Deep Learning Homework 1 Report

Cheng-Liang Chi

March 11, 2025

Contents

1 Introduction

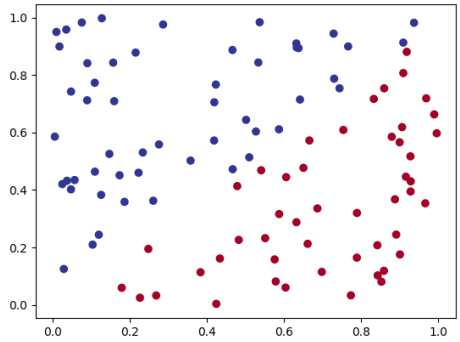
In this homework, we will need to implement a neruon network with backpropagation to solve a classification problem.

But we cannot use any existing libraries for machine learning, such as TensorFlow or PyTorch. We need to implement the neural network from scratch. The only allowed external library is NumPy, and also Matplotlib for plotting.

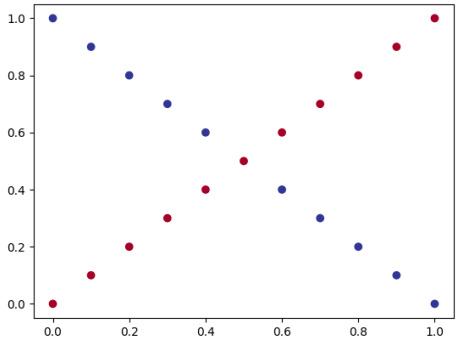
The problem we are going to solve is a binary classification problem. We have two dataset. Both datasets are 2D points. The first dataset is linearly separable, and the second dataset is not linearly separable.

The goal is to implement a neural network with backpropagation to classify the points in the two datasets. We will compare the performance of the neural network on the two datasets.

The following shows the visualization of the two datasets. The first dataset is shown in Figure 1a, and the second dataset is shown in Figure 1b.



(a) Linearly separable dataset



(b) XOR dataset

In this homework, my objective is to implement a machine learning framework from scratch, that is similar to PyTorch.

Therefore, I will implement a tensor class that supports basic operations, such as addition, subtraction, multiplication, and division. This tensor class should also support the gradient computation.

Then, I will implement a neural network class that supports the forward and backward propagation. The neural network class should also support the optimization algorithm, such as the stochastic gradient descent (SGD).

Finally, I will train the neural network on the two datasets and compare the performance of the neural network on the two datasets.

2 Implementation Details

2.1 Actiavtion Functions

2.1.1 Sigmoid Function

Since I have a tensor class that will track the computation graph, I will implement the sigmoid function as a method of the tensor class. The sigmoid function is defined as follows:

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

The derivative of the sigmoid function is:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x)) \tag{2}$$

The implementation shows below:

```
class Sigmoid:
    def __call__(self, x):
        out = Tensor(1 / (1 + np.exp(-x.data)), requires_grad=x.requires_grad)

        def grad_fn():
            if x.grad is not None:
                x.grad += (
                    out.data * (1 - out.data)
                ) * out.grad  # Ensure proper broadcasting

        if out.requires_grad:
            out._grad_fn = grad_fn
            out._prev = {x}

        return out
```

2.1.2 ReLU Function

The ReLU function is defined as follows:

$$\text{ReLU}(x) = \max(0, x) \tag{3}$$

The derivative of the ReLU function is:

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The implementation shows below:

```
class ReLU:
    def __call__(self, x):
        out = Tensor(np.maximum(0, x.data), requires_grad=x.requires_grad)

        def grad_fn():
            if x.grad is not None:
                x.grad += ((x.data > 0).astype(np.float32) * out.grad).reshape(
                    x.grad.shape
                )

        if out.requires_grad:
            out._grad_fn = grad_fn
```

```
            out._prev = {x}

        return out
```

2.2 Neural Network Architecture

As our FakeTorch library supports `nn.Linear`, which will provide a fully connected layer, we can easily build a neural network with multiple layers. The implementation of the neural network is shown below:

```
class MLP:
    def __init__(self, in_features, hidden_size, out_features,
    ↪  activation="sigmoid"):
        self.fc1 = torch.nn.Linear(in_features, hidden_size)
        self.fc2 = torch.nn.Linear(hidden_size, hidden_size)
        self.fc3 = torch.nn.Linear(hidden_size, out_features)

        self.activation_fn = activation_map(activation)

    def __call__(self, x):
        x = self.activation_fn(self.fc1(x))
        x = self.activation_fn(self.fc2(x))
        return self.fc3(x)  # No activation at the output layer

    def parameters(self):
        """Automatically collect all Tensor parameters from layers"""
        params = []
        for layer in self.__dict__.values():
            if hasattr(layer, "__dict__"):  # Check if it's an object with
            ↪  attributes
                for param in layer.__dict__.values():
                    if isinstance(param, torch.Tensor):  # Only include Tensors
                        params.append(param)
        return params
```

And the implementation of the `nn.Linear` is shown below:

```
class Linear:
    def __init__(self, in_features, out_features):
        self.W = Tensor(np.random.randn(in_features, out_features),
        ↪  requires_grad=True)
        self.b = Tensor(
            np.zeros((1, out_features)), requires_grad=True
        )  # Bias should match batch shape

    def __call__(self, x):
        return x.matmul(self.W) + self.b.data.reshape(
```

```
            1, -1
        )  # Ensures correct shape for batch
```

2.3 Loss Function

For the loss function, I implemented the mean squared error (MSE) loss function.

Since our tensor class will track the computation graph, we can calculate the gradient of the loss function with respect to the parameters of the model by calling the `backward` method of the loss tensor, which will update the gradient functions of the parameters. For more details, please refer to the backpropagation section.

2.3.1 Mean Squared Error (MSE)

The MSE loss function is defined as follows:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5}$$

The derivative of the MSE loss function is:

$$\frac{\partial \text{MSE}}{\partial \hat{y}_i} = \frac{2}{n}(\hat{y}_i - y_i) \tag{6}$$

The implementation of the MSE loss function is shown below:

```python
class MSELoss:
    def __call__(self, pred, target):
        return self.forward(pred, target)

    def forward(self, pred, target):
        loss_value = np.mean((pred.data - target.data) ** 2)
        out = Tensor(loss_value, requires_grad=True)

        def grad_fn():
            pred.grad += 2 * (pred.data - target.data) / target.data.shape[0]

        if out.requires_grad:
            out._grad_fn = grad_fn
            out._prev = {pred}

        return out

    def __repr__(self):
        return self.__class__.__name__
```

2.3.2 Binary Cross Entropy

The binary cross entropy (BCE) loss function is defined as follows:

$$\text{BCE} = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \tag{7}$$

The derivative of the BCE loss function is:

$$\frac{\partial \text{BCE}}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i} + \frac{1 - y_i}{1 - \hat{y}_i} \tag{8}$$

The implementation of the BCE loss function is shown below:

```python
class BCELoss:
    def __call__(self, pred, target):
        return self.forward(pred, target)

    def forward(self, pred, target):
        # Apply Sigmoid to logits (numerically stable)
        sigmoid = 1 / (1 + np.exp(-pred.data))  # Sigmoid function

        # Compute BCE Loss (avoid log(0) errors)
        loss_value = -np.mean(
            target.data * np.log(sigmoid + 1e-9)
            + (1 - target.data) * np.log(1 - sigmoid + 1e-9)
        )

        out = Tensor(loss_value, requires_grad=True)

        def grad_fn():
            # Gradient of BCE Loss w.r.t. logits
            pred.grad += (sigmoid - target.data) / target.data.shape[0]

        if out.requires_grad:
            out._grad_fn = grad_fn
            out._prev = {pred}

        return out

    def __repr__(self):
        return self.__class__.__name__
```

2.4 Backpropagation

For the backpropagation, we need to calculate the gradient of the loss function with respect to the parameters of the model.

However, since we have a tensor class that will track the computation graph, we can calculate the gradient of the loss function with respect to the parameters of the model by calling the `backward` method of the loss tensor.

Note that the gradients are tracked by updating their gradint functions during the forward pass. Thus, we can calculate the gradient of the loss function with respect to the parameters of the model by calling the `backward` method of the loss tensor.

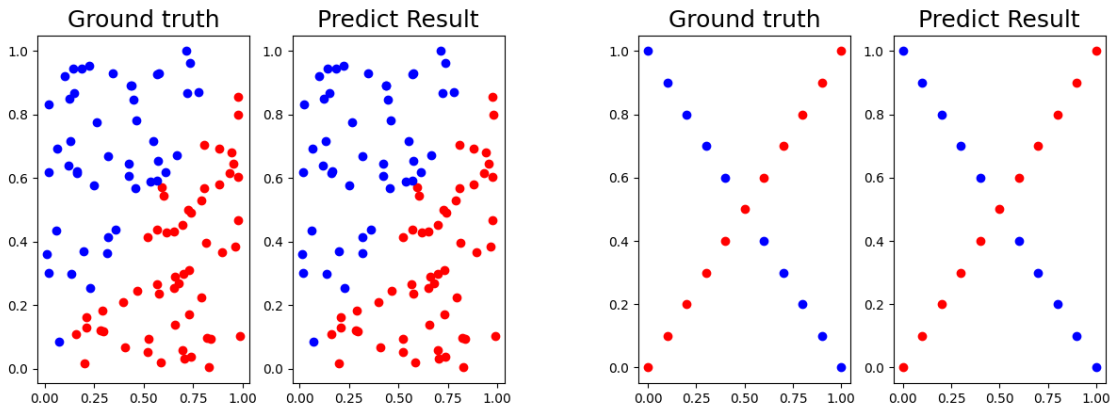
The implementation of the `backward` method is shown below:

```
    def backward(self):
        assert self.requires_grad, "This tensor has no gradient tracking!"
        topo = []
        visited = set()

        def build_topo(tensor):
            if tensor not in visited:
                visited.add(tensor)
                for parent in tensor._prev:
                    build_topo(parent)
                topo.append(tensor)

        build_topo(self)

        if self.grad is None:
            self.grad = np.ones_like(self.data)  # Seed gradient for scalar
            ↪ outputs

        for tensor in reversed(topo):
            if tensor._grad_fn:
                tensor._grad_fn()
```

3 Experimental Results

3.1 Screenshots

Results of the experiments are shown in figure 2.



(a) Dataset: Linear (Accuracy: 100%) (b) Dataset: XOR (Accuracy: 100%)

Figure 2: Results of the experiments

The above experiments are conducted with the following hyperparameters:

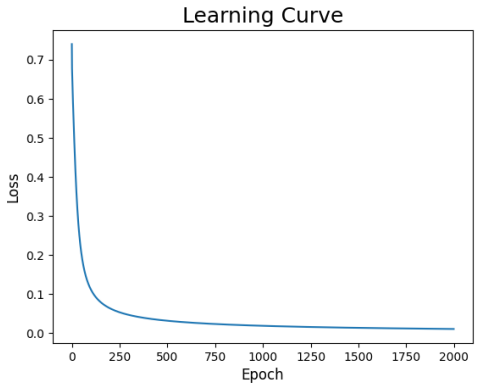
- Architecture: Multi-layer Perceptron
- Hidden units: 16
- Learning rate: 0.01
- Number of epochs: 2000
- Activation function: Sigmoid
- Loss function: Binary Cross Entropy
- Optimizer: SGD
- Dataset: Linear, XOR

3.2 Prediction Accuracy

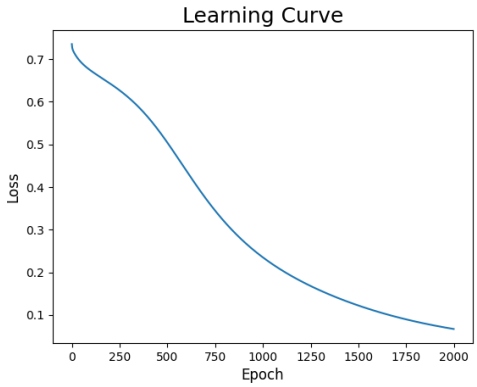
As shown in the screenshots, the model achieved 100% accuracy on both the linear and XOR datasets.

3.3 Learning Curve

The learning curve of the experiments is shown below (the hyperparameters are the same as above):



(a) Dataset: Linear



(b) Dataset: XOR

Figure 3: Learning curve of the experiments. The x-axis represents the number of epochs, and the y-axis represents the loss.

4 Discussion

4.1 Different Learning Rates

I tried to train the model with different learning rates. The results are shown in Table 1.

We can conclude that with a lower learning rate, the model cannot achieve a good accuracy. This is because the model converges too slowly. The reason is that the model updates too slow such that it is still not converged after 2000 epochs when looking at the loss curve. Therefore, the best learning rate is 0.01 for both XOR and linear datasets.

4.2 Different Number of Hidden Units

I also tried to train the model with different numbers of hidden units. The results are shown in Table 1.

We can conclude that the model with 4 hidden units is good enough for both XOR and linear datasets when the model are trained to converged (enough learning rate or epochs). The more hidden units, the more complex the model is, which also means the more parameters the model has. That will cause the training process to be slower and the model to be more likely to overfit.

However, in this homework, we will have the same dataset for both training and testing. That is, we will not observe the overfitting problem.

4.3 Without Activation Function

I also tried to train the model without an activation function. The results are shown in Table 2.

For this experiment, I used the same hyperparameters as the previous experiments except that the activation function is not used. From the implementation details, I implemented a identity function as the activation function to mimic the model without an activation function.

It is interesting that the model without an activation function can achieve 100% accuracy on the linear datasets. However, the model cannot achieve good results on the

XOR Dataset Accuracy (%)

Learning rate	Hidden Layer Size					
	1	**2**	**4**	**8**	**16**	**32**
0.1	52.38%	52.38%	100.00%	100.00%	100.00%	100.00%
0.01	52.38%	52.38%	52.38%	85.71%	100.00%	100.00%
0.001	52.38%	52.38%	52.38%	52.38%	57.14%	71.43%
0.0001	52.38%	52.38%	52.38%	52.38%	52.38%	52.38%
0.00001	52.38%	52.38%	52.38%	52.38%	52.38%	52.38%

Linear Dataset Accuracy (%)

Learning Rate	Hidden Layer Size					
	1	**2**	**4**	**8**	**16**	**32**
0.1	54.00%	100.00%	100.00%	100.00%	100.00%	100.00%
0.01	54.00%	100.00%	100.00%	100.00%	100.00%	100.00%
0.001	54.00%	54.00%	96.00%	100.00%	99.00%	100.00%
0.0001	54.00%	54.00%	54.00%	54.00%	85.00%	90.00%
0.00001	54.00%	54.00%	54.00%	54.00%	63.00%	54.00%

Table 1: Comparison of different learning rates and hidden layer sizes on XOR and linear datasets. All models are trained for 2000 epochs with BCELoss as the loss function. This experiment is trained using **sigmoid** as the activation function and using **SGD** as the optimizer.

XOR dataset. This is because the model without an activation function is a linear model. The XOR dataset is not linearly separable, so the model cannot achieve good results on the XOR dataset.

5 Questions

5.1 Questions 1

What is the purpose of activation functions?

The purpose of activation functions is to introduce non-linearity to the neural network. Without activation functions, the neural network would be a linear model, which is not capable of learning complex patterns in the data. Activation functions allow the neural network to learn complex patterns by introducing non-linearity to the model.

As discussed in Section 4.3, the neural network without activation functions is equivalent to a linear model. The output of the neural network is a linear combination of the input features, which is not capable of learning complex patterns such as XOR. Therefore, activation functions are essential for the neural network to learn complex patterns in the data.

XOR Dataset Accuracy (%)

Learning rate	Hidden Layer Size					
	1	**2**	**4**	**8**	**16**	**32**
0.1	52.38%	52.38%	52.38%	52.38%	52.38%	52.38%
0.01	52.38%	52.38%	52.38%	52.38%	52.38%	52.38%
0.001	52.38%	52.38%	52.38%	52.38%	52.38%	61.90%
0.0001	52.38%	52.38%	52.38%	52.38%	52.38%	52.38%
0.00001	52.38%	61.90%	52.38%	66.67%	52.38%	52.38%

Linear Dataset Accuracy (%)

Learning Rate	Hidden Layer Size					
	1	**2**	**4**	**8**	**16**	**32**
0.1	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
0.01	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
0.001	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
0.0001	54.00%	97.00%	97.00%	99.00%	100.00%	100.00%
0.00001	54.00%	94.00%	88.00%	90.00%	96.00%	99.00%

Table 2: Comparison of different learning rates and hidden layer sizes on XOR and linear datasets. All models are trained for 2000 epochs with BCELoss as the loss function. This experiment is trained using **none** as the activation function and using **SGD** as the optimizer.

5.2 Questions 2

What might happen if the learning rate is too large or too small?

As discussed in Section 4.1, if the learning rate is too large, the model may fail to converge, and the loss may oscillate or diverge. On the other hand, if the learning rate is too small, the model may take a long time to converge, and the training process may be slow. Therefore, it is essential to choose an appropriate learning rate to ensure the model converges quickly and efficiently.

5.3 Questions 3

What is the purpose of weights and biases in a neural network?

The purpose of weights and biases in a neural network is to learn the patterns in the data. Weights are the parameters that the neural network learns during the training process to map the input features to the output. Biases are the parameters that the neural network learns to shift the output of the linear transformation by a constant value. By learning the weights and biases, the neural network can learn the patterns in the data and make accurate predictions.

6 Extra

6.1 Different Optimizers

XOR Dataset Accuracy (%)

Learning rate	Hidden Layer Size					
	1	**2**	**4**	**8**	**16**	**32**
0.1	52.38%	76.19%	100.00%	100.00%	100.00%	100.00%
0.01	52.38%	52.38%	100.00%	100.00%	100.00%	100.00%
0.001	52.38%	52.38%	90.48%	100.00%	100.00%	100.00%
0.0001	52.38%	52.38%	52.38%	52.38%	57.14%	80.95%
0.00001	52.38%	52.38%	47.62%	52.38%	52.38%	52.38%

Linear Dataset Accuracy (%)

Learning Rate	Hidden Layer Size					
	1	**2**	**4**	**8**	**16**	**32**
0.1	54.00%	54.00%	100.00%	100.00%	100.00%	100.00%
0.01	54.00%	100.00%	100.00%	100.00%	100.00%	99.00%
0.001	54.00%	99.00%	100.00%	100.00%	100.00%	100.00%
0.0001	54.00%	96.00%	92.00%	100.00%	100.00%	100.00%
0.00001	54.00%	54.00%	54.00%	66.00%	86.00%	94.00%

Table 3: Comparison of different learning rates and hidden layer sizes on XOR and linear datasets. All models are trained for 2000 epochs with BCELoss as the loss function. This experiment is trained using **sigmoid** as the activation function and using **Adam** as the optimizer.

For the optimizers, I choosed to implement the Adam optimizer. The experimental results for Adam are shown in Table 3.

6.2 Different Activation Functions

For this experiment, we will compare the performance of different activation on both the linear and XOR datasets. The activation functions we will compare are:

- Sigmoid
- ReLU
- Tanh
- Identity (No activation function)

The experimental results for sigmoid and Identity are shown previously in Table 1 and Table 2.

The experimental results for ReLU and Tanh are shown in Table 4 and Table 5.

XOR Dataset Accuracy (%)

Learning rate	Hidden Layer Size					
	1	**2**	**4**	**8**	**16**	**32**
0.1	52.38%	52.38%	100.00%	100.00%	100.00%	100.00%
0.01	52.38%	76.19%	52.38%	100.00%	100.00%	100.00%
0.001	76.19%	76.19%	76.19%	100.00%	100.00%	100.00%
0.0001	52.38%	85.71%	52.38%	76.19%	90.48%	95.24%
0.00001	52.38%	52.38%	71.43%	71.43%	57.14%	80.95%

Linear Dataset Accuracy (%)

Learning Rate	Hidden Layer Size					
	1	**2**	**4**	**8**	**16**	**32**
0.1	54.00%	54.00%	100.00%	100.00%	100.00%	100.00%
0.01	54.00%	54.00%	100.00%	100.00%	100.00%	100.00%
0.001	54.00%	99.00%	100.00%	100.00%	100.00%	100.00%
0.0001	54.00%	54.00%	97.00%	99.00%	99.00%	100.00%
0.00001	54.00%	54.00%	54.00%	94.00%	97.00%	99.00%

Table 4: Comparison of different learning rates and hidden layer sizes on XOR and linear datasets. All models are trained for 2000 epochs with BCELoss as the loss function. This experiment is trained using **relu** as the activation function and using **SGD** as the optimizer.

6.3 Implement Convolutional Layers

I also implemented the convolutional layers in the model. And the performance of the model with convolutional layers can also achieve 100% accuracy on the linear dataset with the following hyperparameters:

- Architecture: Convolutional Neural Network
- Learning rate: 0.1
- Number of epochs: 2000
- Activation function: Tanh
- Loss function: Binary Cross Entropy
- Optimizer: Adam
- Dataset: Linear

The implementation details are shown bellows:

XOR Dataset Accuracy (%)

Learning rate	Hidden Layer Size					
	1	**2**	**4**	**8**	**16**	**32**
0.1	76.19%	52.38%	100.00%	100.00%	100.00%	100.00%
0.01	76.19%	90.48%	100.00%	100.00%	100.00%	100.00%
0.001	52.38%	57.14%	85.71%	100.00%	100.00%	100.00%
0.0001	52.38%	52.38%	66.67%	66.67%	85.71%	90.48%
0.00001	52.38%	52.38%	52.38%	71.43%	57.14%	85.71%

Linear Dataset Accuracy (%)

Learning Rate	Hidden Layer Size					
	1	**2**	**4**	**8**	**16**	**32**
0.1	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
0.01	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
0.001	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
0.0001	95.00%	94.00%	99.00%	99.00%	100.00%	100.00%
0.00001	54.00%	86.00%	94.00%	67.00%	97.00%	95.00%

Table 5: Comparison of different learning rates and hidden layer sizes on XOR and linear datasets. All models are trained for 2000 epochs with BCELoss as the loss function. This experiment is trained using **tanh** as the activation function and using **SGD** as the optimizer.

```
class CNN:
    def __init__(self, in_channels, out_channels, kernel_size,
    ↪  activation="sigmoid"):

        self.fc1 = torch.nn.Linear(in_channels, 1)
        self.conv1 = torch.nn.Conv1d(1, out_channels, kernel_size)

        self.activation = activation_map(activation)

    def __call__(self, x):
        x = self.activation_fn(self.fc1(x))
        x = self.activation_fn(self.conv1(x))
        return x  # No activation at the output layer

    def parameters(self):
        """Automatically collect all Tensor parameters from layers"""
        params = []
        for layer in self.__dict__.values():
            if hasattr(layer, "__dict__"):  # Check if it's an object with
            ↪  attributes
                for param in layer.__dict__.values():
```

```
                    if isinstance(param, torch.Tensor):  # Only include Tensors
                        params.append(param)
        return params
```