Unlocking the Matrix: Your Guide to Supervised Learning, MLPs, and the Path to CNNs!

Minor In AI, IIT Ropar

21st May, 2025

Introduction: The Deep Learning Journey

Welcome, budding AI enthusiast, to the exciting world of Deep Learning! You're about to embark on a journey that will unveil the magic behind systems that can recognize faces, understand spoken words, and even generate art. At the heart of these incredible capabilities lies the deep neural network – a complex structure built from simpler components.

In this document, we won't jump straight into the most complex architectures. Instead, we'll start from the very foundation, understanding the core principles of how machines learn from data. We'll build our way up, starting with the fundamental concept of supervised learning, exploring the power (and limitations) of the basic building block – the artificial neuron – and then see how combining these blocks leads to more powerful models like Multi-Layer Perceptrons (MLPs). This journey will naturally pave the way for understanding more advanced networks like Convolutional Neural Networks (CNNs), which have revolutionized tasks like image recognition.

Our path will be like building with LEGOs. We'll first get familiar with the basic bricks, understand their purpose, and then learn how to combine them in increasingly sophisticated ways. By the end, you'll have a solid conceptual understanding of how these powerful models work.

Let's start with a simple, everyday example.

1 Chapter 1: The Big Picture - Learning from Data (Supervised Learning)

Imagine you have a friend who always seems to know whether to wear a jacket based on the temperature outside. You want to learn their secret! You start collecting data: you record the temperature each day (your *input*) and whether they wore a jacket (your *output*).

Temperature (°C)	**Jacket On/Off**
25	Off
10	On
18	Off
5	On
...	...

Table 1: Daily temperature and jacket status.

You're essentially creating a dataset where you have pairs of inputs (Temperature) and their corresponding correct outputs (Jacket On/Off). This is the essence of **Supervised Learning**. You have the *input* data (let's call it $\mathbf{X}$) and the corresponding correct *output* or 'label' (let's call it $\mathbf{Y}$).

Your goal is to figure out a rule or a function (let's call it F) that can take the temperature ($\mathbf{X}$) as input and predict whether a jacket should be on or off ($\mathbf{Y}$). If you find this function F, you could then predict whether to wear a jacket even on a day you didn't record data, just by knowing the temperature.

However, in the real world, finding the *perfect* function F that works for *all* possible inputs and exactly matches the output $\mathbf{Y}$ is often impossible. The data might be noisy, or the relationship might be too complex. So, instead of finding the perfect F, we aim to find an *approximate* function, let's call it $\hat{F}$ (pronounced F-hat).

When we give our approximate function $\hat{F}$ an input $\mathbf{X}$, it gives us a *predicted* output, which we'll call $\hat{\mathbf{Y}}$ (pronounced Y-hat).

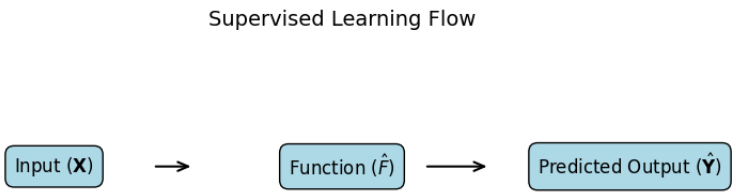

Figure 1: Diagram showing Input ($\mathbf{X}$) $\rightarrow$ Function ($\hat{F}$) $\rightarrow$ Predicted Output ($\hat{\mathbf{Y}}$).

Now, since $\hat{F}$ is just an approximation, our predicted output $\hat{\mathbf{Y}}$ won't always be exactly the same as the actual correct output $\mathbf{Y}$. The difference between $\mathbf{Y}$ and $\hat{\mathbf{Y}}$ is our 'mistake', which we call **Loss**. We want to find an $\hat{F}$ that minimizes this mistake, or minimizes the **Loss**.

Think about our jacket example again. Maybe our $\hat{F}$ predicts "Off" at 12°C, but your friend actually wore a jacket ($\mathbf{Y}$ was "On"). That's a mistake, a loss.

We don't just care about the loss for one single day (one example). We want our function $\hat{F}$ to perform well on *average* across *all* the days we recorded data. So, we calculate the loss for each example and then take the average loss over our entire dataset. Our ultimate goal in supervised learning is to adjust our function $\hat{F}$ so that this *average loss* is as small as possible.

Now, how do we "adjust" the function $\hat{F}$? Our function $\hat{F}$ usually has some internal settings or knobs that we can tune. These settings are called **parameters**. For instance, in a simple line like $\hat{Y} = mX + c$, 'm' and 'c' are the parameters we can change. By changing these parameters, we change the function $\hat{F}$ and thus change the predicted output $\hat{\mathbf{Y}}$, which in turn changes the Loss.

Here's a small Python snippet to illustrate such a function:

```python
def linear_function_F_hat(X, m, c):
  """
  A simple linear function F_hat.
  X: input feature
  m: slope (parameter)
  c: intercept (parameter)
  Returns Y_hat (predicted output)
  """
  Y_hat = m * X + c
  return Y_hat

# Example usage:
```

```
temperature = 20 # degrees Celsius
slope_param = 0.8 # example learned parameter
intercept_param = -5 # example learned parameter

predicted_value = linear_function_F_hat(temperature, slope_param,
    intercept_param)
print(f"For X={temperature}, Y_hat={predicted_value}")
```

Listing 1: A simple Python function representing $\hat{F}$

So, the core problem of supervised learning boils down to this: Given a dataset of inputs ($\mathbf{X}$) and outputs ($\mathbf{Y}$), find the parameters of our function $\hat{F}$ that minimize the average loss between the actual outputs ($\mathbf{Y}$) and the predicted outputs ($\hat{\mathbf{Y}}$).

Depending on the type of output $\mathbf{Y}$ we are trying to predict, supervised learning tasks are typically divided into two main categories:

1. **Regression:** When the output $\mathbf{Y}$ is a continuous number.
   - *Example:* Predicting the price of a house based on its size and location. The price can be any value within a range.
   - *Typical Loss:* Mean Squared Error (MSE), which measures the average squared difference between the actual and predicted values.
2. **Classification:** When the output $\mathbf{Y}$ belongs to a finite set of categories or classes.
   - *Example:* Predicting whether an email is spam or not spam (two categories). Or classifying an image as a cat, dog, or bird (multiple categories).
   - *Typical Loss:* Cross-Entropy Loss (specifically Categorical Cross-Entropy for more than two classes), which is well-suited for measuring the error in predicting probabilities for different classes.

Our jacket example is a classification problem – the output is either "On" or "Off", which are two distinct categories. Let's focus on classification as we build our first building block.

2 Chapter 2: Your First Building Block - The Artificial Neuron (Logistic Regression)

To tackle classification problems, especially those with just two categories (like jacket on or off, spam or not spam, positive or negative), we can use a simple yet powerful model called **Logistic Regression**.

Let's revisit the temperature and jacket example. If you plot the data, you might see a general trend: low temperatures mean "On", high temperatures mean "Off". But what about the temperatures in the middle? This is an area of *uncertainty*. At 12°C, maybe your friend sometimes wears it and sometimes doesn't, depending on other factors (wind, how they feel, etc.).

This uncertain region is key. Instead of directly predicting "On" or "Off", which are just labels (0 or 1), it's often more useful to predict the *probability* of the jacket being "On" given the temperature. A probability is a number between 0 and 1. A probability close to 1 means it's very likely "On", a probability close to 0 means it's very likely "Off", and a probability around 0.5 means it's uncertain.

How do we convert our input (Temperature, which can be any number) into an output that is always between 0 and 1 (a probability)? We use a special function called the **Sigmoid function** (also known as the Logistic function). Its graph looks like a stretched-out 'S'.

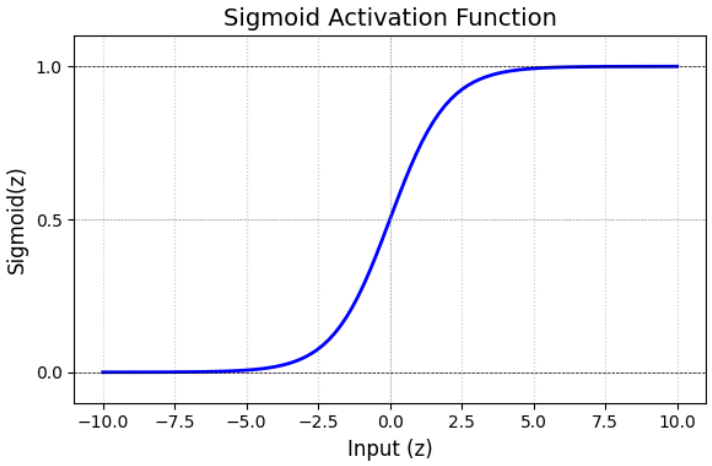

Figure 2: Graph of the Sigmoid function, showing it maps any input value to an output between 0 and 1.

The Sigmoid function is perfect for turning a raw score (which can be any number) into a probability. In Logistic Regression, we first calculate a weighted sum of our inputs (if we had more features than just temperature, like wind speed, we'd include them here). For a single temperature input X and a weight W, this sum might look like $WX + b$ (where 'b' is a bias term, another parameter we learn). Then, we pass this sum through the Sigmoid function. The output is the probability that the input belongs to the 'positive' class (e.g., Jacket On).

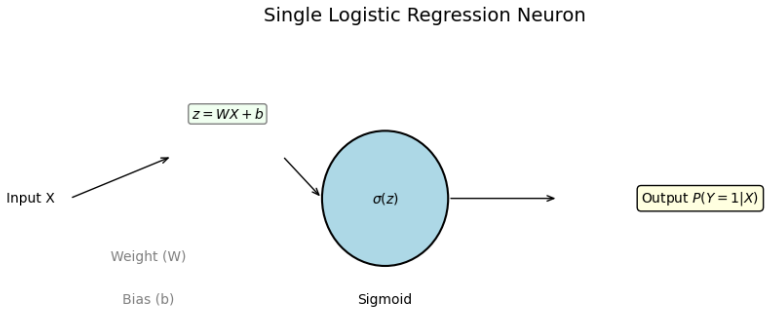

Figure 3: Diagram of a single Logistic Regression neuron: Input (X) → Multiply by Weight (W) → Add Bias (b) → Pass through Sigmoid function → Output ($P(Y = 1|X)$).

If the output probability is greater than a certain threshold (usually 0.5), we predict the positive class ("On"). If it's less than the threshold, we predict the negative class ("Off"). This threshold helps us make a final decision based on the probability. We can even adjust this threshold if needed, but 0.5 is a common default.

This structure – taking inputs, computing a weighted sum, and passing it through an activation function (like sigmoid) – is the fundamental unit of many neural networks. It's known as an **Artificial Neuron**.

So, Logistic Regression isn't just a statistical model; it's our very first building block of deep learning – a single artificial neuron capable of performing binary classification.

However, a single artificial neuron, like logistic regression, has a significant limitation: it can only effectively separate data that is *linearly separable*. This means if you plot your data points for the two classes, you can draw a single straight line (or a flat plane in higher dimensions) to perfectly separate them. Many real-world classification problems are not linearly separable.

And even more importantly, many problems involve *more than two* classes.

3 Chapter 3: Facing Real-World Complexity - Multi-Class Challenges

What happens when you have more than two categories to classify? Imagine classifying types of fruit: Apple, Banana, Orange, Grape. This is a **Multi-Class Classification** problem. Our single artificial neuron (logistic regression) can only tell us if something is in *one* group or *not* in that group.

How can we use our binary classifier (the logistic neuron) to solve a multi-class problem? Two common strategies involve using multiple binary classifiers:

1. **One-vs-Rest (OvR):** For each category, we train a binary classifier to distinguish that category from *all* the other categories combined.
   - *Example:* For our fruit classification (Apple, Banana, Orange, Grape), we'd train:
     - Classifier 1: Apple vs. (Banana + Orange + Grape)
     - Classifier 2: Banana vs. (Apple + Orange + Grape)
     - Classifier 3: Orange vs. (Apple + Banana + Grape)
     - Classifier 4: Grape vs. (Apple + Banana + Orange)
   - When classifying a new fruit, we run it through all four classifiers and see which classifier gives the highest probability for its specific class.
   - *Problem:* This approach often leads to the **Class Imbalance Problem**. In Classifier 1 (Apple vs. Rest), if you have 100 Apple pictures, you might have 300 pictures of other fruits. The classifier sees many more examples of the "Rest" class than the "Apple" class, which can make it biased.

   The class imbalance problem can be tricky. If a model is trained on imbalanced data, it might get very high overall accuracy just by predicting the majority class most of the time. For example, if 95% of emails are *not* spam, a model that *always* predicts "not spam" would have 95% accuracy, even though it misses *all* the spam! To evaluate models on imbalanced data, we often look beyond simple accuracy at metrics like **Precision** and **Recall**, which give us a better picture of how well the model performs on the smaller, minority class. This is often visualized using a **Confusion Matrix**, which shows how many examples from each actual class were predicted as each possible class.

2. **One-vs-One (OvO):** We train a binary classifier for *every possible pair* of categories.
   - *Example:* For our fruit classification (Apple, Banana, Orange, Grape), we'd train classifiers for:
     - Apple vs. Banana
     - Apple vs. Orange
     - Apple vs. Grape
     - Banana vs. Orange
     - Banana vs. Grape
     - Orange vs. Grape
   - When classifying a new fruit, we run it through all these pairwise classifiers. Each classifier 'votes' for one of the two classes it's trained on. The class with the most votes wins.

- *Problem:* The number of classifiers grows very quickly as the number of categories increases (it's $N(N-1)/2$ classifiers for N categories). For 10 categories, that's 45 classifiers! This becomes computationally expensive.

Both OvR and OvO try to force binary classifiers into a multi-class world, but they introduce problems – either class imbalance or an explosion in the number of models needed. We need a more direct, integrated way to handle multi-class classification and learn complex, non-linear relationships.

4 Chapter 4: Stacking Up the Power - Towards Multi-Layer Perceptrons (MLPs)

The limitations of a single artificial neuron for multi-class and non-linearly separable data were clear. For a long time, this was a barrier. Then came a crucial theoretical result: the **Universal Approximation Theorem**.

In simple terms, this theorem tells us that a network created by "stacking" enough artificial neurons, organized into layers, can approximate *any* continuous function to any desired level of accuracy. Think of it like how complex musical notes can be created by combining simpler sine waves (the idea behind Fourier Transforms). Similarly, complex relationships in data can be learned by combining the outputs of multiple simple neurons.

So, the solution to handling complex, multi-class problems isn't just using *many* single neurons side-by-side (like in OvR or OvO), but by *connecting* them in layers, where the output of one layer of neurons becomes the input for the next layer. This is the core idea behind **Multi-Layer Perceptrons (MLPs)**, also known as simple Feedforward Neural Networks.

Let's look at a basic example of stacking neurons:

Imagine our input data has three features (X_1, X_2, X_3). Instead of feeding these directly into a single output neuron (like in Logistic Regression, though typically LR has one output neuron for binary classification), we first feed them into a *layer* of intermediate neurons. Let's say we have two neurons in this intermediate layer (we call this a 'hidden' layer).

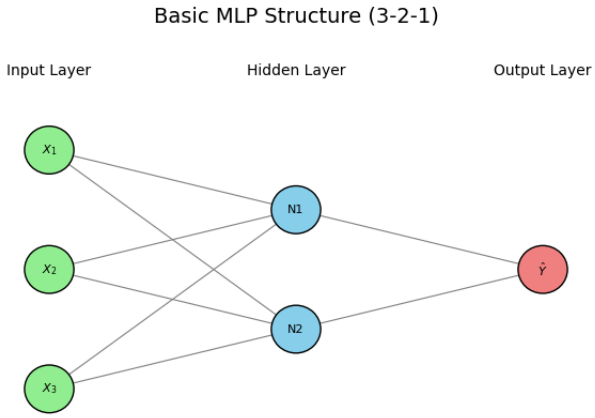

Figure 4: A diagram showing Input Layer (X_1, X_2, X_3) connected to two circles in a Hidden Layer (Neuron 1, Neuron 2), which are then connected to a single circle in the Output Layer.

How does this work?

1. Each input (X_1, X_2, X_3) is connected to *every* neuron in the hidden layer. Each connection has a weight.

2. For Neuron 1 in the hidden layer, we calculate a weighted sum of the inputs: $X_1W_1 + X_2W_3 + X_3W_5$ (using the lecture's simple weight notation, adding bias terms would make this $X_1W_1 + X_2W_3 + X_3W_5 + b_1$). This sum is then passed through an activation function (like Sigmoid, or others we'll learn about later). Let's call its output A_1.

3. For Neuron 2 in the hidden layer, we do the same, using its own set of weights: $X_1W_2 + X_2W_4 + X_3W_6(+b_2)$. This sum is passed through its activation function. Let's call its output A_2.

4. Now, the outputs of the hidden layer neurons (A_1 and A_2) become the inputs for the next layer. In this simple example, we'll connect them to a single output neuron (useful for regression or binary classification probability).

5. The output neuron calculates a weighted sum of *its* inputs (A_1 and A_2): $A_1W_7+A_2W_8(+b_3)$. This sum might be used directly as the final output (for regression) or passed through another activation function (like Sigmoid for binary classification, or Softmax for multi-class classification, which we'll explore soon). Let's call this final output $\hat{Y}$.

By introducing the hidden layer, we've created a network that can learn more complex, non-linear combinations of the input features. The outputs of the hidden layer neurons (A_1, A_2) can be thought of as new, automatically learned features derived from the original inputs. The output layer then combines these new features to make a final prediction.

This is the fundamental concept of stacking neurons to form a multi-layered network. Each layer processes the outputs of the previous layer, building increasingly complex representations of the input data. A Multi-Layer Perceptron (MLP) is essentially a network with one or more hidden layers between the input and output layers.

We've seen how supervised learning provides the framework (minimize loss by adjusting parameters), how the artificial neuron is the basic building block, the challenges of scaling this to real-world multi-class problems, and how stacking neurons based on the Universal Approximation Theorem offers a path to solving more complex tasks.

In the next chapter, we will dive deeper into Multi-Layer Perceptrons, understand how they use activation functions like ReLU (Rectified Linear Unit), see how they solve multi-class classification problems more elegantly, and look at how their operations can be efficiently represented using matrix math. We'll also discuss some of their limitations, which will lead us to the need for specialized architectures like Convolutional Neural Networks – the true bedrock of modern computer vision.

Stay tuned as we continue building!