

Deep Learning and Practice Lab 1

309552007

袁鈺勛

1. Introduction

這次 lab 要實作 neural network，利用 forward propagation 得到預測答案，將預測答案和真實答案之間的差距經過 backpropagation 回傳，藉此來更新 layer 的 weights，便可以使預測答案更接近真實答案。其中實作了 layer class 以及 neural network class，neural network class 會以 layer class 來建構出 network，並且可以透過 command line arguments 來控制 neural network 的 optimizer、activation function 以及 learning rate 等，以此來觀察變更 neural network 的架構或參數會對學習過程以及準確度有什麼影響。

2. Experiment setups

A. Sigmoid functions

```
@staticmethod
def sigmoid(x: np.ndarray) → np.ndarray:
    """
    Calculate sigmoid function
     $y = 1 / (1 + e^{-x})$ 
    :param x: input data
    :return: sigmoid results
    """
    return 1.0 / (1.0 + np.exp(-x))

@staticmethod
def derivative_sigmoid(y: np.ndarray) → np.ndarray:
    """
    Calculate the derivative of sigmoid function
     $y' = y(1 - y)$ 
    :param y: value of the sigmoid function
    :return: derivative sigmoid result
    """
    return np.multiply(y, 1.0 - y)
```

上圖為 sigmoid function 以及 derivative sigmoid function，他們被實作在 layer class 內，sigmoid function 用在 forward propagation，derivative sigmoid 則用於 backpropagation。

B. Neural network

```
class NeuralNetwork:
    def __init__(self, epoch: int = 1000000, learning_rate: float = 0.1, num_of_layers: int = 2, input_units: int =
        hidden_units: int = 4, activation: str = 'sigmoid', optimizer: str = 'gd'): ...

    def forward(self, inputs: np.ndarray) → np.ndarray: ...

    def backward(self, derivative_loss) → None: ...

    def update(self) → None: ...

    def train(self, inputs: np.ndarray, labels: np.ndarray) → None: ...

    def predict(self, inputs: np.ndarray) → np.ndarray: ...

    def show_result(self, inputs: np.ndarray, labels: np.ndarray) → None: ...

    @staticmethod
    def mse_loss(prediction: np.ndarray, ground_truth: np.ndarray) → np.ndarray: ...

    @staticmethod
    def mse_derivative_loss(prediction: np.ndarray, ground_truth: np.ndarray) → np.ndarray: ...
```

上圖為 neural network class，具有 forward 和 train 等功能。

```
self.num_of_epoch = epoch
self.learning_rate = learning_rate
self.hidden_units = hidden_units
self.activation = activation
self.optimizer = optimizer
self.learning_epoch, self.learning_loss = list(), list()

# Setup layers
# Input layer
self.layers = [Layer(input_units, hidden_units, activation, optimizer, learning_rate)]

# Hidden layers
for _ in range(num_of_layers - 1):
    self.layers.append(Layer(hidden_units, hidden_units, activation, optimizer, learning_rate))

# Output layer
self.layers.append(Layer(hidden_units, 1, 'sigmoid', optimizer, learning_rate))
```

上圖為 neural network class 的 initial function，會在這個 function 利用 layer class 來建立 neural network。

```
class Layer:
    def __init__(self, input_links: int, output_links: int, activation: str = 'sigmoid', optimizer: str = 'gd',
        learning_rate: float = 0.1): ...

    def forward(self, inputs: np.ndarray) → np.ndarray: ...

    def backward(self, derivative_loss: np.ndarray) → np.ndarray: ...

    def update(self) → None: ...

    @staticmethod
    def sigmoid(x: np.ndarray) → np.ndarray: ...

    @staticmethod
    def derivative_sigmoid(y: np.ndarray) → np.ndarray: ...

    @staticmethod
    def tanh(x: np.ndarray) → np.ndarray: ...

    @staticmethod
    def derivative_tanh(y: np.ndarray) → np.ndarray: ...
```

上圖為 layer class，具有 forward 等功能，還有 sigmoid 等 activation function。

C. Backpropagation

```
def backward(self, derivative_loss) → None:
    """
    Backward propagation
    :param derivative_loss: loss form next layer
    :return: None
    """
    for layer in self.layers[::-1]:
        derivative_loss = layer.backward(derivative_loss)
```

上圖為 neural network 的 backpropagation，他會從 output layer 開始往 input layer 回推，並且將 loss 送給上一層。

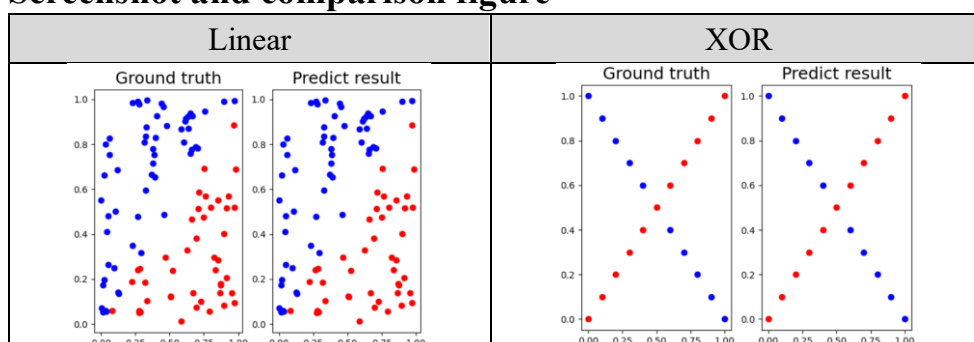
```
def backward(self, derivative_loss: np.ndarray) → np.ndarray:
    """
    Backward propagation
    :param derivative_loss: loss from next layer
    :return: loss of this layer
    """
    if self.activation == 'sigmoid':
        self.backward_gradient = np.multiply(self.derivative_sigmoid(self.output), derivative_loss)
    elif self.activation == 'tanh':
        self.backward_gradient = np.multiply(self.derivative_tanh(self.output), derivative_loss)
    elif self.activation == 'relu':
        self.backward_gradient = np.multiply(self.derivative_relu(self.output), derivative_loss)
    elif self.activation == 'leaky_relu':
        self.backward_gradient = np.multiply(self.derivative_leaky_relu(self.output), derivative_loss)
    else:
        # Without activation function
        self.backward_gradient = derivative_loss

    return np.matmul(self.backward_gradient, self.weight[:-1].T)
```

上圖為 layer 的 backpropagation，會根據所用的 activation function 算出 backward gradient 並且 return 這層的 loss。

3. Results of your testing

A. Screenshot and comparison figure



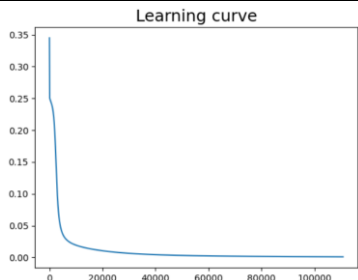
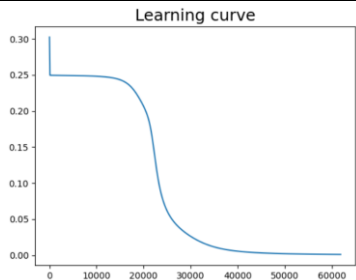
上面的圖表可以看出 network 都準確預測出答案。

B. Show the accuracy of your prediction

Linear	XOR
<pre>[9.99998687e-01] [9.99852269e-01] [9.99998666e-01] [9.57588355e-01] [9.99656478e-01] [9.99997914e-01]] Accuracy : 1.0</pre>	<pre>[0.01167167] [0.99807562] [0.00692729] [0.99811201] [0.00479668] [0.99809027]] Accuracy : 1.0</pre>

由上表可以看出 network 對於 linear 這個比較簡單的問題會給出較接近 1 或 0 的機率。

C. Learning curve

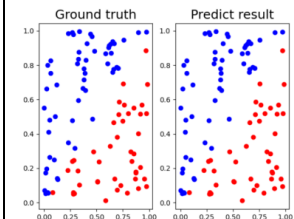
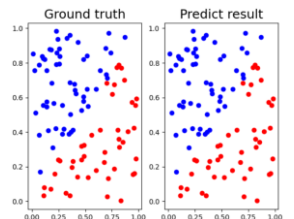
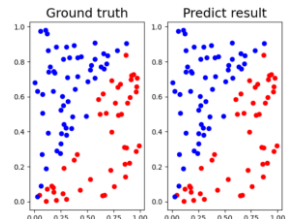
Linear	XOR
	
<pre>Epoch 110100 loss : 0.0010101232706363975 Epoch 110200 loss : 0.001008820522543597 Epoch 110300 loss : 0.001007520637030701 Epoch 110400 loss : 0.0010062236050612159 Epoch 110500 loss : 0.0010049294176359982 Epoch 110600 loss : 0.0010036380657930893 Epoch 110700 loss : 0.0010023495406075264 Epoch 110800 loss : 0.001001063833191119</pre>	<pre>Epoch 61300 loss : 0.0010331263747193763 Epoch 61400 loss : 0.0010282123426948492 Epoch 61500 loss : 0.0010233382990534419 Epoch 61600 loss : 0.00101850379522712 Epoch 61700 loss : 0.001013708388968634 Epoch 61800 loss : 0.0010089516442454039 Epoch 61900 loss : 0.0010042331311354295</pre>

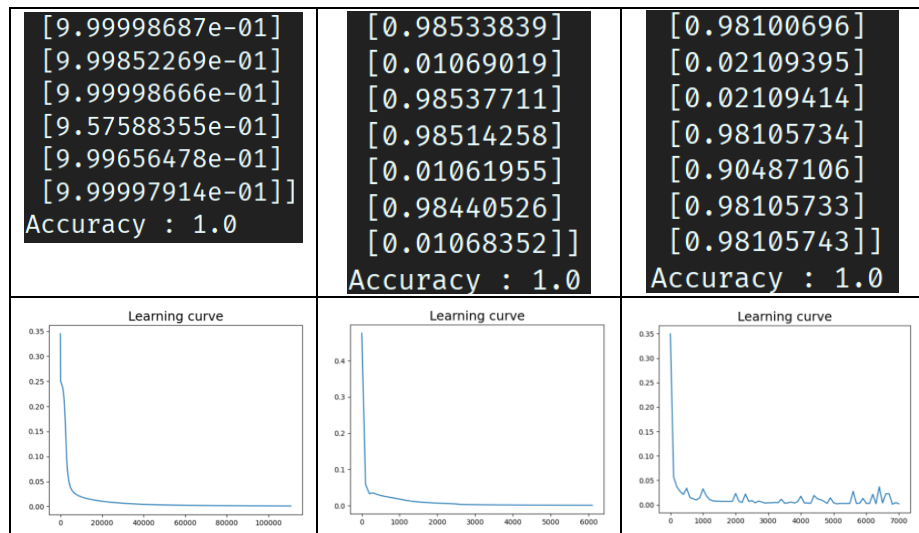
由上表可以看出在 linear 這個比較簡單的問題時不太會出現像 XOR 一樣在收斂前會在某處持續一段 epoch 都是差不多的 loss，然後才開始往下降。

4. Discussion

A. Try different learning rates

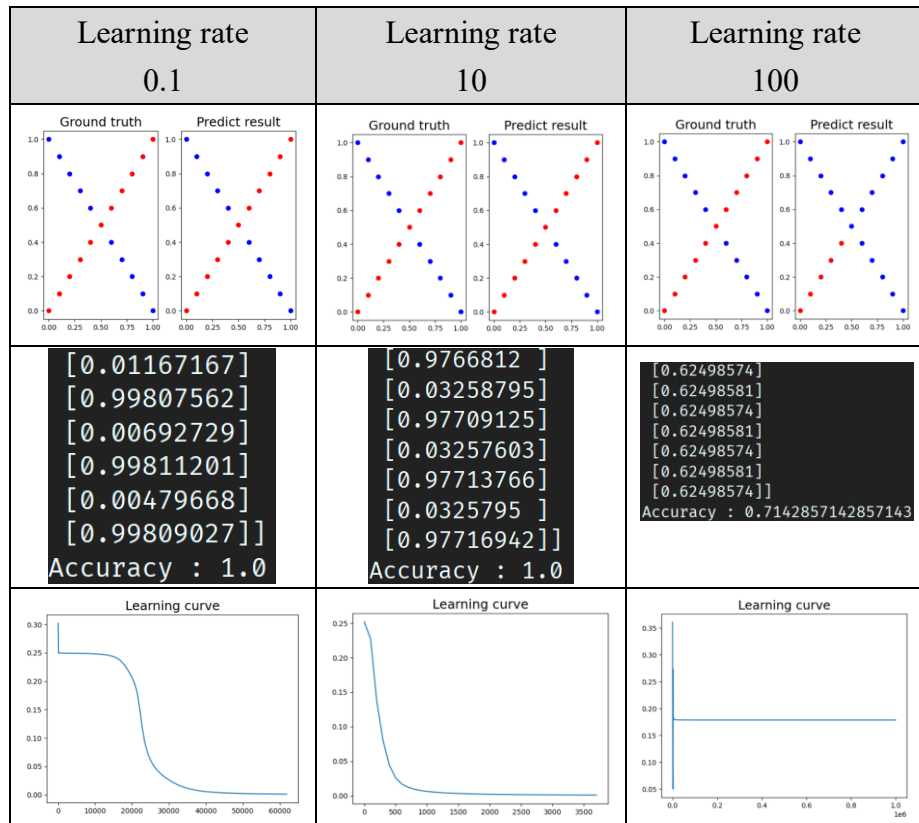
a. Linear

Learning rate 0.1	Learning rate 10	Learning rate 100
		



固定 hidden layer 有 4 個 units，由上表可以看出當 learning rate 太大的時候，learning curve 便會出現震盪。

b. XOR

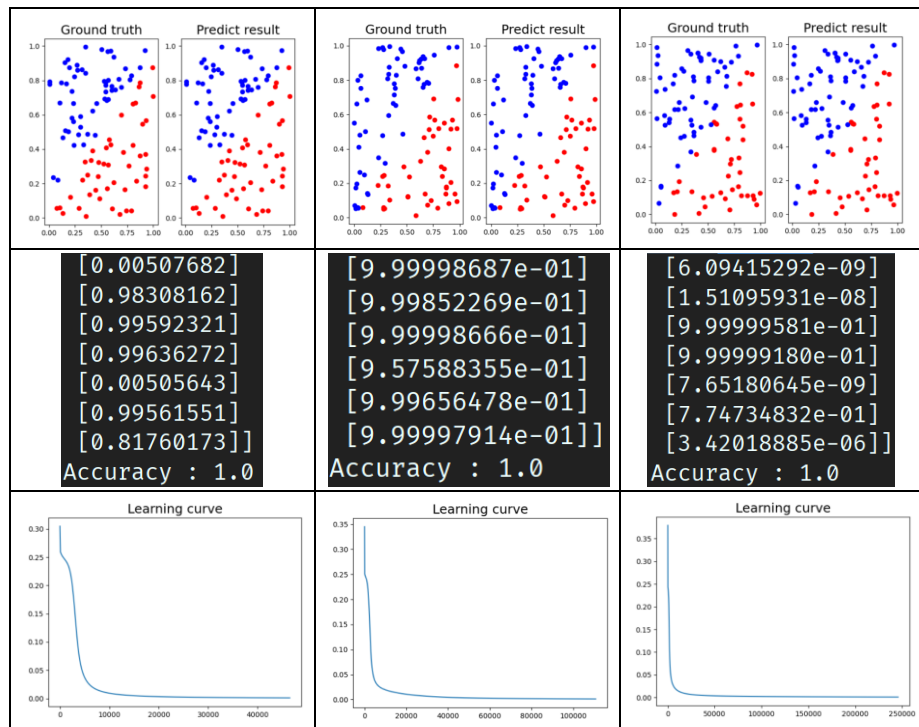


和 linear 一樣固定 units，同樣在 learning rate 太大的時候會出現震盪，甚至影響 accuracy。

B. Try different numbers of hidden units

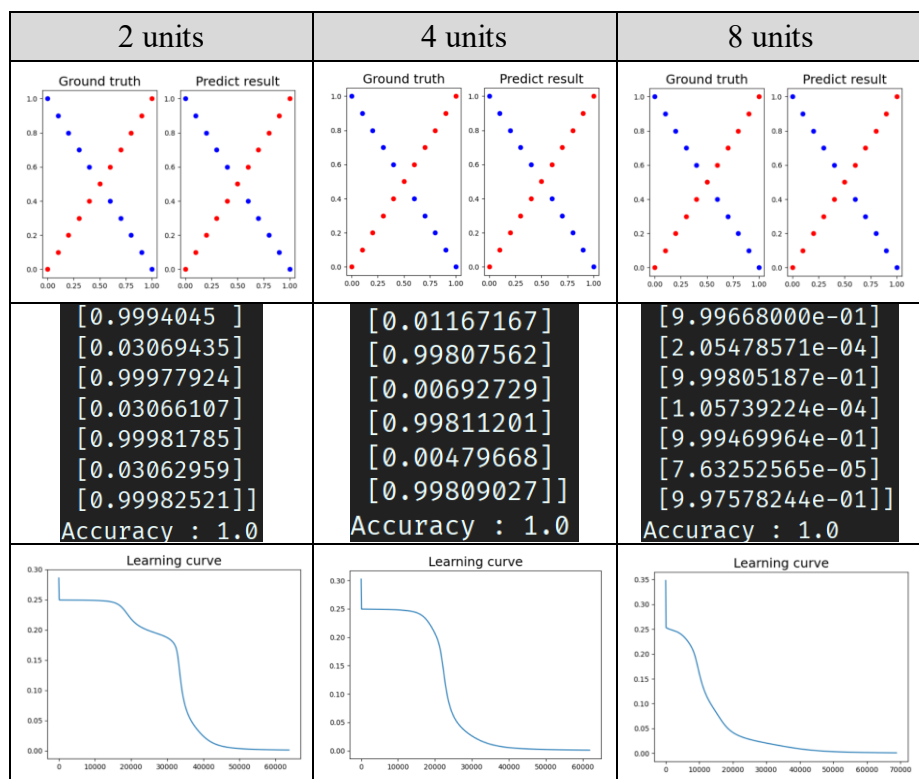
a. Linear

2 units	4 units	8 units
---------	---------	---------



固定 learning rate 為 0.1，由上表可以看出由左至右當 unit 增加會造成較慢收斂，可能是因為 linear 是簡單的問題但用太多 unit 反而只是增加複雜度和 loss 總值而已。

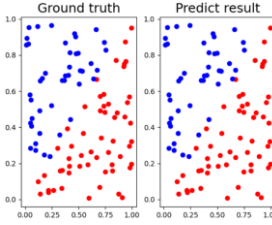
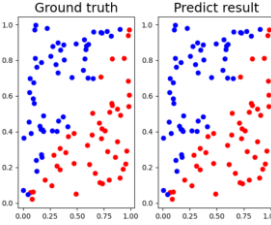
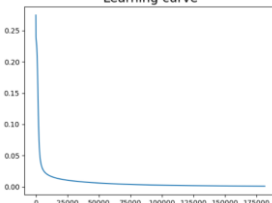
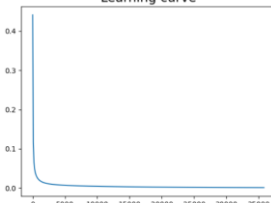
b. XOR



和 linear 一樣固定 learning rate，但表中不同 unit 的收斂時間看起來沒有太大的差異，可是當 unit 增加時，learning curve 變得較為平滑，呈現出 unit 較多時學習成效較好的結果。

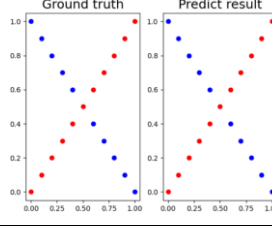
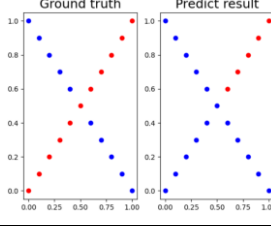
C. Try without activation functions

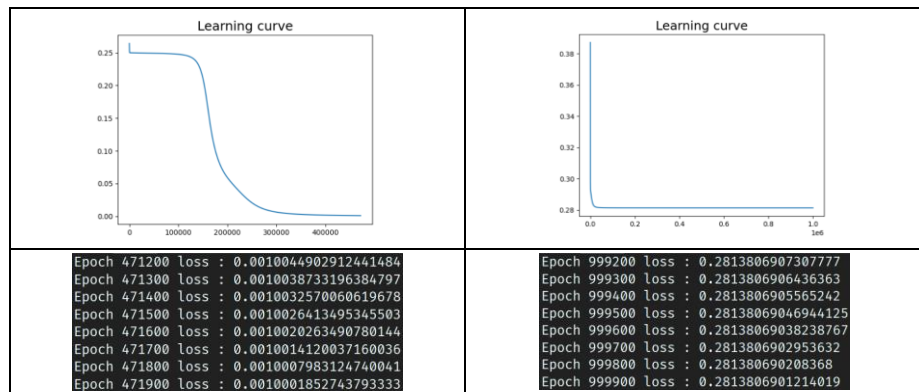
a. Linear

With activation	Without activation
	
<pre>[9.99999891e-01] [9.05340638e-05] [9.99999899e-01] [2.93623947e-06] [9.99999898e-01]] Activation : sigmoid Hidden units : 4 Accuracy : 1.0</pre>	<pre>[9.99998251e-01] [7.91252758e-01] [1.00000000e+00] [3.92637335e-02] [1.00000000e+00]] Activation : none Hidden units : 4 Accuracy : 1.0</pre>
	
<pre>Epoch 181000 loss : 0.0010063555416430702 Epoch 181100 loss : 0.0010054380037735143 Epoch 181200 loss : 0.0010045217441144972 Epoch 181300 loss : 0.0010036067603221002 Epoch 181400 loss : 0.0010026930500576518 Epoch 181500 loss : 0.0010017806109878294 Epoch 181600 loss : 0.0010008694407845273</pre>	<pre>Epoch 35200 loss : 0.0010262356915727345 Epoch 35300 loss : 0.001022192505281284 Epoch 35400 loss : 0.0010181737161752509 Epoch 35500 loss : 0.0010141791230895924 Epoch 35600 loss : 0.0010102085269639727 Epoch 35700 loss : 0.001006261730816727 Epoch 35800 loss : 0.0010023385397192367</pre>

固定 hidden layer 有 4 個 units 且 learning rate 為 0.1，由上表看不太出來 w/o activation function 對 accuracy 有沒有影響，但是 with activation 時會收斂較慢可能是因為 sigmoid 會將 output 限縮在 0 至 1，且會有 vanishing gradient 作用。

b. XOR

With activation	Without activation
	
<pre>[0.99981919] [0.0291145] [0.99984116] [0.0289333] [0.99984352]] Activation : sigmoid Hidden units : 4 Accuracy : 1.0</pre>	<pre>[9.08765843e-01] [2.90285824e-11] [9.08765691e-01] [3.79278352e-14] [9.08765539e-01]] Activation : none Hidden units : 4 Accuracy : 0.7142857142857143</pre>



和 linear 一樣固定 4 個 units，但 learning rate 改為 0.01，由上表便可明顯看出 with activation 會收斂得比較快，without activation 反而要跑到 epoch 的上限才會停，同時 without activation 也對 accuracy 產生了影響。

5. Extra

A. Implement different optimizers

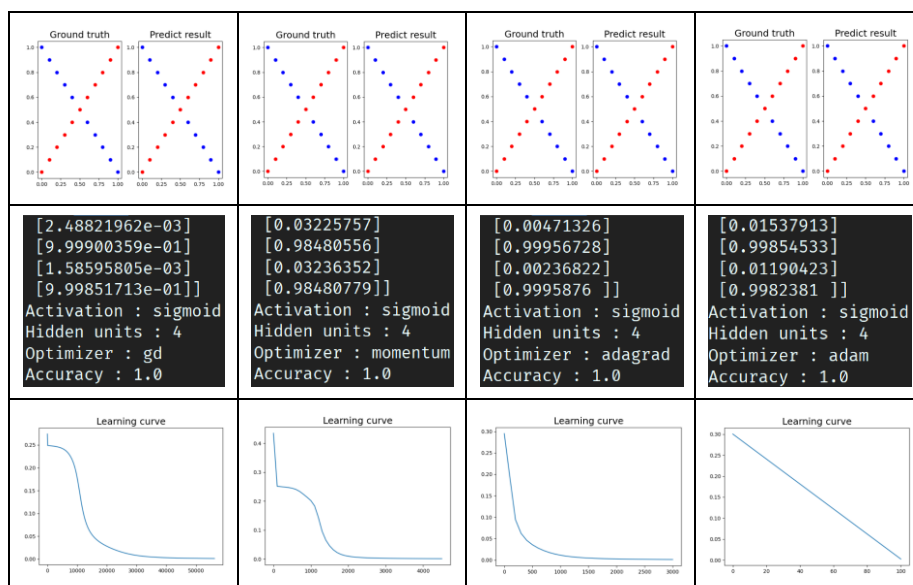
a. Linear

Gradient descent	Momentum	Adagrad	Adam
<pre>[5.43837312e-04] [9.99522278e-01] [9.99649421e-01] [9.99638969e-01]] Activation : sigmoid Hidden units : 4 Optimizer : gd Accuracy : 1.0</pre>	<pre>[3.90445653e-06] [2.48629448e-06] [2.21226426e-06] [3.15579561e-06]] Activation : sigmoid Hidden units : 4 Optimizer : momentum Accuracy : 1.0</pre>	<pre>[1.23602148e-03] [4.68361330e-05] [5.11813966e-05] [4.61372595e-05]] Activation : sigmoid Hidden units : 4 Optimizer : adagrad Accuracy : 1.0</pre>	<pre>[1.36682245e-04] [9.99947257e-01] [9.99948065e-01] [1.41993036e-04]] Activation : sigmoid Hidden units : 4 Optimizer : adam Accuracy : 1.0</pre>

固定 hidden layer 是 4 個 units 且 learning rate 是 0.1，由上面的表格可以看出來由左至右收斂的速度愈快，因為除了 gradient descent 之外的 optimizer 會利用到 momentum 或是降低 learning rate 的方式來阻止震盪和加速收斂。

b. XOR

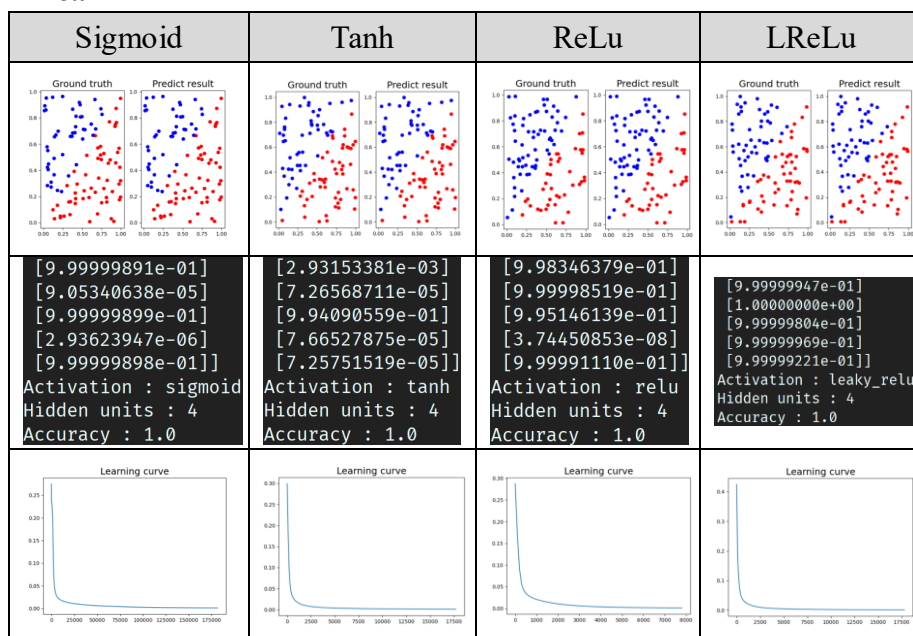
Gradient descent	Momentum	Adagrad	Adam
------------------	----------	---------	------



由上表可以看出 XOR 的結果也和 linear 的一樣，同時 learning curve 看上去也由左至右更加平滑。

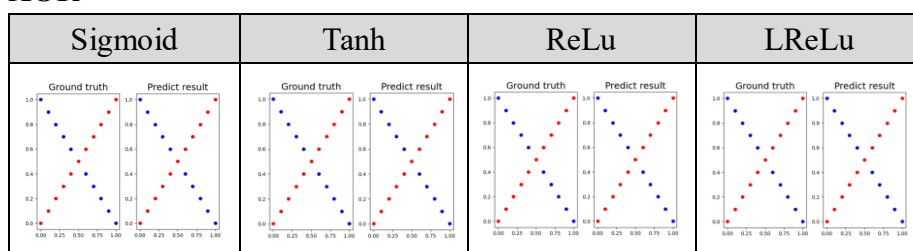
B. Implement different activation functions

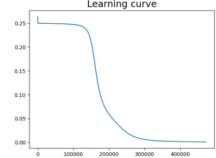
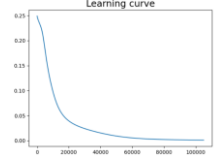
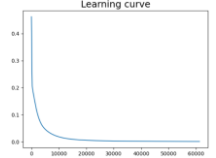
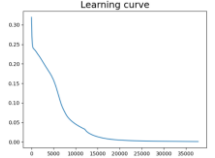
a. Linear



固定 hidden layer 有 4 個 units 且 learning rate 是 0.1，上面的結果看不太出來各 activation 的差距，只有 ReLu 是比較快收斂而已。

b. XOR



<pre> [0.99981919] [0.0291145] [0.99984116] [0.0289333] [0.99984352]] Activation : sigmoid Hidden units : 4 Accuracy : 1.0 </pre>	<pre> [9.99700852e-01] [1.28863330e-04] [9.99201298e-01] [1.15960890e-04] [9.90570954e-01]] Activation : tanh Hidden units : 4 Accuracy : 1.0 </pre>	<pre> [0.99998833] [0.03868819] [0.99999907] [0.03868819] [0.99999993]] Activation : relu Hidden units : 4 Accuracy : 1.0 </pre>	<pre> [0.99990808] [0.03112347] [0.99999333] [0.0311353] [0.99999951]] Activation : leaky_relu Hidden units : 4 Accuracy : 1.0 </pre>
			

和 linear 一樣固定 units 和 learning rate，但上表可以看出由左至右收斂的速度愈快，可能是 vanishing gradient 的影響。