Neural Network

How it works ?

Structure

The conceptual idea of a neural network comes from the natural biology of the brain and its neurons that communicate with each other using electrical and chemical signals. Each neuron receives input signals across the synapse and processes this information and sends a signal to the next one.

The power of this signal depends on the strength of the connection between the neurons, which in Neural Networks are the weights.

We have three essential values in each perceptron, which is the artificial version of its biological version, ***inputs*** arriving from the original processed data or from other perceptrons, the ***bias*** that shifts the function away from the central axis to allow learning of non-linear patterns more easily, regulates the activation of perceptrons, it can make it easier or harder to activate, and at least, the core that gives the ability to learn something to the neural network, the ***weights***, they multiply all the input values, each of them multiplies a connection between the perceptrons, and that's what the neural network is trying to achieve, the collection of weights is what it is learning, these weights will tell when a perceptron will be activated or not.

The neural network has three layers, the **input layer** that receives the data that we are going to process, it does not perform the calculations, it only passes the data that we are going to calculate to the next layers, the **hidden layers**, they do the processing and learning, they receive the data from the previous layers, multiply it by the respective weights, add it to the bias, this raw value goes through what we call the activation function, which depending on the value that passes through it, it will return a different value that will be passed on to the next layer, repeating this finally we arrive at the **output layer**, where we will have the value that we want, the final result.

Processing the Data

We use the following formula to calculate this raw value for a perceptron:

$$z = \sum i \ (x_i w_i + b)$$

x_i	Are the input values
w_i	Are the weight values that determine the importance of each entry
b	Is the bias, the constant value that helps regulate the output
z	Is the value of the weighted sum realized, the gross value

Once we have the raw data, we will pass it through the activation function to see how much the perceptron will be activated. There are several possible activation functions:

1. **Step Function**

   Formula:

   $$f(z) = 1, if\ z \geq 0$$
   $$f(z) = 0, if\ z < 0$$

   Description:

   - Just turns a perceptron on or off

   Problems:

   - It is not differentiable, which prevents the calculation of the gradient.
   - It does not work well for deep learning.

2. **Sigmoid (Logistics)**

   Formula:

   $$f(z) = \frac{1}{1 + e^{-z}}$$

Description:

- Converts any number to a value between 0 and 1.
- Good for binary classification problems.

Issues:

- May cause gradient vanishing in deep networks.
- Outputs tend to saturate near 0 or 1, making learning difficult.

When use:

- Last layer in binary classification (like neural networks for spam recognition).

3. Tanh (Hyperbolic Tangent)

Formula:

$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Description:

- Similar to Sigmoid, but returns values between -1 and 1.
- Improves learning because negative values are considered, centering the data.

Issues:

- It also suffers from gradient disappearance for extreme values.

When use:

- Better than Sigmoid for hidden networks, as the values are more balanced.

4. ReLU (Rectified Linear Unit)

Formula:

$$f(z) = \max(0, z)$$

Description:

- Resets negative values and keeps positive ones.
- Introduces non-linearity without saturation for positive values.
- Very efficient for deep networks.

Issues:

- Neurons may die if z is always negative.

When use:

- The most commonly used activation function today in the hidden layers of deep neural networks.

5. Leaky ReLU

Formula:

$$f(z) = z,\ if\ z > 0$$
$$f(z) = \alpha z,\ if\ z < 0$$

Description:

- Similar to ReLU, but allows small negative values
- Avoids the problem of dead neurons.

When use:

- Good alternative to ReLU when there are problems with dead neurons.

6. Softmax

Formula:

$$f(z_i) = \frac{e^{z_i}}{\sum_j (e^{z_j})}$$

Description:

- Transforms a vector of numbers into probabilities, where the sum of the outputs is always 1.
- Good for multiclass classification.

When use:

- Last layer of neural networks for multi-class classification.

Function	Output Range	Commom Use	Issues
Step	{0,1}	Classical Perceptron	Non differentiable
Sigmoid	(0,1)	Binary classification	Saturation, gradient fades
Tanh	(-1,1)	Hidden layers	Saturation, gradient fades
ReLU	[0,∞)	Hidden layers of deep networks	Dead neurons
Leaky ReLU	(-∞,∞)	Hidden layers (improved version of ReLU)	Hyperparameter needs to be tuned
Softmax	(0,1)	Last layer of multiclass classification	Computationally expensive

Following this pattern, the output of one layer becomes the input of another until we finally reach the final output.

Measure Network Errors

Now that we have obtained a value from the network output, we need to check how correct this value is, for this we compare it with the correct data if it is a supervised model, for this we use the following error calculation formulas:

1. **Mean Squared Error – MSE**

   Formula:

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

y_i	is the real value
$prev_i$	is the predicted value
n	is the number of examples

- Usage: Common in regression problems, where we want to predict continuous values.

2. Mean Absolute Error – MAE

Formula:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - pred_i|$$

- Usage: It is also common in regression, but tends to be more robust to outliers than MSE.

3. Cross-Entropy Loss

Formula:

$$L = -\sum_{i=1}^{n} y_i \log(pred_i)$$

y_i	is the real distribution (usually 0 or 1 in binary classification).
$pred_i$	is the predicted probability (usually obtained by a sigmoid or softmax function)

- Usage: Widely used in classification problems, especially with deep neural networks.
- For example, it compares the probability (confidence) of the true class that the network returned with the actual value, this makes the error more severe based on how much less the network thinks the true class is the true one.

True Learning
Backpropagation

Backpropagation is the algorithm that adjusts the weights of a neural network to minimize the error between the network's predictions and the actual values (labels). It does this by passing the error back through the network (from the output layer to the previous layers) and calculating how much each weight contributed to that error, so that it can be adjusted accordingly.

Now that we have the error, let's start backpropagating that error to adjust the network weights.

Backpropagation uses the chain rule of differential calculus. The idea is to calculate the derivative of the error with respect to the weights, that is, how much the error changes if we change a weight slightly. This tells us how to adjust each weight to minimize the error.

What are we trying to do?

- We want to know how much each weight contributed to the error.
- We'll calculate the gradient of the error with respect to each weight, and from that, we'll know how much we need to adjust each weight to improve the prediction.

Examples:

Sigmoid + Cross-Entropy Loss

1) **Network Configuration**:

   a) Input: 0.5
   b) Initial Weight: 0.2
   c) Activation Function at the Output: Sigmoid
   d) Expected Output: 1 (True Class)
   e) Learning Rate: 0.1

2) **Forward Pass**:

a) Multiply input by weight:

Input * Weight = 0.5 * 0.2 = 0.1//

b) Apply Sigmoid activation:

Predicted Output = 1 / (1 + e^(-0.1)) = 0.525

c) Calculate the error using Cross-Entropy Loss:

Error = -log(Predicted Output) = -log(0.525) = 0.644

3) **Backward Pass**:

a) Calculate the gradient of the output (error):

Gradient Output = Predicted Output - True Output = 0.525 - 1 = -0.475

b) Derivative of the Sigmoid function:

Derivative of Sigmoid = Predicted Output * (1 - Predicted Output) = 0.525 * (1 - 0.525) = 0.249

c) Calculate the weight gradient:

Weight Gradient = Gradient Output * Derivative of Sigmoid * Input Weight Gradient = (-0.475) * 0.249 * 0.5 = -0.118

d) Update the weight:

New Weight = Weight - Learning Rate * Weight Gradient

New Weight = 0.2 - (0.1 * -0.118) = 0.2 + 0.0059 = 0.2059

ReLU + Cross-Entropy Loss

1) **Network Configuration**:

a) Input: 0.5

b) Initial Weight: 0.2
c) Activation Function: ReLU
d) Expected Output: 1
e) Learning Rate: 0.1

2) **Forward Pass**:

a) Multiply input by weight:

Input * Weight = 0.5 * 0.2 = 0.1

b) Apply ReLU activation:

Predicted Output = max(0, Weighted Input) = max(0, 0.1) = 0.1

c) Calculate the error using Cross-Entropy Loss:

Error = -log(Predicted Output) = -log(0.1) = 2.3

3) **Backward Pass**:

a) Calculate the gradient of the output (error):

Gradient Output = Predicted Output - True Output = 0.1 - 1 = -0.9

b) Derivative of ReLU function:

Derivative of ReLU = 1 (since Weighted Input > 0)

c) Calculate the weight gradient:

Weight Gradient = Gradient Output * Derivative of ReLU * Input
Weight Gradient = (-0.9) * 1 * 0.5 = -0.45

d) Update the weight:

New Weight = Weight - Learning Rate * Weight Gradient New
Weight = 0.2 - 0.1 * -0.45 = 0.2 + 0.045 = 0.245

New Weight: 0.245

Softmax + Cross-Entropy Loss

1) Network Configuration:

    a) Input 1: 1.0
    b) Input 2: 2.0
    c) Weight 1: 0.3
    d) Weight 2: -0.2
    e) Activation Function: Softmax Expected
    f) Output: [0, 1] (Class 2)
    g) Learning Rate: 0.1

2) Forward Pass:

    a) Calculate the weighted inputs:

        Weighted Input 1 = Input 1 * Weight 1 = 1.0 * 0.3 = 0.3 Weighted Input 2 = Input 2 * Weight 2 = 2.0 * -0.2 = -0.4

    b) Apply Softmax activation:

        Output 1 = e^0.3 / (e^0.3 + e^-0.4) = 0.645 Output 2 = e^-0.4 / (e^0.3 + e^-0.4) = 0.355

    c) Calculate the error using Cross-Entropy Loss:

        Error = -sum(Expected Output * log(Predicted Output)) Error = -log(0.355) = 1.034

1) Backward Pass:

    a) Calculate the gradient of the output (error):

        Gradient Output 1 = Predicted Output 1 - Expected Output 1 = 0.645 - 0 = 0.645
        Gradient Output 2 = Predicted Output 2 - Expected Output 2 = 0.355 - 1 = -0.645

    b) Calculate the weight gradients:

Weight Gradient 1 = Gradient Output 1 * Input 1 = 0.645 * 1.0 = 0.645
Weight Gradient 2 = Gradient Output 2 * Input 2 = -0.645 * 2.0 = -1.29

c) Update the weights:
New Weight 1 = 0.3 - 0.1 * 0.645 = 0.2355
New Weight 2 = -0.2 - 0.1 * (-1.29) = -0.071

New Weights: 0.2355, -0.071

Gradient Descent

How to Avoid Local Minima in Gradient Descent?

When training machine learning models, especially neural networks, we often face the challenge of getting stuck in **local minima** or **saddle points** during optimization. To address this issue and help the algorithm find the best possible solution, several techniques are employed in the context of **Gradient Descent**.

Gradient Descent Overview: Gradient Descent is an optimization algorithm used to minimize a loss function by iteratively updating the model parameters (weights). The model adjusts its weights in the direction of the negative gradient of the loss function to minimize the error. However, the loss function can have many local minima, and the optimization process may converge to one of these local minima rather than the global minimum.

To help overcome this challenge, the following techniques are commonly applied:

1. **Momentum**
    a. **Momentum** helps the optimization process by "smoothening" the updates, making them less likely to get stuck in local minima. It introduces a concept of **velocity**, where previous gradients are remembered and used to accelerate the search for the minimum.
    b. The algorithm updates the weights based not only on the current gradient but also on a fraction of the previous gradient.
    c. This helps the model avoid oscillating or getting stuck in shallow local minima, allowing it to escape and make faster progress in the right direction.

Formula for Momentum:

d. **Momentum Velocity** = (Momentum Factor × Previous Velocity) + (1 - Momentum Factor) × Current Gradient
e. **Updated Weight** = Current Weight - (Learning Rate × Momentum Velocity)
f. Where:
    i. **Momentum Velocity** is the "speed" at which the weight changes,
    ii. **Momentum Factor** is a hyperparameter (usually between 0 and 1),
    iii. **Current Gradient** is the gradient of the loss at the current step,
    iv. **Current Weight** is the weight at the current step,
    v. **Learning Rate** controls the size of the updates.

2. **Adaptive Learning Rate (Adam, RMSprop, etc.)**
    a. **Adaptive Learning Rate** techniques such as **Adam** or **RMSprop** dynamically adjust the learning rate based on the gradients during training. These methods adapt the learning rate for each parameter in the model.
    b. By adjusting the learning rate, these algorithms allow the model to take larger steps when gradients are small and smaller steps when gradients are large, which helps the algorithm avoid getting stuck in flat or sharp local minima.
    c. This adaptability makes these algorithms more efficient and effective in complex optimization landscapes, such as those encountered in deep learning models.

Adam (Adaptive Moment Estimation):

d. Adam combines the benefits of **Momentum** and **RMSprop** by maintaining both first-order (mean) and second-order (variance) moments of the gradients.
e. This allows it to adjust the learning rate individually for each parameter based on the historical gradients, improving convergence.

Adam Update Rule:

f. **Momentum Estimate** = (Momentum Factor 1 × Previous Momentum Estimate) + (1 - Momentum Factor 1) × Current Gradient
g. **Velocity Estimate** = (Momentum Factor 2 × Previous Velocity Estimate) + (1 - Momentum Factor 2) × (Current Gradient)^2
h. **Corrected Momentum Estimate** = Momentum Estimate / (1 - Momentum Factor 1^t)

i. **Corrected Velocity Estimate** = Velocity Estimate / (1 - Momentum Factor 2^t)
j. **Updated Weight** = Current Weight - (Learning Rate × Corrected Momentum Estimate / (sqrt(Corrected Velocity Estimate) + Small Constant))
k. Where:
    i. **Momentum Estimate** is the "mean" of the past gradients,
    ii. **Velocity Estimate** is the "variance" of the gradients,
    iii. **Momentum Factor 1** and **Momentum Factor 2** are hyperparameters for the estimates,
    iv. **Small Constant** prevents division by zero.

3. **Intelligent Weight Initialization**
    a. Proper **initialization of weights** is crucial in avoiding situations where the network gets stuck in poor local minima. If weights are initialized too large or too small, it can lead to slow learning or the network being trapped in bad regions of the loss function.
    b. Several weight initialization methods can help improve training efficiency and help the algorithm avoid bad local minima:
        i. **Xavier Initialization** (also known as Glorot initialization) is often used for sigmoid and tanh activation functions. It scales the weights based on the number of inputs and outputs.
        ii. **He Initialization** is often used with ReLU activation functions. It scales the weights according to the number of input units.
        iii. **Random Initialization** involves setting weights randomly within a small range of values to break symmetry and avoid starting in flat regions of the loss surface.