1. Introduction

In this lab, I implement a simple neural network that is similar to pytorch/TensorFlow by using NumPy. I use layer structures to implement it, so we can stack different kinds of layer to construct a neural network like in pytorch. In the implementation of fully-connected layer, I allow users to decide the input size and output size and whether it has a bias. In activation functions part, I implement sigmoid and relu function. Besides, in backpropagation, I implement 4 kinds of optimizers, they are SGD, momentum, adagrad, and Adam.

2. Experiment setups

A. Sigmoid function

```
def sigmoid(self, x:np.array):
    return 1.0 / (1.0 + np.exp(-x))
```

This is the implementation of sigmoid function, following the instruction in specs.

```
def backward(self, dy:np.array, opti:optimizer):
    dx = self.sigmoid(self.inputs) * (1 - self.sigmoid(self.inputs)) * dy
    return dx
```

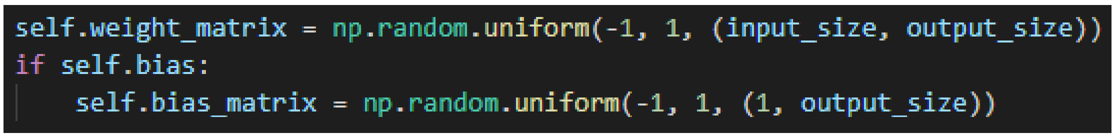
This is the back propagation of sigmoid function, using the following formula.

$$y = \sigma(x), \frac{dL}{dx} = \frac{dy}{dx} \times \frac{dL}{dy} = \sigma'(x)\frac{dL}{dy} = \sigma(x)(1-\sigma(x))\frac{dL}{dy}$$

B. Neural network

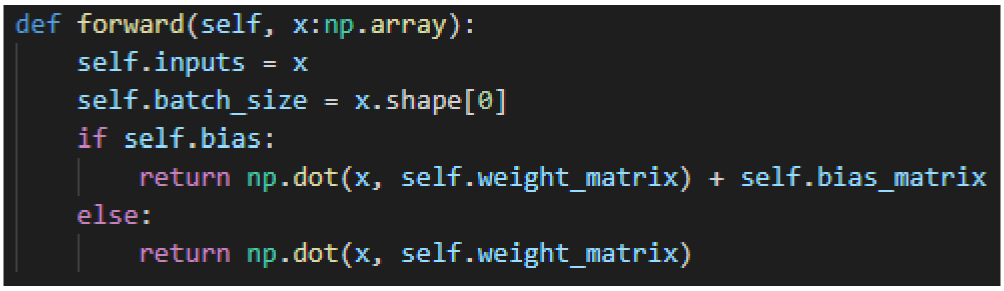
```
layers = [
    NN.FC_layer(2, 32, True),
    NN.relu(),
    NN.FC_layer(32, 32, True),
    NN.relu(),
    NN.FC_layer(32, 1, True),
    NN.sigmoid()
]
DNN = NN.myNN(layers)
```

In my neural network, I use 2 hidden layer that has 32 units with bias. In the hidden layer, I use relu as their activation function. In the output layer, I use sigmoid function as its activation function to produce a probability value between 0 to 1.



```python
self.weight_matrix = np.random.uniform(-1, 1, (input_size, output_size))
if self.bias:
    self.bias_matrix = np.random.uniform(-1, 1, (1, output_size))
```

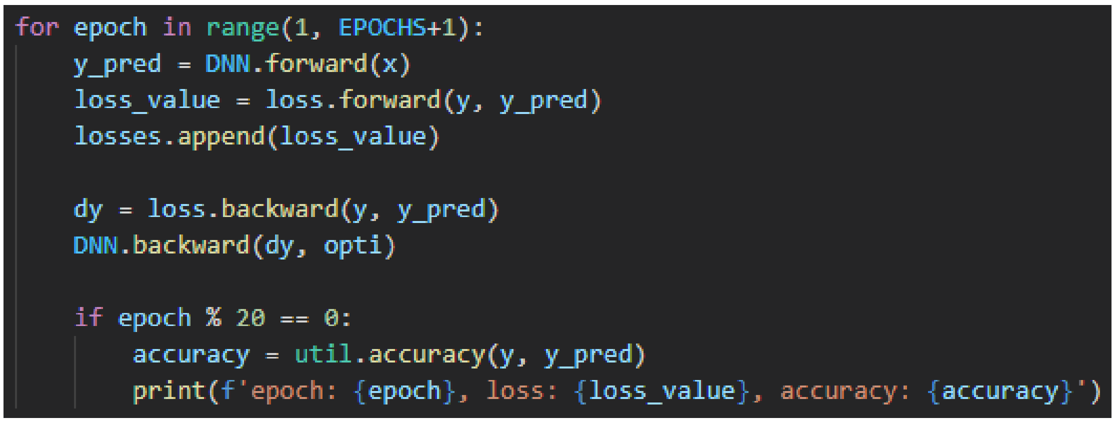
When constructing the fully-connected layer, I initialize it with random weights.



```python
def forward(self, x:np.array):
    self.inputs = x
    self.batch_size = x.shape[0]
    if self.bias:
        return np.dot(x, self.weight_matrix) + self.bias_matrix
    else:
        return np.dot(x, self.weight_matrix)
```

In forward propagation, I would save the input data first, since we need this information to calculate the gradient in the back propagation. Then calculating the output value, it's just a simple matrix multiplication.

C. Backpropagation

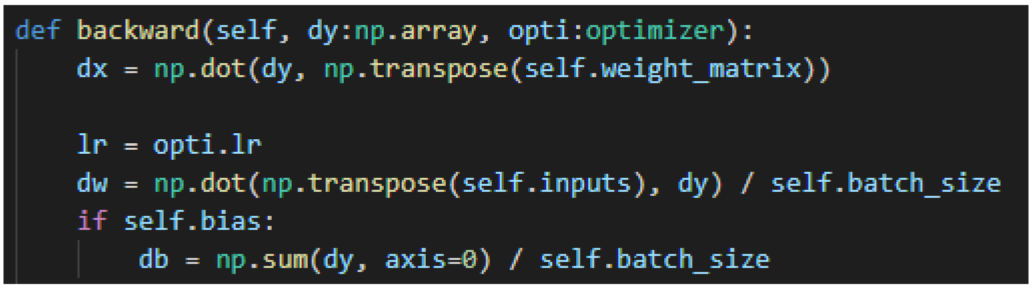


```python
for epoch in range(1, EPOCHS+1):
    y_pred = DNN.forward(x)
    loss_value = loss.forward(y, y_pred)
    losses.append(loss_value)

    dy = loss.backward(y, y_pred)
    DNN.backward(dy, opti)

    if epoch % 20 == 0:
        accuracy = util.accuracy(y, y_pred)
        print(f'epoch: {epoch}, loss: {loss_value}, accuracy: {accuracy}')
```

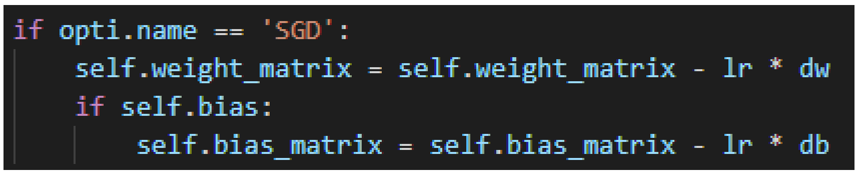
During training of each epoch, I first feed input data 'x' into my neural network and calculate its loss value. After that, I would first calculate the derivative of y $\frac{dL}{dy}$. Then calculate the neural network's back propagation.



```python
def backward(self, dy:np.array, opti:optimizer):
    dx = np.dot(dy, np.transpose(self.weight_matrix))

    lr = opti.lr
    dw = np.dot(np.transpose(self.inputs), dy) / self.batch_size
    if self.bias:
        db = np.sum(dy, axis=0) / self.batch_size
```

In the back propagation of fully connected layer, I first calculate the derivative of x $\frac{dL}{dx} = \frac{dy}{dx} \times \frac{dL}{dy} = w \times \frac{dL}{dy}$. In the end of back propagation this $\frac{dL}{dx}$ needs to propagate to the previous layer to calculate its gradient. Then calculate the derivative of w $\frac{dL}{dw} = \frac{dy}{dw} \times \frac{dL}{dy} = x \times \frac{dL}{dy}$. As for the bias, its derivative is $\frac{dL}{db} = \frac{dy}{db} \times \frac{dL}{dy} = 1 \times \frac{dL}{dy} = \frac{dL}{dy}$. Since we fed numbers of input data at the same time, the derivatives need to be divided by the number of input data.

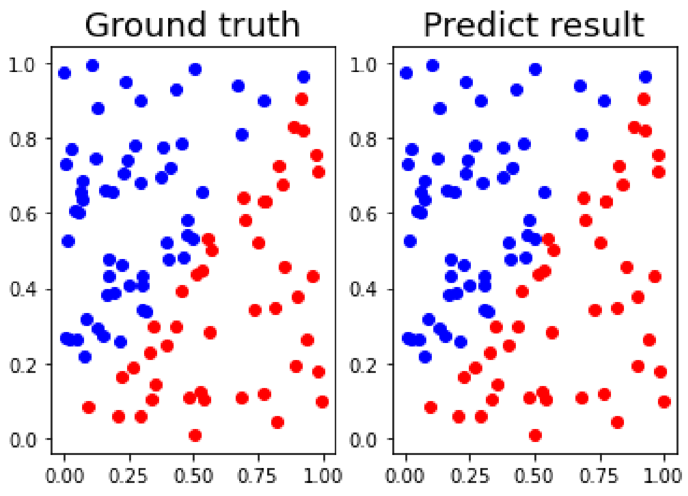


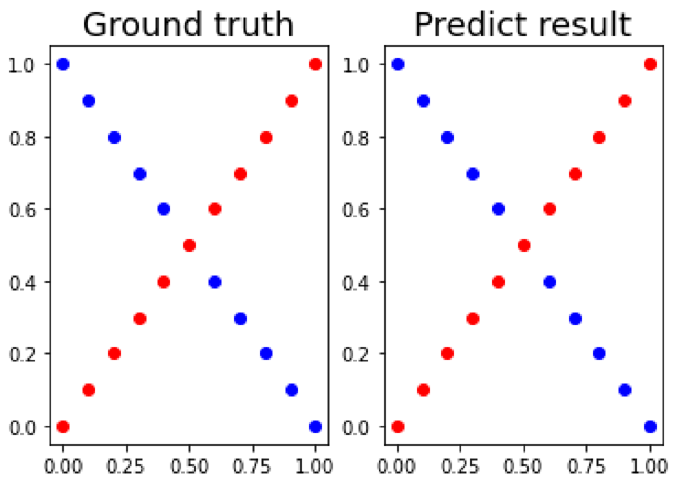
```
if opti.name == 'SGD':
    self.weight_matrix = self.weight_matrix - lr * dw
    if self.bias:
        self.bias_matrix = self.bias_matrix - lr * db
```

After calculating the gradients, we need to update the weight value. In this part, I use stochastic gradient decent $w = w - \eta \frac{dL}{dw}$ to update it.

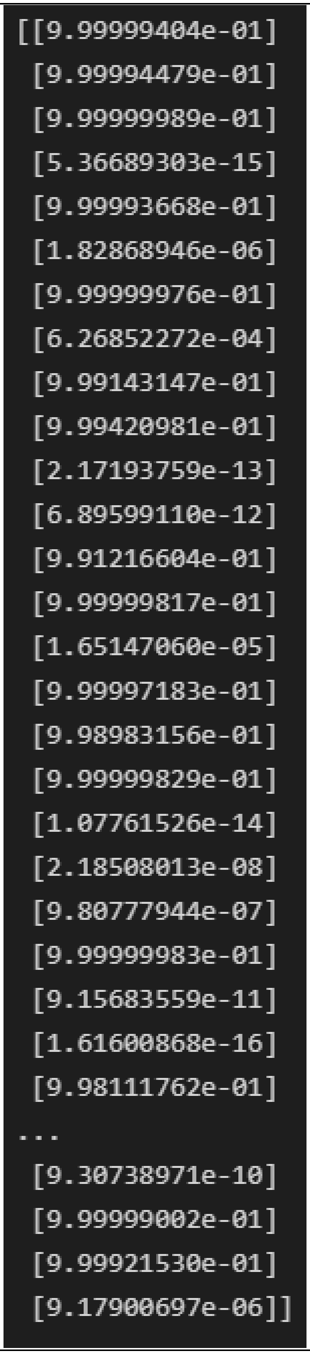
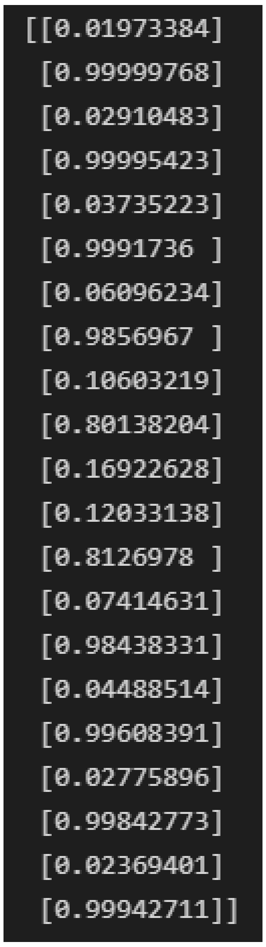
3. Results of your testing

A. Screenshot and comparison figure





B. Show the accuracy of your prediction

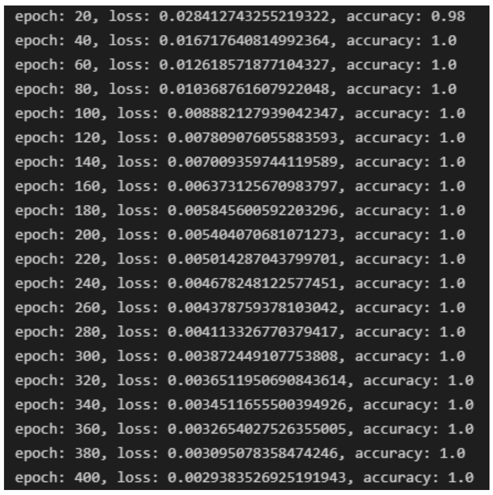
Linear (accuracy 100%)	XOR (accuracy 100%)
	

```
[[9.99999404e-01]
 [9.99994479e-01]
 [9.99999989e-01]
 [5.36689303e-15]
 [9.99993668e-01]
 [1.82868946e-06]
 [9.99999976e-01]
 [6.26852272e-04]
 [9.99143147e-01]
 [9.99420981e-01]
 [2.17193759e-13]
 [6.89599110e-12]
 [9.91216604e-01]
 [9.99999817e-01]
 [1.65147060e-05]
 [9.99997183e-01]
 [9.98983156e-01]
 [9.99999829e-01]
 [1.07761526e-14]
 [2.18508013e-08]
 [9.80777944e-07]
 [9.99999983e-01]
 [9.15683559e-11]
 [1.61600868e-16]
 [9.98111762e-01]
...
 [9.30738971e-10]
 [9.99999002e-01]
 [9.99921530e-01]
 [9.17900697e-06]]
```

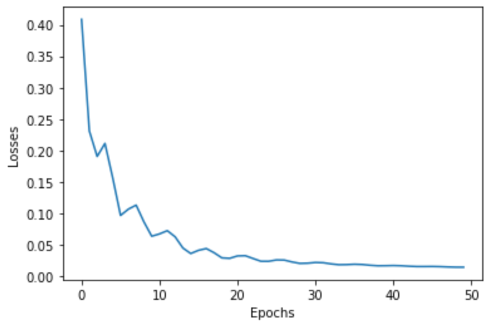
```
[[0.01973384]
 [0.99999768]
 [0.02910483]
 [0.99995423]
 [0.03735223]
 [0.9991736 ]
 [0.06096234]
 [0.9856967 ]
 [0.10603219]
 [0.80138204]
 [0.16922628]
 [0.12033138]
 [0.8126978 ]
 [0.07414631]
 [0.98438331]
 [0.04488514]
 [0.99608391]
 [0.02775896]
 [0.99842773]
 [0.02369401]
 [0.99942711]]
```

C. Learning curve (loss, epoch curve)

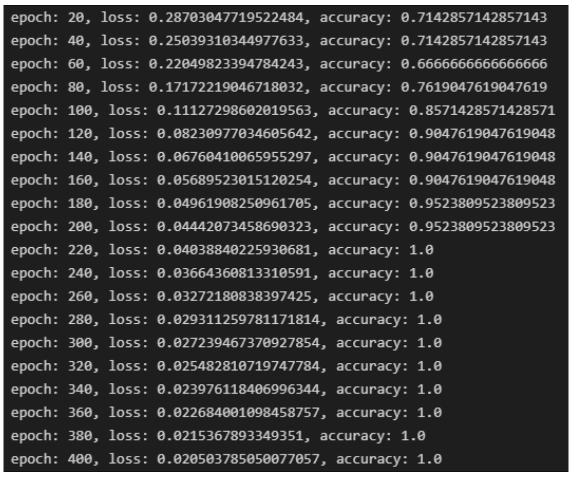
Linear



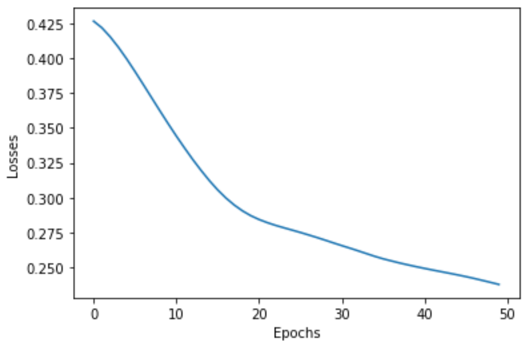
```
epoch: 20, loss: 0.028412743255219322, accuracy: 0.98
epoch: 40, loss: 0.016717640814992364, accuracy: 1.0
epoch: 60, loss: 0.012618571877104327, accuracy: 1.0
epoch: 80, loss: 0.010368761607922048, accuracy: 1.0
epoch: 100, loss: 0.008882127939042347, accuracy: 1.0
epoch: 120, loss: 0.007809076055883593, accuracy: 1.0
epoch: 140, loss: 0.007009359744119589, accuracy: 1.0
epoch: 160, loss: 0.006373125670983797, accuracy: 1.0
epoch: 180, loss: 0.005845600592203296, accuracy: 1.0
epoch: 200, loss: 0.005404070681071273, accuracy: 1.0
epoch: 220, loss: 0.005014287043799701, accuracy: 1.0
epoch: 240, loss: 0.004678248122577451, accuracy: 1.0
epoch: 260, loss: 0.004378759378103042, accuracy: 1.0
epoch: 280, loss: 0.004113326770379417, accuracy: 1.0
epoch: 300, loss: 0.003872449107753808, accuracy: 1.0
epoch: 320, loss: 0.0036511950690843614, accuracy: 1.0
epoch: 340, loss: 0.0034511655500394926, accuracy: 1.0
epoch: 360, loss: 0.0032654027526355005, accuracy: 1.0
epoch: 380, loss: 0.003095078358474246, accuracy: 1.0
epoch: 400, loss: 0.0029383526925191943, accuracy: 1.0
```



XOR

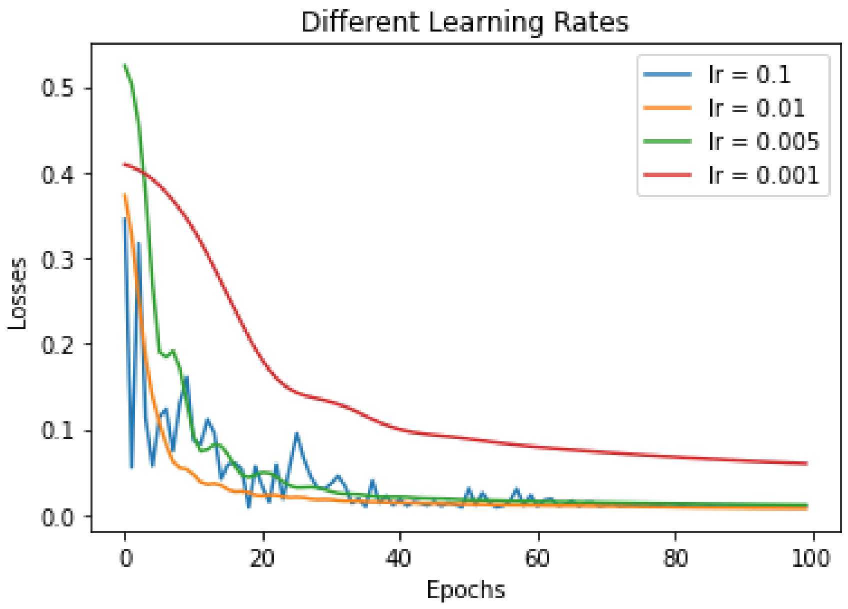


```
epoch: 20, loss: 0.28703047719522484, accuracy: 0.7142857142857143
epoch: 40, loss: 0.25039310344977633, accuracy: 0.7142857142857143
epoch: 60, loss: 0.22049823394784243, accuracy: 0.6666666666666666
epoch: 80, loss: 0.17172219046718032, accuracy: 0.7619047619047619
epoch: 100, loss: 0.11127298602019563, accuracy: 0.8571428571428571
epoch: 120, loss: 0.08230977034605642, accuracy: 0.9047619047619048
epoch: 140, loss: 0.06760410065955297, accuracy: 0.9047619047619048
epoch: 160, loss: 0.05689523015120254, accuracy: 0.9047619047619048
epoch: 180, loss: 0.04961908250961705, accuracy: 0.9523809523809523
epoch: 200, loss: 0.04442073458690323, accuracy: 0.9523809523809523
epoch: 220, loss: 0.04038840225930681, accuracy: 1.0
epoch: 240, loss: 0.03664360813310591, accuracy: 1.0
epoch: 260, loss: 0.03272180838397425, accuracy: 1.0
epoch: 280, loss: 0.029311259781171814, accuracy: 1.0
epoch: 300, loss: 0.027239467370927854, accuracy: 1.0
epoch: 320, loss: 0.025482810719747784, accuracy: 1.0
epoch: 340, loss: 0.023976118406996344, accuracy: 1.0
epoch: 360, loss: 0.022684001098458757, accuracy: 1.0
epoch: 380, loss: 0.0215367893349351, accuracy: 1.0
epoch: 400, loss: 0.020503785050077057, accuracy: 1.0
```



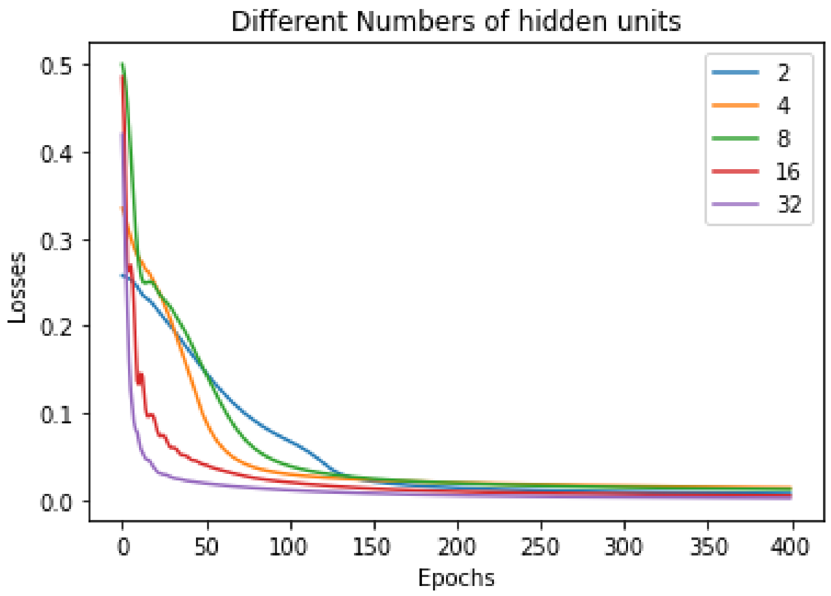
4. Discussion

A. Try different learning rates



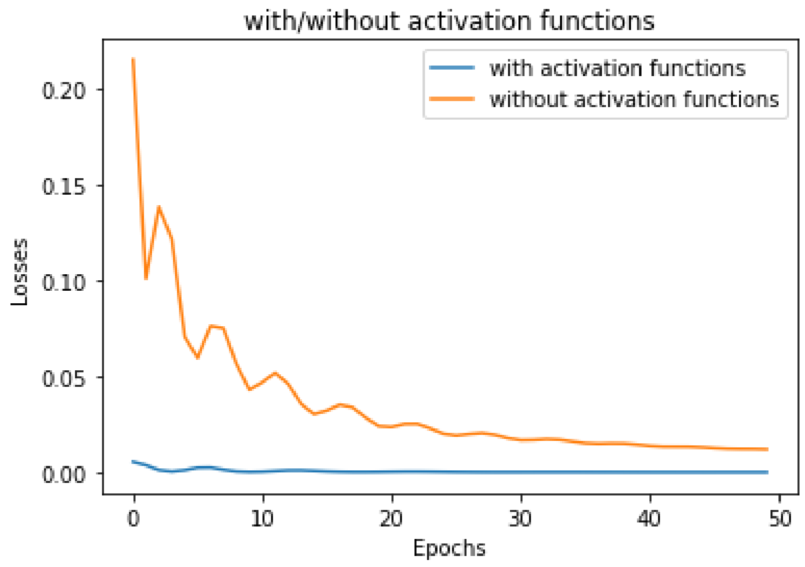
In this part, we can see that when the learning rate is too high (0.1), its loss value would be up and down. As for low learning rate (0.001) its loss value decreases very slowly. In this figure, we can find that 0.01 is a suitable learning rate for this case.

B. Try different numbers of hidden units



In this part, I try different numbers of hidden units from 2 to 32. As you can see the speed of converge has the same trend of the number of hidden units.

C. Try without activation functions



In this part, we found that the speed of converge between with and without activation functions are very huge. With activation function, the neural network can converge in less than 10 epochs. But it need 30 to 40 to converge without activation functions.

5. Extra

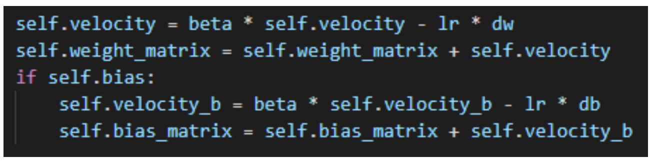
A. Implement different optimizers

In this part, I implement the momentum, adagrad, and adam optimizers.

I. Momentum

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

$$W \leftarrow W + V_t$$



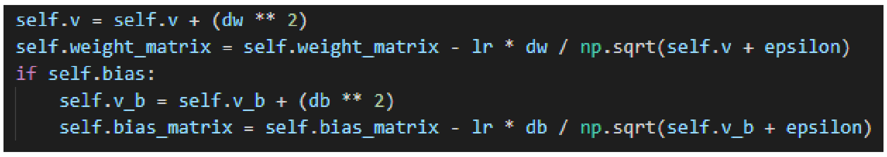
```
self.velocity = beta * self.velocity - lr * dw
self.weight_matrix = self.weight_matrix + self.velocity
if self.bias:
    self.velocity_b = beta * self.velocity_b - lr * db
    self.bias_matrix = self.bias_matrix + self.velocity_b
```

In this part, I follow the instruction of momemtum. Calculate the velocity according previous velocity and the gradient of weight. Then update the weight by the velocity.

II. Adagrad

$$W \leftarrow W - \eta \frac{1}{\sqrt{n+\epsilon}} \frac{\partial L}{\partial W}$$

$$n = \sum_{r=1}^{t} \left(\frac{\partial L_r}{\partial W_r}\right)^2$$



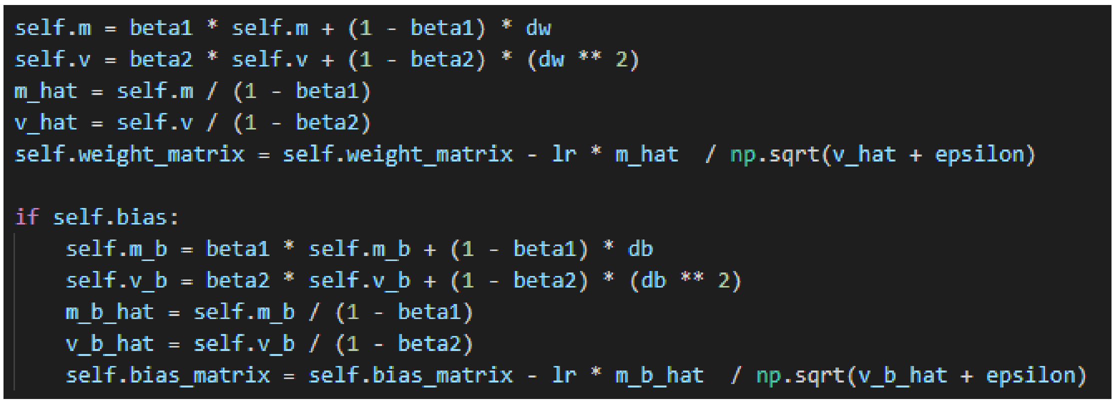
```
self.v = self.v + (dw ** 2)
self.weight_matrix = self.weight_matrix - lr * dw / np.sqrt(self.v + epsilon)
if self.bias:
    self.v_b = self.v_b + (db ** 2)
    self.bias_matrix = self.bias_matrix - lr * db / np.sqrt(self.v_b + epsilon)
```

In this part, I follow the instruction of Adagrad, calculate the adaptive learning rate by the gradient of weight first. Then update the weight matrix.

III. Adam

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L_t}{\partial W_t} \qquad \widehat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad W \leftarrow W - \eta \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t}+\epsilon}$$

$$v_t = \beta_1 v_{t-1} + (1-\beta_2)\left(\frac{\partial L_t}{\partial W_t}\right)^2 \qquad \widehat{v}_t = \frac{v_t}{1-\beta_2^t}$$

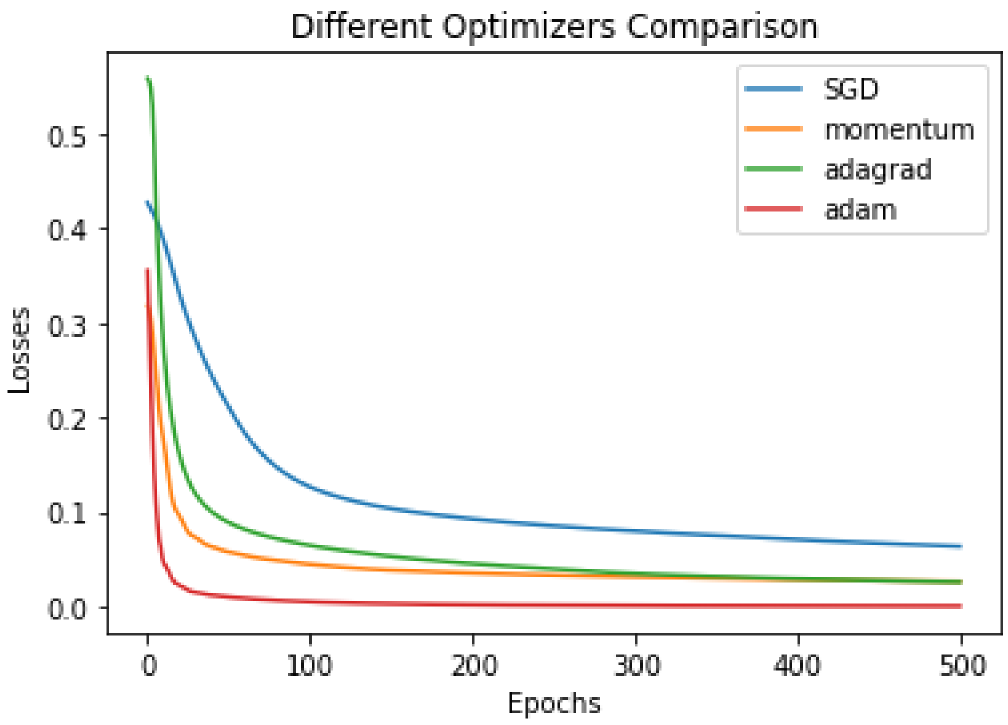


```
self.m = beta1 * self.m + (1 - beta1) * dw
self.v = beta2 * self.v + (1 - beta2) * (dw ** 2)
m_hat = self.m / (1 - beta1)
v_hat = self.v / (1 - beta2)
self.weight_matrix = self.weight_matrix - lr * m_hat  / np.sqrt(v_hat + epsilon)

if self.bias:
    self.m_b = beta1 * self.m_b + (1 - beta1) * db
    self.v_b = beta2 * self.v_b + (1 - beta2) * (db ** 2)
    m_b_hat = self.m_b / (1 - beta1)
    v_b_hat = self.v_b / (1 - beta2)
    self.bias_matrix = self.bias_matrix - lr * m_b_hat  / np.sqrt(v_b_hat + epsilon)
```

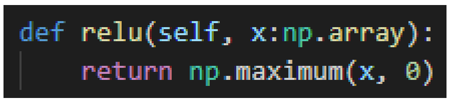
In this part, I follow the instruction of adam, calculate the biased 1st and 2nd moment estimates by the gradient w and the hyperparameter beta1 and beta2 first. Then calculate the bias-corrected moment estimates and update the weight matrix.

Comparison:



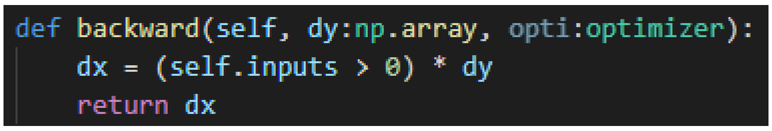
In the experiment of testing different optimizers, we found that SGD has the worst performance. And adam optimizer has the greatest performance since it combines the skills in momentum and adagrad.

B. Implement different activation functions.



```
def relu(self, x:np.array):
    return np.maximum(x, 0)
```

In this part, I implement the relu function by using NumPy's maximum function.



```
def backward(self, dy:np.array, opti:optimizer):
    dx = (self.inputs > 0) * dy
    return dx
```

Since the derivative of relu function is 1 for x > 0, 0 for x < 0, we can simply use the ">" operation in NumPy to calculate dradient.