Tales of a Traveller

By Jai Sharma

Journey through the woods of Artificial Intelligence & Machine Learning (AI & ML)

The woods are lovely dark and deep,
But I have promises to keep.
And miles to go before I sleep, and miles to go before I sleep
Robert Frost

Each has their own journey through the deep woods of AI & ML. My own exploration has had many stops and starts, stumbles and falls and change in direction. Along the way I have been picked up and helped along by fellow travellers who have told me wondrous tales of their adventures - in the form of blogs, papers, videos and books and, I have quoted from these extensively.

This series of posts is written more in the style of personal reminder notes. If you have stumbled upon them, you are welcome to it. Indeed if you find some use, my pleasure will be doubled. On the other hand, if you feel some observations are not quite right, please feel free to send me an email mailto:jai.sharma@bigpond.com or a DM so I can course correct.

Deep Learning in perspective

There has been an enduring fascination at the prospect of machines "thinking" like humans and perhaps even out-thinking them! This fascination has spawned an entire genre of sci-fi and virtual reality movies and now, a whole new industry with driver-less cars, intelligent devices and what have you.

Artificial Intelligence (AI) has fundamentally been concerned with computer systems simulating essentially intellectual processes. One of the earliest forays into so called AI was, programming a computer to play chess. This was really a set of pre-programmed winning moves fed into the computer as an instruction set. Francois Chollet describes this as *symbolic AI.* This conforms to the early view of Lady Ada Lovelace: *The analytical engine has no pretensions whatsoever to originate anything. It can only perform whatever we know how to order it to perform!*

This remark was later quoted and picked up by Dr Alan Turing as "Lady Lovelace's objection" and forms the crux of the "Turing test" which determines what can truly be classified as *AI.*

Machine Learning (ML) arises from the question - *Can a computer go beyond what we order it to perform?* ML may be considered as a subset of AI whereby a computer system is "trained" and thereby learns based on known data.

In classical programming, a computer system applies a set of pre-programmed rules to evaluate input data and produces an output. In contrast, Machine learning is a different paradigm whereby a computer is expected to *derive* the rules from a labelled input dataset!

How does it to do it? As we shall see, it's all mathematics.

Deep learning (DL) may be considered a further refinement of general ML approaches utilising a number of layers of artificial neural networks (ANNs) to extract progressively accurate information. Thus raw data is progressively transformed into a more meaningful representation as it passes through each successive layer.

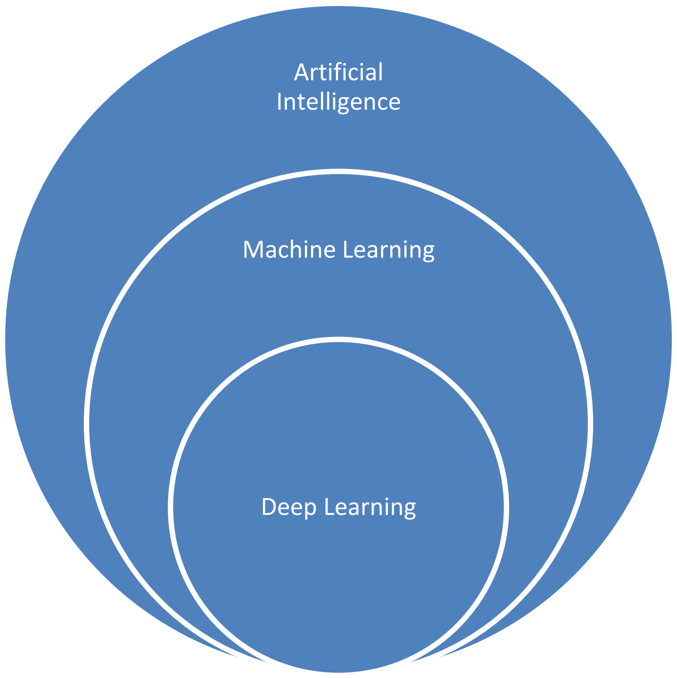


Figure 1: From the book – "Deep learning with Python" by Francois Chollet

Deep Learning Scenarios

The following diagram illustrates basic deep learning types

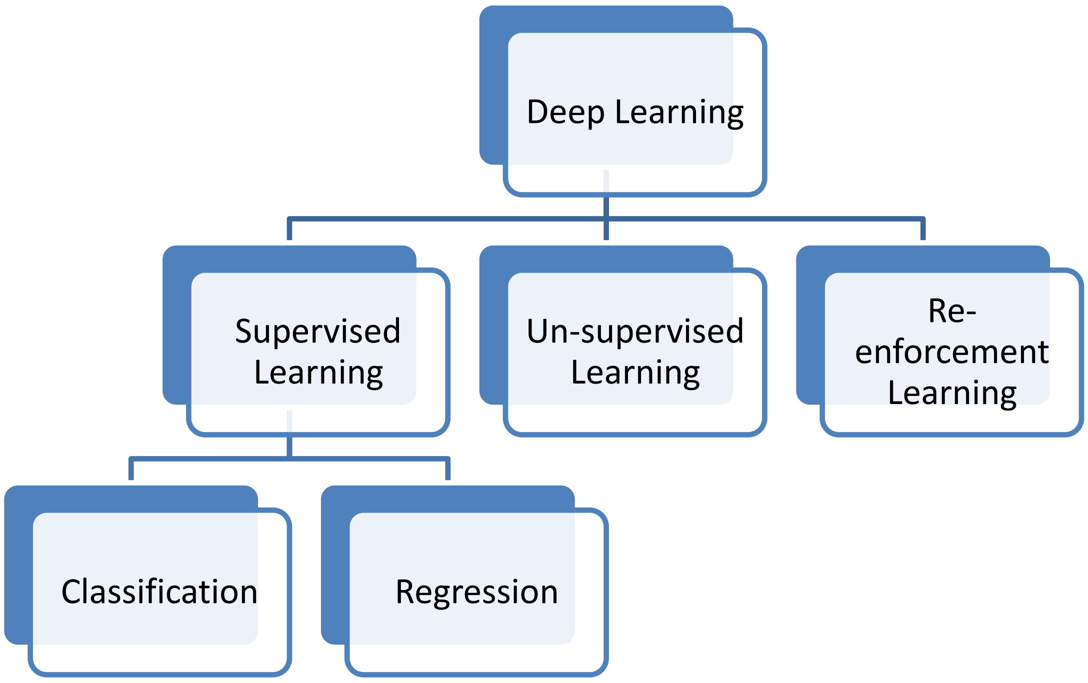


Figure 2: Deep Learning Scenarios

Supervised Learning – a mathematical framework

In this example, one or more predictors have been identified as influencing an outcome. Thus for instance, the factors influencing the price of a house in a given suburb may have been identified to be:

- X_1: Area of the plot
- X_2: Floor area of the house
- X_3: Age of the house
- X_4: Number of bedrooms
- X_5: Number of bathrooms
- X_6: Size of garage
- X_7: Outdoor entertainment area
- X_8: Manicured garden
- X_9: Proximity to public transport
- X_{10}: Affluence of neighbourhood
- X_{11}: Status of local schools
- X_{12}: Median price of homes in the street

  And so on.

When so many variables come into play it becomes extremely difficult to manually articulate a hypothesis which explains to what extent in dollar terms each of these factors have influenced the final price of any given house.

In "Supervised learning", the machine is provided a labelled "training" data set – in this case, the machine is provided with house prices along with each of the attributes or features mentioned for each house.

In mathematical terms, the price y may be represented as a *multiple regression equation*:
$Y = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 \ldots\ldots\ldots + w_ix_i$

The variables x_1, x_2, x_3 etc represent each of the aforementioned attributes and w_1, w_2 etc are regression coefficients that represent the weight or bias of each of these attributes to the final outcome. "w_0"is some constant. Also note that the individual weights may be positive or negative. For example, whilst a larger plot area would positively influence the price, larger age of the house would negatively influence the price.

Based on this labelled data set, the machine is required to derive a set of weights for each of these influencing factors that would explain the final cost figure of the houses in the data set! In essence, the machine is required to model a relationship between multiple explanatory variables and a response variable – in essence, an *objective function.*

In practice, this is done by repeated learning over the provided training data set. The model is fine-tuned for closeness of fit until the *loss function* (difference between predicted and actual value) is a minimum.

Thus, in plain English the model may be re-stated as: *Data = Fit + Residual*

The equation may now be re-written as: $y1 = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 \ldots\ldots + w_ix_i + \dot{\varepsilon}_1$ where $\dot{\varepsilon}$ is the residual.

When the residual $\dot{\varepsilon}$ is zero, the model is a perfect fit.

Based on our general equation, the predicted values for the houses may now be represented as:

$y_1 = w_0 + w_1x_{11} + w_2x_{12} + w_3x_{13} + w_4x_{414} \ldots\ldots + w_nx_{1n} + \dot{\varepsilon}_1$

$y_2 = w_0 + w_1x_{21} + w_2x_{22} + w_3x_{23} + w_4x_{24} \ldots\ldots + w_nx_{2n} + \dot{\varepsilon}_2$

$y_3 = w_0 + w_1x_{31} + w_2x_{32} + w_3x_{33} + w_4x_{34} \ldots\ldots + w_nx_{3n} + \dot{\varepsilon}_3$

$y_4 = w_0 + w_1x_{41} + w_2x_{42} + w_3x_{43} + w_4x_{44} \ldots\ldots + w_ix_{4n} + \dot{\varepsilon}_4$

....

$y_m = w_0 + w_1x_{m1} + w_2x_{m2} + w_3x_{m3} + w_4x_{m4} \ldots\ldots + w_ix_{mn} + \dot{\varepsilon}_m$

In the above set of equations:

- y_1 represents the predicted value of house number 1 , x_{11} represents area of the plot for house number 1, x_{12} represents the floor- area of house number 1 and so on for other attributes
- y_2 represents the predicted value of house number 2 , x_{21} represents area of the plot for house number 2, x_{22} represents the floor- area of house number 2 and so on for other attributes
- y_3 represents the predicted value of house number 3 , x_{31} represents area of the plot for house number 3, x_{32} represents the floor- area of house number 3 and so on for other attributes

In the above notations, we are using two sub-scripts. The first subscript stands for the house number and the second subscript denotes the attribute.

The above series of equations can be represented in matrix notation as follows:

$$y = Xw + e$$

The matrix [X] is the *design matrix* which contains the influencing variables. The vector [W] represents the regression coefficients or "weights".

Y is a vector = $\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ .. \\ .. \\ Y_m \end{bmatrix}$ X is a matrix = $\begin{bmatrix} 1 & X_{11} & X_{12} & X_{13} & .. & .. & X_{1n} \\ 1 & X_{21} & X_{22} & X_{23} & .. & .. & X_{2n} \\ 1 & X_{31} & X_{32} & X_{33} & .. & .. & X_{3n} \\ 1 & .. & .. & .. & .. & .. & .. \\ 1 & X_{m1} & X_{m2} & X_{m3} & .. & .. & X_{mn} \end{bmatrix}$

In the above equations, the very first attribute W_0 which is a constant may be considered as w_0 times 1 – or a coefficient of 1! This explains why the very first column in the [X] matrix is a 1!

W is a vector of coefficients = $\begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \\ W_4 \\ .. \\ .. \\ W_m \end{bmatrix}$ e is also a vector (of residuals) = $\begin{bmatrix} \epsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \varepsilon_4 \\ \varepsilon_5 \\ .. \\ \varepsilon_m \end{bmatrix}$

The objective function in full matrix notation is as follows:

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ .. \\ Y_m \end{bmatrix} = \begin{bmatrix} 1 & X_{11} & X_{12} & X_{13} & .. & .. & X_{1n} \\ 1 & X_{21} & X_{22} & X_{23} & .. & .. & X_{2n} \\ 1 & X_{31} & X_{32} & X_{33} & .. & .. & X_{3n} \\ 1 & .. & .. & .. & .. & .. & .. \\ 1 & X_{m1} & X_{m2} & X_{m3} & .. & .. & X_{mn} \end{bmatrix} * \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \\ W_4 \\ .. \\ .. \\ W_m \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \varepsilon_4 \\ \varepsilon_5 \\ .. \\ \varepsilon_m \end{bmatrix}$$

In summary, the problem is now one of determining the weights i.e. elements of vector W for the best fitting line.

Intuitively, the best fit will be when the sum of differences or residuals is a minimum. Here again, maths comes to the rescue.

Mathematically, the best fit is obtained using *Least Squares*. The squared residual of the sample $\sum_{i=1}^{i=m} \varepsilon_i{}^2$ should be as small as possible.

When this happens, all of the points are as close to the line as possible. In some cases, the difference may be positive i.e. predicted value is greater than actual value. In other cases when predicted value is less than actual value, the difference would be negative. In this example, a prediction of plus $50,000 is just as incorrect as a prediction of minus $50,000! To prevent a positive error in prediction for one set of variables off-setting a negative error in prediction for another set of variables, we are considering the square of the residuals which is always positive. Also, by doing this, many small deviations would be preferred over a single larger deviation!

To summarize, we are now seeking to minimise$\sum_{i=1}^{i=m} \varepsilon_i{}^2$. Again, this may be stated in matrix notation as a product of a single row and a single column vector.

(1) $$[\varepsilon_1 \quad \varepsilon_2 \quad \varepsilon_3 \quad \cdot\cdot_{\cdot\cdot} \quad \varepsilon_m] * \begin{bmatrix} \epsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \varepsilon_4 \\ \varepsilon_5 \\ \cdot\cdot \\ \varepsilon_m \end{bmatrix}$$

Note that one vector is a transpose of the other and therefore the product of vectors E^TE (where E^T is the transpose of vector E).

Now by definition, the residual is the difference between actual and predicted value. Thus,

Substituting in (1) $$\varepsilon = y – wX$$

So in effect we are minimising the square:

(2) $(y – wX)^T(y – wX)$

(3) $y^Ty – y^TwX – w^TX^Ty + w^TX^TwX$

Now, in matrix computation $y^TwX = yw^TX^T$

(4) $y^Ty – 2w^TX^Ty + w^TX^TwX$

To minimize the above, by calculus we need to differentiate the above expression with respect to the **vector w** (& its transpose w^T**)** and set derivative to 0.

This gives us the following:

(5) $-2X^Ty + 2X^TXw = 0$

(6) $X^TXw = X^Ty$

Multiplying both sides by $(X^TX)^{-1}$

(7) $w = (X^TX)^{-1} X^Ty$

So there we have it. Given a matrix of features X and a corresponding vector y, we are able to compute a vector of weights w whereby the residual will be minimised, thus providing line of best fit.

The key takeaway here is: ***<u>Tensor (matrix) operations are at the heart of machine learning! ALL inputs and targets have to be in the form of tensors (matrices)</u>***

This was an example of linear regression. What of nonlinear relationships? Well, the central idea is the same as linear regression just with non-linear features! For example:

$$\varnothing(x_i) = \begin{bmatrix} x_i^2 \\ x_i \\ 1 \end{bmatrix}$$

Unlike traditional programming, AI & ML are more deeply entrenched in classical mathematics like vector algebra, tensor operations, calculus and, statistics. In an effort to democratise deep learning, it is often mentioned that this understanding is not essential. The Keras libraries for example, provide the necessary in-built functions. However, I do feel some time and effort must be invested in brushing up mathematics. Else you are left with too many black boxes to contend with.

Supervised learning has two sub-scenarios:

- **Classification Algorithms**: The model is a predictor for a discrete outcome .This includes both binary and multiclass or multinomial classification. For example a tumour is benign or not benign (binary classification). The picture is of a dog or a cat, or recognising handwritten digits as one of 0 to 9 (multinomial classification).
- **Regression Algorithms**: Provide a continuous output based on one or more inputs. For example, the price of different houses can take different values based on various parameters. In this case we are not predicting a yes/no answer. Rather we are estimating along a continuous value curve.

Un-supervised Learning

In this scenario, no predictors have been identified up front. Instead, the computer seeks to find a "pattern" in the data set using clustering algorithms. It essentially determines a separation plane.

Reinforcement learning

In this scenario, as opposed to minimising the loss function, the emphasis is on maximising a "rewards" function with sparse feed-back.

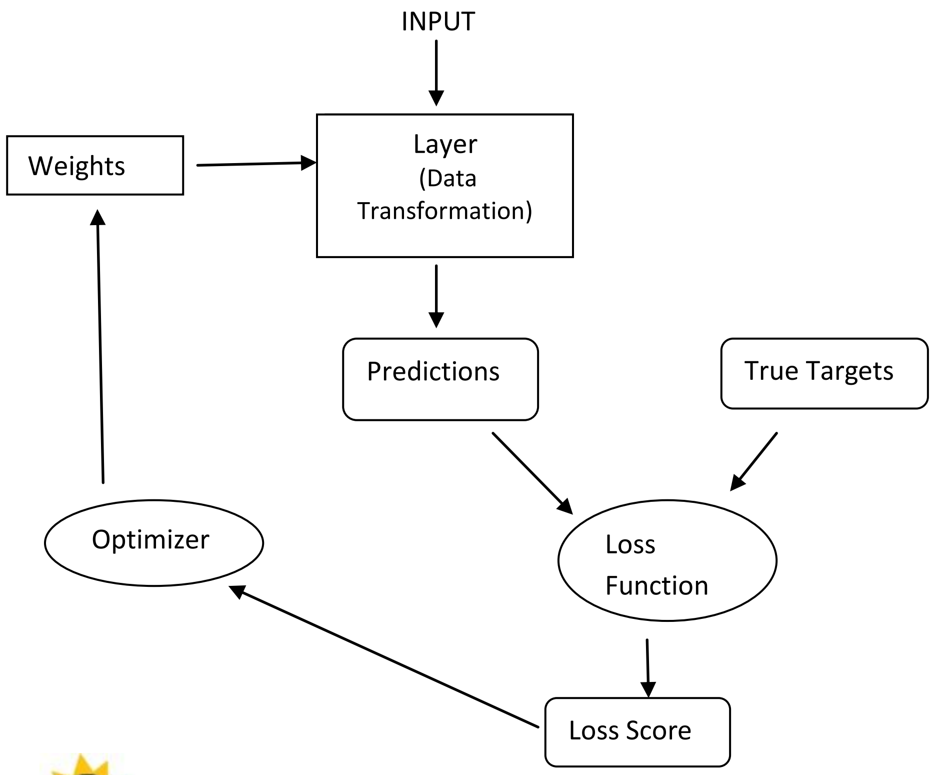
The Mathematics of Artificial Intelligence

"Pure mathematics is, in its way, the poetry of logical ideas" – Sir Albert Einstein

Algorithms are the building blocks of artificial neural networks. Essentially mathematical & statistical in nature, these seek to identify a relationship between an outcome and recognizable features in an input data set. You can think of them as a clustering and classification mechanism. The example on predicting house prices was a merely to illustrate the following:

- Data is the life-blood of Machine Learning. **All inputs and targets have to be in the form of tensors.**
- Algorithms help us glean patterns in this data set , extract generalisations and extrapolate the findings

Whilst artificial Intelligence obviously stretches far beyond simple linear regression models, the general process is one of modelling data, measuring closeness of fit and refining the model. Algorithms play a huge part in this process. In his book *Deep Learning with Python* Francois Chollet has neatly illustrated this process: **Figure 3: Artificial Neural Networks**



In Deep Learning architecture, there would be multiple Data transformation layers each layer comprising of a number of artificial neurons.

Data as Tensors

Now the data itself is not just numbers as in our house price example. It could well be static or moving images for character and image recognition; sound for voice recognition and commands; words, tweets and social media comments etc! ***As mentioned earlier, data has to be represented in the form of tensors***!

Thus a data set of images would be represented as a 4D tensor: *samples, height, width, and channel.* For grey scale images there is a single channel whilst for colour images there will be three channels: Red, Green, and Blue (RGB).

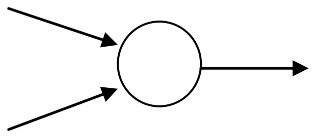
Videos or moving images will be represented as a 5D tensor with *frame* being another additional dimension.

Activation Functions

Deep learning is the "stacking" of multiple layers of neurons to form a neural network.

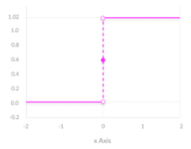
Data in the form of a tensor of features and a tensor of weights is fed into the outer layer. Each of the artificial neurons in the layer then uses a weight and a bias to transform this input to an output which is fed into the next layer of neurons.

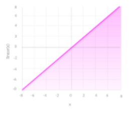
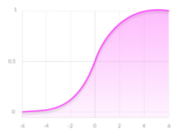
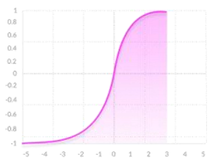
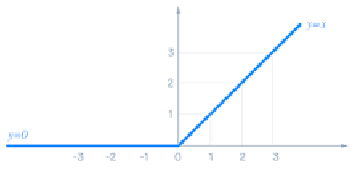
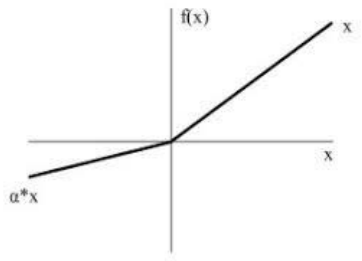
$y = \sum_{i=1}^{i=n} w * xi + \mathrm{B}$



In the outermost layer, the role of the activation function is to determine if a specific neuron should be activated ("fired"). Thus for instance if the output of the final layer is to determine if a tumour is benign or malignant, we would need two neurons to represent each of the two states, one of the neurons to be fired. If on the other hand we need to classify a handwritten digit, we would need 10 neurons to represent digits 0 to 9!

The following table describes the most common activation functions:

No.	Name	Type	Use Case & Explanation
1	Step Function 	Binary classification	Simple threshold activation function: $f(x) = 1\,; if\ x \geq 0$ $f(x) = 0; if\ x < 0$ Since it has just two outputs it is appropriate for binary classification only

2	Linear Activation 	Regression	This is of the form: $A = cx$ This creates an output proportional to the input. So the output is a value - not just 0 or 1 as in the case of the step function above.
3	Sigmoid Function 	Classification	Function has an 'S' shaped graph where output values are between 0 and 1. This is represented by the equation $f(x) = 1/(1 + e^{-x})$.
4	Tanh Function 	Classification	Very similar to the Sigmoid function but is zero centred giving a value range from -1 to +1. It thus helps model strongly negative, neutral and strongly positive input values. Typically used in intermediate layers of a neural network as it helps in centring the data.
5	RELU (*Rectified linear unit)* 	Classification with multiple classes	Most popular & actively used with a value range from 0 to Infinity. Positive values are passed through as-is and negative values are passed as zero. $$f(x) = \max(0, x)$$ However because the function passes a zero for negative values, the network cannot 'learn" when input values are zero or negative (*Dying RELU)*
6	Leaky RELU 		This is a modified RELU. Instead of returning a 0 for negative values, it returns a small linear component. $$f(x) = ax;\ for\ x < 0$$ $$f(x) = x\ ;\ for\ x > 0$$
7	SOFTMAX	Classification with multiple classes	Provides a probability for each class with the total across all classes adding to 1.The target class would have highest probability

*The above table is **not** an exhaustive list. It is at best a starting point for better understanding. Activation functions are an ongoing subject of research and study. Selecting an appropriate activation function is as much an art as a science. RELU is the most popular and used by default whilst Sigmoid and Tanh are natural classifiers. Then there is SOFTMAX that provides a probability across classes and* *activates the one with the highest probability. Often times underlying characteristics of the data dictate the type of activation function to be used.*

Loss/Cost Functions

As we discussed in figure 3, going hand-in-hand with modelling the data is determining closeness of fit. This is measured via a *loss function.* This tells us how far off our model's predictions are from the actual results of a labelled data set. In our example on house price predictions, we used the method of *Least Mean Square Error.* Some of the more common *loss functions* are mentioned below:

No.	Name	Keras Call	Use Case & Explanation
1	Mean Square Error (MSE)	loss='mean_squared_error'	Typically used in regression models
2	Mean Squared Logarithmic Loss	loss='mean_squared_logarithmic_error'	As above. However, is more tolerant of larger losses for larger target values whereas MSE punishes large offsets heavily without considering proportion to the target value
3	Mean Absolute Error Loss	loss='mean_absolute_error'	As above. It is more robust to outliers. It takes the average of the absolute difference between predicted and target values.
4	Multi Class Cross Entropy Loss	loss='categorical_crossentropy'	Used for multi-class classification problems. It calculates a score that summarizes the average difference between the actual and predicted probability distributions for all classes in the problem. The score is minimized and a perfect cross-entropy value is 0.

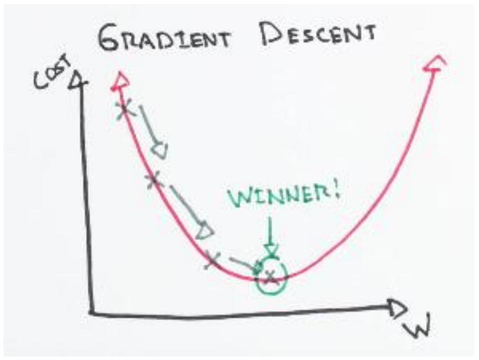
*The above table is **not** an exhaustive list. It is at best a starting point for better understanding*

Optimizer Functions

So, we now have looked at algorithms and activation functions to model a data set and *loss functions* to measure how well this model fits the data or how wrong the predictions are. The model now needs to be iteratively refined for a better fit i.e. we need to reduce the gap (or loss) between predicted and actual values. During this "training" process, the parameters (weights) of the model are continually tweaked to minimize the loss. The process of minimising the loss is achieved via optimizer functions.

To understand how the process of optimization works, let us consider a simple linear model $y_i = mx_i + c$. By initially selecting arbitrary values for *m* and *c* it is unlikely the model is a good fit. As we discussed above, the mean squared error (MSE) for the various predictions is a measure of closeness of fit. This could be denoted by the equation: E = $1/N(\sum(y_i - y'_i)^2$

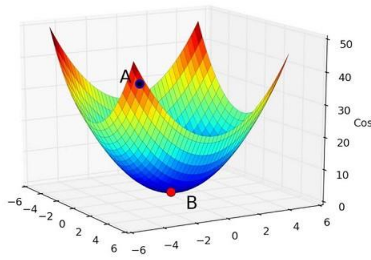
*W*here *N* is the sample size, y_i is the actual value and y'_i is the predicted value.



Such an equation is represented by a parabola. Intuitively, the error is least at the bottom of the parabola where the error is close to zero! The error is a maximum at the top of the parabola. A positive error would be right of centre and a negative error would be left of centre!

Clearly, to minimize the error, we are traversing down the ***slope*** of this parabola in an effort to reach the bottom where the error is least!

In mathematical terms, slope equates to gradient! Intuitively, the higher you are, greater is the slope! So we need to move in the direction where the slope or gradient is decreasing. This encapsulates the principle of gradient descent.



The above representation was for a single variable. In case of multiple variables, the parabola would be a surface similar to a bowl as shown alongside.

Consider the equation for *Mean Squared Error* as follows:

E = $1/N(\sum(y_i - y'_i)^2$

Let us assume the predicted value y'_i models a linear equation$(w_0 + w_1x_i)$. Substituting in the above:

E = $1/N(\sum(y_i - (w_0 + w_1x_i))^2$

Expanding the above,

E = $1/N(\sum(y_i{}^2 - 2\,y_i(w_0 + w_1x_i) + (w_0 + w_1x_i)^2)$

E =$1/N(\sum y_i^2 - 2y_iw_0 - 2y_iw_1x_i + w_0^2 + 2w_0w_1\,x_1 + w_1^2\,x_i^2)$

In calculus, the gradient of a function with multiple variables is represented by a vector of partial derivatives. So let us consider the derivatives with respect to w_0 and w_1.

$dw0 =$ 1/N $\sum(-2y_i + 2w_0 + 2w_1\ x_i\ )$
$\boldsymbol{dw0 =}$ **1/N** $\boldsymbol{(\sum 2(w_0 +\ w_1 x_i -\ y_i)}$

$dw1 = 1/N(\sum -2y_i x_1 + 2w_0 x_i + 2w_1 x_i^2\ )$
$\boldsymbol{dw1 = 1/N(\sum 2\ x_i\ (w_0 + w_1 x_i - y_i)}$

Thus the Gradient descent implementation provides a feedback to the algorithm as a vector of adjustments so that it adjusts the weights ever so slightly and thereby minimises the loss.

The weights w_0 and w_1 would be adjusted as:

$$\begin{bmatrix} w_0 \\ w_1 \end{bmatrix} := \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} - \eta \begin{bmatrix} 2(w_0 +\ w_1 x_i -\ y_i) \\ 2x_i\ (w_0\ +\ w_1 x_i\ -\ y_i) \end{bmatrix}$$

Here η stands for the *"learning rate".*

Gradient Descent is the father of all optimizers.

Now imagine a very large dataset for example those handled by Google and social media platforms like Face book, Twitter etc. These would contain multi-billions of elements. Performing Gradient Descent computations on such large batch sizes would be very time consuming. This brings us to the concept of *Stochastic Gradient Descent* wherein examples are chosen at random (stochastically) to compute the gradient. Given enough iterations, the randomness gives a very good approximation at a fraction of computing cost.

The following points need to be considered:

1. *Changing our weights too fast i.e. taking large steps could cause us to miss the minimum point!*
2. *Changing weights too slowly i.e. taking too many small steps could make for very long learning cycles*
3. *Getting stuck in 'local minima' – when dealing with high volume data sets it is possible that we find an area where it appears the loss function has been minimised however, it's really just a local minimum. In hiking analogy we have reached a small valley but not the bottom!*
4. *The concept of 'Learning rate' encapsulates the above.*

Some of the more popular optimizers have been listed below:

No.	Name	Keras Call	Use Case & Explanation
1	Stochastic Gradient Descent	SGD	One of the early optimizers that uses randomly selected samples. Intuitively it is more efficient to use a subset of training samples in batches rather than all of the data. Most suitable for objective functions that are differentiable i.e. logistic regression. It works out the path to be

			taken by the algorithm to reach global minima.
2	Adaptive Gradient Algorithm	Adagrad	This provides for feature specific learning rates useful when dealing with sparse datasets. In principle, learning rates are scaled in proportion to the frequency of the features: decreases faster for frequent parameters.
3	Root Mean Square Propagation	RMSProp	Developed by Prof. Geoffrey Hinton, this uses the moving average of the squared gradients to normalize the gradient itself.
4	Adaptive Moment Estimation	Adam	It improves on Adagrad by taking the exponential moving average of the gradient and squared gradient. Currently this is a very popular optimizer.

Where to From Here

Each of the concepts and algorithms touched upon in this blog can be discussed in much greater detail, especially the underlying maths. The approach in this paper was to highlight key concepts – without making it too esoteric and at the same time not dumbing down the maths too much. Hopefully this was a happy medium. In future blogs, we will examine some implementations of Machine Learning algorithms and deep learning neural network architecture using Python and Keras libraries. Stay tuned and happy travels.