What is a Neural Network

Nandinii Yeleswarapu

September 1, 2025

Introduction

This blog post is my way of summarizing what I learned on my journey to building a solid foundation in machine learning — from linear models to the training dynamics of deep networks. These concepts shaped how I understand why we need machine learning. Here, I focus specifically on what a neural network really is and why it matters.

Linear Models and the Need for Non-Linearity

Lets start by looking at a simple model, functions like bitwise `AND` and `OR`. These functions are linearly separable and thus can be modeled using a straight line, or more formally an affine transformation:

$$y = Wx + b$$

Of course, you don't actually need a neural network (or all the heavy computational machinery) just to plot or identify something as simple as `AND` or `OR`. But these examples are useful because they highlight the limits of linear models in the clearest way possible.

Then comes the `XOR` problem. No matter how you draw a line, you can't separate the two classes, they are just not linearly separable. Linear models alone just aren't enough. This is why we need **non-linear activation functions** to break free from linear limitations and allow networks to capture more complex patterns.

What Makes a Neural Network

When I started building small networks, I realized they are made up of a few core ingredients. Each of these plays a very specific role:

- **Layers (the model class):** A layer is the basic block of a NN. At minimum it consists of a linear (affine) transformation and an activation function to introduce non-linearity. Stacking many layers allows the network to transform data step by step, moving from simple patterns (like edges in an image) to very complex ones (like faces or objects). This stacking ability is what gives neural networks their power as *universal function approximators*.

- **Loss functions (measuring mistakes):** The loss function measures the correctness of the models predictions. It tells us how far off for the correct value was the network's learned value. There are several loss function, each suited for a different type of task: - **Mean Squared Error (MSE):** penalizes large differences heavily, which makes it a good fit for regression (predicting continuous values). - **Negative Log-Likelihood (NLL):** compares predicted probability distributions to actual labels, making it natural for classification problems. - **Hinge Loss:** tries to keep decision boundaries wide and margins strong, which is why it shows up in SVMs.

- **Activation functions (adding non-linearity):** Without them, stacking layers would just collapse back into one big linear transformation. Activations bend the decision boundaries, letting networks capture complexity. - **ReLU (Rectified Linear Unit):** simple, fast, and avoids most vanishing gradient issues. It outputs zero for negative values and grows linearly for positives. - **Sigmoid:** squashes outputs between $(0, 1)$, making it useful for probabilities, but gradients vanish near 0 or 1. - **Tanh:** similar to sigmoid but zero-centered, mapping values into $(-1, 1)$, which often helps training converge faster.

- **Optimizers (learning from mistakes):** Once the loss function says "you're this wrong," the optimizer decides *how to change the weights* to improve next time. They use gradients — slopes of the loss with respect to weights — as a compass. - **Stochastic Gradient Descent (SGD):** takes small steps in the opposite direction of the gradient. It's simple but can be slow and get stuck in valleys. - **SGD with momentum:** remembers past updates like pushing a ball downhill — it smooths the path and helps escape small bumps. - **Adam:** adapts the step size for each parameter individually,

combining momentum and adaptive learning rates, which often leads to faster and more reliable training.

Together, these components form the core recipe: layers to transform data, a loss to measure mistakes, activations to add flexibility, and optimizers to steadily improve performance.

Summary

At its core, a neural network is just a chain of functions: the output of one becomes the input of the next. Each layer transforms the data a little more until you finally reach the desired output.

For those with a programming background, this idea maps neatly to familiar patterns:

- **Scala: for-comprehension** — each step pipes its result forward.

```
for {
  layer1 <- func1()
  layer2 <- func2()
  output <- func3()
} yield output
```

- **Java: builder pattern** — each method call adds or modifies the object, and the final call produces the result.

```
String result = new StringBuilder()
    .append("Neural ")
    .append("Networks ")
    .append("Rock!")
    .toString();
```

Both examples show the same principle: each function (or method call) builds on the last, and the true power comes from chaining them together — exactly how neural networks work.

References and Resources

Much of what I learned while building this foundation came from the course: CS182: Deep Learning at UC Berkeley (Spring 2021).