Mathematical Investigations 3: Machine Learning

Joe Barry, Will Cooke, Ciaran McMenamin, Elliot Moore

2023

Abstract

We will discuss the application of the methods of linear algebra and calculus to the context of machine learning. We will discuss methods for improving our models as well as the ways in which we can evaluate their proficiency. We will also look at modern applications of machine learning models and how they are changing the world we live in.

Contents

1 Introduction

Machine learning is an exciting and rapidly evolving subset of artificial intelligence that is changing the way we handle data across the globe. At its core, machine learning focuses on using models and algorithms in order to allow a machine to interpret data without being explicitly programmed. In this paper we will compare the differences between supervised and unsupervised machine learning, and how one can choose between them based on what type of model they need. We will also explore the methods of linear regression, logistic and gradient descent, and in particular how these can be used to create an efficent regression model. We go on to discuss in detail Neural Networks including both front and back propagation algorithms as well as weights and biases of the networks. We use our understanding of calculus and linear algebra, in particular The Chain Rule to justify these algorithms. This then leads us to consider the importance of the modern applications of machine learning, within the fields of medical diagnostics and image recognition, in particular, within CCTV and the effect it has had on crime. By the end of the essay, the reader will have a strong fundamental understanding of machine learning techniques as well as the vast possibilities and applications it may provide in the future.

2 Supervised and Unsupervised Machine Learning

2.1 Supervised Machine Learning

Supervised machine learning is characterised by its use of *labelled datasets*. In essence, this means the data is already labelled with the correct outputs for each input. To define labelled data, let's first discuss what we mean by data, and a dataset.

Definition 2.1.1. Data is simply information, and a **dataset** is a table consisting of rows (data points) and columns (features of the data point).

This may seem fairly abstract so as an example consider animals, where each row is a different animal (lion, elephant etc) and each column is a feature of the animal (wings, fur, claws etc). *Labels* are dependent on the context of the model and its purpose - for example if we are trying to predict what type of animal from an image (image recognition is discussed further in section 7), then the label is the name of the animal. If we are trying to predict the type of disease an animal has got based of symptoms, then the disease is the label. Therefore we can define labelled data as follows:

Definition 2.1.2. Labelled data is data that comes with a label. **Unlabelled data** is data that does not come with a label.

Therefore within supervised machine learning, we are only using labelled datasets and so these datasets are "supervising" the model to help it learn, classify data and predict outcomes. An advantage of supervised machine learning is that using labelled inputs and outputs makes it easy to measure the models accuracy, since we know the correct output for each input and so we know if the model is right. Supervised machine learning can be split up into two main types of model, *Classification* and *Regression*.

Definition 2.1.3. Classification models use algorithms that sort data into different categories, and predict a *state* (for example predicting the type of animal from an image).

Another example of a classification model is predictive text, where the model predicts what word you will type next in the sentence. Since the outputs of a classification model are chosen from a finite list of states, the outputs are called *discrete*.

Definition 2.1.4. Regression models are used to understand the relationship between dependent and independent variables, and output a *numerical value*.

For example a regression model would be one that predicts the weight of an animal in kilograms. Outputs of regression models are always numerical, and since the output can be any real value in a continuous interval, they are called *continuous*.

2.2 Unsupervised Machine Learning

Unsupervised machine learning is also a very common type of machine learning, and it's key feature is that it uses *unlabelled datasets*. An unlabelled dataset is one with only features, and not specific targets. Consider a dataset of emails, where you are trying to sort into "spam" and "not spam". Then the emails aren't labelled and so you have an unlabelled dataset, and you have to predict based of the features of the email (i.e. its content). Similar to supervised machine learning, there are two main branches of unsupervised machine learning, namely *Clustering* and *Dimensionality Reduction*.

Definition 2.2.1. Clustering is a technique for grouping similar data, where data that is associated is clustered.

A common clustering model would be when you are doing online shopping and they suggest "customers who bought this product also bought:". Clustering is commonly used for market segmentation, where customers are divided into groups based off their demographic and behaviour. The groups can then be targeted by different marketing strategies so that the company can maximise their chance of profiting from the customer. Clustering is also used specifically in medical imaging, where images are split up into different parts in order to examine different types of tissues or bones.

Definition 2.2.2. Dimensionality Reduction is a technique used to reduce the dimension (number of columns) of the dataset into a manageable size without losing the integrity of the data.

Dimensionality reduction is used when the number of features (columns) of the dataset is too large, for example consider a model in which we are predicting the price of a house and the 5 features we have are number of bedrooms, number of bathrooms, square footage of the house (size), crime rate in the area and distance to schools. The first three features can be reduced into a more general feature "size" and the last two features can be reduced to "neighbourhood quality". This keeps the integrity of the dataset but makes it much easier to process. On a much larger scale where we have thousands of features this technique plays a much more important role.

2.3 Comparison of Supervised and Unsupervised Machine Learning

The obvious distinction between supervised and unsupervised machine learning is the type of dataset used - whether it is labelled or not, however aside from this there are other reasons why one method may be more preferable to the other depending on the goal of the model. Drawbacks of supervised learning is that labelling the dataset can be difficult, and requires much more human intervention than unsupervised learning. The model can also take much longer to train than an unsupervised model. However, supervised learning is highly accurate and reliable, and so if you need the model to be trustworthy, for example in medical imaging, then supervised learning can be very useful. Unsupervised learning has key benefits such as the ability to handle very large datasets much more efficiently. Unsupervised learning requires some human intervention when the outputs have to validated as true or false, however much less than a supervised model which is a key benefit. *Semi-supervised* learning can give you the best of both worlds, as both labelled and unlabelled data is used in the model to try and make the most accurate predictions.

3 Linear Regression

Linear Regression and The Method of least mean squares was first documented by R. J. Adock, an attorney in Iowa writing for the American Mathematical journal, *The Analyst*, as a general paper in 1877 but it contained grave mathematical errors. These errors were corrected by Kummel in 1879.[5]

Definition 3.0.1. Linear regression is defined as the process by which a dependent variable can be predicted based upon one or more independent variables.

Definition 3.0.2. If there is just one independent variable this is called "Simple Linear Regression", however, if there are several this is known as "Multiple Linear Regression".

3.1 Simple Linear Regression

Simple Linear Regression is a allows to create a model that is easily represented by a graph of $y = \beta_0 + \beta_1 x$, where β_1, β_0 parameters. Customarily this case, we can consider x to be the independent variable whilst y is the dependent one. x can also be known as the predictor or regressor variable and y, as the response variable. [8]

If we want to create an accurate model for predicting dependent variable y, we must first find the most accurate values of β_0, β_1 possible given a data set with recorded values for x and y, $\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$, $n \in \mathbb{N}$. We can do this by using a technique called The Method of Least Mean Squares.

3.1.1 Method of Least Mean Squares

Theorem 3.1.1 (Method of Least Mean Squares). *(Adapted from [7])*

Our goal is to limit the mean square of the distance that a point, representing a sample value (y_n, X_n), is away from our graph, $y = \beta_0 + \beta_1$. We consider This distance, calculated by

$$\sum_{n=1}^{N} (y_n - (\beta_0 + \beta_1 x_n))^2,$$

as our error term $E(\beta_1, \beta_0)$.

We can see that error is clearly a function of 2 variables, our unknown parameters, β_0 and β_1. To find the best model of $y = \beta_0 + \beta_1 x$, our goal will be to find the best possible values of β_1 and β_0 that will minimize our error term to the highest amount possible. The values at which our error term is at a minimum occurs when the gradient of E with respect to our variables β_0, β_1 vanishes, according to results from multivariable calculus.

Therefore we need to attain

$$\nabla E = (\frac{\partial E}{\partial \beta_1}, \frac{\partial E}{\partial \beta_0}) = (0, 0)$$

. or

$$\frac{\partial E}{\partial \beta_1} = 0, \ \frac{\partial E}{\partial \beta_0} = 0$$

. Differentiating $E(\beta_1, \beta_0)$ gives us the following:

$$\frac{\partial E}{\partial \beta_1} = \sum_{n=1}^{N} 2(y_n - (\beta_0 + \beta_1 x_n)) \cdot (-x_n)$$

$$\frac{\partial E}{\partial \beta_0} = \sum_{n=1}^{N} 2(y_n - (\beta_0 + \beta_1 x_n)) \cdot (-1).$$

If as previously mentioned we set $\partial E / \beta_1 = \partial E / \beta_0 = 0$ and then also simplify by dividing by -2 on both sides of the equation, we get the result:

$$\sum_{n=1}^{N} 2(y_n - (\beta_1 + \beta_1 x_n)) \cdot x_n = 0$$

$$\sum_{n=1}^{N} 2(y_n - (\beta_0 + \beta_1 x_n)) = 0.$$

These equations are also able to be written in the form

$$(\sum_{n=1}^{N} x_n^2)\beta_1 + (\sum_{n=1}^{N} x_n)\beta_0 = \sum_{n=1}^{N} x_n y_n$$

$$(\sum_{n=1}^{N} x_n)\beta_1 + (\sum_{n=1}^{N} 1)\beta_0 = \sum_{n=1}^{N} y_n.$$

Using this we are able to construct a matrix equation which uses our values of β_0 and β_1 which, minimize the previously defined error:

$$\begin{pmatrix} \sum_{n=1}^{N} x_n^2 & \sum_{n=1}^{N} x_n \\ \sum_{n=1}^{N} x_n & \sum_{n=1}^{N} 1 \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_0 \end{pmatrix} = \begin{pmatrix} \sum_{n=1}^{N} x_n y_n \\ \sum_{n=1}^{N} y_n \end{pmatrix}$$

We know this matrix is invertible (its determinate is not equal to 0), so long as at least 2 of the values of x_n are not equal to each other to avoid $\sum_{n=1}^{N} x_n^2$ equaling $(\sum_{n=1}^{N} x_n)^2$ because this would mean the determinate would be 0. We can use this fact to say that

$$\begin{pmatrix} \beta_1 \\ \beta_0 \end{pmatrix} = \begin{pmatrix} \sum_{n=1}^{N} x_n^2 & \sum_{n=1}^{N} x_n \\ \sum_{n=1}^{N} x_n & \sum_{n=1}^{N} 1 \end{pmatrix}^{-1} \begin{pmatrix} \sum_{n=1}^{N} x_n y_n \\ \sum_{n=1}^{N} y_n \end{pmatrix}$$

To find the inverse of the matrix

$$\begin{pmatrix} \sum_{n=1}^{N} x_n^2 & \sum_{n=1}^{N} x_n \\ \sum_{n=1}^{N} x_n & \sum_{n=1}^{N} 1 \end{pmatrix}$$

, that will hence be referred to by letter M, it is required that we find its determinate. This is calculated as such:

$$\det M = \sum_{n=1}^{N} x_n^2 \cdot \sum_{n=1}^{N} 1 - \sum_{n=1}^{N} x_n \cdot \sum_{n=1}^{N} x_n$$

We know by definition that the mean of a data set, $\overline{x}$,$= \frac{1}{N}\sum_{n=1}^{N} x_n$ and its variance, σ_x^2, $= \frac{1}{N}\sum_{n=1}^{N} x_n^2 - \overline{x}^2)$. This allows us to show the following:

$$\det M = \sum_{n=1}^{N} x_n^2 - (N\overline{x})^2$$

$$= N^2(\frac{1}{N}\sum_{n=1}^{N} x_n^2 - \overline{x}^2)$$

$$= N^2\sigma_x^2.$$

Using the result we have obtained for $\det M$, as well as the definition of $\overline{x}$ it is possible to write M^{-1} as

$$\frac{1}{N^2\sigma_x^2}\begin{pmatrix} N & -N\overline{x} \\ -N\overline{x} & \sum_{n=1}^{N} x_n^2 \end{pmatrix}.$$

This allows us to come up with expressions for β_1 and β_0 by expanding the equation:

$$\begin{pmatrix} \beta_1 \\ \beta_0 \end{pmatrix} = \frac{1}{N^2\sigma_x^2}\begin{pmatrix} N & -N\overline{x} \\ -N\overline{x} & \sum_{n=1}^{N} x_n^2 \end{pmatrix} \begin{pmatrix} \sum_{n=1}^{N} x_n y_n \\ \sum_{n=1}^{N} y_n \end{pmatrix},$$

$$\beta_1 = \frac{N\sum_{n=1}^{N} x_n y_n - N\overline{x}\sum_{n=1}^{N} y_n}{N^2\sigma_x^2}$$

$$\beta_0 = \frac{-N\overline{x}\sum_{n=1}^{N} x_n y_n + \sum_{n=1}^{N} {x^2}_n \sum_{n=1}^{N} y_n}{N^2\sigma_x^2}$$

We now have basic model of prediction for real-world problems involving just 2 variables which can be improved upon or **trained** to be a more accurate predictor of the real world by more data samples being collected.

3.2 Linear Regression in machine learning

Whilst a long-known area of mathematics, Linear Regression was limited to the cases of Simple Linear Regression, but starting in the 1970s, the advent of modern computers supervised machine learning models allowed the creation of more complex regression models for predicting future events based upon a greater number of parameters since it was no longer too computationally expensive to be practical [10]. This allows for much better models for real-world complex events such as changes in the stock market.

We have used Linear regression in a wide-reaching range of tasks from law enforcement, like analysing how likely any given transaction is a case of credit card fraud as it is happening or identifying potential suspects of a crime after it has taken place. It's also used to better understand consumer behaviour so retailers can better choose what items to stock and in what quantities as well as assisting warehouses and logistic operations to be more efficient [9]. All of the machine learning applications discussed in section 7 are, at least in part, dependent on regression models to function.

4 Gradient Descent

Before thinking about the mathematical definition of gradient descent, lets consider the intuition for the algorithm.

Imagine you're walking and for some reason you need to get to the lowest point on the landscape, along with that say that you're blindfolded and can only tell what the tangent plane is at your feet. To make things more difficult still you have no memory of where you were the previous step. The logical thing to do is to go downwards until whatever way you go is upwards, and then at least you have found a local minimum. This may seem like a bit of an outlandish situation to find yourself in, but this is exactly what a computer would see if it were performing gradient descent.

4.1 Gradient Descent in $\mathbb{R}$

Gradient Descent is an approach used to optimise a set of parameters when the perfectly efficient values are unknown.

Definition 4.1.1 (Gradient Descent)**.** The **Gradient Descent** Algorithm is an approach to minimising the value of a function $f : \mathbb{R}^n \to \mathbb{R}$.

We start at a given point, which is chosen arbitrarily, and then follow the steepest negative gradient for some defined distance proportional to the "steepness" of the function at that point. Then we repeat until we find a value which is approximately a minimum.

It's easier to see how this works with an example:

Example 4.1.2. Consider the function $f : \mathbb{R} \to \mathbb{R}; x \mapsto x^3 + 3x^2 + x$:

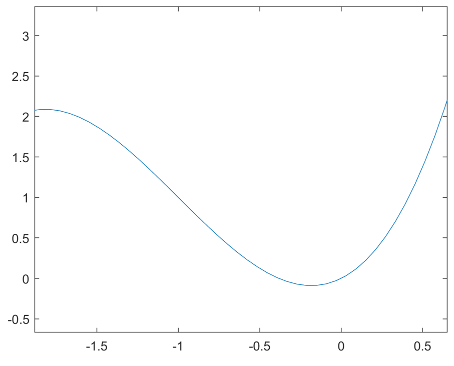

Figure 1: Plot of f

In this example, we can directly calculate the minimum of the function, however for the sake of the example, let's use gradient descent.

We begin by calculating the gradient function of f, which gives us

$$\frac{\partial f}{\partial x} = 3x^2 + 6x + 1.$$

Then we can move down from our starting point a_0 to a_1 and so on, by the forumula

$$a_n = a_{n-1} - \alpha \frac{\partial f}{\partial x}(a_{n-1}).$$

[11] This formula includes this α term which in some contexts, namely neural network backpropagation (See Section 6), is called the **learning rate**. This defines how quickly the algorithm moves towards the local minimum. This begs the question, why not set α to be extremely large. If we look at what alpha means geometrically we can see why this isn't a good idea.

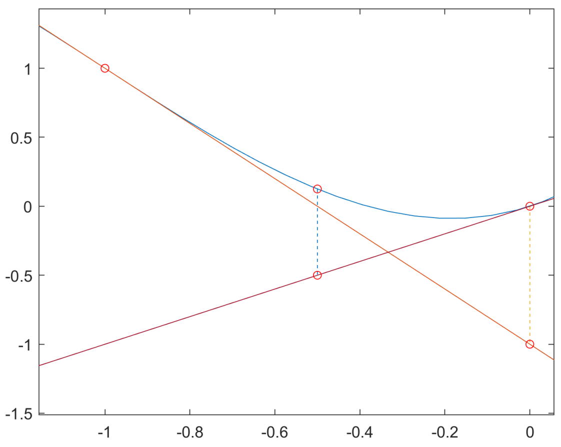

Figure 2: Gradient Descent on f

Here, we've applied gradient descent with $\alpha = \frac{1}{2}$ for 2 iterations from a starting point at $x = -1$. Clearly, we can see the geometric intuition for the gradient descent algorithm. Since the rate at which we change our estimate of the value for which f is minimised is proportional to the gradient at the point of the current estimate, as we get closer to the true value, we see that the change between each iteration reduces. If α were significantly larger, it may be the case that this approach would take much longer, if it happens at all.

Remark. Notice that I have used "local minimum" instead of just minimum here. This is important, especially with the function we have chosen, because it is a cubic. This means that it doesn't actually have a global minimum. In particular, f is minimised, in a sense, as $x \to -\infty$. This is one of the major drawbacks of the gradient descent algorithm, since we can never be sure if we've reached the truly optimal position.

4.2 Gradient Descent in $\mathbb{R}^n$

We can extend the work we have done with a single input dimension to many dimensions. For the sake of visualisation, we will talk about functions
$f : \mathbb{R}^2 \to \mathbb{R}$ but this same approach will apply to any function from $\mathbb{R}^n$ to $\mathbb{R}$.

Example 4.2.1 (The Circular Paraboloid)**.** Conisder,

$$f : \mathbb{R}^2 \to \mathbb{R}; \begin{pmatrix} x \\ y \end{pmatrix} \mapsto x^2 + y^2.$$

Note: This function is called a circular paraboloid, since the vertical cross sections form parabolas and the horizontal cross sections form circles.

To descend this function, we take the same approach as we did for $\mathbb{R}$, but extended. This gives us the revised iterative formula,

$$\mathbf{a}_n = \mathbf{a}_{n-1} - \alpha \nabla f(\mathbf{a}_{n-1}).$$

The only change is instead of considering a value in $\mathbb{R}$, we are considering a value in $\mathbb{R}^n$, and instead of considering the derivative, we are now thinking about the gradient function.

By considering the normal to the level set of this function, we can easily find the descent direction. The level set $L_c = \{(x, y) | x^2 + y^2 = c\}$. This forms about the origin with radius $\sqrt{c}$. The normal to a circle is always in the direction away from the centre of the circle. This is a result from geometry. Hence, it is clear to see that each descent step will be pointing towards the origin.

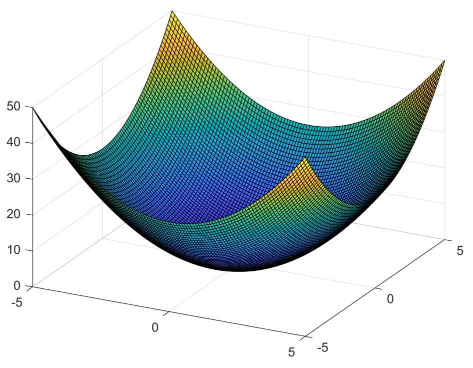

Figure 3: Surface of f

Then we see that the limit of our approximation is the origin of the function, which as it turns out is the global minimum of this function.

We will now see a case where our choice of starting position will impact our estimation,

Example 4.2.2 (Peaks)**.** For this example, we use the built-in MATLAB function `peaks()`:

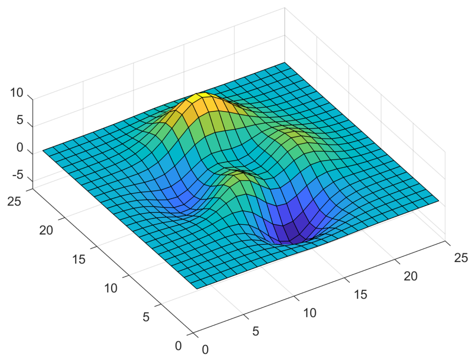

Figure 4: Plot of the peaks function

For this example, it is clear to see that picking different starting points will result in different minima being achieved. This highlights the importance of picking starting points when considering gradient descent.

4.3 Comments on Gradient Descent

In the examples given, it is possible to find a solution to the global minimum analytically. However, in many applications this isn't the case. Generally we use gradient descent on complicated functions, where changing the parameters of one has a large impact on the other, so simply solving for $\nabla f = \mathbf{0}$ isn't an option. The primary modern day context for gradient descent is in neural network training, however it could be used in many examples.

5 Logistic Regression

Logistic Regression falls under the umbrella of supervised machine learning, in particular, it is used for predicting the probability of a dichotomous (binary) event occurring. The difference between linear regression and logistic regression is that with linear regression, the output is continuous whereas with logistic regression, the output is discrete. Logistic regression is an example of a *classification model.* There are three main types of logistic regression models based on the type of outputs we are predicting.

1. Binary logistic regression - this is when the event has two outcomes, e.g. whether a patient has COVID-19 or not.
2. Multinomial logistic regression - this is when we have multiple binary events as outcomes, e.g. whether a patient has COVID-19, flu, an allergy, or not.
3. Ordinal logistic regression - this is when the outcomes are ordered, e.g. if we are predicting the severity of COVID-19 out of mild, moderate, severe.

First we need to define the *sigmoid function* which is also referred to as the *logistic function.*

Definition 5.0.1 (The Sigmoid Function)**.**

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

The goal of logistic regression is to take a linear regression graph and squeeze it so its between the values 0 and 1 (because all values for probability are in this range). We know from above that the equation of best fit in linear regression is $y = \beta_0 + \beta_1 x$. However instead of taking values y from the whole real number line, we are taking probabilities from the range 0 to 1. To overcome this issue we take the *logit odds* which are defined as follows.

Definition 5.0.2 (Logit Odds)**.** Let p be the probability of an event.Then

$$logit(p) = log\left(\frac{p}{1-p}\right)$$

Hence we can then solve this equation for p (as our goal is to predict the probability p)

$$log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x$$

Taking exponents gets

$$\frac{p}{1-p} = e^{\beta_0+\beta_1 x}$$

Rearrange to get

$$p = e^{\beta_0+\beta_1 x} - pe^{\beta_0+\beta_1 x}$$

$$p(1 + e^{\beta_0+\beta_1 x}) = e^{\beta_0+\beta_1 x}$$

$$p = \frac{e^{\beta_0+\beta_1 x}}{1 + e^{\beta_0+\beta_1 x}}$$

$$p = \frac{1}{1 + e^{-(\beta_0+\beta_1 x)}}$$

This is our logistic function (notice how it is in the same form as the sigmoid function).

6 Neural Networks

Neural networks describe a broad category of models, all of which take their basic structure from that of the brain, although the goal is in no way to model the brain and where we spot inefficiency in nature we are quick to discard it for the sake of our algorithms. [4, page 3]

Neural networks essentially are a complicated example of a regression algorithm. (Called a universal approximator) [11, page 316] We give the machine n inputs and we ask it to produce m outputs. The machine's job is to make the outputs as close to what they "should be" as possible. So, to put it more mathematically, the goal of a neural network is to approximate a function $f : \mathbb{R}^n \to \mathbb{R}^m$ as best as possible.

6.1 Deep feedforward networks

Definition 6.1.1 (Deep feedforward network)**. Deep feedforward networks**, also called **feedforward neural networks** or **multilayer perceptrons** (MLPs), are examples of a neural network. A feedforward network's goal is to approximate some unknown function f^*. It does this by defining some generalised function $\mathbf{y} = f(\mathbf{x}; \theta)$ with parameters θ which are to be approximated by our algorithm. These are called **feedforward** because the information being evaluated by the model flows exclusively forwards, with no feedback, to our output. [6]

Although, we will only discuss networks with a layered structure, it is important to point out that a neural network could be of any structure. This structure is quite different from a typical network design and in some

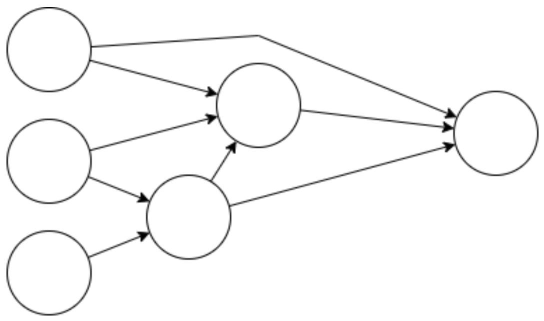

Figure 5: Example of a non-standard network structure

applications may in fact be the best option for the particular situation, however for the ease of computing we typically ignore these kinds of network structures in favour of more simplistic ones.[6]

Remark. It is also the case that a layered structure can mimic a non-layered network in general. [3, page 230]

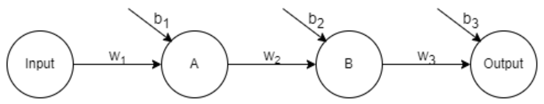

Figure 6: Simple Standard Neural Network

This is a typical structure for a neural network, although it is sparse compared to more complicated networks we will begin to discuss. This example is useful, because by developing our framework here first, it becomes very easy to scale up to the case of large networks. We'd like for the output of each node to be dependent on it's bias, weight to the previous node and the previous node. It may be the case that we could write this formula in a multitude of ways, but by convention we use the following,

$$z^{(n)} = b^{(n)} + w^{(n)} a^{(n-1)}$$

where $a^{(n)}$ is the output of the node on layer n. Notice that if we say that $z^{(n)} = a^{(n)}$ i.e. the output of the node is simply equal to this combination defined above, we find that each node is linear to the last. By following this through the network, we find that our output $\hat{y} = \alpha + \beta x$ for some α, β. This significantly limits the sorts of functions our model can converge to. We solve this using the activation function; an activation function de-linearises our results in order to let the model approximate a far more complicated set of functions. So, write $z^{(n)} = \sigma(a^{(n)})$, where $\sigma : \mathbb{R} \to \mathbb{R}$ is our activation function. So now we can write out a function for the output of our neural network,

$$\hat{y} = \sigma(b_3 + \sigma(b_2 + \sigma(b_1 + xw_1)w_2)w_3).$$

The equations we have found here define, for a simple case, an algorithm called **forward propagation**. To generalise to the case of multiple nodes on each layer we simply add up across all the different influences into the node, as we do in this simple case between the input from the previous node and the bias. Now, let's look at a few definitions:

Definition 6.1.2 (Weights and Biases)**.** We define **weights** and **biases** as follows:

- **Weight.** A weight is a measure of the influence a node has over a node in the next layer. For the weight connecting from node j on layer $l-1$ to node i on layer l we write $w_{ij}^{(l)}$. This is useful notation when considered in the context of Linear Algebra, since for each layer we can define a weights matrix $\mathbf{W}^{(l)} = (w_{ij}^{(l)})$, which as we will see will make computations far easier.

- **Bias.** A bias can almost be thought of as an extra weight from a node which always takes a value of 1. We write the bias of node i of layer l as $b_i^{(l)}$

Definition 6.1.3 (Activation Functions and Node Activations)**.** The **node activation** is the output of the given node. We calculate this using the following formula:

$$a_i^{(l)} = \sigma\left(b_i^{(l)} + \sum_k w_{ik}^{(l)} a_k^{(l-1)}\right),$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the **activation function**, and

$$z_i^{(l)} = b_i^{(l)} + \sum_k w_{ik}^{(l)} a_k^{(l-1)}$$

are the **pre-activations**.
If we define σ on a vector to work element-wise we can do something more useful:

$$\mathbf{a}^{(l)} = \sigma\left(\mathbf{b}^{(l)} + \mathbf{W}^{(l)}\mathbf{a}^{(l-1)}\right),$$

where $\mathbf{a}^{(l)}$ is a vector of all the activations on the layer l.
Similarly,

$$\mathbf{z}^{(l)} = \mathbf{b}^{(l)} + \mathbf{W}^{(l)}\mathbf{a}^{(l-1)}$$

is the vector of the pre-activations.

Now that we have defined the weights, biases and activations, we can write an algorithm for forward propagation of our network.

Algorithm 1 Forward Propagation. The algorithm starts from the inputs nodes $\mathbf{x}$ and works step by step until we find the activations on the final layer l, which are our predicted outputs $\hat{\mathbf{y}}$.
Note: We will see why this notation for the outputs is used when looking at backpropagation

```
Require: Network depth, $l$
Require: $\mathbf{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model.
Require: $\mathbf{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model.
Require: $\mathbf{x}$, the input to the network.
  $\mathbf{a}^{(0)} \leftarrow \mathbf{x}$
  for $k = 1, \ldots, l$ do
  $\mathbf{z}^{(k)} = \mathbf{W}^{(k)}\mathbf{a}^{(k-1)} + \mathbf{b}^{k}$        ▷ Here we calculate the pre-activations for each node
  $\mathbf{a}^{(k)} = \sigma\left(\mathbf{z}^{(k)}\right)$  ▷ Here we apply the activation function to the pre-activation in order to generate the activation
  for that layer.
  end for
  $\hat{\mathbf{y}} \leftarrow \mathbf{a}^{(l)}$
[6, p. 205]
```

6.2 Cost and Loss Functions

In order to evaluate the performance of a network, we need to define some function which tells us exactly how wrong it is compared to other networks of the same architecture. The function we use to evaluate this is called a **cost function**. Typically we use something called the **mean-squared error** (**MSE**) which is calculated as follows:

$$C = \sum_{i=0}^{n} (\hat{y}_i - y_i)^2$$

Here, $\hat{y}_i$ is the predicted value created by the network, while y_i is the true output of the training example.

Remark. You may wonder why we omit the $\frac{1}{n}$ from this definition, but since the number of outputs is the same each time, our MSE error is just scaled so comparison is still useful. (More importantly, it simply applies a scalar to the learning rate, which we will see more about later).

6.2.1 Loss Functions

A **Loss Function** is slightly different to a cost function in the fact that it is computed over a large number of forward propagations. The point of this is to reduce the sensitivity of the network to anomalous data, which might otherwise slow down learning. In general we take the loss function of a network over a set of training data and then perform repeated backpropagation on that sample to let the network converge. This is typically quite computationally expensive, so we often subdivide our sample data and work with the loss functions of these portions of the overall data one at a time. This will make more sense after the next section. [1]

6.3 Training the Network

An important part of a nerual network is training. To do this we'd think that moving in the direction which reduces the error on the output is a reasonable approach. So, we set out with the goal of developing an algorithm that finds which way to adjust each weight and bias in order to reduce the error function, with the goal of reaching a local minima on error, or if we're lucky, eliminate the error entirely. This is the basis of the backpropogation algorithm.

In the earlier examples of linear regression, we saw that error can be minimised by an analytical approach, however in general this isn't the case for a neural network.

6.3.1 Backpropagation

To discuss backpropogation, let's start with a trivial example in order to develop an understanding for higher dimensions. Look again at Figure 6.1. We want to find out the derivative of the cost function with respect to each of the variables we can alter directly. To do this we apply the chain rule. For the sake of completeness we write the chain rule as follows:

Theorem 6.3.1 (The Chain Rule)**.** *Let $y = f(g(x))$ for some differentiable functions f, g, and define $u = g(x)$. Then*

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}.$$

This result is so useful for us because it means that we can find the gradients with respect to the weights and biases in a sequential process rather than directly computing each result individually.

Let's go back to the simplistic network of Figure 6.1, since it can provide our intuition for the larger networks. So, the activation of the node on layer n is defined as follows:

$$a^{(n)} = \sigma(z^{(n)}) = \sigma(b^n + a^{(n-1)}w^{(n)}).$$

Hence, by the chain rule, we find:

$$\frac{\partial a^{(n)}}{\partial b^{(n)}} = \frac{\partial z^{(n)}}{\partial b^{(n)}}\frac{\partial a^{(n)}}{\partial z^{(n)}} = 1 \cdot \sigma'(z^{(n)}) = \sigma'(z^{(n)}),$$

$$\frac{\partial a^{(n)}}{\partial w^{(n)}} = \frac{\partial z^{(n)}}{\partial w^{(n)}}\frac{\partial a^{(n)}}{\partial z^{(n)}} = a^{(n-1)} \cdot \sigma'(z^{(n)})$$

and,

$$\frac{\partial C}{\partial a^{(n)}} = \frac{\partial z^{(n+1)}}{\partial a^{(n)}}\frac{\partial a^{(n+1)}}{\partial z^{(n+1)}}\frac{\partial C}{\partial a^{(n+1)}} = w^{(n+1)} \cdot \sigma'(z^{(n+1)}) \cdot \frac{\partial C}{\partial a^{(n+1)}}.$$

This isn't yet exactly what we wanted but then notice that,

$$\frac{\partial C}{\partial b^{(n)}} = \frac{\partial C}{\partial a^{(n)}}\frac{\partial a^{(n)}}{\partial b^{(n)}}$$

and,

$$\frac{\partial C}{\partial w^{(n)}} = \frac{\partial C}{\partial a^{(n)}}\frac{\partial a^{(n)}}{\partial w^{(n)}}.$$

So given $\frac{\partial C}{\partial a^{(n+1)}}$, we know $\frac{\partial C}{\partial a^{(n)}}$. This gives us every value apart from the starting one, which is given by the nodes on the output layer. In our example, this would be layer 3, but for generality we will call the last layer L.

To unravel what to do here, we need to remember that $a^{(L)} = \hat{y}$.

Then,

$$C = (a^{(L)} - y)^2.$$

Hence,

$$\frac{\partial C}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

So from this we can work our way through all of the derivatives with respect to the layer activations, and then compute the gradients along the weights and biases. We can extend this definition to our general network just as we did when moving to our full definition of forward propagation. It can be seen that we arrive at the following equations:

$$\frac{\partial C}{\partial a_j^{(l)}} = \begin{cases} \sum_j \left(w_{jk} \cdot \sigma'(z_j^{(l-1)}) \cdot \frac{\partial C}{\partial a_j^{(l)}} \right) & \text{if } l \neq L \\ 2(a_j^{(L)} - y_j) & \text{otherwise} \end{cases},$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \cdot \sigma'(z_j^{(l)}) \cdot \frac{\partial C}{\partial a_j^{(l)}},$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \sigma'(z_j^{(l)}) \cdot \frac{\partial C}{\partial a_j^{(l)}}.$$

So, we have the gradients needed to backpropagate, but we're missing what we do with these gradients. Previously, we looked at gradient descent, and again here that algorithm is how we develop the network. We write the following,

$$w_{jk}^{(l)} \leftarrow w_{jk}^{(l)} - \alpha \frac{\partial C}{\partial w_{jk}^{(l)}},$$
$$b_j^{(l)} \leftarrow b_j^{(l)} - \alpha \frac{\partial C}{\partial b_j^{(l)}}.$$

Here we call α the **learning rate**, however by thinking about this as a gradient descent we can see that a very high learning rate won't let our network spiral ever closer towards the singularity and in fact will most likely make our network converge slowly if at all.

So now we have all the tools to write out an algorithm for backpropagation:

Algorithm 2 Backpropagation. This algorithm takes the results of forward propagation and then calculates all our gradients from those values.

```
Require: Network depth, $l$
Require: $\mathbf{W}^{(i)}, i \in \{1, \dots, l\}$, the weight matrices of the model.
Require: $\mathbf{b}^i, i \in \{1, \dots, l\}$, the bias parameters of the model.
Require: $\mathbf{x}$, the input to the network.
Require: $\mathbf{y}$, the desired output given that input.
Require: $\alpha$, the learning rate.
  Using the Forward Propagation Algorithm we calculate all the $\mathbf{z}^{(k)}$ and $\mathbf{a}^{(k)}$, including $\hat{\mathbf{y}}$.
  for all $\mathbf{a}^{(l)}$ do                          ▷ We move from the output layer backwards.
      for all $a_j^{(l)}$ in $\mathbf{a}^{(l)}$ do
          $\frac{\partial C}{\partial a_j^{(l)}} \leftarrow \begin{cases} \sum_j \left( w_{jk} \cdot \sigma'(z_j^{(l-1)}) \cdot \frac{\partial C}{\partial a_j^{(l)}} \right) & \text{if } l \neq L \\ 2(a_j^{(L)} - y_j) & \text{otherwise} \end{cases}$
      end for
  end for
  Now we have the activation gradients, we compute the weight and bias gradients:
  for all $w_{jk}^{(l)}$ do
          $\frac{\partial C}{\partial w_{jk}^{(l)}} \leftarrow a_k^{(l-1)} \cdot \sigma'(z_j^{(l)}) \cdot \frac{\partial C}{\partial a_j^{(l)}}$
          $w_{jk}^{(l)} \leftarrow w_{jk}^{(l)} - \alpha \frac{\partial C}{\partial w_{jk}^{(l)}}$
  end for
  for all $b_j^{(l)}$ do
          $\frac{\partial C}{\partial b_j^{(l)}} \leftarrow \sigma'(z_j^{(l)}) \cdot \frac{\partial C}{\partial a_j^{(l)}}.$
          $b_j^{(l)} \leftarrow b_j^{(l)} - \alpha \frac{\partial C}{\partial b_j^{(l)}}$
  end for
[1]
```

6.3.2 Evolutionary Algorithms

The backpropagation algorithm works well because it ensures every parameter moves in the direction which improves the output of the network, however in some cases it may be computationally expensive or perhaps impossible to compute the gradients of the parameters of the network. In this case, we need a different approach. A whole family of solutions, which is often associated with Neural Networks and machine learning in general, is the idea of evolutionary algorithms. These methods use randomness in order to improve the network over time.

Taking inspiration from real-world evolution, we clearly need some sort of environmental pressures in order to get our network to evolve in the direction we'd like. So we define something called a **Fitness Function**. Obviously, this is reminiscent of the idea of "survival of the fittest", which is exactly how we train our network here.

Broadly, to train a network with an evolutionary algorithm, we follow these steps:

1. Generate a collection of networks where each parameter is chosen at random from a given random variable.
2. Evaluate each networks fitness (often just the loss function) and disregard all but the best performing networks.
3. Duplicate the remaining networks and modify the parameters of the new set of networks by random values.
4. Repeat.

Remark. To keep our tie-in with evolution, we call each iteration of this learning algorithm a **generation**. We talk about members of a generation as the networks which were generated at the start of that generation.

Although this algorithm may look simple and elegant, it is often very inefficient computationally and only useful where a way to evaluate the derivatives is impossible, or too complicated. It also doesn't seem immediately obvious that this will converge. By thinking about what the resulting 'best' network will look like after each generation, we can convince to ourselves that it will approximate the function after a sufficient number of generations. Evolutionary Algorithms are a type of a broader method of simulation, which leverage randomness to develop a solution.

6.4 Other Types of Neural Network

Although Deep feedforward networks are an interesting and essential thing to understand in the field of machine learning, modern research focuses on much more advanced models that deal with the issues of the deep feedforward network.

For instance, we brushed over the issue that the dimensions of the network were chosen by a human at the start. How do we know that this is the most efficient structure? There are neural networks that are capable of adding and removing nodes as need be.

Also, what happens when we put an image into our network? We may just be able to enter a list of pixel values as inputs to the network, but its more effective to consider the image as a whole first. We apply a series of "filters" to the image, called Convolutions, then input the final result into a deep feedforward network as we have discussed. The goal of a convolutional neural network is to also pick the parameters on these convolution layers in order to improve the output.

Unfortunately, we have been unable to discuss these interesting types of neural network in more detail, instead focussing on the deep feedforward network, as it is arguably the basis of modern machine learning.

7 Applications of Machine Learning

By now we have seen that machine learning, and Artificial Intelligence in general, has a variety of uses and applications. These uses are present in all fields from the creative fields such as art - where AI can take a few words and generate a unique piece of art from those words - to the medical sciences - where AI can be used to aid precise and easy diagnosis. Here we will talk about the range of applications of machine learning.

7.1 Medical Diagnosis

Throughout history faulty or absent medical diagnosis has lead to an uncountable amount of curable diseases and cancers left undiscovered. . This large issue has been addressed by many scientists and using Machine Learning we are developing ways to predict and classify cancers. This is all mainly through image recognition which we investigate further later in the chapter.

For example, Rashmi Agrawal plunged into investigating Machine Learning in breast cancer and published her findings in 2019 [2]. First we define data mining; which can be seen as a process which involves machine learning which extracts important data from a large data set. Within data mining we observe two other techniques known as classification and clustering which help sort data units into stratum where the units in each strata have the features that we are looking for.

This relates to breast cancer diagnosis as the data mining and machine learning can predict whether the cancer is malignant or benign and thus determine the action needed to aid the patients health. Agrawal also discussed how these techniques also reduce the number of false positives and negatives which further advantages the medical field and society as a whole.

However, the prediction use in medicine doesn't end at breast cancer, there is also positive development in the use of Machine Learning to predict future heart problems. The AI filters through countless medical files to find patterns that lead to heart diseases. This can be done with much greater efficiency than if a human was to try.

7.2 Language Translation Algorithms

We can also see that Machine Learning holds a quintessential role in the accuracy of computer translations, most commonly known as Google Translate. For this application to function we input a large list of examples of phrases into the AI then the AI learns through pattern recognition through the large amount of examples we inputted.

Here, we will look at the example of translating English to French. Without the translation algorithm being able to adapt and learn we would have to tell it what to do given each of infinite phrases, this is clearly impossible as you cant translate infinite phrases and input them into a program. One way around this would be to translate each word in the sentence into French; this causes problems. For example, if we were to translate "The boy had blue eyes." using this method we would obtain "Le garcon a bleu yeux.". Although a french person should be able to perceive the message it isn't proper French, the correct translation would be "Le garcon a les yeux bleus.".

Now we can see that there are important grammar rules that come into play in both English and French but these aren't necessarily the same, so the machine needs to learn these grammar rules from the examples we inputted and evolve. As computers don't generally use English we must have a network that converts sentences into vectors so the AI can process it then another network that converts vectors into French.

7.3 Image Recognition

Image recognition is essentially used for facial recognition and to help make self-driving cars a possibility. Using similar techniques we have seen before, such as inputting an abundance of images into the machine learning algorithm, the AI is able to spot features in images very precisely. With Face ID on Apple iPhones the phone takes infrared photos every couple of seconds when you are on it so it can learn how to recognise your face better.

With this, the AI can easily distinguish between a photo of a cat and a dog. Through the help of linear discriminant analysis (and other algorithms) the machine can separate and characterise different features and therefore place the image into the different categories. Now this on its own seems pointless as the human brain can easily distinguish between a photo of a dog and a cat, however, we see its practicality come through when we start looking at law enforcement and in the self-driving cars.

The growing popularity of CCTV imaging in the last 40 years has lead to a great decrease in crime and an increase in prosecutions for crimes. However, this would not be as useful if you were unable to identify who it is in the picture carrying out the crime. Machine learning, importantly image recognition, has helped develop this and in turn develop the world into a safer place.

Furthermore, with the first models of self-autonomous cars appearing in the recent years there is a high likelihood that the transport industry will be revolutionised, subsequently eliminating human error within driving and leading to less car crashes and deaths due to driving .

7.4 Predictive Maintenance

Predictive maintenance relates to the use of machine learning algorithms to forecast when machinery is likely to breakdown and malfunction. This enables the workers to prevent predicted problems before they happen, thus decreasing the cost for firms and production lines to replace damaged machinery. This consequently decreases the cost of production and therefore makes goods cheaper for us consumers.

This process is done by placing sensors on the machines that monitor the temperature and the accuracy of their production. This collected data is then imputed into a machine learning algorithm which can detect when the machine is likely to fail and notify the workers.

7.5 Recommendations

For users of Google, Safari and other search engines, you may have found that if you search something into Amazon, such as a new aftershave, that (even if purchase said product) you are bombarded with advertisements for similar products no matter what website you are on. Machine learning is to blame.

Here the data collected is tour search and purchase history, then the algorithms spot patterns in your behaviour and use this to determine what you may want to purchase in the future.

This can be seen as a positive for both consumers and firms. Consumers are subject to more products they may not have seen if not for the advertisements pushing them, this may increase their quality of life. Firms also see their revenue go up as their products are now target at consumers who are more likely to buy them, increasing revenues.

7.6 Limitations

This section has commented on the many applications of machine learning and their effect on society. However, there are some difficulties in the performance of the algorithms.

One possible limitation of machine learning could be the data set. We have seen that the machine algorithms analyse the data that has been inputted into the system and then they find patterns so they can evolve, this can be considered problematic if the original data that the machine observes is corrupted. Bias and limited quantity of data could lead to false predictions.

Furthermore, the problems that arise from the faulty data set can be difficult to identify and may take a while to fix or find where the source came from.

References

[1] 3Blue1Brown. *Backpropagation calculus — Chapter 4, Deep learning.* Nov. 2017. URL: `https://www.youtube.com/watch?v=tIeHLnjs5U8&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=4`.

[2] Rashmi Agrawal. 'Predictive Analysis Of Breast Cancer Using Machine Learning Techniques'. In: *Ingeniería Solidaria* 15.3 (Sept. 2019), pp. 1–23. DOI: `10.16925/2357-6014.2019.03.01`. URL: `https://revistas.ucc.edu.co/index.php/in/article/view/2927`.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[4] Wolfgang Ertel. *Introduction to AI.* 2nd ed. Springer Cham, 2016.

[5] David J Finney. ‘A note on the history of regression’. In: (1996).

[6] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning.* The MIT Press, 2016.

[7] Steven J Miller. ‘The method of least squares’. In: *Mathematics Department Brown University* 8 (2006).

[8] Douglas C Montgomery, Elizabeth A Peck and G Geoffrey Vining. *Introduction to linear regression analysis.* John Wiley & Sons, 2021, p. 2.

[9] Iqbal H Sarker. ‘Machine learning: Algorithms, real-world applications and research directions’. In: *SN computer science* 2.3 (2021), p. 160.

[10] Cosma Rohilla Shalizi. ‘The Truth About Linear Regression’. In: *Online Manuscript. http:///www. stat. cmu. edu/˜ cshalizi/TALR* (2015).

[11] Jeremy Watt, Reza Borhani and Aggelos K. Katsaggelos. *Machine Learning Refined.* Cambridge University Press, 2020.

Appendices

A Python Deep Feedforward Neural Network

For the research of neural networks, I programmed a python neural network. **Note:** The Code is inefficient and suffers from rounding errors, which prevent it from being useable, but it was a useful tool in testing my intuition of the backpropagation algorithm.

```python
import numpy as np

def sigmoid(x): #The sigmoid function, which gives us the non-linear activations
    return 1/(1+np.exp(x))

def diffsigmoid(x): #The first derivative of the sigmoid function
    return -(np.exp(x))/((1+np.exp(x))**2)
class layer:
    def __init__(self, inputs, nodes): #nodes = The number of nodes in this layer, inputs = the number of nodes in the previous layer
        self.w = np.random.uniform(-10, 10,(nodes, inputs))
        self.length = nodes

    def run(self, values):
        return self.w@values


class network:
    def __init__(self, dimensionality): #layers is an ordered list of the dimensionality of our network
        self.layers = []
        self.preActivations = []
        self.activations = []
        self.dimensionality = dimensionality
        i = 0
        for dimension in dimensionality:
            self.layers.append(layer(dimension, dimensionality[i+1]))
            i+=1
            if(i+1 >= len(dimensionality)):
                break



    def run(self, inputs): # We run through our network forwards.
        for level in self.layers:
            inputs = level.run(inputs)
            self.preActivations.append(inputs)
            inputs = sigmoid(inputs) #We apply the sigmoid function after in order to record the pre-activation values
            self.activations.append(inputs)
        return self.activations

    def backprop(self, error): # error = the difference between the network output and the true value.
        return True



#We need some way to backpropogate our network

class trainer:
    def __init__(self, network, inputs, outputs): #Inputs are the raw inputs we put into the model, while outputs are the expected outputs.
        self.network = network
        self.networkActivations = network.run(inputs)
        self.outputs = outputs
        self.activationGradients = [None] * len(self.network.layers)
        self.weightGradients = [None] * len(self.network.layers)
        self.biasGradients = [None] * len(self.network.layers)

    def calcActivationGradients(self):
        #For our last layer the activations are different since
        layerActivationGradients = []
        for j in range(0,self.network.layers[-1].length): #Since the last layer uses a different equation, we calculate the gradient of that layer seperately
            layerActivationGradients.append(2*(self.networkActivations[-1][j] - self.outputs[j]))
        self.activationGradients[-1] = layerActivationGradients

        for l in range(2,len(self.network.layers)+1):
            for k in range(0,self.network.layers[-l].length):
                layerActivationGradients = []
                for j in range(0,self.network.dimensionality[-l-1]):
                    layerActivationGradients.append(self.network.layers[-l].w[k][j]*diffsigmoid(self.network.preActivations[-l][j])*self.activationGradients[-l+1][j])
                self.activationGradients[-l]=layerActivationGradients

    def calcWeightGradients(self):
        for l in range(1,len(self.network.layers)+1):
            layerWeightGradients = []
            for j in range(0,self.network.dimensionality[-l-1]):
                nodeWeightGradients = []
                for k in range(0,self.network.layers[-l].length):
                    nodeWeightGradients.append(self.network.activations[-l][j] * diffsigmoid(self.network.preActivations[-l][j]) * self.activationGradients[-l][j])
                layerWeightGradients.append(nodeWeightGradients)
            self.weightGradients[-l] = layerWeightGradients

    def calcBiasGradients(self):
        for l in range(1,len(self.network.layers)+1):
            layerBiasGradients = []
            for j in range(0,self.network.dimensionality[-l-1]):
                nodeBiasGradients = []
                for k in range(0,self.network.layers[-l].length):
                    nodeBiasGradients.append(diffsigmoid(self.network.preActivations[-l][j]) * self.activationGradients[-l][j])
                layerBiasGradients.append(nodeBiasGradients)
            self.biasGradients[-l] = layerBiasGradients

    def backpropogate(self):
        self.calcActivationGradients()
        self.calcWeightGradients()
        for l in range(0,len(self.network.layers)):
            self.network.layers[l].w = np.array(self.network.layers[l].w) - np.transpose(np.array(self.weightGradients[l])) #Update the weight gradients
        return True
network1 = network([2,3,3,3])
trainer1 = trainer(network1, [1,1], [1,1,1])
trainer1.calcActivationGradients()
for i in range(0,1000):
    print(i)
    trainer1.backpropogate()
trainer1.network.run([1,1])
print(trainer1.network.activations)
```

Program 1: Python Deep Feedforward Network