HyperionDev

Neural Networks

Task

Visit our website

Introduction

Welcome to the neural networks lesson! This lesson introduces the main concepts and practical implementations of neural networks, particularly focusing on multi-layer perceptrons (MLPs). At their core, artificial neural networks (ANNs) are inspired by the human brain's structure and functionality, mimicking the way biological neurons signal to one another. Just as neurons are the building blocks of our nervous system, artificial neurons form the backbone of ANN models. These models are powerful tools for solving a wide range of problems, from image and speech recognition to predicting stock market trends.

By the end of this lesson, you will not only understand the theoretical foundations of MLPs but also be equipped to implement and train your own models using Python. Whether you're a beginner or looking to deepen your understanding, this guide will provide an overview of the basic structure of neural networks.

Neural networks

Neural networks are designed to recognise patterns and make decisions with minimal human intervention. This capability makes them many-sided, applicable in areas ranging from medical diagnosis to autonomous driving and even in financial forecasting. We will explore each component of these networks and see how they interact to process information. Understanding these interactions will provide you with the foundational knowledge needed to implement and train your own neural networks using Python and popular deep learning frameworks. Now, let's begin by discussing the biological inspiration behind the design of neural networks and how this inspiration has shaped the development of artificial intelligence.

Biological inspiration

At the heart of our nervous system are neurons, biological cells capable of processing and transmitting information chemically and electrically within the brain. These neurons are interconnected by synapses, which act as sites of signal transmission, influencing the neuron's potential to fire a subsequent signal depending on the strength and timing of the incoming signals. The goal of creating artificial neural networks was to mimic these biological processes. The artificial neuron, also known as a perceptron, functions similarly to its biological counterpart. It receives input signals, processes them, and produces an output, which is determined by an activation function. This function is analogous to the threshold potential in biological neurons that must be exceeded for a neuron to fire.

The structural organisation of the brain, with its network of neurons and synapses, enables complex behaviours and decision-making processes. ANNs attempt to

replicate this complexity by layering multiple perceptrons in various configurations, allowing the system to learn and make sophisticated decisions. For instance, just as our brain can recognise faces by integrating information from billions of neurons, ANNs can be trained to identify patterns and perform tasks like image and speech recognition. The idea of training a network also parallels biological learning mechanisms. In our brains, learning involves adjusting the strengths of synaptic connections in response to new experiences. Similarly, training an ANN involves adjusting the weights and biases of connections between artificial neurons based on data inputs. This process improves the network's performance over time, improving its accuracy in tasks such as classification and prediction.

Through this biomimicry, neural networks not only become capable of performing complex tasks but also provide insights into the potential for artificial systems to simulate human cognitive functions. This biological inspiration continues to guide novel approaches in neural network architectures and learning algorithms, pushing the boundaries of what artificial intelligence can achieve.

Artificial neuron and activation functions

An artificial neuron, the main actor of the functionality of ANNs, represents a mathematical model inspired by the biological neurons. Each neuron in a neural network processes incoming data, applies a specific function, and passes on an output to other neurons. The inputs to an artificial neuron are typically numerical values, which represent data from the external environment or outputs from other neurons. These inputs are multiplied by weights (commonly annotated with w), numerical factors that signify the importance or strength of the inputs in the neuron's decision-making process.

The products of these inputs and weights are summed together, and a bias (b in Fig. 1), a unique parameter associated with each neuron, is added. The bias allows the neuron to shift the activation function curve up or down, which is used for fine-tuning the neuron's output. The sum of the weighted inputs and the bias forms the total input (U) to the neuron's activation function.

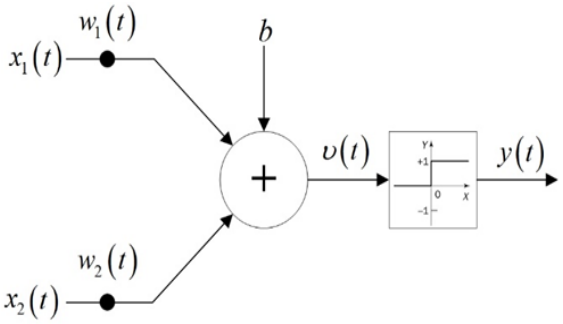

***Figure 1**: Example of a perceptron with two inputs (x_1 and x_2) and one output (y)*

An activation function represents a vital component because it defines the output value of a neuron based on the summed input. This function can be linear, but more commonly, it is non-linear to allow the network to capture complex patterns and behaviours in the data.

A common example of activation functions is the sigmoid function (Fig. 2a), which compresses outputs into a range between 0 and 1, making it useful for probabilities and models where binary outcomes are predicted. In medical diagnosis, for example, sigmoid functions help assess whether a tumour is benign or malignant. Imagine a dataset with variables like age, height, weight, and tumour size from a medical study – the sigmoid function would convert these inputs into a probability score. If the score exceeds 0.5, the tumour is classified as malignant; otherwise, it is benign.

Another popular choice is the hyperbolic tangent (tanh) function (Fig. 2b), which outputs values between -1 and 1, and it is efficient for cases where the model needs to handle negative values explicitly (such as in stock market trend prediction).

One more widely exploited function is the Rectified Linear Unit (ReLU) (Fig. 2c) which outputs the input directly if it is positive, otherwise, it will output zero, which introduces non-linearity while keeping the computation simple and efficient. It is often preferred for deep networks, especially in applications like image recognition, due to its computational efficiency and ability to mitigate the vanishing gradient problem.

We will end our exploration of activation functions by introducing the Softmax function (Fig. 2d) which is often used in multi-class classification problems where the network needs to assign probabilities to different classes, ensuring that the sum of the probabilities is 1. For instance, if we are building a neural network to classify images of handwritten digits (0-9), then in the final layer, we apply the softmax activation function to get the probability distribution over the 10 classes.

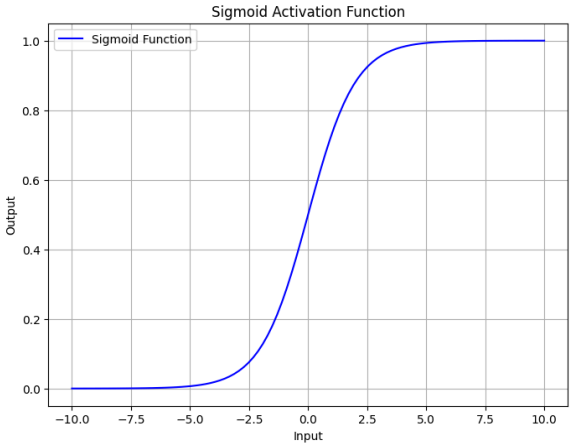

***Figure 2a**: Sigmoid function*

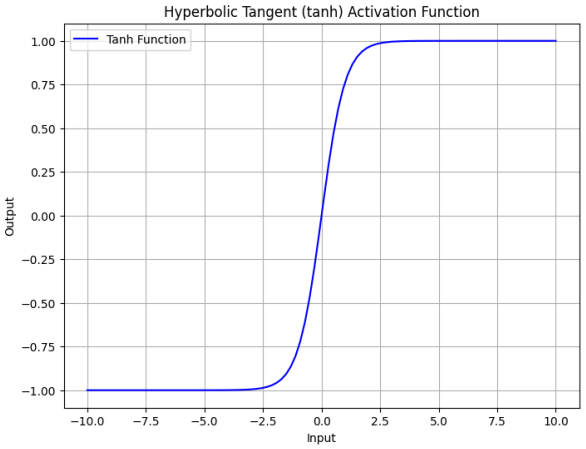

***Figure 2b**: Hyperbolic tangent function*

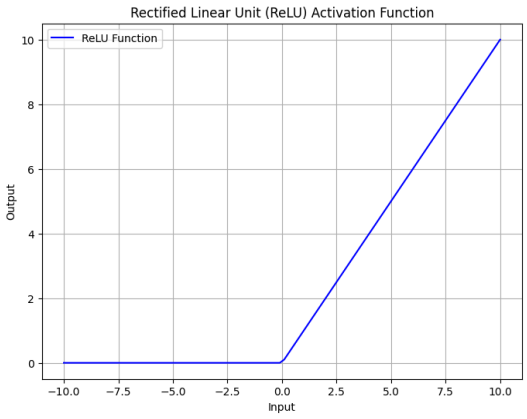

***Figure 2c**: Rectified Linear Unit*

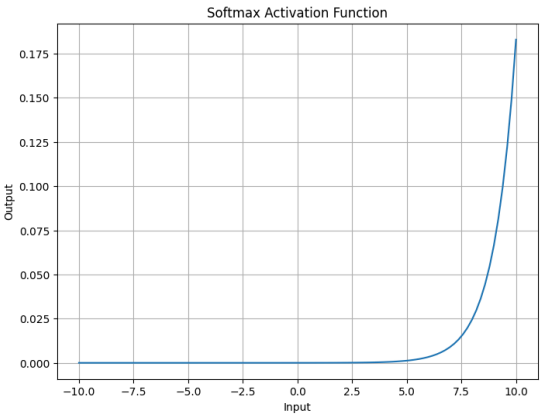

***Figure 2d**: Softmax function*

Using what you have learnt, you can design a neural network by combining multiple neurons, each with its own weights, biases, and activation functions. In the upcoming section, we will explore the architecture and components of a MLP, including the roles of the input layer, hidden layers, and output layer.

Multi-layer perceptron

An MLP represents a form of artificial neural network that consists of multiple layers of neurons, each designed to process the input data hierarchically. Hierarchical processing is when input data is broken into smaller and more abstract representations. This allows the model to learn complex patterns. MLPs are one of the most common network architectures used for classification and regression tasks.

Architecture and components

The architecture of an MLP consists of an input layer, one or more hidden layers, and an output layer, each containing a set of neurons (Fig. 3). The input layer receives the initial data which is then processed through successive hidden layers. These hidden layers are the core computational engines of the MLP, where each layer transforms its input into a slightly more abstract and composite representation. Neurons within these layers are fully connected, meaning each neuron in one layer connects to every neuron in the next layer, facilitating the flow and transformation of information through the network.

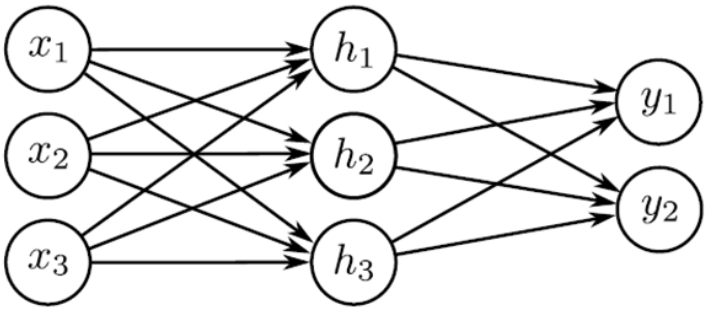

***Figure 3**. Visual representation of neural network (Prince, 2024)*

Technically, any neural network that includes at least one hidden layer is known as an MLP. Networks featuring just one hidden layer are commonly termed shallow neural networks. In contrast, those with multiple hidden layers are called deep neural networks. All the networks discussed in this lesson are examples of feed-forward networks. In feed-forward networks, information flows in one direction only, from input to output. There are no loops or circles in these networks, meaning the connections don't feed back into themselves.

Input layer, hidden layers, and output layer

An MLP is structured into three primary types of layers: the input layer, hidden layers, and the output layer.

The input layer serves as the initial point of data entry, where each neuron corresponds to a feature in the input data. This layer does not perform any calculations; it merely passes the data to the next layer.

Hidden layers, which are sandwiched between the input and output layers, are the core computational engines of the MLP. Each hidden layer consists of neurons that process inputs from the previous layer through weighted sums followed by an activation function, allowing the network to learn complex patterns and relationships.

The output layer, at the end of the network, consolidates the learnt features from the hidden layers into a format suitable for making predictions or decisions, with each neuron typically representing a possible output category or a continuous value, depending on the task. A network example based on a single hidden layer (a shallow neural network) is given in Fig. 4.

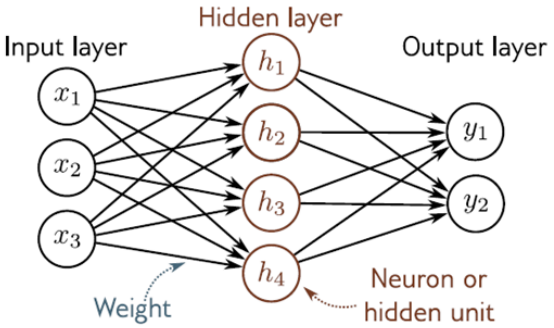

Figure 4. Shallow neural network (Prince, 2024)

Weights and biases

The process of learning in an MLP involves adjusting these weights and biases based on the error of the output compared to the expected result. For the initialisation of a neural network, the weights are usually randomised, and the biases are set to small random values close to or equal to zero. Weights are the parameters that adjust the strength of the connection between neurons across different layers. Each weight multiplies the input it receives, which effectively scales the input's influence on the network's output. The cumulative input to a neuron, after being modified by these weights, is then offset by a bias term.

Biases are added to the input sum before it is passed through a neuron's activation function. The addition of a bias value allows the activation function to shift left or right, which can be important for fine-tuning the output of the neuron and allowing the model to better fit the data. To understand how bias shifts the activation function, consider the activation function as a threshold that needs to be surpassed for the neuron to activate. For example, if we have a simple linear activation function $f(x) = x$, adding a bias shifts this function along the x-axis. Imagine a scenario where you have two inputs, x_1 and x_2, and a bias b. The neuron calculates the weighted sum of these inputs as $y = \omega_1 x_1 + \omega_2 x_2 + b$. Here, ω_1 and ω_2 are the weights. If the bias b is positive, it means that the neuron can activate even if the weighted sum of x_1 and x_2 is small. Conversely, if b is negative, it requires a larger weighted sum of inputs for the neuron to activate. This ability to shift the activation threshold helps the neural network better fit the data by adjusting how inputs are combined and processed.

Let's see an example here where the task will be to predict whether a student will pass or fail based on hours studied (x_1) and hours slept (x_2). Initially, we randomise the

weights and set the bias to zero. Assume the initial weights are $\omega_1 = 0.5$ and $\omega_2 = 0.3$. Without a bias, the neuron's output is calculated as:

$$y = 0.5 \times hours\ studied + 0.3 \times hours\ slept$$

Suppose a student studies for 4 hours and sleeps for 6 hours. The output will be:

$$y = 0.5 \times 4 + 0.3 \times 6 = 2 + 1.8 = 3.8$$

Now, let's add a bias of $b = 1$. The new output is:

$$y = 0.5 \times 4 + 0.3 \times 6 + 1 = 2 + 1.8 + 1 = 4.8$$

Here, the bias shifts the activation function to the right, allowing the neuron to output a higher value even if the inputs are the same. This shift can help the network make better predictions by fine-tuning the output.

Spot check 1

Let's see what you can remember from the first two sections.

1. Highlight the key components of an artificial neuron.
2. What are MLPs, and what are their core parts?

Feed-forward neural networks

Feed-forward neural networks are a type of neural network where connections between the nodes do not form cycles. This structure is analogous to a one-way street where information moves in only one direction: forward from the input layer, through the hidden layers, to the output layer without any loops or cycles (Fig. 4). This straightforward architecture ensures that the computational model is easy to understand and implement because it processes data in a clear and predictable sequence. The lack of cycles in the network makes feed-forward neural networks applicable for tasks that involve direct mappings from input to output, such as classification and regression problems. They are simpler compared to networks with cycles such as recurrent neural networks, which makes them easier to analyse and train.

Forward propagation

Forward propagation is the method by which values are transmitted through a feed-forward neural network. Starting at the input layer, the process involves taking the initial inputs, applying weights to them, and passing the resultant values through a chosen activation function to the next layer. This process is repeated layer by layer until the output layer is reached. Each neuron in a layer takes the outputs from the previous layer, multiplies those outputs by the weights associated with the connections, and adds a bias term. This sum is then passed through an activation function, which determines the neuron's output based on its input.

The ultimate goal of forward propagation is to compute the final output of the network, which represents the network's prediction or decision based on the given input. This output is then used to compute the error of the prediction by comparing it against the actual target values in the training data. The error value derived from this comparison is considered important as it feeds into the backpropagation process, where the network learns by adjusting the weights and biases to minimise the error. We will learn more about backpropagation later. Forward propagation is efficient and straightforward, allowing the network to quickly evaluate new inputs once training is complete.

Activations and outputs

Calculating activations and outputs in a neural network during forward propagation involves a step-by-step transformation of inputs into outputs through the network's layers. An activation refers to the output of a neuron after applying an activation function to the weighted sum of inputs plus a bias. Each neuron in a layer computes its activation by first aggregating the weighted sum of its inputs. This weighted sum includes each input to the neuron multiplied by the corresponding weight for that connection, plus a bias term that shifts the activation function to fit the neural network model better, whether it is for classification or regression tasks.

$$z_j^{(l)} = \sum_i \omega_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)},$$

Where:

$z_j^{(l)}$ – is the weighted sum for the j-th neuron in the l-th layer.

$\omega_{ji}^{(l)}$ – represents the weight from the i-th neuron in the $(l-1)$-th layer to the j-th neuron in the l-th layer.

$a_i^{(l-1)}$ – activation from the i-th neuron in the $(l-1)$-th layer

$b_j^{(l)}$ – bias associated with the j-th neuron in the l-th layer.

Once the weighted sum is calculated, it is passed through an activation function:

$$a_j^{(l)} = \sigma\left(z_j^{(l)}\right),$$

Where:

$a_j^{(l)}$ – activation of the j-th neuron in the l-th layer.

σ – activation function applied to the weighted sum $z_j^{(l)}$

As already stated, common choices for σ could include:

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$

Hyperbolic Tangent (tanh): $\sigma(x) = tanh\,(x)$

Rectified Linear Unit (ReLU): $\sigma(x) = (0, x)$

Next, the output of the activation function becomes the output of that neuron, serving as an input to the next layer in the network. In the final layer, the activation function often differs depending on the specific task the network is designed to perform. For instance, a softmax activation function might be used in the output layer of a classifier to represent the probabilities of the input being one of the several classes.

$$a_k^{(L)} = softmax\left(z_k^{(L)}\right) = \frac{e^{z_k^{(L)}}}{\sum_{i=1}^{K} e^{z_i^{(L)}}},$$

Where:

$a_k^{(L)}$ – activation (interpreted as the probability) of the k-th class in the output layer.

$z_k^{(L)}$ – weighted sum for the k-th output neuron.

K – total number of output classes.

Thus, the described process will guide the flow of data through a neural network, transforming inputs into outputs by sequentially applying linear and non-linear transformations. Each layer's output becomes the input for the next layer, culminating in the final output that can be used for making predictions or decisions based on the learned features and patterns in the data. Calculating activations and outputs involves a

sequence of linear (weighted sums and biases) and non-linear operations (activation functions) that process the input data into a format suitable for making predictions or decisions, effectively mapping inputs to desired outputs based on learned patterns.

Example

This example will illustrate the basic mechanism of how neural networks process input data to produce outputs. Let's have a look at how we would calculate the weighted sum that would be passed to the activation function for one row of data with five features or inputs. Recall that the weights are typically randomised for the initialisation of an MLP. The range used for the initial weights depends on the activation function but the values are typically small. In this example, we will implement four neurons in a processing layer and the ReLu activation function.

```
# Feed-forward calculation
import numpy as np

# Set the seed for reproducibility
np.random.seed(42)

# Five inputs
inputs = [0.5, -1.2, 3.3, -0.7, 2.5]

# 4 neurons in a processing layer: results with a 4x5 weights array (5
weights associated to each processing layer neuron)
weights = np.random.randn(4, 5)

# Each processing layer neuron has a bias. We will manually select their
values now. You can also use the 'randn' function if you want.
biases = [0.1, -0.2, 0.3, -0.1]

# Next, we will create an empty list to store the results from each neuron in
the processing layer.
layer_outputs = []

''' For each processing neuron, we will use the zip function to pair
initialised weights and biases. This function combines the weights and biases
lists element-wise, so 'neuron_weights' takes each row of the 'weights'
array, and 'neuron_bias' takes each element of the 'biases' list.'''

for neuron_weights, neuron_bias in zip(weights, biases):
# Set output to 0
    neuron_output = 0

    # For each input/weight pair
```

```
    # We use the zip function again to pair inputs and weights.
    for n_input, weight in zip(inputs, neuron_weights):

        # multiply each input by its corresponding weight and accumulate
        # the result in the output
        neuron_output += n_input * weight

    # Add the bias to the result
    neuron_output += neuron_bias

    # Append the neuron result to the layer output
    layer_outputs.append(neuron_output)
 # print the results
print(layer_outputs)
```

Output:

```
[1.0001420773493654, 2.005442928240517, -1.5473560559761335,
-1.0238726231281556]
```

Once the outputs are generated, they would then be passed through the ReLU activation function.

```
# Defining the ReLU activation function
def relu(x):
    return max(0, x)

# Applying the function to each output
relu_outputs = [relu(output) for output in layer_outputs]

# Print ReLU outputs
print("ReLU outputs:", relu_outputs)
```

Output:

```
[1.0001420773493654, 2.005442928240517, 0, 0]
```

Spot check 2

Let's see what you can remember from the last two sections.

1. What are feed-forward neural networks, and why are they advantageous for certain tasks?
2. What is forward propagation, and how does it contribute to the functioning of a neural network?

Instructions

In this task, you will implement an MLP to evaluate the quality of a basketball player based on five input ratings: speed, jump, shooting, intelligence, and strength. All scores are integers in the range from 1 to 100, and the quality of the player that the MLP should generate is a floating-point number with one decimal place, in the range between 0 and 10.

Practical task

- Create a Jupyter notebook called **mlp_task.ipynb**.
- Define the MLP architecture with specific sizes for the input and output layers. You are free to select the number of units for the hidden layer. **Hint:** start with a simple architecture.
- Initialise random weights and biases for the connections between layers.
- Use the sigmoid activation function for both the hidden and output layers.
- Perform forward propagation of inputs through the network by computing the weighted sums of inputs for each neuron in both the hidden and output layers, and then apply the sigmoid activation function to these sums during forward propagation through the network.
- After implementing the MLP, validate its functionality by passing sample input data through the network and printing the resulting outputs.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Spot check 1 answers

1. **Highlight the key components of artificial neurons.**

   The key components of artificial neurons include inputs, weights, bias, and activation functions. Inputs represent the data received by the neuron, weights signify the importance of each input, bias adjusts the output of the neuron, and the activation function determines the neuron's output based on the total input.

2. **What are MLPs, and what are their core parts?**

   MLPs are a type of artificial neural network architecture composed of multiple layers of neurons. The core parts of an MLP include an input layer, one or more hidden layers, and an output layer. The input layer receives the initial data, hidden layers process and transform the data through weighted sums and activation functions, and the output layer produces the final predictions or decisions based on the learned features from the hidden layers.

Spot check 2 answers

1. **What are feed-forward neural networks, and why are they advantageous for certain tasks?**

   Feed-forward neural networks are a type of neural network architecture which facilitates a one-way flow of information from the input layer through the hidden layers to the output layer. This structure is beneficial for tasks involving direct mappings from input to output, such as classification and regression.

2. **What is forward propagation, and how does it contribute to the functioning of a neural network?**

   Forward propagation is the process by which values are transmitted through a neural network from the input layer to the output layer. It involves computing the activations of neurons in each layer by applying weights to inputs, adding biases, and passing the results through activation functions. Forward propagation is crucial for generating predictions or decisions based on input data and serves as the foundation for subsequent steps in training and optimisation.

Reference list

Haykin, S. (2004). *Feedforward Neural Networks: An Introduction*. Wiley.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2), 251-257.

Nielsen, M. (2019). Neural Networks and Deep Learning. Retrieved from **http://neuralnetworksanddeeplearning.com/**

Prince, S. J. D. (2023). *Understanding Deep Learning*. The MIT Press. **http://udlbook.com**