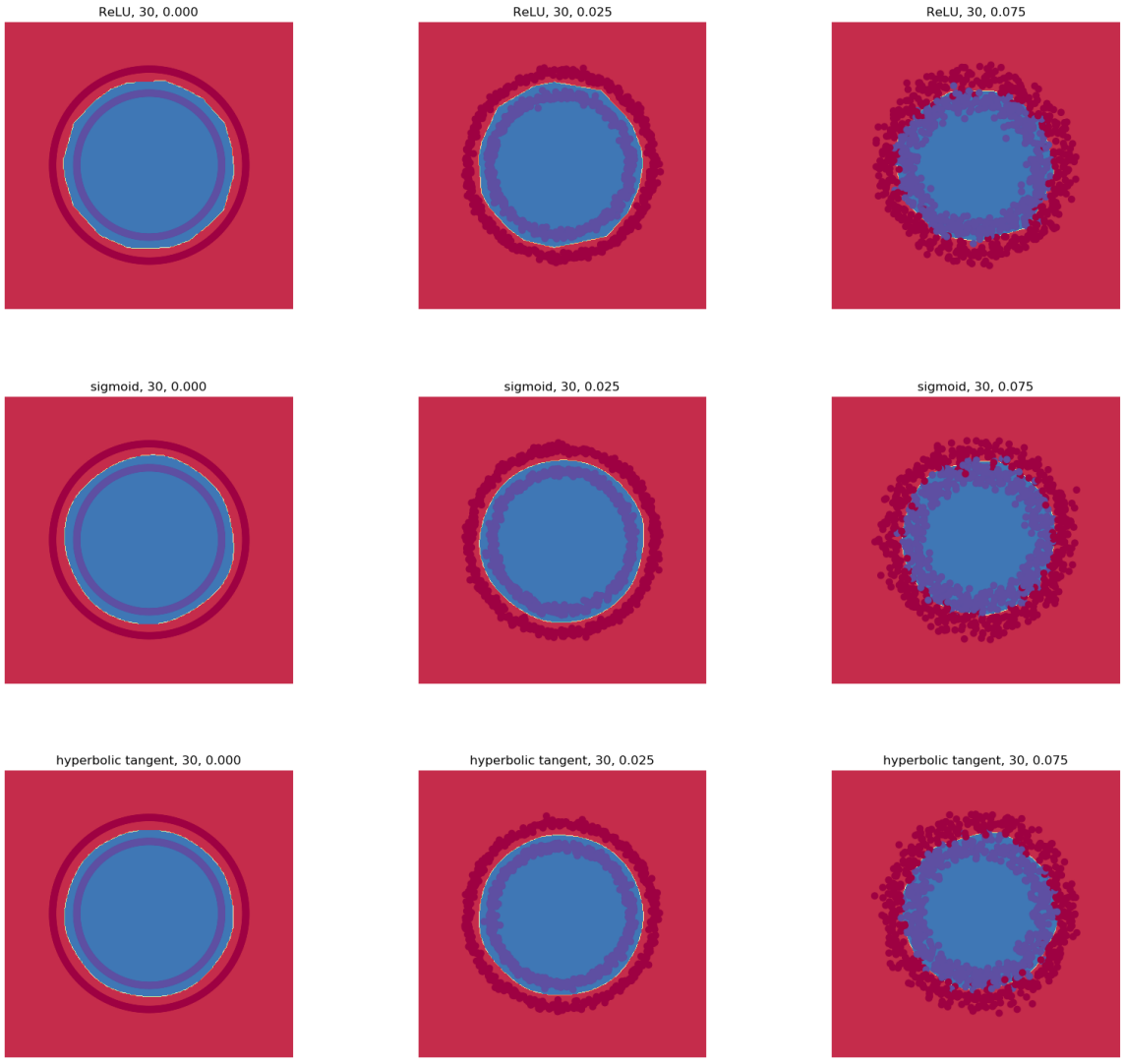
1.3.4 $n_2 = 10$, make_circles Data Set



1.3.5 $n_2 = 20$, make_circles Data Set



1.3.6 $n_2 = 30$, make_circles Data Set



2 Deeper Neural Network of n Layers

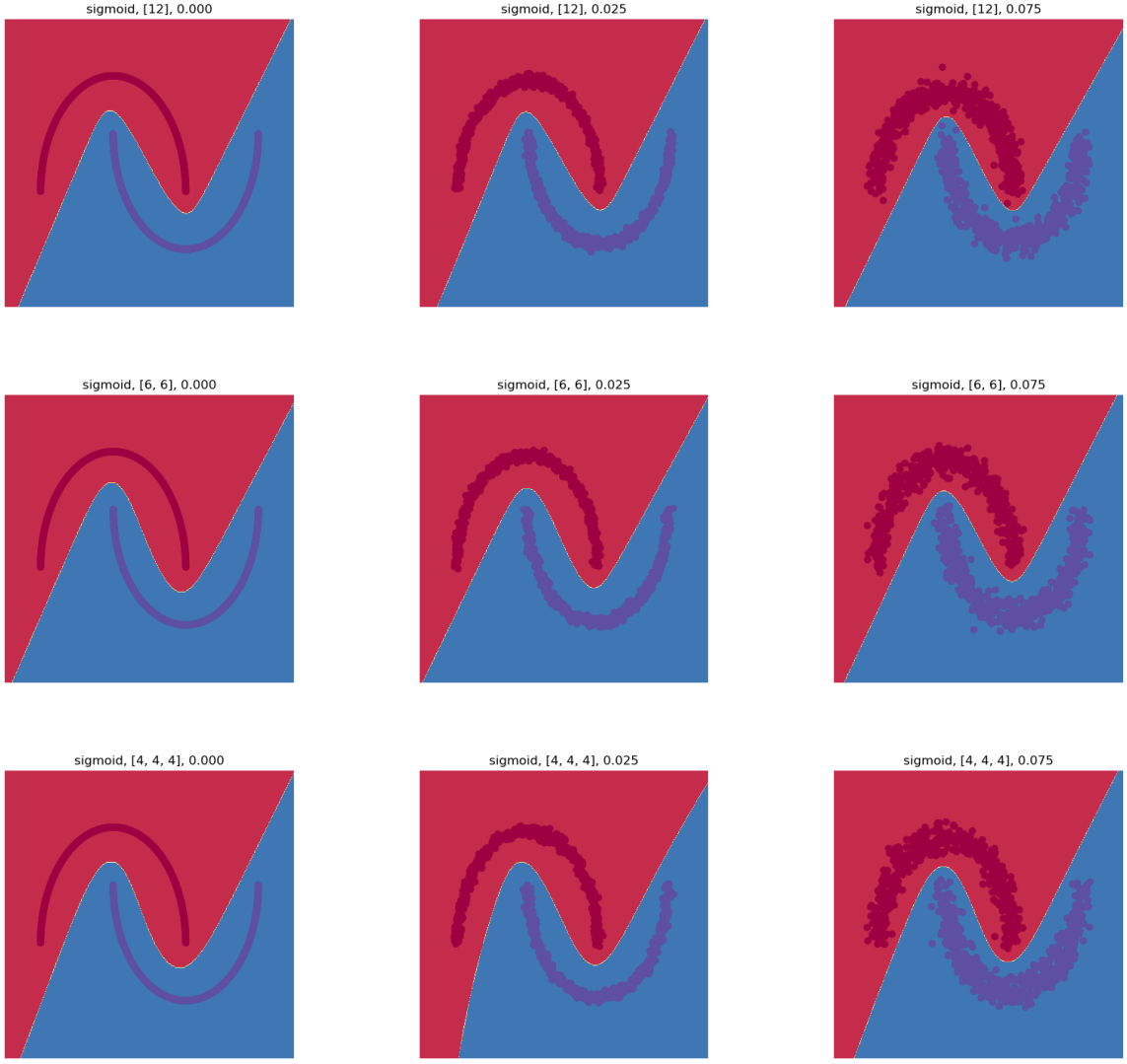
2.1 Experiments of n-layers NN

Each picture in the next few pages is a visual representation of 1000 distinct elements of some square $[a, b] \times [c, d] \subset \mathbb{R}^2$, each belonging to one of two classes (red or blue), where 0.000, 0.025, and 0.075 represent different levels of *noise* (0.000 represents no noise), and of the *decision boundary* of a trained neural network of 5 layers with $n_1 = 2$, $(n_2, n_3, n_4) \in \{(12, 12, 12), (16, 8, 16), (10, 16, 10)\}$, $n_5 = 2$,

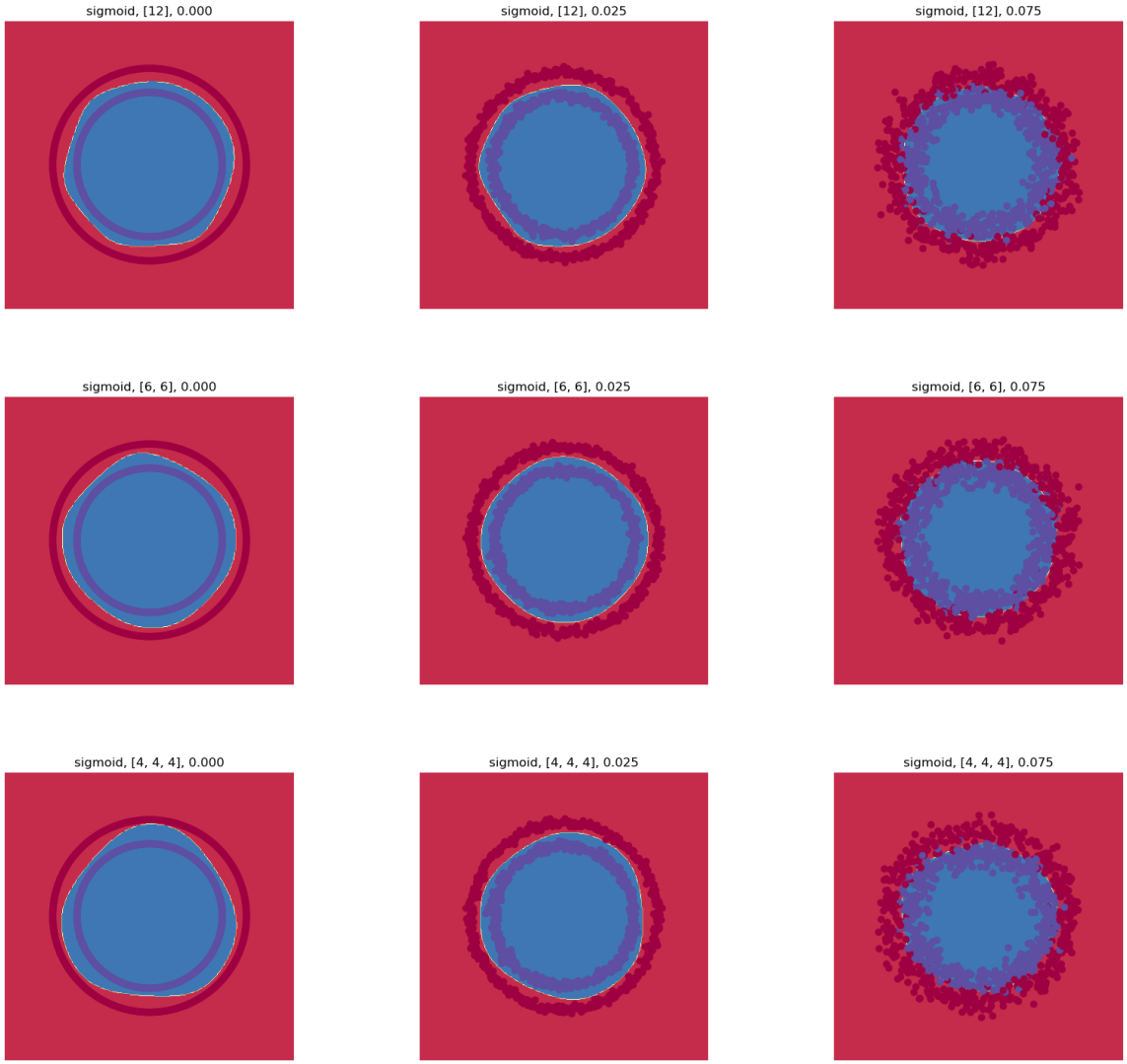
We implement n-layers neural network in Python as a class with the following fields and methods:

```
self.dimensions      # NumPy Array
self.activation_type # String
self.reg_lambda      # regularization coefficient
self.W               # Dictionary of NumPy Arrays
self.b               # Dictionary of NumPy Arrays
self.hidden          # Dictionary of NumPy Arrays
self.z               # Dictionary of NumPy Arrays
self.probs           #probability vector of the output
def __init__(self,
             dimensions = np.array([2, 10, 2]),
             activation_type = 'relu',
             reg_lambda = 0,
             random_seed = None)
def feed_forward(self, X)
def activation(self, x)
def activation_derivative(self, x)
def soft_max(self,x)
def back_propagation(self, X, y)
def train(self, X, y, train_rate, passes, print_loss, print_rate)
def calculate_loss(self, X, y)
def predict(self, X)
```

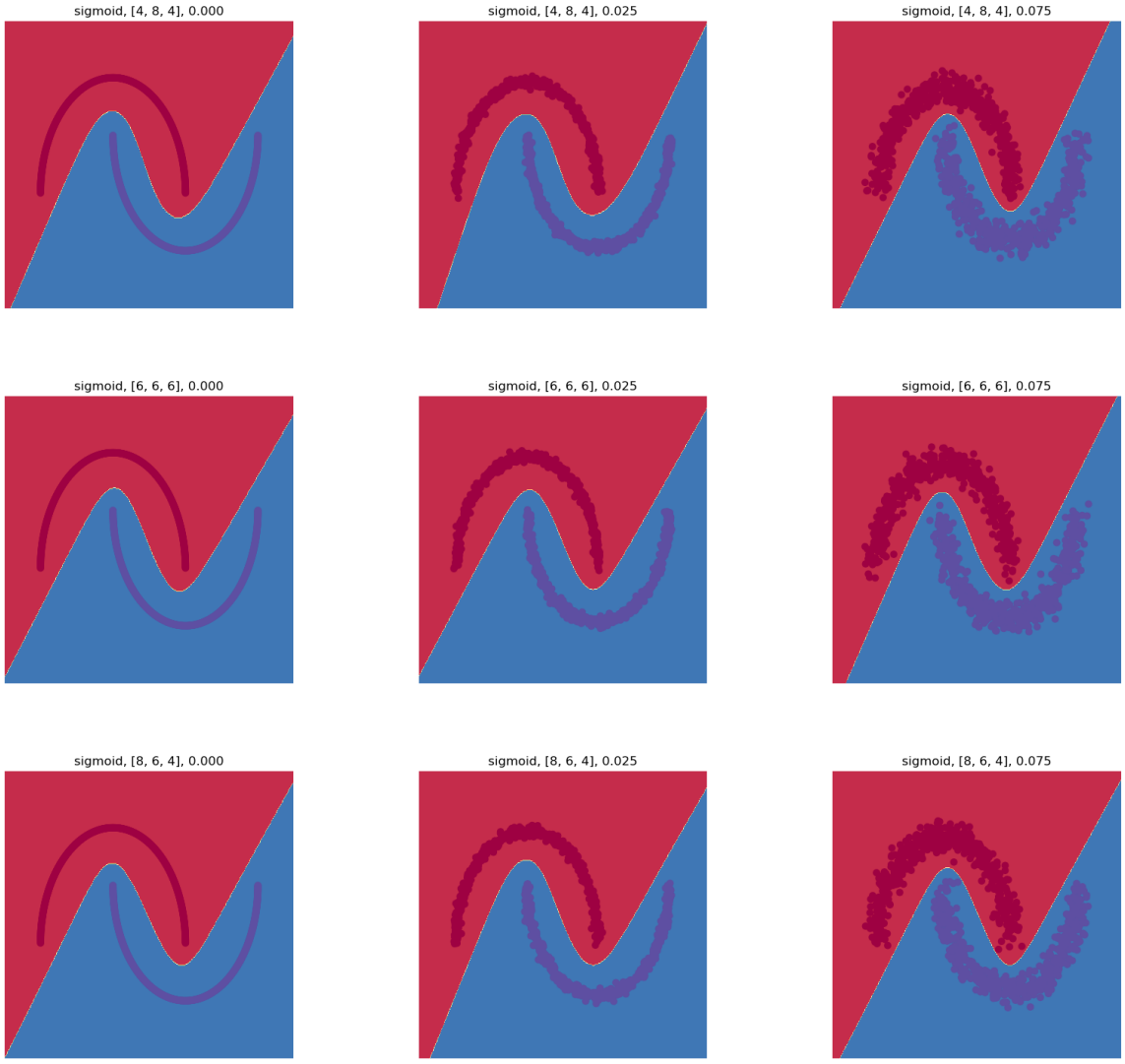
2.1.1 $n_2 + n_3 + n_4 = 12$, the number of layers:$l = (3, 4, 5)$, make_moons Data Set



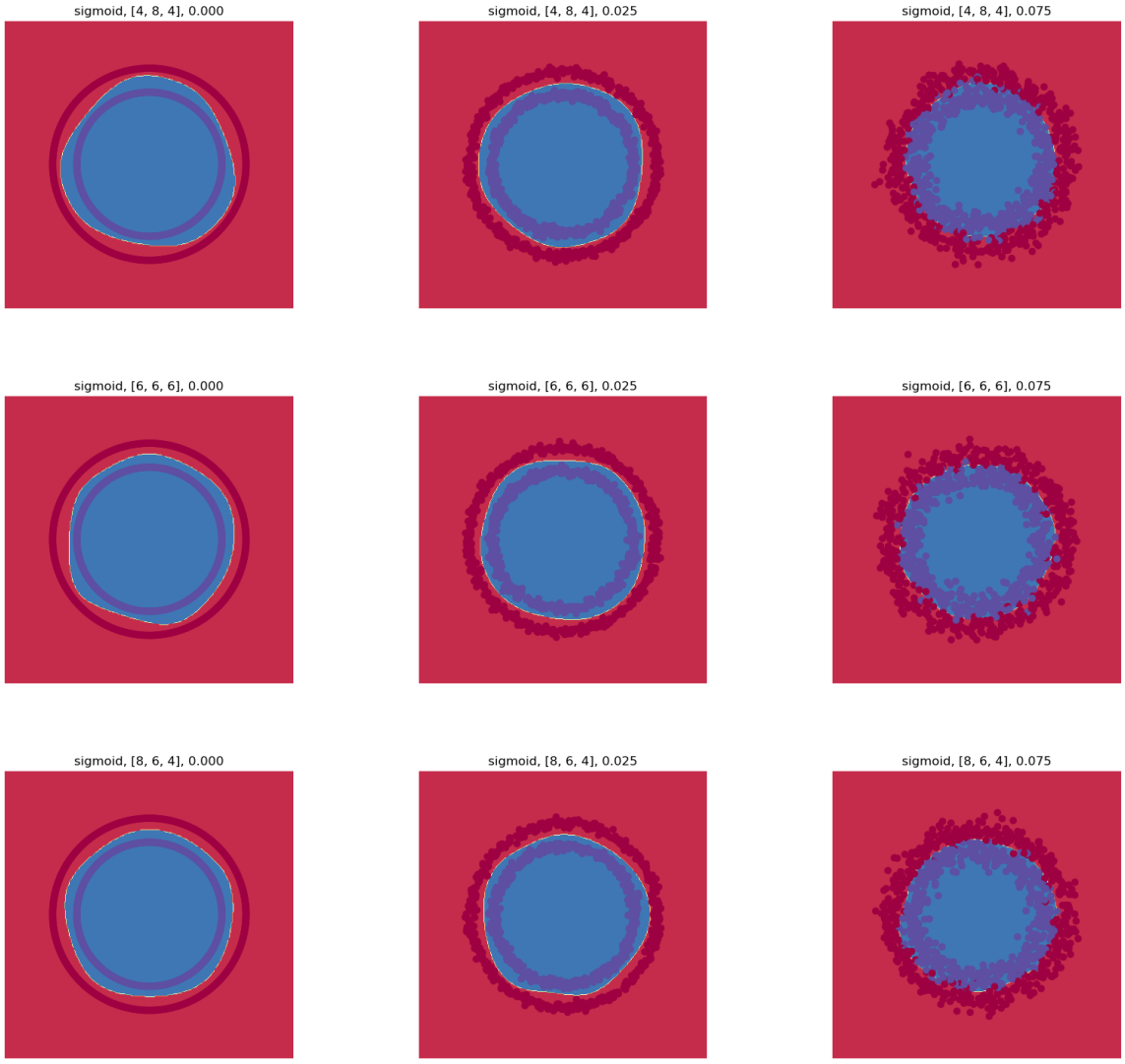
2.1.2 $n_2 + n_3 + n_4 = 12$, the number of layers:$l = (3, 4, 5)$, make_circles Data Set



2.1.3 $n_2, n_3, n_4 = (4,8,4), (6,6,6)(8,6,4)$, the number of layers:$l = 5$, make_moons Data Set



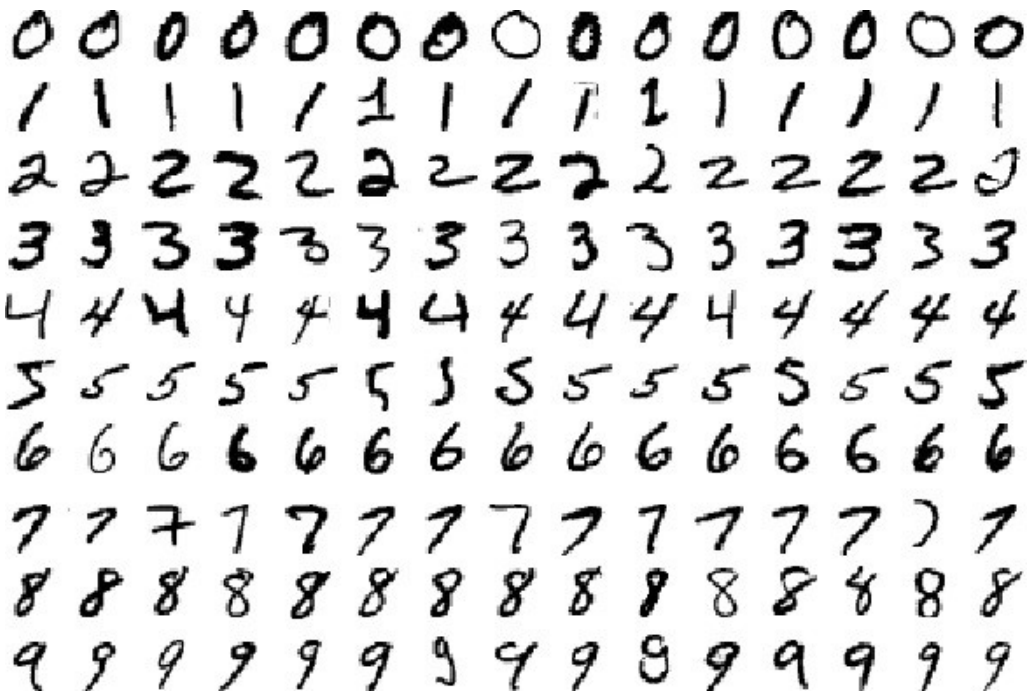
2.1.4 $n_2, n_3, n_4 = (4,8,4), (6,6,6)(8,6,4)$, the number of layers:$l = 5$, make_circles Data Set



3 Convolutional Neural Network

3.1 Architecture of CNN for MINST

The MINST data set is the collection of handwritten digits. It consists of scans (images) of digits that people wrote. The input is a 28 by 28 image of grey scale pixels, showing an image of a handwritten digit. The output is simply which of the 10 different digits ($0 \sim 9$). There are totally $N = 60,000$ examples in the data set. The digits have been size-normalized and centered in a fixed-size image (16×16 pixels) with values range from 0 to 255, where 0 is black and 255 is white. For simplicity, each image has been flattened and converted to a 1×784 vector. For output layer, there are $r = 10$ distinct classes $(0 \sim 9)$, presented by 10 independent standard basis vector $\{e_1, \ldots, e_{10}\}$, such that $e_1 = \{1, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$. e_1 stand for digit 0, e_2 stand for digit 1, and so on.



We implement a convolutional neural network with the following architecture:

```
Convolution(5-5-1-32) - ReLU - MaxPool(2-2) - Convolution(5-5-1-64) - ReLU
                      - MaxPool(2-2) - Flatten(1024) - ReLU - DropOut(.5) - SoftMax(10)
```

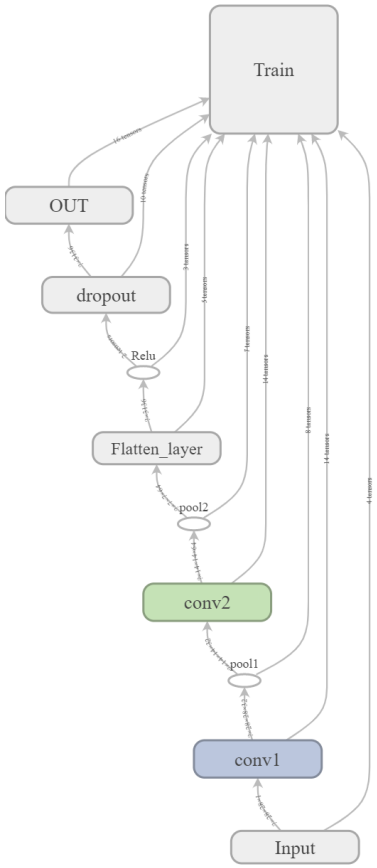


Figure 12: Graphs of Convolutional Neural Network

3.2 Implement with Tensorflow

Using Tensorflow, this can be done as follows:

```
with tf.name_scope('Input'):
    x = tf.placeholder(tf.float32, shape=[None, img_h, img_w, n_channels], name='X')
    y = tf.placeholder(tf.float32, shape=[None, n_classes], name='Y')

conv1 = conv_layer(x, filter_size1, num_filters1, stride1, name='conv1')
pool1 = max_pool(conv1, ksize=2, stride=2, name='pool1')
conv2 = conv_layer(pool1, filter_size2, num_filters2, stride2, name='conv2')
pool2 = max_pool(conv2, ksize=2, stride=2, name='pool2')
layer_flat = flatten_layer(pool2)
fc1 = fc_layer(layer_flat, h1, 'FC1', use_relu=True)
dropped = tf.nn.dropout(fc1, 0.5)
output_logits = fc_layer(dropped, n_classes, 'OUT', use_relu=False)

# Define the loss function, optimizer, and accuracy
with tf.variable_scope('Train'):
    with tf.variable_scope('Loss'):
```

```
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y,\\ logits=output_log
tf.summary.scalar('loss', loss)
with tf.variable_scope('Optimizer'):
    optimizer = tf.train.AdamOptimizer(learning_rate=lr, name='Adam-op').minimize(loss)
with tf.variable_scope('Accuracy'):
    correct_prediction = tf.equal(tf.argmax(output_logits, 1), tf.argmax(y, 1),\\ name='correct
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')
tf.summary.scalar('accuracy', accuracy)
```

We train our convolutional neural network for 5 epochs with a batch size of 50. This can be done as follows:

```
with tf.Session() as sess:
    sess.run(init)
    global_step = 0
    summary_writer = tf.summary.FileWriter(logs_path, sess.graph)
    num_tr_iter = int(len(y_train) / batch_size)
    for epoch in range(epochs):
        x_train, y_train = randomize(x_train, y_train)
        lr = lr_0 / (epoch + 1)
        for iteration in range(num_tr_iter):
            global_step += 1
            start = iteration * batch_size
            end = (iteration + 1) * batch_size
            x_batch, y_batch = get_next_batch(x_train, y_train, start, end)

            # Run optimization op (backprop)
            feed_dict_batch = {x: x_batch, y: y_batch}
            sess.run(optimizer, feed_dict=feed_dict_batch)

            if iteration % display_freq == 0:
                # Calculate and display the batch loss and accuracy
                _, summary_tr = sess.run([merged],feed_dict=feed_dict_batch)
                summary_writer.add_summary(summary_tr, global_step)

        # Run validation after every epoch
        feed_dict_valid = {x: x_valid, y: y_valid}
        _, summary_val = sess.run([merged], feed_dict=feed_dict_valid)
        summary_writer.add_summary(summary_val, global_step)
```

With this setup, we obtain a validation accuracy of 98.7%.

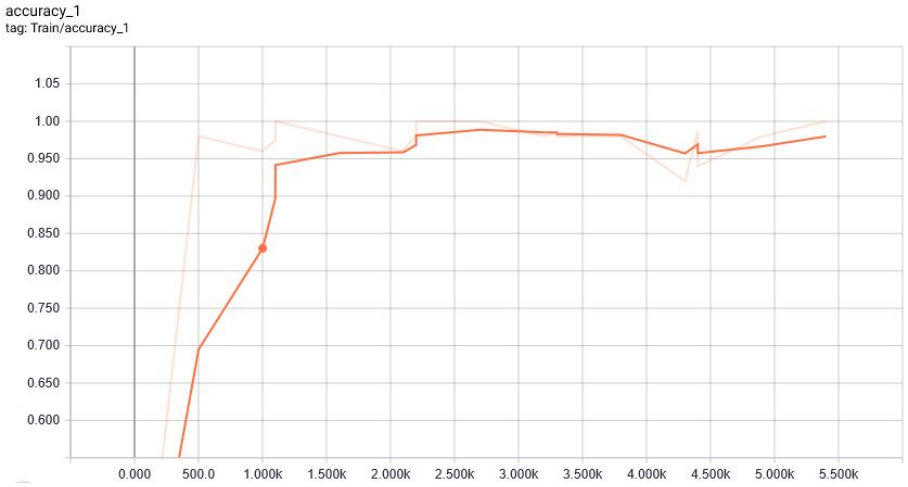


Figure 13: Accuracy

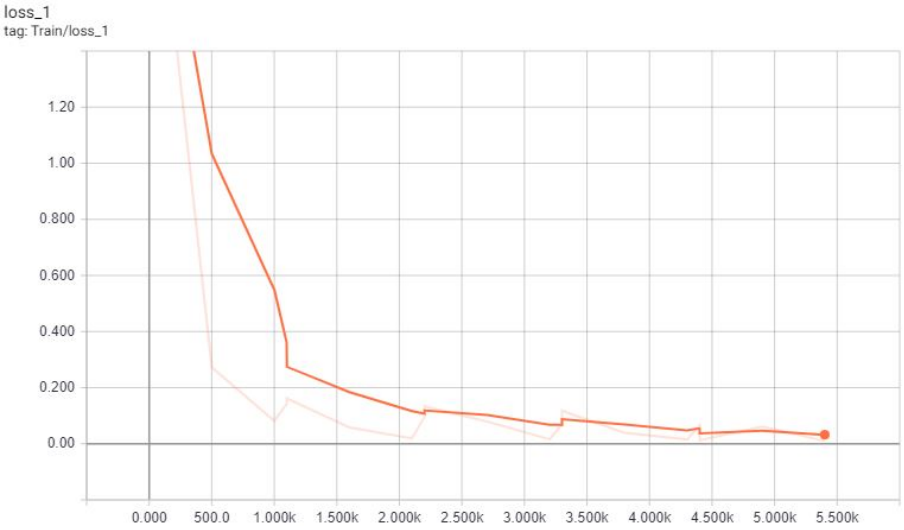
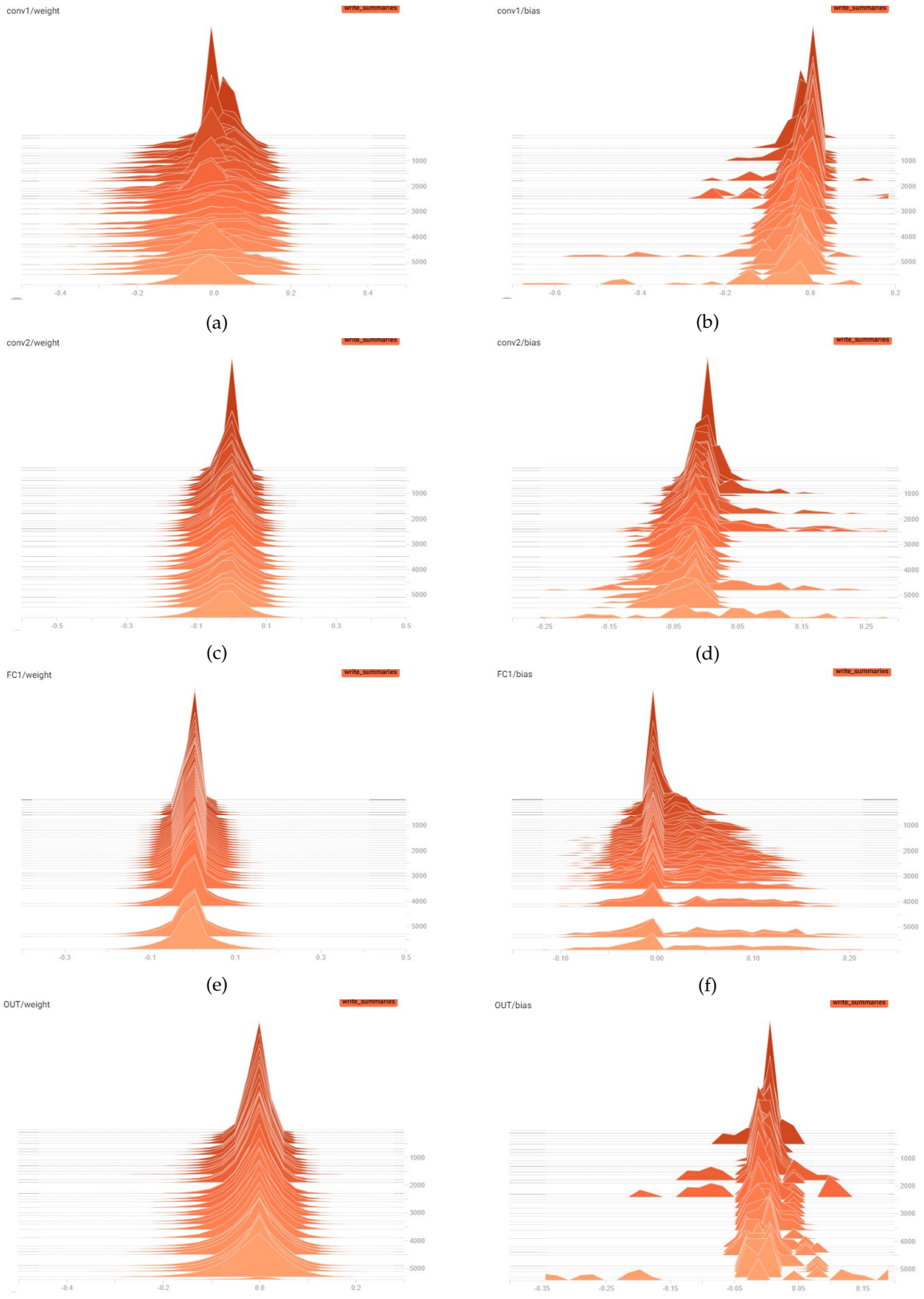


Figure 14: LOSS

In the next few pages, we visualize our training using Tensorboard.

3.3 Visualization with Tensorboard

Figure 15: Statistical plot (Max,Min,Mean,SD) of weights in each layers



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 16: Histogram of Weights in varied layers

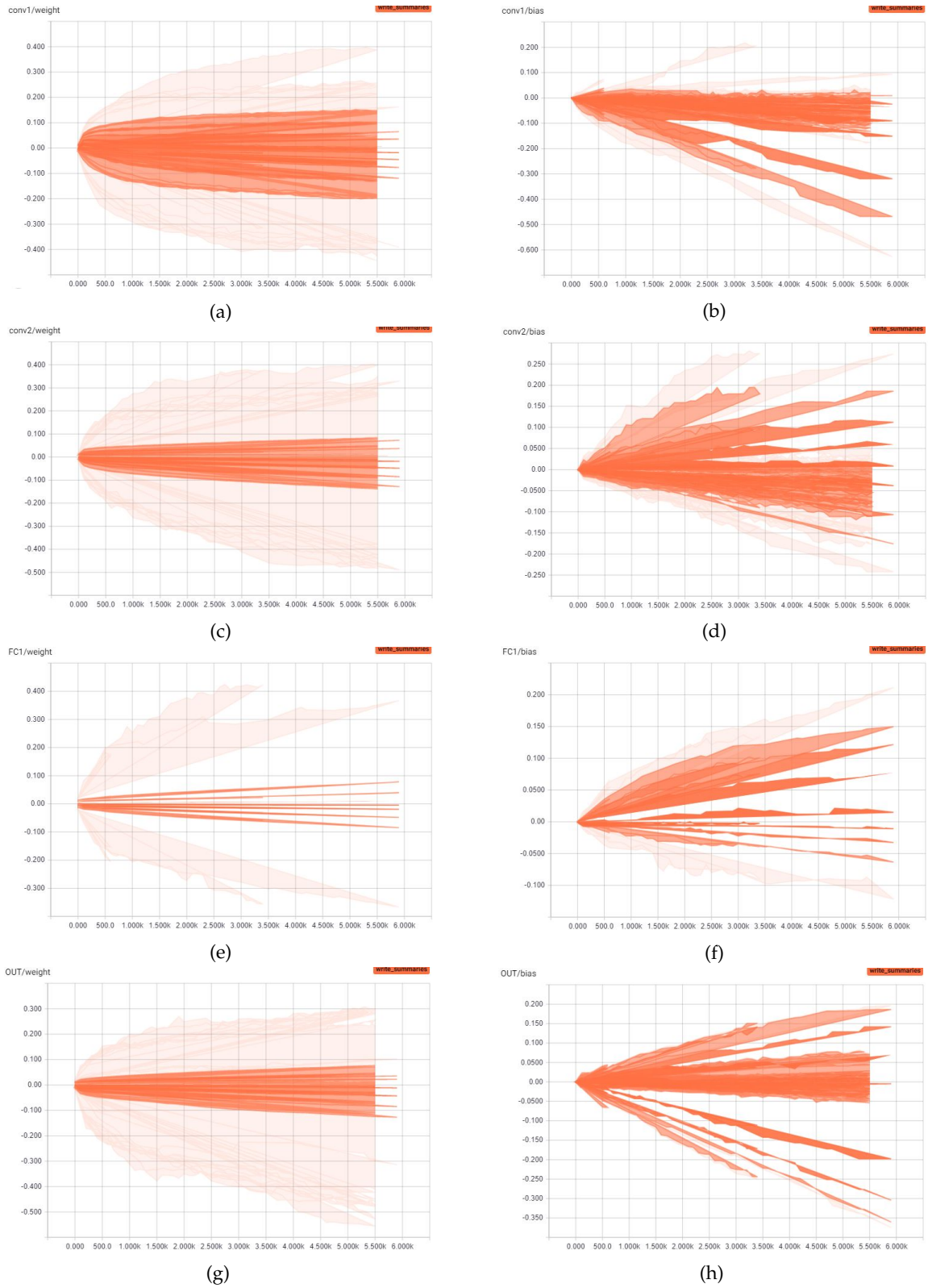


Figure 17: Distribution of Weights in varied layers