Linear algebra and calculus: The maths behind artificial learning.

Kaelen Turner

1. Introduction

How did what was nothing more than a concept, 70 years ago, become arguably the most important technology of the present day? Artificial intelligence (AI), the cornerstone of modern technological advancement, is already redefining all aspects of society, showing no signs of stopping. AI is defined as the application of a computer system to a task that would be expected to require human intelligence. Its increasing prevalence makes it more and more important for us to understand how it is able to carry out such remarkable functions. I mean seriously, it's amazing that as a species we've created machines that can process and understand our natural language. We've made chess engines that can easily topple the world's best players, and we've made self-driving cars - which will likely, if they haven't already, be safer than any human driver. At the root of its brilliance lies mathematics. This essay sets out to explore how linear algebra alongside calculus as branches of mathematics have allowed humanity to simulate their own intelligence within the hardware of computers and machines. To begin with, we will form a necessary understanding of the basics of linear algebra before moving on to applying it to neural networks, gradient descent and optimisation, and many algorithms crucial for machine learning.

2. Fundamentals of linear algebra

Before looking at the more complex linear algebra used in AI, it is important to first establish a solid understanding of the basics on which it is built. Vectors (which lie at the core of linear algebra) are ordered lists of numbers. In the context of AI, they typically represent data. However, for intuition, vectors can be visualised geometrically as points/directions in space, with each number in the vector corresponding to a coordinate along an axis.
Examples of vectors include: $\begin{bmatrix}\pi \\ \sqrt{2}\end{bmatrix} \begin{bmatrix}2.433 \\ 9.01\end{bmatrix} \begin{bmatrix}-8 \\ 2 \\ -4\end{bmatrix} \begin{bmatrix}-5 & 3 & 1\end{bmatrix}$

Vectors can be written as column or row vectors, but in linear algebra they are usually treated as column vectors by default. As structures, vectors are 1D arrays; however when interpreted, they have a dimensionality equivalent to the number of elements in the array. A matrix is a 2D array, made up of multiple vectors, either as rows or columns. Examples of matrices include: $\begin{bmatrix}\pi & 4 & -1 \\ \sqrt{5} & 17 & 1\end{bmatrix} \begin{bmatrix}1 & 2 \\ 3 & 4 \\ 5 & 6\end{bmatrix} \begin{bmatrix}1 & 1 & 2 \\ 3 & 5 & 8 \\ 13 & 21 & 34\end{bmatrix}$

A tensor is a general term that includes scalars, vectors, matrices and structures of higher dimensionality.
Vectors and matrices can be combined and manipulated with various operations, the most foundational being addition and multiplication.
Matrix addition works simply by adding corresponding components:

$$\begin{bmatrix}2 & 4\\6 & 8\end{bmatrix} + \begin{bmatrix}1 & -3\\-5 & 7\end{bmatrix} = \begin{bmatrix}3 & 1\\1 & 15\end{bmatrix}$$

To add matrices, they must be of the same size and shape.
Scalar multiplication of a vector/matrix simply works by applying the scalar multiple to each element in the vector/matrix.

$$3\begin{bmatrix}1\\9\\8\\9\end{bmatrix} = \begin{bmatrix}3\\27\\24\\27\end{bmatrix} \qquad t\begin{bmatrix}a & b\\c & d\end{bmatrix} = \begin{bmatrix}at & bt\\ct & dt\end{bmatrix}$$

Matrix multiplication works by taking the dot product of rows of the first matrix and columns of the second. If matrix A is of size m × n and matrix B is of size n × p, then their product AB is a new matrix of size m × p:

$$\begin{bmatrix}a_1 & a_2\\a_3 & a_4\\a_5 & a_6\end{bmatrix} \times \begin{bmatrix}b_1 & b_2 & b_3\\b_4 & b_5 & b_6\end{bmatrix} = \begin{bmatrix}a_1b_1 + a_2b_4 & a_1b_2 + a_2b_5 & a_1b_3 + a_2b_6\\a_3b_1 + a_4b_4 & a_3b_2 + a_4b_5 & a_3b_3 + a_4b_6\\a_5b_1 + a_6b_4 & a_5b_2 + a_6b_5 & a_5b_3 + a_6b_6\end{bmatrix}$$

$$\begin{bmatrix}1 & 2\\3 & 4\\5 & 6\end{bmatrix} \times \begin{bmatrix}7 & 8 & 9\\10 & 11 & 12\end{bmatrix} = \begin{bmatrix}27 & 30 & 33\\61 & 68 & 75\\95 & 106 & 117\end{bmatrix}$$

Matrix multiplication only works when the number of columns in the first matrix equals the number of rows in the second.
In this way matrix multiplication can be used to transform data inputs.

3. Differentiation

Differentiation is a process used to find the gradient function of a function, which if the function is denoted as f(x), will be denoted as f'(x) or $\frac{dy}{dx}$. And if an x value is substituted into that function the output will be the gradient of the tangent to that point/the gradient of the curve at that point.$\frac{\partial f}{\partial x}$ is used as notation for the derivation of a function or in the case of multiple variables.

$[\frac{d}{dx}(x^n) = n \cdot x^{n-1}]$ This is the rule for differentiating each term.

E.g. $If\ f(x) = 3x^4 - 5x^3 + 2x^2]\ then\ [f'(x) = 12x^3 - 15x^2 + 4x]$

4. Neural Networks

Neural networks are a type of machine learning model inspired by the neurons within the human brain. They are mainly used to recognise patterns, classify data, make predictions and give recommendations.

A neural network is made up of layers of interconnected neurons (nodes).

There are three classes of layers:

Input layer: This receives the data - could be pixels in an image, audio waveform values,

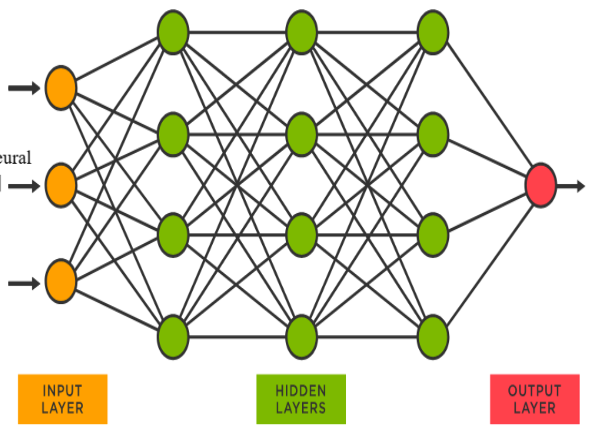


Figure 1: Example neural network [1]

specifications of a car, etc. Data is received in the form of numerical tensors.
Hidden layer: Information is processed through weights and activations.
Output layer: Provides an output (usually a prediction or a classification or just a decision on how to proceed).

Each neuron takes multiple inputs via a vector, with every input having an associated weight which effectively tells the neuron how important this input is in influencing its decision; a bias is also used to influence the neuron's output by shifting the result up or down. The bias is the value the neuron would produce if all inputs were 0 and can be thought of as the neuron's inclination towards a specific output that allows the activation threshold to shift.
If the inputs given are $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ with associated weights $\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ then the neuron will calculate the dot product of the two vectors (which determines the weighted sum for that neuron) and then add the bias: $z = (x_1 \times w_1) + (x_2 \times w_2) + b$ or in matrix form Z = WX + b, where W is the weight matrix, X is the input matrix and Z is the output matrix, which is fed as an input into the next layer. This value z is then plugged into an activation function, a common and simply intuitive example being the sigmoid function (Figure 2). Which, if we have z along the x-axis, will look like: $\sigma(z) = \frac{1}{1+e^{-z}}$ and will squish z down to a number between 0 and 1, making z a possibility - specifically the possibility of that node being activated. A node is activated when its output - after the activation function - passes a certain threshold. And when applied to Z as a matrix, it will be individually applied to each element in the matrix, creating an output matrix. The output, whether a scalar value or a matrix, will then be fed into the next layer. The values of z or Z held by all the neurons in the output layer will determine how the neural network chooses to respond/its answer, prediction or classification.

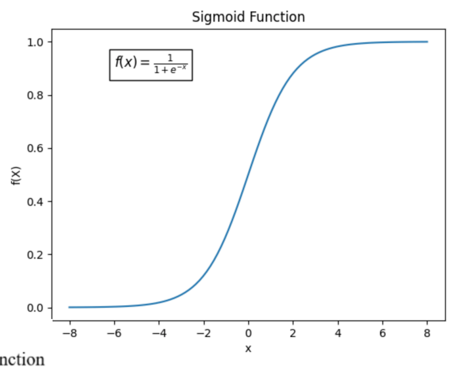


Figure 2: Sigmoid function [2]

4. Training of neural networks

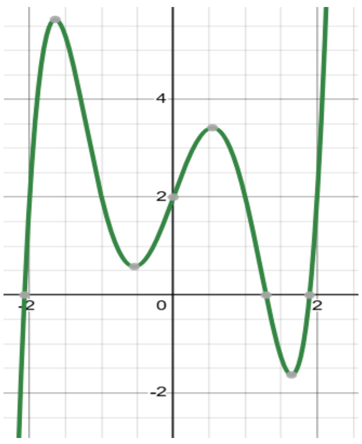
Neural networks don’t start off with much of a clue as to what they are doing; they need to learn, and so programmers teach or train them. With the goal of adjusting the weights and biases of each neuron so that the output of the network is correct or accurate for all test cases. For example, a neural network may be used to predict car prices based on an input vector containing make/model, miles, manufacturing year, accessories and add-ons, etc. The goal would then be to ensure that when those inputs based on cars with known prices are given, the prices outputted by the neural network are equal to or very close to the known price.
At the start of the training process the neural network uses pseudo-random weights and biases, as naturally at the start the network has no idea how to work with the input data to achieve valid outputs. Using this random data and a test input, the neural network performs a

forward pass, producing an output with its randomised weights and bias. The network then uses a loss function, for example, the mean square error function, to calculate how off the true value it was on average.

L = Mean square error = $\frac{1}{n}\sum_{i=1}^{n}(z_i - \hat{z}_i)^2$ where z is the true known value and $\hat{z}$ is the value produced by the network. The squaring ensures that errors above and below the known value don't cancel each other out while also making sure the larger errors are significantly more penalised than the smaller errors.

4.1. Gradient descent

Figure 3: Quintic graph example [3]



For the next step, imagine substituting in different values of w (weight) into the loss function, and then plotting the resulting graph of L(w). Using figure 3 as a simplified example, we would want to find a minimum value for L(w), as this is one of the points where the mean square error is lowest. To do this, we would first differentiate the curve, giving us an expression for L'(w). Then we would sub in a pseudo-random value of w into the differentiated expression of L'(w); if the gradient was negative, you would increase the value of w. If it was positive, you would decrease the value of w, and so you would follow the curve downwards either way, this is the same as moving in the direction of the negative of the curve's gradient. This process would be repeated, ensuring the increase or decrease in w is proportional to the magnitude of the gradient, as this ensures, if you cross the minimum, you don't go too far in that direction and that you can actually hit the exact minimum. 3Blue1Brown[4] offers a brilliant way to see this - visualising it as a ball rolling down the curve, which lets us picture how we are always trying to go downwards (following the negative gradient as a ball would due to gravity) and this downwards movement being proportional to the steepness of the slope as a ball would in real life (a shallower slope means a high proportion of gravity acting into the slope than down the slope) as well as it showing how it is possible to overshoot it by a little - a ball won't lose its momentum as soon as it starts moving uphill.

Mathematically let's say the curve had equation L(w) = $w^5 - 5w^3 + 4w + 2$

Then the differentiated expression would be L'(w) = $5w^4 - 15w^2 + 4$

If we start randomly at the point (0, 2) - I chose a point with whole number coordinates to make it easier - subbing that into L'(w) a gradient of +4, as its positive and not very steep we subtract a small value from w, say 0.5 to give us a new coordinate of (-0.5, 0.59375) which we then plug in again giving us a gradient of +0.5625, this is still positive but much smaller than 4 so we would decrease the w value by a smaller increment say 0.1 to give us a new coordinate (-0.6, 0.60224) which plugged into L'(w) gives us a gradient of -0.752, we have now overstepped the minimum since the gradient has changed in sign and so we can deduce the minimum is -0.6 < w < -0.5, we would then need to add to w by a very small amount and repeat this process until we reached a gradient of 0 and therefore the minimum. This in turn gives us the value of w that results in the highest accuracy for our neural network - it's important to note this process is done by the neural network; we just have to feed it the test data.

However, in the case of neural networks, usually there are multiple weights and biases (they can have thousands, millions, or even billions of weights and biases in a neural network), and so this process would need to be completed in higher dimensions. Thankfully linear algebra comes into play and makes this relatively simple, at least computationally. By forming a vector with all of the pseudo-random weights, we can consider each row in turn, meaning we will be taking one value of w at a time and repeating the above process for each row/value of w to form a second vector made up of all the negative gradients acquired. By then adding this second vector to the first vector, we would move towards a minimum in the higher dimension. We can then repeat this process until we reach the minimum.

Take this example where the network has four weights to compute (again, in reality there would be a lot more, but four allows me to explain how the process works for multiple weights without it being too complicated). You would calculate the negative gradient of each of the four weights, forming a new vector with these values, then you would feed that vector through an algorithm (η in the below example) that determines an appropriate increment size for each of the four weights to be increased or decreased by. These values would in turn be put into a vector (increment vector), which would then be added to the vector with the initial weights in, moving the weight value towards a minimum in this four-dimensional graph. This process would be repeated until a minimum for all rows of the vector is reached.

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

Weights vector - separates the weights so they can be handled individually.

$$-\nabla L(w) = \begin{bmatrix} -\frac{\partial L}{\partial w_1} \\ -\frac{\partial L}{\partial w_2} \\ -\frac{\partial L}{\partial w_3} \\ -\frac{\partial L}{\partial w_4} \end{bmatrix}$$

Negative gradients of weights on their respective L(w) graphs in vector form - as seen, the process for gradient descent is applied to all weights individually.

$$-\nabla L(w) = \begin{bmatrix} -\eta\frac{\partial L}{\partial w_1} \\ -\eta\frac{\partial L}{\partial w_2} \\ -\eta\frac{\partial L}{\partial w_3} \\ -\eta\frac{\partial L}{\partial w_4} \end{bmatrix}$$

Increment vector - Each value by which each associated w needs to change is stored in a vector, meaning that each of the weights can be changed by their corresponding increments all at once.

And so a neural network given plenty of test data is able to establish the weights and biases (biases are determined by the same process of gradient descent) that provide it with the most accurate results, and so it is able to "learn" whatever we want it to just using a mix of linear algebra and calculus.

5. Conclusion

As seen, neural networks are an amazing application of maths allowing a computer to learn in the way we want it to and to actually simulate human intelligence. By understanding the basics on a lower scale and how those basics are scaled up we can actually comprehend how behind one of the most pivotal and impossible technologies of today there simply lies maths. This is the maths behind artificial learning and the basis of voice and facial recognition, medical diagnostics, popular and incredibly useful chatbots such as GPT and Deepseek, and the chatbots found on many company sites. And that's just the beginning of their modern day applications, who knows what these maths powered learning models will be able to pull of in the future.

6. References

1. Spotfire | Neural Networks Explained: A Look at Functions Applications
2. AI | Neural Networks | Sigmoid Activation Function | Codecademy
3. Desmos | Graphing Calculator used for quintic graph example
4. Gradient descent, how neural networks learn | DL2 - intuition behind gradient descent
5. Equation Editor for online mathematics - create, integrate and download - used for all vectors, matrices and equations visual representations.