Can machines learn?

Rowan Clarke

2020

Abstract

This is an abstract.

Contents

1 Introduction

intro

2 Literature Review

2.1 Inspiration from Human Learning

Our brains process information through neurons, which combine many electrical inputs, called dendrites, into one output, the axon. These neurons are connected via synapses, which chemically transmit information via neurotransmitters. We learn by changing the amount of neurotransmitter released.

By connecting multiple neurons together, we allow for more complex outputs. Our cerebellum - the part of the brain that controls language, motor function, and cognitive learning - consists of roughly 10^{10} neurons, each connected to approximately 10000 other neurons.

2.2 Hebbian Theory

To digitalise this biological structure of the brain, we need a mathematical model. The Hebbian Theory attempts to model this behaviour.

The Hebbian Theory suggests that if two neurons in space activate in phase - at the same time - the connection should strengthen between them. If in opposite phase, the connection should become weaker. No correlation between the activations should not affect the connection.

Hebbian learning tries to replicate learning in animals such that

$$w_{ij} = x_i x_j \tag{1}$$

where w_{ij} is the weighted synapse between neuron i and neuron j, and x_i is the value at neuron i, at a given time in space. Linking this with the biological neuron, x_i is the axon of i - as with j - and w_{ij} is the strength of the synapse connecting the axon of i to the dendrite of j.
We can rectify Equation 1 to

$$\Delta w_{ij} = x_i x_j \tag{2}$$

such that w_{ij} changes based on its current state.

2.3 Artificial Neural Networks

Unfortunately, we cannot directly use the Hebbian Theory for machine learning where we want the machine to learn a desired output for a specified input. This is because it doesn't know what the output is supposed to give, but rather how the input data is related - this type of learning is called unsupervised learning, which we will cover in Types of Learning.

2.3.1 Neuron Model

Consider a neuron y. There are many neurons connected to y, $x_1, x_2, \cdots, x_I$. The axon from x_i, $\mathrm{A}(x_i)$, is connected to the i^{th} dendrite of y, $\mathrm{D}_i(y)$, via a weighted synapse w_i. Therefore $\mathrm{D}_i(y) = w_i\mathrm{A}(x_i)$. The dendrites of y are combined to give the axon of y, $\mathrm{A}(y)$.

The axon of y can be simply modelled as

$$\mathrm{A}(y) = \sum_{i=1}^{I} \mathrm{D}_i(y) \tag{3}$$

or simply

$$\mathrm{A}(y) = \sum_{i=1}^{I} w_i \mathrm{A}(x_i) \tag{4}$$

Since we are dealing with just the axons of x_i and y, we can simplify Equation 4 as

$$y = \sum_{i=1}^{I} w_i x_i \tag{5}$$

2.3.2 Network Architecture

x_i and y are neurons, which can be modeled as nodes

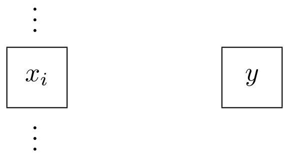

Figure 1: Model of x neurons and y neuron

These neurons are connected via synapses, which can be modeled as a line connecting the neurons.

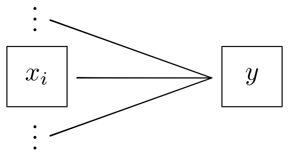

Figure 2: Model of x neurons connected to y neuron with synapses

We can add more neurons, y_j, from the same input neurons, x_i, using different synapses. This will increase the complexity of the neural network, allowing for more flexibility.

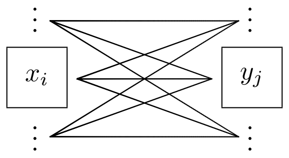

Figure 3: Model of x neurons to y neurons with synapses

In a neural network, there are input, hidden, and output layers, where a layer contains neurons that activate at the same time at a specific depth within the network.

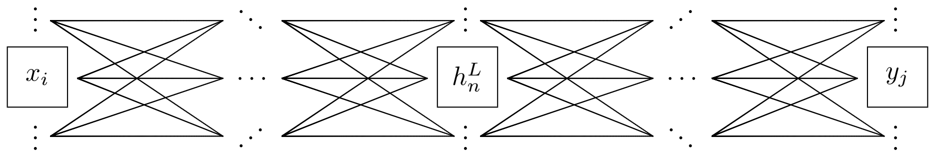

Figure 4: Model of x input neurons connected to h^L neurons connected to y output neurons

More layers allow for more complex functions, but if there are too many neurons, the neural network will overfit to the data, which essentially means it is memorising instead of learning. When it overfits to data, it is unlikely to produce similar results to input data that is slightly different. This is because when a neural network learns, it finds patterns in the data it is given, allowing it to be more flexible an give the correct output for a new given input.

2.4 Gradient Descent

So far we have discussed how information is propagated forward through the network to produce an output, however there is no way we can change the weights of each synapse to achieve a different output, and thus learn.

Gradient Descent attempts to minimise the cost, ∇C, of a neural network, which is a measure of how well it maps the target data. There are many ways of doing this, but for our simple feedforward architecture, the most common method is backpropagation.

2.4.1 Backpropagation

Backpropagation aims to change the weights of a neural network in the most efficient way such that the output of the neural network fits closer to the desired output, given a specified input.

Consider the following neural network.

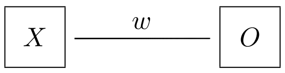

Figure 5: Model of neuron X connected to neuron O with a weighted synapse w

We can describe the how close the output, O, is to the target, Y, with the cost, C, which is equal to the square of the difference of O and Y. We square this value so that the cost is positive, and it is continuously differentiable, which will prove very useful later on.

$$C = (O - Y)^2$$

We can make a graph of how w effects C, so we can find what weight gives the minimum cost, and therefore the closest output to our target.

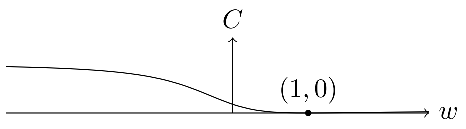

Figure 6: A graph of C against w

We know that where $w = 1$, we achieve a cost of 0, so the output is equal to the input. However, we do not want to iterate through all possible weights in a neural network to find the minimum cost, as that is very computationally expensive. Instead, we should start at a random w, say -1, and take small steps downhill, hence gradient descent.

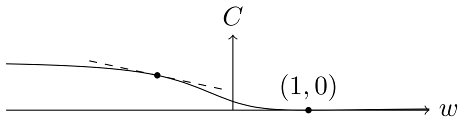

Figure 7: A graph of C against w showing the gradient at $w = -1$

We can decrease the cost, C, by repeatedly changing w proportionally to the negative gradient at w, or

$$\Delta w = -\alpha \frac{\partial C}{\partial w} \tag{6}$$

where α is the learning rate, which is essentially how big of a step in the given direction you take - too small, and it will take too long finding the minima - too large, and you will overshoot the minima, almost rocking back and forth.

Now, all we need to do is find the gradient.

2.4.2 Using the Chain Rule

Using Figure 5, we can express C as a series of functions

$$C = (O - Y)^2$$
$$O = \sigma(z)$$
$$z = wX$$

Using the chain rule, we can equate

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial O}\frac{\partial O}{\partial z}\frac{\partial z}{\partial w} = 2(O - Y)\sigma'(z)X \tag{7}$$

Equation 7 only works for Figure 5, so we need a more general model.

Since we are dealing with multiple neurons, weights, and layers, we need a syntax to index these:

x_i^L represents the ith neuron in layer L; Y_i^L represents the target for x_i^L; n_L represents the number of neurons in layer L; w_{ij}^L represents the weight of the synapse connected to the ith neuron in layer L, from the jth neuron in layer $L - 1$

Consider the following

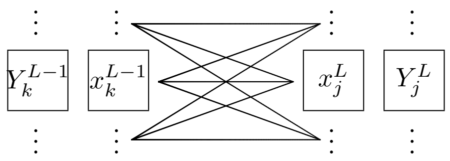

Figure 8:

The cost from multiple outputs is the sum of each cost of the output to the corresponding output

$$C = \sum_{j=0}^{n_L - 1} (x_j^L - Y_j)^2$$

The output remains similar to before

$$x_j^L = \sigma(z_j^L)$$

where

$$z_j^L = \sum_{k=0}^{n_{L-1}-1} w_{jk}^L x_k^{L-1}$$

We can reevaluate the gradient again using the chain rule

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{\partial C}{\partial x_j^L} \frac{\partial x_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L} = 2(x_j^L - Y_j^L)\sigma'(z_j^L)x_k^{L-1} \tag{8}$$

To find out what x_k^{L-1} should be, Y_k^{L-1}, we can use the chain rule against x_k^{L-1} instead of w_{jk}^L

$$Y_k^{L-1} = \frac{\partial C}{\partial x_k^{L-1}} = \frac{\partial C}{\partial x_j^L} \frac{\partial x_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial x_k^{L-1}} = 2(x_j^L - Y_j^L)\sigma'(z_j^L)w_{jk}^L \tag{9}$$

We can then repeat this for each layer going backwards, until all the weights have been rectified to decrease the cost in the most efficient way possible, ∇C. If we repeat this process for each $\{X_i, Y_i\}$ in training set T, we would have trained a function to fit to T.

3 Architectures

3.1 Learning Paradigms

Consider the different types of learning paradigms, supervised, unsupervised, and reinforcement learning.

3.1.1 Supervised Learning

Supervised learning is where the machine is tasked to fit target data. The training set T can be expressed as

$$T = \{X_1, Y_1\}, \{X_2, Y_2\}, \cdots, \{X_I, Y_I\}$$

where X_i is an input, Y_i is the desired outcome of X_i, and I is the length of the training set T. The machine propagates back through the neural network to adjust the synapses in order to fit closer to the target data.

3.1.2 Unsupervised Learning

Unsupervised learning is where the machine alters the synapses in the neural network based upon the inputs. This can be used to classify data or cluster data based on how alike the data is.

3.1.3 Reinforced Learning

Reinforcement learning is where the machine is given a score for the output results, instead of in supervised learning where the machine is given a grade. This typically doesn't have a training set, as it is generally used for control and movement - where the input is given from the situation.

3.2 The Extent of Machine Learning

3.2.1 Autoencoders

Autoencoders are unsupervised neural networks, as they do not require target data. They encode (compress) the input data into a latent vector, and then decode the latent vector into an output that should resemble the input.

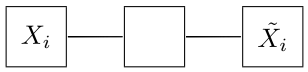

Figure 9: $\tilde{X}_i$ is the decoded output that resembles the input

If you replace X_i with a function of X_i, $f(X_i)$, you can train an inverse function as the autoencoder.

They train an inverse function f^{-1} from a given function f.

$$X = \{X_1, X_2, \cdots, X_I\}$$

There are some common functions that lead to useful outcomes. Where $f(x) = x$, the autoencoder acts like a compressor, making it an effective lossy compression technique to reduce broadband. Where $f(x)$ is a noise filter, the autoencoder acts as a denoiser, making it faster to render animations as you do not have to calculate every pixel.

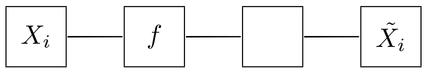

Figure 10:

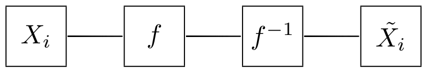

Figure 11:

3.2.2 Convolutional Neural Networks

A convolutional neural network uses convolutions, which are layers that process spatial data such that the neural network can find patterns in context of the spacial data easier. This means that the network requires less neurons, which reduces the risk of overfitting.

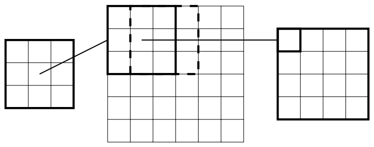

Figure 12: $\tilde{X}_i$ is the decoded output that resembles the input

3.2.3 Recurrent Neural Networks

3.3 The Limitations of Machine Learning

4 Conclusion