DP lab1 report

309553005 You-Jia Lai

March 2021

1 Introduction

Backpropagation is the process of tuning a neural network's weights to better the prediction accuracy. There are two directions in which information flows in a neural network
Forward propagation: is when data goes into the neural network and out pops a prediction
Backpropagation: the process of calculating the gradient and adjusting the weights by looking at the difference between prediction and the actual result

2 Experiment setups

2.1 Sigmoid function

I use the document of homework assignment to implement my Sigmoid function and derivative Sigmoid function

2.2 Neural Network

I create two classes Net and Layer to build my neural network. Class Net contains multiple class Layer. Class Net triggers the forward, backward and optimization of each layer
Class layer implements the detail of forward, backward and optimization. The implement details are follows three points:

- Forward
  - Using input data multiply the weight matrix and pass it to the activation function to generate the layer output
- Backward
  - Use the forward result to calculate the gradient of the layer
- Optimization
  - Use the gradient of the backward result to perform gradient descent to update the weights of the layer

2.3 Backpropagation

$z \to h \to y \to C$
Assume that $C = L(y, \hat{y}), h = zw, y = \sigma(h)$
My loss function is MSE

$\frac{\partial C}{\partial h} = \frac{\partial C}{\partial y} \frac{\partial y}{\partial h}$
We can get $\frac{\partial C}{\partial y}$, $\frac{\partial y}{\partial h}$ by derivative loss function and derivative activation function

Finally, we can get $\frac{\partial C}{\partial z}$
$\frac{\partial C}{\partial z} = \frac{\partial C}{\partial h} \frac{\partial h}{\partial z}$
And $\frac{\partial h}{\partial z}$ is w because of $h = zw$

About updating weights, we need to calculate $\frac{\partial C}{\partial w}$
$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial z} \frac{\partial z}{\partial w}$

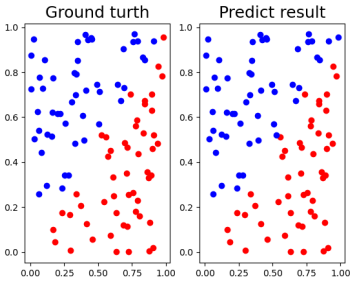
After we calculate $\frac{\partial C}{\partial w}$, we can use this gradient to update our weights by gradient descent

3 Results of your testing

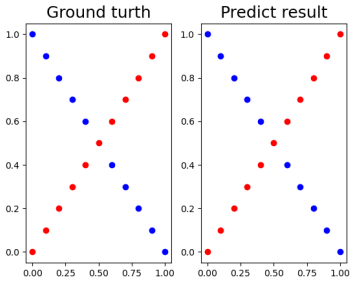
3.1 Predict result

Linear prediction: [[0.99286082] [0.99040986] [0.73861969] [0.06468437] [0.99002428]
[0.06433415] [0.01900579] [0.03774367] [0.99241762] [0.81977004] [0.01145097]
[0.74353096] [0.02020469] [0.82964923] [0.24367426] [0.88859872] [0.93771357]
[0.14108857] [0.82117384] [0.0079034] [0.95227579] [0.9107485] [0.01023311]
[0.96713559] [0.95044667] [0.98938695] [0.01232775] [0.40369691] [0.93382019]
[0.82067143] [0.26230033] [0.01692034] [0.02214629] [0.99419212] [0.01148171]
[0.35949569] [0.01630143] [0.98545431] [0.86862931] [0.05680751] [0.04157128]
[0.99375966] [0.0991941] [0.94712522] [0.73683362] [0.01024209] [0.66529788]
[0.59911252] [0.85488026] [0.00814541] [0.03242997] [0.45776791] [0.06122913]
[0.05399566] [0.82551726] [0.01552833] [0.0116595] [0.99361967] [0.85192377]
[0.20343592] [0.68792596] [0.09378233] [0.98252078] [0.01898454] [0.99366202]
[0.99192953] [0.96024049] [0.86115444] [0.88160108] [0.99294836] [0.0278813]
[0.10999321] [0.64031532] [0.01025032] [0.49547356] [0.99482766] [0.99186786]
[0.99370836] [0.06744555] [0.37294352] [0.86955274] [0.07845466] [0.03248406]
[0.24409132] [0.6328028] [0.78437287] [0.4139298] [0.98547117] [0.67800113]
[0.64663501] [0.38933359] [0.98372244] [0.99385308] [0.01154365] [0.95432803]
[0.98694771] [0.98897461] [0.21158325] [0.02957553] [0.0625814]]

XOR prediction: [[0.12470104] [0.95452795] [0.13788888] [0.94796598] [0.14960498]
[0.93465629] [0.15872749] [0.8917712] [0.16453603] [0.58634419] [0.16684878]
[0.16599341] [0.59158785] [0.16265465] [0.92108184] [0.15767362] [0.96134306]
[0.15186691] [0.9710286] [0.1459059] [0.97528621]]



(a) Linear



(b) XOR

Figure 1: Comparison figure

3.2 Accuracy

The figure of accuracy is figure 2

```
epoch 240 loss :0.041381
epoch 250 loss :0.039771
epoch 260 loss :0.038281
epoch 270 loss :0.036899
epoch 280 loss :0.035613
epoch 290 loss :0.034413
epoch 300 loss :0.033291
epoch 310 loss :0.032238
epoch 320 loss :0.031250
epoch 330 loss :0.030320
Converge
Accuracy: 100.00%
```

(a) Linear

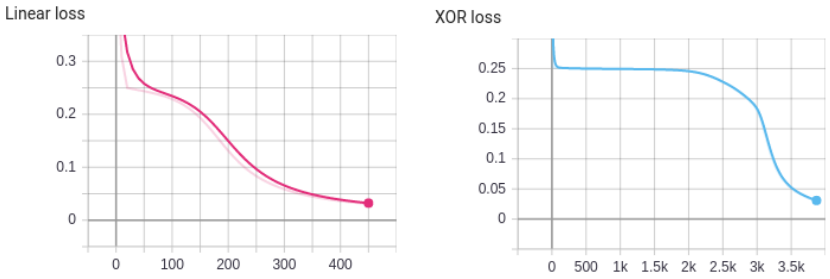
```
epoch 2530 loss :0.039951
epoch 2540 loss :0.039170
epoch 2550 loss :0.038393
epoch 2560 loss :0.037620
epoch 2570 loss :0.036851
epoch 2580 loss :0.036087
epoch 2590 loss :0.035327
epoch 2600 loss :0.034573
epoch 2610 loss :0.033824
epoch 2620 loss :0.033082
epoch 2630 loss :0.032346
epoch 2640 loss :0.031616
epoch 2650 loss :0.030894
epoch 2660 loss :0.030180
Converge
Accuracy: 100.00%
```

(b) XOR

Figure 2: Accuracy

3.3 Loss curve

The figure of loss curve is figure 3



(a) Linear (b) XOR

Figure 3: Loss curve

4 Discussion

4.1 Different learning rate

When the learning rate is higher, the speed of converge will be faster. But if the learning rate is too high, the training may not converge or its accuracy will be lower.

4.2 Different numbers of hidden units

I found the higher number of hidden units. The speed of converge will be faster. I think the reason is the capability of model increases

4.3 Without activation functions

Non-linear activation functions between the layers allows for the network to solve a larger variety of problems. Hence, Without activation function the performance of network is bad

4.4 Bias

Adding bias in the network will improve the speed of converge. I think the reason is the decision boundary is more flexible than the network without bias

5 Extra

5.1 Different optimizers

Besides the normal gradient descent, I implement the momentum method. And use parse arguement to control whether use the momentum

```
def optimize(self,lr):  # Gradient Descent w-=lr*dC/dw dC/dw = dC/d
    gradient=self.x.T @ self.backward_gradient
    if args.M:
        self.w = self.w - lr * gradient + 0.9 * self.last_graident
        self.last_graident = gradient
    else:
        self.w -= lr * gradient
```

Figure 4: Momentum

5.2 Different activation

Besides the Sigmoid function, I implement the tanh and its derivative function

```
def tanh(x):
    return (np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))

def derivative_tanh(x):
    return 1.-tanh(x)**2
```

Figure 5: Tanh