[Lab1] Back-propagation

0616014 楊政道

April 6, 2020

1 Introduction

1.1 Lab Objective

In this assignment, I will implement a simple neural network with both forward pass and back-propagation. The structure of the simple network will consist of fully-connected layers and sigmoid functions. The loss function will use mean squared error and the optimizer will use the stochastic gradient decent method.

Furthermore, I will try different activation functions, loss functions and optimizers to make the network converged better and faster with some experimental verification.

1.2 Project Structure

```
Lab1_0616014.zip
├── dataset/
│   ├── displayer.py
│   ├── generator.py
│   └── processor.py
└── model/
    ├── function/
    │   ├── activation.py
    │   └── loss.py
    ├── network.py
    ├── optimizer.py
    └── structure.py
```

1.2.1 dataset/displayer.py

There are some functions to display data including

- line plot(error plot or accuracy plot)
- decision region plot
- comparison plot between ground truth and prediction

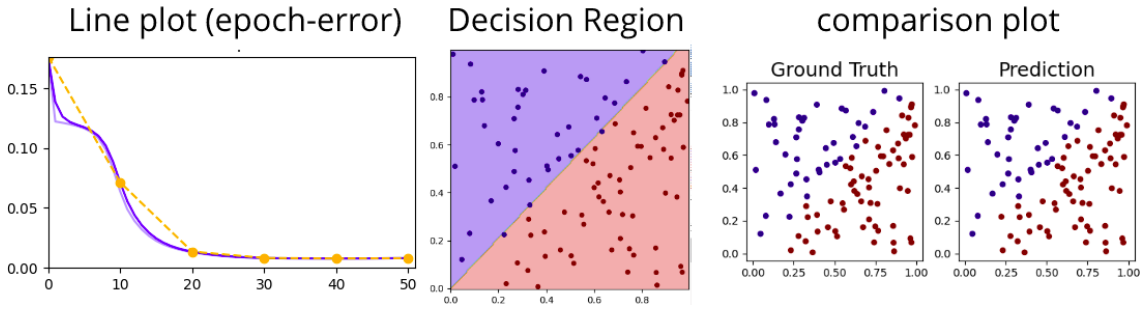


Figure 1: All kinds of displayers

1.2.2 dataset/generator.py

There are some classes to generate data including Linear, XOR and NSpirals(N is a variable).

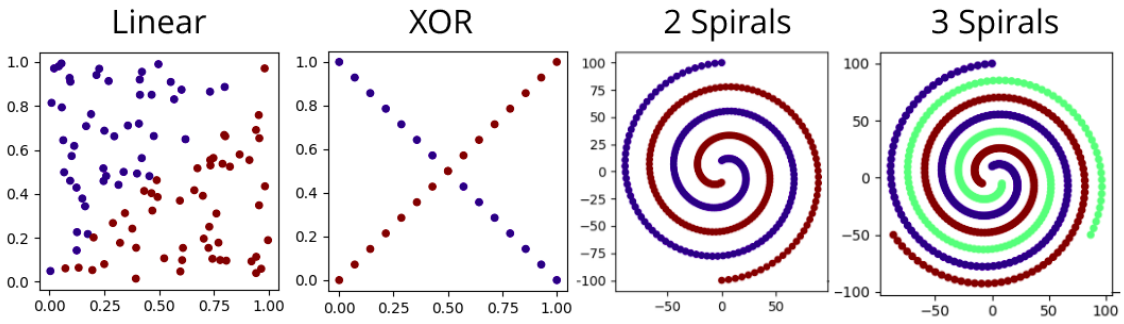


Figure 2: All kinds of generators

1.2.3 dataset/processor.py

There are some data handlers about one-hot encoder and decoder, data shuffling handler.

1.2.4 model/function/activation.py

There are some classes about activation functions layers including Sigmoid, ReLu(Leaky ReLu) and SoftMax.

1.2.5 model/function/loss.py

There are some classes about loss functions including mean squared error and binary cross entropy.

1.2.6 model/network.py

There is a class about neural networks to make the network construction easier.

1.2.7 model/optimizer.py

There are some classes about optimizers including SGD and Adam.

1.2.8 model/structure.py

There is a class about network structures including FullyConnectedLayer.

2 Experimental Setup

2.1 Fully Connected Layers

```
class FullyConnected:

    def __init__(self, m, n):
        self.W = np.random.uniform(-1, 1, (m + 1, n)).astype('float128')

    def forward(self, X):
        self.X = np.concatenate((
            X, np.array([ 1 for x in X ]).reshape(-1, 1)
        ), axis=1)
        return np.dot(self.X, self.W)

    def backward(self, E, optimizer):
        return optimizer.update(self, E)
```

A `m` to `n` fully-connected layer is a `m + 1` by `n` matrix (weight and bias). Its forward pass is just a dot operation with input matrix. When updating the fully-connected layer, we can call the update function of its optiomizer.

2.2 Sigmoid Function

```
class Sigmoid:

    def evaluate(self, X):
        return 1 / (1 + np.exp(-X))

    def gradient(self, X):
        return self.evaluate(X) * (1 - self.evaluate(X))

    def forward(self, X):
        self.X = X
        return self.evaluate(X)

    def backward(self, E, optimizer):
        return self.gradient(self.X) * E
```

An activation function contains two basic functions, evaluate and gradient. They correspond to defination and differential of sigmoid function.

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

$$\frac{\partial \text{Sigmoid}(x)}{\partial x} = \text{Sigmoid}(x)(1 - \text{Sigmoid}(x))$$

Then, its forward pass is to call evaluate function and its backward pass is to call gradient function and times the error passed from previous layer.

2.3 Mean Squared Error

```
class MeanSquaredError:

    def evaluate(self, Y_hat, Y):
        return (Y - Y_hat) ** 2 / 2.0

    def gradient(self, Y_hat, Y):
        return - (Y - Y_hat)
```

A loss function contains two basic functions which are similar to activation functions. They correspond to defination and differential of mean squared error.

$$\text{MSE}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

$$\frac{\partial \text{MSE}(y, \hat{y})}{\partial \hat{y}} = -(y - \hat{y})$$

2.4 Optimizer

```
class SGD:

    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, layer, E):
        gradient = np.dot(layer.X.T, E)
        E = np.dot(E, layer.W.T)[:, :-1]
        layer.W -= self.lr * gradient
        return E
```

An optimizer need to update weight of layers and pass the error to the next layers.

2.5 Neural Network

```
class Network:

    def __init__(self, structure, optimizer, loss):
        self.structure = structure
        self.optimizer = optimizer
        for layer in self.structure:
            self.optimizer.initial(layer)
        self.loss = loss
```

Input structure, optimizer and loss to define a network. A structure list will be like this below.

```
structure = [
    FullyConnected(2, 5),
    Sigmoid(),
    FullyConnected(5, 5),
    Sigmoid(),
    FullyConnected(5, 2),
    Sigmoid()
]
```

2.5.1 Forward Pass

```
class Network:

    def forward(self, Y):
        for layer in self.structure:
            Y = layer.forward(Y)
        return Y
```

Forward pass of the network is just to go through forward functions of all layers and return the result value.

2.5.2 Back-propagation

```
class Network:

    def backward(self, E):
        for layer in self.structure[::-1]:
            E = layer.backward(E, self.optimizer)
```

Backward pass of the network is to go through backward functions of all layers in the reversed order and update weight with optimizer.

2.5.3 Train

```
class Network:

    def train(self, X, Y, epochs = 10):
        for epoch in range(epochs):
            for x, y in zip(X, Y):
                x = x.reshape(1, -1)
                y = y.reshape(1, -1)
                y_hat = self.forward(x)
                self.backward(self.loss.gradient(y_hat, y))
```

Training function of the network is to do a forward and a backward pass for all data pairs with severals epochs.

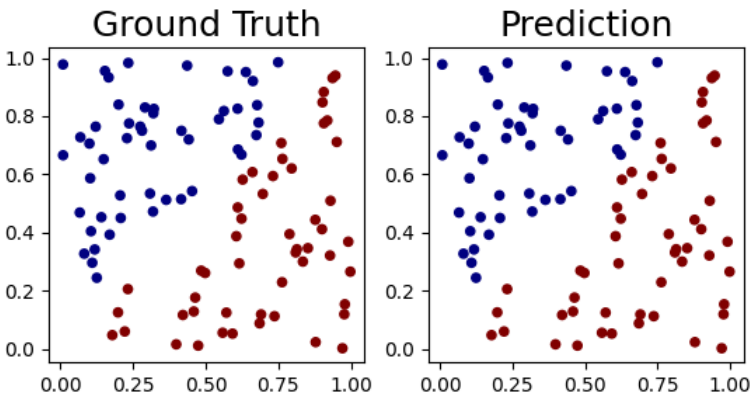
3 Experimental Result

3.1 Linear

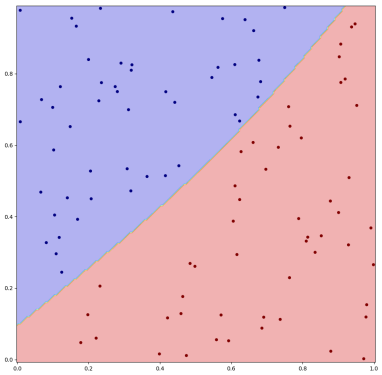
3.1.1 Network Settings

```
# dataset
dataset = Linear()
dataset_num = 100

# network
network_structure = [
    FullyConnected(2, 3),
    Sigmoid(),
    FullyConnected(3, 3),
    Sigmoid(),
    FullyConnected(3, 2),
    Sigmoid(),
]

# train
epochs = 50
optimizer = SGD(0.5)
loss = MeanSquaredError()
```

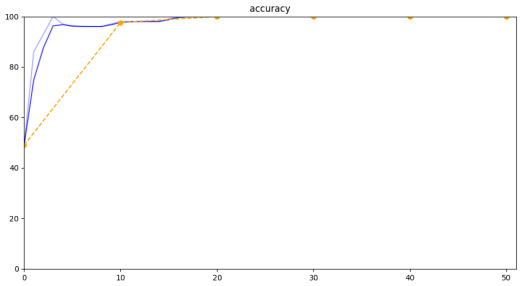
3.1.2 Experimental Result



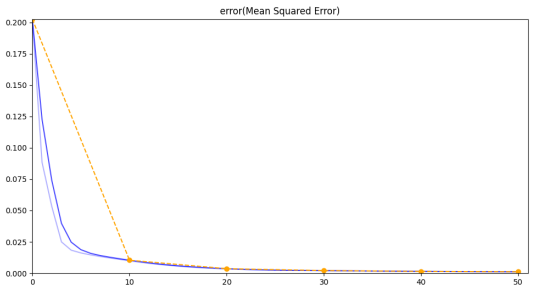
(a) comparsion plot



(b) decision region



(c) accuracy plot



(d) error plot

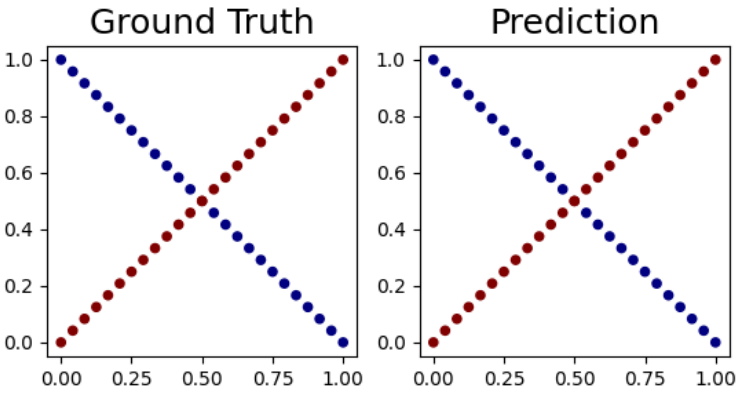
Figure 3: linear

3.2 XOR

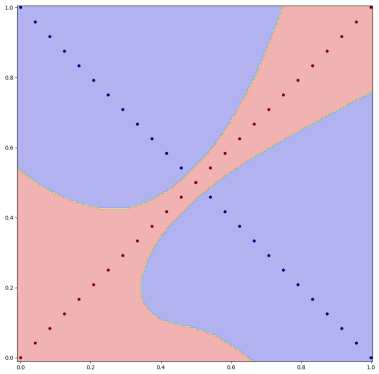
3.2.1 Network Settings

```
# dataset
dataset = XOR()
dataset_num = 50

# network
network_structure = [
    FullyConnected(2, 8),
    Sigmoid(),
    FullyConnected(8, 8),
    Sigmoid(),
    FullyConnected(8, 2),
    Sigmoid(),
]

# train
epochs = 200
optimizer = SGD(1)
loss = MeanSquaredError()
```

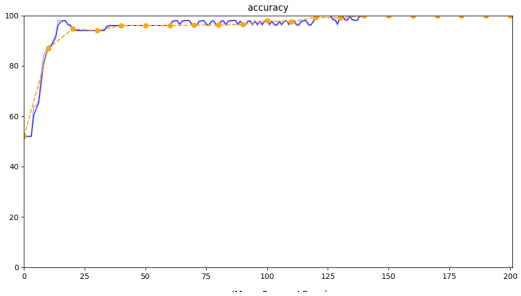
3.2.2 Experimental Result



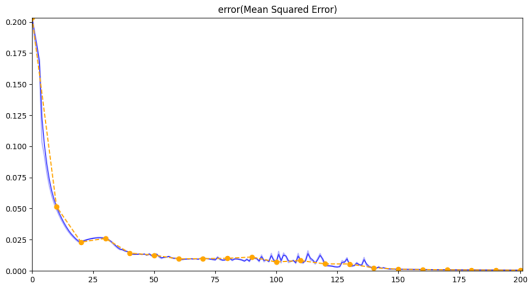
(a) comparsion plot



(b) decision region



(c) accuracy plot



(d) error plot

Figure 4: xor

4 Discussion

4.1 Activation Function - Sigmoid and Others

4.1.1 Sigmoid

Sigmoid function has two fatal problems

1. The computation of the exponential function is costly and slow.
2. The gradient shrinks when the input value is far from the zero.

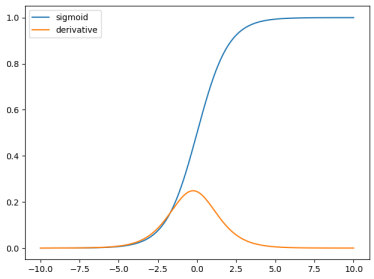


Figure 5: sigmoid function

Therefore, we need to find a kind of activation function which doesn't have high-computation operation and gradient shrinking.

4.1.2 ReLu

ReLu activation function improves the disadvantage of the sigmoid function memtioned above. It doesn't have any exponential operations and the gradient doesn't shrink for the large input value.

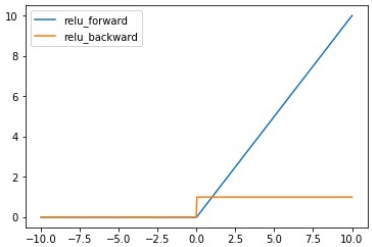


Figure 6: relu function

```
class ReLu:

    def __init__(self, r):
        self.r = r

    def evaluate(self, X):
        return X * (X > 0) + self.r * (X < 0) * X

    def gradient(self, X):
        return 1 * (X > 0) + self.r * (X < 0)

    def forward(self, X):
        self.X = X
        return self.evaluate(X)

    def backward(self, E, optimizer):
        return self.gradient(self.X) * E
```

However, when we replace the activation function of the last layer, the range of the output value will not be bounded in (0, 1). Therefore, we need another activation function to bound the range of the output value.

4.1.3 SoftMax

We can use softmax function to make the range of the output value into (0, 1).

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

```
class SoftMax:

    def evaluate(self, X):
        X = X - np.max(X, axis=1).reshape(-1, 1)
        e_X = np.exp(X)
        return e_X / np.sum(e_X, axis=1).reshape(-1, 1)

    def gradient(self, X):
        return self.evaluate(X).T * (np.identity(X.shape[1]) - self.evaluate(X))

    def forward(self, X):
        self.X = X
        return self.evaluate(X)

    def backward(self, E, optimizer):
        return np.dot(E, self.gradient(self.X))
```

The gradient of the softmax activation function is more complex than other activation functions.

$$\frac{\partial \text{softmax}(x_i)}{\partial x_i} = \text{softmax}(x_i)(\mathbb{I}(i = j) - \text{softmax}(x_j))$$

4.2 Loss Function

4.2.1 Mean Squared Error

Mean squared error has a fatal error the gradient at the position close to zero is too smal to train. Therefore, when the late of the training stage, the network is hard to converge.

4.2.2 Binary Cross Entropy

```
class BinaryCrossEntropy:

    def evaluate(self, Y_hat, Y):
        return - (Y * np.log(1e-9 + Y_hat) + (1 - Y) * np.log(1e-9 + 1 - Y_hat))

    def gradient(self, Y_hat, Y):
        return - (Y / (1e-9 + Y_hat) - (1 - Y) / (1e-9 + 1 - Y_hat))
```

A loss function contains two basic functions(evaluate and gradient). They correspond to defination and differential of binary cross entropy.

$$\text{BCE}(y, \hat{y}) = y \times log(\hat{y}) + (1 - y) \times log(1 - \hat{y})$$

$$\frac{\partial \text{BCE}(y, \hat{y})}{\partial \hat{y}} = \frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}}$$

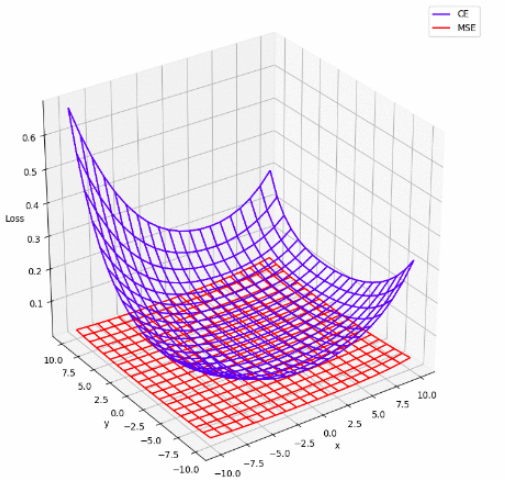


Figure 7: mean squared error vs cross entropy

4.3 Optimizer

4.3.1 Stochastic Gradient Decent

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

Stochastic gradient decent has some fatal problems.

- cannot do anything when stucking in local minimum.
- use the same learning rate to update all the elements of the weight.

4.3.2 Adam

Adam optimizer use m and v two variables to update weight.

$$W \leftarrow W - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

About m

Gradient desent is like putting a ball on a surface and the ball will fall into low. The ball will have the momentum so that it can slide out from the local minimum. Variable m is like the momentum of the ball. For each iteration, it refer to past momentum like the inertia of the ball.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L_t}{\partial W_t}$$

About v

The variable v can make the sensitive parameters change less and the smoothing parameters change more, so that the whole parameters can be close to the optimized value at the same time without causing same parts to be prematurely trained but interfered by other parameters.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\frac{\partial L_t}{\partial W_t})^2$$

About decay

When the late of the training stage, the learning rate will be more less than the begining of the training stage's. Therefore, we can use the following formula to make the change smaller and smaller over time.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

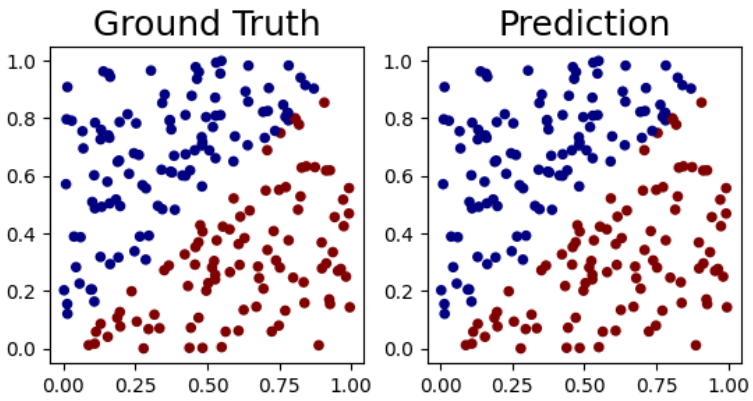
5 Extra Experiments

5.1 Linear with Advanced Model

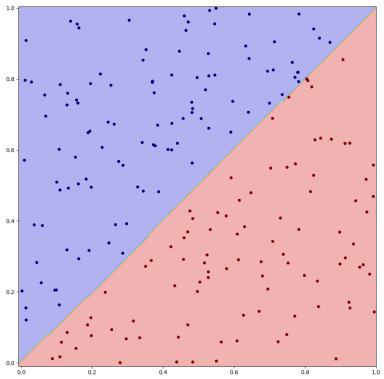
5.1.1 Network Settings

```
# dataset
dataset = Linear()
dataset_num = 200

# network
network_structure = [
    FullyConnected(2, 5),
    ReLu(0.1),
    FullyConnected(5, 5),
    ReLu(0.1),
    FullyConnected(5, 2),
    SoftMax()
]

# train
epochs = 30
optimizer = Adam(0.1, 0.99)
loss = BinaryCrossEntropy()
```

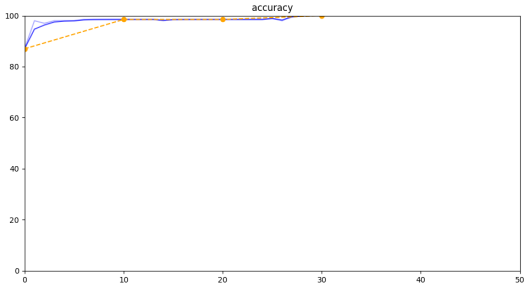
5.1.2 Experimental Result



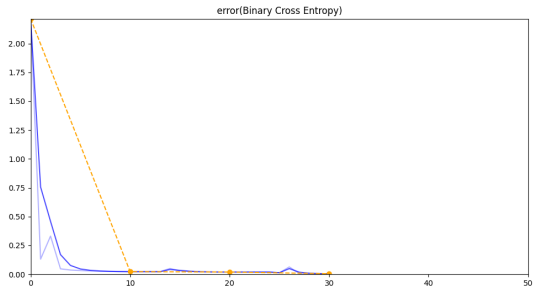
(a) comparsion plot



(b) decision region



(c) accuracy plot



(d) error plot

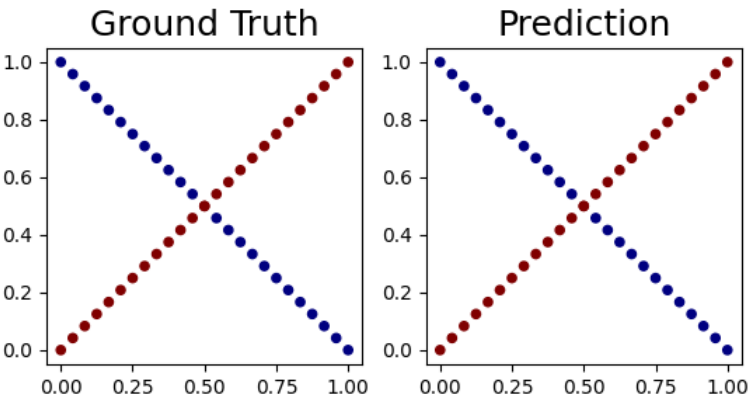
Figure 8: linear with advanced model

5.2 XOR with Advanced Model

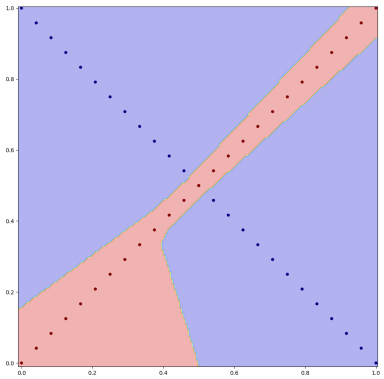
5.2.1 Network Settings

```
# dataset
dataset = XOR()
dataset_num = 50

# network
network_structure = [
    FullyConnected(2, 5),
    ReLu(0.1),
    FullyConnected(5, 5),
    ReLu(0.1),
    FullyConnected(5, 2),
    SoftMax()
]

# train
epochs = 100
optimizer = Adam(0.1, 0.99)
loss = BinaryCrossEntropy()
```

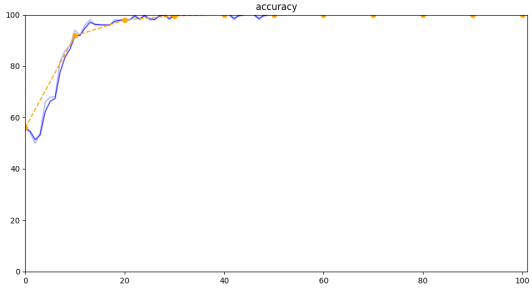
5.2.2 Experimental Result



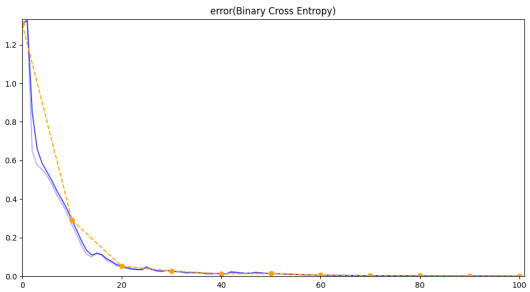
(a) comparsion plot



(b) decision region



(c) accuracy plot



(d) error plot

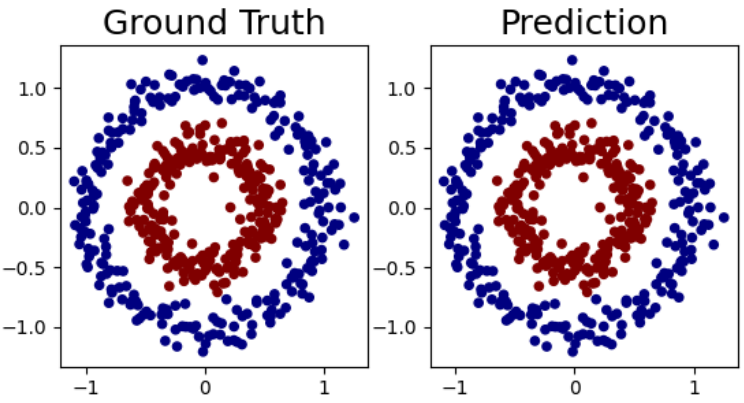
Figure 9: xor with advanced model

5.3 2 Circles with Advanced Model

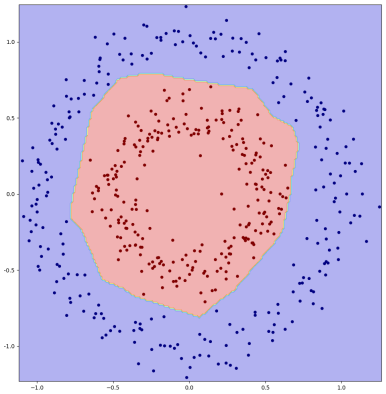
5.3.1 Network Settings

```
# dataset
dataset = TwoCircles()
dataset_num = 500

# network
network_structure = [
    FullyConnected(2, 15),
    ReLu(0.1),
    FullyConnected(15, 15),
    ReLu(0.1),
    FullyConnected(15, 2),
    SoftMax()
]

# train
epochs = 30
optimizer = Adam(0.01, 0.99)
loss = BinaryCrossEntropy()
```

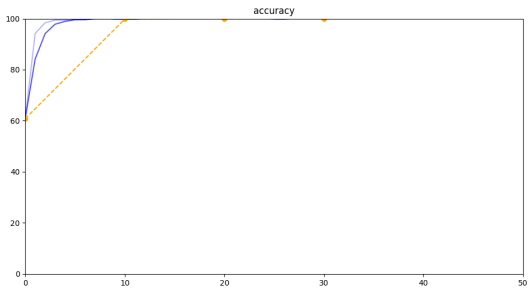
5.3.2 Network Settings



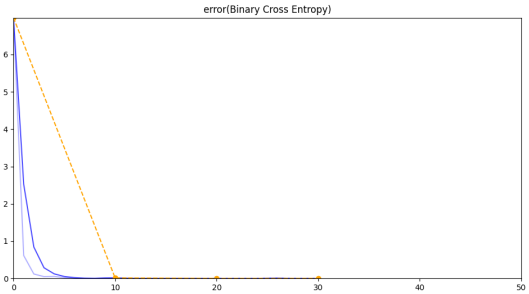
(a) comparsion plot



(b) decision region



(c) accuracy plot



(d) error plot

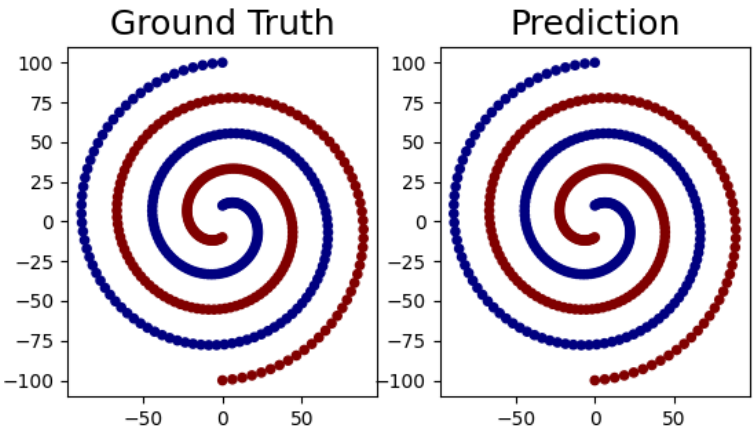
Figure 10: 2 circles

5.4 2 Spirals with Advanced Model

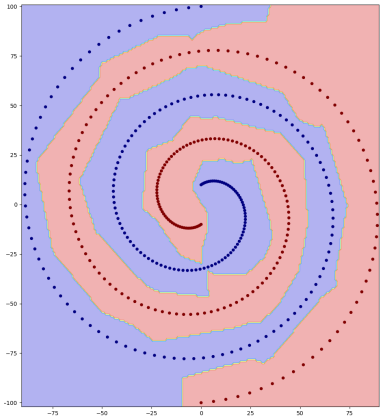
5.4.1 Network Settings

```
# dataset
dataset = NSpirals(2)
dataset_num = 400

# network
network_structure = [
    FullyConnected(2, 10, -1, 1),
    ReLu(0.1),
    FullyConnected(10, 10, -1, 1),
    ReLu(0.1),
    FullyConnected(10, 2, -1, 1),
    SoftMax()
]

# train
epochs = 200
optimizer = Adam(0.01, 0.99)
loss = BinaryCrossEntropy()
```

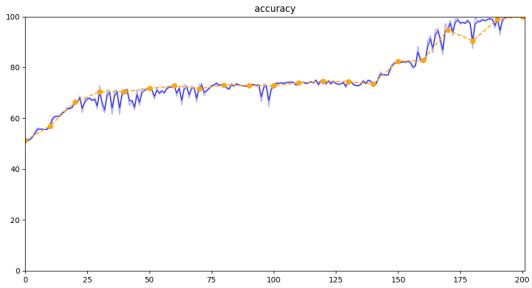
5.4.2 Experimental Result



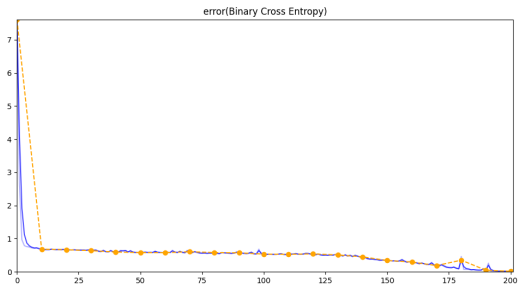
(a) comparsion plot



(b) decision region



(c) accuracy plot



(d) error plot

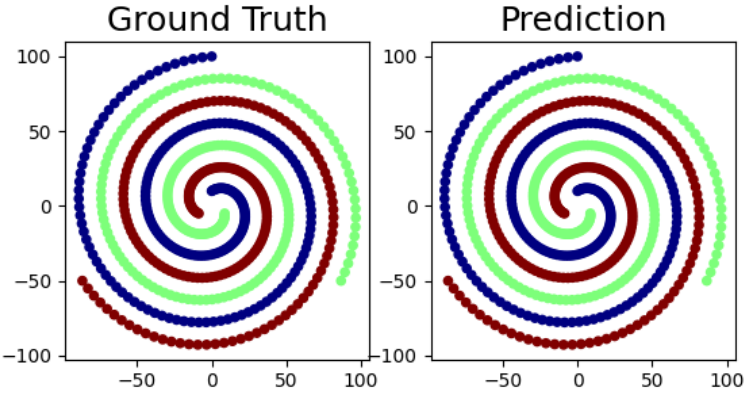
Figure 11: 2 spirals

5.5 3 Spirals with Advanced Model

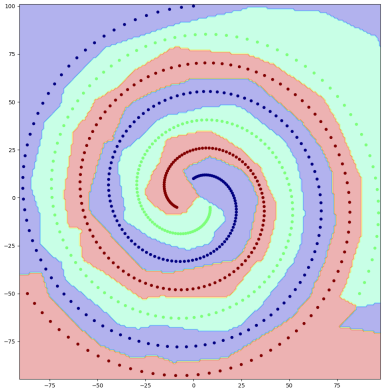
5.5.1 Network Settings

```
# dataset
dataset = NSpirals(3)
dataset_num = 600

# network
network_structure = [
    FullyConnected(2, 15, -1, 1),
    ReLu(0.1),
    FullyConnected(15, 15, -1, 1),
    ReLu(0.1),
    FullyConnected(15, 3, -1, 1),
    SoftMax()
]

# train
epochs = 200
optimizer = Adam(0.01, 0.99)
loss = BinaryCrossEntropy()
```

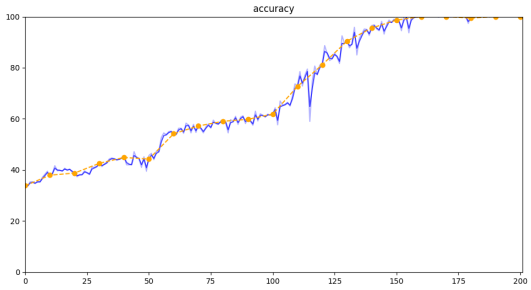
5.5.2 Experimental Result



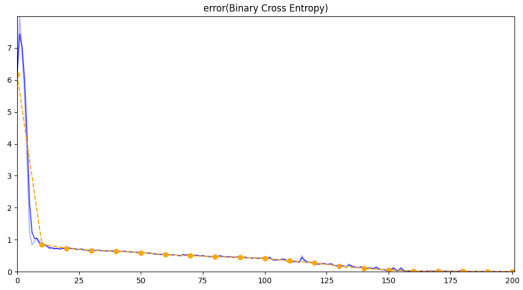
(a) comparsion plot



(b) decision region



(c) accuracy plot



(d) error plot

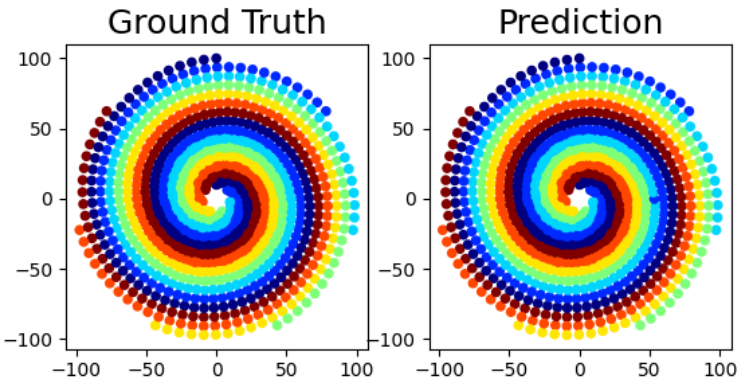
Figure 12: 3 spirals

5.6 7 Spirals with Advanced Model

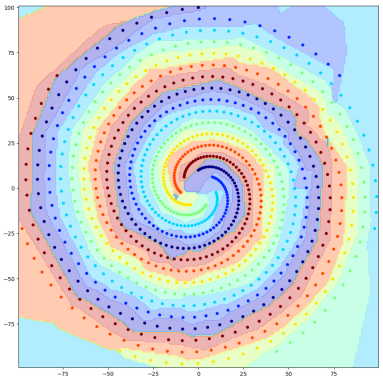
5.6.1 Network Settings

```
# dataset
dataset = NSpirals(7)
dataset_num = 1000

# network
network_structure = [
    FullyConnected(2, 15, -1, 1),
    ReLu(0.1),
    FullyConnected(15, 15, -1, 1),
    ReLu(0.1),
    FullyConnected(15, 7, -1, 1),
    SoftMax()
]

# train
epochs = 2000
optimizer = Adam(0.006, 0.99)
loss = BinaryCrossEntropy()
```

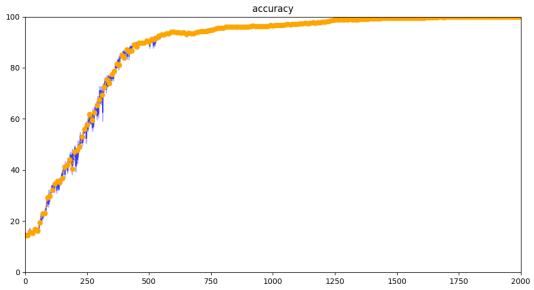
5.6.2 Network Settings



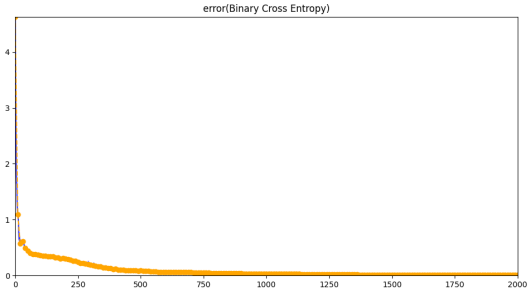
(a) comparsion plot



(b) decision region



(c) accuracy plot



(d) error plot

Figure 13: 7 spirals