ELEC576 – Fall 2022
Assignment1

Student's Name: Hsuan-You (Shaun) Lin

Student ID: S01435165

Git: https://github.com/PiscesLin/ELEC576_Assignment1.git

1. Backpropagation in a Simple Neural Network:

(a) Dataset: uncomment the "generate and visualize Make_Moons dataset" section and run the code.

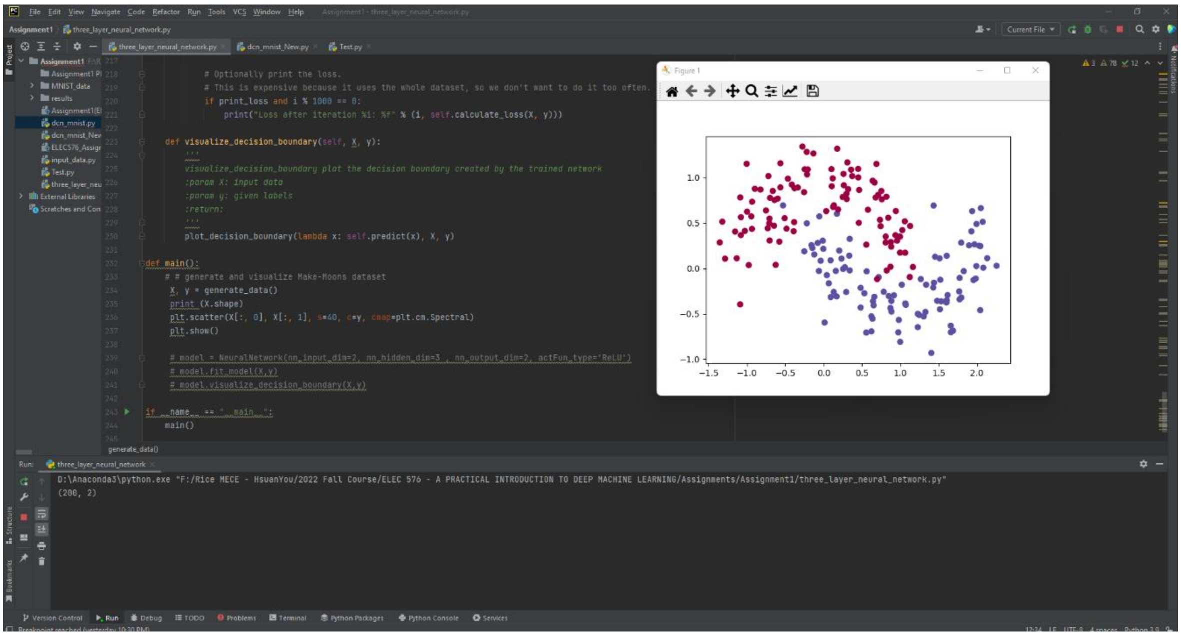


Figure 1. Visualize Make_Moons dataset

(b) Activation Function:

1. Implement function actFun(self, z, type).

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

    # YOU IMPLMENT YOUR actFun HERE
    if type == 'Tanh':
        activations = np.tanh(z)
    elif type == 'Sigmoid':
        activations = 1 / (1 + np.exp(-z))
    elif type == 'ReLU':
        activations = z * (z > 0)
    else:
        print('Invalid activation function type.')

    return activations
```

Figure 2. Function actFun(self, z, type)

2. Derive the derivatives of Tanh, Sigmoid and ReLU.

a. Derivative of Tanh :

$$Tanh(z) = f(z) = X = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$dX = ((e^{z} + e^{-z}) * \mathrm{d}(e^{z} - e^{-z})) - (\frac{(e^{z} - e^{-z}) * d(e^{z} + e^{-z})}{(e^{z} + e^{-z})^2})$$

$$dX = ((e^{z} + e^{-z}) * (e^{z} + e^{-z})) - (\frac{(e^{z} - e^{-z}) * (e^{z} - e^{-z})}{(e^{z} + e^{-z})^2})$$

$$dX = (e^{z} + e^{-z})^2 - \frac{(e^{z} - e^{-z})^2}{(e^{z} + e^{-z})^2}$$

$$dX = 1 - (\frac{e^{z} - e^{-z}}{e^{z} + e^{-z}})^2$$

$$dX = 1 - X^2$$

The result as shown in the *Figure 3*.

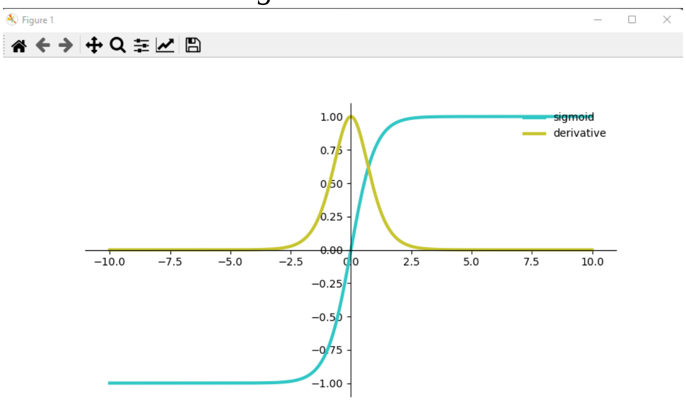


Figure 3. Tanh(z) and Derivative of Tanh(z)

b. Derivative of Sigmoid :

$$\mathrm{Sigmoid(z)} = \mathrm{f(z)} = X = \frac{1}{1 + \exp(-z)}$$

$$dX = \frac{1 + \exp(-z)(d(1)) - d(1 + \exp(-z) * 1}{(1 + \exp(-z))^2}$$

➔ $d(1) = 0,$

➔ $d(1 + \exp(-z)) = \mathrm{d}(1) + \mathrm{d}(\exp(-z)) = -\exp(-z)$

so :

$$dX = \frac{\exp(-z)}{(1 + \exp(-z))^2}$$

$$dX = \frac{1}{1 + \exp(-z)} * (1 - \frac{1}{1 + \exp(-z)})$$

$$dX = \mathrm{f(z)} * (1 - \mathrm{f(z)})$$

The result as shown in the *Figure 4*.

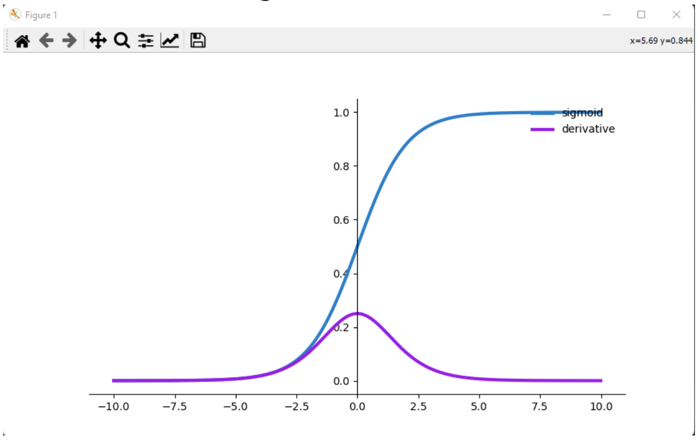


Figure 4. Sigmoid(z) and Derivative of Sigmoid(z)

c. Derivative of ReLU :

f(z)=max(0,x). It gives an output x if x is positive and 0 otherwise.

The result as shown in the *Figure 5*.

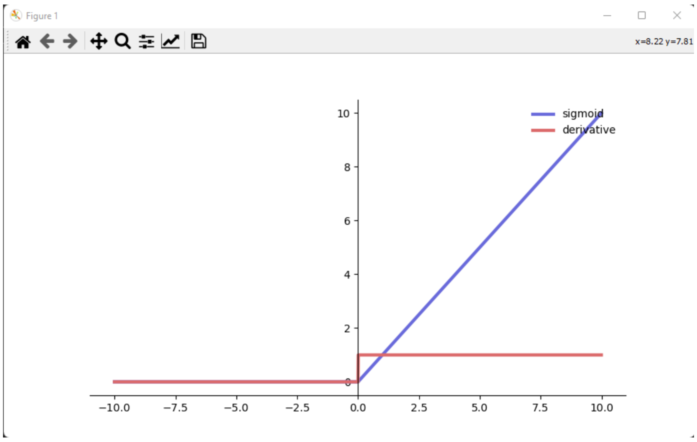


Figure 5. ReLU(z) and Derivative of ReLU(z)

3. Implement function diff_actFun(self, z, type).

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

    # YOU IMPLEMENT YOUR diff_actFun HERE
    if  type == 'Tanh':
        activations = np.tanh(z)
        diff_activations =  1 - (activations * activations)
    elif type == 'Sigmoid':
        activations = 1 / (1 + np.exp(-z))
        diff_activations = activations * (1 - activations)
    elif type == 'ReLU':
        diff_activations = np.array(z)
        diff_activations[diff_activations < 0] = 0
        diff_activations[diff_activations > 0] = 1
    else:
        print('Invalid activation function type.')

    return diff_activations
```

Figure 6. Function diff_actFun(self, z, type)

(c) Build the Neural Network:

1. Implement the function feedforward(self, X, actFun).

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

    # YOU IMPLEMENT YOUR feedforward HERE
    self.z1 = np.dot(X, self.W1) + self.b1 # z1 = W1x+b1
    self.a1 = actFun(self.z1) # a1 = actFun(z1)
    self.z2 = np.dot(self.a1, self.W2) + self.b2 # z2 = W2a1 + b2
    exp_scores = np.exp(self.z2) # a2 = yˆ = softmax(z2)
    self.probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    return None
```

Figure 7. Function feedforward(self, X, actFun)

2. Implement the function calculate_loss(self, X, y).

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

    # YOU IMPLEMENT YOUR CALCULATION OF THE LOSS HERE
    data_loss = np.sum(-np.log(self.probs[range(num_examples), y]))

    # 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
```

Figure 8. Function calculate_loss(self, X, y)

(d) Backward Pass – Backpropagation:

1. Derive the following gradients: $\frac{\partial L}{\partial \mathrm{W2}}$, $\frac{\partial L}{\partial \mathrm{b2}}$, $\frac{\partial L}{\partial \mathrm{W1}}$, $\frac{\partial L}{\partial \mathrm{b1}}$ mathematically.

$$\frac{\partial L}{\partial \mathrm{W2}} = a1^T * d(z2) * \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z2}$$

$$\frac{\partial L}{\partial \mathrm{b2}} = \sum \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z2}$$

$$\frac{\partial L}{\partial \mathrm{W1}} = X^T * d(z1) * \left(d(z2) * \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z2} * W2^T\right)$$

$$\frac{\partial L}{\partial \mathrm{b1}} = \sum d(z1) * \left(d(z2) * \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z2} * W2^T\right)$$

2. Implement the function backprop(self, X, y) .

```python
def backprop(self, X, y):
    '''
    backprop run backpropagation to compute the gradients used to update the parameters in the backward step
    :param X: input data
    :param y: given labels
    :return: dL/dW1, dL/db1, dL/dW2, dL/db2
    '''

    # IMPLEMENT YOUR BACKPROP HERE
    num_examples = len(X)
    delta3 = self.probs
    delta3[range(num_examples), y] -= 1
    dW2 = (self.a1.T).dot(delta3) # dW2 = dL/dW2
    db2 = np.sum(delta3, axis=0, keepdims=True) # db2 = dL/db2
    delta2 = delta3.dot(self.W2.T) * self.diff_actFun(self.z1, self.actFun_type)
    dW1 = np.dot(X.T, delta2) # dW1 = dL/dW1
    db1 = np.sum(delta2, axis=0) # db1 = dL/db1

    return dW1, dW2, db1, db2
```

Figure 9. Function backprop(self, X, y)

(e) Time to Have Fun - Training!

1. Train the network using different activation functions (Tanh, Sigmoid and ReLU).

a. Train the network using Tanh activation functions.

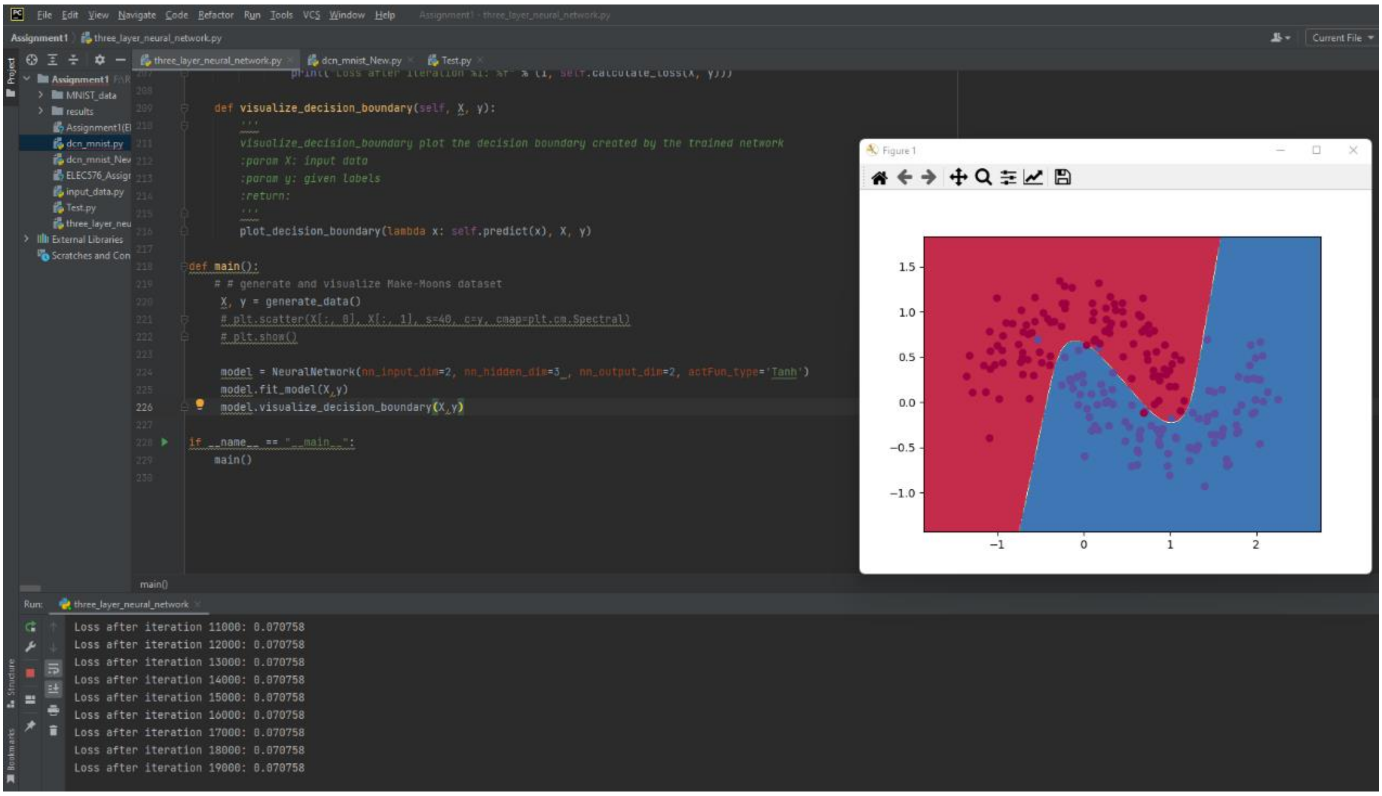


Figure 10. Tanh activation functions training result

b. Train the network using Sigmoid activation functions.

Figure 11. Sigmoid activation functions training result

c. Train the network using ReLU activation functions.

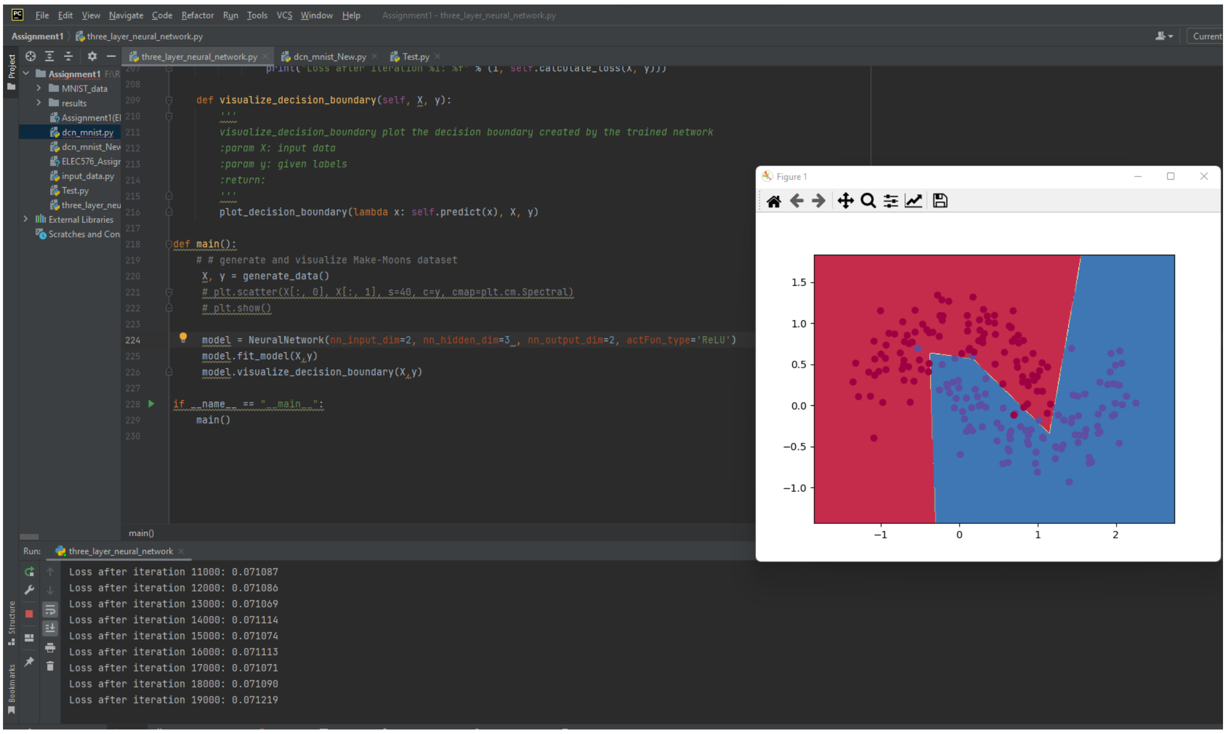


Figure 12. ReLU activation functions training result

My Observation:

First, from the above training results, we can see that the loss function result of the Tanh activation function is 0.070758, the Sigmoid activation function is 0.78155, and the ReLU activation function is 0.071219. Compared with the Sigmoid function, the Tanh function tends to be more better, mainly because the Sigmoid function is sensitive to changes in the function value when the input is between [-1, 1]. Once it approaches or exceeds the interval, it loses its sensitivity and is in a saturated state, which affects the accuracy of the neural network prediction.

Secondly, from the three figures above, the tangent line of Tanh and Sigmoid are smoother than that of ReLU, this is because of its linear and unsaturated. Sigmoid and Tanh activation functions need to calculate exponents and they have high complexity, while ReLU can converge quickly.

2. Increase the number of hidden units and retrain the network using Tanh as the activation function.

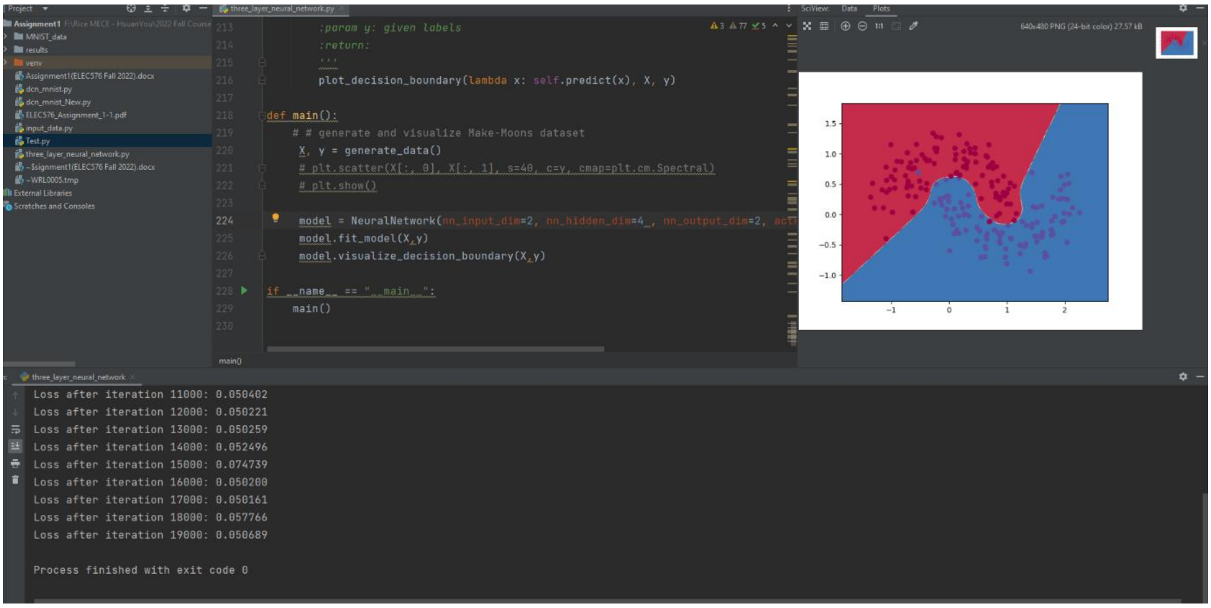


Figure 13. hidden units = 4 and retrain the network using Tanh activation functions

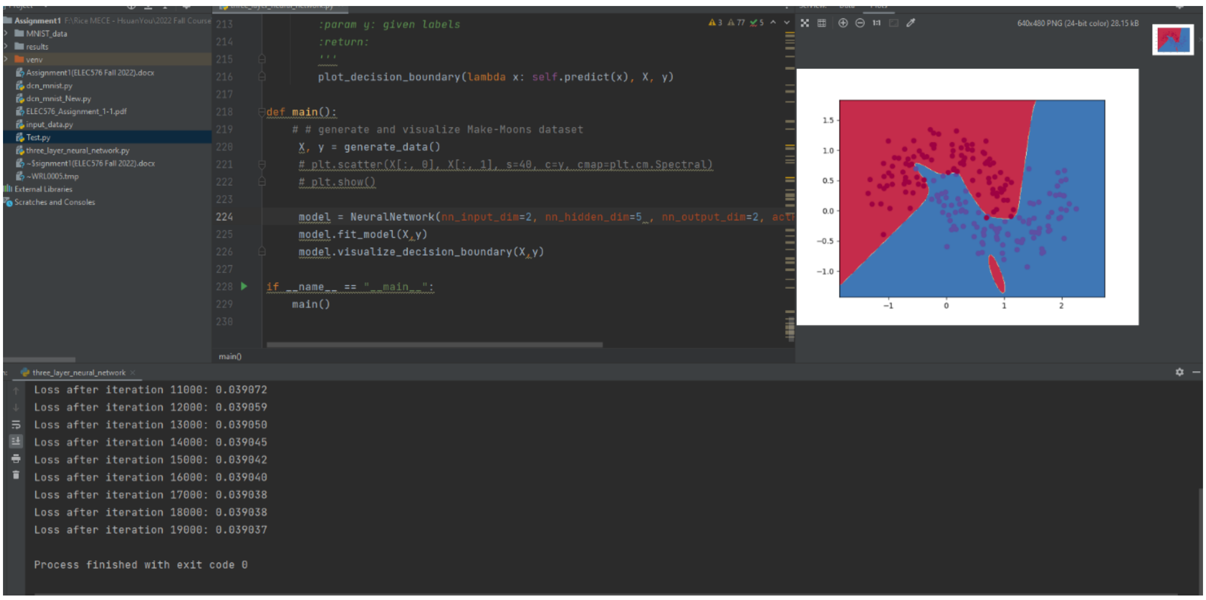


Figure 14. hidden units = 5 and retrain the network using Tanh activation functions

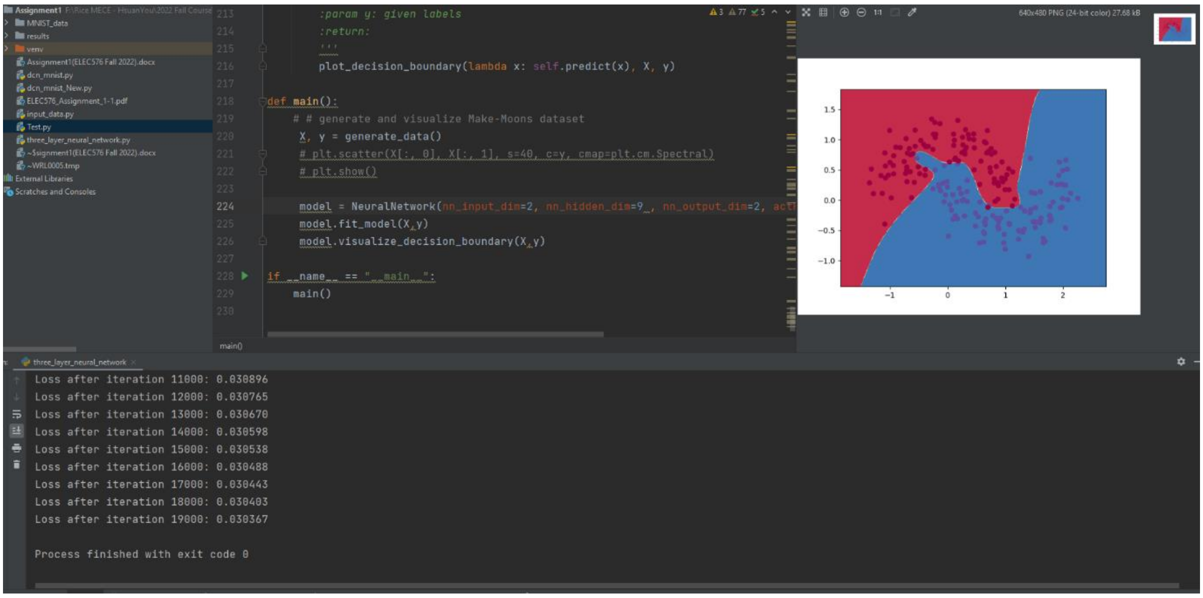


Figure 15. hidden units = 9 and retrain the network using Tanh activation functions

My Observation:

From the above training results, I increased the number of hidden units to 4, 5, 9 and used the Tanh activation function to retrain the network, we can see the loss function results are much better than the original results of using 3 hidden units. But when the hidden layer is 5, overfitting occurs, it hasn't learnt the trend and thus it is not able to generalize to new data.

(f) Even More Fun - Training a Deeper Network!!!

1. Create a new class, e.g DeepNeuralNetwork.

```python
class DeepNeuralNetwork(object):
    """
    This class builds and trains a neural network
    """
    def __init__(self, nn_input_dim, nn_hidden_dim, nn_output_dim, num_layers, actFun_type='Tanh', reg_lambda=0.01, seed=0):
        '''
        :param nn_input_dim: input dimension
        :param nn_hidden_dim: the number of hidden units
        :param nn_output_dim: output dimension
        :param num_layers: input layer
        :param actFun_type: type of activation function. 3 options: 'tanh', 'sigmoid', 'relu'
        :param reg_lambda: regularization coefficient
        :param seed: random seed
        '''
        self.nn_input_dim = nn_input_dim
        self.nn_hidden_dim = nn_hidden_dim
        self.nn_output_dim = nn_output_dim
        self.num_layers = num_layers
        self.actFun_type = actFun_type
        self.reg_lambda = reg_lambda

        # Create "n" Layers:
        self.LayerList = []
        for count in range(num_layers - 1):
            if count == 0:
                x = Layer(nn_input_dim=self.nn_input_dim, nn_output_dim=self.nn_hidden_dim, last_layer=0,
                          actFun_type=self.actFun_type, seed=count)
            elif (count < num_layers - 2):
                x = Layer(nn_input_dim=self.nn_hidden_dim, nn_output_dim=self.nn_hidden_dim, last_layer=0,
                          actFun_type=self.actFun_type, seed=count)
            else:
                x = Layer(nn_input_dim=self.nn_hidden_dim, nn_output_dim=self.nn_output_dim, last_layer=1,
                          actFun_type=self.actFun_type, seed=count)
            self.LayerList.append(x)
```

Figure 16. class DeepNeuralNetwork() in my n_layer_neural_network.py program

2. In DeepNeuralNetwork, change function feedforward, backprop, calculate_loss and fit_model.

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

    # YOU IMPLEMENT YOUR feedforward HERE
    for count in range(self.num_layers - 1):
        if count == 0:
            self.LayerList[count].feedforward(X, lambda x: self.actFun(x, type=self.actFun_type))
        elif (count < self.num_layers - 2):
            self.LayerList[count].feedforward(self.LayerList[count - 1].a,
                                              lambda x: self.actFun(x, type=self.actFun_type))
        else:
            self.LayerList[count].feedforward(self.LayerList[count - 1].a,
                                              lambda x: self.actFun(x, type=self.actFun_type))
            self.probs = self.LayerList[count].probs

    return None
```

Figure 17. Function feedforward(self, X, actFun) in n_layer_neural_network.py

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

    # IMPLEMENT YOUR BACKPROP HERE
    dW = []
    db = []
    delta = []
    for count in range(self.num_layers - 1):
        dW.append([])
        db.append([])
        delta.append([])

    num_examples = len(X)
    delta[self.num_layers - 2] = self.probs
    delta[self.num_layers - 2][range(num_examples), y] -= 1

    for count in range(self.num_layers - 2, -1, -1):
        if count == self.num_layers - 2:
            dW[count] = (self.LayerList[count - 1].a.T).dot(delta[count])
            db[count] = np.sum(delta[count], axis=0, keepdims=True)

        elif count == 0:
            delta[count] = delta[count + 1].dot(self.LayerList[count + 1].W.T) * self.diff_actFun(
                self.LayerList[count].z, self.actFun_type)
            dW[count] = (X.T).dot(delta[count])
            db[count] = np.sum(delta[count], axis=0, keepdims=True)
        else:
            delta[count] = delta[count + 1].dot(self.LayerList[count + 1].W.T) * self.diff_actFun(
                self.LayerList[count].z, self.actFun_type)
            dW[count] = (self.LayerList[count - 1].a.T).dot(delta[count])
            db[count] = np.sum(delta[count], axis=0, keepdims=True)

    return dW, db
```

Figure 18. Function backprop(self, X, y) in n_layer_neural_network.py

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

    # YOU IMPLEMENT YOUR CALCULATION OF THE LOSS HERE
    data_loss = np.sum(-np.log(self.probs[range(num_examples), y]))

    # Add regulatization term to loss
    tempsum = 0
    for count in range(self.num_layers - 1):
        tempsum += np.sum(np.square(self.LayerList[count].W))
    data_loss += self.reg_lambda / 2 * tempsum

    return (1. / num_examples) * data_loss
```

Figure 19. Function calculate_loss(self, X, y) in n_layer_neural_network.py

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

        # Add regularization terms (b1 and b2 don't have regularization terms)
        for count in range(self.num_layers - 1):
            dW[count] += self.reg_lambda * self.LayerList[count].W
            self.LayerList[count].W += -epsilon * dW[count]
            self.LayerList[count].b += -epsilon * db[count]

        # 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):
```

Figure 20. Function fit_model(self, X, y) in n_layer_neural_network.py

3~5. Create a new class, e.g. Layer(), that implements the feedforward and backprop steps for a single layer in the network.

```python
class Layer(object):
    """
    This class builds for a single layer in the neural network
    """
    def __init__(self, nn_input_dim, nn_output_dim, last_layer = 0, actFun_type='tanh', reg_lambda=0.01, seed=0):
        '''
        :param nn_input_dim: input dimension
        :param nn_output_dim: output dimension
        :param actFun_type: type of activation function. 3 options: 'tanh', 'sigmoid', 'relu'
        :param reg_lambda: regularization coefficient
        :param seed: random seed
        :param last_layer: input last layer of the network
        '''
        self.nn_input_dim = nn_input_dim
        self.nn_output_dim = nn_output_dim
        self.actFun_type = actFun_type
        self.reg_lambda = reg_lambda
        self.last_layer = last_layer

        # initialize the weights and biases in the network
        np.random.seed(seed)
        self.W = np.random.randn(self.nn_input_dim, self.nn_output_dim) / np.sqrt(self.nn_input_dim)
        self.b = np.zeros((1, self.nn_output_dim))

    def feedforward(self, X, actFun):
        '''
        feedforward builds the feedforward steps for a single layer in the network
        :param X: input data
        :param actFun: activation function
        :return:
        '''

        # YOU IMPLEMENT YOUR feedforward HERE
        self.z = np.dot(X, self.W) + self.b

        # Intermediate Layer:
        if self.last_layer == 0:
            self.a = actFun(self.z)
        # Last Layer:
        else:
            exp_scores = np.exp(self.z)
            self.probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

        return None
```

Figure 21. class Layer() in my n_layer_neural_network.py program

6. Notice that we have L2 weight regularizations in the final loss function in addition to the cross entropy. Make sure you add those regularization terms in DeepNeuralNetwork.calculate_loss and their derivatives in DeepNeuralNetwork.fit_model.

```python
# Add regulatization term to loss
tempsum = 0
for count in range(self.num_layers - 1):
    tempsum += np.sum(np.square(self.LayerList[count].W))
```

Figure 24. Regularizations terms in DeepNeuralNetwork.calculate_loss

```python
# Add regularization terms (b1 and b2 don't have regularization terms)
for count in range(self.num_layers - 1):
    dW[count] += self.reg_lambda * self.LayerList[count].W
    self.LayerList[count].W += -epsilon * dW[count]
    self.LayerList[count].b += -epsilon * db[count]
```

Figure 25. Regularizations terms in DeepNeuralNetwork.fit_model

7. Train your network on the Make_Moons dataset using different number of layers, different layer sizes, different activation functions and, in general, different network configurations.

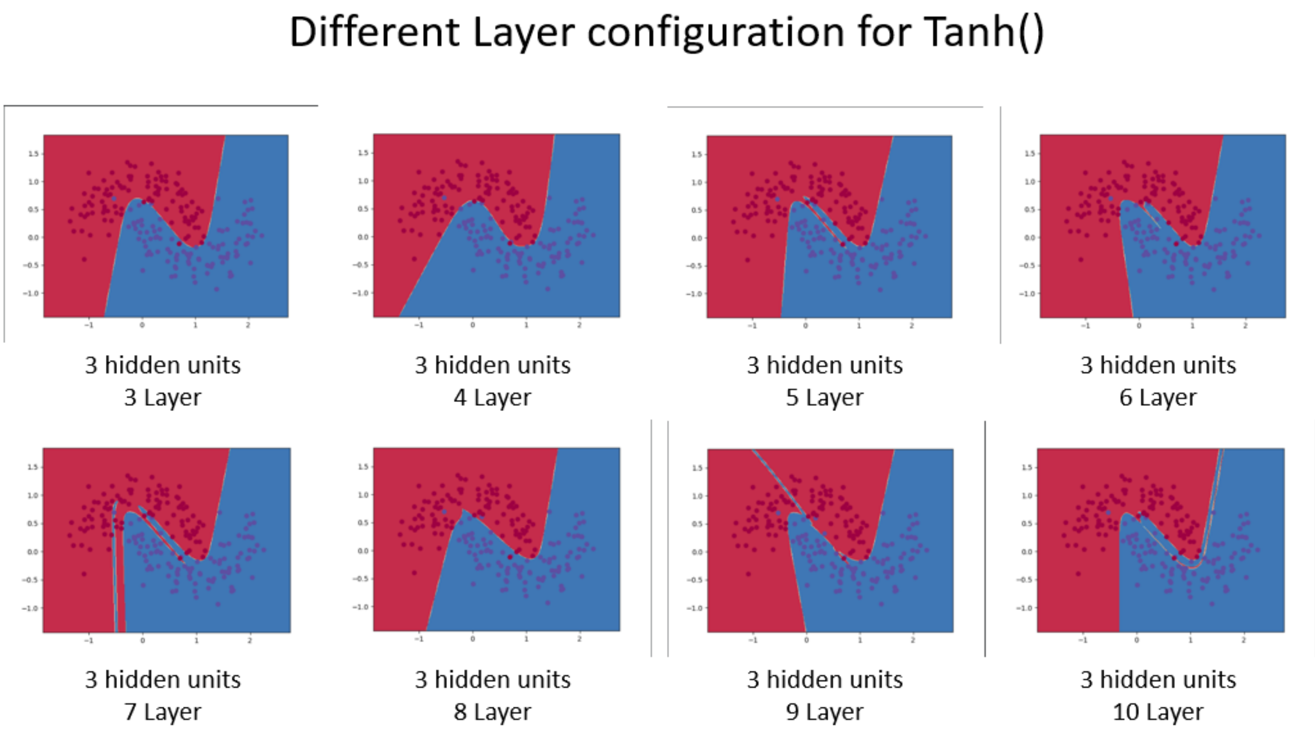


Figure 26. Using Tanh activation functions and different layer configuration

Table 1. Training loss function results using Tanh activation functions and different layer configuration in Make_Moons dataset

	Tanh (3 Hidden Units; 3 Layer)	Tanh (3 Hidden Units; 4 Layer)	Tanh (3 Hidden Units; 5 Layer)	Tanh (3 Hidden Unit; 6 Layer)
Loss result	0.068160	0.059109	0.027966	0.027341
	Tanh (3 Hidden Units; 7 Layer)	Tanh (3 Hidden Units; 8 Layer)	Tanh (3 Hidden Units; 9 Layer)	Tanh (3 Hidden Unit; 10 Layer)
Loss result	0.004708	0.052291	0.071217	0.027665

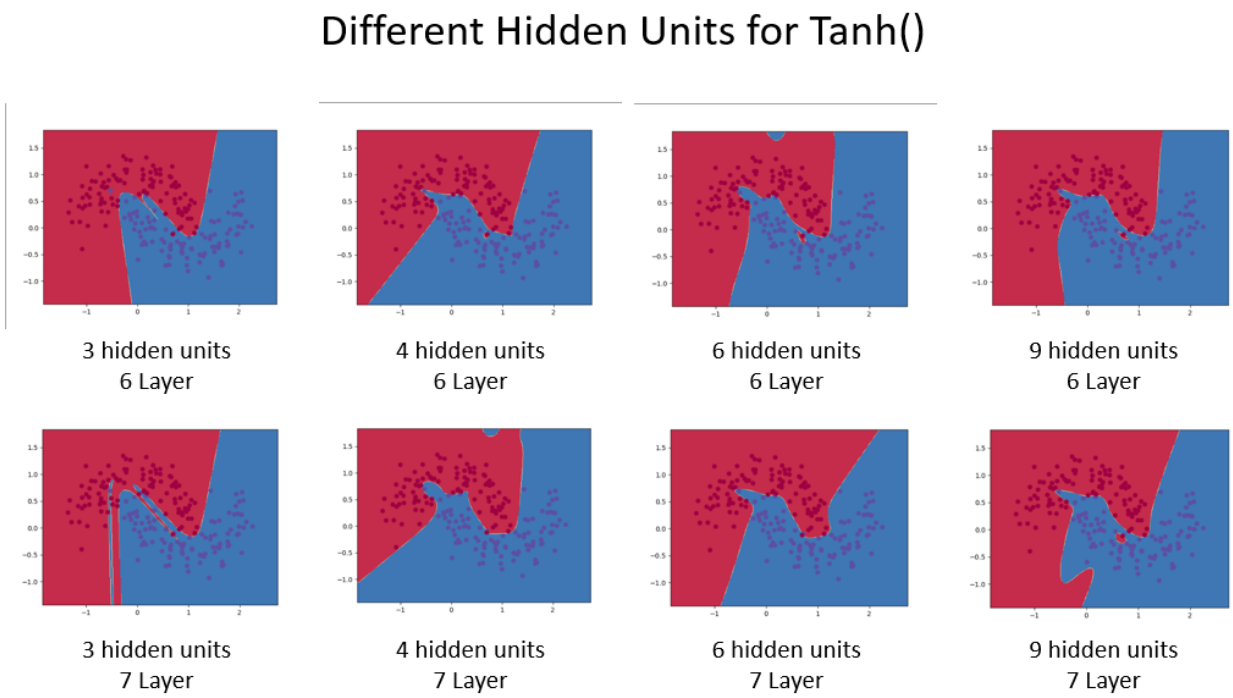


Figure 27. Using Tanh activation functions and different hidden units

Table 2. . Training loss function results using Tanh activation functions and different hidden units in Make_Moons dataset

	Tanh (3 Hidden Units; 6 Layer)	Tanh (4 Hidden Units; 6 Layer)	Tanh (6 Hidden Units; 6 Layer)	Tanh (9 Hidden Unit; 6 Layer)
Loss result	0.027341	0.010988	0.005553	0.004899
	Tanh (3 Hidden Units; 7 Layer)	Tanh (4 Hidden Units; 7 Layer)	Tanh (6 Hidden Units; 7 Layer)	Tanh (9 Hidden Unit; 7 Layer)
Loss result	0.004708	0.011440	0.010534	0.003838

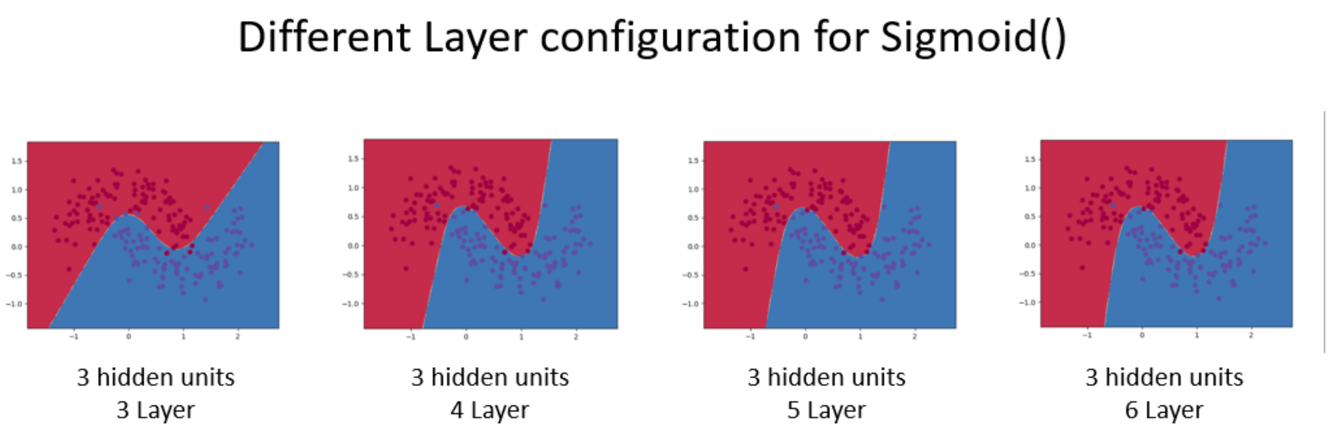


Figure 28. Using Sigmoid activation functions and different layer configuration

Table 3. Training loss function results using Sigmoid activation functions and different layer configuration in Make_Moons dataset

	Sigmoid (3 Hidden Units; 3 Layer)	Sigmoid (3 Hidden Units; 4 Layer)	Sigmoid (3 Hidden Units; 5 Layer)	Sigmoid (3 Hidden Unit; 6 Layer)
Loss result	0.201572	0.072161	0.070458	0.071994

Figure 29. Using Sigmoid activation functions and different hidden units

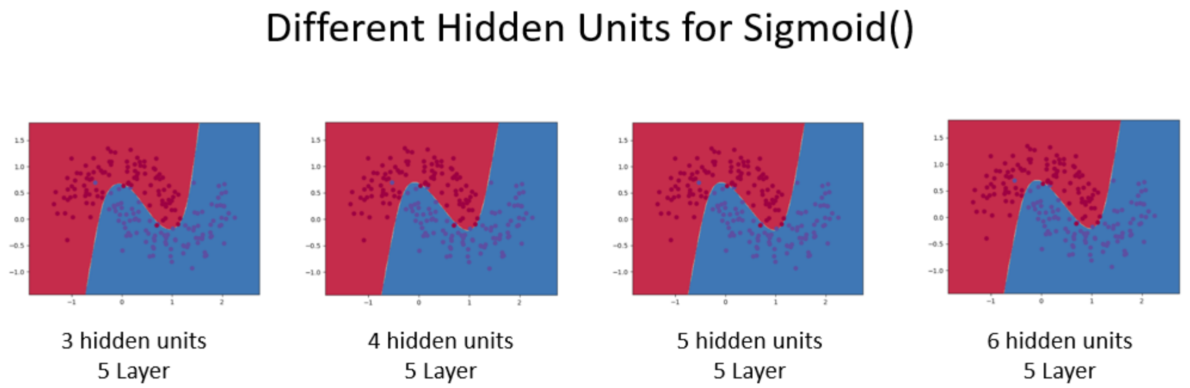


Table 4. Training loss function results using Sigmoid activation functions and different hidden units in Make_Moons dataset

	Sigmoid (3 Hidden Units; 5 Layer)	Sigmoid (4 Hidden Units; 5 Layer)	Sigmoid (5 Hidden Units; 5 Layer)	Sigmoid (6 Hidden Unit; 5 Layer)
Loss result	0.070458	0.077270	0.076870	0.076931

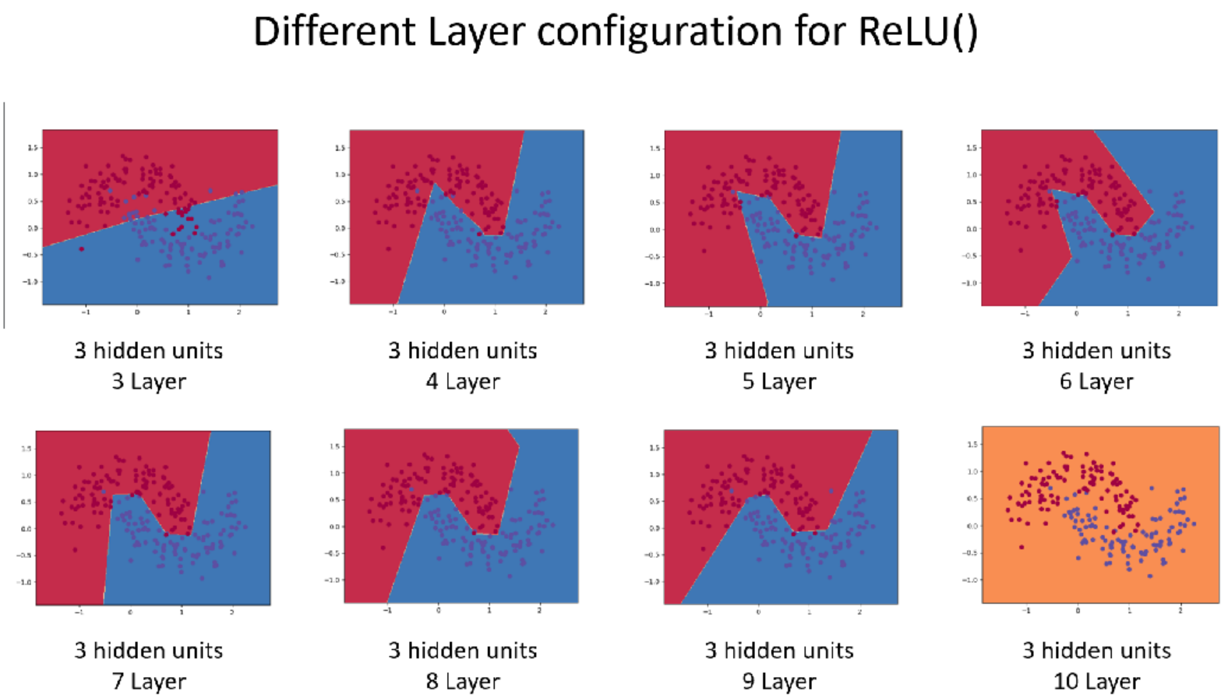


Figure 30. Using ReLU activation functions and different layer configuration

Table 5. Training loss function results using ReLU activation functions and different layer configuration in Make_Moons dataset

	ReLU (3 Hidden Units; 3 Layer)	ReLU (3 Hidden Units; 4 Layer)	ReLU (3 Hidden Units; 5 Layer)	ReLU (3 Hidden Unit; 6 Layer)
Loss result	0.299422	0.076118	0.048086	0.048109
	ReLU (3 Hidden Units; 7 Layer)	ReLU (3 Hidden Units; 8 Layer)	ReLU (3 Hidden Units; 9 Layer)	ReLU (3 Hidden Unit; 10 Layer)
Loss result	0.046176	0.042916	0.100279	0.693788

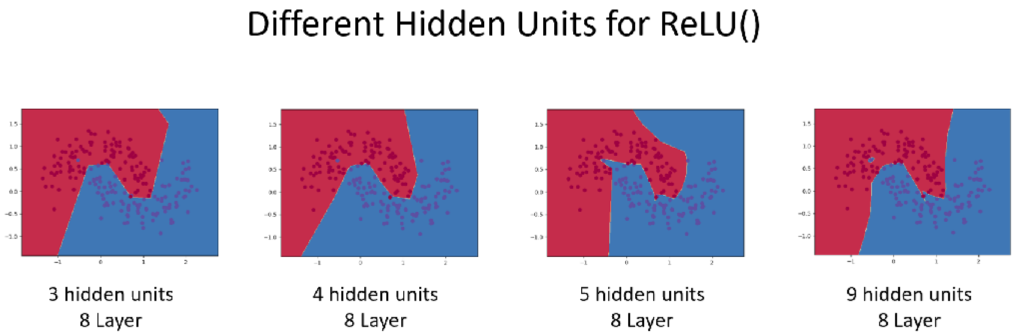


Figure 31. Using ReLU activation functions and different hidden units

Table 6. Training loss function results using ReLU activation functions and different hidden units in Make_Moons dataset

	ReLU (3 Hidden Units; 8 Layer)	ReLU (6 Hidden Units; 8 Layer)	ReLU (8 Hidden Units; 8 Layer)	ReLU (9 Hidden Unit; 8 Layer)
Loss result	0.042916	0.027186	0.003903	0.002909

My Observation:

From the above training results, it can be found that the training results of ReLU activation functions are the best, I made a training result table for each activation function individually, I tried the different layer configuration first, then extract the best score from the table, then changed the hidden units in orderto get the best loss function result. As you can see from the Table 6, using 9 hidden units and 8 layer for ReLU activation function, we got the only 0.002909 loss function result, even better than Tanh activation function, which in Table 2 using 9 hidden unit and 7 layer, it's only got the 0.003838, I have also done further scrutinize for higher parameter, but that's the best I acquired.

Compare to the best results from Tanh and ReLU activation function, I also tried a few different network configurations for Sigmoid activation function. However, the training results didn't exceed my expectations, even if I changed hidden units or layers number, the loss function results did not change significantly.

8. Train your network on another dataset different from Make_Moons.

a. Train my network on Make_Circles dataset.

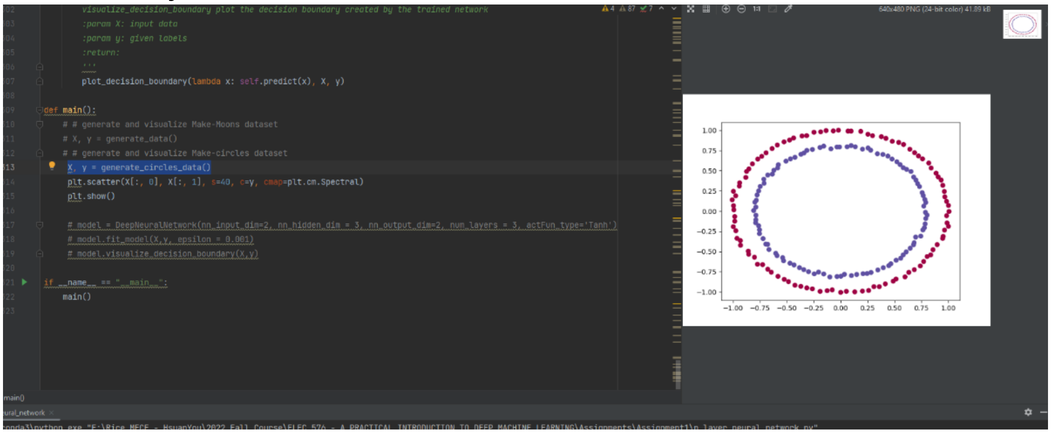


Figure 32. Visualize Make_Circles dataset

b. Using different network configurations to train my network on Make_Circles dataset.

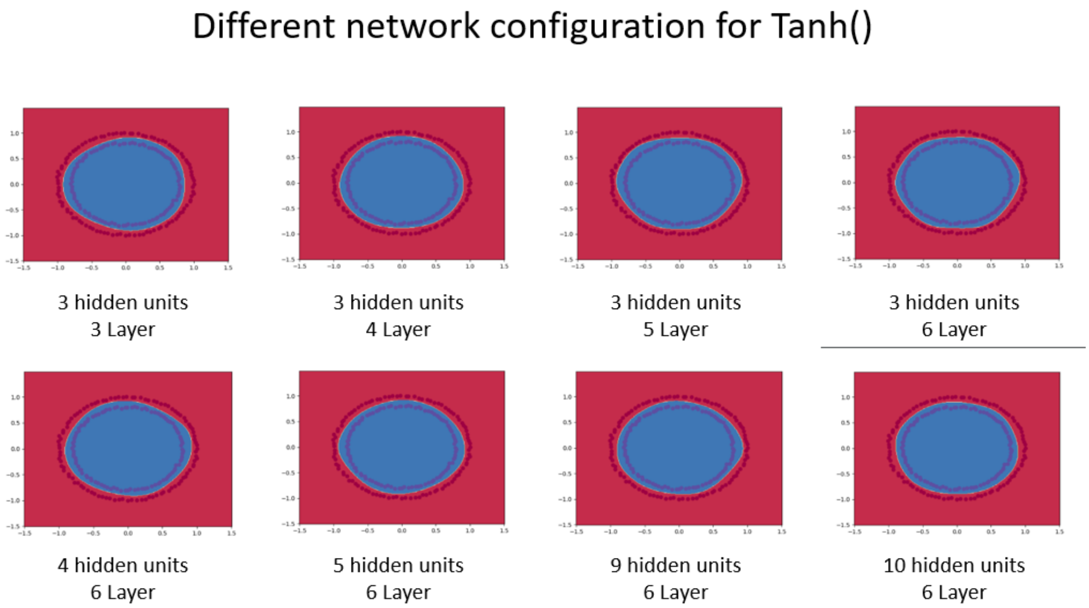


Figure 33. Using Tanh activation function and different network configurations on Make_Circles dataset

Table 7. Training loss function results using Tanh activation functions and different network configuration in Make_Circles dataset

	Tanh (3 Hidden Units; 3 Layer)	Tanh (3 Hidden Units; 4 Layer)	Tanh (3 Hidden Units; 5 Layer)	Tanh (3 Hidden Unit; 6 Layer)
Loss result	0.069340	0.005347	0.003336	0.002961
	Tanh (4 Hidden Units; 6 Layer)	Tanh (5 Hidden Units; 6 Layer)	Tanh (10 Hidden Units; 6 Layer)	Tanh (9 Hidden Unit; 6 Layer)
Loss result	0.002252	0.002189	0.002398	0.002253

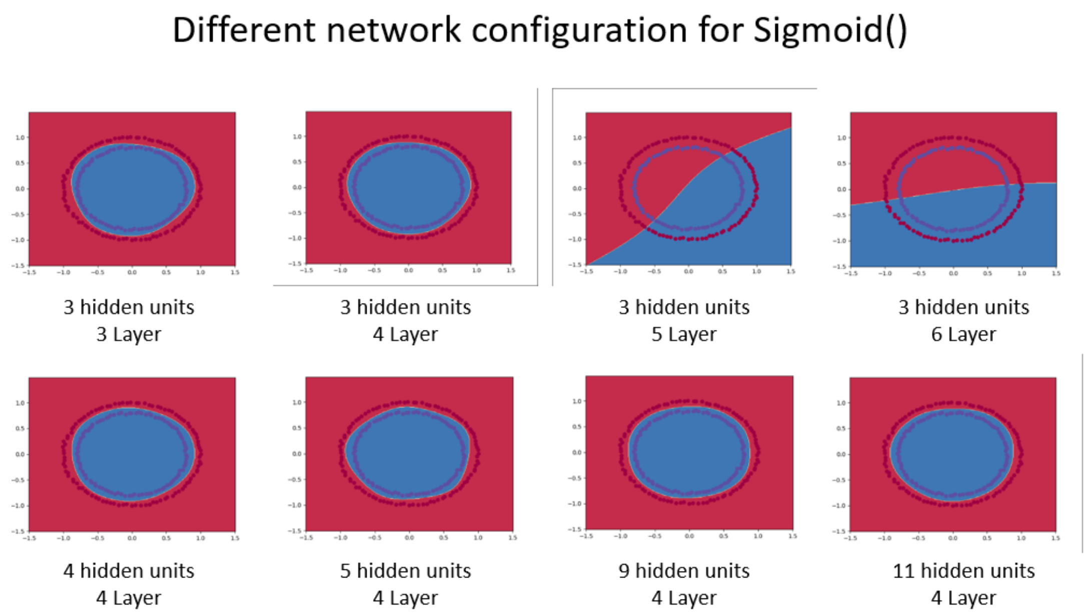


Figure 34. Using Sigmoid activation function and different network configurations on Make_Circles dataset

Table 8. Training loss function results using Sigmoid activation functions and different network configuration in Make_Circles dataset

	Sigmoid (3 Hidden Units; 3 Layer)	Sigmoid (3 Hidden Units; 4 Layer)	Sigmoid (3 Hidden Units; 5 Layer)	Sigmoid (3 Hidden Unit; 6 Layer)
Loss result	0.163729	0.028441	0.693429	0.693492
	Sigmoid (4 Hidden Units; 4 Layer)	Sigmoid (5 Hidden Units; 4 Layer)	Sigmoid (9 Hidden Units; 4 Layer)	Sigmoid (11 Hidden Unit; 4 Layer)
Loss result	0.027062	0.026341	0.019822	0.018900

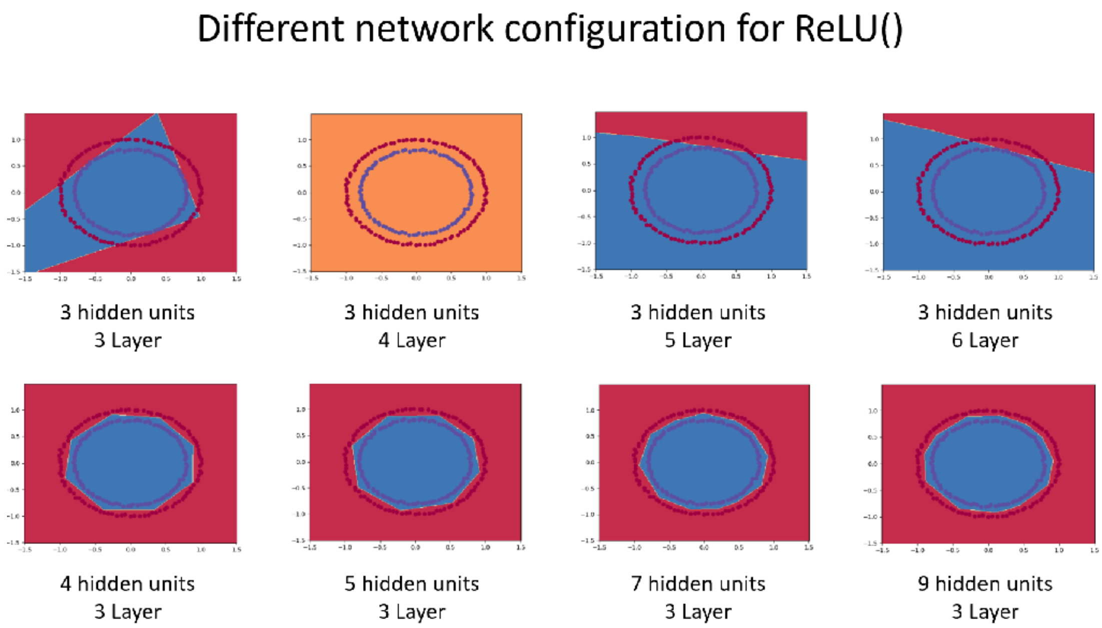


Figure 35. Using ReLU activation function and different network configurations on Make_Circles dataset

Table 9. Training loss function results using ReLU activation functions and different network configuration in Make_Circles dataset

	ReLU (3 Hidden Units; 3 Layer)	ReLU (3 Hidden Units; 4 Layer)	ReLU (3 Hidden Units; 5 Layer)	ReLU (3 Hidden Unit; 6 Layer)
Loss result	0.429663	0.694220	0.623867	0.623461
	ReLU (4 Hidden Units; 3 Layer)	ReLU (5 Hidden Units; 3 Layer)	ReLU (7 Hidden Units; 3 Layer)	ReLU (9 Hidden Unit; 3 Layer)
Loss result	0.015421	0.013605	0.011218	0.010969

My Observation:

In this part, I choose to use the Make_Circles dataset, and the way to make the table is the same as the analysis of the Make_Moons above. Adjust the different layers first, then take the layer with the best score to further test different hidden units. The biggest difference in the part is that the Tanh activation function achieves the best loss function result of 0.002189 when the number of layers is 6 and used 5 hidden units, the decision boundary of Tanh activation function is obviously the best without any overfitting. Also you can noticed that, after ReLU activation function with more than three layers, it can't be classified accurately, which means that ReLU activation function is not suitable for in the circles dataset.

2. Training a Simple Deep Convolutional Network on MNIST:

(a) Build and Train a 4-layer DCN:

1. Read the tutorial Deep MNIST for Expert to learn how to use Tensorflow.
2. Complete functions weight_variable(shape), bias_variable(shape), conv2d(x, W), max_pool_2x2(x) in dcn_mnist.py.

```
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 channels of the filters
    Cout: the number of filters
    :return: a tensor variable for weights with initial values
    '''

    # IMPLEMENT YOUR WEIGHT_VARIABLE HERE
    W = tf.Variable(tf.truncated_normal(shape, stddev=0.1))

    return W
```

Figure 36. Function weight_variable(shape)

```
def bias_variable(shape):
    '''
    Initialize biases
    :param shape: shape of biases, e.g. [Cout] where
    Cout: the number of filters
    :return: a tensor variable for biases with initial values
    '''

    # IMPLEMENT YOUR BIAS_VARIABLE HERE
    b = tf.Variable(tf.constant(0.1, shape=shape))

    return b
```

Figure 37. Function bias_variable(shape)

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

    # IMPLEMENT YOUR CONV2D HERE
    h_conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

    return h_conv
```

Figure 38. Function conv2d(x, W)

```python
def max_pool_2x2(x):
    ''']
    Perform non-overlapping 2-D maxpooling on 2x2 regions in the input data
    :param x: input data
    :return: the results of maxpooling (max-marginalized + downsampling)
    '''

    # IMPLEMENT YOUR MAX_POOL_2X2 HERE
    h_max = tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

    return h_max
```

Figure 39. Function max_pool_2x2(x)

3. Build your network: complete " FILL IN THE CODE BELOW TO BUILD YOUR NETWORK"

```
# FILL IN THE CODE BELOW TO BUILD YOUR NETWORK

# placeholders for input data and input labeles
x = tf.placeholder(tf.float32, [None, 784], name='x')
y_ = tf.placeholder(tf.float32, [None, 10],  name='y_')

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

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

# 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.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2, name='y')
```

Figure 40. dcn_mnist.py build network

4. Set up Training: complete " FILL IN THE FOLLOWING CODE TO SET UP THE TRAINING".

```
# FILL IN THE FOLLOWING CODE TO SET UP THE TRAINING

# setup training
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv), reduction_indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')
```

Figure 41. dcn_mnist.py set up the training

5. Run Training:

```
step 4000, training accuracy 0.98
step 4100, training accuracy 1
step 4200, training accuracy 1
step 4300, training accuracy 0.98
step 4400, training accuracy 0.98
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 0.98
step 5000, training accuracy 0.96
step 5100, training accuracy 0.98
step 5200, training accuracy 1
step 5300, training accuracy 1
step 5400, training accuracy 0.98
test accuracy 0.9868
The training takes 215.799195 second to finish
```

Figure 42. Training result

6. Visualize Training:

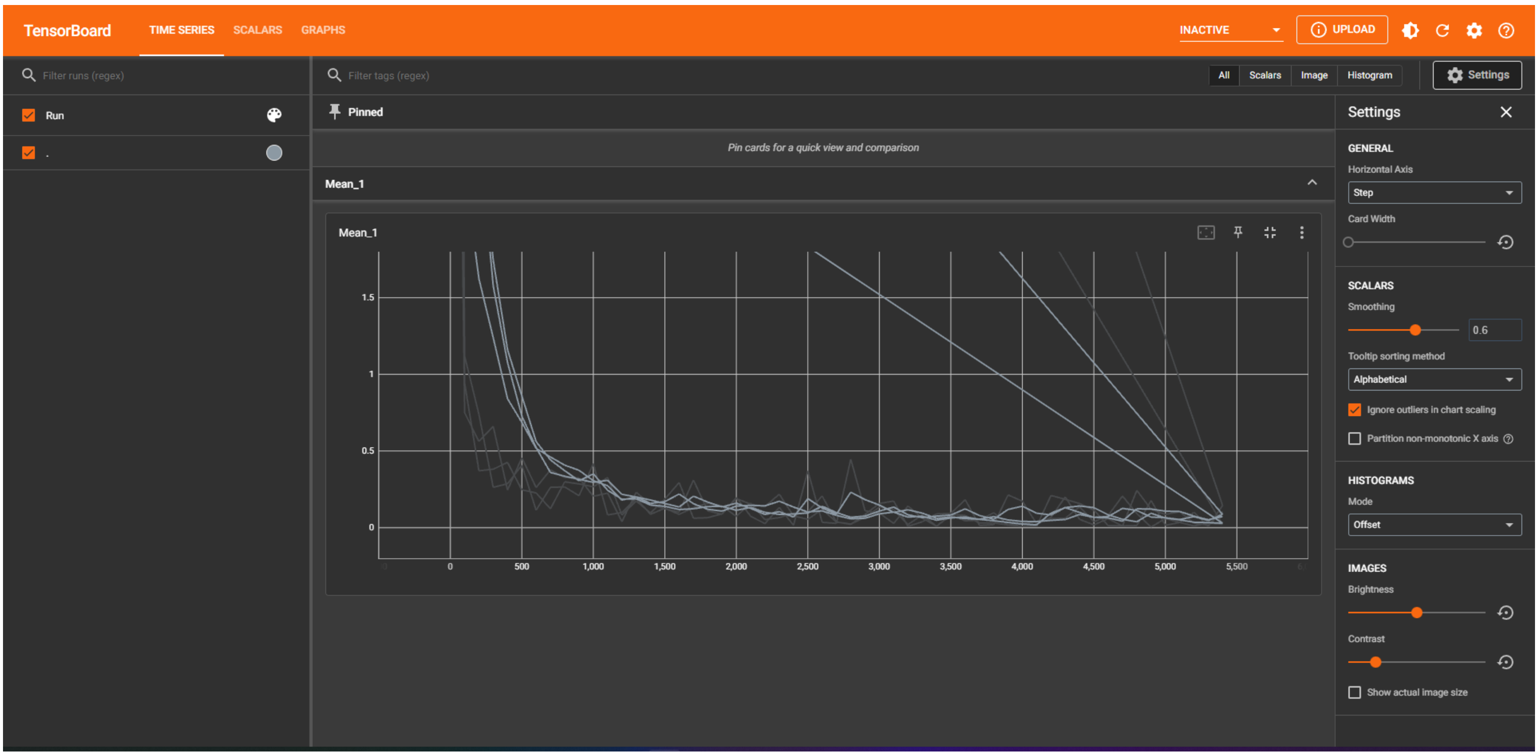


Figure 43. Training result visualize in TensorBoard

(b) More on Visualizing Your Training:

1. Run the training again and visualize the monitored terms in TensorBoard.

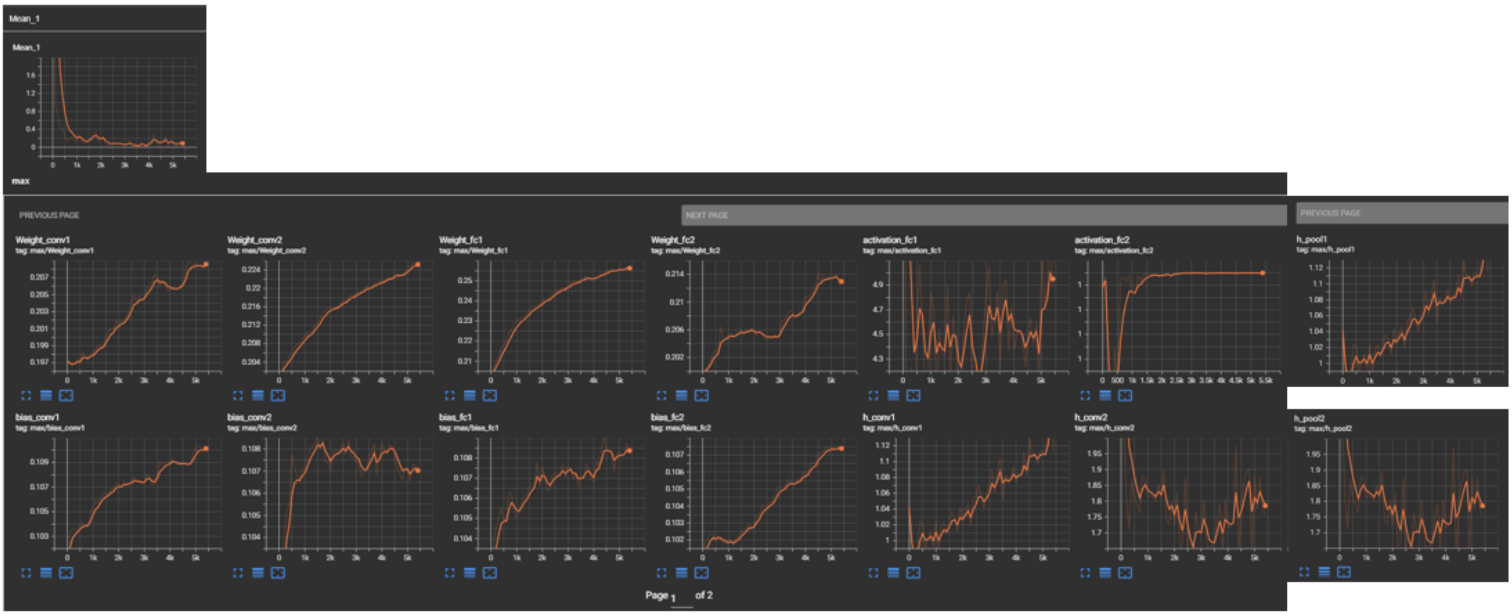


Figure 44. Training result visualize the monitored terms in TensorBoard (Mean_1&max)

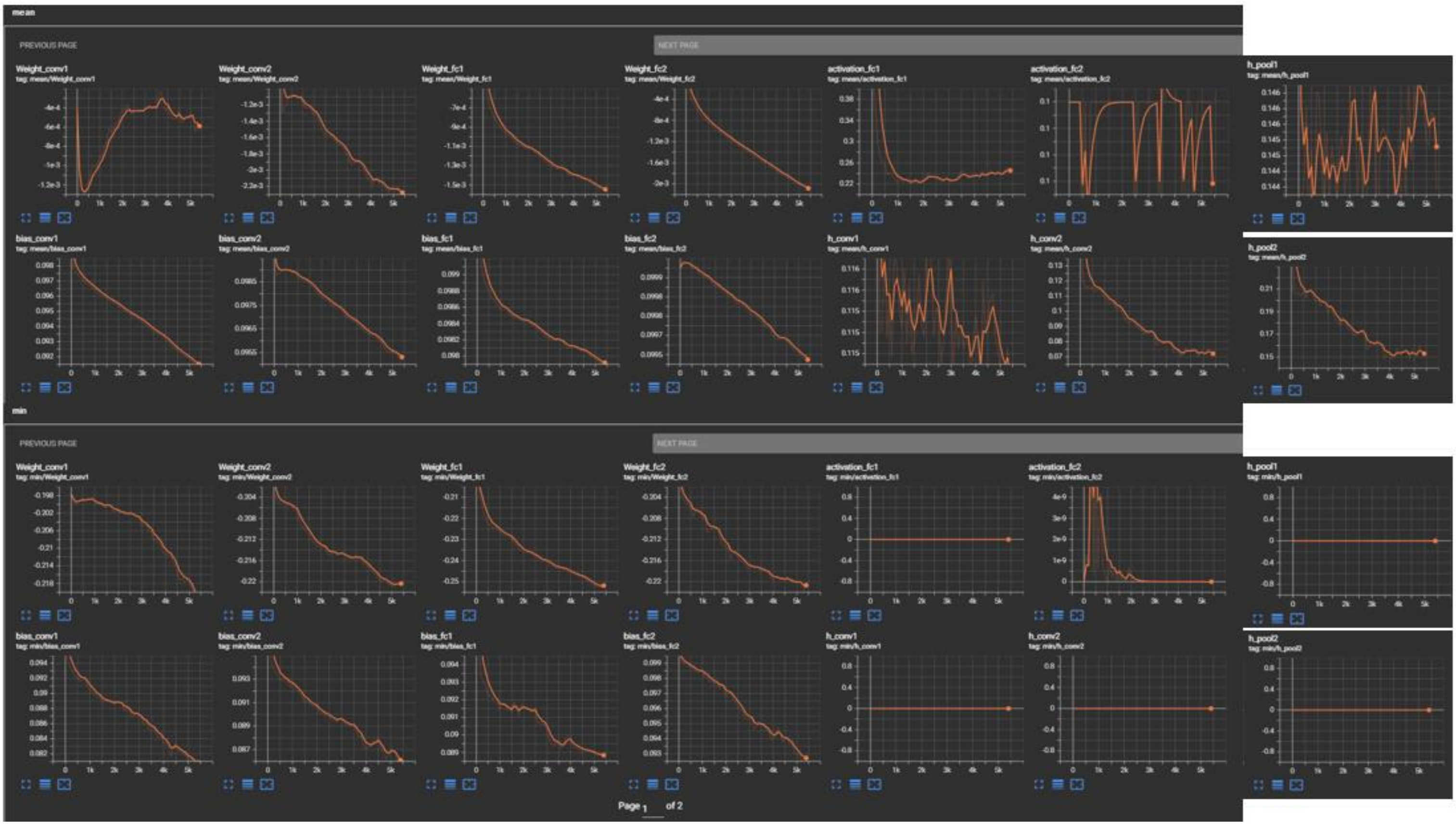


Figure 45. Training result visualize the monitored terms in TensorBoard (mean&min)

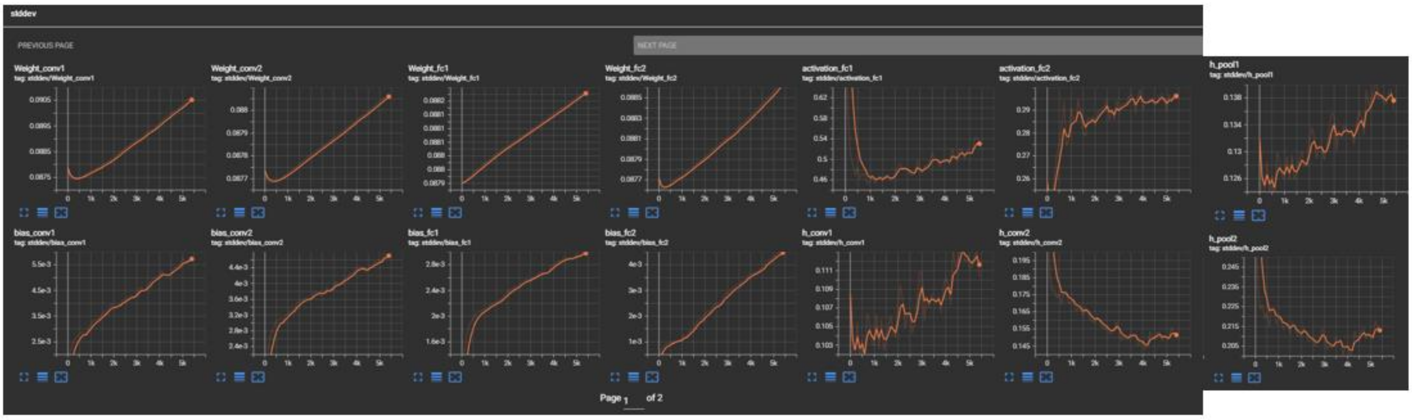


Figure 46. Training result visualize the monitored terms in TensorBoard (stddev)

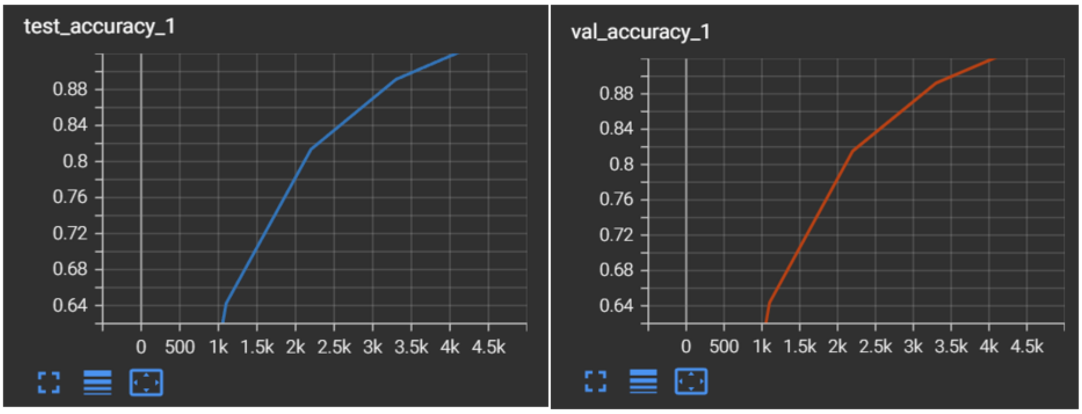


Figure 47. Training result visualize the monitored terms in TensorBoard (test_accuracy&validation_accuracy)

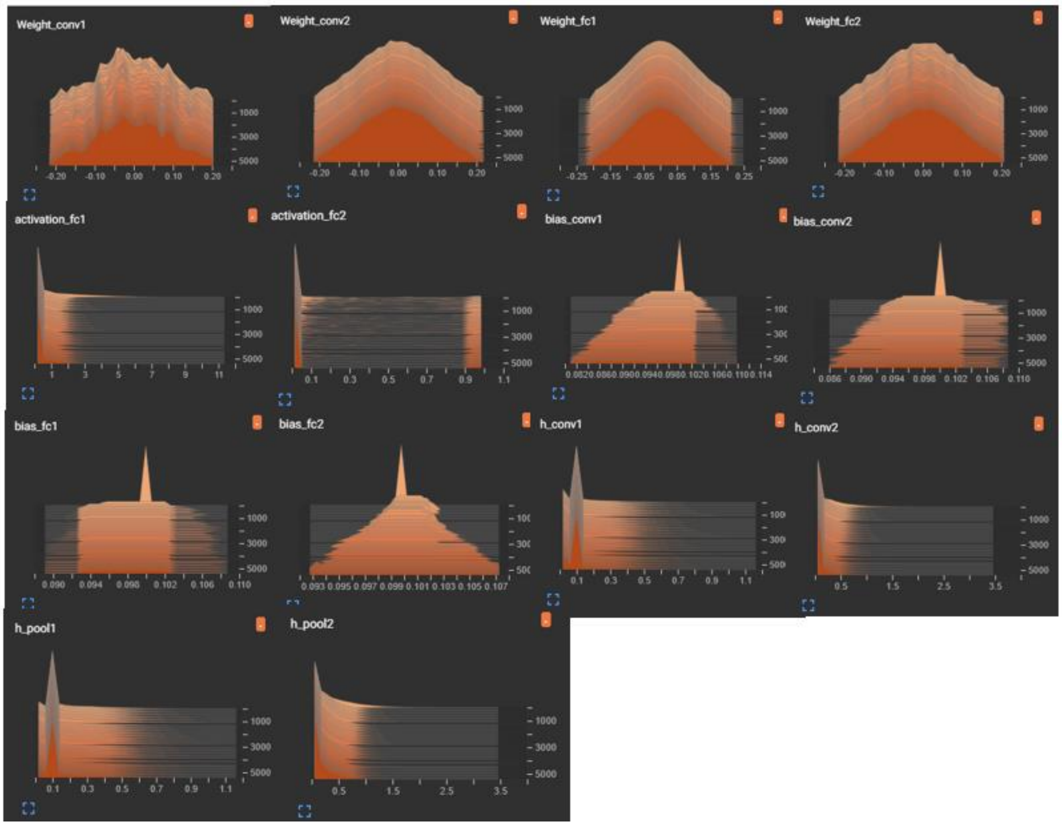


Figure 48. Training result visualize the monitored terms in TensorBoard (Histograms)

(c) Time for More Fun!!!

1. Run the network training with different nonlinearities (Tanh, Sigmoid, leaky-ReLU, MaxOut,...), initialization techniques (Xavier...) and training algorithms (SGD, Momentum-based Methods, Adagrad..).

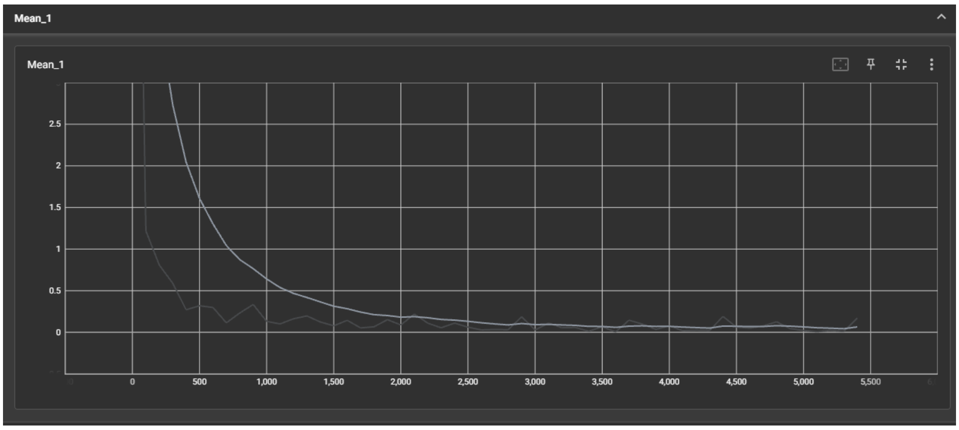
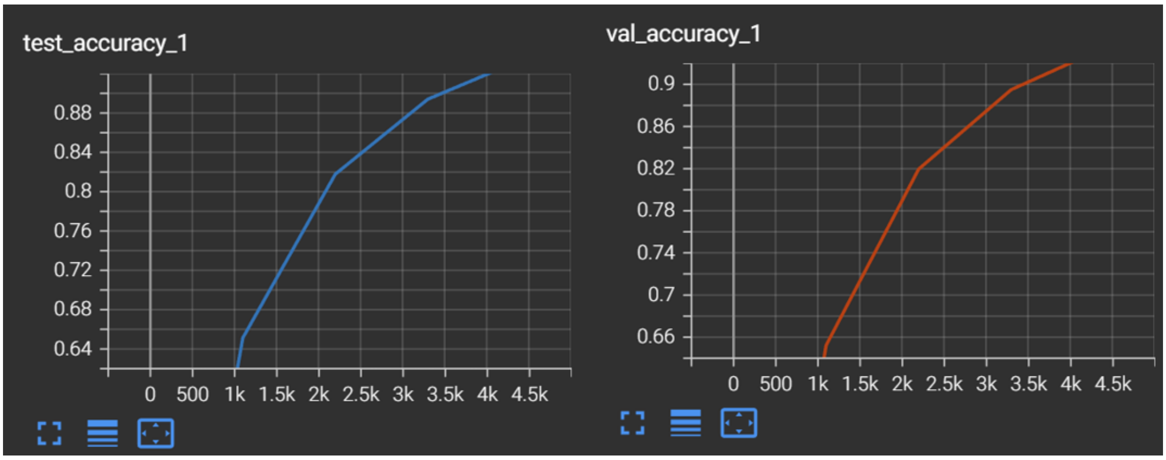


Figure 49. Training result visualize in TensorBoard (leaky-ReLU+Random+Adam)

Figure 50. Training result visualize the monitored terms in TensorBoard



(test_accuracy&validation_accuracy) (leaky-ReLU+Random+Adam)

Table 10. Using different nonlinearities, initialization techniques and training algorithms test accuracy results table

	Tanh test accuracy	Sigmoid test accuracy	ReLU test accuracy	leaky-ReLU test accuracy
Random+Adagrad	0.8528	0.1135	0.8649	0.8553
Xavier+Adagrad	0.7982	0.1135	0.7702	0.7939
Random+Adam	0.9837	0.9571	0.9861	0.9873

My Observation:

I saved all generated data in the A_results, B_results, C_results, and I have tried two type of initialization techniques and training algorithms for each activation functions in the final part. As you can see from the *Table 10*, apparently using leady-ReLU + Random + Adam got the best test accuracy, whereas using Sigmoid almost always got the worst results.