Lab1: back-propagation

Author: 311553007 應耀德

1. Introduction

Lab Objective:

In this lab, you will need to understand and implement simple neural networks with forwarding pass and backpropagation using two hidden layers. Notice that you can only use NumPy and the python standard libraries, any other frameworks (ex : TensorFlow、PyTorch) are not allowed in this lab.

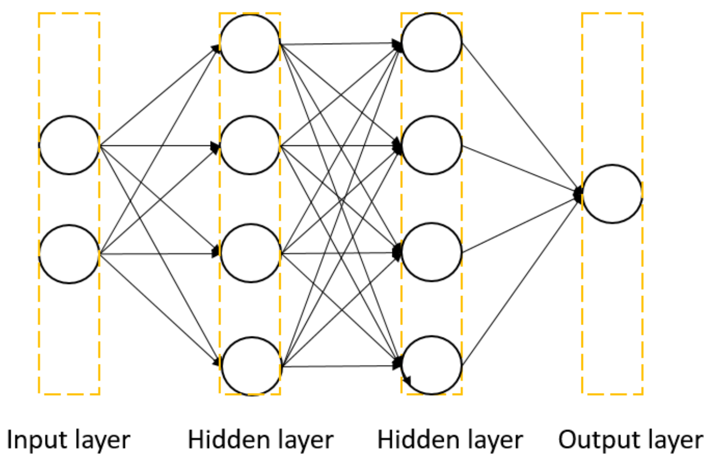


Figure 1. Two-layer neural network

Requirements:

1. Implement simple neural networks with two hidden layers.
2. You must use backpropagation in this neural network and can only use NumPy and other python standard libraries to implement.
3. Plot your comparison figure that show the predicted results and the ground-truth.

Data:

Use two types of data to test the neural network.

- Uniform distribution (blue points are labeled to 1, red points are labeled to 0)

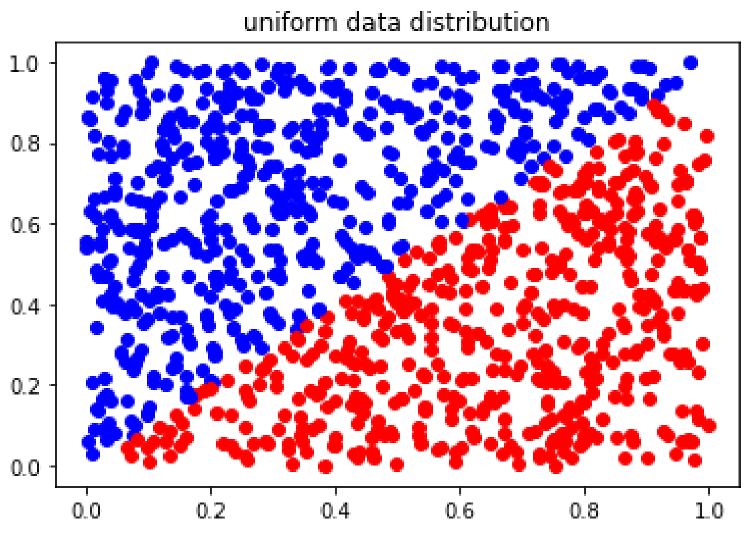


Figure 2. Data points generated from uniform distribution

- XOR (X shape) (blue points are labeled to 1, red points are labeled to 0)

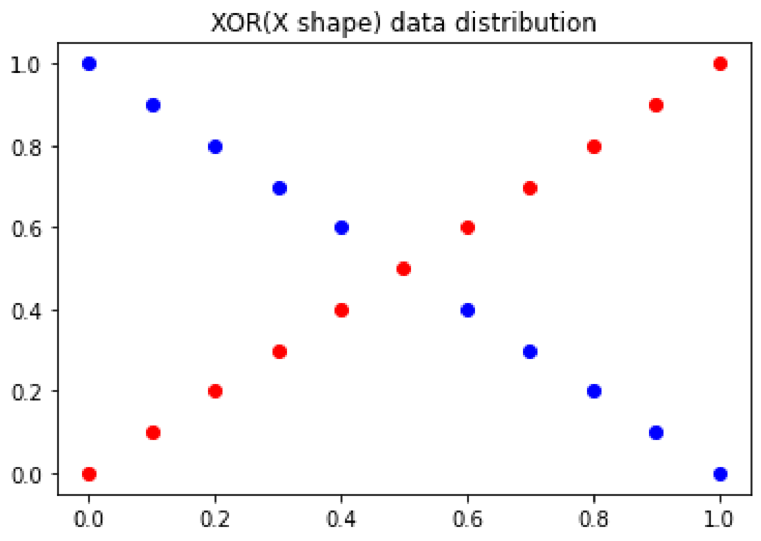
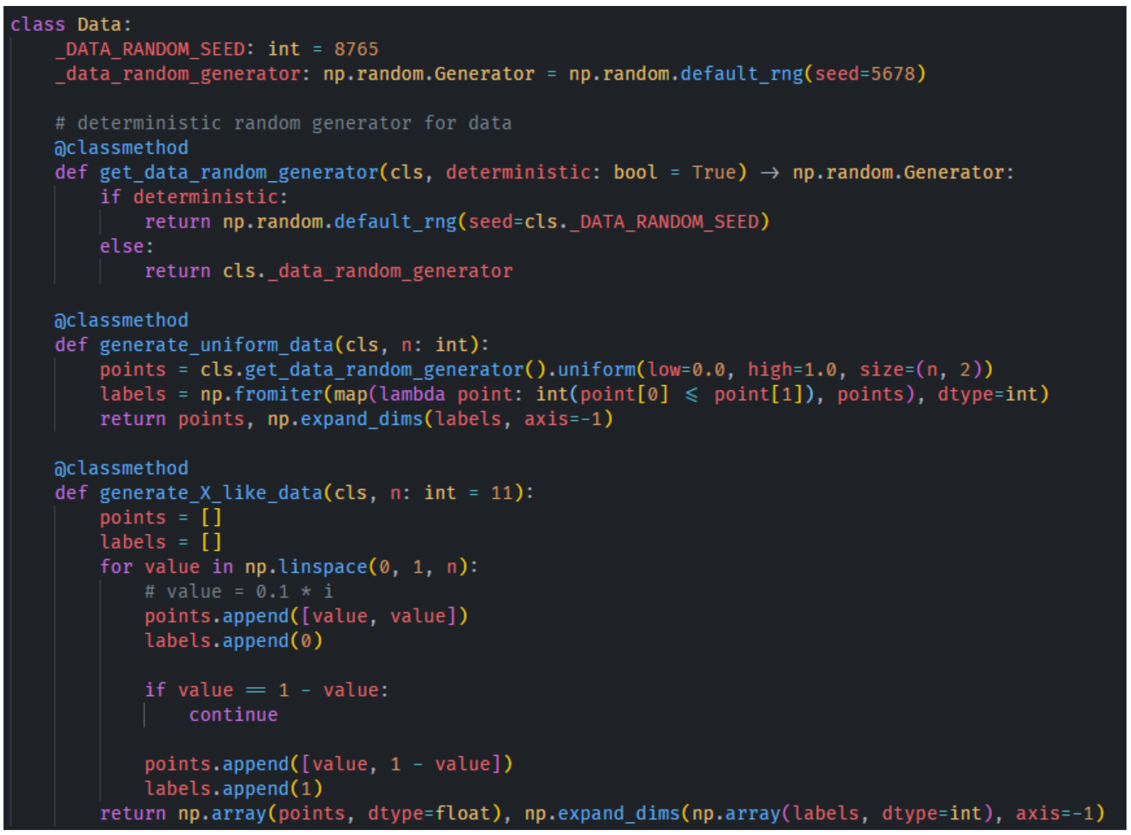


Figure 3. Data points generated from XOR

- Code for data generation

  To make the data generation deterministic, specify the seed of the random generator and recreate the random generator for every generation.

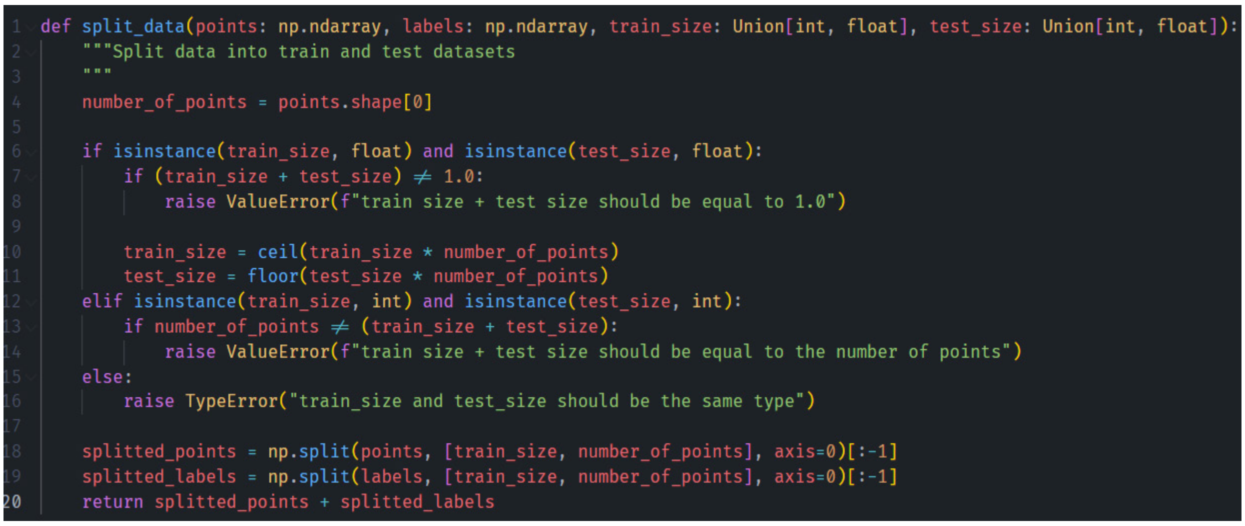


```
class Data:
    _DATA_RANDOM_SEED: int = 8765
    _data_random_generator: np.random.Generator = np.random.default_rng(seed=5678)

    # deterministic random generator for data
    @classmethod
    def get_data_random_generator(cls, deterministic: bool = True) -> np.random.Generator:
        if deterministic:
            return np.random.default_rng(seed=cls._DATA_RANDOM_SEED)
        else:
            return cls._data_random_generator

    @classmethod
    def generate_uniform_data(cls, n: int):
        points = cls.get_data_random_generator().uniform(low=0.0, high=1.0, size=(n, 2))
        labels = np.fromiter(map(lambda point: int(point[0] <= point[1]), points), dtype=int)
        return points, np.expand_dims(labels, axis=-1)

    @classmethod
    def generate_X_like_data(cls, n: int = 11):
        points = []
        labels = []
        for value in np.linspace(0, 1, n):
            # value = 0.1 * i
            points.append([value, value])
            labels.append(0)

            if value == 1 - value:
                continue

            points.append([value, 1 - value])
            labels.append(1)
        return np.array(points, dtype=float), np.expand_dims(np.array(labels, dtype=int), axis=-1)
```

Figure 4. Code for data generation

Data Setups:

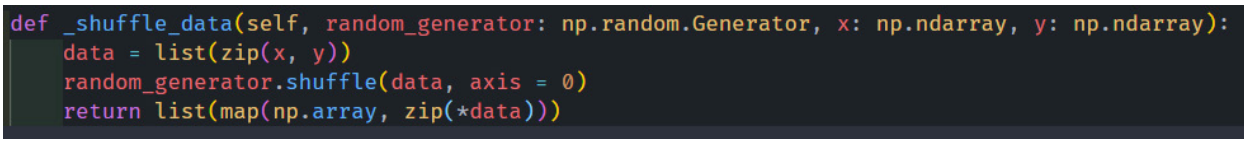
- Code for splitting data (output format is x_train, x_test, y_train, y_test)



```python
def split_data(points: np.ndarray, labels: np.ndarray, train_size: Union[int, float], test_size: Union[int, float]):
    """Split data into train and test datasets
    """
    number_of_points = points.shape[0]

    if isinstance(train_size, float) and isinstance(test_size, float):
        if (train_size + test_size) ≠ 1.0:
            raise ValueError(f"train size + test size should be equal to 1.0")

        train_size = ceil(train_size * number_of_points)
        test_size = floor(test_size * number_of_points)
    elif isinstance(train_size, int) and isinstance(test_size, int):
        if number_of_points ≠ (train_size + test_size):
            raise ValueError(f"train size + test size should be equal to the number of points")
    else:
        raise TypeError("train_size and test_size should be the same type")

    splitted_points = np.split(points, [train_size, number_of_points], axis=0)[:-1]
    splitted_labels = np.split(labels, [train_size, number_of_points], axis=0)[:-1]
    return splitted_points + splitted_labels
```

Figure 5. Code for splitting data

- Code for shuffling data (output format is x, y)



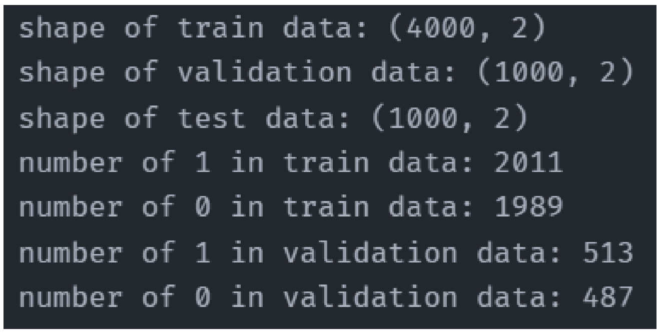
```python
def _shuffle_data(self, random_generator: np.random.Generator, x: np.ndarray, y: np.ndarray):
    data = list(zip(x, y))
    random_generator.shuffle(data, axis = 0)
    return list(map(np.array, zip(*data)))
```

Figure 6. Code for shuffling data

Data Preprocessing:

- Uniform data

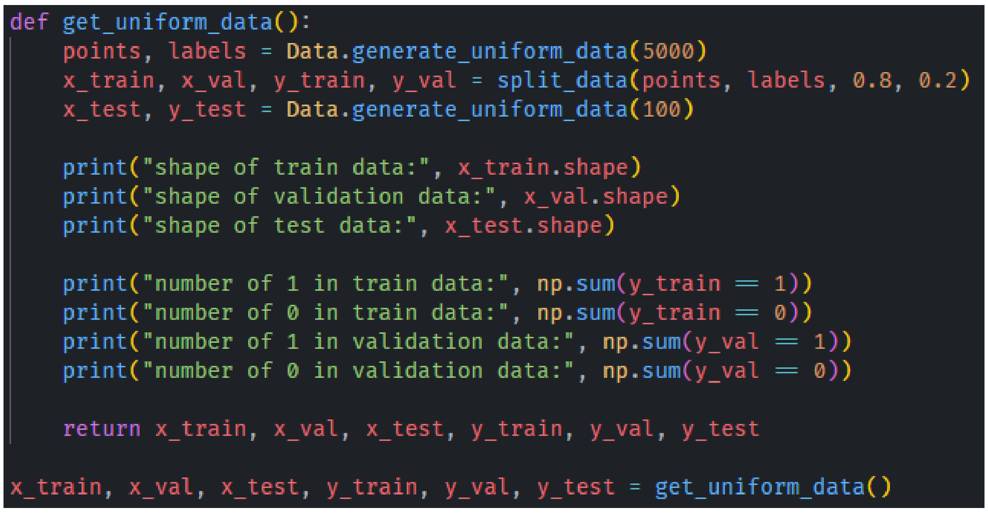
1. Generate 5000 data points for the training dataset.
    a. Split the training dataset into the training set and validation set by 0.8 and 0.2.
    b. Shuffle the training set before starting training for each epoch.



```
shape of train data: (4000, 2)
shape of validation data: (1000, 2)
shape of test data: (1000, 2)
number of 1 in train data: 2011
number of 0 in train data: 1989
number of 1 in validation data: 513
number of 0 in validation data: 487
```

Figure 7. Label distributions of the training and validation set

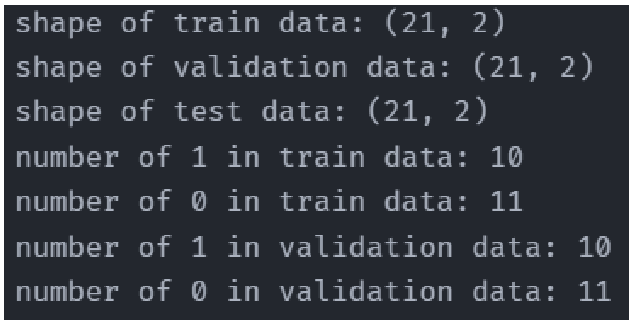
2. Generate 1000 data points for the testing dataset.



```python
def get_uniform_data():
    points, labels = Data.generate_uniform_data(5000)
    x_train, x_val, y_train, y_val = split_data(points, labels, 0.8, 0.2)
    x_test, y_test = Data.generate_uniform_data(100)

    print("shape of train data:", x_train.shape)
    print("shape of validation data:", x_val.shape)
    print("shape of test data:", x_test.shape)

    print("number of 1 in train data:", np.sum(y_train == 1))
    print("number of 0 in train data:", np.sum(y_train == 0))
    print("number of 1 in validation data:", np.sum(y_val == 1))
    print("number of 0 in validation data:", np.sum(y_val == 0))

    return x_train, x_val, x_test, y_train, y_val, y_test

x_train, x_val, x_test, y_train, y_val, y_test = get_uniform_data()
```

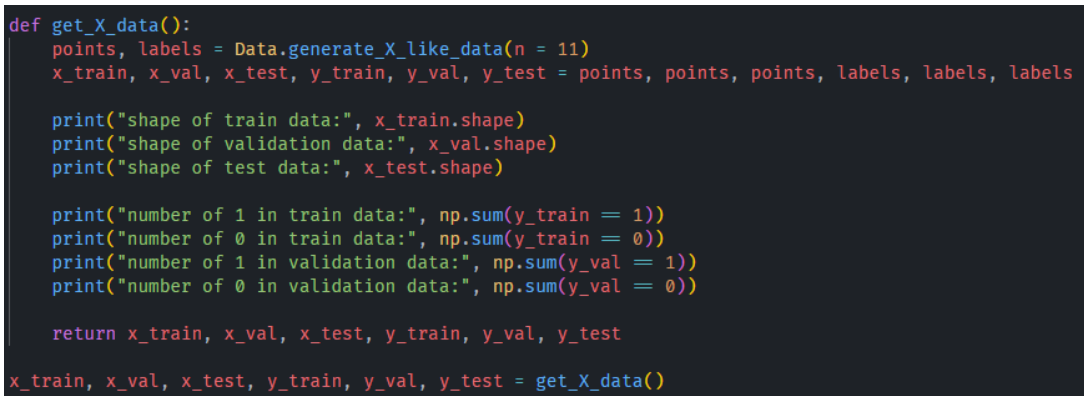
Figure 8. Code for preprocessing uniform data

- XOR data
  1. Generate the same data for the training, validation, and testing dataset.
  2. Shuffle the training set before starting training for each epoch.



```
shape of train data: (21, 2)
shape of validation data: (21, 2)
shape of test data: (21, 2)
number of 1 in train data: 10
number of 0 in train data: 11
number of 1 in validation data: 10
number of 0 in validation data: 11
```

Figure 9. Label distributions



```python
def get_X_data():
    points, labels = Data.generate_X_like_data(n = 11)
    x_train, x_val, x_test, y_train, y_val, y_test = points, points, points, labels, labels, labels

    print("shape of train data:", x_train.shape)
    print("shape of validation data:", x_val.shape)
    print("shape of test data:", x_test.shape)

    print("number of 1 in train data:", np.sum(y_train == 1))
    print("number of 0 in train data:", np.sum(y_train == 0))
    print("number of 1 in validation data:", np.sum(y_val == 1))
    print("number of 0 in validation data:", np.sum(y_val == 0))

    return x_train, x_val, x_test, y_train, y_val, y_test

x_train, x_val, x_test, y_train, y_val, y_test = get_X_data()
```

Figure 10. Code for preprocessing XOR data

Flowchart for training a model:

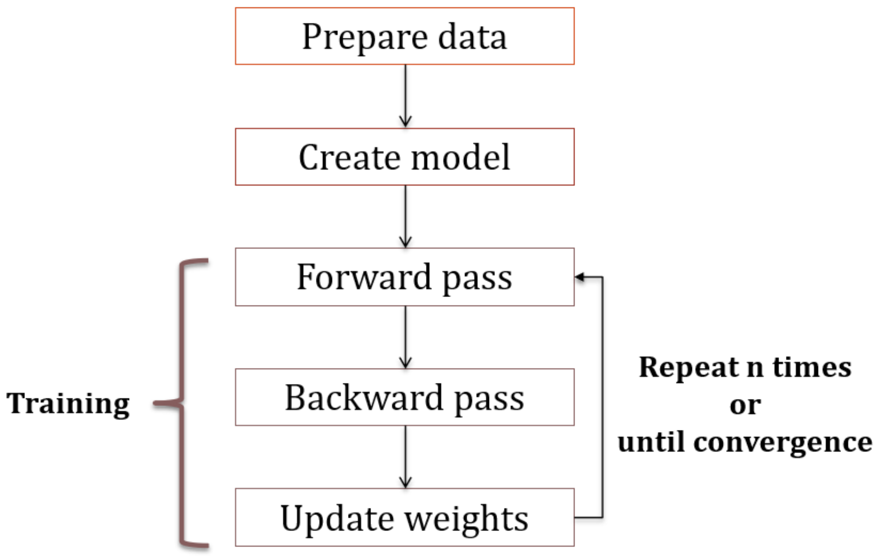


Figure 11. Flowchart for training a model

Perceptron:

A basic computation unit for neural network.

$$y = \mathbf{w}^\mathrm{T}\mathbf{x} + b$$

Figure 12. Equation of linear transformation with bias

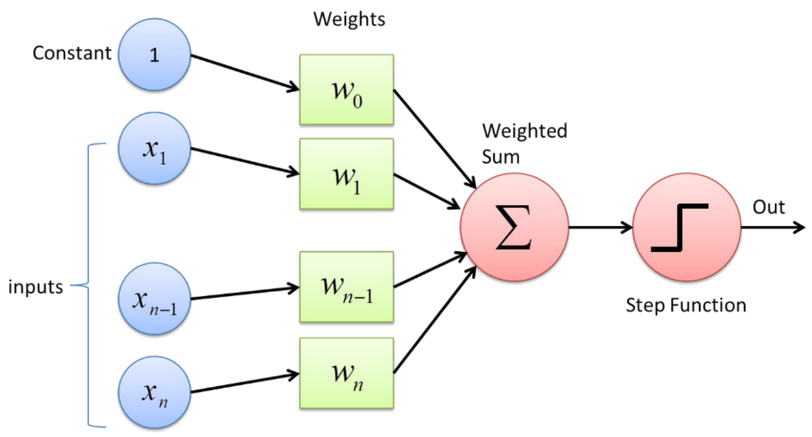


Figure 13. Details of calculating linear transformation

Neural Network (Multilayer perceptron MLP):

For each hidden layer and output layer, it contains at least one unit (perceptron).

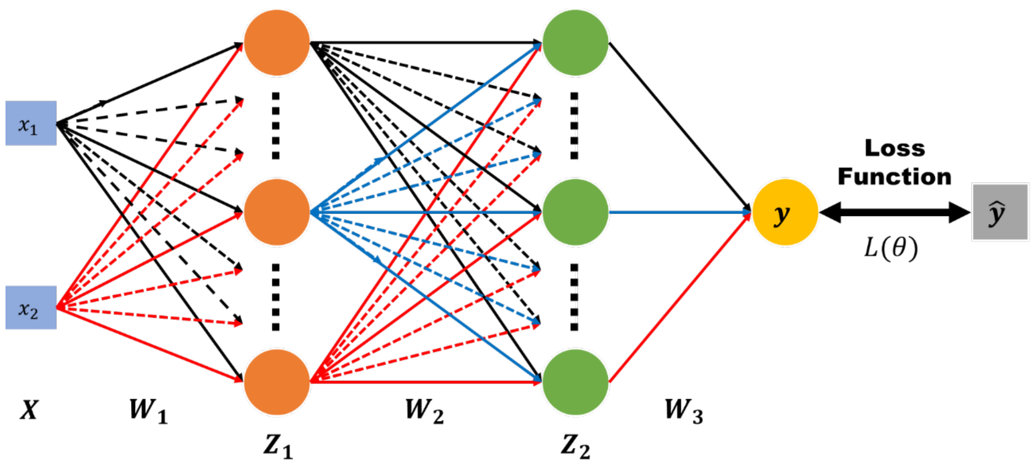


Figure 14. Architecture of Neural Network

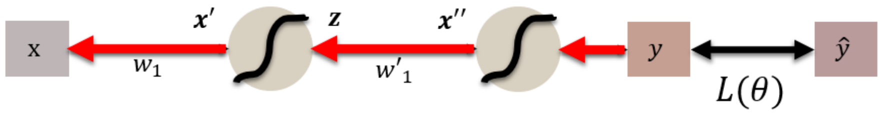
$\boldsymbol{X} : [x_1, x_2]$ $\quad \boldsymbol{y} : \textit{outputs}$ $\quad \hat{\boldsymbol{y}} : \textit{ground truth}$

$\boldsymbol{W_1, W_2, W_3} :$ *weight matrix of network layers*

$$Z_1 = \sigma(XW_1) \quad Z_2 = \sigma(Z_1W_2) \quad y = \sigma(Z_2W_3)$$

$$\sigma(\mathrm{x}) = \frac{1}{1+e^{-x}} \text{ (Sigmoid)}$$

Backpropagation:



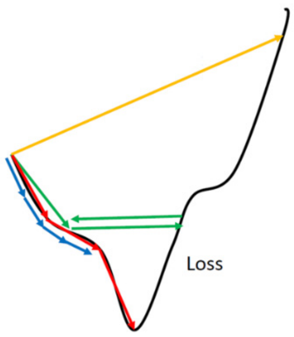
$x' = xw_1$ $\quad z = \sigma(x')$ $\quad x'' = zw'_1$ $\quad y = \sigma(x'')$

Chain rule

$y = g(x) \quad z = h(y)$

$$\mathbf{x} \xrightarrow{g()} y \xrightarrow{h()} z \qquad \frac{dz}{dx} = \frac{dz}{\cancel{dy}}\frac{\cancel{dy}}{dx}$$

$$\begin{aligned}
\frac{\partial L(\theta)}{\partial w_1} &= \frac{\partial y}{\partial w_1}\frac{\partial L(\theta)}{\partial y} \\
&= \frac{\partial x''}{\partial w_1}\frac{\partial y}{\partial x''}\frac{\partial L(\theta)}{\partial y} \\
&= \frac{\partial z}{\partial w_1}\frac{\partial x''}{\partial z}\frac{\partial y}{\partial x''}\frac{\partial L(\theta)}{\partial y} \\
&= \frac{\partial x'}{\partial w_1}\frac{\partial z}{\partial x'}\frac{\partial x''}{\partial z}\frac{\partial y}{\partial x''}\frac{\partial L(\theta)}{\partial y}
\end{aligned}$$



$$\theta^1 = \theta^0 - \rho \nabla L(\theta^0)$$
$$\theta^2 = \theta^1 - \rho \nabla L(\theta^1)$$
$$\theta^3 = \theta^2 - \rho \nabla L(\theta^2)$$

ρ : Learning rate

Network Parameters $\theta = \{w_1, w_2, w_3, w_4, \cdots\}$

Activation functions:

- Sigmoid

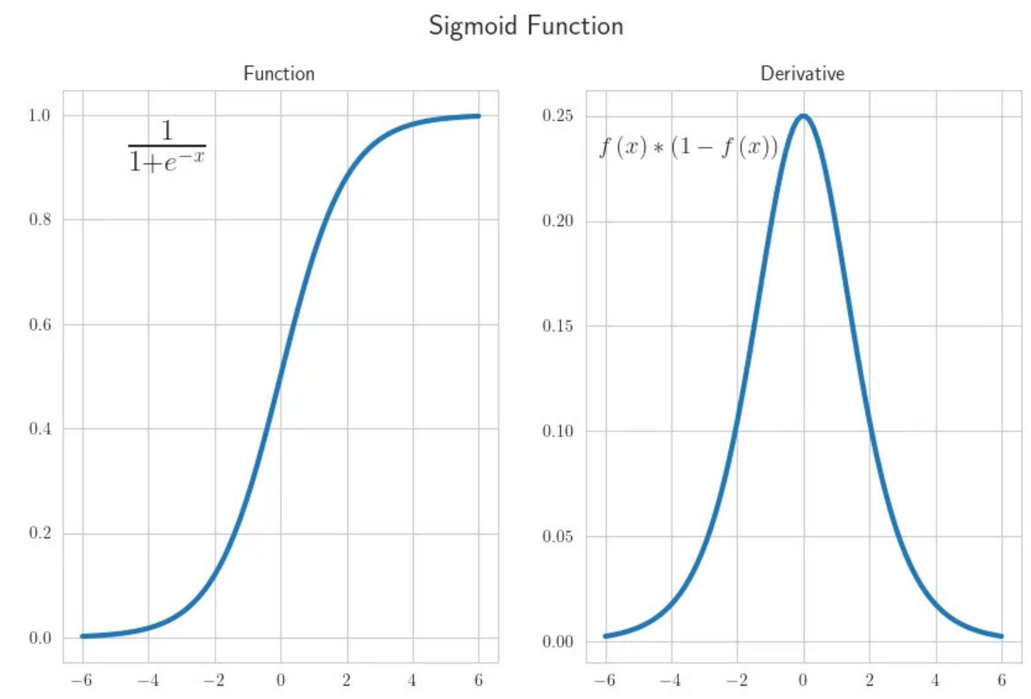


Figure 15. Visualization of Sigmoid function

- The function of the forward pass

$$f(x) = \frac{1}{1+e^{-x}}$$

- The function of the backward pass (derivative function)

$$f'(x) = f(x) * (1 - f(x))$$

- Tanh

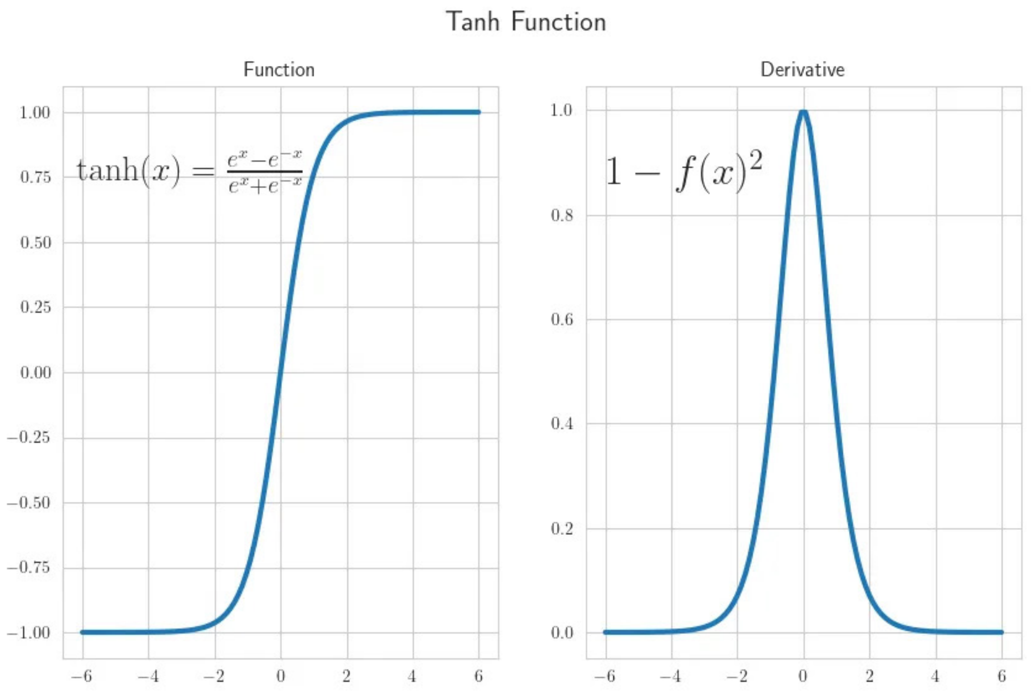


Figure 16. Visualization of Tanh function

- The function of the forward pass

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- The function of the backward pass (derivative function)

$$f'(x) = 1 - f(x)^2$$

- ReLU

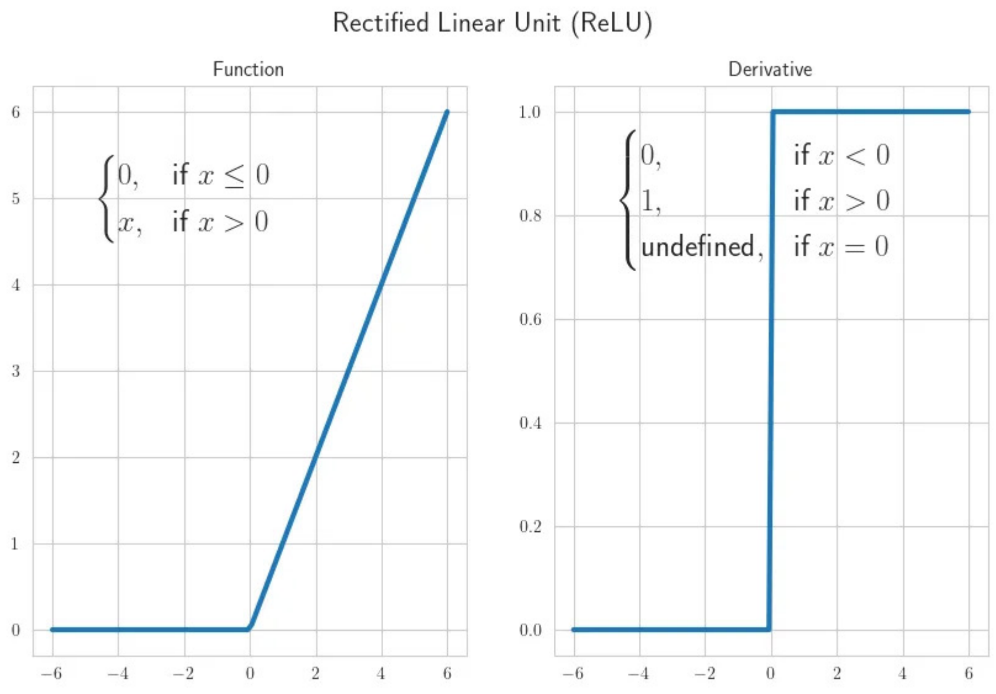


Figure 17. Visualization of ReLU function

- The function of the forward pass

$$\mathrm{ReLU}(x) = (x)^+ = \max(0, x)$$

$$f(x) = \begin{cases} 0, & \text{if } x \le 0 \\ x, & \text{if } x > 0 \end{cases}$$

- The function of the backward pass (derivative function)

  Note: In this lab, if x = 0, I define the value 0 instead of undefined.

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

- Leaky ReLU

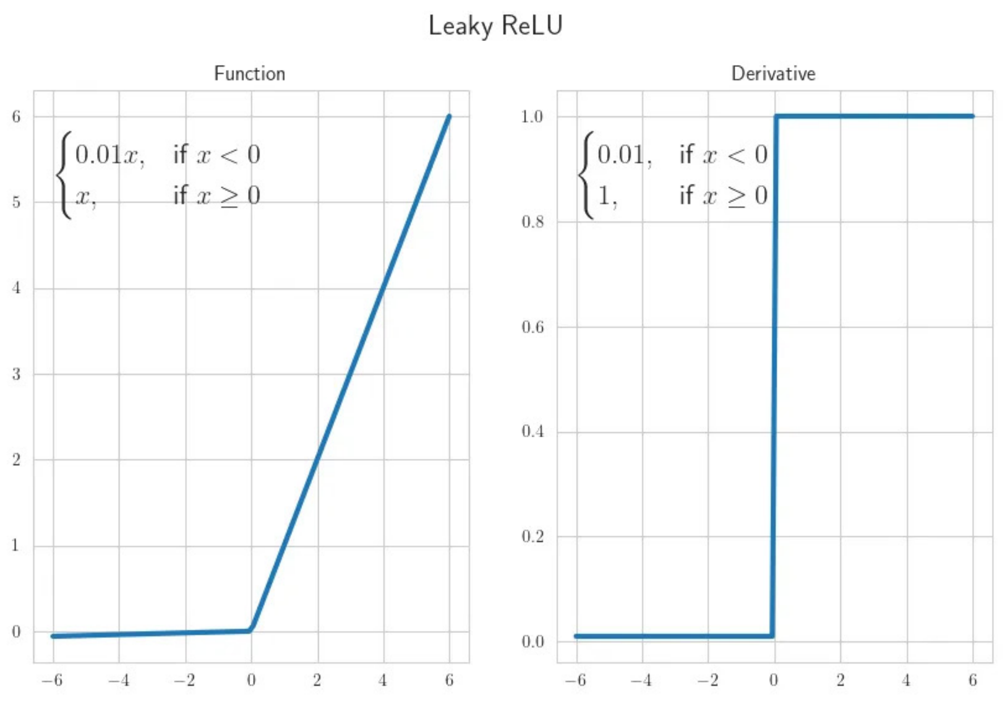


Figure 18. Visualization of Leaky ReLU function

- The function of the forward pass

$$f(x) = \begin{cases}0.01x, & \text{if } x < 0\\ x, & \text{if } x \geq 0\end{cases}$$

- The function of the backward pass (derivative function)

$$f'(x) = \begin{cases}0.01, & \text{if } x < 0\\ 1, & \text{if } x \geq 0\end{cases}$$

Loss functions:

- Mean Squared Error

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2.$$

Where

- n: *number of data points*
- i: *index of data points*
- $\hat{Y}$: *predict result*
- Y: *ground truth*

- Negative Log-Likelihood for Bernoulli Distribution

$$l(\theta) = -\textstyle\sum_{i=1}^{n}\left(y_i \log \hat{y}_{\theta,i} + (1 - y_i)\log(1 - \hat{y}_{\theta,i})\right)$$

Where

- n: $number\ of\ data\ points$
- i: $index\ of\ data\ points$
- θ: $weights$
- $\hat{y}$: $predict\ result$
- y: $ground\ truth$

Optimizers:

- Stochastic Gradient Descent (SGD)

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

Where

- W: $weights$
- L: $loss\ function$
- η: $learning\ rate$

- Momentum

$$V_t \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial W}$$

$$W \leftarrow W + V_t$$

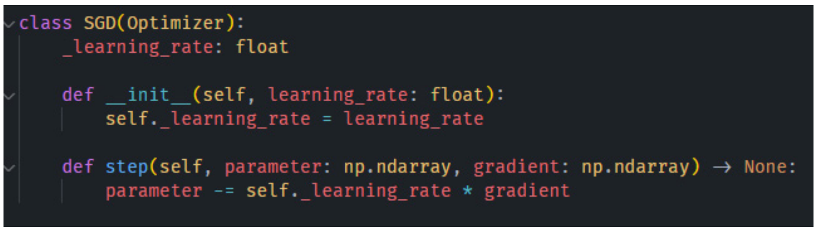
Where

- W: $weights$
- L: $loss\ function$
- η: $learning\ rate$
- β: $momentum$
- Vt: $velocity\ at\ time\ step\ t$

2. Experiment setups

Shared settings:

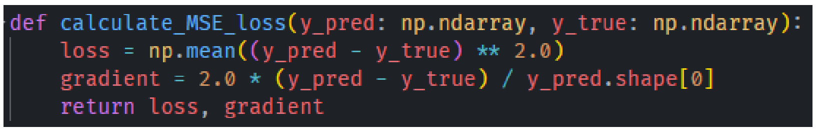
- **Optimizer – Stochastic Gradient Descent (SGD)**



```python
class SGD(Optimizer):
    _learning_rate: float

    def __init__(self, learning_rate: float):
        self._learning_rate = learning_rate

    def step(self, parameter: np.ndarray, gradient: np.ndarray) -> None:
        parameter -= self._learning_rate * gradient
```

Figure 19. Implementation of stochastic gradient descent (SGD)

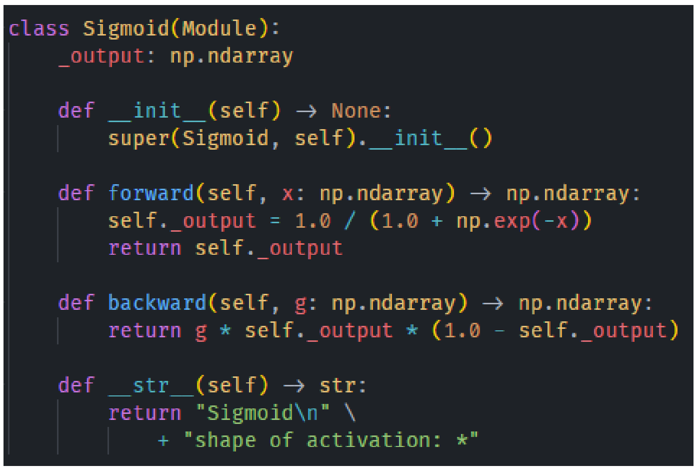
- **Loss function – Mean Squared Error**



```python
def calculate_MSE_loss(y_pred: np.ndarray, y_true: np.ndarray):
    loss = np.mean((y_pred - y_true) ** 2.0)
    gradient = 2.0 * (y_pred - y_true) / y_pred.shape[0]
    return loss, gradient
```

Figure 20. Implementation of mean squared error

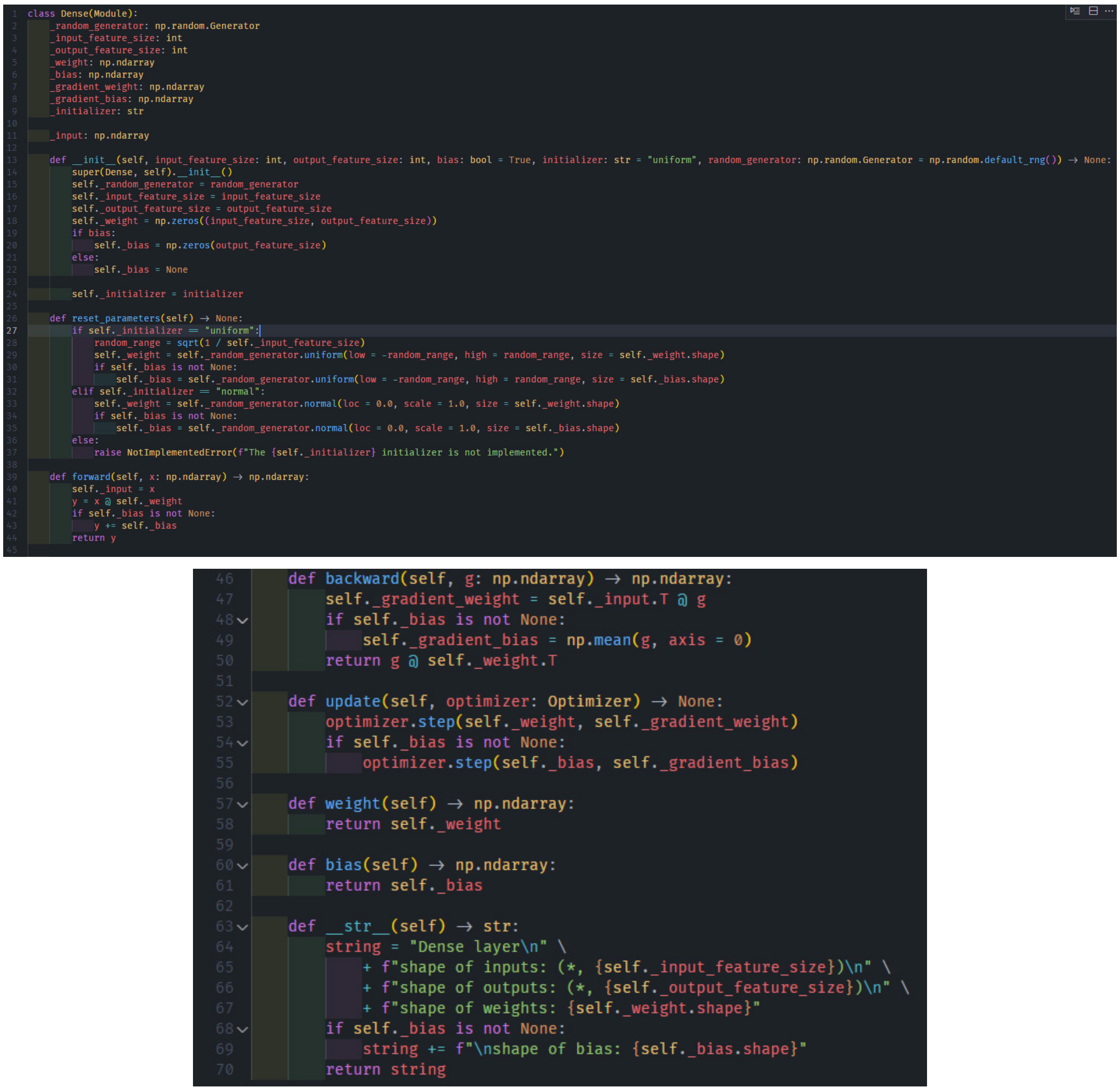
- **Output unit – Sigmoid**



```python
class Sigmoid(Module):
    _output: np.ndarray

    def __init__(self) -> None:
        super(Sigmoid, self).__init__()

    def forward(self, x: np.ndarray) -> np.ndarray:
        self._output = 1.0 / (1.0 + np.exp(-x))
        return self._output

    def backward(self, g: np.ndarray) -> np.ndarray:
        return g * self._output * (1.0 - self._output)

    def __str__(self) -> str:
        return "Sigmoid\n" \
            + "shape of activation: *"
```

Figure 21. Implementation of sigmoid function

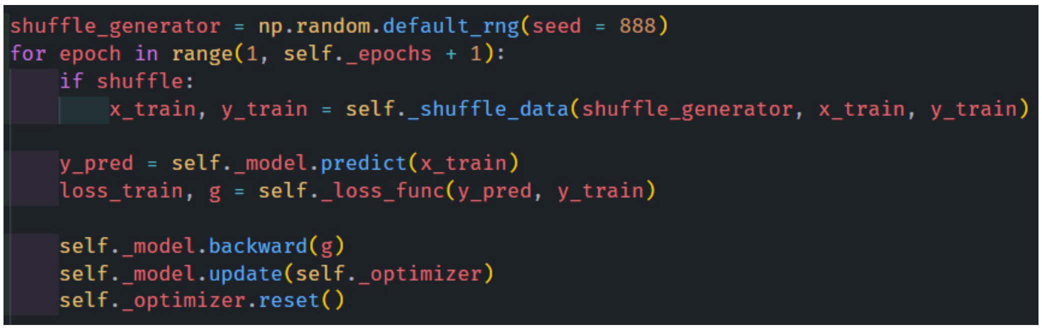
- **Dense layer (layers in neural network)**
  - forward: forward pass function
  - backward: backward pass function, calculate the gradients
  - update: step function, update parameters by optimizer



```python
class Dense(Module):
    _random_generator: np.random.Generator
    _input_feature_size: int
    _output_feature_size: int
    _weight: np.ndarray
    _bias: np.ndarray
    _gradient_weight: np.ndarray
    _gradient_bias: np.ndarray
    _initializer: str

    _input: np.ndarray

    def __init__(self, input_feature_size: int, output_feature_size: int, bias: bool = True, initializer: str = "uniform", random_generator: np.random.Generator = np.random.default_rng()) -> None:
        super(Dense, self).__init__()
        self._random_generator = random_generator
        self._input_feature_size = input_feature_size
        self._output_feature_size = output_feature_size
        self._weight = np.zeros((input_feature_size, output_feature_size))
        if bias:
            self._bias = np.zeros(output_feature_size)
        else:
            self._bias = None

        self._initializer = initializer

    def reset_parameters(self) -> None:
        if self._initializer == "uniform":
            random_range = sqrt(1 / self._input_feature_size)
            self._weight = self._random_generator.uniform(low = -random_range, high = random_range, size = self._weight.shape)
            if self._bias is not None:
                self._bias = self._random_generator.uniform(low = -random_range, high = random_range, size = self._bias.shape)
        elif self._initializer == "normal":
            self._weight = self._random_generator.normal(loc = 0.0, scale = 1.0, size = self._weight.shape)
            if self._bias is not None:
                self._bias = self._random_generator.normal(loc = 0.0, scale = 1.0, size = self._bias.shape)
        else:
            raise NotImplementedError(f"The {self._initializer} initializer is not implemented.")

    def forward(self, x: np.ndarray) -> np.ndarray:
        self._input = x
        y = x @ self._weight
        if self._bias is not None:
            y += self._bias
        return y

    def backward(self, g: np.ndarray) -> np.ndarray:
        self._gradient_weight = self._input.T @ g
        if self._bias is not None:
            self._gradient_bias = np.mean(g, axis = 0)
        return g @ self._weight.T

    def update(self, optimizer: Optimizer) -> None:
        optimizer.step(self._weight, self._gradient_weight)
        if self._bias is not None:
            optimizer.step(self._bias, self._gradient_bias)

    def weight(self) -> np.ndarray:
        return self._weight

    def bias(self) -> np.ndarray:
        return self._bias

    def __str__(self) -> str:
        string = "Dense layer\n" \
            + f"shape of inputs: (*, {self._input_feature_size})\n" \
            + f"shape of outputs: (*, {self._output_feature_size})\n" \
            + f"shape of weights: {self._weight.shape}"
        if self._bias is not None:
            string += f"\nshape of bias: {self._bias.shape}"
        return string
```

Figure 22. Implementation of dense layer

- **Backpropagation**

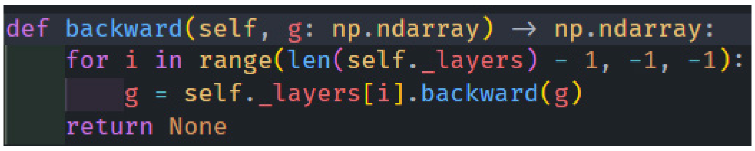


```python
shuffle_generator = np.random.default_rng(seed = 888)
for epoch in range(1, self._epochs + 1):
    if shuffle:
        x_train, y_train = self._shuffle_data(shuffle_generator, x_train, y_train)

    y_pred = self._model.predict(x_train)
    loss_train, g = self._loss_func(y_pred, y_train)

    self._model.backward(g)
    self._model.update(self._optimizer)
    self._optimizer.reset()
```

Figure 23. Implementation of training process

a. Calculate the gradient of loss function, *g*

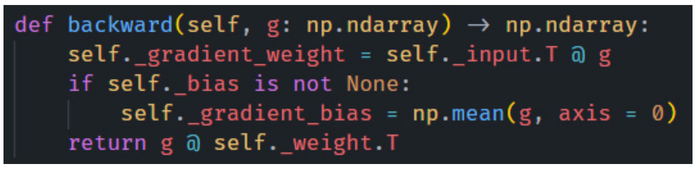
b. Propagate the gradient *g* to neural network (dense layers and activation functions)



```python
def backward(self, g: np.ndarray) → np.ndarray:
    for i in range(len(self._layers) - 1, -1, -1):
        g = self._layers[i].backward(g)
    return None
```

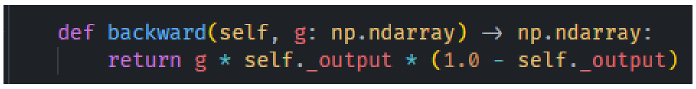
Figure 24. Propagate the gradient *g* to neural network

c. Calculate the gradients



```python
def backward(self, g: np.ndarray) → np.ndarray:
    self._gradient_weight = self._input.T @ g
    if self._bias is not None:
        self._gradient_bias = np.mean(g, axis = 0)
    return g @ self._weight.T
```

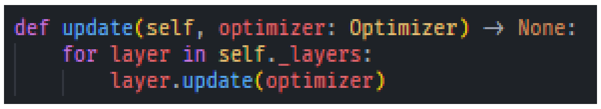
Figure 25. Calculate the gradients of dense layer



```python
def backward(self, g: np.ndarray) → np.ndarray:
    return g * self._output * (1.0 - self._output)
```

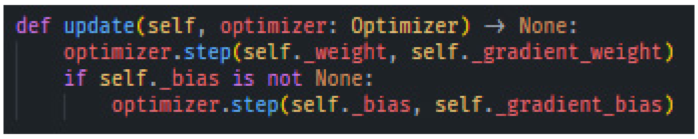
Figure 26. Calculate the gradients of sigmoid function

d. Update parameters



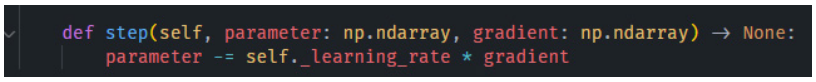
```python
def update(self, optimizer: Optimizer) → None:
    for layer in self._layers:
        layer.update(optimizer)
```

Figure 27. Propagate the optimizer to neural network



```python
def update(self, optimizer: Optimizer) → None:
    optimizer.step(self._weight, self._gradient_weight)
    if self._bias is not None:
        optimizer.step(self._bias, self._gradient_bias)
```

Figure 28. Update the parameters of dense layer



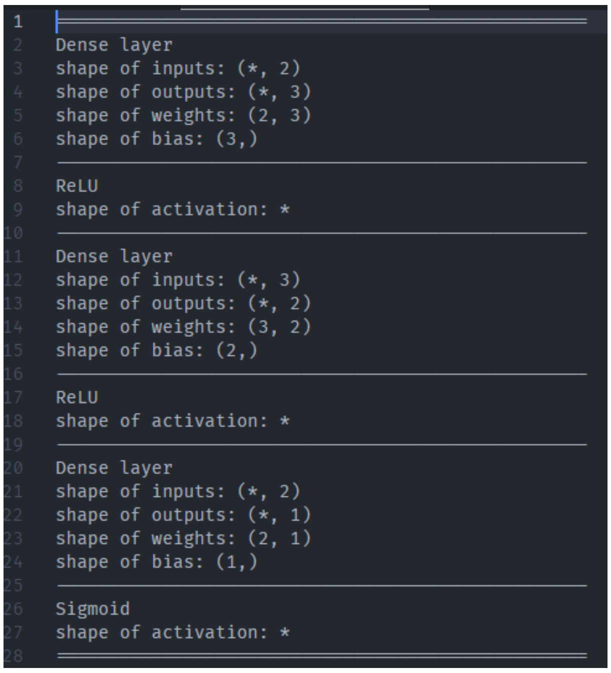
```python
def step(self, parameter: np.ndarray, gradient: np.ndarray) → None:
    parameter -= self._learning_rate * gradient
```

Figure 29. Implementation of step function of SGD optimizer

Uniform data:

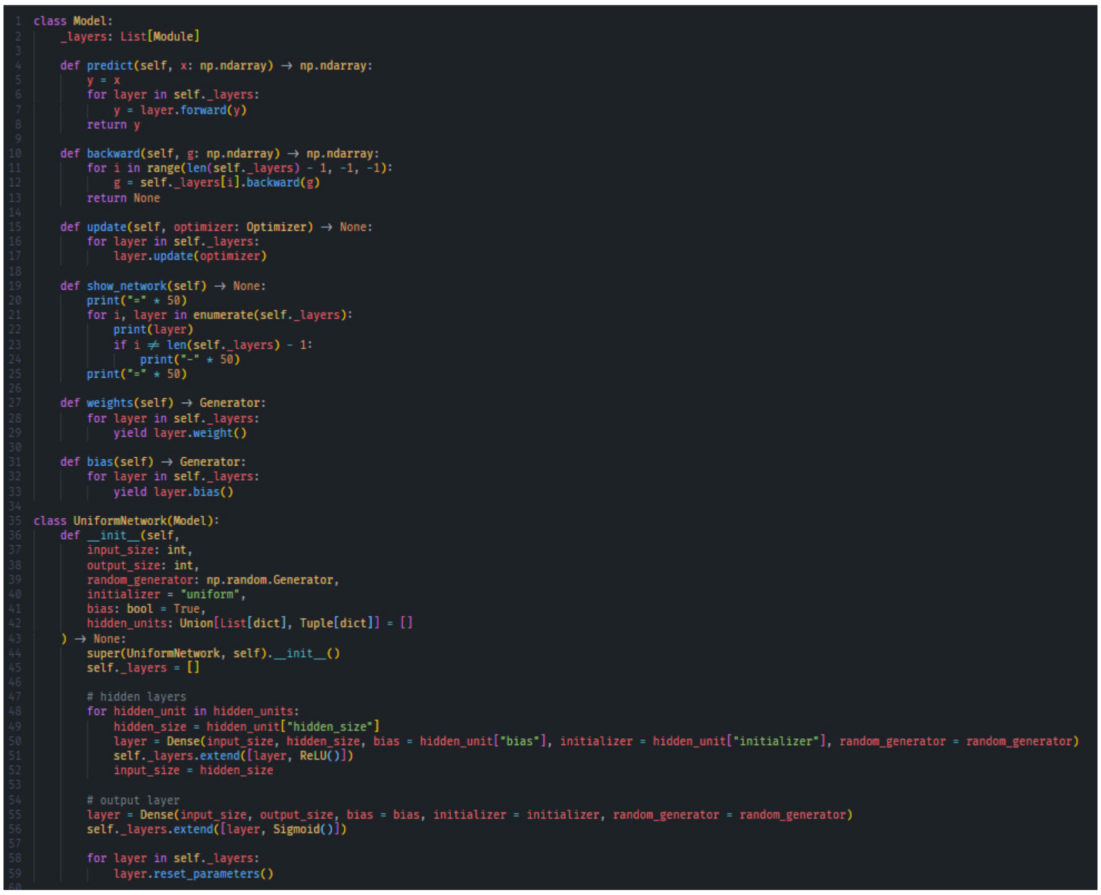
- **Neural network**
  - Input size: (*, 2)
  - Output size: (*, 1)
  - Initializer: normal distribution
  - Learning rate: 1.5

- Hidden layers: (2 x 3) with bias, (3 x 2) with bias
- Hidden units: ReLU
- Output layer: with bias



```
==================================================
Dense layer
shape of inputs: (*, 2)
shape of outputs: (*, 3)
shape of weights: (2, 3)
shape of bias: (3,)
--------------------------------------------------
ReLU
shape of activation: *
--------------------------------------------------
Dense layer
shape of inputs: (*, 3)
shape of outputs: (*, 2)
shape of weights: (3, 2)
shape of bias: (2,)
--------------------------------------------------
ReLU
shape of activation: *
--------------------------------------------------
Dense layer
shape of inputs: (*, 2)
shape of outputs: (*, 1)
shape of weights: (2, 1)
shape of bias: (1,)
--------------------------------------------------
Sigmoid
shape of activation: *
==================================================
```

Figure 30. Architecture of neural network for uniform data



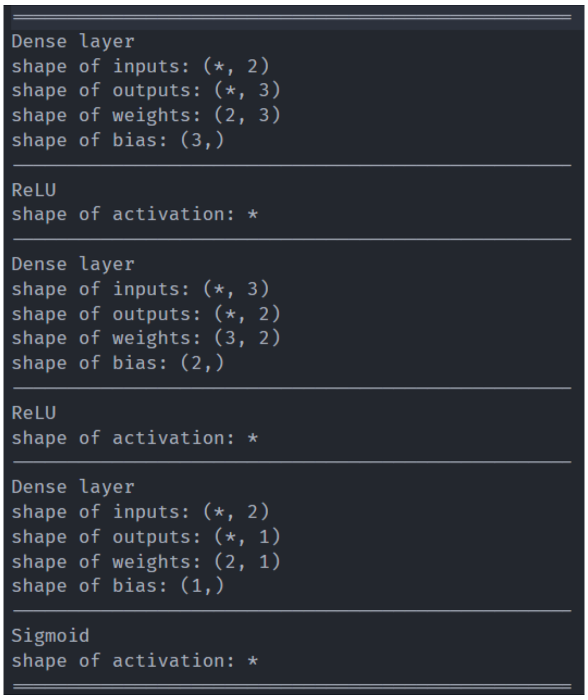
```python
class Model:
    _layers: List[Module]

    def predict(self, x: np.ndarray) → np.ndarray:
        y = x
        for layer in self._layers:
            y = layer.forward(y)
        return y

    def backward(self, g: np.ndarray) → np.ndarray:
        for i in range(len(self._layers) - 1, -1, -1):
            g = self._layers[i].backward(g)
        return None

    def update(self, optimizer: Optimizer) → None:
        for layer in self._layers:
            layer.update(optimizer)

    def show_network(self) → None:
        print("=" * 50)
        for i, layer in enumerate(self._layers):
            print(layer)
            if i ≠ len(self._layers) - 1:
                print("-" * 50)
        print("=" * 50)

    def weights(self) → Generator:
        for layer in self._layers:
            yield layer.weight()

    def bias(self) → Generator:
        for layer in self._layers:
            yield layer.bias()

class UniformNetwork(Model):
    def __init__(self,
        input_size: int,
        output_size: int,
        random_generator: np.random.Generator,
        initializer = "uniform",
        bias: bool = True,
        hidden_units: Union[List[dict], Tuple[dict]] = []
    ) → None:
        super(UniformNetwork, self).__init__()
        self._layers = []

        # hidden layers
        for hidden_unit in hidden_units:
            hidden_size = hidden_unit["hidden_size"]
            layer = Dense(input_size, hidden_size, bias = hidden_unit["bias"], initializer = hidden_unit["initializer"], random_generator = random_generator)
            self._layers.extend([layer, ReLU()])
            input_size = hidden_size

        # output layer
        layer = Dense(input_size, output_size, bias = bias, initializer = initializer, random_generator = random_generator)
        self._layers.extend([layer, Sigmoid()])

        for layer in self._layers:
            layer.reset_parameters()
```

Figure 31. Implementation of neural network for uniform data

XOR data:

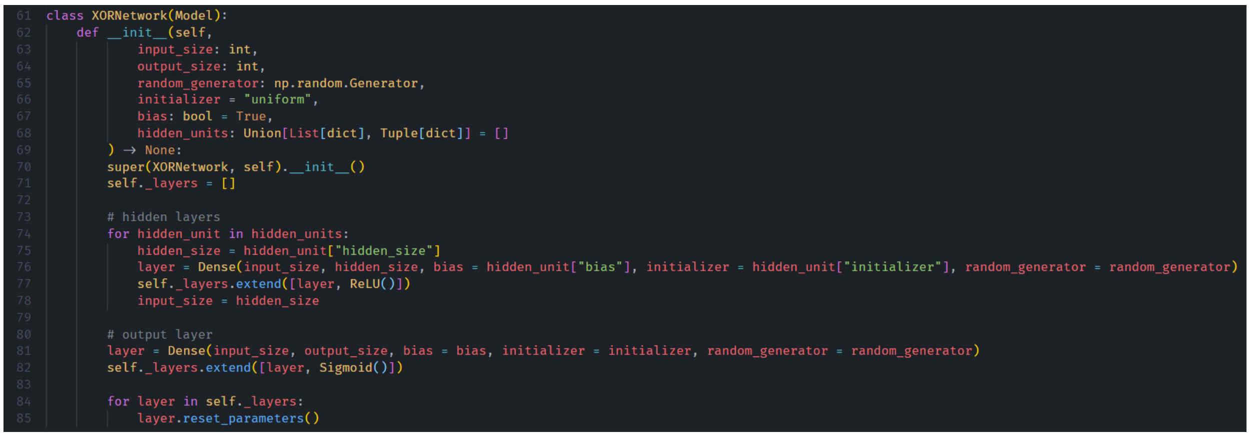
- **Neural network**
  - Input size: (*, 2)
  - Output size: (*, 1)

- Initializer: normal distribution
- Learning rate: 1.0
- Hidden layers: (2 x 3) with bias, (3 x 2) with bias
- Hidden units: ReLU
- Output layer: with bias



```
Dense layer
shape of inputs: (*, 2)
shape of outputs: (*, 3)
shape of weights: (2, 3)
shape of bias: (3,)

ReLU
shape of activation: *

Dense layer
shape of inputs: (*, 3)
shape of outputs: (*, 2)
shape of weights: (3, 2)
shape of bias: (2,)

ReLU
shape of activation: *

Dense layer
shape of inputs: (*, 2)
shape of outputs: (*, 1)
shape of weights: (2, 1)
shape of bias: (1,)

Sigmoid
shape of activation: *
```

Figure 32. Architecture of neural network for XOR data



```python
class XORNetwork(Model):
    def __init__(self,
            input_size: int,
            output_size: int,
            random_generator: np.random.Generator,
            initializer = "uniform",
            bias: bool = True,
            hidden_units: Union[List[dict], Tuple[dict]] = []
        ) → None:
        super(XORNetwork, self).__init__()
        self._layers = []

        # hidden layers
        for hidden_unit in hidden_units:
            hidden_size = hidden_unit["hidden_size"]
            layer = Dense(input_size, hidden_size, bias = hidden_unit["bias"], initializer = hidden_unit["initializer"], random_generator = random_generator)
            self._layers.extend([layer, ReLU()])
            input_size = hidden_size

        # output layer
        layer = Dense(input_size, output_size, bias = bias, initializer = initializer, random_generator = random_generator)
        self._layers.extend([layer, Sigmoid()])

        for layer in self._layers:
            layer.reset_parameters()
```

Figure 33. Implementation of neural network for XOR data

3. Results of your testing

Uniform data:

- **Learning curve (train loss, validation loss, epoch)**

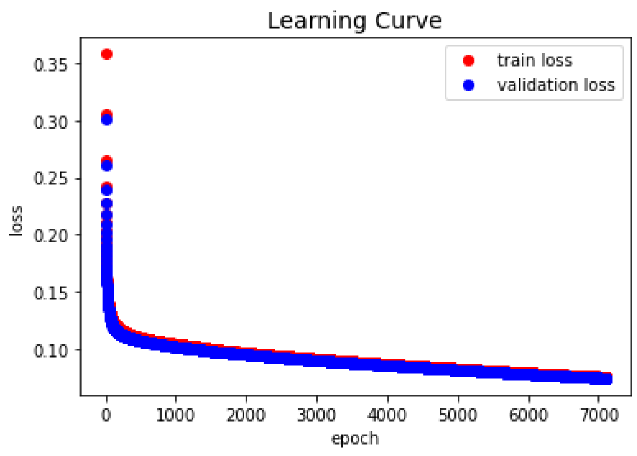


Figure 34. Learning curve of neural network for uniform data

- **Accuracy (train accuracy, validation accuracy, epoch)**

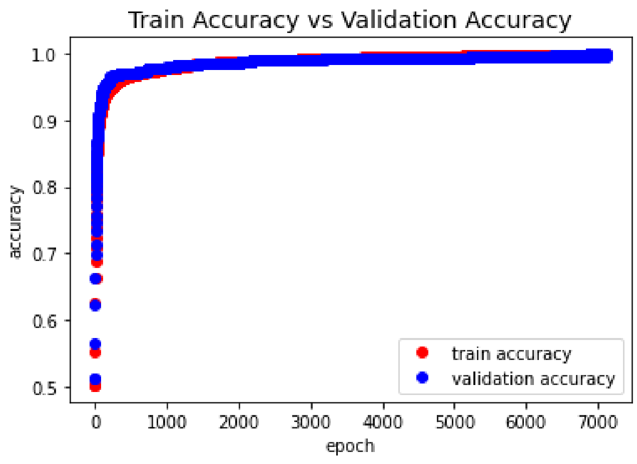


Figure 35. Accuracy comparison between training and validation

- **Predict result:** accuracy 99.4%

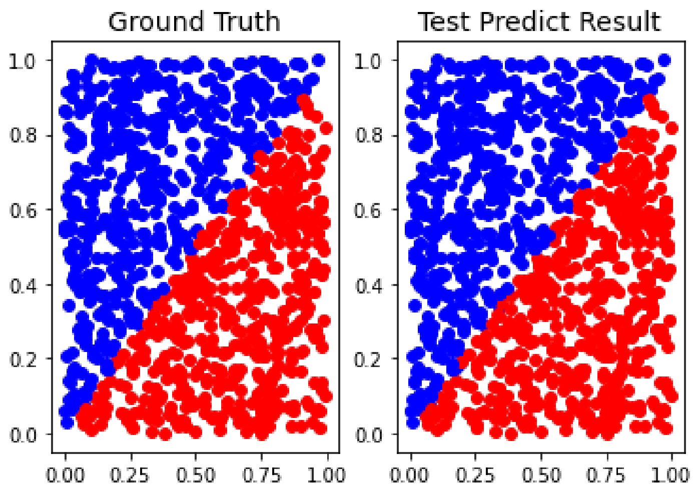


Figure 36. Predict results of neural network for uniform data

XOR data:

- **Learning curve (train loss, validation loss, epoch)**

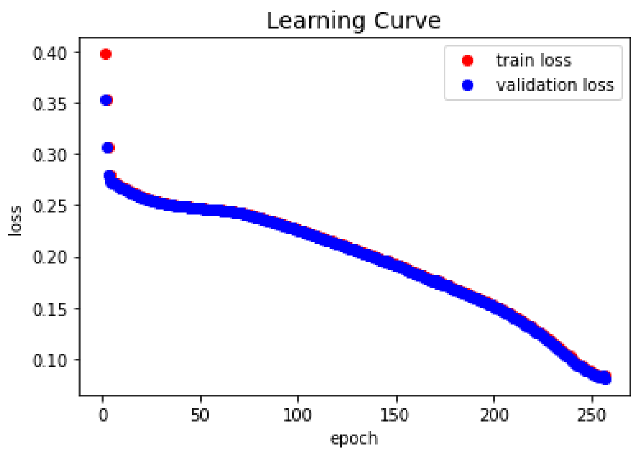


Figure 37. Learning curve of neural network for XOR data

- **Accuracy (train accuracy, validation accuracy, epoch)**

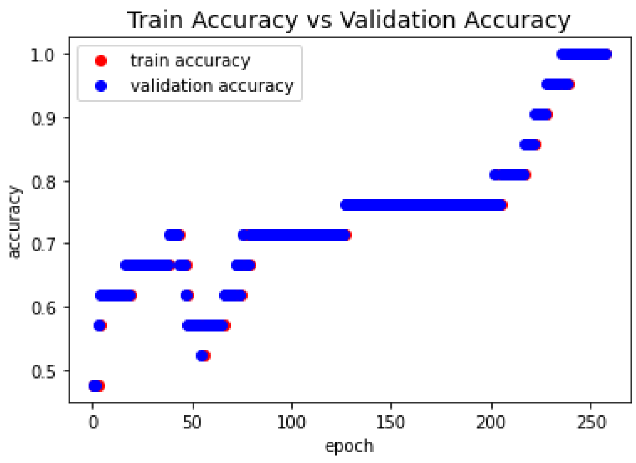


Figure 38. Accuracy comparison between training and validation

- **Predict result:** accuracy 100%

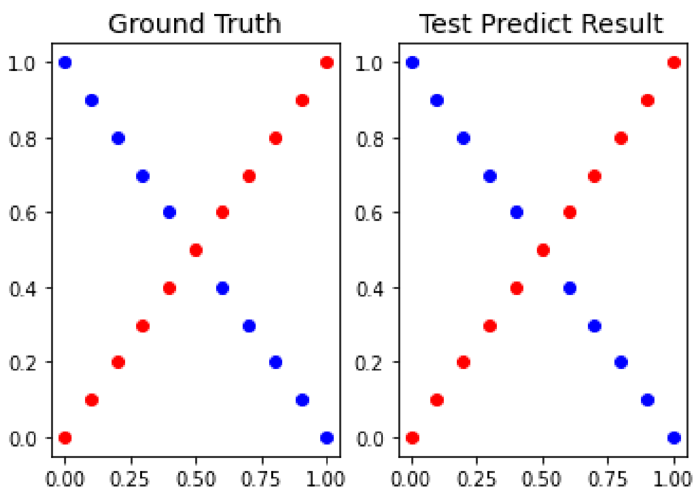
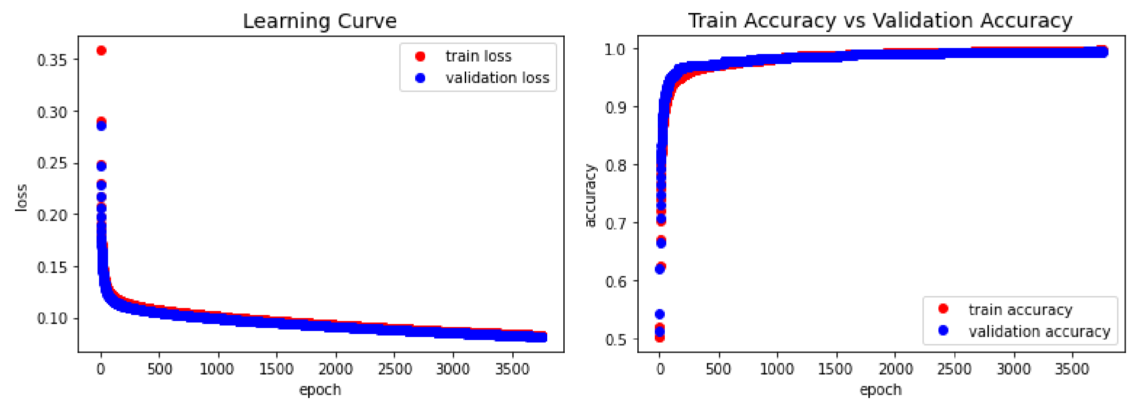


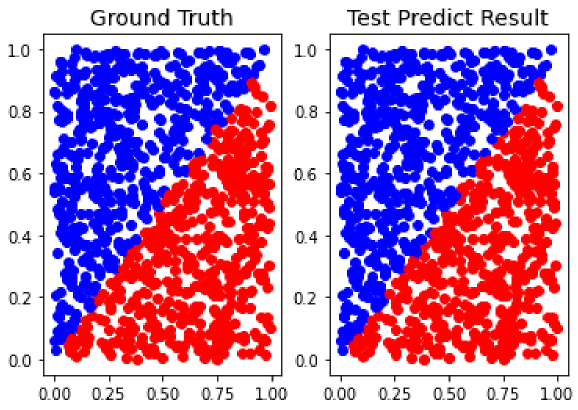
Figure 39. Predict results of neural network for XOR data

4. Discussion

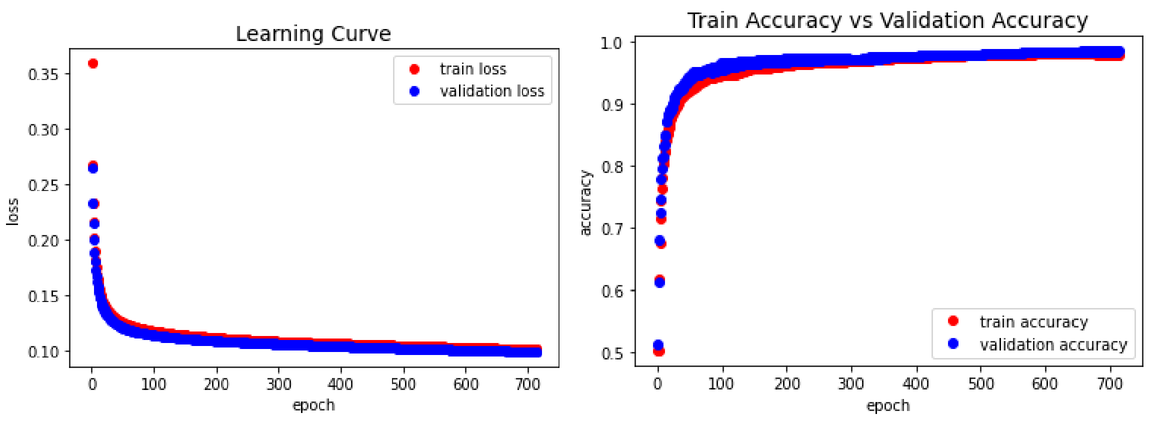
- **Try different learning rates:** use uniform data for this experiment

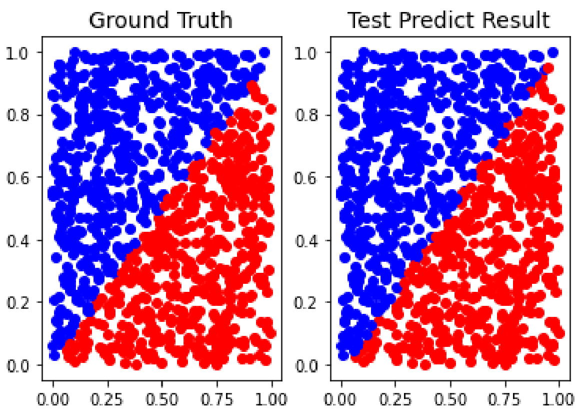
  <u>**Learning rate = 2.0:**</u> the epochs is totally decreased by 50%; the accuracy is 99.1%





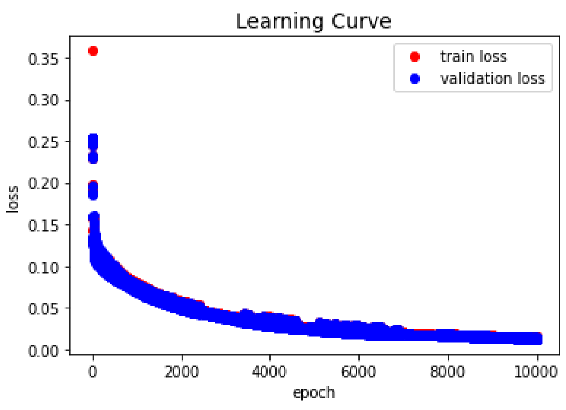
<u>Learning rate = 3.0:</u> the epochs is totally decreased by 20%; the accuracy is 97.9%

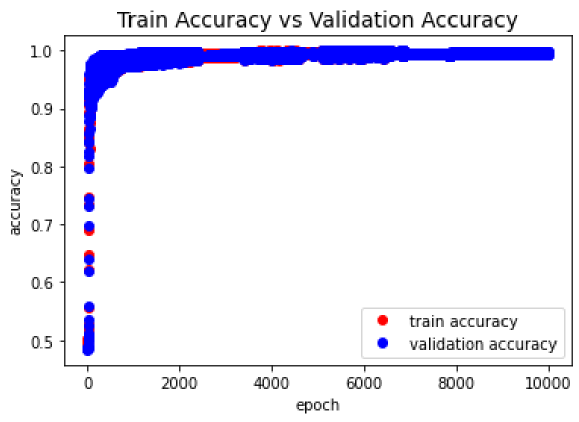


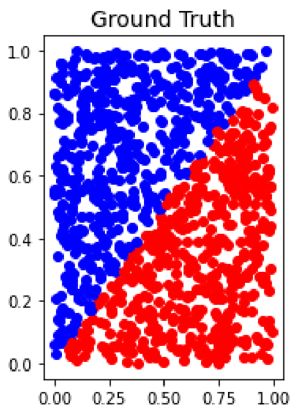


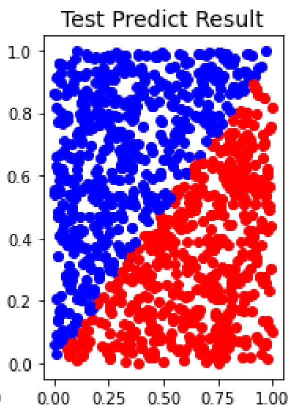
<u>Learning rate = 15.0:</u> the epochs is increased; but the accuracy is 99.4%

Compared with other learning rates, we can see the learning curve drops down dramatically and the accuracy goes up. The model still has good performance, but there are more and more glitches during the training and validation.





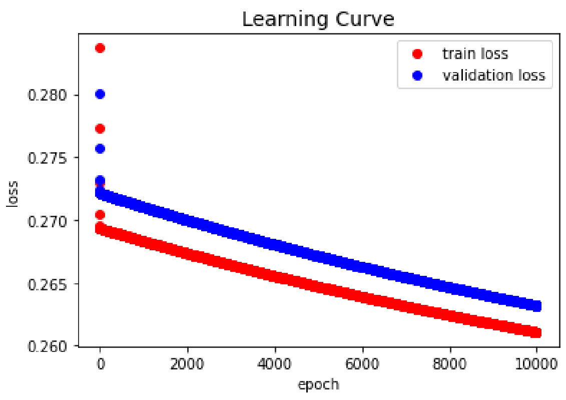


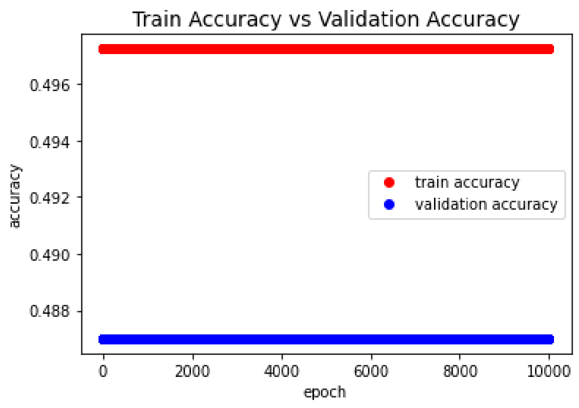


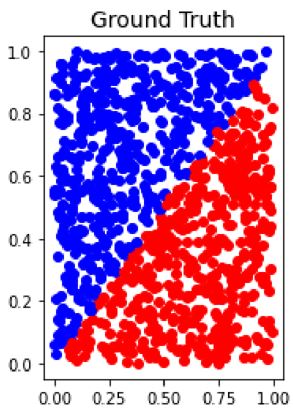
- **Try different numbers of hidden units:** use uniform data for this experiment

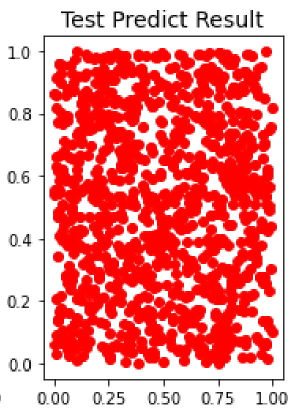
  <u>If we change the size of the last hidden layer to 1, 3, 4:</u>

  The model cannot predict correctly.



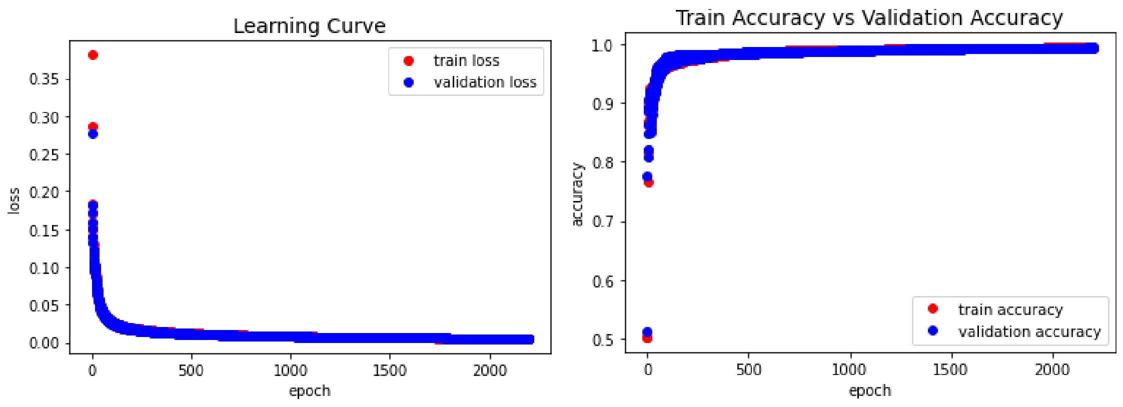


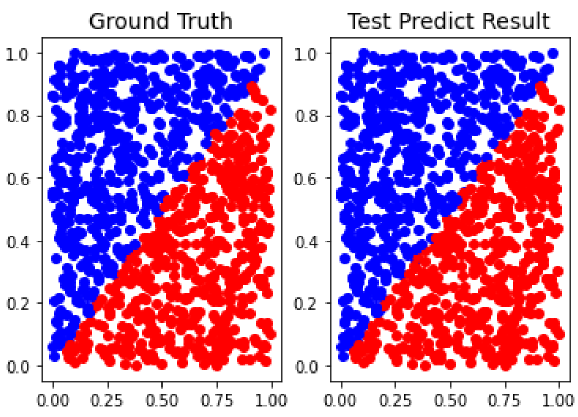




<u>If we change the size of the last hidden layer to 5, 6, 7… (above 5):</u>

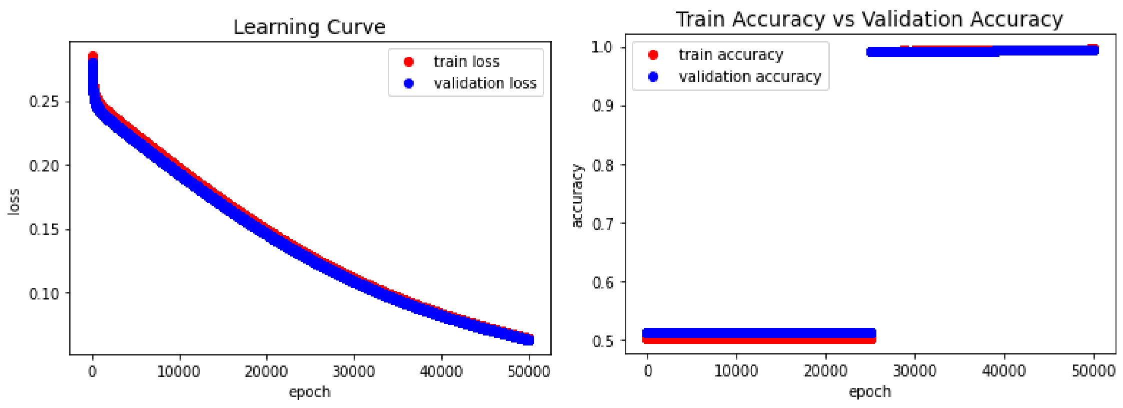
The model can predict normally and there is no overfitting. But the model is for ideal only (not realistic) because the data is from uniform distribution, and the decision line of labels is diagonal.

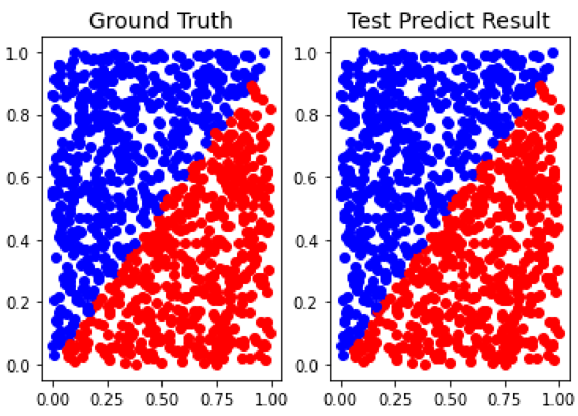




<u>If we remove one hidden layer and change the size of another hidden layer to 1:</u>

The model can also predict normally! That means we can use only one line to fit the curve. (In fact, applying deep learning is redundant. Perceptron can do it well.)

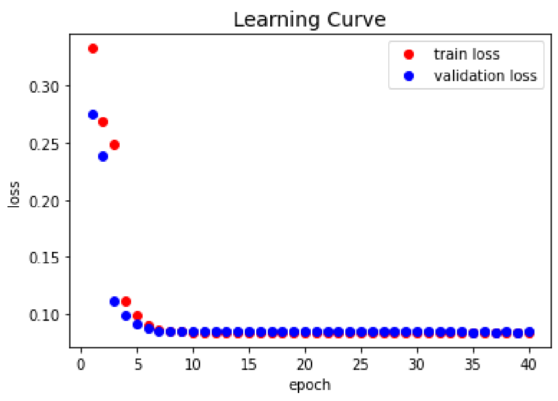


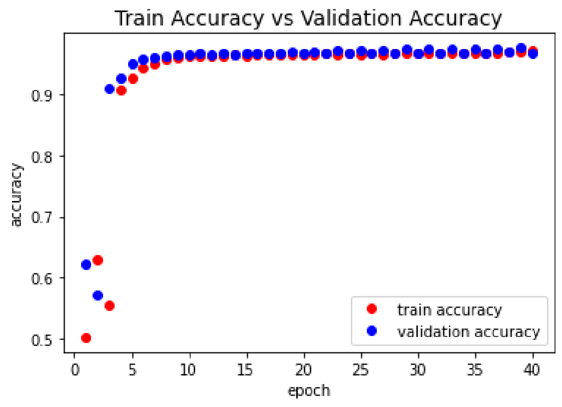


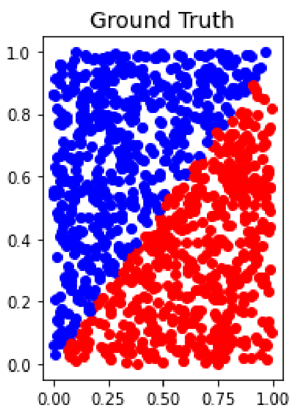
- **Try without activation functions:**

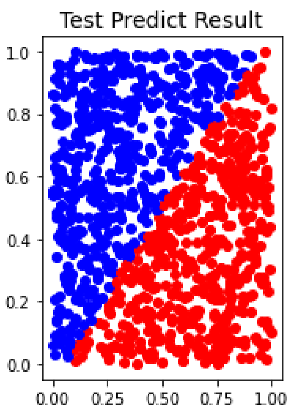
<u>Uniform data:</u>

The linear equation can easily fit the curve. The model doesn't need a nonlinear equation.



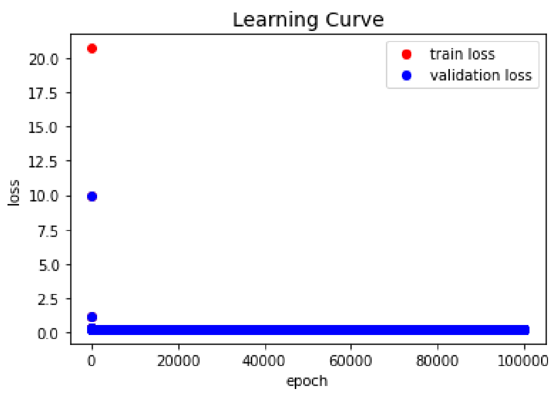


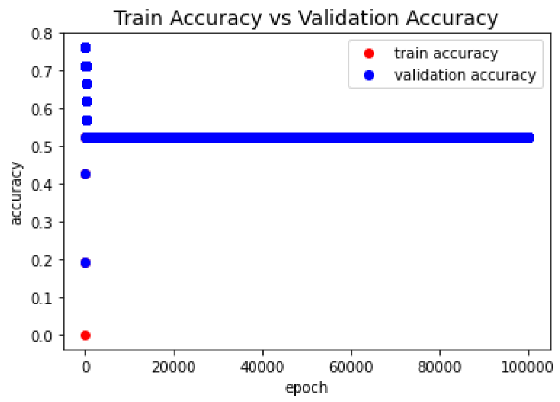


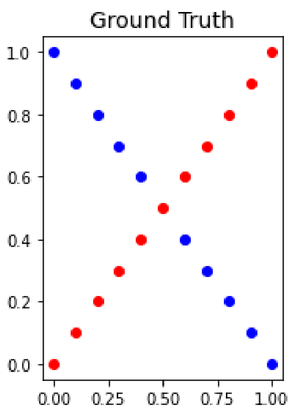


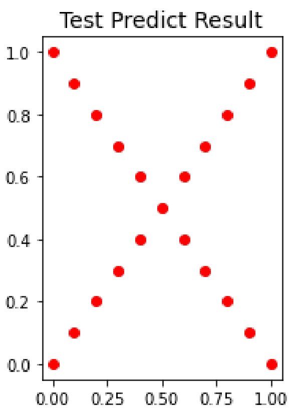
<u>But what if the data is XOR?</u>

Gradient vanishing! The model cannot only use a linear equation to fit the curve.



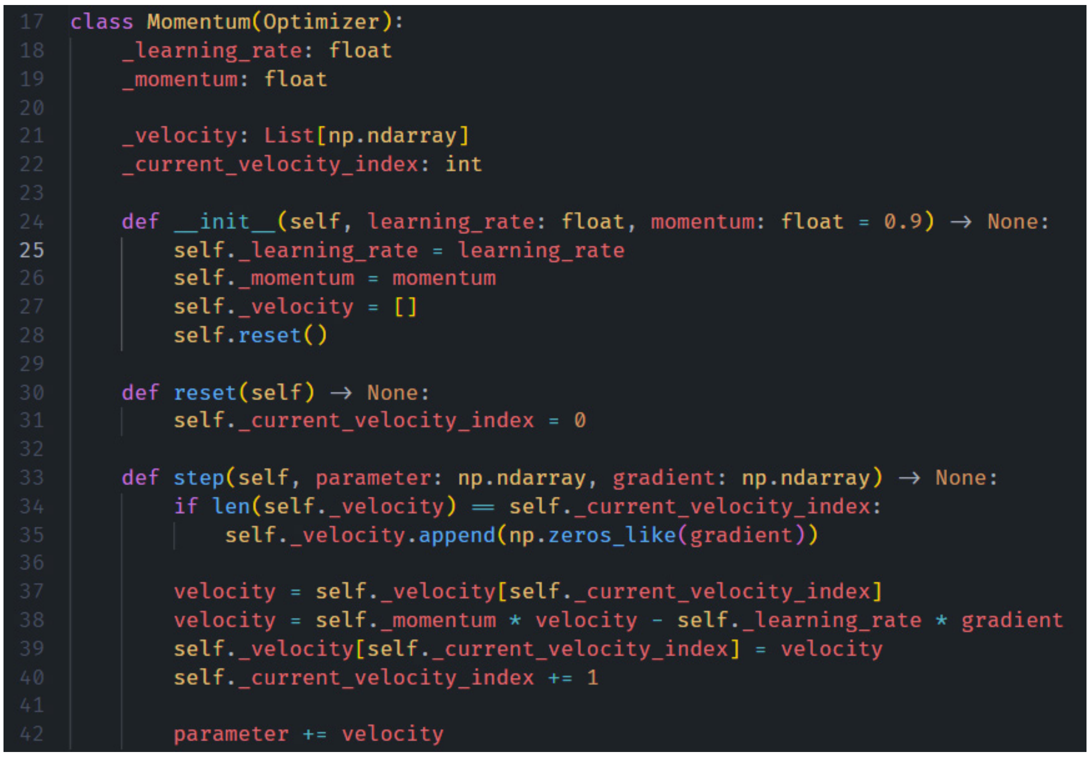






5. Extra

<u>Momentum optimizer</u>



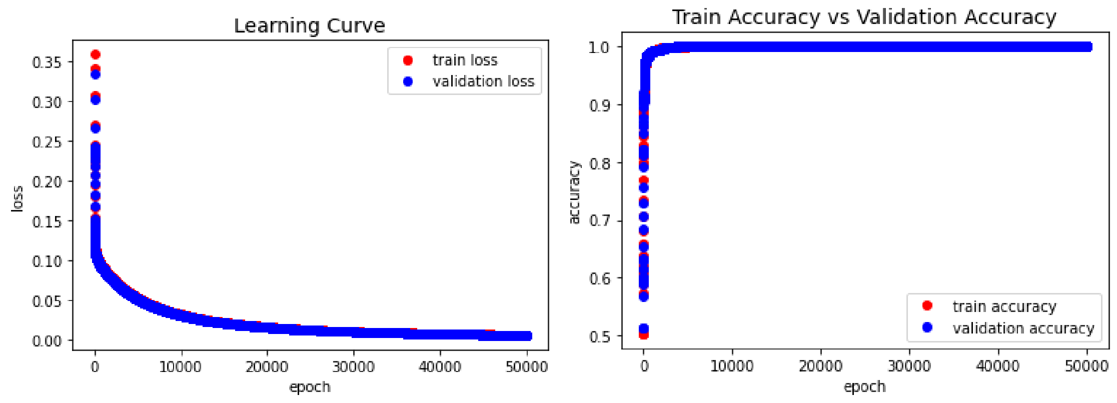
```python
class Momentum(Optimizer):
    _learning_rate: float
    _momentum: float

    _velocity: List[np.ndarray]
    _current_velocity_index: int

    def __init__(self, learning_rate: float, momentum: float = 0.9) → None:
        self._learning_rate = learning_rate
        self._momentum = momentum
        self._velocity = []
        self.reset()

    def reset(self) → None:
        self._current_velocity_index = 0

    def step(self, parameter: np.ndarray, gradient: np.ndarray) → None:
        if len(self._velocity) == self._current_velocity_index:
            self._velocity.append(np.zeros_like(gradient))

        velocity = self._velocity[self._current_velocity_index]
        velocity = self._momentum * velocity - self._learning_rate * gradient
        self._velocity[self._current_velocity_index] = velocity
        self._current_velocity_index += 1

        parameter += velocity
```

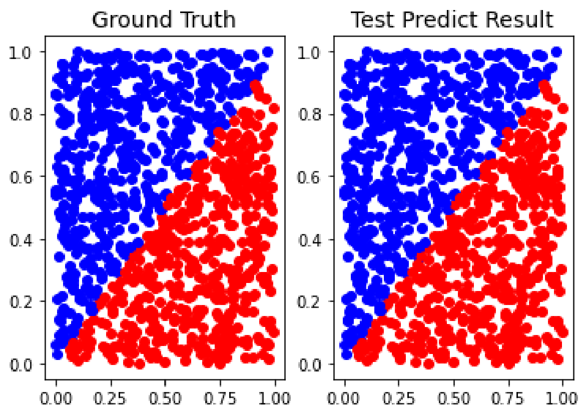
- Uniform data: use the same settings in experiment setups

  Learning rate: 0.5

  Momentum: 0.9

  Accuracy: 100% (very smooth but slow convergence)



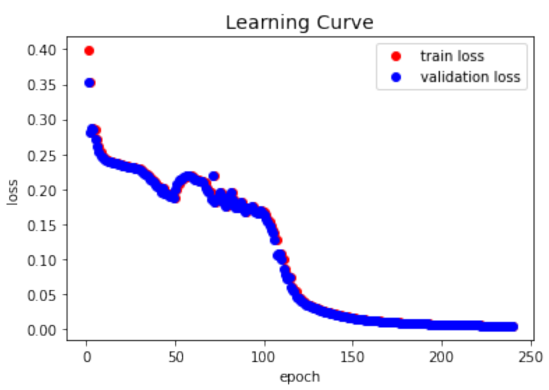


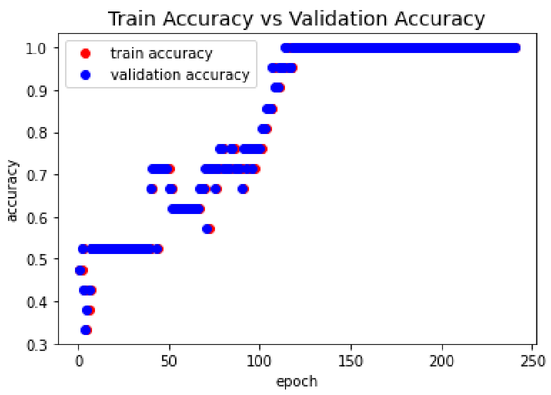
- XOR data: use the same settings in experiment setups

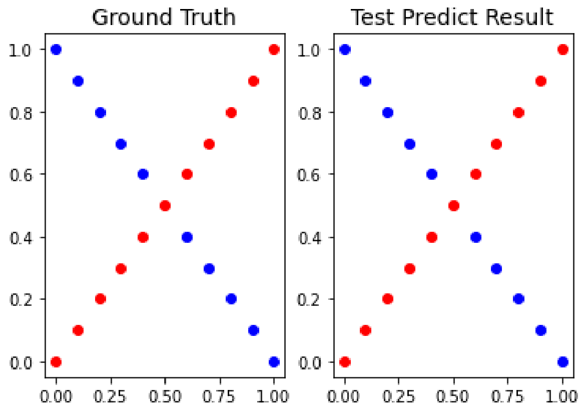
  Learning rate: 1.0

  Momentum: 0.9

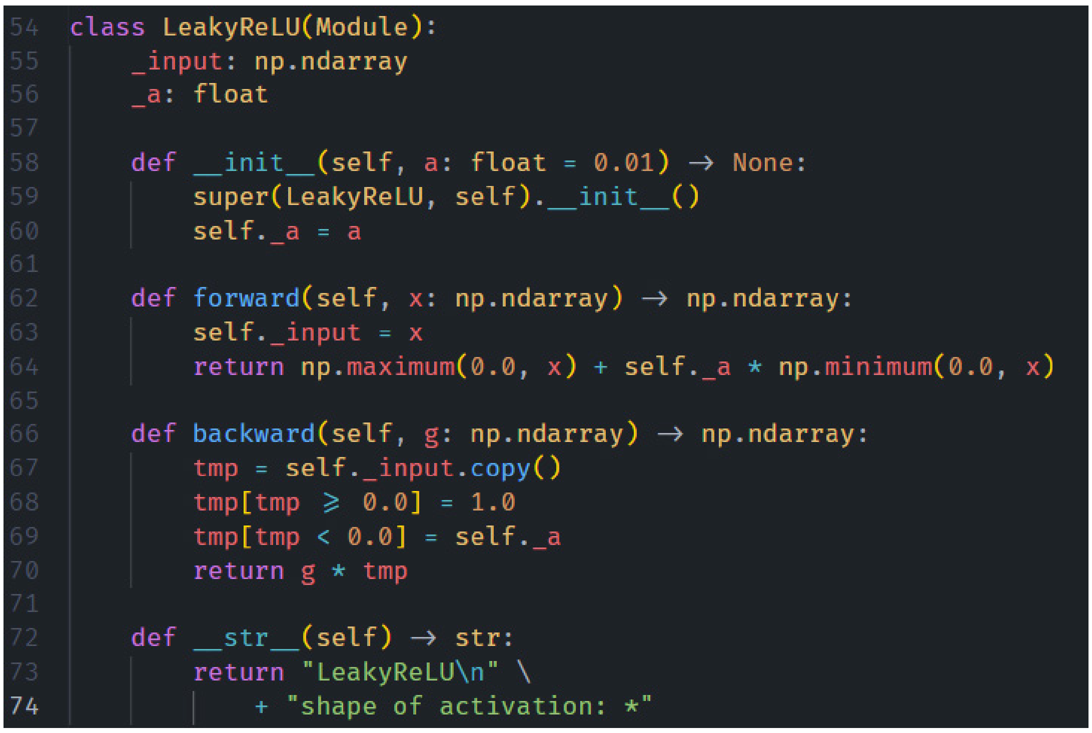
  Accuracy: 100%





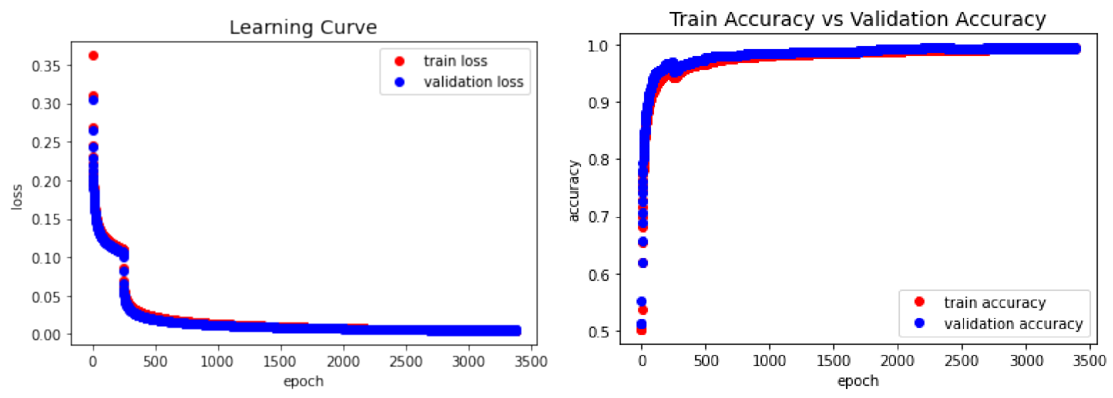


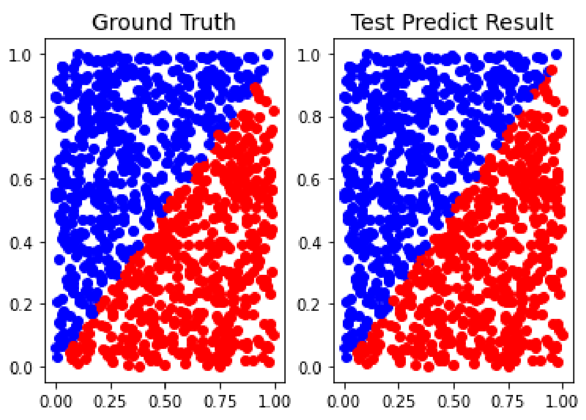
Leaky ReLU activation function as hidden units



```python
class LeakyReLU(Module):
    _input: np.ndarray
    _a: float

    def __init__(self, a: float = 0.01) -> None:
        super(LeakyReLU, self).__init__()
        self._a = a

    def forward(self, x: np.ndarray) -> np.ndarray:
        self._input = x
        return np.maximum(0.0, x) + self._a * np.minimum(0.0, x)

    def backward(self, g: np.ndarray) -> np.ndarray:
        tmp = self._input.copy()
        tmp[tmp >= 0.0] = 1.0
        tmp[tmp < 0.0] = self._a
        return g * tmp

    def __str__(self) -> str:
        return "LeakyReLU\n" \
            + "shape of activation: *"
```

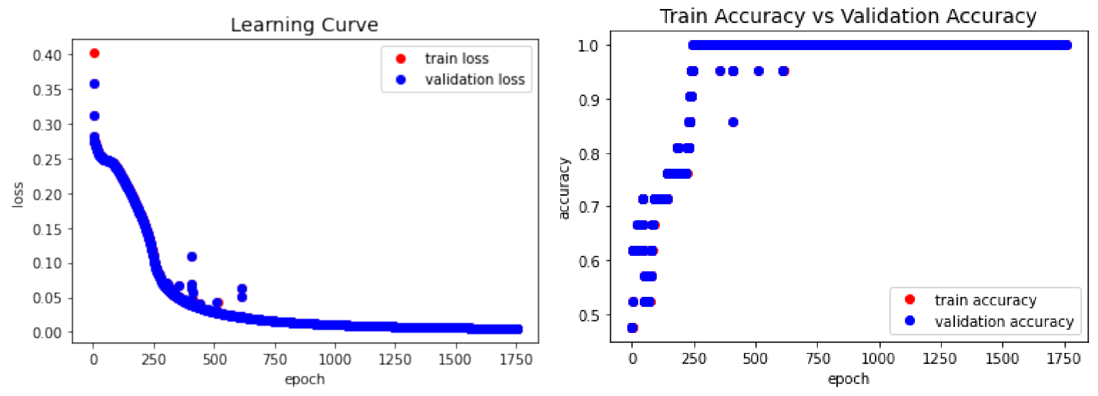
- Uniform data: use the same settings in experiment setups
  Accuracy: 99% (fast convergence)

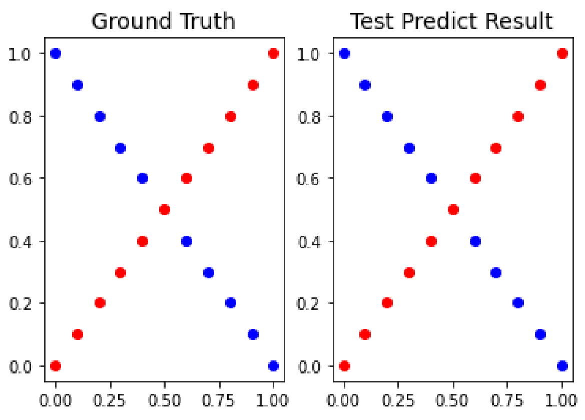




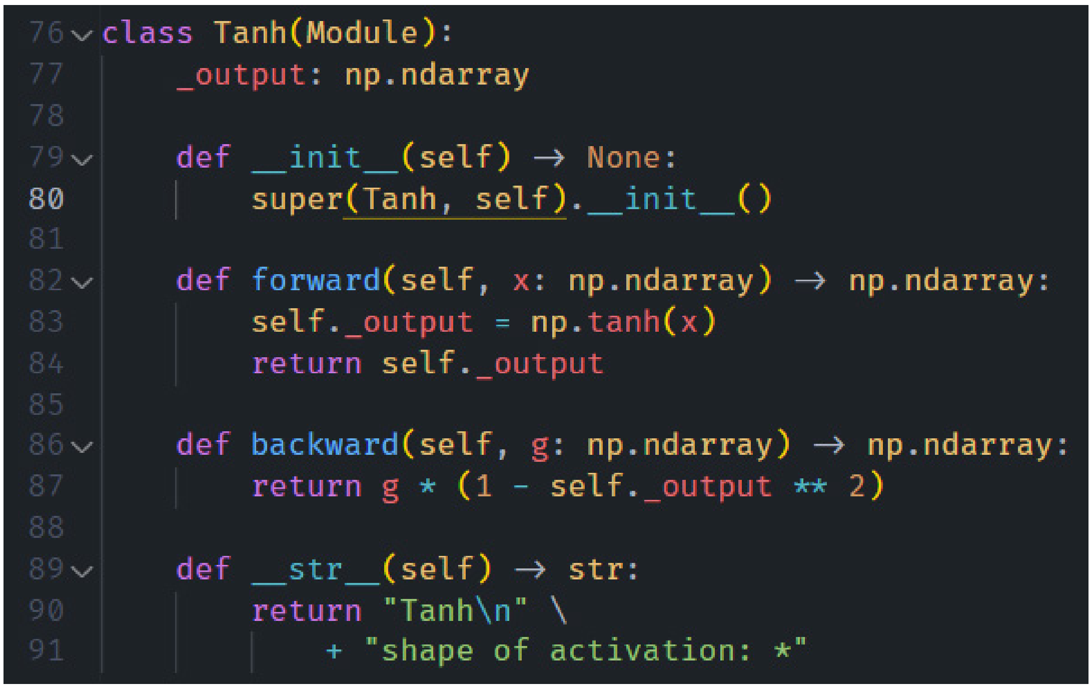
- XOR data: use the same settings in experiment setups

  Accuracy: 100% (fast convergence)



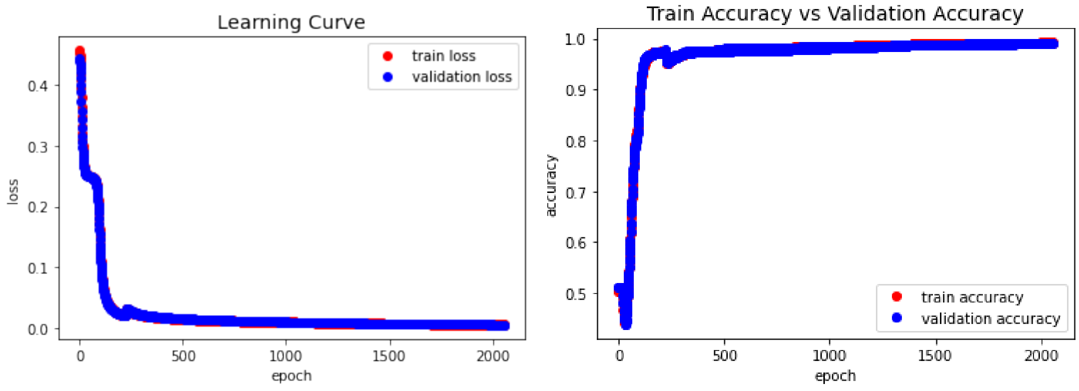


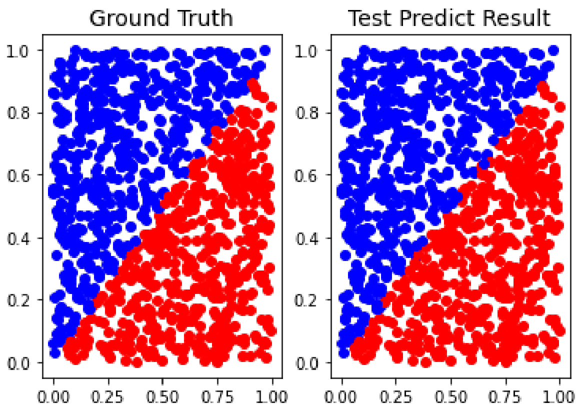
<u>**Tanh activation function as hidden units**</u>



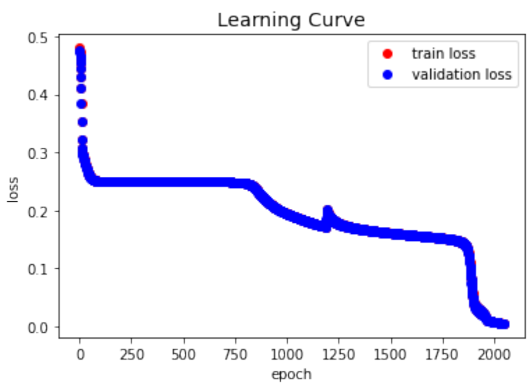
```python
class Tanh(Module):
    _output: np.ndarray

    def __init__(self) → None:
        super(Tanh, self).__init__()

    def forward(self, x: np.ndarray) → np.ndarray:
        self._output = np.tanh(x)
        return self._output

    def backward(self, g: np.ndarray) → np.ndarray:
        return g * (1 - self._output ** 2)

    def __str__(self) → str:
        return "Tanh\n" \
            + "shape of activation: *"
```

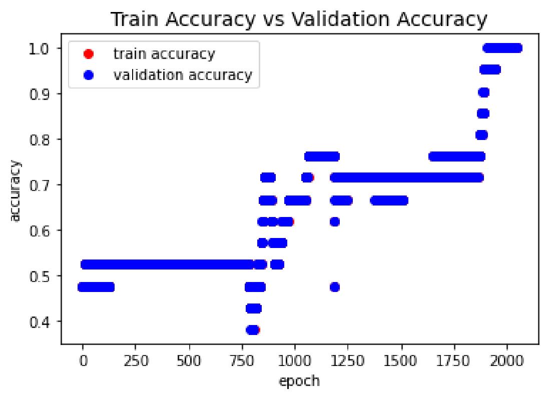
- Uniform data: use the same settings in experiment setups
  The generalization error between training and validation is almost disappeared.
  Accuracy: 99.1%

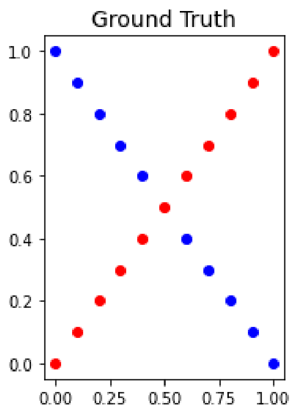


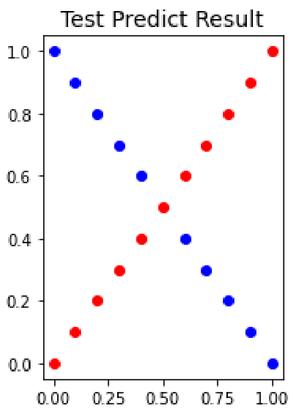


- XOR data: use the same settings in experiment setups
  Accuracy: 100%





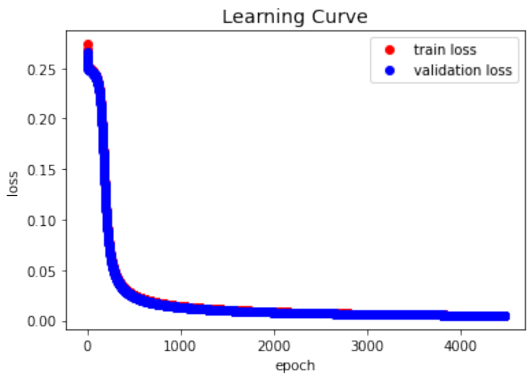


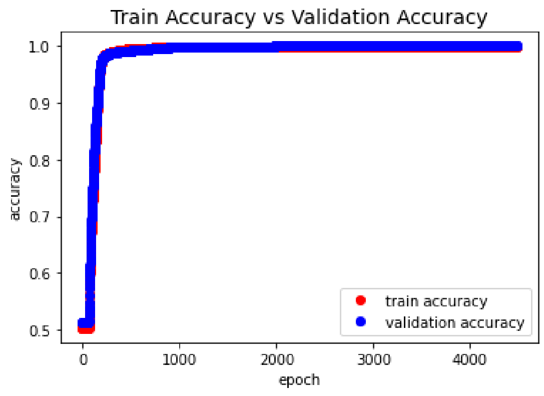


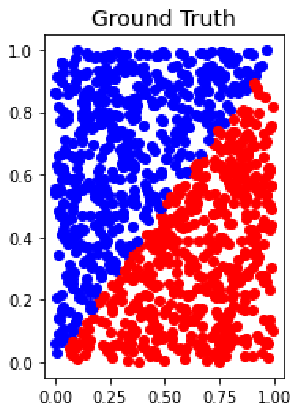
<u>**Sigmoid activation function as hidden units**</u>

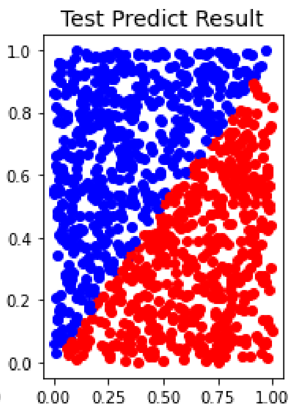
- Uniform data: use the same settings in experiment setups

  Accuracy: 99%



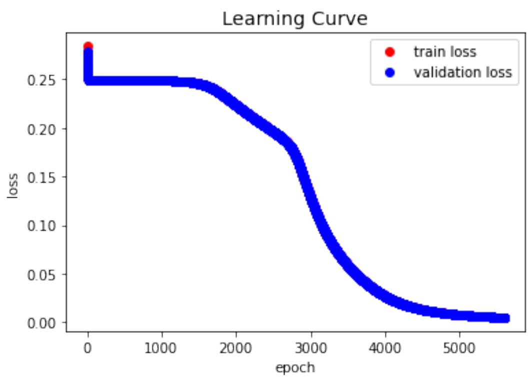


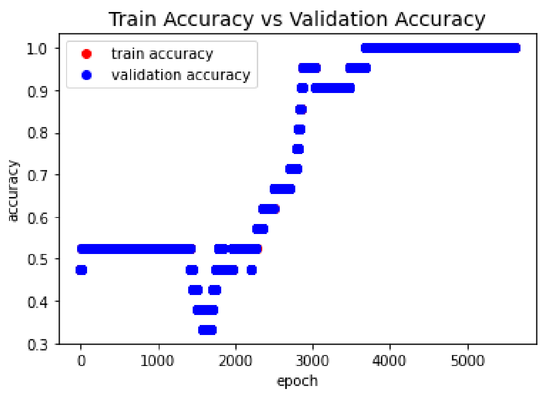


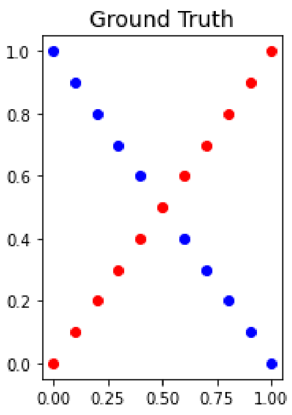


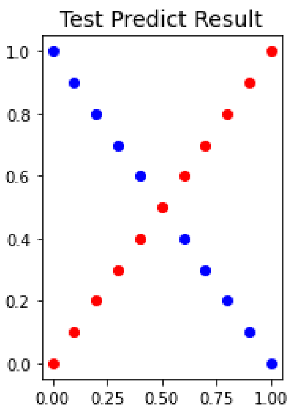
- XOR data: use the same settings in experiment setups

  Accuracy: 100%









6. Reference

A. Activation functions: https://ml-explained.com/blog/activation-functions-explained

B. PyTorch document: https://pytorch.org/docs/stable/index.html

C. Optimizers: https://medium.com/%E9%9B%9E%E9%9B%9E%E8%88%87%E5%85%94%E5%85%94%E7%9A%84%E5%B7%A5%E7%A8%8B%E4%B8%96%E7%95%8C/%E6%A9%9F%E5%99%A8%E5%AD%B8%E7%BF%92ml-note-sgd-momentum-adagrad-adam-optimizer-f20568c968db

D. Lab 1 Word & PowerPoint