Assignment 1

ELEC 576 - Prof. Patel

Adrian E. Radillo – S01254516

October 5, 2017

Contents

GitHub: The Python code used for this project is available at the public repository, `https://github.com/aernesto/IntroMachineLearningCourse.git`

1 Backpropagation in a simple neural network

1.a Dataset

1.b Activation function

Our implementation of actFun is:

```python
def actFun(self, z, type):
        '''
        actFun computes the activation functions
        :param z: net input
        :param type: Tanh, Sigmoid, or ReLU
        :return: activations
        '''
        if type == 'Tanh':
            return np.tanh(z)
        elif type == 'Sigmoid':
            return 1. / (1 + np.exp(-z))
        elif type == 'ReLU':
            return np.maximum(z, np.zeros(z.shape))
```

The derivatives of the three activation functions (denoted by f) are:

Tanh $f'(z) = 1 - (f(z))^2$

Sigmoid $f'(z) = f(z)(1 - f(z))$

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

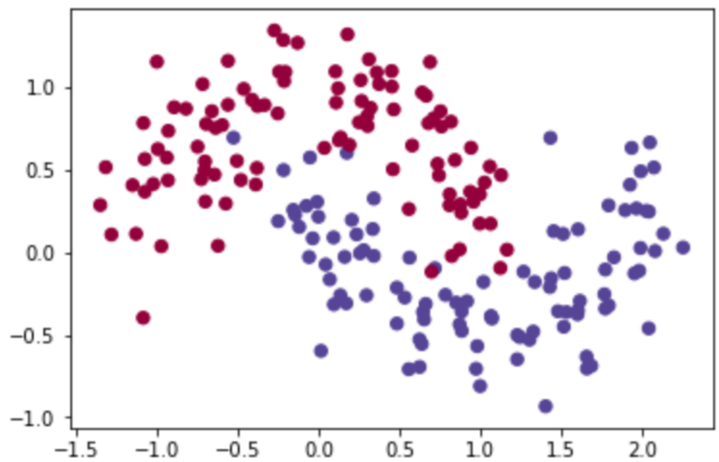

Figure 1: Data from Make-Moons dataset

ReLU

$$f'(z) = \begin{cases} 0 & \text{if } z < 0 \\ \text{undefined} & \text{if } z = 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Our implementation of diff_actFun therefore is:

```python
def diff_actFun(self, z, type):
        '''
        diff_actFun computes the derivatives of the activation functions wrt the net input
        :param z: net input
        :param type: Tanh, Sigmoid, or ReLU
        :return: the derivatives of the activation functions wrt the net input
        '''
        if type == 'Tanh':
            return 1 - (self.actFun(z, type))**2
        elif type == 'Sigmoid':
            return self.actFun(z, type) * (1 - self.actFun(z, type))
        elif type == 'ReLU':
            return (z > 0).astype(float)
```

1.c Build the neural network

Our implementation of feedforward and calculate_loss is:

```python
def feedforward(self, X, actFun_2):
                '''
        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:
        '''
        self.z1 = X.dot(self.W1) + np.tile(self.b1, (X.shape[0],1)) # dim 200 x nH
        self.a1 = actFun_2(self.z1)
        self.z2 = self.a1.dot(self.W2) + np.tile(self.b2, (self.a1.shape[0],1))
        exp_scores = np.exp(self.z2)
```

```python
        self.probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
        return None
```

```python
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
        y_hot = np.array([[1, 0] if yy == 0 else [0, 1] for yy in y])
        data_loss = -num_examples * np.sum(y_hot * np.log(self.probs))

        # 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 Backward pass - backpropagation

This is our implementation of backprop:

```python
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
        '''
        num_examples = len(X)
        delta3 = self.probs
        delta3[range(num_examples), y] -= 1
        dW2 = np.zeros(self.W2.shape)
        db2 = np.zeros(self.b2.shape)
        dW1 = np.zeros(self.W1.shape)
        db1 = np.zeros(self.b1.shape)

        for example in range(num_examples):
            dW2 += np.multiply(np.tile(delta3[example,:],(self.nn_hidden_dim,1)),
                               np.tile(self.a1[example,:],(self.nn_output_dim,1)).T)
            db2 += delta3[example,:]
            dW1 += np.tile(np.sum(np.tile(delta3[example,:],
                                          (self.nn_hidden_dim,1)) * self.W2,
                                  axis = 1), (self.nn_input_dim, 1)) * \
                   np.tile(self.diff_actFun(self.z1[example,:], self.actFun_type),
                           (self.nn_input_dim,1)) * \
                   np.tile(X[example, :], (self.nn_hidden_dim,1)).T
            db1 += np.sum(np.tile(delta3[example,:], (self.nn_hidden_dim,1)) * self.W2, axis = 1) * \
                   self.diff_actFun(self.z1[example,:], self.actFun_type)

        dW2 /= num_examples
        db2 /= num_examples
        dW1 /= num_examples
        db1 /= num_examples
```

```
        return dW1, dW2, db1, db2
```

1.e Time to have fun - training

We present below the results of training our network with three distinct activation functions, always with 3 hidden nodes:

```
Loss after iteration 0: 0.582912
Loss after iteration 1000: 0.332595
Loss after iteration 2000: 0.311568
Loss after iteration 3000: 0.306815
Loss after iteration 4000: 0.304534
Loss after iteration 5000: 0.303069
Loss after iteration 6000: 0.301871
Loss after iteration 7000: 0.300608
Loss after iteration 8000: 0.299103
Loss after iteration 9000: 0.297249
Loss after iteration 10000: 0.294981
Loss after iteration 11000: 0.292257
Loss after iteration 12000: 0.289076
Loss after iteration 13000: 0.285479
Loss after iteration 14000: 0.281548
Loss after iteration 15000: 0.277399
Loss after iteration 16000: 0.273161
Loss after iteration 17000: 0.268958
Loss after iteration 18000: 0.264898
Loss after iteration 19000: 0.261058
```

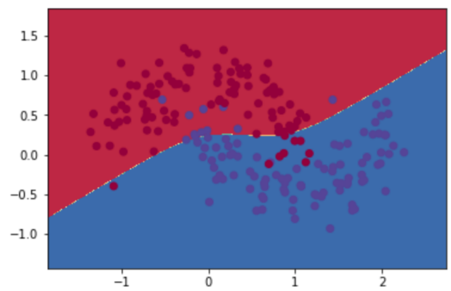

Figure 2: 'Tanh' activation function and 3 hidden nodes

```
Loss after iteration 0: 0.622949
Loss after iteration 1000: 0.327359
Loss after iteration 2000: 0.304278
Loss after iteration 3000: 0.300512
Loss after iteration 4000: 0.298879
Loss after iteration 5000: 0.298062
Loss after iteration 6000: 0.297413
Loss after iteration 7000: 0.296828
Loss after iteration 8000: 0.296323
Loss after iteration 9000: 0.295605
Loss after iteration 10000: 0.294933
Loss after iteration 11000: 0.294127
Loss after iteration 12000: 0.292910
Loss after iteration 13000: 0.291803
Loss after iteration 14000: 0.290787
Loss after iteration 15000: 0.289947
Loss after iteration 16000: 0.289227
Loss after iteration 17000: 0.288511
Loss after iteration 18000: 0.287823
Loss after iteration 19000: 0.287297
```

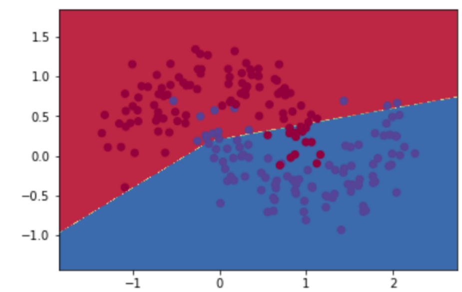

Figure 3: 'ReLU' activation function and 3 hidden nodes

```
Loss after iteration 0: 0.656440
Loss after iteration 1000: 0.525975
Loss after iteration 2000: 0.430989
Loss after iteration 3000: 0.382175
Loss after iteration 4000: 0.359215
Loss after iteration 5000: 0.347746
Loss after iteration 6000: 0.341537
Loss after iteration 7000: 0.337960
Loss after iteration 8000: 0.335806
Loss after iteration 9000: 0.334462
Loss after iteration 10000: 0.333597
Loss after iteration 11000: 0.333025
Loss after iteration 12000: 0.332632
Loss after iteration 13000: 0.332354
Loss after iteration 14000: 0.332148
Loss after iteration 15000: 0.331990
Loss after iteration 16000: 0.331864
Loss after iteration 17000: 0.331761
Loss after iteration 18000: 0.331673
Loss after iteration 19000: 0.331598
```

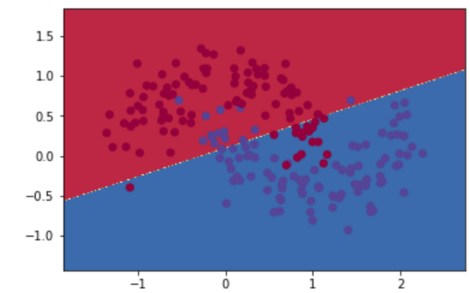

Figure 4: 'Sigmoid' activation function and 3 hidden nodes

The 'Tanh' activation function seems to result in the most flexible decision boundary. It also produces the smallest loss at the end of the training. The 'Sigmoid' activation function seems to be the worst of the three, in terms of decision boundary, which is a simple line, and final loss.

When retraining the network with 'Tanh' and 50 hidden nodes, we obtain the following output:

```
Loss after iteration 0: 0.506189
Loss after iteration 1000: 0.297850
Loss after iteration 2000: 0.284993
Loss after iteration 3000: 0.276774
Loss after iteration 4000: 0.270326
Loss after iteration 5000: 0.264630
Loss after iteration 6000: 0.258983
Loss after iteration 7000: 0.252944
Loss after iteration 8000: 0.246327
Loss after iteration 9000: 0.239181
Loss after iteration 10000: 0.231729
Loss after iteration 11000: 0.224278
Loss after iteration 12000: 0.217131
Loss after iteration 13000: 0.210523
Loss after iteration 14000: 0.204597
Loss after iteration 15000: 0.199411
Loss after iteration 16000: 0.194953
Loss after iteration 17000: 0.191166
Loss after iteration 18000: 0.187975
Loss after iteration 19000: 0.185300
```

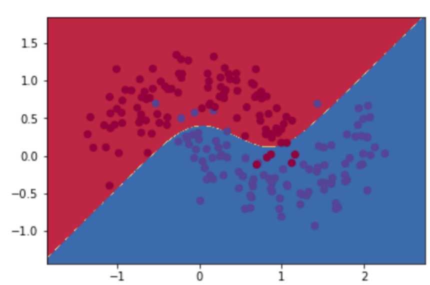

Figure 5: 'Tanh' activation function and 50 hidden nodes

In comparing figures 2 and 5, we notice that augmenting the dimension of the hidden layer has two main effects. Firstly, it reduces the final loss, and secondly, it produces a decision boundary which has more degrees of freedom.

1.f Even more fun - training a deeper network!!!

Our full new code is presented here:

```
class NeuralNetwork(object):
    """
    This class builds and trains a neural network
    """
    def __init__(self, layer_sizes, num_examples, actFun_type='Tanh', reg_lambda
        =0.01, seed=0):
        '''
        :param layer_sizes: list of positive integers.
        :param actFun_type: type of activation function. 3 options: 'tanh', 'sigmoid
            ', 'relu'
        :param reg_lambda: regularization coefficient
        :param seed: random seed
        '''

        '''
        !!NOTE!! In this code, layer numbering is inverted (this is hidden to the
            user when he
        instantiates the class though), therefore the first integer of the list
            attribute self.layer_sizes
        is dim of output layer and last integer is dimension of the input layer.
        '''
        self.layer_sizes = list(reversed(layer_sizes))
        self.num_layers = len(self.layer_sizes)

        # Instantiate all the layers, except for the input which contains no
            intrinsic weights.
        layers = {}
        for n in np.arange(1, self.num_layers):
            layers[str(n)] = Layer(layer_size = self.layer_sizes[n - 1],
                                   prec_layer_size = self.layer_sizes[n],
                                       num_examples = num_examples,
                                   seed = seed, depth = n)

        # Instantiate input layer separately since no preceding layer
        layers[str(self.num_layers)] = Layer(layer_size = self.layer_sizes[-1],
            prec_layer_size = 0,
                                             num_examples = num_examples, seed =
                                                 seed,
                                             depth = self.num_layers)

        self.layers = layers
        self.actFun_type = actFun_type
        self.reg_lambda = reg_lambda

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

```
        elif type == 'Sigmoid':
            return 1. / (1 + np.exp(-z))
        elif type == 'ReLU':
            return np.maximum(z, np.zeros(z.shape))

    def diff_actFun(self, z, type):
        '''
        diff_actFun computes the derivatives of the activation functions wrt the net input
        :param z: net input
        :param type: Tanh, Sigmoid, or ReLU
        :return: the derivatives of the activation functions wrt the net input
        '''
        if type == 'Tanh':
            return 1 - (self.actFun(z, type))**2
        elif type == 'Sigmoid':
            return self.actFun(z, type) * (1 - self.actFun(z, type))
        elif type == 'ReLU':
            return (z > 0).astype(float)

    def feedforward(self, X, actFun_2):
        '''
        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:
        '''
        self.layers[str(self.num_layers)].activations = X
        for n in reversed(np.arange(1, self.num_layers)):
            self.layers[str(n)].feedforward(inputs = self.layers[str(n + 1)].activations,
                                                                   actFun = actFun_2)

        # correct for activations of output layer
        exp_scores = np.exp(self.layers['1'].linear_output)
        self.probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
        self.layers['1'].activations = self.probs
        return None

    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))
#         self.feedforward(X, self.actFun)  # WHY NOT THIS?
        # Calculating the loss
        y_hot = np.array([[1, 0] if yy == 0 else [0, 1] for yy in y])
        data_loss = -np.sum(y_hot * np.log(self.probs))

        # Add regulatization term to loss (optional)
        coef = 0
        for L in np.arange(1,self.num_layers):
            coef += np.sum(np.square(self.layers[str(L)].weights))
```

```
        data_loss += self.reg_lambda / ((self.num_layers - 1) * coef)
        return (1. / num_examples) * data_loss

    def predict(self, X):
        '''
        predict infers the label of a given data point X
        :param X: input data
        :return: label inferred
        '''
        self.feedforward(X, lambda x: self.actFun(x, type=self.actFun_type))
        return np.argmax(self.probs, axis=1)

    def backprop(self, X, y):
        '''
        backprop 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/db1, dL/dW2, dL/db2
        '''
        num_examples = len(X)
        delta3 = self.probs
        delta3[range(num_examples), y] -= 1
        for L in np.arange(1,self.num_layers):
            if L == 1:
                self.layers[str(L)].backprop(lambda x: self.diff_actFun(x, type=self.actFun_type),
                                             prev_activations = self.layers[str(L + 1)].activations,
                                             delta_y = delta3)
            else:
                self.layers[str(L)].backprop(lambda x: self.diff_actFun(x, type=self.actFun_type),
                                             prev_activations = self.layers[str(L + 1)].activations,
                                             post_errors = self.layers[str(L - 1)].error,
                                             post_weights = self.layers[str(L - 1)].weights)

        return None

    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
            self.backprop(X, y)

            # Add regularization terms (b1 and b2 don't have regularization terms)
```

```
            for L in np.arange(1,self.num_layers):
                self.layers[str(L)].dW += self.reg_lambda * self.layers[str(L)].weights

                # Gradient descent parameter update
                self.layers[str(L)].weights += -epsilon * self.layers[str(L)].dW
                self.layers[str(L)].biases += -epsilon * self.layers[str(L)].db

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

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

class Layer(object):
    def __init__(self, layer_size, prec_layer_size, num_examples, seed, depth):
        self.num_examples = num_examples
        self.depth = depth
        self.layer_size = layer_size
        self.prec_layer_size = prec_layer_size
        self.linear_output = np.zeros((num_examples, self.layer_size))
        self.activations = np.zeros(self.linear_output.shape)
        self.error = np.zeros((self.num_examples, self.layer_size))

        # initialize the weights and biases in the network
        np.random.seed(seed)
        self.weights = np.random.randn(self.prec_layer_size, self.layer_size) / np.sqrt(self.prec_layer_size)
        self.biases = np.zeros((1, self.layer_size))

        # differentials for weights and biases
        self.dW = np.zeros(self.weights.shape)
        self.db = np.zeros(self.biases.shape)

    def backprop(self, diff_actFun, prev_activations,
                 post_errors = None, post_weights = None, delta_y = None):
        '''
        returns derivative of the loss function with respect to param
        '''

        # compute deltas
        if self.depth == 1:
            self.error = delta_y
        else:
            self.error = diff_actFun(self.linear_output) * post_errors.dot(post_weights.T)

        # compute weight or bias differential as average over examples
```

```
        for example in range(self.num_examples):
            vec1 = prev_activations[example, :]
            vec1.shape = (1, len(vec1))
            vec2 = self.error[example, :]
            vec2.shape = (1, len(vec2))
            self.dW += vec1.T.dot(vec2)

        self.dW /= self.num_examples
        self.db = self.error.mean(axis = 0)

    def feedforward(self, inputs, actFun):
        self.linear_output = inputs.dot(self.weights) + np.tile(self.biases, (inputs.shape[0], 1))
        self.activations = actFun(self.linear_output)
        return None

def main():
    # generate and visualize Make-Moons dataset
    X, y = generate_data()
#     plt.scatter(X[:, 0], X[:, 1], s=40, c=y, cmap=plt.cm.Spectral)
#     plt.show()
    model = NeuralNetwork(layer_sizes=[2, 3, 4, 2], num_examples= 200, actFun_type='Tanh')
    # model.fit_model(X,y_hot)
    aa = datetime.datetime.now().replace(microsecond=0)
    model.fit_model(X,y)
    bb = datetime.datetime.now().replace(microsecond=0)
    print('total time for training (hours:min:sec)', bb - aa)
    model.visualize_decision_boundary(X,y)
#     print(np.shape(self.W1), np.shape(self.W2))

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

We trained a network with 2 hidden layers, first of sizes 3 and 4, and then, with sizes 30 and 40. The results are below, in figures 6 and 7.

```
Loss after iteration 0: 0.865495
Loss after iteration 1000: 0.312576
Loss after iteration 2000: 0.301356
Loss after iteration 3000: 0.289616
Loss after iteration 4000: 0.273894
Loss after iteration 5000: 0.252445
Loss after iteration 6000: 0.226776
Loss after iteration 7000: 0.201108
Loss after iteration 8000: 0.179541
Loss after iteration 9000: 0.163622
Loss after iteration 10000: 0.152680
Loss after iteration 11000: 0.145359
Loss after iteration 12000: 0.140475
Loss after iteration 13000: 0.137190
Loss after iteration 14000: 0.134956
Loss after iteration 15000: 0.133417
Loss after iteration 16000: 0.132344
Loss after iteration 17000: 0.131587
Loss after iteration 18000: 0.131046
Loss after iteration 19000: 0.130655
total time for training (hours:min:sec) 0:00:47
```

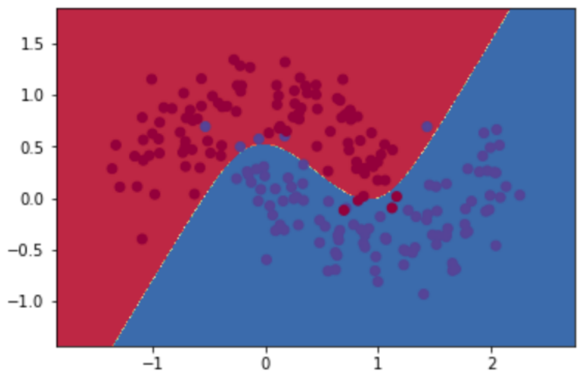

Figure 6: ‘Tanh’ activation function and 2 hidden layers with sizes 3 and 4

```
Loss after iteration 0: 1.215800
Loss after iteration 1000: 0.291953
Loss after iteration 2000: 0.262270
Loss after iteration 3000: 0.230261
Loss after iteration 4000: 0.198338
Loss after iteration 5000: 0.172955
Loss after iteration 6000: 0.155583
Loss after iteration 7000: 0.144192
Loss after iteration 8000: 0.136598
Loss after iteration 9000: 0.131339
Loss after iteration 10000: 0.127539
Loss after iteration 11000: 0.124682
Loss after iteration 12000: 0.122462
Loss after iteration 13000: 0.120688
Loss after iteration 14000: 0.119245
Loss after iteration 15000: 0.118056
Loss after iteration 16000: 0.117069
Loss after iteration 17000: 0.116248
Loss after iteration 18000: 0.115564
Loss after iteration 19000: 0.114997
total time for training (hours:min:sec) 0:01:04
```

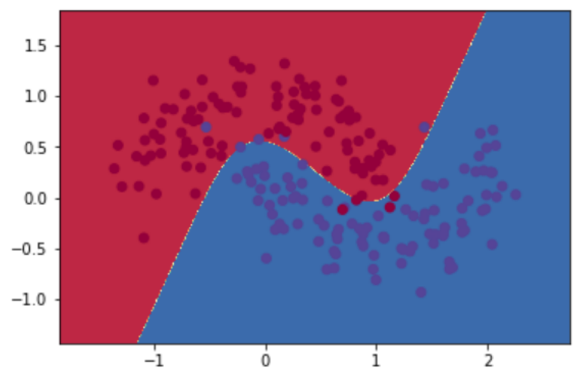

Figure 7: ‘Tanh’ activation function and 2 hidden layers with sizes 30 and 40

2 Training a simple Deep Convolutional Network on MNIST

2.a Build and train a 4-layer DCN

2.a.1 Build network

Here are our personal implementations:

```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
    '''
    initial = tf.truncated_normal(shape, stddev=0.1)
    W = tf.Variable(initial, name = "W")

    return W
```

```python
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
    '''
    initial = tf.constant(0.1, shape=shape)
    b = tf.Variable(initial, name = "b")
    return b
```

```
def conv2d(x, W):
    '''
    Perform 2-D convolution
    :param x: input tensor of size [N, W, H, Cin] where
    N: the number of images
    W: width of images
    H: height of images
    Cin: the number of channels of images
    :param W: weight tensor [w, h, Cin, Cout]
    w: width of the filters
    h: height of the filters
    Cin: the number of the channels of the filters = the number of channels of images
    Cout: the number of filters
    :return: a tensor of features extracted by the filters, a.k.a. the results after convolution
    '''
    h_conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
    return h_conv
```

```
def max_pool_2x2(x):
    '''
    Perform non-overlapping 2-D maxpooling on 2x2 regions in the input data
    :param x: input data
    :return: the results of maxpooling (max-marginalized + downsampling)
    '''
    h_max = tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')
    return h_max
```

2.a.2 Set up training

The rest of our dcn˙mnist.py file looks like this:

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

    start_time = time.time() # start timing

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

    # 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

    # setup training

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

    # Add a scalar summary for the snapshot loss.
    tf.summary.scalar(cross_entropy.op.name, cross_entropy)

    # Build the summary operation based on the TF collection of Summaries.
    summary_op = tf.summary.merge_all()

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

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

    # Instantiate a SummaryWriter to output summaries and the Graph.
    summary_writer = tf.summary.FileWriter(result_dir, sess.graph)

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

    # run the training
    for i in range(max_step):
        batch = mnist.train.next_batch(50) # make the data batch, which is used in
            the training iteration.
                                           # the batch size is 50
```

```
        if i%100 == 0:
            # output the training accuracy every 100 iterations
            train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_:batch[1],
                keep_prob: 1.0})
            print("step %d, training accuracy %g"%(i, train_accuracy))

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

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

2.a.3 Run training

After training, the final test accuracy of this network is 98.87%, as shown in the figure below:

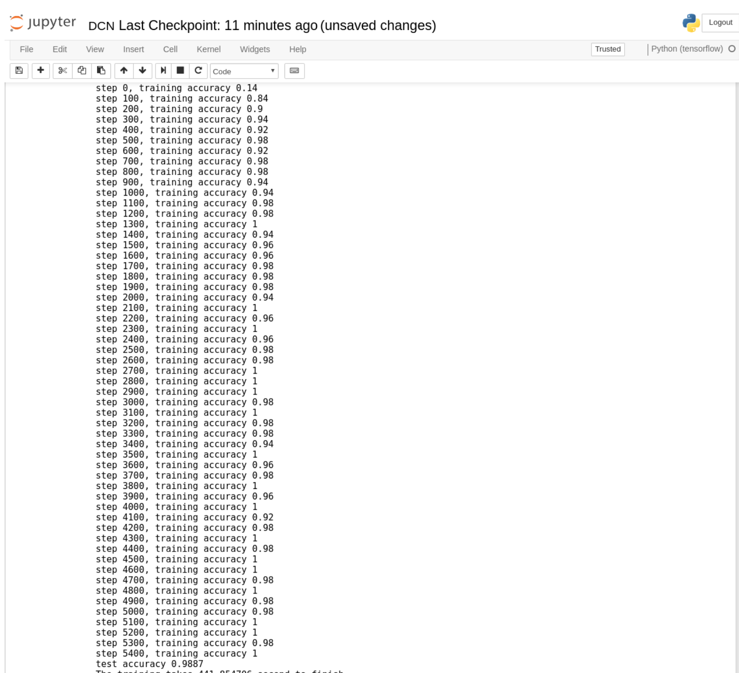

Figure 8: Console output of first training of DCN on MNIST dataset

2.a.4 Visualize training

The visualization available via TensorBoard at this point are:

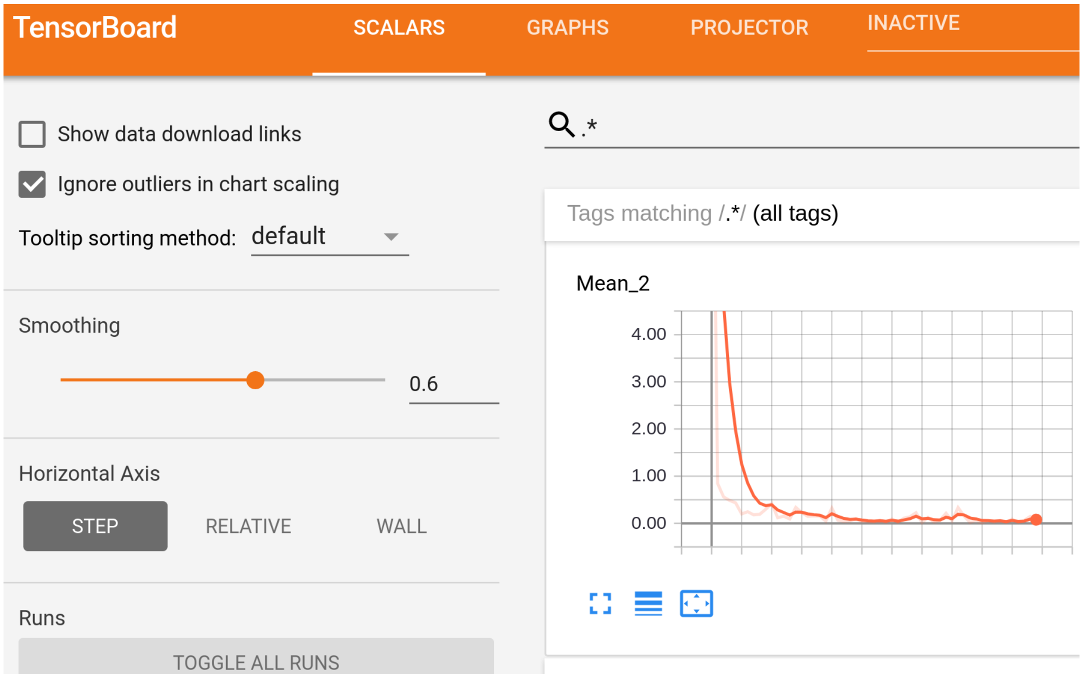

Figure 9: TensorBoard plot of loss as function of iterations, for first DCN training

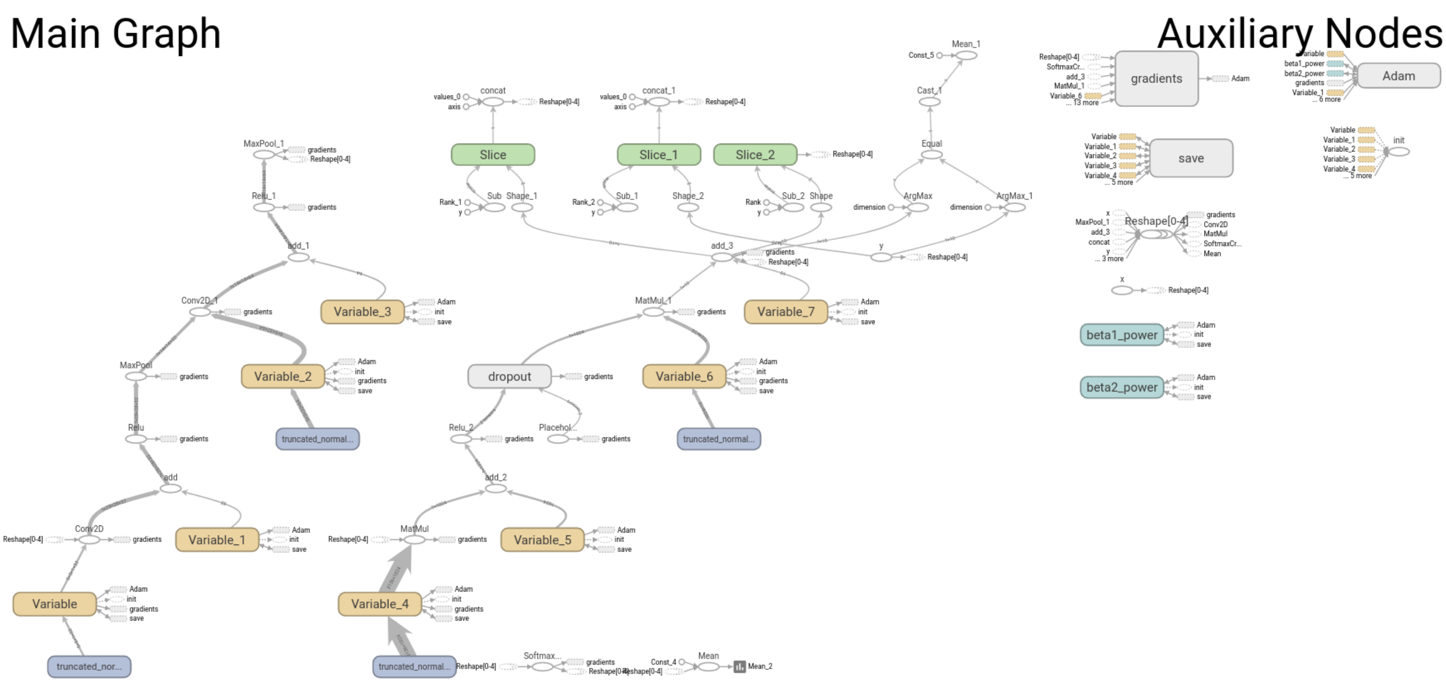

Figure 10: TensorBoard graph for first DCN training

2.b More on visualizing your training

After using namescopes and summaries in our code, the console output becomes: and TensorBoard graphs are:

2.c Time for more fun!

```
step 0, training accuracy 0.06
test accuracy 0.0981
validation accuracy 0.0954
step 100, training accuracy 0.8
step 200, training accuracy 0.8
step 300, training accuracy 0.82
step 400, training accuracy 0.92
step 500, training accuracy 0.98
step 600, training accuracy 0.94
step 700, training accuracy 0.98
step 800, training accuracy 0.92
step 900, training accuracy 0.96
step 1000, training accuracy 0.88
step 1100, training accuracy 0.94
test accuracy 0.9637
validation accuracy 0.9656
step 1200, training accuracy 0.96
step 1300, training accuracy 1
step 1400, training accuracy 0.94
step 1500, training accuracy 0.94
step 1600, training accuracy 0.96
step 1700, training accuracy 0.96
step 1800, training accuracy 1
step 1900, training accuracy 1
step 2000, training accuracy 0.98
step 2100, training accuracy 1
step 2200, training accuracy 1
test accuracy 0.9766
validation accuracy 0.9774
step 2300, training accuracy 0.98
step 2400, training accuracy 0.96
step 2500, training accuracy 1
step 2600, training accuracy 0.96
step 2700, training accuracy 1
step 2800, training accuracy 0.96
step 2900, training accuracy 0.98
step 3000, training accuracy 0.94
step 3100, training accuracy 0.96
step 3200, training accuracy 0.94
step 3300, training accuracy 0.98
test accuracy 0.9821
validation accuracy 0.9832
step 3400, training accuracy 0.98
step 3500, training accuracy 1
step 3600, training accuracy 1
step 3700, training accuracy 0.96
step 3800, training accuracy 1
step 3900, training accuracy 0.98
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
test accuracy 0.9845
validation accuracy 0.9868
step 4500, training accuracy 0.98
step 4600, training accuracy 1
step 4700, training accuracy 1
step 4800, training accuracy 1
step 4900, training accuracy 0.98
step 5000, training accuracy 1
step 5100, training accuracy 0.98
step 5200, training accuracy 1
step 5300, training accuracy 1
step 5400, training accuracy 1
The training takes 467.104159 second to finish
```

Figure 11: Console output for second training of DCN

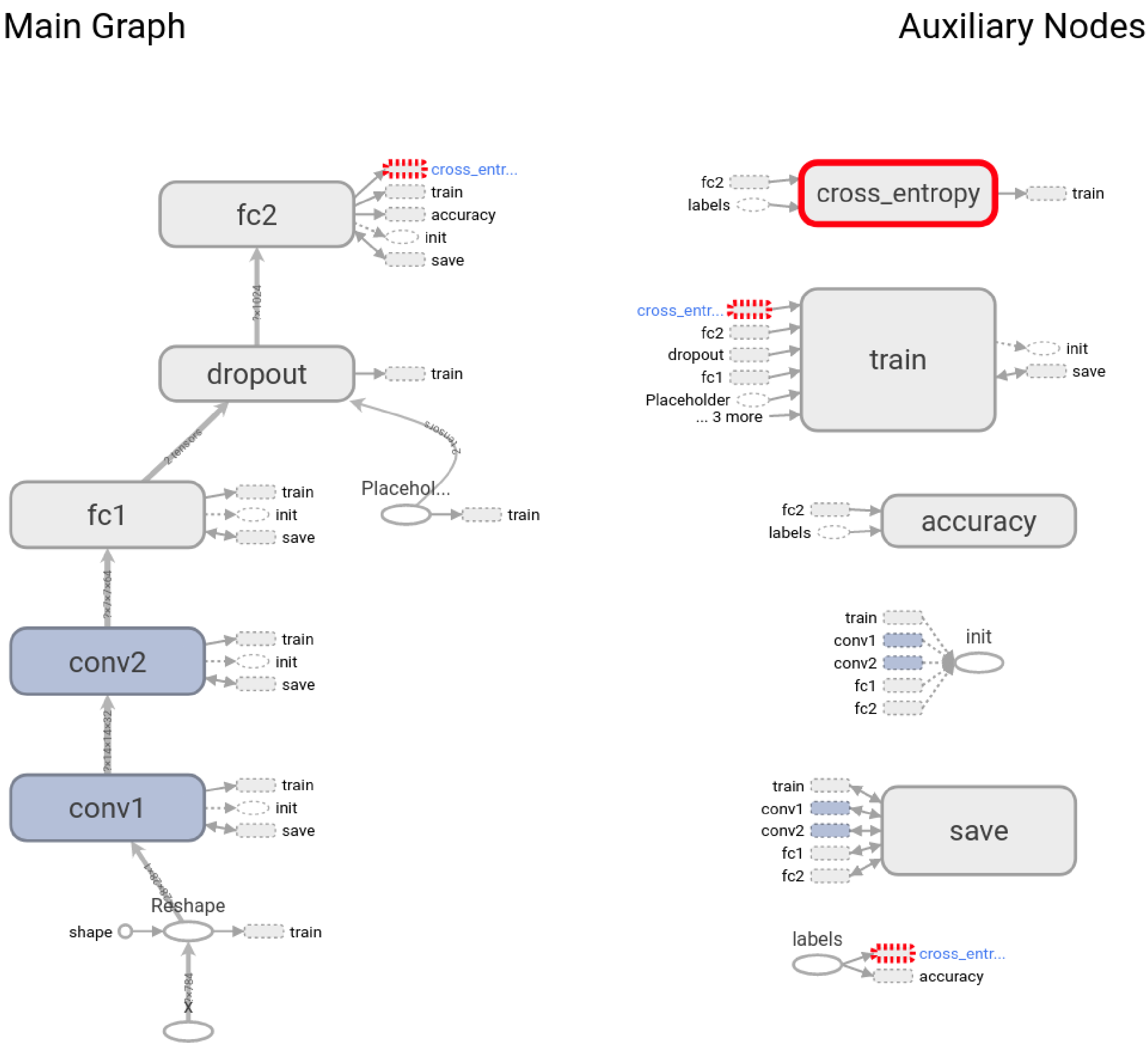

Figure 12: TensorBoard graph for second DCN training

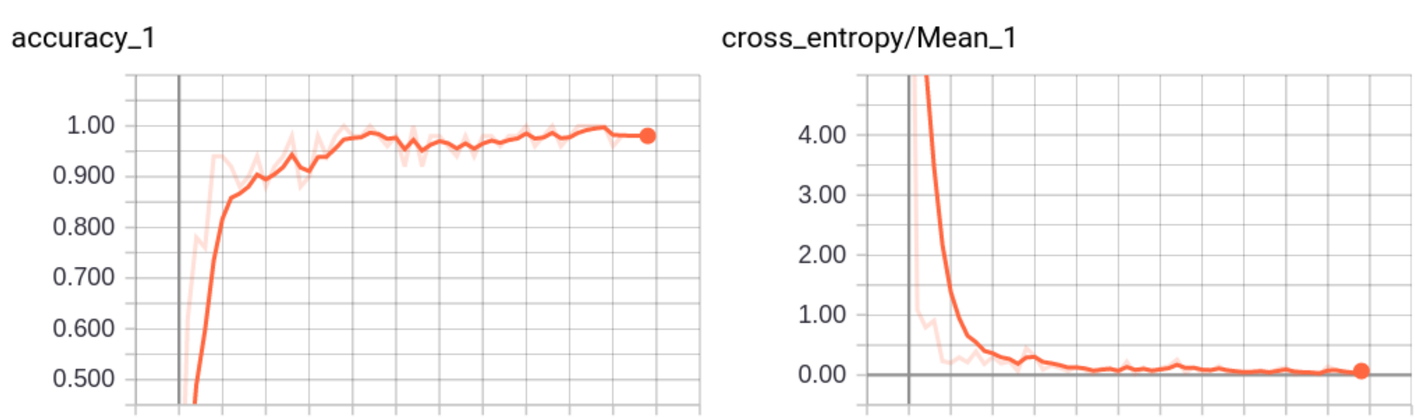

Figure 13: TensorBoard plot of scalar summaries, for second DCN training

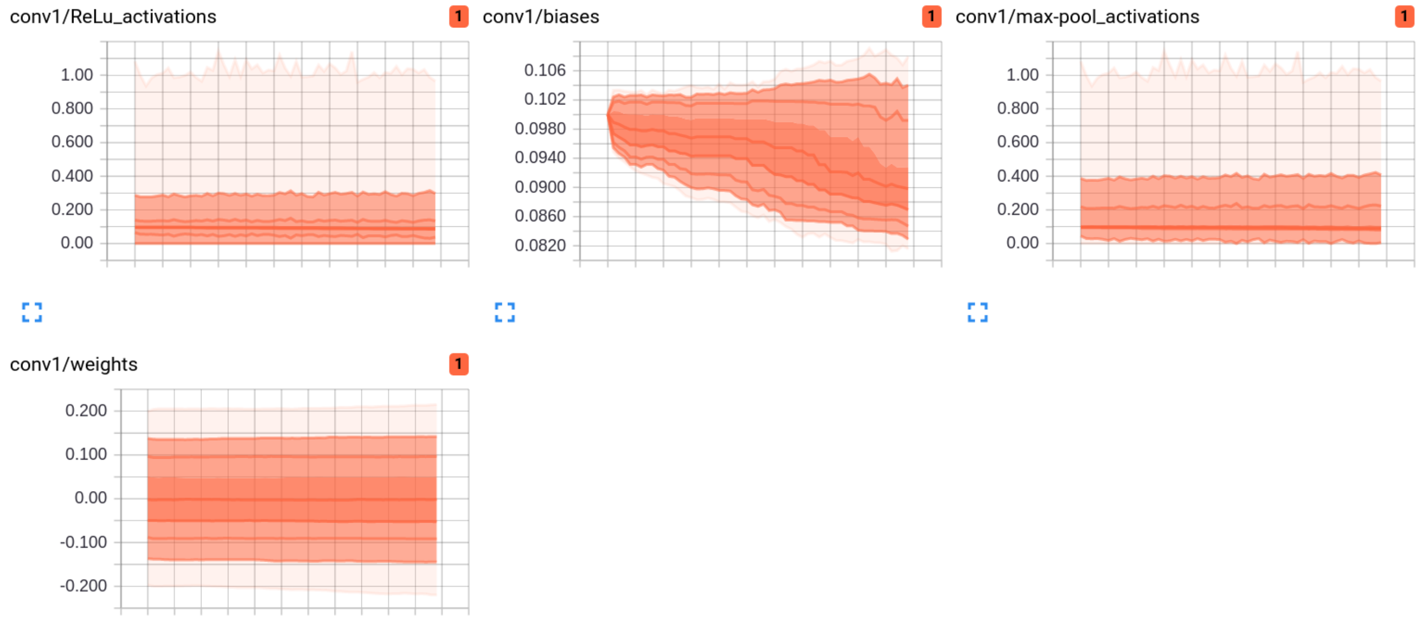

Figure 14: TensorBoard distributions in first convolutional layer, for second DCN training

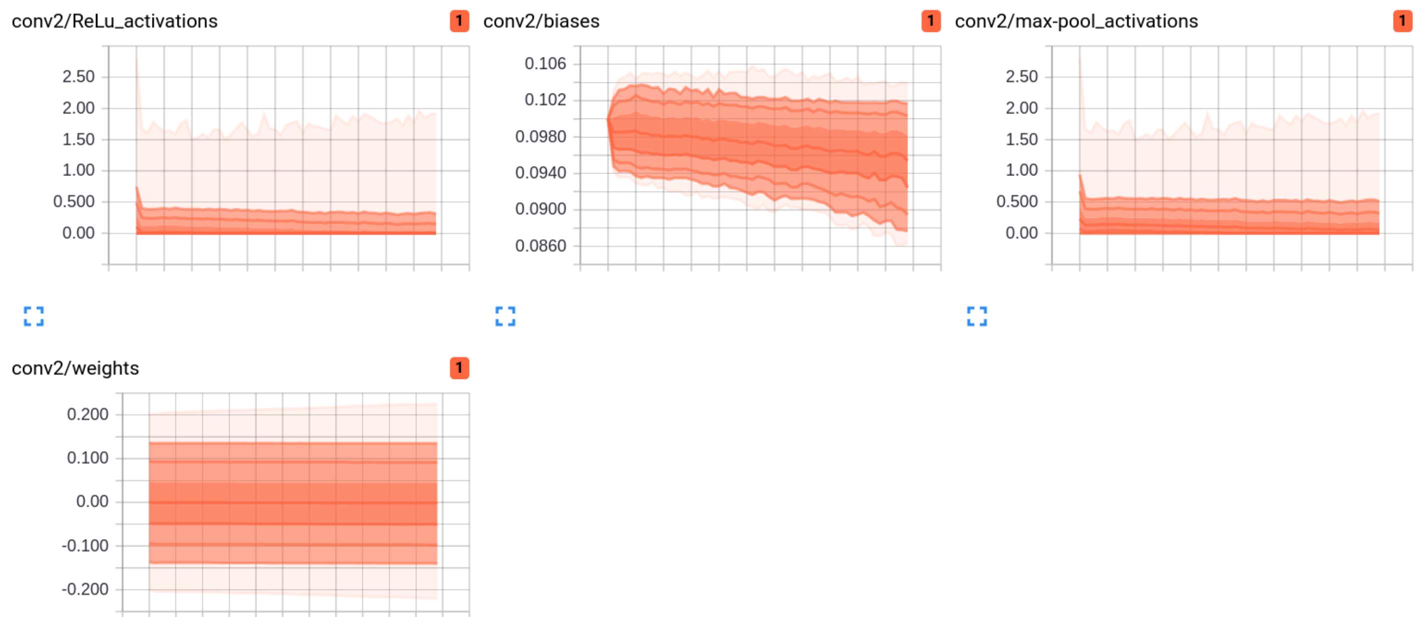

Figure 15: TensorBoard distributions in second convolutional layer, for second DCN training

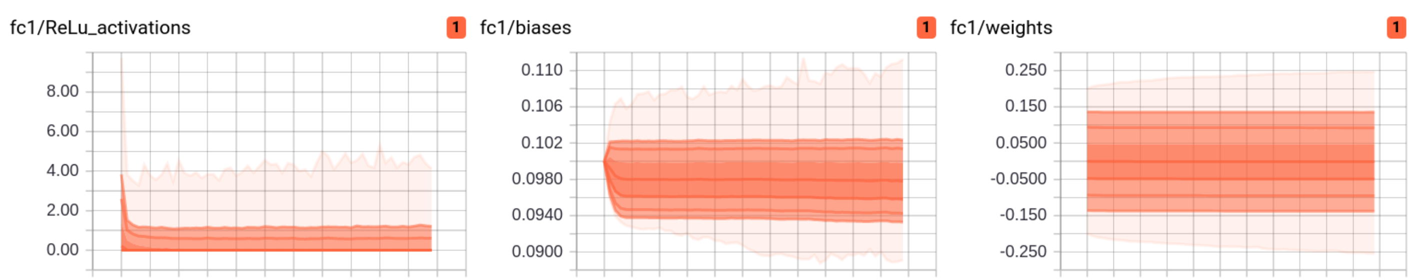

Figure 16: TensorBoard distributions in first fully connected layer, for second DCN training

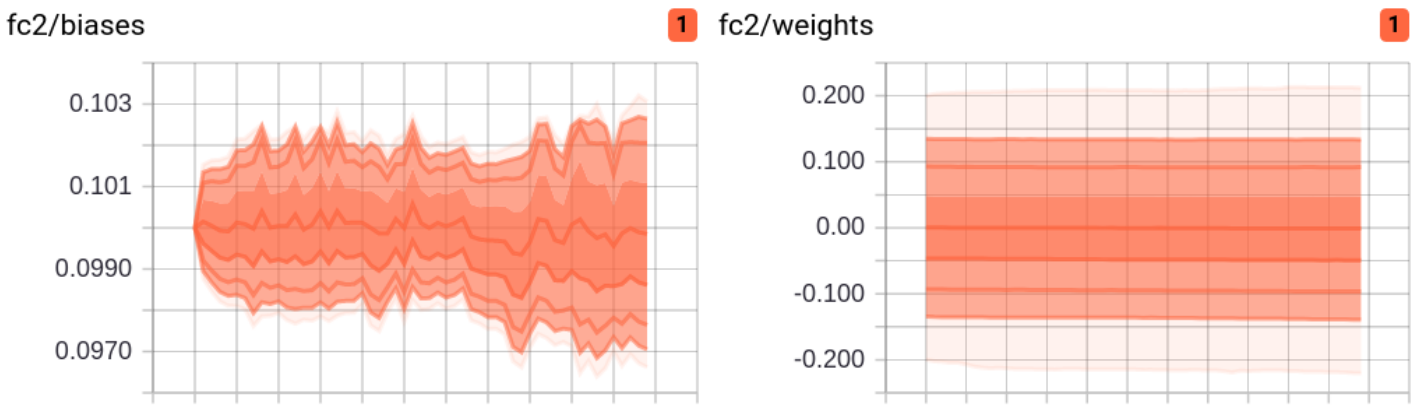

Figure 17: TensorBoard distributions in output layer, for second DCN training

Figure 18: TensorBoard histograms in first convolutional layer, for second DCN training

Figure 19: TensorBoard histograms in second convolutional layer, for second DCN training

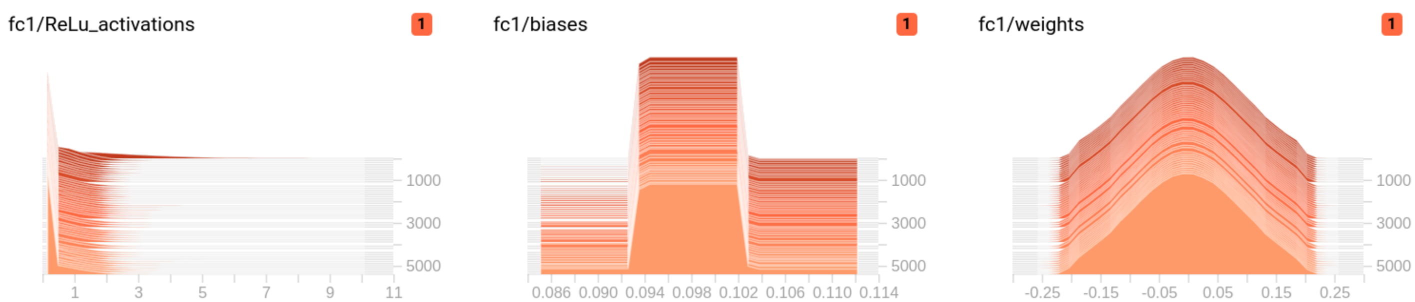

Figure 20: TensorBoard histograms in first fully connected layer, for second DCN training

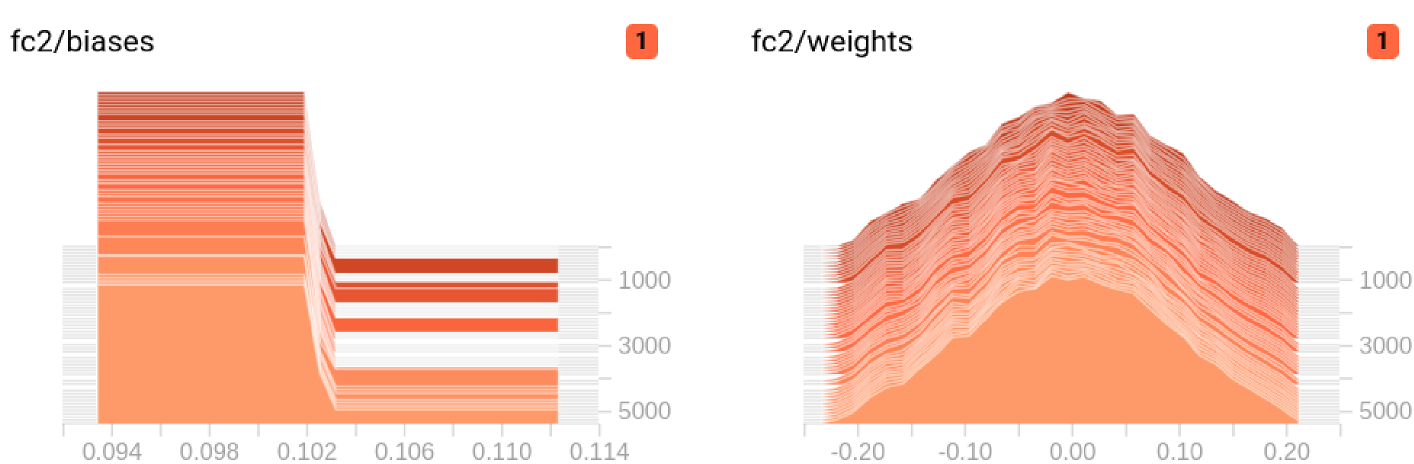

Figure 21: TensorBoard histograms in output layer, for second DCN training