NEURAL NETWORKS DECODED

From Perceptrons to Backpropagation: A Practical Guide

Minor In AI, IIT Ropar
1st May, 2025

Welcome, Future AI Enthusiasts!

Imagine you're blindfolded and someone offers you a sweet. What sweet is it that you can *always* identify, no matter what? Think about it. It's not just "ladoo," because there are Mothichur ladoos and Besan ladoos. It's not just "Rasgulla," because there are different kinds of Rasgullas too.
What is *that one sweet* that you can identify for certain? It requires a lot of your experiences, and feelings, and bias that you have with your previous memories. And if you're wrong? You correct yourself the next time.
That's what we're going to dive into: how machines, specifically neural networks, "taste" data, learn from their "mistakes," and become increasingly accurate at identifying patterns. This book is your practical guide to understanding the fundamental building blocks of these networks, starting with the simplest decision-making unit: the perceptron.

1 The Neuron - A Decision-Making Unit

Let's begin with the most basic building block, the neuron. Now, don't be fooled – a single neuron isn't intelligent. It's simply a decision-making unit. It receives inputs, processes them, and produces an output. The magic happens when we connect many of these neurons together.

Think of each of your senses as a neuron. When you are blindfolded, there are neurons in your brain that receive sensory input. If the neurons collectively reach a particular experience threshold, it makes a decision. The "sweet" example above is like a neuron receiving different sensory inputs that are collectively making a decision to identify the sweet.

But how does this "decision" happen?

1.1 Weights and Bias: The Neuron's Memory

Every piece of information in our memory has a certain weight, a bias. You might remember visiting a place vaguely, or eating a food vaguely, indicating your brain assigns different weights to different features. Similarly, our artificial neuron assigns **weights** to each input it receives. These weights represent the importance of each input in the decision-making process.

```python
# Example: Initializing weights
weights = np.zeros(2) # Starting with zero weights
bias = 0              # Initial bias

print(f"Initial Weights: {weights}")
print(f"Initial Bias: {bias}")
```

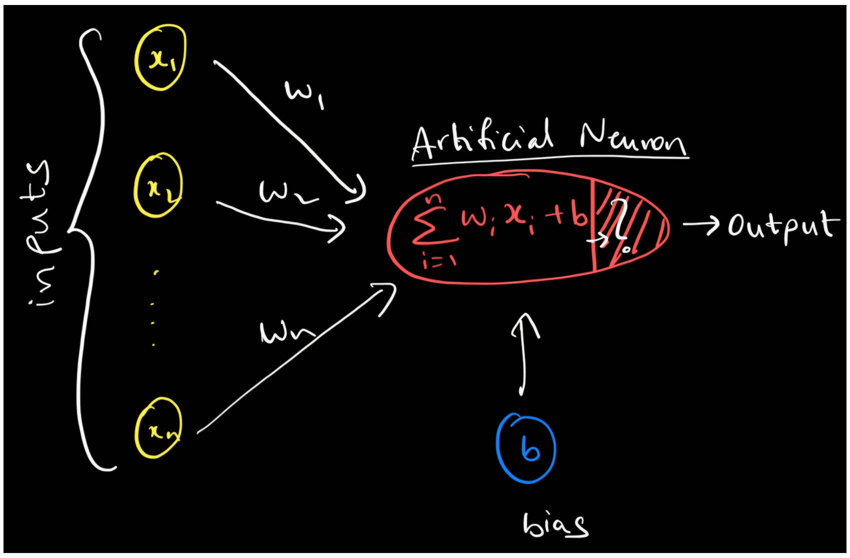

Figure 1: Artificial Neuron Structure (PS: If you don't understand the inside of the neuron, that is okay for now!)

Bias is another crucial component. It's like a constant value that helps the neuron activate even when all inputs are zero. It shifts the activation function to the left or right, giving the neuron more flexibility.

1.2 Making a Prediction: The Weighted Sum

The neuron calculates a **weighted sum** of its inputs, adds the bias, and then passes the result through an **activation function.** The activation function determines the output of the neuron.

```python
import numpy as np

# Sample Input
X = np.array([1,0])
# Sample Weights
weights = np.array([0.5, 0.5])
# Sample Bias
bias = 0.5

# Calculating z
z = np.dot(X, weights) + bias
print(f"Weighted Sum (z): {z}")
```

1.3 The Step Function: A Simple Decision

One of the simplest activation functions is the **step function**. It outputs 1 if the weighted sum is greater than or equal to zero, and 0 otherwise. This is a binary decision: yes or no, true or false.

```python
def step(x):
    return 1 if x >= 0 else 0
```

```python
# Using z from the last example:
y_pred = step(z)
print(f"Predicted Output: {y_pred}")
```

2 The Perceptron - Learning to Classify

A perceptron is a single-layer neural network with a step activation function. It's used for binary classification problems – deciding whether an input belongs to one class or another.

2.1 The NAND Gate Example

Let's train a perceptron to implement a NAND gate. A NAND gate outputs 1 only if *both* inputs are 0, otherwise, it outputs 0. This also applies to *any* possible decision you are making.

First, here's the ground truth of what it should look like.

Input 1	Input 2	Output (NAND)
0	0	1
0	1	1
1	0	1
1	1	0

Table 1: NAND Gate Truth Table

2.2 Training the Perceptron: Iterative Updates

The perceptron learns by adjusting its weights and bias based on the errors it makes. This iterative process is called **training**.

We follow these steps:

1. **Initialize:** Start with random weights and bias (often set to zero initially for simplicity).
2. **Predict:** For each input, calculate the weighted sum and apply the step function to get a prediction.
3. **Calculate Error:** Compare the prediction to the actual output and calculate the error.
4. **Update:** Adjust the weights and bias based on the error and a **learning rate.** The learning rate controls the size of the adjustments.
5. **Repeat:** Repeat steps 2-4 for a certain number of **epochs** (iterations over the entire training dataset) or until the error is sufficiently small.

```python
import numpy as np

# NAND input
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# NAND output
y = np.array([1, 1, 1, 0])

# Initialize weights and bias
weights = np.zeros(2)
bias = 0
learning_rate = 1
epochs = 10

# Step activation function
def step(x):
    return 1 if x >= 0 else 0

# Training loop
for epoch in range(epochs):
    print(f"\nEpoch {epoch+1}")
    total_error = 0
    for i in range(len(X)):
        z = np.dot(X[i], weights) + bias
        y_pred = step(z)
        error = y[i] - y_pred

        # Update weights and bias
        weights += learning_rate * error * X[i]
        bias += learning_rate * error
        total_error += abs(error)

        print(f"Input: {X[i]}, Predicted: {y_pred}, Error: {error}, Weights:
        ↪  {weights}, Bias: {bias}")

    if total_error == 0:
        print("\nTraining converged.")
        break
```

2.3 Convergence: Finding the Right Values

The goal of training is to find the weights and bias that minimize the error. When the error is zero (or very close to zero), we say that the perceptron has **converged.**

But sometimes, no matter how long we train, the perceptron doesn't converge. This happens when the data is not **linearly separable**.

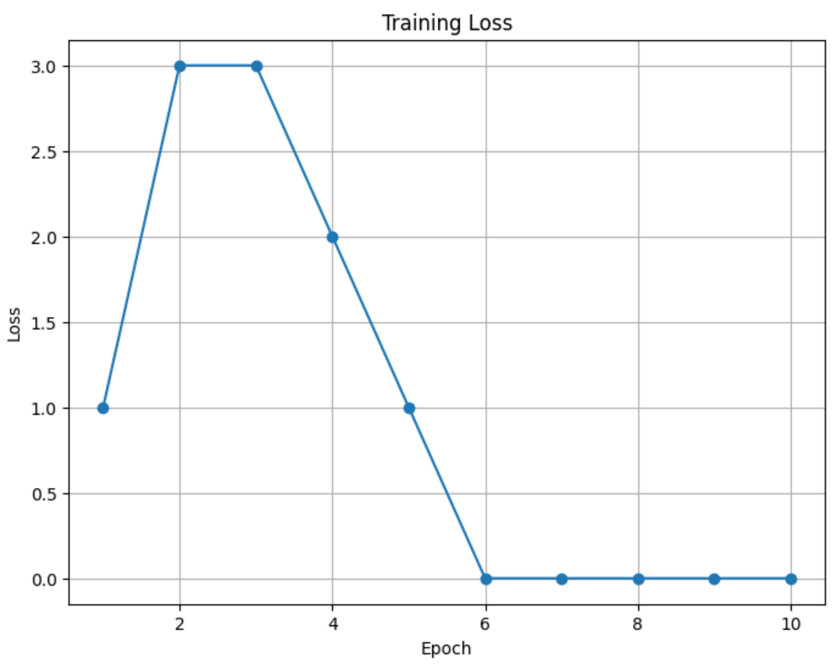

Figure 2: Error Reduction During Training

3 Linear Separability and the XOR Problem

A problem is linearly separable if we can draw a straight line (or hyperplane in higher dimensions) to separate the different classes. The NAND gate *is* linearly separable, which is why the perceptron could learn it.

However, the XOR (exclusive OR) gate is *not* linearly separable. XOR outputs 1 only if one input is 1 and the other is 0. If both inputs are the same (both 0 or both 1), XOR outputs 0.

Let's try to train a perceptron on the XOR problem:

```
import numpy as np

# XOR input: 2 features
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# XOR output
y = np.array([0, 1, 1, 0])

# Initialize weights and bias
weights = np.zeros(2)
bias = 0
learning_rate = 1
epochs = 10
```

```
# Step activation function
def step(x):
    return 1 if x >= 0 else 0

# Training loop
for epoch in range(epochs):
    print(f"\nEpoch {epoch+1}")
    total_error = 0
    for i in range(len(X)):
        z = np.dot(X[i], weights) + bias
        y_pred = step(z)
        error = y[i] - y_pred

        # Update weights and bias
        weights += learning_rate * error * X[i]
        bias += learning_rate * error
        total_error += abs(error)

        print(f"Input: {X[i]}, Predicted: {y_pred}, Error: {error}, Weights:
        ↪  {weights}, Bias: {bias}")

    if total_error == 0:
        print("\nTraining converged.")
        break
```

If you run this code, you'll see that the error never converges to zero, even after many epochs. This is because a single perceptron can't learn non-linearly separable problems like XOR.

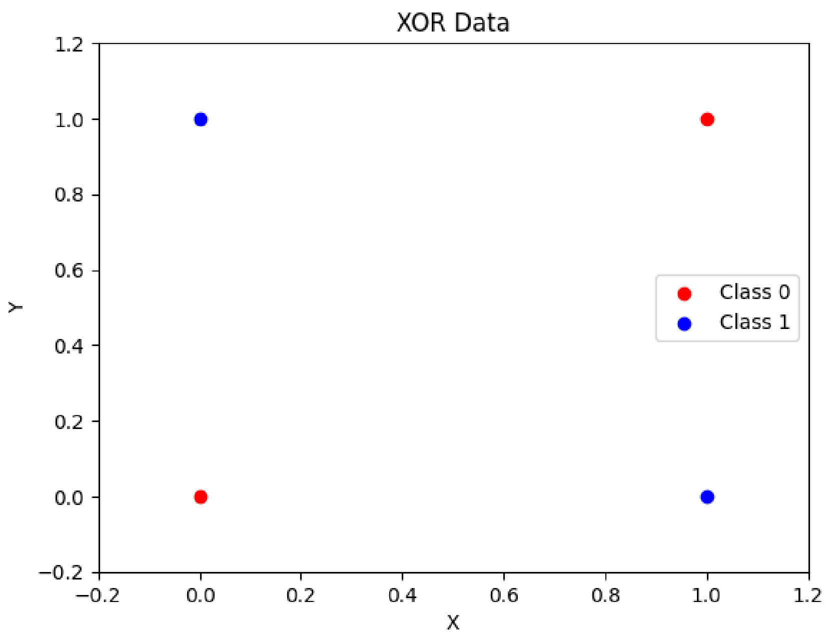

Figure 3: XOR Problem: Cannot be separated by a straight line

3.1 Beyond Linearity: Enter the Sigmoid Function

To handle non-linearly separable problems, we need a more powerful tool. One way to make the activation function more non-linear is by using the **sigmoid function.** The sigmoid function outputs a value between 0 and 1, representing a probability.

This is a big change from the "yes or no" decisions that the linear step function takes. With non-linearity, we can now classify inputs with a *confidence level.*

4 The Sigmoid Neuron: A Probabilistic Decision-Maker

Instead of a binary output, the sigmoid function gives us a probability. This is especially useful for classification problems.

4.1 The Sigmoid Function

The sigmoid function is defined as:

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

Where z is the weighted sum of the inputs and e is the exponential function.

```python
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

#Example Usage
z = 1.5
output = sigmoid(z)
print(f"Sigmoid Output for z={z}: {output}")
```

4.2 Training with Sigmoid: Gradient Descent

When we use the sigmoid function, we need to adjust our training process. Instead of simply adding the error to the weights and bias, we use **gradient descent** to find the optimal values.

```python
import numpy as np

# Input data (AND gate)
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# Expected output
y = np.array([[0], [0], [0], [1]])
```

```python
# Sigmoid activation function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Derivative of sigmoid (for backprop)
def sigmoid_derivative(z):
    return sigmoid(z) * (1 - sigmoid(z))

# Initialize weights and bias
weights = np.random.randn(2, 1)
bias = 0
learning_rate = 0.1
epochs = 10000

# Training loop using gradient descent
for epoch in range(epochs):
    # Forward pass
    z = np.dot(X, weights) + bias
    y_pred = sigmoid(z)

    # Compute loss (MSE)
    loss = np.mean((y - y_pred) ** 2)

    # Backward pass (gradients)
    error = y_pred - y
    d_weights = np.dot(X.T, error * sigmoid_derivative(z))
    d_bias = np.sum(error * sigmoid_derivative(z))

    # Update weights and bias
    weights -= learning_rate * d_weights
    bias -= learning_rate * d_bias

    # Print every 1000 epochs
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Final output
print("\nFinal predictions:")
for i in range(len(X)):
    z = np.dot(X[i], weights) + bias
    y_pred = sigmoid(z)
    print(f"Input: {X[i]}, Output: {y_pred[0]:.4f}")
```

4.3 The Loss Function: Measuring Error

To guide the gradient descent, we need a **loss function** to quantify the error. A common loss function is the **mean squared error (MSE)**.

5 Multi-Layer Neural Networks

To solve non-linearly separable problems like XOR, we need to go beyond a single perceptron. We need to build a multi-layer neural network. In the next section, the same idea is used. But instead of one neuron, we are using two, and assigning values to the input for each of the neurons.

```python
# Understanding Backpropogation

# Inputs: Ears, Whiskers, Fur
inputs = [1, 1, 1]

# Hidden Layer Weights (2 neurons, each with 3 inputs)
hidden_weights = [
    [0.6, 0.5, 0.4],  # Neuron 1
    [0.3, 0.7, 0.8]   # Neuron 2
]
hidden_thresholds = [0.9, 1.2]

# Output Layer Weights (2 inputs from hidden layer)
output_weights = [0.6, 0.9]
output_threshold = 1.0

# Step 1: Compute hidden layer outputs
hidden_outputs = []
for i in range(2):
    sum_hidden = sum([inputs[j] * hidden_weights[i][j] for j in range(3)])
    output = 1 if sum_hidden >= hidden_thresholds[i] else 0
    hidden_outputs.append(output)

# Step 2: Compute output layer result
final_sum = sum([hidden_outputs[i] * output_weights[i] for i in range(2)])
final_output = 1 if final_sum >= output_threshold else 0

# Step 3: Print result
if final_output == 1:
    print("Mini-brain says: It's a cat!")
else:
    print("Mini-brain says: Not a cat.")
```

5.1 Layers: Input, Hidden, and Output

A multi-layer neural network consists of:

- **Input Layer:** Receives the raw data.
- **Hidden Layer(s):** Perform non-linear transformations of the input data. A network can have multiple hidden layers.
- **Output Layer:** Produces the final prediction.

5.2 Connecting the Neurons

Each neuron in one layer is connected to every neuron in the next layer. The connections have weights associated with them, just like in the perceptron.

5.3 Forward Propagation

The process of feeding data through the network, from input to output, is called **forward propagation**.

Final Thoughts

This book has taken you on a journey from the simplest decision-making unit, the perceptron, to the foundation of multi-layer neural networks. You've learned about linear separability, the sigmoid function, and the importance of weights, biases, and activation functions.

Remember, the world of neural networks is vast and ever-evolving. This is just the beginning! Keep experimenting, keep learning, and keep building!