Deep Learning Lab1: Back Propagation

312553024 江尚軒

1. Introduction (20%)

In this lab, I understand and implement a simple neural network with two hidden layers, including a forwarding pass and backpropagation. I only use Python standard libraries and Numpy to complete this task.
(later)

2. Experiment setups (30%)

A. Sigmoid functions

The sigmoid function is implemented like this.

```python
@staticmethod
def sigmoid(x: np.ndarray) -> np.ndarray:
    # Calculate sigmoid function
    # y = 1 / (1 + e^(-x))
    return 1.0 / (1.0 + np.exp(-x))
```

I also implemented the derivative sigmoid function for backpropagation.

```python
@staticmethod
def derivative_sigmoid(y: np.ndarray) -> np.ndarray:
    # Calculate the derivative of sigmoid function
    # y' = y(1 - y)
    return np.multiply(y, 1.0 - y)
```

B. Neural network

My neural network combines one input layer, two hidden layers, and one output layer.

```python
# Setup layers
self.layers = []
# Input layer
self.layers.append(layer.Layer(input_units, hidden_units, activation, optimizer, learning_rate))
# Hidden layers
for _ in range(num_of_layers - 1):
    self.layers.append(layer.Layer(hidden_units, hidden_units, activation, optimizer, learning_rate))
# Output layer
self.layers.append(layer.Layer(hidden_units, 1, 'sigmoid', optimizer, learning_rate))
```

In the forward pass, it will iteratively call the forward function of each layer from the input layer to the output layer.

```python
def forward(self, inputs: np.ndarray) -> np.ndarray:
    # Forward pass
    for layer in self.layers:
        inputs = layer.forward(inputs)
    return inputs
```

The forward function of each layer. Multiple inputs and weights first, and then go through the activation function.

```python
def forward(self, inputs: np.ndarray) -> np.ndarray:
    # Forward feed
    # y = wx + b
    self.forward_gradient = np.append(
        inputs, np.ones((inputs.shape[0], 1)), axis=1)

    if self.activation == 'sigmoid':
        self.output = self.sigmoid(
            np.matmul(self.forward_gradient, self.weight))
    elif self.activation == 'tanh':
        self.output = self.tanh(
            np.matmul(self.forward_gradient, self.weight))
    elif self.activation == 'relu':
        self.output = self.relu(
            np.matmul(self.forward_gradient, self.weight))
    elif self.activation == 'leaky_relu':
        self.output = self.leaky_relu(
            np.matmul(self.forward_gradient, self.weight))
    else:
        # Without activation function
        self.output = np.matmul(self.forward_gradient, self.weight)

    return self.output
```

C. Backpropagation

In the backward pass, it will iteratively call the forward function of each layer from the output layer to the input layer. Then, it updates the weights of each layer by gradient descent.

```python
def backward(self, derivative_loss) -> None:
    # Backward pass
    for layer in self.layers[::-1]:
        derivative_loss = layer.backward(derivative_loss)

def update(self) -> None:
    # Update weights
    for layer in self.layers:
        layer.update()
```

The backward function for each layer. Do the derivative activation function, then multiply with derivative loss. Finally, Multiply the backward gradient with the transposed weight.

```
def backward(self, derivative_loss: np.ndarray) -> np.ndarray:
    # Backward propagation
    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)
```

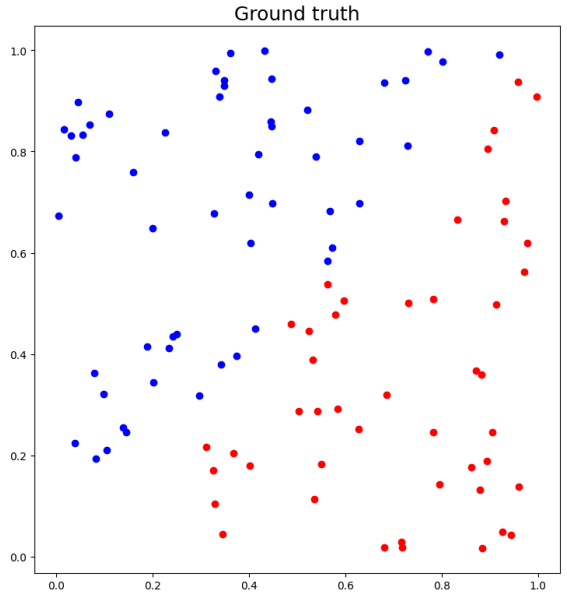
The update function for each layer. First, multiply the forward gradient by the backward gradient, and multiply by the negative learning rate. Lastly, update the weights.

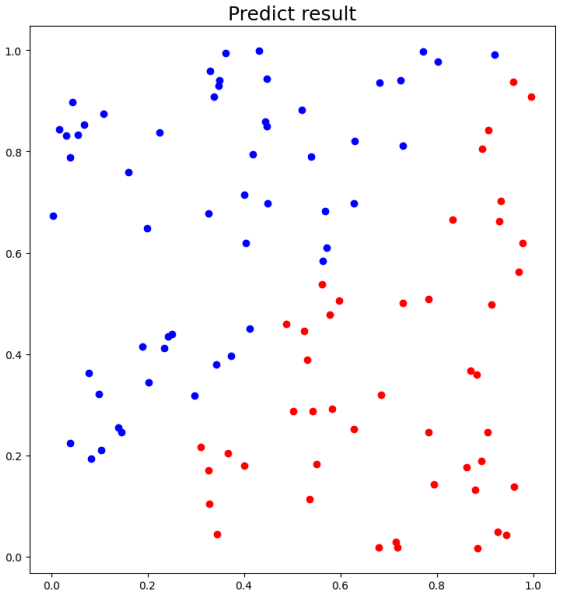
```
def update(self) -> None:
    # Update weights
    gradient = np.matmul(self.forward_gradient.T, self.backward_gradient)
    delta_weight = -self.learning_rate * gradient

    self.weight += delta_weight
```

3. Results of your testing (20%)

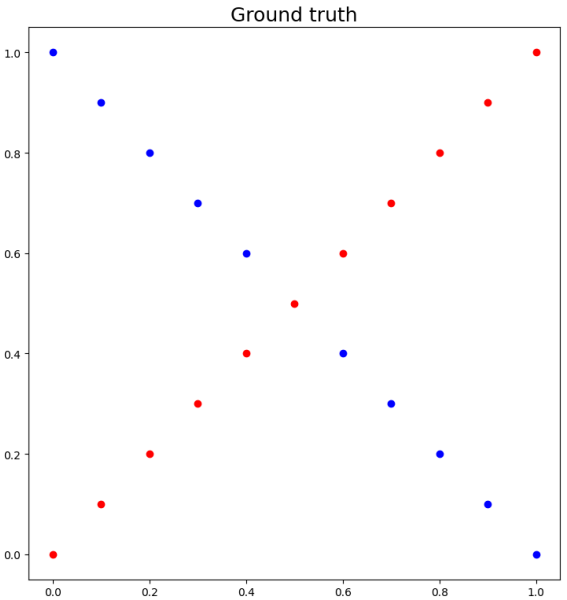
A. Screenshot and comparison figure

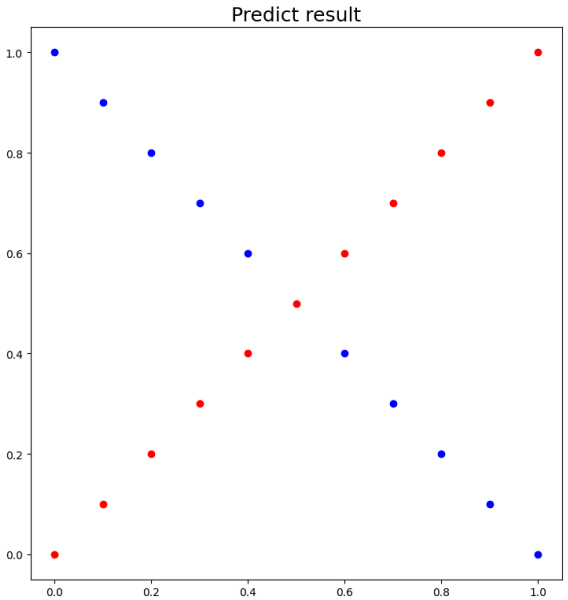
Linear data





XOR data





B. Show the accuracy of your prediction

Linear data

```
Accuracy: 1.0
```

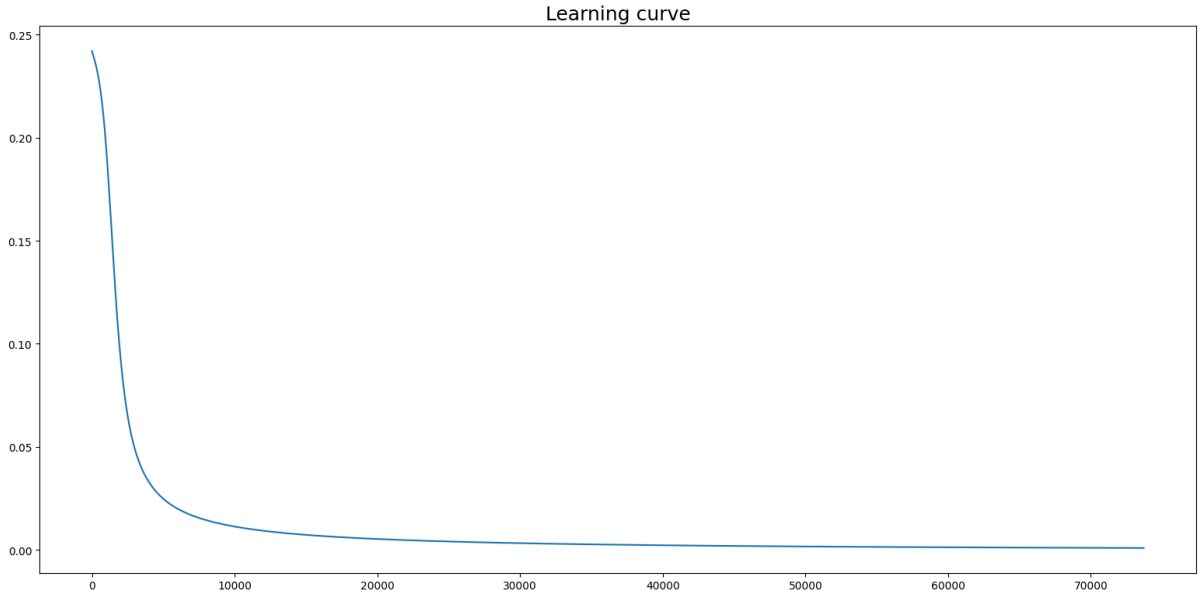
XOR data

```
Accuracy: 1.0
```

C. Learning curve (loss, epoch curve)

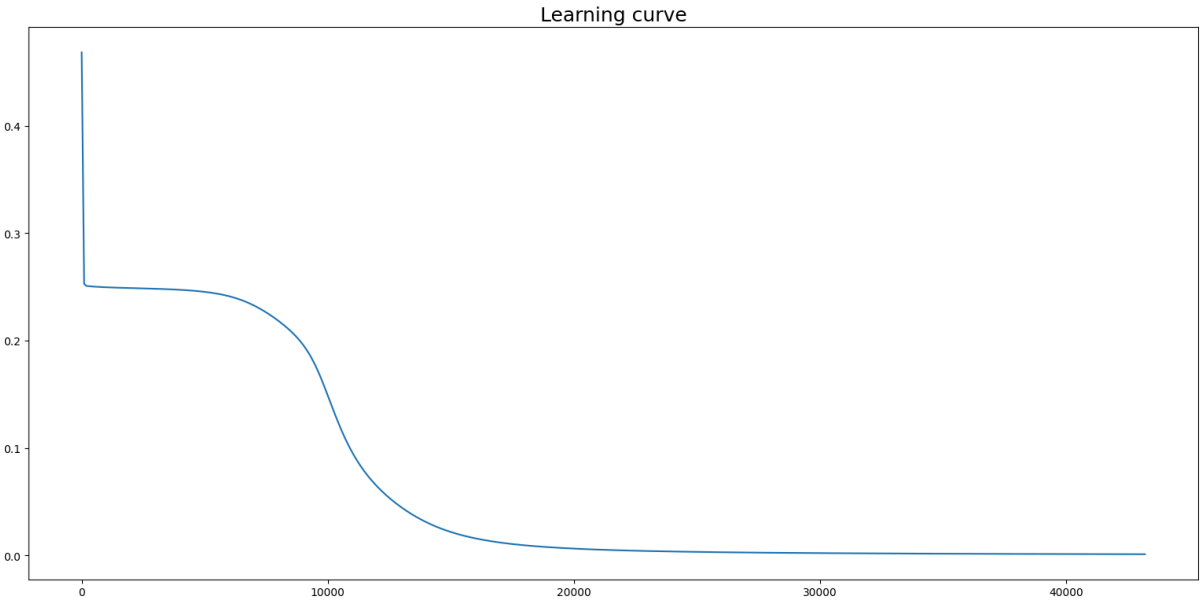
Linear data

Loss converges quickly, about 70,000



XOR data

Loss converges more quickly, about 40,000



D. Anything you want to present

Linear data

Training log

```
Epoch: 73300    Loss: 0.0010097074637215 54
Epoch: 73400    Loss: 0.0010077341581693434
Epoch: 73500    Loss: 0.0010057670793225287
Epoch: 73600    Loss: 0.0010038062003217732
Epoch: 73700    Loss: 0.001001851494455292
```

Testing log

```
Point: 95       Label: [0]      Prediction: [0.]
Point: 96       Label: [0]      Prediction: [0.]
Point: 97       Label: [0]      Prediction: [0.]
Point: 98       Label: [1]      Prediction: [1.]
Point: 99       Label: [1]      Prediction: [1.]
```

XOR data

Training log

```
Epoch: 42800    Loss: 0.0010197650717524 13
Epoch: 42900    Loss: 0.001015655223549505
Epoch: 43000    Loss: 0.0010115766179789 76
Epoch: 43100    Loss: 0.0010075289105865135
Epoch: 43200    Loss: 0.0010035117618614564
```

Testing log

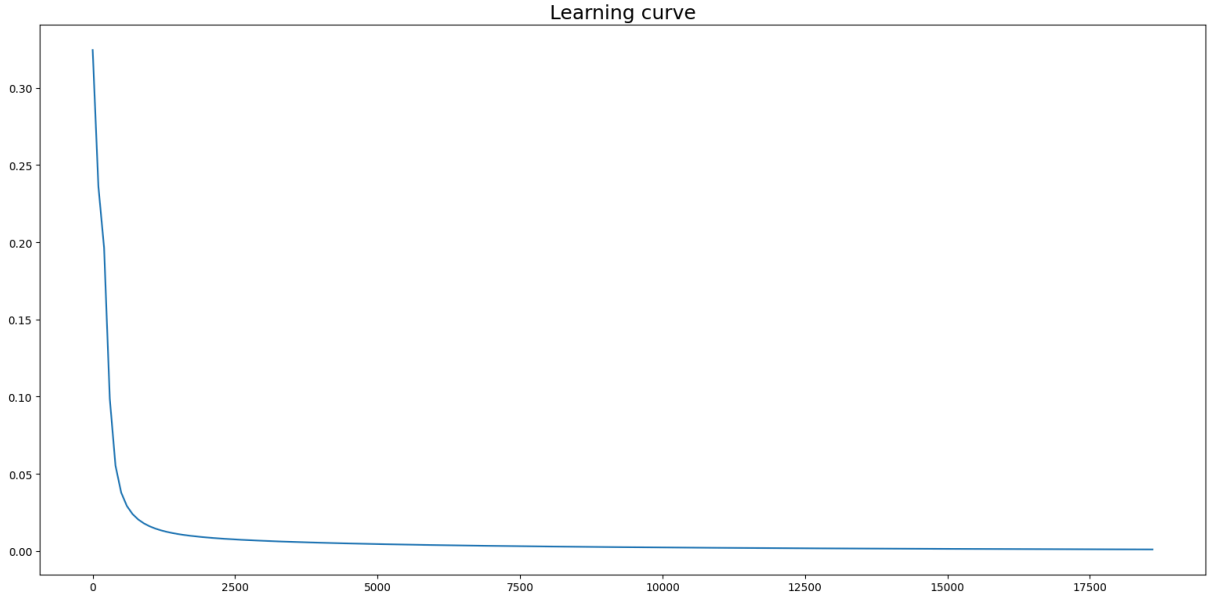
```
Point: 16       Label: [1]      Prediction: [1.]
Point: 17       Label: [0]      Prediction: [0.]
Point: 18       Label: [1]      Prediction: [1.]
Point: 19       Label: [0]      Prediction: [0.]
Point: 20       Label: [1]      Prediction: [1.]
```

4. Discussion (30%)

A. Try different learning rates

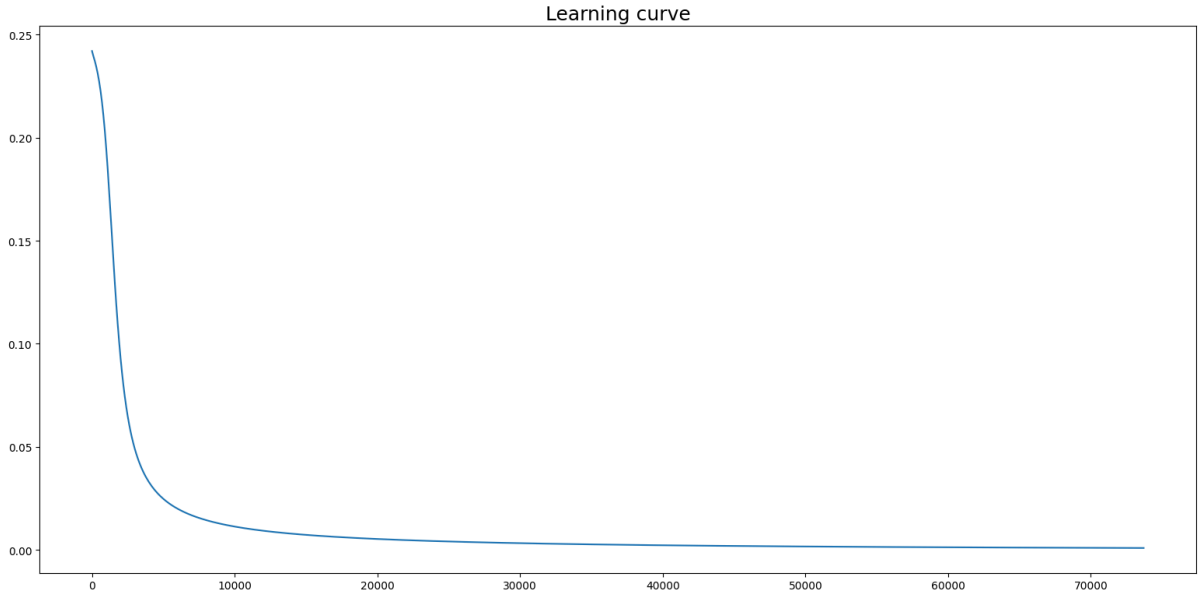
learning rate = 1

Loss converges quickly, about 17,500



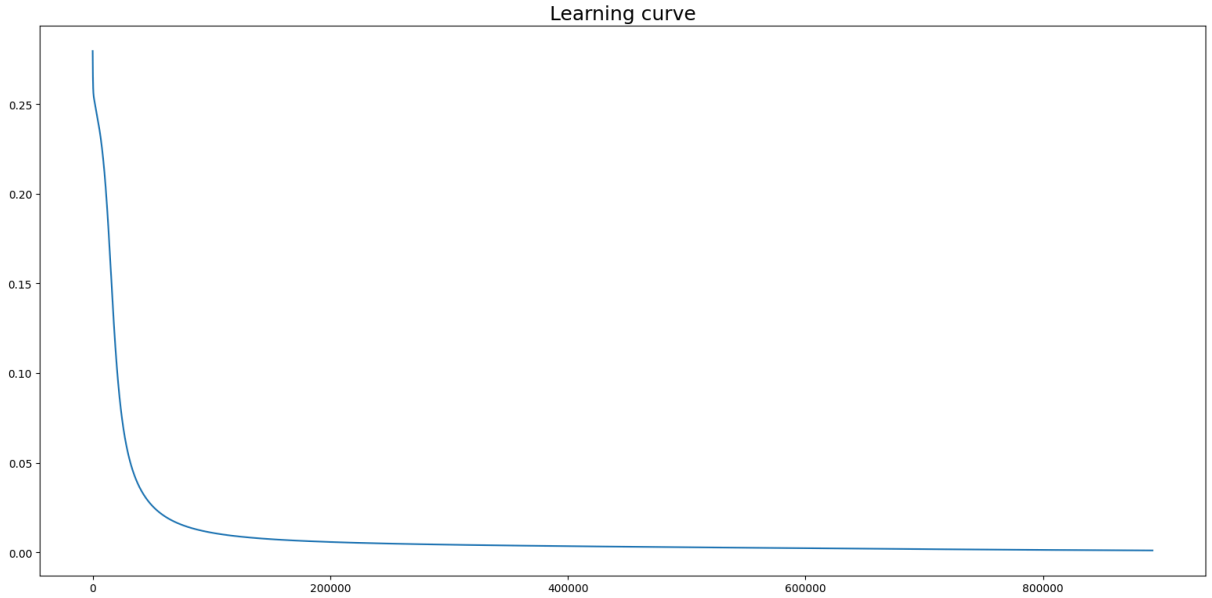
learning rate = 0.1

Loss converges normally, about 70,000



learning rate = 0.01

Loss converges slowly, about 800,000

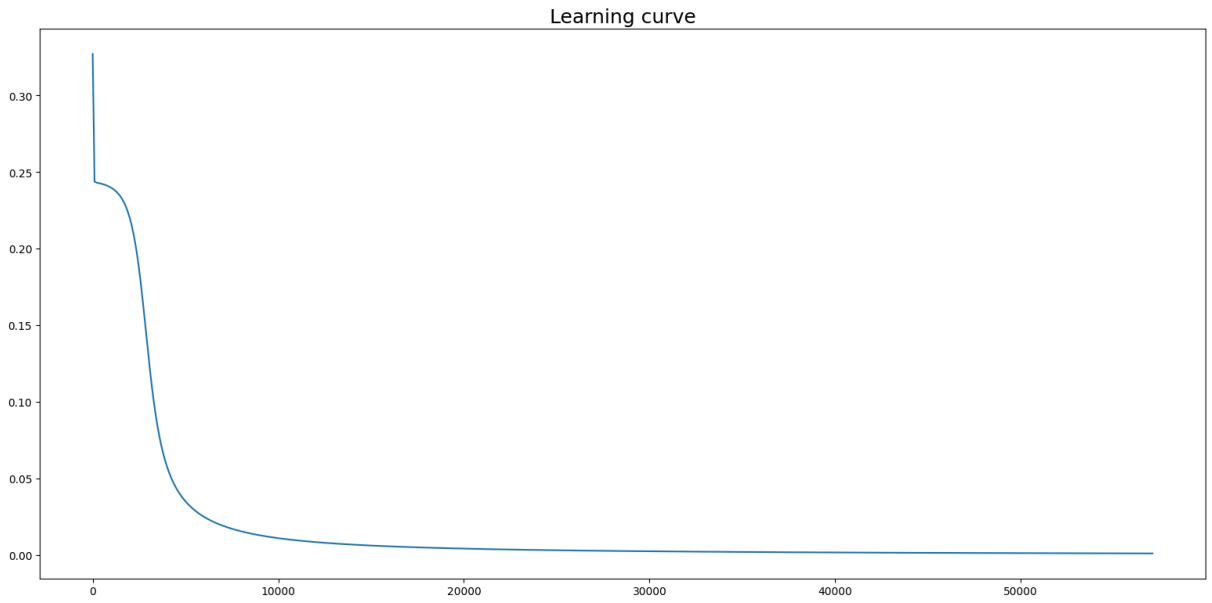


Conclusion: The bigger the learning rate, the faster the loss convergence.

B. Try different numbers of hidden units

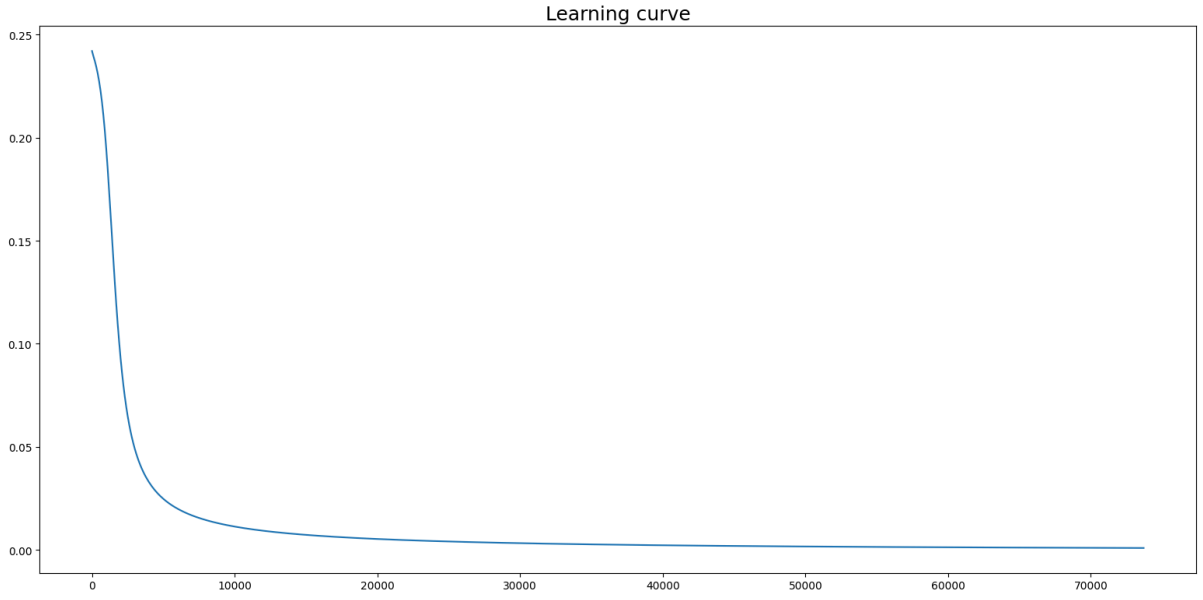
hidden units = 2

Loss converges quickly, about 50000



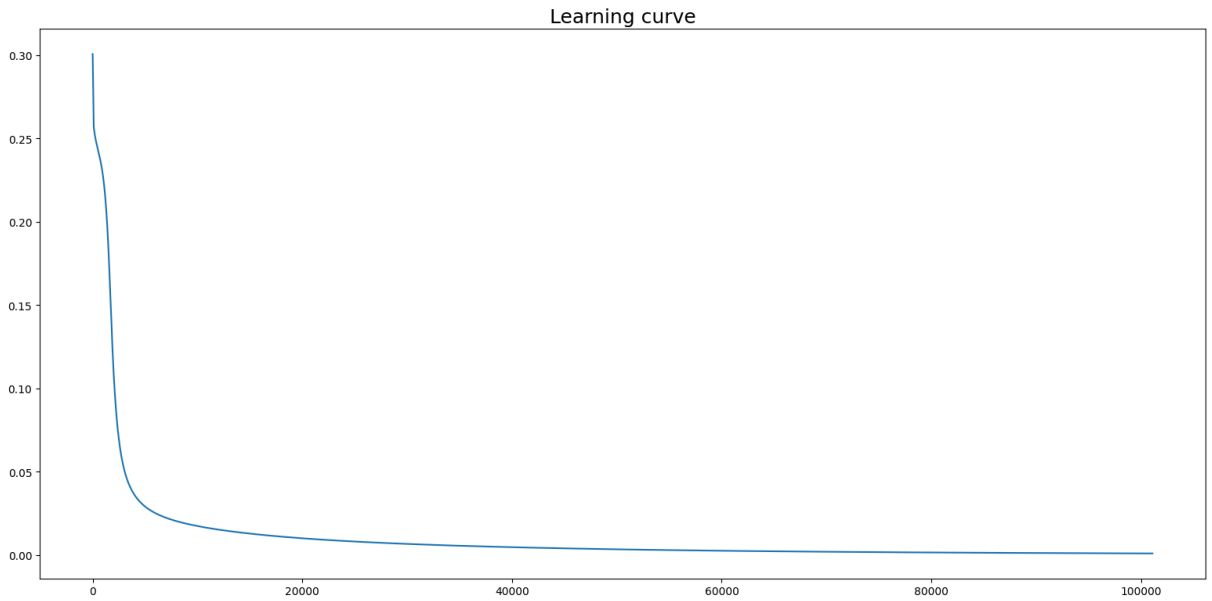
hidden units = 4

Loss converges normally, about 70,000



hidden units = 8

Loss converges slowly, about 100,000



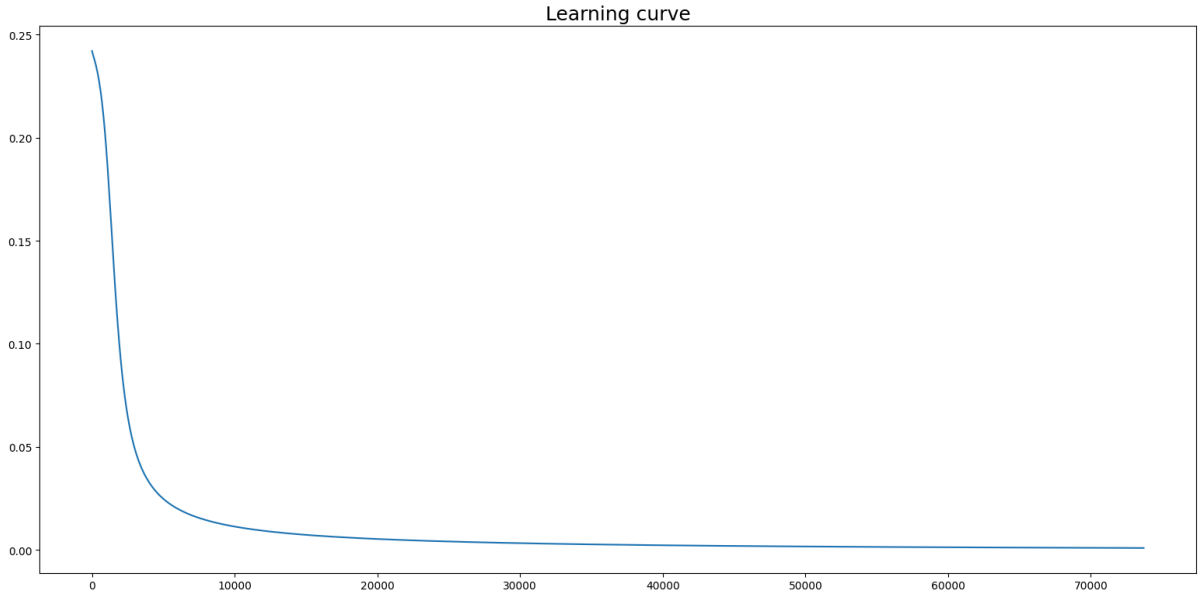
Conclusion: The smaller the hidden units, the faster the loss convergence.

C. Try without activation functions

With activation function

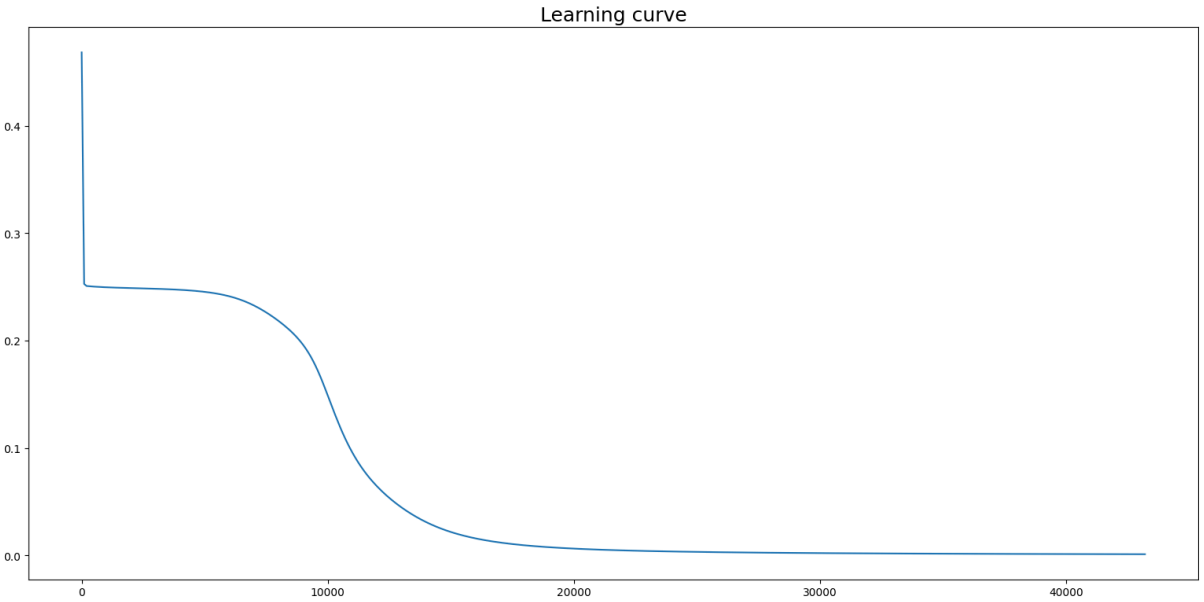
Linear data

Can converge



XOR data

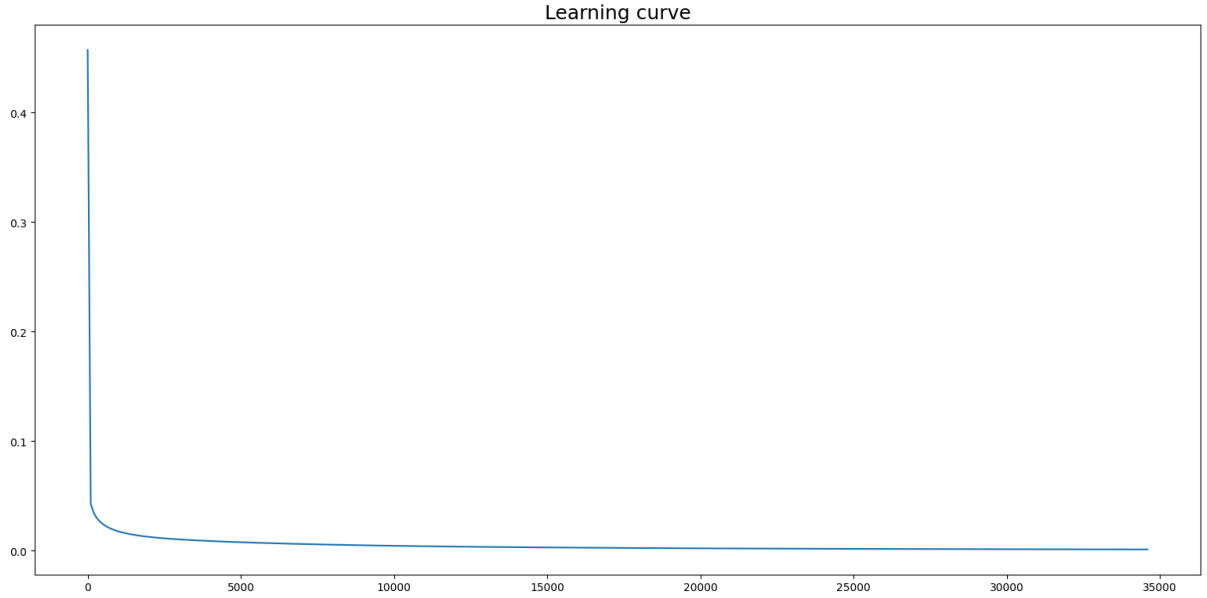
Can converge



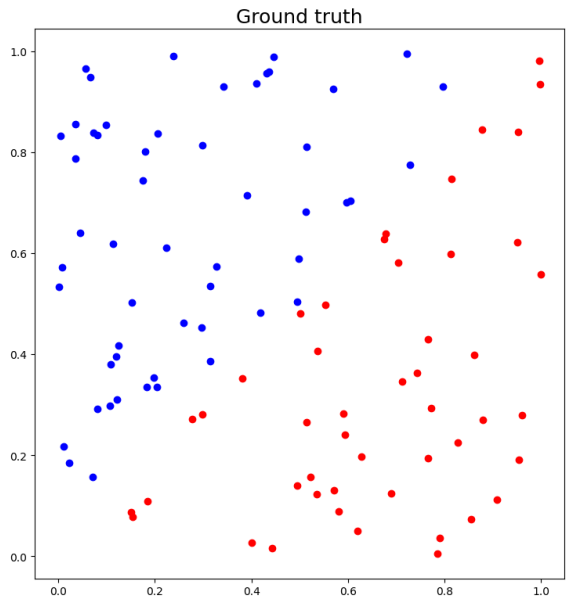
Without activation function

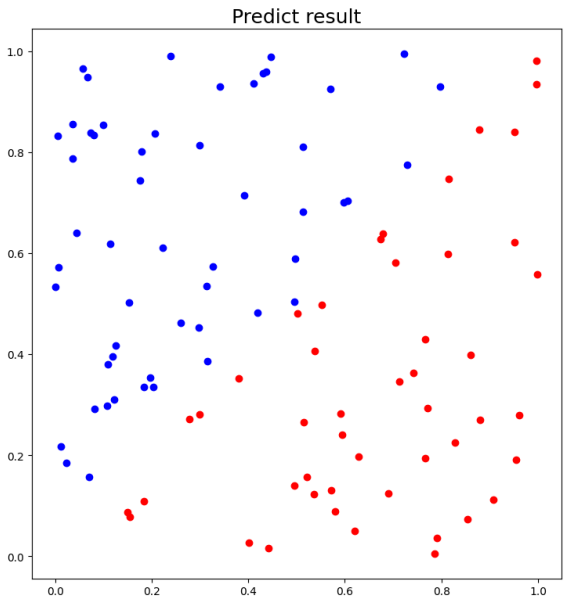
Linear data

Can converge, even faster



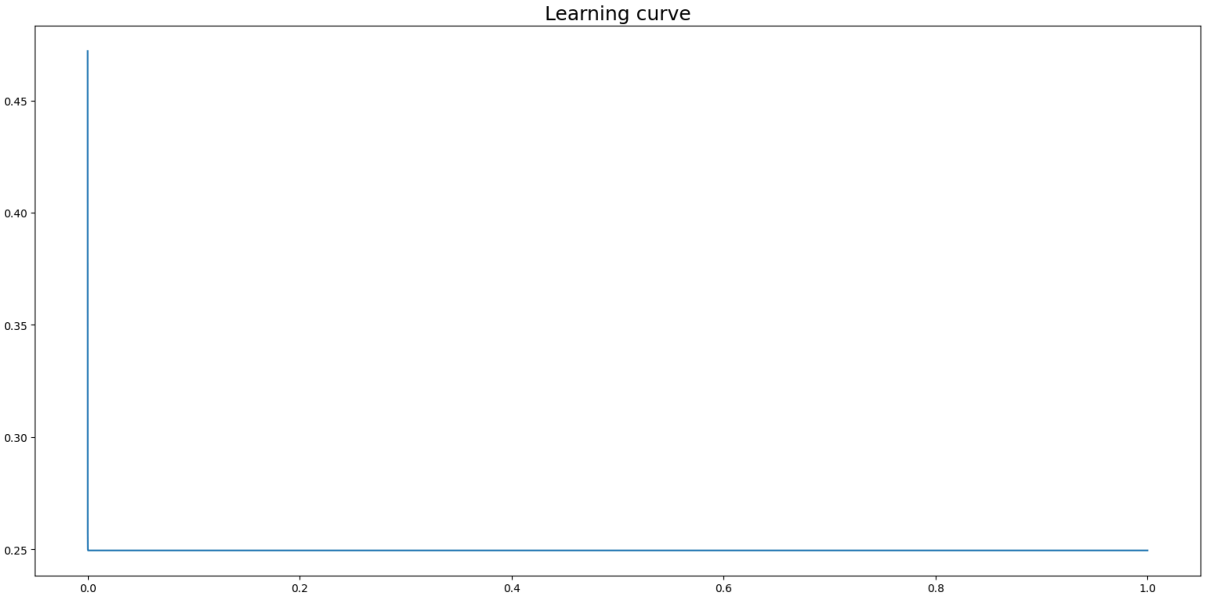
Predict correct



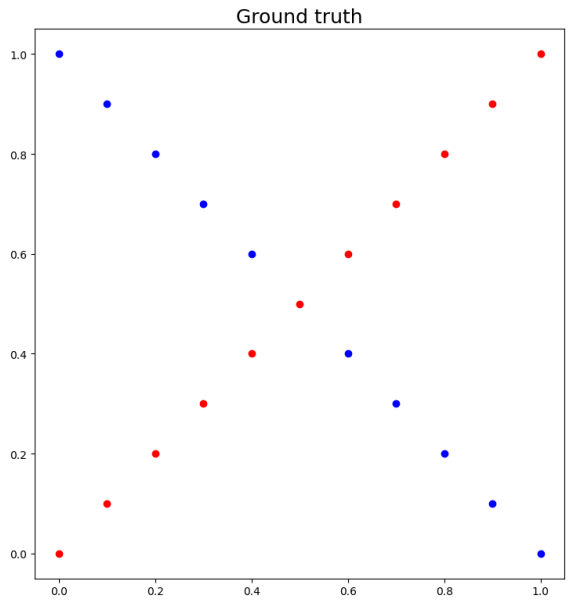


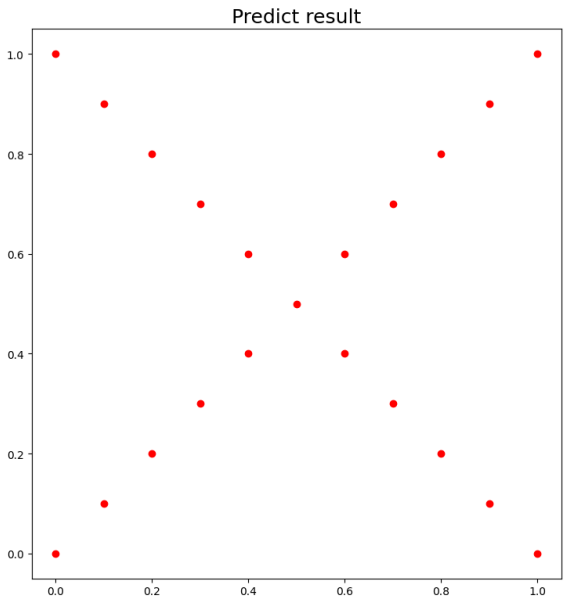
XOR data

Can not converge



Predict wrong





D. Anything you want to share

The loss of training without activation function in XOR data is stuck at 0.2494, and I do not know why.

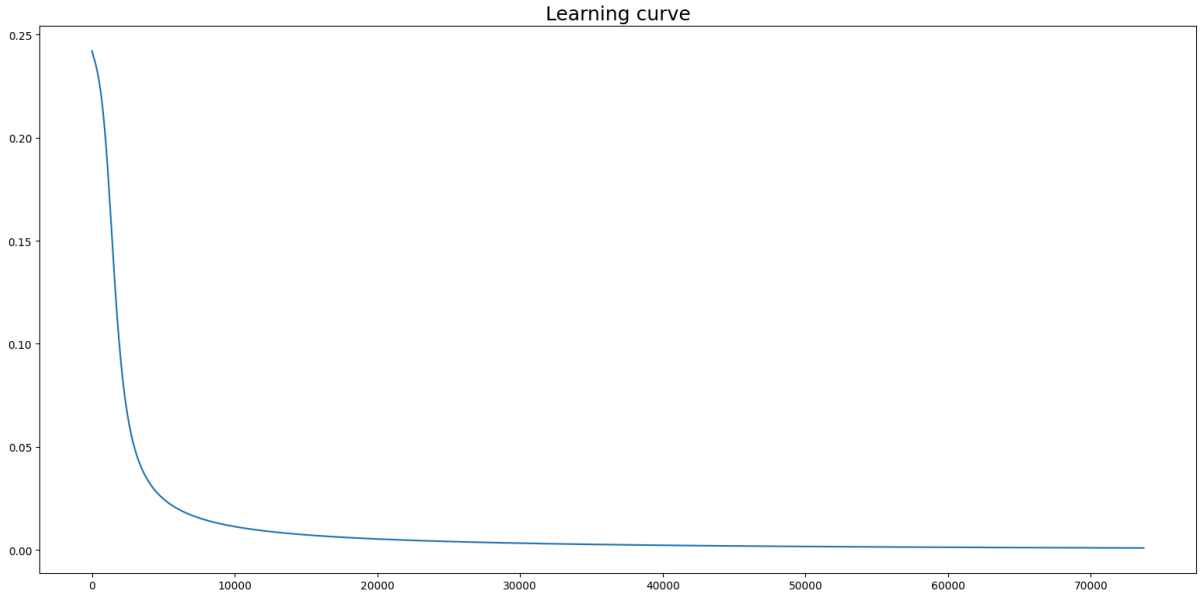
5. Extra (10%)

A. Implement different optimizers. (2%)

```python
def update(self) -> None:
    # Update weights
    gradient = np.matmul(self.forward_gradient.T, self.backward_gradient)

    # Optimizer
    if self.optimizer == 'momentum':
        self.momentum = 0.9 * self.momentum - self.learning_rate * gradient
        delta_weight = self.momentum
    elif self.optimizer == 'adagrad':
        self.sum_of_squares_of_gradients += np.square(gradient)
        delta_weight = -self.learning_rate * gradient / \
            np.sqrt(self.sum_of_squares_of_gradients + 1e-8)
    elif self.optimizer == 'adam':
        self.moving_average_m = 0.9 * self.moving_average_m + 0.1 * gradient
        self.moving_average_v = 0.999 * \
            self.moving_average_v + 0.001 * np.square(gradient)
        bias_correction_m = self.moving_average_m / \
            (1.0 - 0.9 ** self.update_times)
        bias_correction_v = self.moving_average_v / \
            (1.0 - 0.999 ** self.update_times)
        self.update_times += 1
        delta_weight = -self.learning_rate * bias_correction_m / \
            (np.sqrt(bias_correction_v) + 1e-8)
    else:
        # Without optimizer
        delta_weight = -self.learning_rate * gradient

    self.weight += delta_weight
```

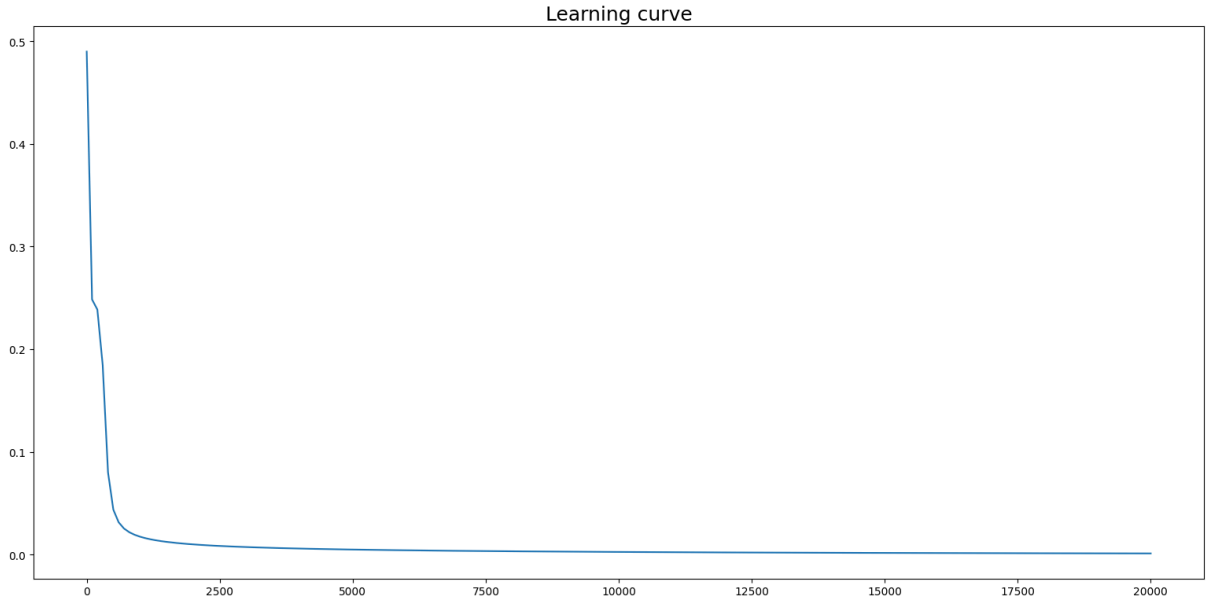
Without optimizer

Loss converges normally, about 70,000



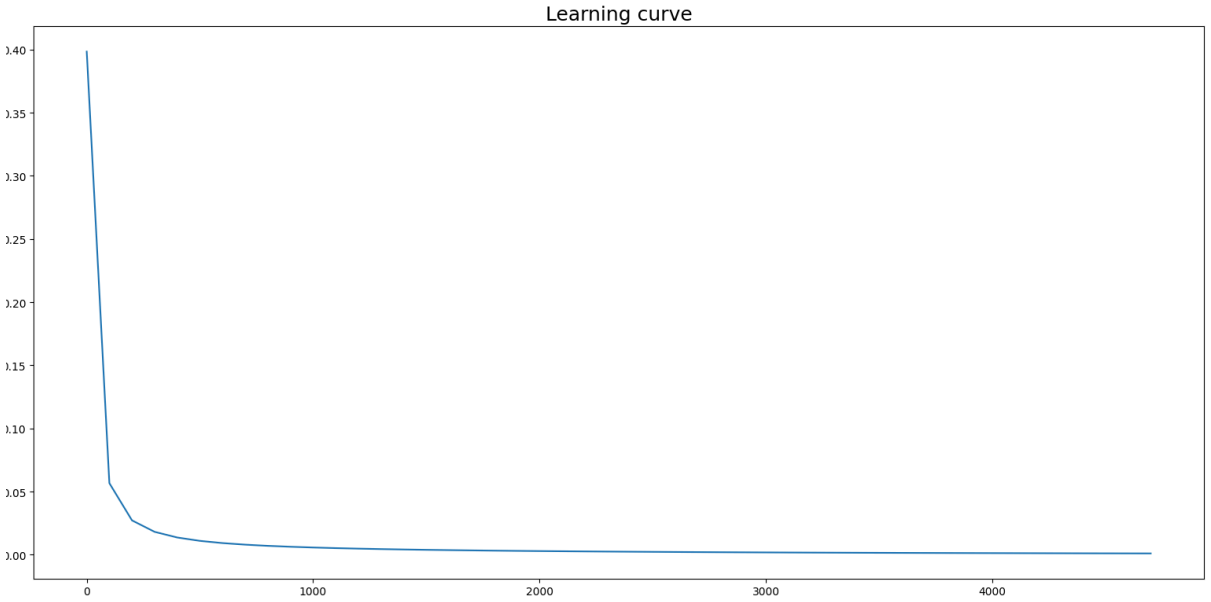
Momentum

Loss converges quickly, about 20,000



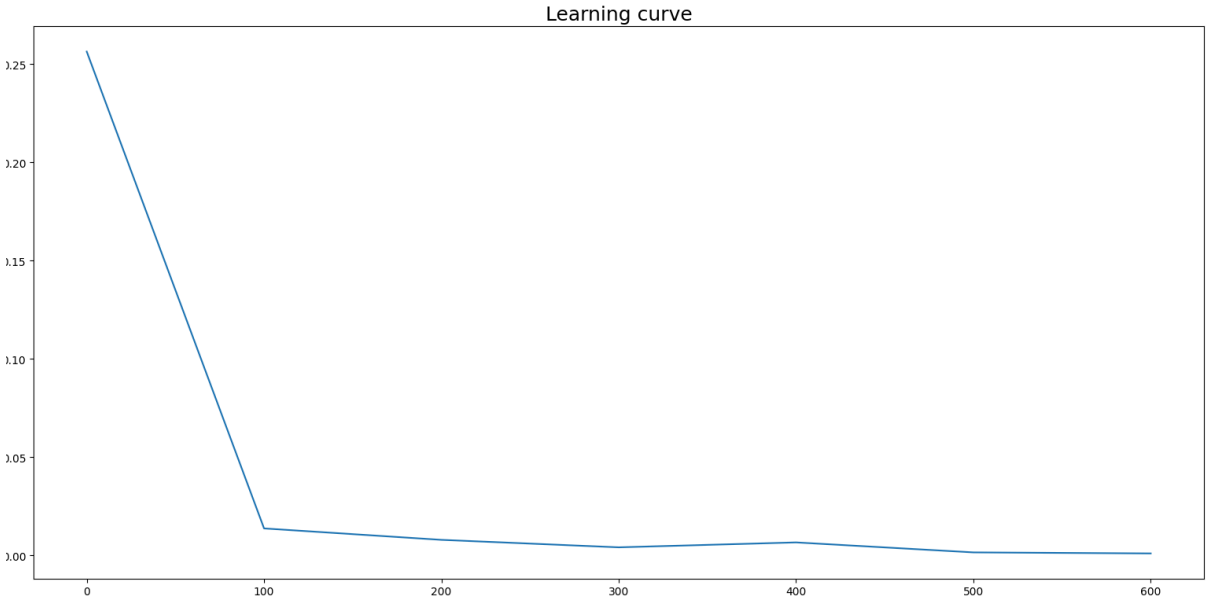
AdaGrad

Loss converges more quickly, about 5,000



Adam

Loss converges most quickly, about 600

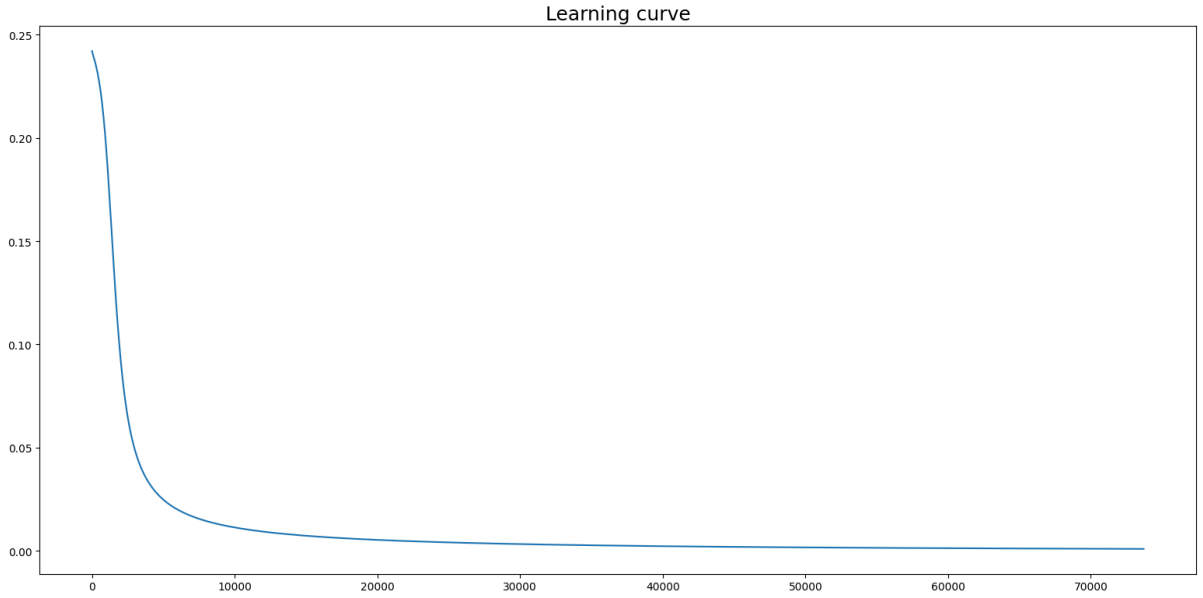


B. Implement different activation functions. (3%)

Sigmoid

```python
@staticmethod
def sigmoid(x: np.ndarray) -> np.ndarray:
    # Calculate sigmoid function
    # y = 1 / (1 + e^(-x))
    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)
    return np.multiply(y, 1.0 - y)
```

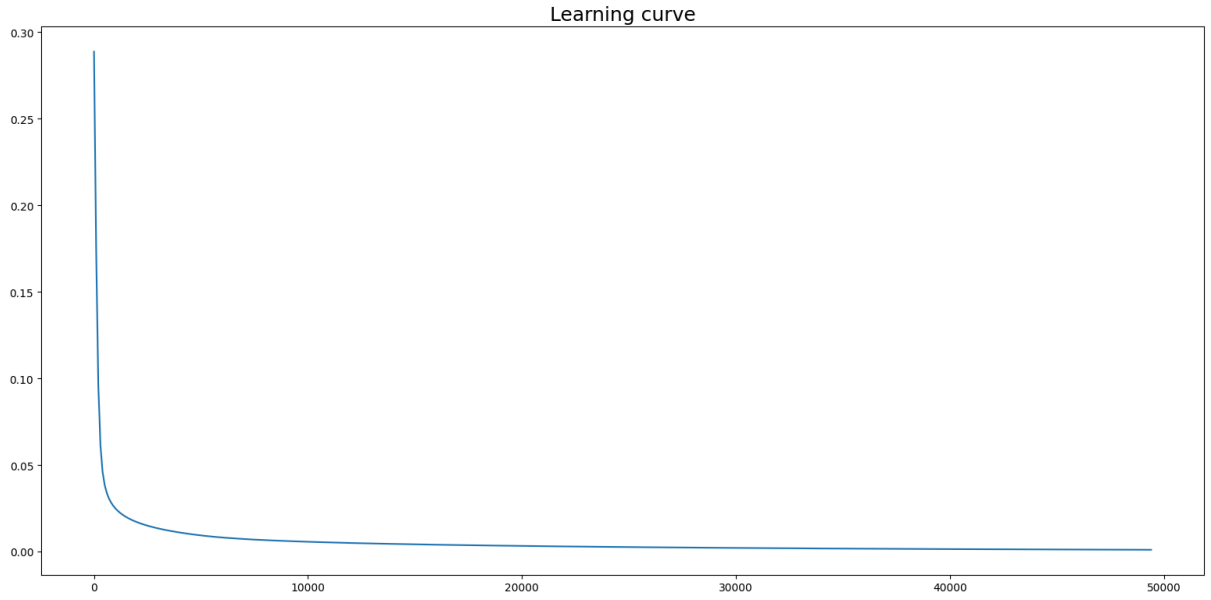
Loss converges normally, about 70,000



Tanh

```python
@staticmethod
def tanh(x: np.ndarray) -> np.ndarray:
    # Calculate tanh function
    # y = tanh(x)
    return np.tanh(x)

@staticmethod
def derivative_tanh(y: np.ndarray) -> np.ndarray:
    # Calculate the derivative of tanh function
    # y' = 1 - y^2
    return 1.0 - y ** 2
```

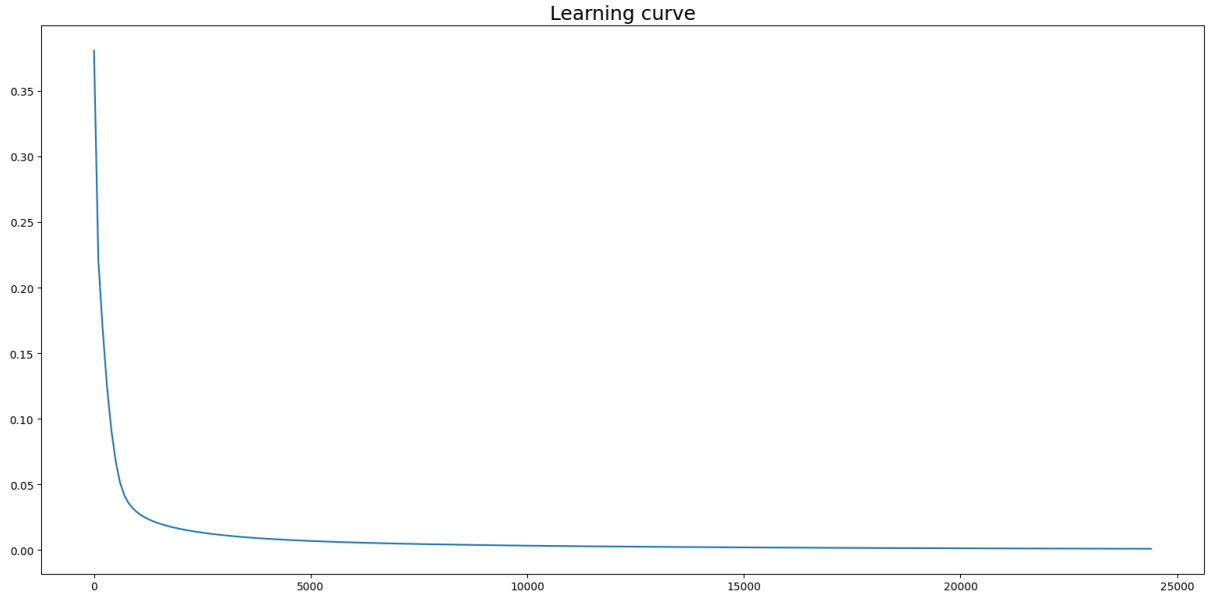
Loss converges a little quickly, about 50,000



ReLU

```python
@staticmethod
def relu(x: np.ndarray) -> np.ndarray:
    # Calculate relu function
    # y = max(0, x)
    return np.maximum(0.0, x)

@staticmethod
def derivative_relu(y: np.ndarray) -> np.ndarray:
    # Calculate the derivative of relu function
    # y' = 1 if y > 0
    # y' = 0 if y <= 0
    return np.heaviside(y, 0.0)
```

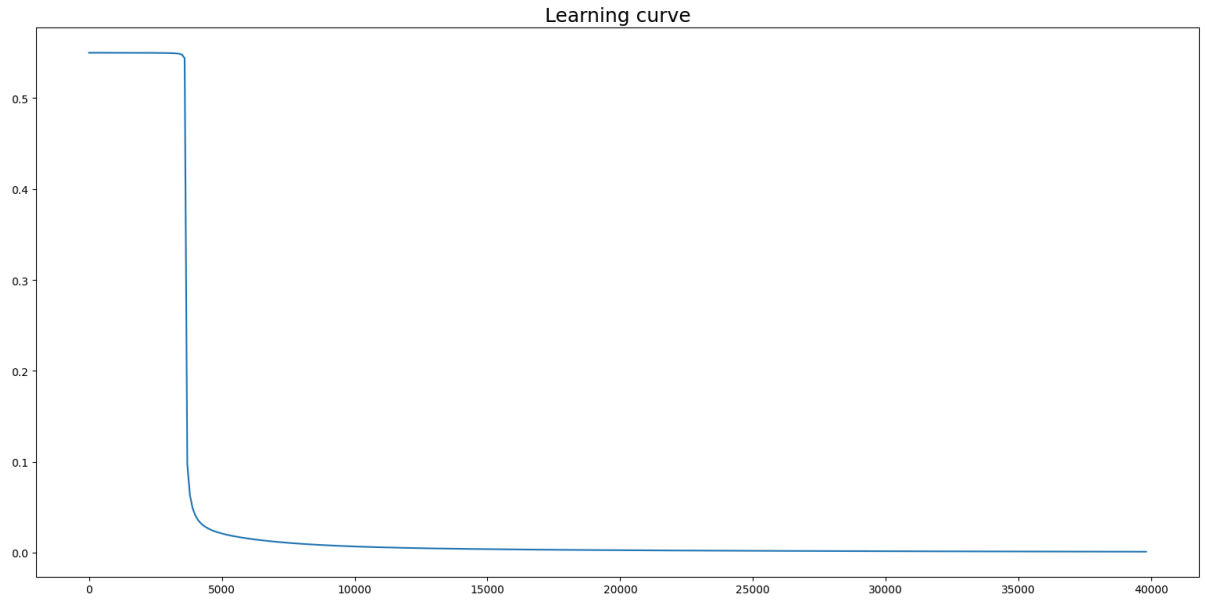
Loss converges more quickly, about 25,000



Leaky ReLU

```python
@staticmethod
def leaky_relu(x: np.ndarray) -> np.ndarray:
    # Calculate leaky relu function
    # y = max(0, x) + 0.01 * min(0, x)
    return np.maximum(0.0, x) + 0.01 * np.minimum(0.0, x)

@staticmethod
def derivative_leaky_relu(y: np.ndarray) -> np.ndarray:
    # Calculate the derivative of leaky relu function
    # y' = 1 if y > 0
    # y' = 0.01 if y <= 0
    y[y > 0.0] = 1.0
    y[y <= 0.0] = 0.01
    return y
```

Loss converges a little quickly, about 40,000



C. Implement convolutional layers. (5%)