#0

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
```

#1

```python
# Generate some toy data for the neural network
# We'll use a simple XOR problem for demonstration
X = torch.tensor( data: [[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
y = torch.tensor( data: [[0], [1], [1], [0]], dtype=torch.float32)
```

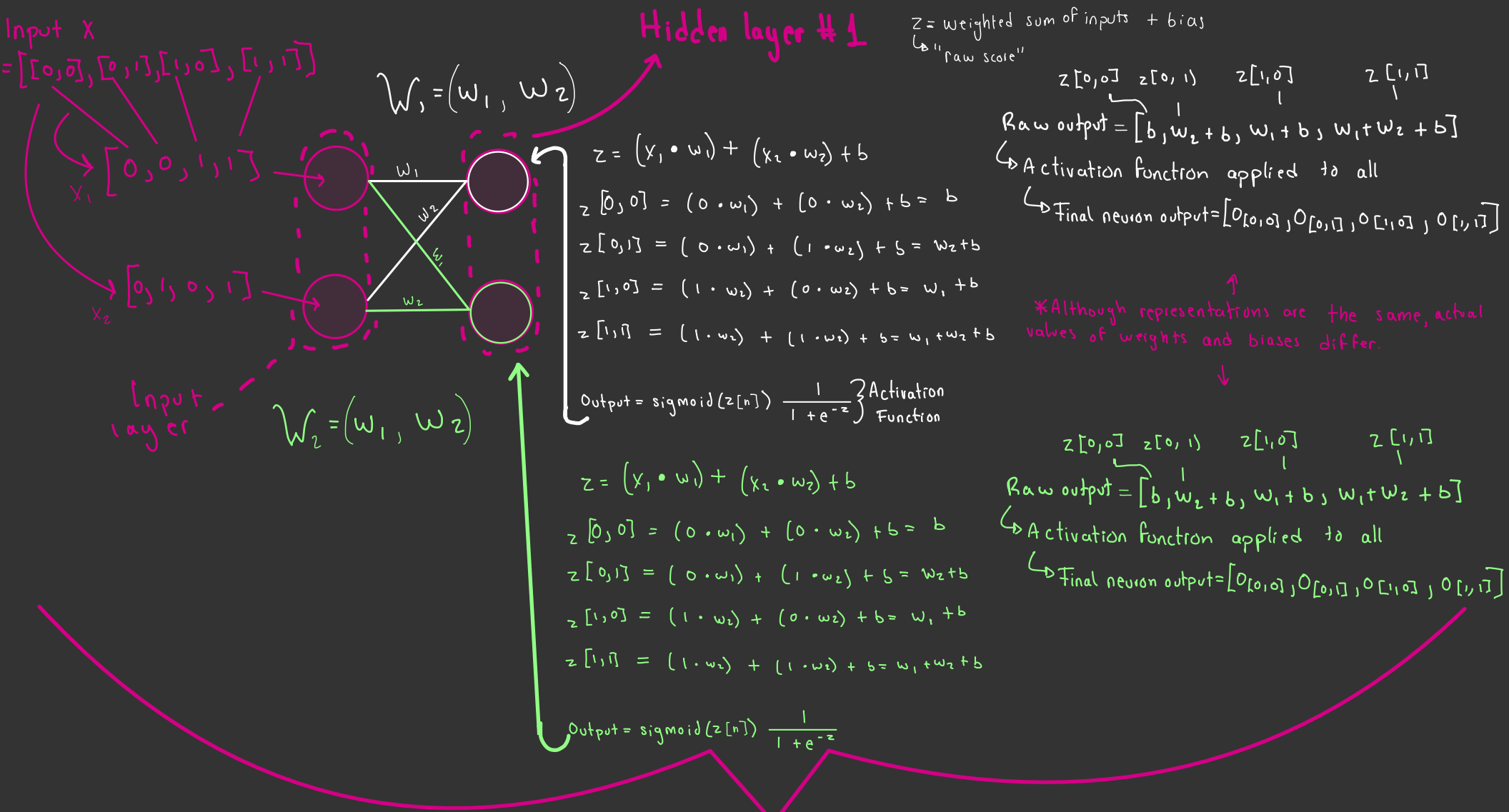
#2

```python
# Define the neural network architecture
2 usages
class XORNeuralNetwork(nn.Module):  # Defining the class, format: ClassName(ParentClassName)
    def __init__(self, input_dim, hidden_dim, output_dim):  # Create a class instance
        super(XORNeuralNetwork, self).__init__()  # Initializing parent class inheritance, same as class ClassName : public ParentClass in C++
        self.hidden_layer = nn.Linear(input_dim, hidden_dim)
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        hidden_output = torch.sigmoid(self.hidden_layer(x))
        output = torch.sigmoid(self.output_layer(hidden_output))
        return output
```

#2.2

self.hidden_layer = nn.Linear(input_dim, hiddendim) $\equiv$ $y = x W^T + b$



z = weighted sum of inputs + bias
↳ "raw score"

z[0,0] z[0,1] z[1,0] z[1,1]

Raw output = [b, w2 + b, w1 + b, w1 + w2 + b]

↳ Activation function applied to all

↳ Final neuron output = [O[0,0], O[0,1], O[1,0], O[1,1]]

*Although representations are the same, actual values of weights and biases differ.

z[0,0] z[0,1] z[1,0] z[1,1]

Raw output = [b, w2 + b, w1 + b, w1 + w2 + b]

↳ Activation function applied to all

↳ Final neuron output = [O[0,0], O[0,1], O[1,0], O[1,1]]

Forward Pass

Step #1:

Input → 1st Hidden layer

Calculations for all neurons within 1st layer.

Layer-to-layer process is the same

Hidden layer # 1

$W_1 = (w_1, w_2)$

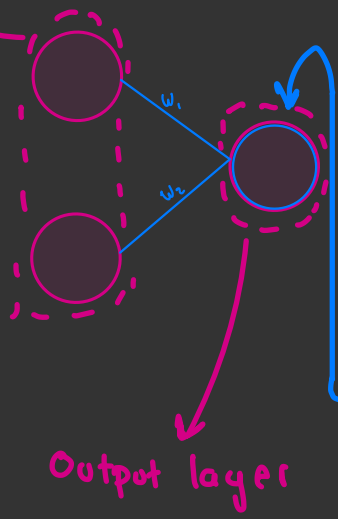
z = weighted sum of inputs + bias
↳ "raw score"

$z[0,0]$ $z[0,1]$ $z[1,0]$ $z[1,1]$

Raw output = $[b, w_2 + b, w_1 + b, w_1 + w_2 + b]$

↳ Activation function applied to all

↳ Final neuron output = $[O_{[0,0]}, O_{[0,1]}, O_{[1,0]}, O_{[1,1]}]$



$z = (x_1 \cdot w_1) + (x_2 \cdot w_2) + b$

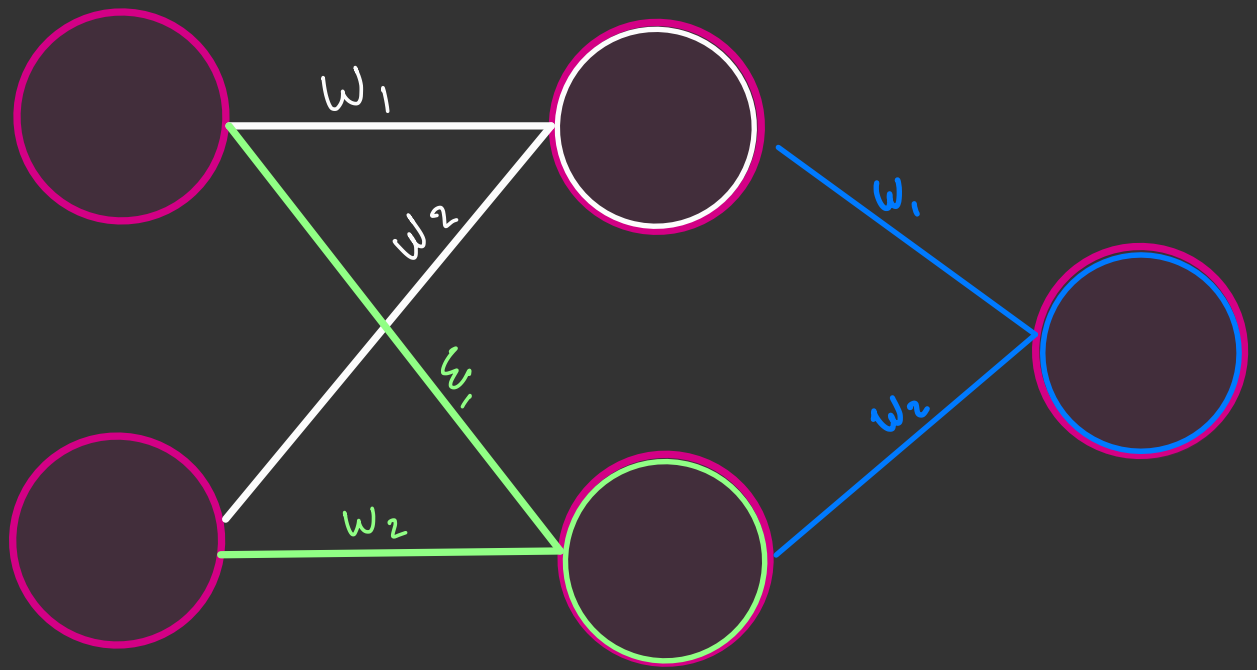
$z[0,0] = (0 \cdot w_1) + (0 \cdot w_2) + b = b$

$z[0,1] = (0 \cdot w_1) + (1 \cdot w_2) + b = w_2 + b$

$z[1,0] = (1 \cdot w_2) + (0 \cdot w_2) + b = w_1 + b$

$z[1,1] = (1 \cdot w_2) + (1 \cdot w_2) + b = w_1 + w_2 + b$

Output = sigmoid(z[n]) $\frac{1}{1 + e^{-z}}$ } Activation Function



#3

```python
# Initialize the neural network
input_dim = 2
hidden_dim = 2
output_dim = 1
model = XORNeuralNetwork(input_dim, hidden_dim, output_dim)
```

→ create model

#4

```python
# Define the loss function (Mean Squared Error) and the optimizer (Stochastic Gradient Descent)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

Class | We're creating a class instance

#5

```python
# Training loop
epochs = 100000

for epoch in range(epochs):
    optimizer.zero_grad() # reset the gradients of the params
    # Forward pass
    predictions = model(X)
    # Calculate the loss
    loss = criterion(predictions, y)
    # Backpropagation
    loss.backward()
    # Update weights
    optimizer.step()

    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item()}")
```

5.1

5.2 → model.forward(x) → (step # 2.2)

5.3 → criterion.forward(predictions, y) → nn.MSELoss.forward(criterion, predictions, y)

self

→ distance from predicted to target

5.4 → compute gradients

5.5 → update params based on gradients to minimize loss

5.3 Sum of squared errors divided by data points

$\hat{y}$

Predictions $= [0.4, 0.7, 0.6, 0.3]$

(target)

$y = [0, 1, 1, 0]$

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

1. $(y_i - \hat{y}_i)^2 = (0 - 0.4)^2 = 16$

   $\vdots$

   $(y_i - \hat{y}_i)^2 = (0 - 0.3)^2 = 0.09$

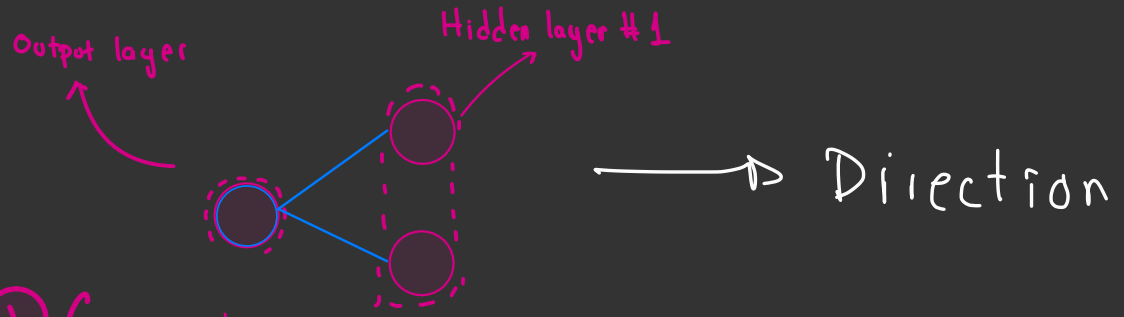
2. $MSE = \frac{1}{4}(0.16 + 0.09 + 0.16 + 0.09$

   $MSE = \frac{1}{4}(0.5)$

   $MSE = 0.125$

Loss = 0.125

5.4 Backpropagation: calculate neurons' contribution to error



① Compute gradient (Partial derivative) of loss w/respect to output $\hat{y}$ (within output neuron)

$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

② Chain rule: Propagate error through activation function | Basically calculate gradient w/respect to pre-activation func output (z). Also taking sigmoided z in mind, or rather, its derivative: $\hat{y}(1-\hat{y})$

$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y) \cdot \hat{y}(1-y)$$

③ Gradient w/respect to weights and biases connected to hidden layer.

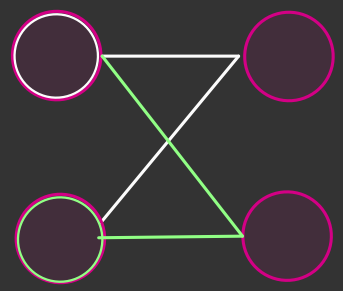
Weights gradient: $\frac{\partial L}{\partial W_{output}} = \frac{\partial L}{\partial z} \cdot h$ ← Hidden layer output

How much the weights of the output layer should change to reduce loss.

Bias gradient: $\frac{\partial L}{\partial b_{output}} = \frac{\partial L}{\partial z}$

How much bias adjustment in output neuron to reduce loss.

Backprop to hidden layer



4 Gradient w/respect to hidden layer outputs, pre-activation values

For each neuron, the gradient of loss w/respect to its output h is calculated by summing the contributions of neurons in the output layer.

$$\frac{\partial L}{\partial h} = \sum_{i=1}^{n_{output}} \left(\frac{\partial L}{\partial z_{output}} \cdot W_{output,i} \right)$$

W_{output} = weight connecting hidden neuron to the i-th output n.

5 Gradient w/respect to hidden layer weights and biases

Weights gradient: $\frac{\partial L}{\partial W_{hidden}} = \frac{\partial L}{\partial h} \cdot X$

Bias gradient: $\frac{\partial L}{\partial b_{hidden}} = \frac{\partial L}{\partial h}$

- After backprop, the gradients for each parameter (weights and biases) have been calculated and stored in their .grad attribute. In the next step, the optimizer's job is to use these gradients to update the parameters in order to minimize loss.

5.5 Update parameters to minimize loss - Optimizer: SGD (Stochas

① Recall gradient calculation: At the end of backprop, for every weight W and calculated the gradient of loss with respect to those parameters.

- For the weights in the output layer:
$$\frac{\partial L}{\partial W_{output}}$$
- For the biases in the output layer:
$$\frac{\partial L}{\partial b_{output}}$$
- For the weights and biases in the hidden layer:
$$\frac{\partial L}{\partial W_{hidden}} \quad \text{and} \quad \frac{\partial L}{\partial b_{hidden}}$$

These gradients tell us how much should change to minimize the loss.

② SGD Formula: Updates weights and biases by taking a step in the direction that minimizes gradients. The SGD update rule:

$$\theta_{new} = \theta_{old} - \eta \cdot \frac{\partial L}{\partial \theta}$$

θ: parameter (weights / bias)

η: learning rate, controls how big update step is

$\frac{\partial L}{\partial \theta}$: gradient of loss w/ respect to parameter.