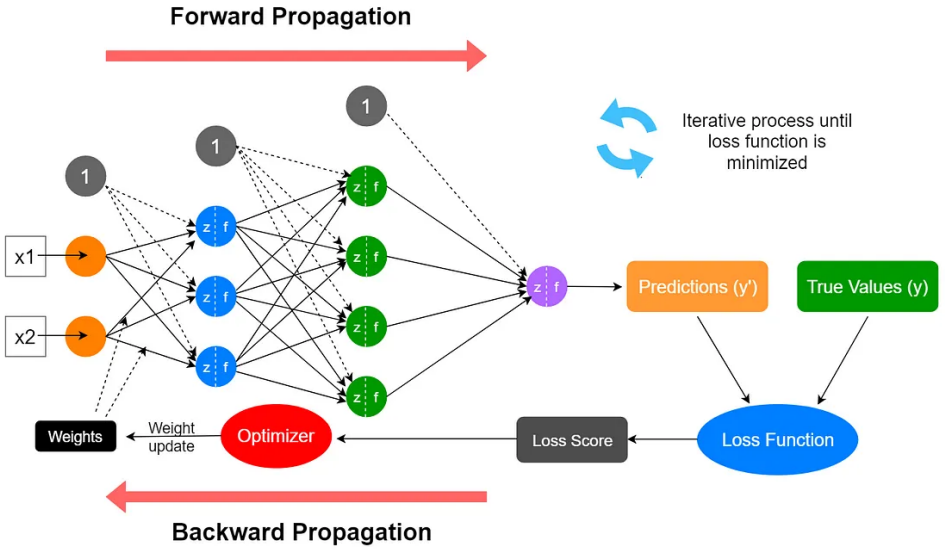
Overview of a Neural Network Learning Process



The learning process of a Neural Network include `Forward Propagation` and `Backward Propagation.`

Forward Propagation

1. The input data is fed into the neural network.
2. The input data is multiplied by the weight of the first layer then added with a bias term.
3. An activation is applied to the above weighted sum.
4. The output of the first layer is propagated to the next layer.
5. Steps 2-4 are repeated for each layer in the Neural Network.
6. The output of the final layer is the prediction of the Neural Network.

→ Main job of the `Forward Prop` is to feed the training examples into different layers of the NN to output the predicted value.

```
# Pseudo code for the Forward Propagation
def forward_prop(X, weights, biases):
    # Initialize the activations with the input data
    activations = X

    # Propagate the input through the layers
    for i in range (len(weights)):
        # z = W^T.X + b
        # a = g(z)
        z = np.dot(activations, weights[i]) + biases[i]
        a = activation_function(z)

    # output of the Neural Network is the final activation
    output = activations

    return output
```

Backward Propagation

1. The error between the predicted output and the actual target value is calculated.
2. The error is propagated backward through the Neural Network, layer by layer.
3. The gradient of the error with respect to the weights and biases of each layer is calculated.
4. The weights and biases of each layer are updated using the calculated gradients and a weight update rule such as Gradient Descent.
5. Steps 2-4 are repeated until the error reaches and acceptable level or the Neural Network converges.

→ main job of the `Backward Prop` is:

1. **Calculate the loss value**: Back propagation calculates the difference between the actual output of the Neural Network and the desired target values (loss value).
2. **Compute the Gradients**: Back propagation efficiently computes the gradients of the loss value with respect to the weights and biases of the neural network. Read more.
3. **Update parameters**: The calculated gradients are used to update the weights and biases of the Neural Network in a way that minimizes the loss value. This process of Gradient Descent iteratively adjusts the parameters to reduce the error and improve the network's performance.

```
# Pseudo code for the Backward Propagation
def back_propagation(X, y, weights, activation_function_derivative):
    # Initialize the gradients with zeros
    gradients = []

    # Calculate the error at the output layer
    error = (output - y) * activation_function_derivative(output)

    # Propagate the error backward through the layers
    for i in range(len(weights) -1, -1, -1):
        w = weights[i]
        b = biases[i]
        a = activations[i]

        delta = np.dot(error, w.T) * activation_function_derivative(a)

        dw = np.dot(a.T, delta)
        db = np.sum(delta, axis = 0)

        gradients.append((dw, db))

    # Reverse the order of gradients to match the order of layers
    gradients.reserve()

    return gradients
```

Iterative Process

Forward propagation and backward propagation are performed iteratively until the Neural Network learns to predict the target output accurately for the given training data.

The following is a more detailed explanation of each step in the learning process:

Input Data

The input data to a Neural Network can be any type of data, such as images, text, or audio.

The input data is typically represented as a `vector of numbers`, witch each number representing a single feature of the data.

Weights and Biases

The `weights` and `biases` of a Neural Network are the parameters that are learned during the training process.

- The `weights` control the strength of the connections between neurons in different layers of the network.
- The `biases` are added to the weighted sum of the inputs to a neuron before the activation function is applied.

Activation Function

The `activation function` is a non-linear function that is applied to the weighted sum of the inputs to a neuron.

The `activation function` introduces **non-linearity** into the Neural Network, which allows it to learn complex patterns in the data.

Error Function

The `Error Function` is used to measure the **difference between** the **predicted output** of the Neural Network and the **Actual Target Value**.

→ The goal of a Neural Network is to minimize the error function.

Weight Update Rule

The `weight update rule` is used to **update the weights and biases** of the Neural Network in a way that **minimizes the error function**.

A common weight update rule is `Gradient Descent`, which iteratively adjusts the weights and biases in the direction that minimizes the error function.

Convergence

`Convergence` is achieved when the error function reaches an acceptable level or the Neural Network learns to predict the target output accurately for the given training data.

Once the Neural Network has converged, it can be used to predict the output for new data that it has never seen before.

```
# define the hyperparameters
learning_rate = 0.01
epochs = 100

# Traing the NN
for epoch in range(epochs):
    # Forward propagation
    # y^ = forward_prop(W, w, b)
    output = forward_propagation(X, weights, biases)

    # Calculate the loss (square error)
    # loss = (y^ - y)^2
    loss = (output - y)**2

    # backward propagation to calculate gradients
    # db/dL and dw/dL
    gradients = backward_propagation(X, y, weights, biases, activation_function_derivative)

    # Update weights and biases using gradients
    for i in range(len(weights)):
        # w := w - alpha * dw/dL
        # b := b - alpha * db/dL
        weights[i] -= learning_rate * gradients[i][0]
        biases[i] -= learning_rate * gradients[i][1]
```

Subpages

Purpose of using Biases in ML and DL

How do Gradients help guiding the Neural Network