Lab1: Back-propagation

311551147 高振群

March 2023

1 Introduction

1.1 Backpropagation

Backpropagation is an algorithm to update parameters in neural network by applying gradient descent. In the big picture, we are going to build a computational graph which represents the relationship of the operations inside the neural network. During training, the gradient of the model parameters are computed by propagating the gradient from the loss function. By doing so, the model parameters can be updated to minimize the loss and learn how to do a certain task.

1.2 Computational graph

The computational graph is a directed graph that represents the relationship of the operations inside the neural network. In lab1, the neural network is composed of one input layer, two hidden layers and one output layer as shown in Figure 1. A single hidden layer is composed of an affine transformation function $f(x) = Wx + b$, where

$$W = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{pmatrix} x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

, and an activation function (sigmoid). They are equivalent to nn.Linear and nn.Sigmoid modules in torch.nn package respectively. Hence, a hidden layer $h(x)$ is of the form

$$h(x) = \sigma(f(x)) \tag{1}$$

where σ is the sigmoid function. The computational graph of the network can be built upon equation (1) and Figure 1 as shown in Figure 2 (for simplicity, let's assume the output layer is an identity function).

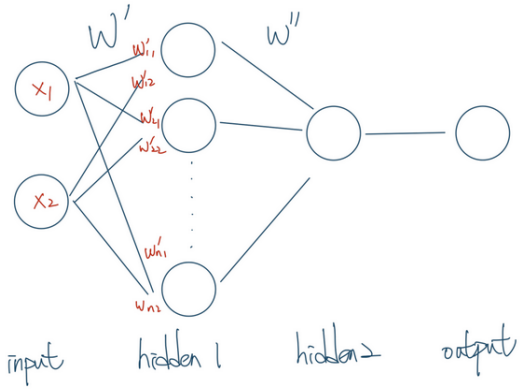

Figure 1: A simplified version of the network architecture, where W' and W'' are the parameters of the model. In my implementation, the hidden layer is more complicated.

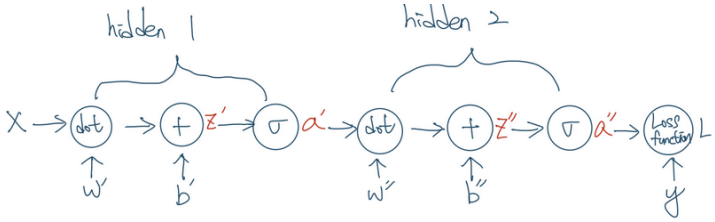

Figure 2: The computational graph of the network in Figure 1.

Note that the dimension of x, W and b depends on the input matrix shape and output matrix shape of the hidden layer. For example, in the first hidden layer $n = 2$ since the input vector $x \in \mathbb{R}^2$ and $m = 10$ if there are 10 neurons in the first hidden layer.

1.3 Gradient propgation

In order to update the parameters in the hidden layers, it is required to compute the gradient of the loss L w.r.t W and b for each hidden layer ——that is, $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. However, computing the gradient directly is somewhat tedious and extremely complicated in some cases. To simplify the computation, we can exploit chain rule and the computational graph (Figure 3). For instance, if we want to update the parameter b'', then we can follow the path highlighted in yellow and calculate the gradient of L w.r.t b'' by

$$\frac{\partial L}{\partial b''} = \frac{\partial z''}{\partial b''} \frac{\partial a''}{\partial z''} \frac{\partial L}{\partial a''}$$

as shown in Figure 4. The other parameters can be updated by following the similar fashion.

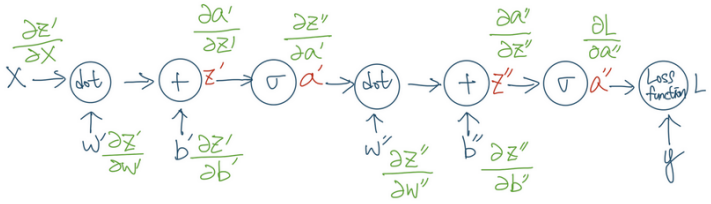

Figure 3: The computational graph and partial derivatives of Figure 2.

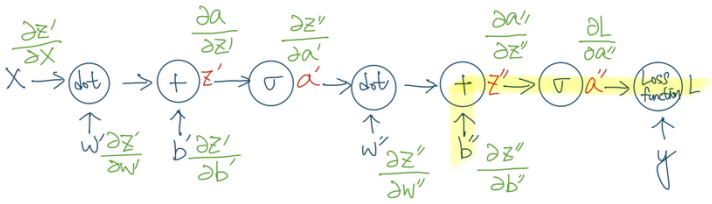

Figure 4: The path to update b''.

The gradient propagation reuses the calculated intermediate gradient and pass them upstream(from loss to model parameters). This method provide a more structured way to tackle the problem and also lower the computational cost, since some of the gradients can be shared by multiple update steps.

1.4 Gradient descent

Finally, once the gradients are computed, they can be used to update the model parameters by applying gradient descent as shown in Algorithm 1.

Data: n the number of epochs, D the dataset, N the neural network, α the learning rate

$epoch \leftarrow n$;
$i \leftarrow 0$;
while $i < epoch$ **do**
  $j \leftarrow 0$;
  while $j < D.size$ **do**
    $data, label \leftarrow D[j]$;
    $\hat{y} \leftarrow N.forward(data)$;
    $N.parameters \leftarrow N.parameters - \alpha \frac{\partial LossFunction(\hat{y}, label)}{\partial N.parameters}$;
    $j \leftarrow j + 1$
  end
  $i \leftarrow i + 1$
end

Algorithm 1: Gradient descent

2 Experiment setups

2.1 Sigmoid functions

The Sigmoid function is implemented in the Sigmoid class as shown in Source Code 1.

Source Code 1: Sigmoid class

```
class Sigmoid:
    def __init__(self, upper_clip=10, lower_clip=-10):
        self.x = None
        self.updatable = False
        self.upper_clip = upper_clip
        self.lower_clip = lower_clip

    def forward(self, x):
        # the clip function here is to prevent overflow in the np.exp operation
        self.x = np.clip(x, self.lower_clip, self.upper_clip)
        return 1 / (1 + np.exp(-self.x))

    def backward(self, downstream_grad):
        return np.multiply(downstream_grad, self.derivative_sigmoid())

    def derivative_sigmoid(self):
        # note that all the operations are elementwise
        x = self.forward(self.x)
        return np.multiply(x, 1 - x)
```

More specifically, the sigmoid function call is the forward function call in Source Code 1, which is equivalent to the math expression

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

.

2.2 Neural network

The neural network is implemented in the Net class as shown in Source Code 2.

Source Code 2: Net class

```
class Net:
    def __init__(self, layers, lr):
        self.layers = layers
        self.lr = lr

    def forward(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def backward(self, pred_grad):
        downstream_grad = pred_grad
        for layer in reversed(self.layers):
            downstream_grad = layer.backward(lr=self.lr,
            ↪  downstream_grad=downstream_grad) if
            ↪  layer.updatable else
            ↪  layer.backward(downstream_grad=downstream_grad)
```

The forward function call takes an input x, passes it through all the layers and finally returns the output. The layers is a list of linear functions and non-linear functions. More specifically, the linear function is the Linear class in Source code 3 and the non-linear function is the Sigmoid class in Source code 1.

Source Code 3: Hidden layers

```
class Linear:
    def __init__(self, in_features, out_features):
        self.w = np.random.randn(out_features, in_features)
        self.b = np.random.randn(out_features, 1)
        self.x = None
        self.updatable = True

    def forward(self, x):
        self.x = x
        return np.dot(self.w, self.x) + self.b
```

```
    def backward(self, lr, downstream_grad):
        w_grad = np.dot(downstream_grad, self.x.T)
        b_grad = downstream_grad
        x_grad = np.dot(self.w.T, downstream_grad)


        self.w -= lr * w_grad
        self.b -= lr * b_grad

        return x_grad
```

2.3 Backpropagation

The backpropagation is implemented in the backward function in Source Code 1, 2 and 3. The details can be referred to Figure 3 and [1]. The loss function used in this lab is the mean square error function and the code of it is in Source Code 4.

Source Code 4: Loss function

```
class MSELoss:
    def __init__(self):
        self.pred = None
        self.y = None
        self.size = None

    def forward(self, pred, y):
        # note that the order of pred and y need to be followed
        ↪  strictly, otherwise self.pred and self.y will save
        ↪  the wrong data
        self.pred, self.y, self.size = pred, y, y.size
        return np.mean((self.pred - self.y) ** 2)

    def backward(self):
        # note that there is no downstream_grad of the loss
        ↪  function, since it is the last node in the
        ↪  computational graph
        return (self.pred - self.y) / self.size
```

The first downstream gradient is computed by the backward function call of the MSELoss class and propagate all the way to the parameters of the network via the backward function call of the Net class in Source Code 2.

The code snippet of the entire training process that concatenate the forward propagation and backward propagation is shown in Source Code 5.

Source Code 5: training

```
def main(args):
    task = generate_linear if args.task == 'linear' else
    ↪  generate_XOR_easy
    x, y = task()

    # note that sometimes it might perform horribly on XOR task
    ↪  if the number of neurons are not large enough
    hidden_layers_features = [256, 64, 32, 16]
    layers = [
        module.Linear(in_features=2,
        ↪  out_features=hidden_layers_features[0]),
        module.Sigmoid(),
        module.Linear(in_features=hidden_layers_features[0],
        ↪  out_features=hidden_layers_features[1]),
        module.Sigmoid(),
        module.Linear(in_features=hidden_layers_features[1],
        ↪  out_features=1),
    ]
    net = module.Net(layers, lr=args.lr)
    criterion = module.MSELoss()

    epochs = 1000
    eps = 1e-3
    prev_total_loss = 0
    loss_history = []


    for i in range(epochs):
        total_loss = 0
        for data, label in zip(x, y):
            pred = net.forward(np.expand_dims(data, axis=0).T)
            total_loss += criterion.forward(pred, label)
            pred_grad = criterion.backward()
            net.backward(pred_grad)
        print(f'epoch {i} loss : {total_loss}')
        loss_history.append(total_loss)
        if np.abs(prev_total_loss - total_loss) < eps:
            break

    final_pred = []
    for data in x:
        data = net.forward(np.expand_dims(data, axis=0).T)
        final_pred.append(threshold(data))
```

```
show_result(x, y, final_pred)
```

Note that the default learning rate is 10^{-2}.

3 Results of your testing

The learning rate is set to the default learning rate and the network structure is the same as Source Code 5 if there is no specification.

3.1 Screenshot and comparison figure

The result of the linear problem and the XOR problem are shown in Figure 5 and Figure 6, respectively.

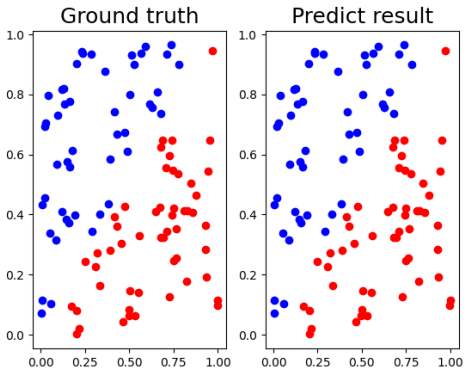

Figure 5: Result of linear task

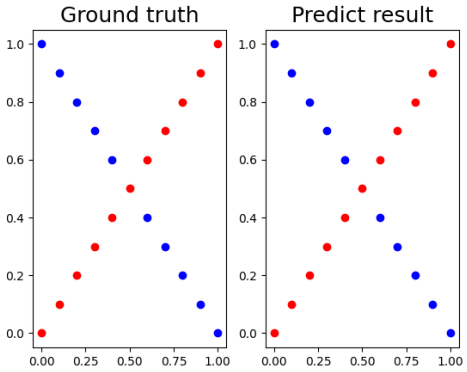

Figure 6: Result of XOR task

3.2 Show the accuracy of your prediction

The accuracy of the tasks are shown in Figure 7 and Figure 8.

```
Iter97 |         Ground truth: 0|        prediction: [[0.00252927]] |
Iter98 |         Ground truth: 0|        prediction: [[0.03283009]] |
Iter99 |         Ground truth: 1|        prediction: [[0.94056871]] |
loss=1.7389261342654876 accuracy=99.0%
```

Figure 7: Accuracy of linear problem

```
Iter17 |         Ground truth: 0|        prediction: [[-0.02785223]] |
Iter18 |         Ground truth: 1|        prediction: [[0.99477087]] |
Iter19 |         Ground truth: 0|        prediction: [[0.01655169]] |
Iter20 |         Ground truth: 1|        prediction: [[0.98691955]] |
loss=0.39793399691340614 accuracy=100.0%
```

Figure 8: Accuracy of XOR problem

3.3 Learning curve (loss, epoch curve)

The loss, epoch curve of linear problem and xor problem are shown in Figure 9 and Figure 10, respectively.

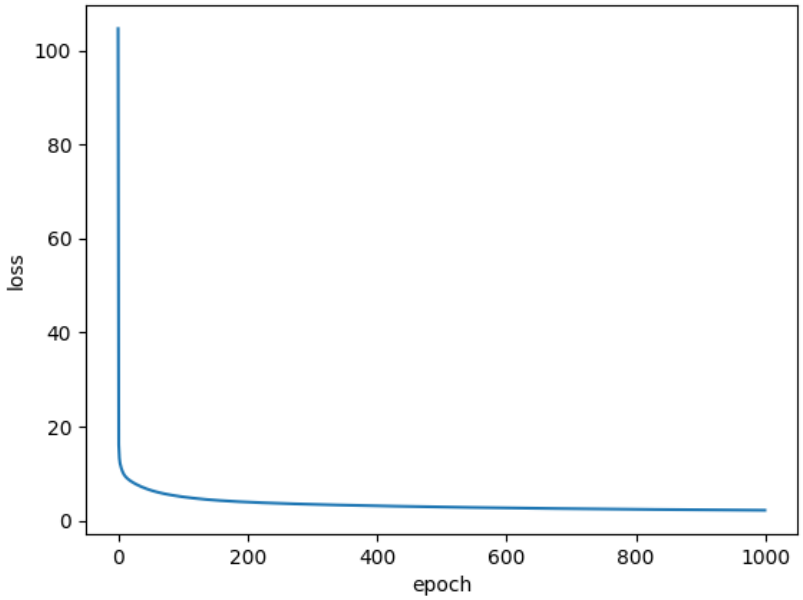

Figure 9: Loss to epoch curve of the linear problem

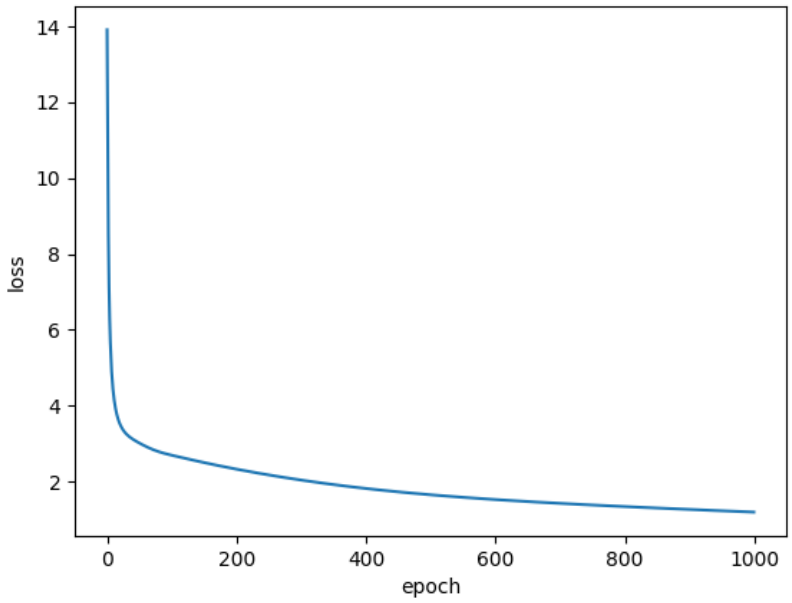

Figure 10: Loss to epoch curve of the XOR problem

Note that the training will stop at the 1000th epoch or the loss is already converge.

3.4 Anything you want to present

In this subsection, I want to show whether this approach lead us to severe overfitting. The dataset is divide to 8:2, where the training set is 8, the testing set is 2. The result are shown in Figure 11 and 12.

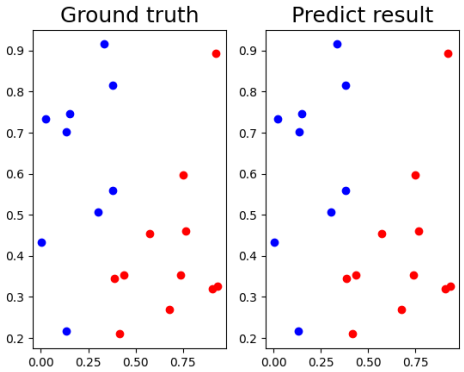

Figure 11: Result of testing set in linear problem

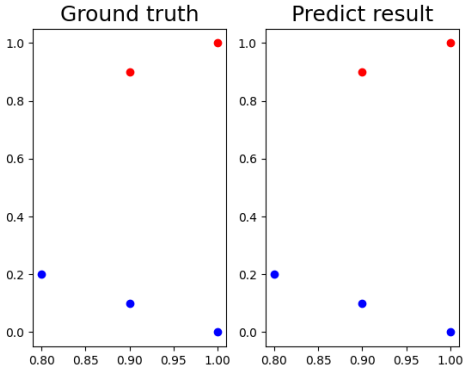

Figure 12: Result of training set in XOR problem

The result seems to be okay :).

4 Discussion

4.1 Try different learning rates

The result of this subsection are shown in Table 1 and Table 2. However, the result is not stable. This might due to the random initialization of the model parameters and the training epoch is not large enough.

learning rate	loss	accuracy
1	46	0.54
10^{-1}	1.80	0.99
10^{-2}	1.73	0.99
10^{-3}	3.14	0.99
10^{-4}	6.70	0.98

Table 1: Result of changing the learning rate on the Linear problem

learning rate	loss	accuracy
1	11.0	0.47
10^{-1}	0.03	1
10^{-2}	0.39	1
10^{-3}	1.12	0.90
10^{-4}	3.04	0.85

Table 2: Result of changing the learning rate on the XOR problem

4.2 Try different numbers of hidden units

In this subsection, 4 types of network are tested, the details are shown in Table 3 and Table 4. The hidden_layers_features is set to [256, 64, 32, 16] in Source Code 5 and corresponding layers are added to the net.

# of hidden layers	loss	accuracy
1	4.30	0.99
2(default)	1.73	0.99
3	1.17	1
4	0.63	1

Table 3: Result of changing the number of layers on the linear problem

# of hidden layers	loss	accuracy
1	2.78	0.76
2(default)	0.61	1
3	0.32	1
4	0.31	1

Table 4: Result of changing the number of layers on the XOR problem

The result shows that the deeper the network, the lower the loss.

4.3 Try without activation functions

The activation functions in the neural networks enable it to become a non-linear function approximator of some underlying functions. When the activation functions are removed, the output will simply be a linear combination of the input features no matter how many layers are added to the network. Thus, it is conceptually equivalent to perform a single affine transformation on the input features. In the linear task, the result is fine since the data are linear-seperable (Figure 13). As for the XOR problem, the network suffered from the lack of non-linearity since the data are not linear-seperable (Figure 14).

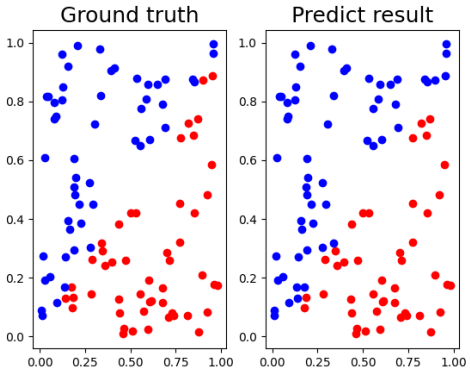

Figure 13: Result of removing activation functions in linear problem

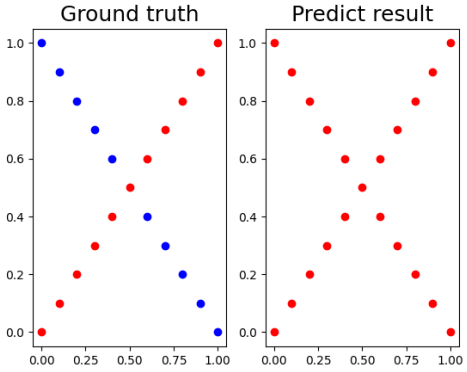

Figure 14: Result of removing activation functions in XOR problem

4.4 Anything you want to share

In this section, we do some simple experiments to understand how each factor can affect the result of the network. One thing to notice is that when setting extremely large learning rate, the network might fail to work simply because of overflow.

In section 4.3, I did an experiment to show that when applying a feature map $\phi : \mathbb{R}^2 \to \mathbb{R}^5$, where

$$\phi(x) = \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{pmatrix}$$

, the network can still carry out a decent performance (0.90 accuracy) even with the absence of activation functions, since the feature map itself is a non-linear function.

5 Extra

5.1 Implement different activation functions

In this part, two other activation functions were implemented, which are LeakyReLU and hyperbolic tangent. The implementation is in the source code.

5.1.1 LeakyReLU

The LeakyReLU function is

$$LeakyReLU(x; s) = \begin{cases} x & \text{, if } x \geq 0 \\ xs & \text{, otherwise} \end{cases} \tag{2}$$

, where s is a fixed, not updatable parameter.

Given equation (2), we can derive the derivative of LeakyReLu $w.r.t$ x, which is

$$\frac{\partial LeakyReLU(x; s)}{\partial x} = \begin{cases} 1 & \text{, if } x \geq 0 \\ s & \text{, otherwise} \end{cases}$$

. Note that since LeakyReLU is not differentiable at x = 0 $w.r.t$ x, I directly let the derivative be 1 when x = 0.

5.1.2 Hyperbolic tangent

The Hyperbolic tangent function, Tanh, is

$$Tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3}$$

. Given equation (3), we can derive the derivative of Tanh $w.r.t$ x, which is

$$\frac{\partial Tanh(x)}{\partial x} = sech^2(x)$$

5.1.3 Results

Here, we briefly compare the loss and accuracy of three activation functions, the results are shown in Table 5.

activation function	loss	accuracy
Sigmoid	1.73	0.99
LeakyReLU	3.68	0.99
Tanh	1.03	1

Table 5: Result of using differenent activation function on linear problem

6 Reference

[1] Youtube: Neural Network from Scratch | Mathematics & Python Code