BACK

PROPAGATION

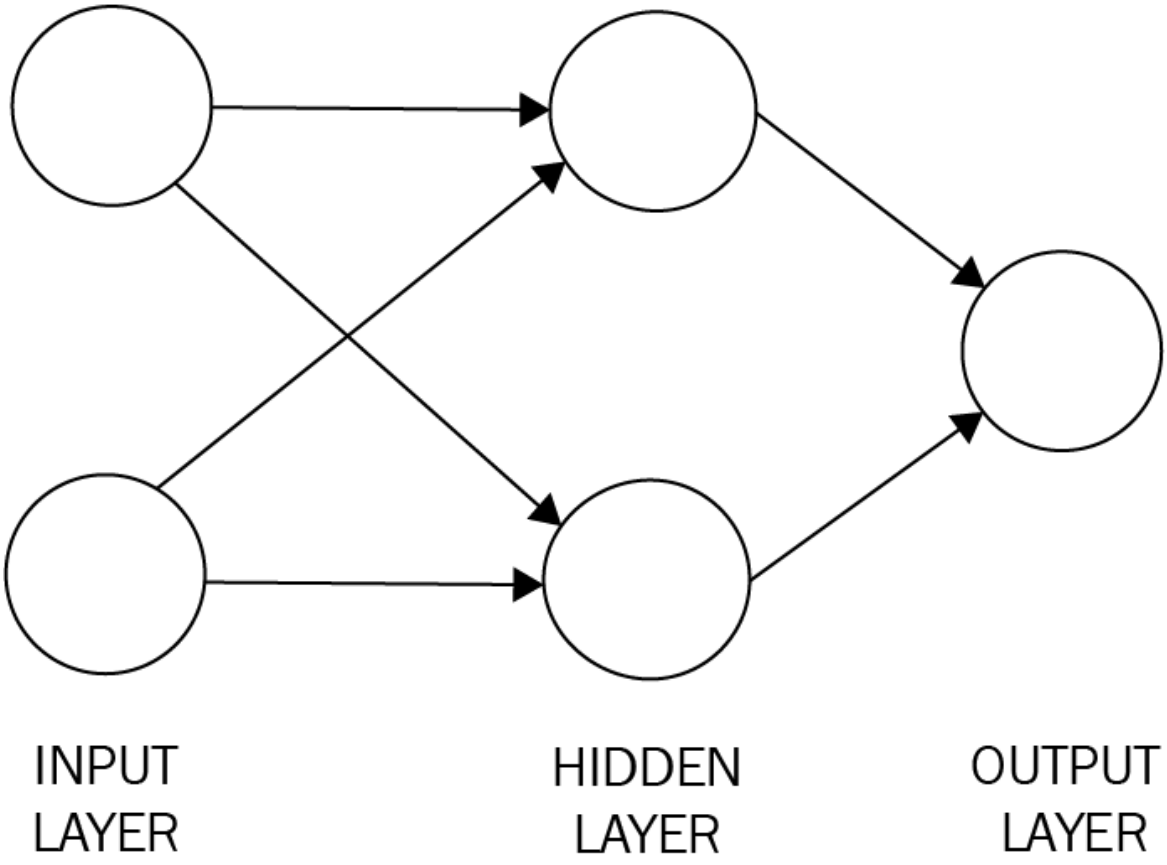
ARIHARASUDHAN

BACK PROPAGATION

The Back-Propagation is A SuperVised Learning Technique used for training our Neural Networks. Here, in this book, we guys will learn how learning in Neural Networks are achieved by The Back-Propagation technique. Let's take a simple example of computing XOR Value of two inputs using A Trained Neural Network. We'll be looking on how Back-Propagation is performed on The Neural Network so that it can learn to perform XOR operation.

Input		Output	
A	**B**	**P**	**Q**
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

We'll use a neural network with one hidden layer containing two nodes.



Step 1: Initialization

Let's start by randomly initializing the weights and biases for our neural network.

Input to Hidden Layer Weights

w1 = 0.3, w2 = 0.5

Hidden Layer Biases

b1 = 0.1

Hidden to Output Layer Weights:

w3 = 0.2, w4 = 0.4

Output Layer Bias:

b2 = 0.6

We've just randomly initialized these values.

Step 2: Forward Propagation

Now, we will perform forward propagation to compute the output for the **Input: (1, 0)**

$$\textbf{SIGMOID}(x) = 1/(1 + e^{-x})$$

★ Compute the activations of the hidden layer nodes (a_{h1}, a_{h2}) :

a_{h1} = sigmoid(1*w1 + 0 * w2 + b1)

a_{h1} = sigmoid(0.3 + 0 + 0.1)

≈ sigmoid(0.4) ≈ **0.5987**

a_{h2} = sigmoid(1*w2 + 0*w2 + b1)

a_{h2} = sigmoid(0.5 + 0 + 0.1)

≈ sigmoid(0.6) ≈ **0.6457**

★ Compute the output of the neural network (a_o):

a_o = sigmoid(a_{h1}*w3 + a_{h2}*w4+ b2)

a_o = sigmoid(0.5987 * 0.2 + 0.6457 * 0.4 + 0.6)

≈ sigmoid(0.30901) ≈ **0.5762**

Step 3: Calculate the Loss

Now, we compare the network's output ($a_o \approx 0.5762$) with the target output (1) for the input (1, 0) to compute the loss.

$$\text{Loss} = 0.5*(\text{target} - \text{output})^2$$
$$\text{Loss} = 0.5*(1 - 0.5762)^2$$
$$\approx \mathbf{0.0809}$$

Step 4: Backpropagation

The goal of backpropagation is to adjust the weights and biases to reduce the loss. Wait… Why should we reduce the loss? Because, if the loss is reduced, it means, the output is approximately the target. For example, if the target is 1 and the output is 0.98, the loss is 0.0004. (**1≈0.98**)

We calculate the gradients of the loss with respect to the model parameters and update them accordingly. In math, a gradient is like a compass that tells you which way is "uphill" in a landscape. Imagine you are standing on a hill, and you want to go up as quickly as possible. The gradient is a vector (a direction with a length) that points in the direction of the steepest slope.

In the context of functions, the gradient shows the direction and rate of fastest increase. If you have a function with more than one variable (like x and y), the gradient is like an arrow that points in the direction where the function increases the most. The length of the arrow tells you how steep the increase is.

In practical applications, the gradient is used in many fields, like finding the best route on a map, optimizing computer programs, and training artificial intelligence systems to improve their performance. It helps us make better decisions and find solutions to complex problems.

Now, let's come to the case.

★ Compute the gradient of the loss with respect to the output layer activation (δ_{output}):

$\delta_{output} = (target - a_o) * \sigma'(a_o)$

$= (1 - 0.5762) * \sigma'(0.5762)$

$\approx \mathbf{0.1444}$

NOTE : $\sigma'(x) = d/dx(1/(1+e^{-x}))$

★ Compute the gradients of the loss with respect to the weights and biases of the hidden-to-output connections:

$\delta_{w3} = \delta_{output} * a_{h1}$

$= 0.1444 * 0.5987$

≈ 0.0864

$\delta_{w4} = \delta_{output} * a_{h2}$

$= 0.1444 * 0.6457$

≈ 0.0931

$\delta_{b2} = \delta_{output} \approx 0.1444$

★ Compute the gradients of the loss with respect to the activations of the hidden layer nodes (δ_{h1}, δ_{h2}):

$\delta_{h1} = \delta_{output} * w3 * \sigma'(a_{h1})$

$= 0.1444 * 0.2 * \sigma'(0.5987)$

≈ 0.0182

$\delta_{h2} = \delta_{output} * w4 * \sigma'(a_{h2})$

$=0.1444*0.4*\sigma'(0.6457)$

≈ 0.0371

★ Compute the gradients of the loss with respect to the input-to-hidden weights and biases:

$\delta_{w1} = \delta_{h1} * input_1$

$= 0.0182 * 1$

≈ 0.0182

$\delta_{w2} = \delta_{h2} * input_2$

$= 0.0371 * 0$

≈ 0

$\delta_{b1} = \delta_{h1} + \delta_{h2} \approx 0.0182 + 0.0371 \approx 0.0553$

Step 5: Update Weights & Biases

Now, we update the weights and biases of the neural network using the gradients calculated in the backpropagation step and a learning rate (η).

Let's assume the learning rate (η) is 0.1.

★ Update Hidden-to-Output Weights and Biases:

$w_3 = w_3 + \eta * \delta_{w3}$

$= 0.2 + 0.1 * 0.0864 \approx \mathbf{0.2086}$

$w_4 = w_4 + \eta * \delta_{w4}$

$= 0.4 + 0.1 * 0.0931 \approx \mathbf{0.4093}$

$b_2 = b_2 + \eta * \delta_{b2}$

$= 0.6 + 0.1 * 0.1444 \approx \mathbf{0.6144}$

★ Update Input-to-Hidden Weights and Biases:

$w_1 = w_1 + \eta * \delta_{w1}$

$= 0.3 + 0.1 * 0.0182 \approx \mathbf{0.3018}$

$w_2 = w_2 + \eta * \delta_{w2}$

$= 0.5 + 0.1 * 0 \approx \mathbf{0.5}$

$b_1 = b_1 + \eta * \delta_{b1}$

$= 0.1 + 0.1 * 0.0553 \approx \mathbf{0.1055}$

Step 6: Testing

Now, the updated neural network can be tested with different inputs to see if it approximates the XOR operation correctly. We would repeat the forward propagation step for different inputs (0, 0), (0, 1), and (1, 1) to observe the network's outputs and check how well it approximates the XOR function.

Please note that this is a simple example, and the neural network with only one hidden layer and two nodes may not be able to perfectly approximate the XOR operation. For better results, deeper networks with more hidden layers and nodes are often used.

LET'S CODE IT

Let's import the necessaries and create an input and target dataset.

```
import torch
import torch.nn as nn
import torch.optim as optim

# XOR data: input and target output
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)
```

It is the same Neural Network we have discussed.

```
# Define the neural network class
class XORModel(nn.Module):
    def __init__(self):
        super(XORModel, self).__init__()
        self.hidden = nn.Linear(2, 2)  # One hidden layer with 2 nodes
        self.output = nn.Linear(2, 1)  # Output layer with 1 node

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = torch.sigmoid(self.output(x))
        return x
```

Create an instance for the model. Define the loss function and the optimizer.

```
# Create an instance of the XORModel
model = XORModel()
# Define the loss function (Binary Cross Entropy) and the optimizer
criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

Let’s start The Training..

```
# Training loop
epochs = 10000
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = model(X)
    loss = criterion(outputs, y)
    loss.backward()
    optimizer.step()

    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

And let’s make predictions.

```
# Make predictions
with torch.no_grad():
    predictions = model(X)
    rounded_predictions = torch.round(predictions)

print("Predictions:")
print(rounded_predictions)
```

IN CODE

The code includes the **`loss.backward()`** call, which computes the gradients and performs the backpropagation step through the neural network. The automatic differentiation feature of PyTorch handles this backpropagation process, so we guys don't need to explicitly implement it.

optimizer.zero_grad() : This step is done at the beginning of each training iteration to clear the gradients of the parameters from the previous iteration. It ensures that the gradients are fresh and ready to be updated in the current iteration.

outputs = model(X) : This step involves forward propagation, where the input data `X` is passed through the model, and the predicted outputs are computed.

loss = criterion(outputs, y) : The next step is to compute the loss between the predicted outputs and the target outputs `y`. The loss function measures how far the predicted outputs are from the actual target outputs.

loss.backward() : Here, The Backpropagation is performed to compute the gradients of the loss with respect to the model's parameters (weights and biases).

These gradients are calculated using the chain rule, propagating the error back through the layers of the neural network.

optimizer.step() : Finally, the optimization step is performed. The optimizer uses the computed gradients to update the model's parameters (weights and biases) to minimize the loss. It usually involves updating the parameters using gradient descent or one of its variants.

By following this order, we ensure that the gradients are calculated based on the current batch of data and are then used to update the model's parameters accordingly.

MERCI