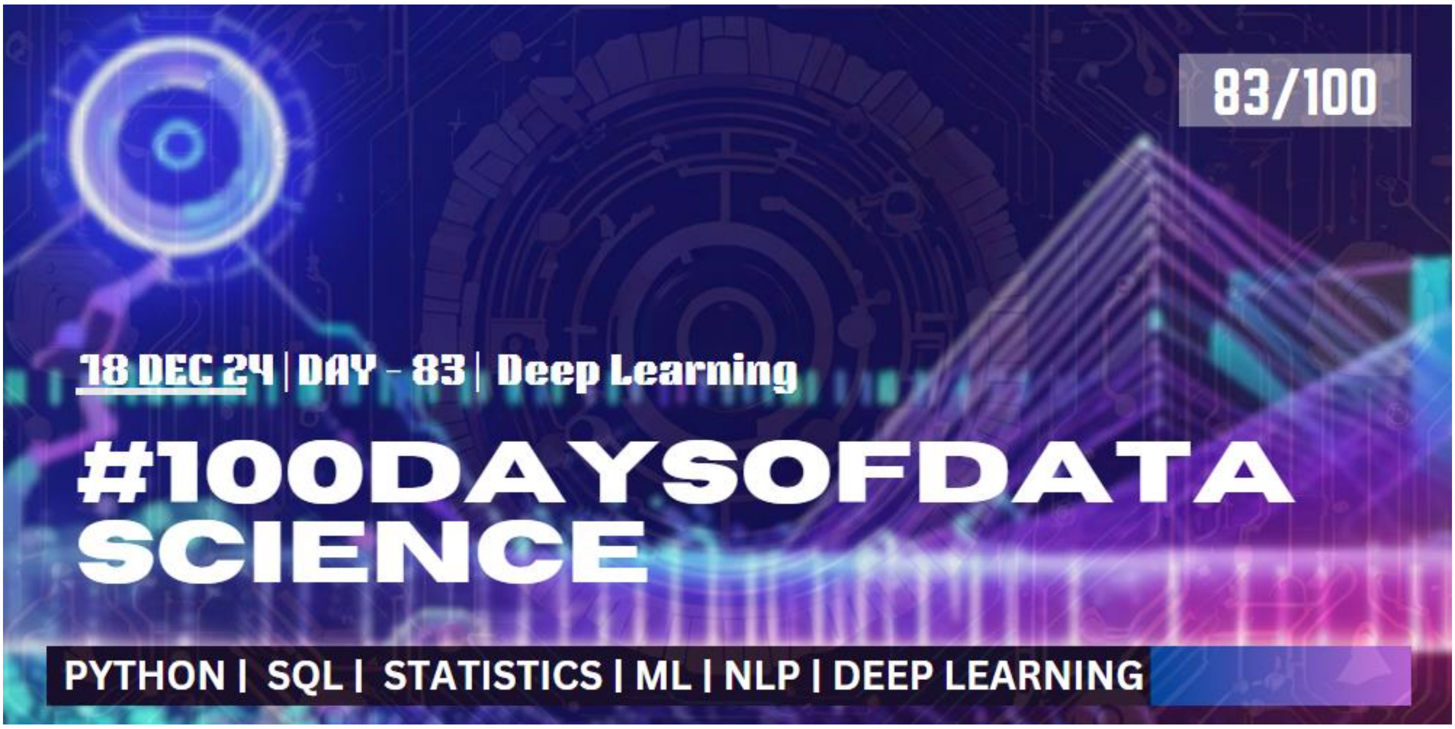
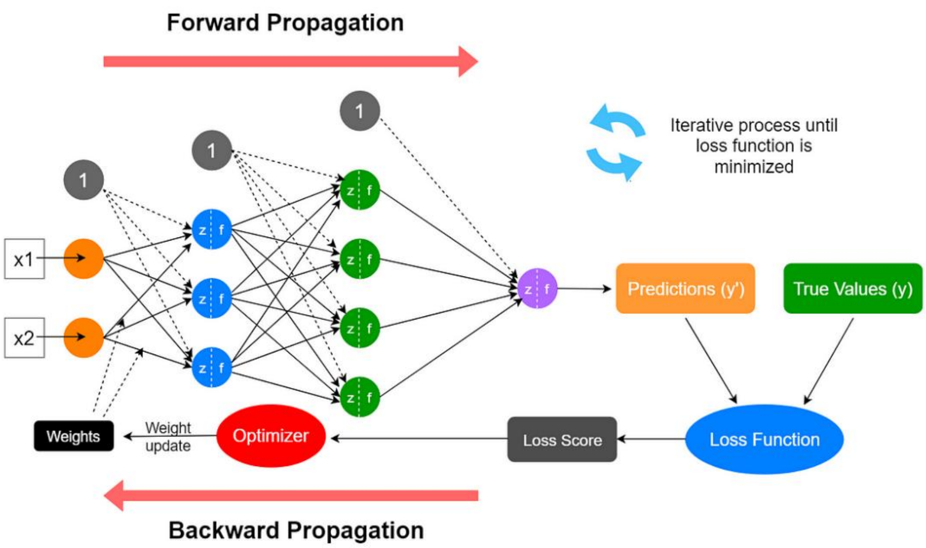


Forward Propagation and Backward Propagation

Forward and Backward Propagation in Neural Networks

Forward and backward propagation are the cornerstones of training neural networks. These processes work together to ensure that a model learns by minimizing the error in its predictions, forming the backbone of modern deep learning systems.



Key Features of Forward and Backward Propagation

1. Forward Propagation

This process involves calculating the output of the neural network by passing inputs through multiple layers.

- **Prediction Generation**: The network uses its current weights and biases to compute the predicted value.
- **Layer-by-Layer Computation**: Inputs are transformed using weights, biases, and activation functions across layers.
- **Non-Linearity**: Activation functions like Sigmoid or ReLU introduce non-linear decision-making capabilities.

2. Backward Propagation

Backward propagation, or "backprop," fine-tunes the model parameters to reduce the error between predictions and actual target values.

- **Error Propagation**: Gradients of the loss function are computed layer by layer.
- **Weight Adjustment**: Weights and biases are updated using the gradients and a learning rate.
- **Iterative Optimization**: This process repeats for multiple epochs to improve model accuracy.

Applications of Forward and Backward Propagation

- **Image Classification**: Training convolutional networks to identify objects in images.
- **Text Generation**: Optimizing recurrent or transformer networks for natural language processing.
- **Medical Diagnosis**: Predicting diseases from patient data.
- **Stock Market Analysis**: Modeling time-series data for financial predictions.

Workflow of Forward and Backward Propagation

1. Initialize Parameters

Start with random values for weights and biases for all layers of the network.

2. Forward Pass

- Compute the weighted sum of inputs at each layer.
- Apply an activation function to generate outputs for the next layer.
- Continue until the final prediction is obtained.

3. Compute Loss

Calculate the difference between the predicted and actual values using a loss function like Mean Squared Error (MSE).

4. Backward Pass

- Propagate the error backward through the network.
- Compute gradients of the loss function with respect to weights and biases.
- Update weights and biases using the formula:

$$w_i^{\text{new}} = w_i - \eta \cdot \frac{\partial L}{\partial w_i}$$

Where:

η: Learning rate

$\frac{\partial L}{\partial w_i}$: Gradient of the loss function.

5. Repeat

Iterate through forward and backward passes for several epochs until the model converges to an optimal solution.

Limitations of Forward and Backward Propagation

1. Computational Cost

Training deep networks can be resource-intensive, requiring significant time and computational power.

2. Vanishing Gradients

For deep networks with many layers, gradients can become very small, slowing down learning.

3. Overfitting

If not regularized properly, networks can overfit to training data, reducing generalization on unseen data.

Example: Training a Neural Network with Forward and Backward Propagation

- **Forward Pass**: Implemented to calculate the network's predictions for input data.
- **Backward Pass**: Used to optimize weights by reducing the error.
- **Training Loop**: Repeated the process over multiple epochs to minimize loss.

[13]:

```python
# Import required library
import numpy as np

# Target value and input data
Y = np.array([[0.875]])  # Target value (modified)
X = np.array([[0.6, 0.4]])  # Input data (modified)

# Initialize weights and biases with random values
W = [np.random.randn(2, 2), np.random.randn(2, 2), np.random.randn(2, 1)]
B = [np.random.randn(1, 2), np.random.randn(1, 2), np.random.randn(1, 1)]

# Activation function (sigmoid) and its derivative
sig = lambda x: 1 / (1 + np.exp(-x))  # Sigmoid function
dsig = lambda A: A * (1 - A)  # Derivative of sigmoid

# Loss function (mean squared error) and its derivative
mse = lambda x, y: 0.5 * np.square(x - y).sum()  # Mean Squared Error (MSE)
dmse = lambda x, y: (x - y)  # Derivative of MSE

# Forward propagation function
def forward_pass(X, W, B):
    """Performs forward propagation through the network"""
    A, dA = [], []  # To store activations and derivatives of activations
    for i, w in enumerate(W):
        A.append(X)  # Store current input as activation
        X = sig(np.dot(X, w) + B[i])  # Compute output for the current layer
        dA.append(dsig(X))  # Store derivative of activation
    return X, A, dA  # Return final output, activations, and their derivatives

# Backward propagation function
def backward_pass(W, B, A, dA, pred, Y, learning_rate=0.5):
    """Performs backward propagation and updates weights and biases"""
    E = dmse(pred, Y) * dA[-1]  # Compute error for the output layer
    for i, w in reversed(list(enumerate(W))):
        dw = np.dot(A[i].T, E)  # Compute gradient for weights
        db = np.dot(np.ones(shape=(1, E.shape[0])), E)  # Compute gradient for biases
        W[i] -= dw * learning_rate  # Update weights
        B[i] -= db * learning_rate  # Update biases
        if i > 0:  # Propagate error to the previous layer
            E = np.dot(E, w.T) * dA[i - 1]  # Compute error for previous layer

# Deep copy weights and biases to update them during training
updated_W = [w.copy() for w in W]
updated_B = [b.copy() for b in B]

# Training loop
for epoch in range(501):
    # Perform forward propagation
    pred, A, dA = forward_pass(X, updated_W, updated_B)

    # Perform backward propagation
    backward_pass(updated_W, updated_B, A, dA, pred, Y)

    # Log progress every 20 epochs
    if epoch % 20 == 0:
        print(f"Epoch: {epoch}, Prediction: {pred}, Loss: {mse(pred, Y):.6f}")
```

```
Epoch: 0, Prediction: [[0.52946706]], Loss: 0.059697
Epoch: 20, Prediction: [[0.71272151]], Loss: 0.013167
Epoch: 40, Prediction: [[0.77830423]], Loss: 0.004675
Epoch: 60, Prediction: [[0.8098765]], Loss: 0.002121
Epoch: 80, Prediction: [[0.82816845]], Loss: 0.001097
Epoch: 100, Prediction: [[0.83996418]], Loss: 0.000614
Epoch: 120, Prediction: [[0.84810007]], Loss: 0.000362
Epoch: 140, Prediction: [[0.85397103]], Loss: 0.000221
Epoch: 160, Prediction: [[0.85834477]], Loss: 0.000139
Epoch: 180, Prediction: [[0.86167989]], Loss: 0.000089
Epoch: 200, Prediction: [[0.86426787]], Loss: 0.000058
Epoch: 220, Prediction: [[0.86630316]], Loss: 0.000038
Epoch: 240, Prediction: [[0.86792054]], Loss: 0.000025
Epoch: 260, Prediction: [[0.86921641]], Loss: 0.000017
Epoch: 280, Prediction: [[0.87026148]], Loss: 0.000011
Epoch: 300, Prediction: [[0.87110872]], Loss: 0.000008
Epoch: 320, Prediction: [[0.87179847]], Loss: 0.000005
Epoch: 340, Prediction: [[0.87236194]], Loss: 0.000003
Epoch: 360, Prediction: [[0.87282352]], Loss: 0.000002
Epoch: 380, Prediction: [[0.87320251]], Loss: 0.000002
Epoch: 400, Prediction: [[0.87351427]], Loss: 0.000001
Epoch: 420, Prediction: [[0.8737711]], Loss: 0.000001
Epoch: 440, Prediction: [[0.87398296]], Loss: 0.000001
Epoch: 460, Prediction: [[0.8741579]], Loss: 0.000000
Epoch: 480, Prediction: [[0.87430249]], Loss: 0.000000
Epoch: 500, Prediction: [[0.87442206]], Loss: 0.000000
```

Forward and backward propagation are integral to training neural networks. By understanding these processes, we gain insights into how models learn and improve over time. Mastering these concepts is a critical step toward building more advanced AI systems. Implementing forward and backward propagation provides hands-on experience with core neural network operations and optimization techniques.