CSE473s (UG2018) - Computational Intelligence

Major task part 1

Name	ID
Mariam Ashraf Mahmoud	2100849
Omnia Said Mohamed	2100742
Habiba Khaled Ahmed	2100656
Shahd Essam Ahmed	2100659
Shahd Amir Yehia	2100367

- **Library design and architecture choices.**
    1. Layers library:

```
import numpy as np
#Layer Abstraction section
class Layer:
    def forward(self, x):
        pass

    def backward(self, grad):
        pass

    def update(self, lr):
        pass
```

This is the parent class for all neural network layers.

Every layer in your library (Dense, Activation, etc.) must implement:

forward(x)

- Used during forward propagation
- Computes the output of this layer

backward(grad)

- Used during backpropagation
- Receives gradient from the next layer
- Returns gradient for the previous layer

update(lr)

- Used by optimizers to update parameters
- Only layers with weights (like Dense) will use it as input and output layers don't have parameters

```
#dense layer has inputs from neurons and outputs to other neurons, only
layer with parameters (w)
class Dense(Layer):
  #in_features is the size of a single sample
  #out_features is the no. of neurons
    def __init__(self, in_features, out_features):
      #random values for weights, randn gives normal distribution (mean
0, std 1)
      #the weights are reversed so that we don't need to perform
transpose on the weights matrix
        self.W = np.random.randn(in_features, out_features) * 1.0
        #biases are initialized to zero
        self.b = np.zeros((1, out_features))
        self.x = None
        self.dW = None
        self.db = None

    def forward(self, x):
      #Save this input x inside the layer object so backward() can use
it later
        self.x = x
        #it multiples x by weights then adds bias, y=xw+b
        return np.dot(x, self.W) + self.b


    def backward(self, grad_out):
        # Gradients
        #dL/dW = x T * dL/dy
        self.dW = self.x.T @ grad_out
        #sum of output gradients
        self.db = np.sum(grad_out, axis=0, keepdims=True)
        #dL/dX = dL/dy * W T
        grad_input = grad_out @ self.W.T
        return grad_input

    def update(self, lr):
      #lr is the learning rate
        self.W -= lr * self.dW
        self.b -= lr * self.db
```

Weights (W) : They are random initialized to break symmetry, creates a matrix of shape **[in_features, out_features]**

randn gives normal distribution (good for small networks like XOR)

Bias (b) : Each neuron has a bias with shape **[1, out_features]**

The forward pass computes the output: $Y = XW + b$

The backward pass this is the most important part.

Input to backward: grad_out $\frac{dL}{dy}$

We calculate gradient with respect to weights $\frac{dL}{dW} = x^T \frac{dL}{dy}$

Then we calculate gradient with respect to bias (Bias affects output directly, so we sum over samples) $\frac{dL}{db} = \sum \frac{dL}{dy}$

Then gradient with respect to input x (sent to previous layer) $\frac{dL}{dx} = \frac{dL}{dy} W^T$

Finally return gradient for next layer by :return grad_input

<u>SGD (stochastic gradient descent) update rule to update weights of dense layer:</u>

$$\mathrm{W}_{\mathrm{n}+1} = \mathrm{W}_{\mathrm{n}} - \eta \frac{\mathrm{dl}}{\mathrm{dW}}$$

$$b_{n+1} = b_n - \eta \frac{dl}{db}$$

lr = learning rate

dW, db are gradients computed in backward pass

2. Activation Layer Library;

- Sigmoid

Non-linear → helps learning
Range (0, 1)

- Tanh

Even better → symmetric around 0
Range (-1, 1)

This helps hidden layer neurons capture XOR’s shape.

- ReLU

Only outputs positive values → *not useful for XOR hidden layer*, but still a fundamental activation for bigger networks.

<u>**Sigmoid activation:**</u>

```
class Sigmoid:
    def forward(self, x):
        self.out = 1 / (1 + np.exp(-x))
        return self.out
#i need to save sigmoid value to use it while calculating its gradient
#dL/dx = dL/dy * (gradient of sigmoid)
    def backward(self, grad):
        return grad * (self.out * (1 - self.out))
```

Sigmoid formula:

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

We return self.out in the forward pass to save the value of $\sigma(x)$ as we'll need while calculating the gradient of sigmoid

$\grave{\sigma}= \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(1-\sigma)$

Then in the backward pass $\frac{dL}{dx} = \frac{dL}{dy} * \dot{\sigma} = \frac{dL}{dy} * \sigma(1-\sigma)$

Tanh activation:

```
class Tanh:
    def forward(self, x):
        self.out = np.tanh(x)
        return self.out
#derivative of tanh(x)= 1- (tanh(x)^2)
    def backward(self, grad):
        return grad * (1 - self.out**2)
```

Tanh formula:

$$tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$t\grave{a}nh = 1 - (tanh)^2$$

In the backward pass, we calculate $\frac{dL}{dx}$

$$\frac{dL}{dx} = \frac{dL}{dy} * 1 - (tanh)^2$$

3. Layers library:

```
import numpy as np
#mean squared error to caclulate loss
class MSE:
  #Loss= (1/(2*n)) sum(y - y^)^2)
    def forward(self, y_true, y_pred):
        return np.mean((y_true - y_pred)**2)

    def backward(self, y_true, y_pred):
        return 2 * (y_pred - y_true) / y_true.size
```

We calculated the loss by the Mean Squared Error (MSE):

$$\xi = \frac{1}{N}\Sigma(y - \hat{y})^2$$

In backward pass: $\grave{\xi} = \frac{2}{N}\sum(y - \hat{y})$

4. Optimizer library:

The optimizer is the component responsible for updating the weights and biases of your network after backpropagation computes gradients.

```
class SGD:
  #stochastic gradient descent
  #sets learning rate for gradient descent
    def __init__(self, lr=0.5):
        self.lr = lr

#only updates parameters in dense layers , checks if layer has
attribute of update
    def step(self, layers):
        for layer in layers:
            if hasattr(layer, "update"):
                layer.update(self.lr)
```

init sets the learning rate (lr) here we assumed it be 0.5

We update weights and biases using these formulas:

$$\mathrm{W_{n+1} = W_n - \eta\frac{dl}{dW}}$$

$$b_{n+1} = b_n - \eta\frac{dl}{db}$$

Lastly, the step function :

- Loops through all layers in the network
- Only updates layers that *have parameters* (i.e., Dense layers)
- Calls their update(lr) method to change W and b in each iteration.

5. Network library:

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

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

    def backward(self, grad):
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

    def update(self, optimizer):
        optimizer.step(self.layers)
```

This is the most important library in the project, because it is the brain that connects:

- Dense layers
- Activation layers
- Loss
- Optimizer

The init function: Passes a list of layers (Dense + activation layers), The network stores them in order

The forward pass: sends data through each layer one-by-one.

The backward pass: reverses gradients flow backwards

Loss → Output layer → Hidden layer → Input layer

That's what reversed(self.layers) does.

- **Results from the XOR test.**

```
import numpy as np


# XOR dataset
X = np.array([[0,0],
              [0,1],
              [1,0],
              [1,1]])

y = np.array([[0],
              [1],
              [1],
              [0]])

# Build network: 2 → 4 → 1
#2 input neurons (x1,x2), 4 hidden neurons, 1 output neuron
model = Network([
    Dense(2, 4),
    #tanh provide non linearity to the 4 hidden neurons
    Tanh(),
    #takes 4 inputs from the hidden layer and outputs 0 or 1
    Dense(4, 1),
    #sigmoid outputs values between 0 and 1
    Sigmoid()
])

loss_fn = MSE()
optimizer = SGD(lr=0.1)

# Training loop, 5000 iterations
for epoch in range(5000):
    pred = model.forward(X) #y^
    loss = loss_fn.forward(y, pred)
    grad = loss_fn.backward(y, pred) #dL/dy^
    model.backward(grad) #backpropagate through the network
    model.update(optimizer)
#print loss each 50 iterations
    if epoch % 500 == 0:
        print(f"Epoch {epoch}, Loss = {loss:.5f}")

# Final predictions
print("\nFinal Predictions:")
print(model.forward(X))
```

<u>The results:</u>

We can see the loss is decreasing by each iteration

```
Epoch 0, Loss = 0.40811
Epoch 500, Loss = 0.13989
Epoch 1000, Loss = 0.02369
Epoch 1500, Loss = 0.00983
Epoch 2000, Loss = 0.00588
Epoch 2500, Loss = 0.00411
Epoch 3000, Loss = 0.00313
Epoch 3500, Loss = 0.00252
Epoch 4000, Loss = 0.00210
Epoch 4500, Loss = 0.00179

Final Predictions:
[[0.02249629]
 [0.9540998 ]
 [0.96065617]
 [0.04571082
```