Deep Learning and Neural Networks

Üzeyir Saçıkay[1], Bengier Ülgen Kılıç[2]

Abstract
Nowadays, it is quite difficult to go through a day and not interact with machine learning: face recognition on our phones, smart search algorithms, travel time estimation, product recommendations and many more. It turns out that machine learning is a purely mathematical concept that lies at the intersection of linear algebra, real analysis and statistics. We explored the mathematics behind neural networks.

[1] *M.Sc. Theoretical and Mathematical Physics, Ludwig Maximilian University, Munich, Germany*
[2] *Department of Mathematics University at Buffalo, SUNY*

Contents

Introduction

In the general sense, machine learning is the study of methods that can 'learn' by themselves. Although impressive and efficient, pre-deep learning era machine learning applications are a bit "meh" when considered as programs that can 'learn'. At most, they can find linear or a bit more complex (by the use of radial basis functions) decision boundaries that can best separate certain separable sets. But they feel too analytic and hand-designed to be seen as having 'learned'. They are too task specific. Which is kind of why as the amount of training data is increased, traditional machine learning algorithms only perform slightly better. The essence of the methods of deep learning is that we can catch much general patterns therefore 'learn' better and make efficient use of data.

We will be delving into the mathematics behind the deep learning and introduce the motivations behind the concepts. The most important concept is the neural networks which is the structure we use to perform 'learning'.

1. History of Neural Networks

Perceptron The first breakthrough in the history of neural networks was in 1943, when McCulloch and Pitts invented the "perceptron". Perceptron is a simple binary decision maker with binary inputs. We have $x_i \in \{0,1\}$ and $w_i \in \mathbb{R}$

$$y = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq \text{threshold} \\ 1 & \text{if } \sum_i w_i x_i > \text{threshold} \end{cases} \tag{1}$$

Here, x_i and w_i are said to inputs and weights. The 'threshold' is also a real number, we named it as it is for brevity. Perceptron can be depicted as: Now, as can be seen from (1), each

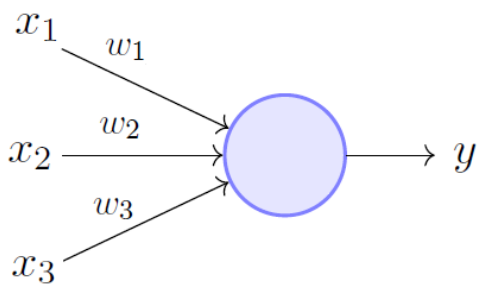


Figure 1. A perceptron with 3 inputs

input is assigned a weight and the decision is made based on whether the weighted average of inputs achieves a certain threshold or not. This can be seen as a simple model of the biological neuron: synapses of a biological neuron take inputs in the form of electrical signals, called 'impulse'. The weights in this case can be interpreted as how frequent that synapse is[1]. Then, the neuron fires if the activation threshold is achieved.[1]

[1] Neuroscientists reveal how the brain can enhance connections

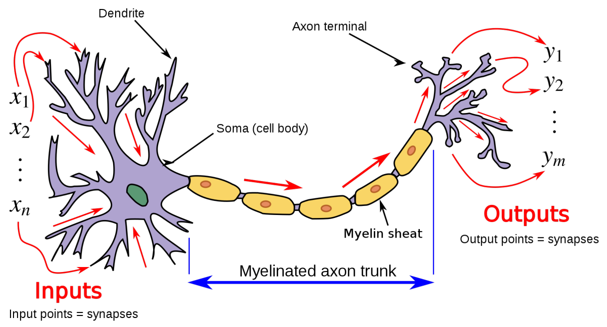


Figure 2. Biological neuron

1.1 Network of perceptrons

A single perceptron on its own can only do so much. To make things more complex, we can connect multiple perceptrons end to end to get a network structure. The biggest advantage of doing so is that we are dividing the task at hand into smaller subtasks that can be performed by layers of our networks. So each layer would perform the respective subtask and transfer its outcome as input to the next layer which would then make use of that outcome to perform its own subtask. This way, we can make subtle and complex decisions accounting for various factors.

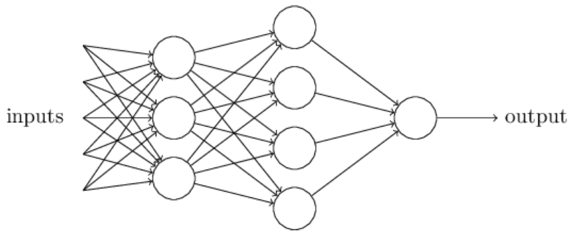


Figure 3. Network of perceptrons, [1]

There is one quite hard to detect problem with perceptrons. Say we have designed such a network in order to perform a specific task but we are not quite there in terms of the overall outcome. What we could then do is to tune the weights to get the desired effect but as we will soon discover, that would be impossible to do with perceptrons. As perceptrons are discrete, any tuning we might perform on one of the weights might lead to significant changes for the rest of the network. If our tuning were to flip the result of the perceptron that the tuned weight belongs to, then all the perceptrons in the next layer will have an extra term as they will now be inputting our flipped perceptron too. Then some of the perceptrons in the next layer might flip too and we can argue similarly for the upcoming layers. So, any tuning might lead to an uncontrollable ”butterfly effect” that would cascade throughout the network and give rise to unwanted outcomes. Therefore, in general, it would be impossible to tune a network of perceptrons.

1.2 Discreteness problem

As mentioned, the problem of tuning arises due to the discrete nature of perceptrons. The jump from 0 to 1 after a certain threshold is quite sharp. What we need is a continous relationship between the output and the weights. So, to deal with this, we remove the threshold condition altogether and incorporate it as a constant term in the form of ’bias’:

$$y = \sum_i w_i x_i + b = w^T x + b$$

where $b = -$threshold. Here, we also simplified the notation by introducing:

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}, \qquad x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

This solves the problem of discreteness as now our outputs are continuous functions of weights and inputs. But this modification comes with the problem of its own: this modified perceptrons are now affine functions and if we make a network out of them, we would get an affine function in the end as composition of affine functions are still affine. This is a problem because having a network would not have any benefits over a single perceptron as we can instead use a single perceptron representing the overall affine behaviour of our network.

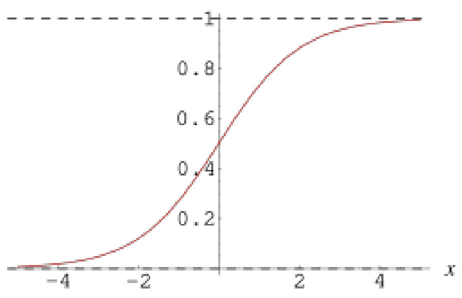
1.3 Activation function

We need break free of the linear behaviour. We do this by introducing non-linearity to our perceptrons:

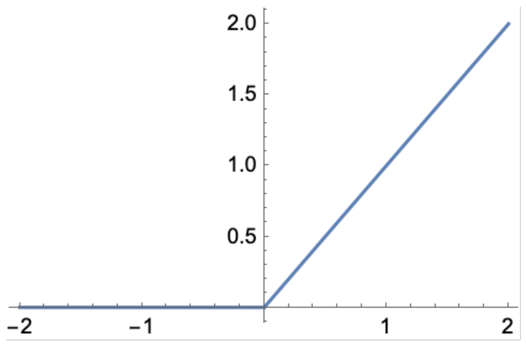
$$y = \sigma\left(w^T x + b\right)$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is non-linear. σ is usually called the activation function. The most commonly used activation functions are :

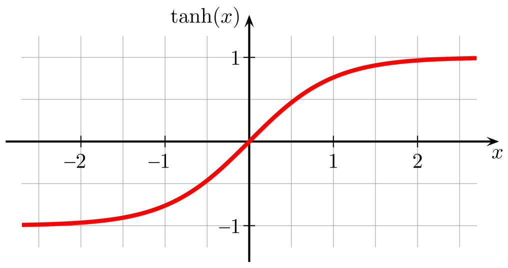
- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$



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



- tanh: $\sigma(x) = \tanh(x)$



After the addition of this activation function, we arrive at the modern unit in a neural network called 'neuron'. One important aspect of this added non-linearity is that, now we have a powerful existance theorem stating that:

Theorem 1 (Universal Approximation Theorem): $\mathcal{NN}^d_{\sigma,L}$ *is dense in* $\mathcal{C}(K,\mathbb{R})$ *for* $L \geq 1$.

Here $\mathcal{NN}^d_{\sigma,L} \subset \{F : \mathbb{R}^d \to \mathbb{R}\}$ is the class of neural networks with d inputs, L hidden layers and σ activation function. And $K \subset \mathbb{R}^d$ is compact. Note that there are some mild conditions on σ such as locally boundedness, almost everywhere continuity and almost everywhere non-polynomiality.

This theorem tells us using neural networks we can arbitrarily approximate any continous function on a compact Euclidean domain. This gives us a glimpse of how powerful neural networks can be!

2. Forward propagation

Now that we laid out the structure of a neural network, we can move on to it's inner workings. The first step is to define the forward flow of information which is usually called the feedforward, forward propagation or forward pass. As before, output of each neuron is fed into all the neurons in the next layer as input. In order to express these in a compact form, we use the following notation:

n_l : number of neurons in the l^{th} layer,

$w^{[l]}_{jk}$: weight from k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer,

$z^{[l]}_j$: linear output of the j^{th} neuron in the l^{th} layer,

$a^{[l]}_j$: non-linear output or activation of the j^{th} neuron in the l^{th} layer.

Then, we have

$$\begin{aligned} a^{[l]}_j &= \sigma\left(z^{[l]}_j\right) \\ &= \sigma\left(\left[\sum_{k=1}^{n_{l-1}} w^{[l]}_{jk} a^{[l-1]}_k\right] + b^{[l]}_j\right) \end{aligned}$$

Now, letting

$$W^{[l]} = \begin{bmatrix} w^{[l]}_{11} & \cdots & w^{[l]}_{1n_{l-1}} \\ \vdots & \ddots & \vdots \\ w^{[l]}_{n_l 1} & \cdots & w^{[l]}_{n_l n_{l-1}} \end{bmatrix},$$

$$A^{[l-1]} = \begin{bmatrix} a^{[l-1]}_1 \\ \vdots \\ a^{[l-1]}_{n_{l-1}} \end{bmatrix}, \qquad b^{[l]} = \begin{bmatrix} b^{[l]}_1 \\ \vdots \\ b^{[l]}_{n_l} \end{bmatrix}$$

and denoting the number of layers by L, we get

$$\begin{aligned} &A^{[0]} = x, \\ &A^{[l]} = \sigma\left(W^{[l]} A^{[l-1]} + b^l\right), \quad 1 \leq l \leq L. \end{aligned}$$

Following this iterative process, we get the final output as $A^{[L]}$. Another way of representing this final output is as a function of inputs, weights and bias:

$$A^{[L]} = a^L(x, W, b)$$

Note that since it is composed of continuous functions, a^L is continuous.

3. Cost function

We need a way of measuring the performance of our neural network in terms of how closer it is to the expected behaviour. Denoting the expected results as $y(x)$, what we need is a measure of distance between y and a^L.

For an individual training example x, we call that metric the loss function, usually denoted as $J(x, W, b)$. Then the cost is the average of loss over all training examples, usually denoted $C(W, b)$. There are many viable cost functions each best suited for certain applications, a list of such functions and their applications can be found here.

Now as cost gives us a distance between the expected and real behaviour of our network, we want to minimize the cost.

We can think of the cost function as a surface over the domain of weights and biases, i.e., $C : \mathbb{R}^N \to \mathbb{R}$ where N is the number of all weights and biases. N is usually quite large as we have

$$|W| = \sum_{l=1}^{L} |W^{[l]}| = \sum_{l=1}^{L} n_l n_{l-1}$$
$$|b| = \sum_{l=1}^{L} |b^{[l]}| = \sum_{l=1}^{L} n_l$$
$$\implies N = |W| + |b| = \sum_{l=1}^{L} n_l(n_{l-1}+1)$$

So, our aim is to find the global minimum of this surface in $\mathbb{R}^N$. This is done automatically using the backpropagation algorithm, as explained in the following section.

4. Backpropagation

Say we are located at a certain point of this surface and we want to get to the minimum. What we can do is use the gradient of cost at the point we are located to determine the direction of steepest ascent and then move in the opposite direction as that is the direction of steepest descent. This process would land us at a local minimum nearby our starting point. This is called the gradient descent and we can express it more compactly as

$$w^{[l]}_{jk,\,t+1} = w^{[l]}_{jk,\,t} - \eta \nabla_{w^{[l]}_{jk,\,t}} C$$
$$b^{[l]}_{j,\,t+1} = b^{[l]}_{j,\,t} - \eta \nabla_{b^{[l]}_{j,\,t}} C$$

where the t subscript denotes the step number.

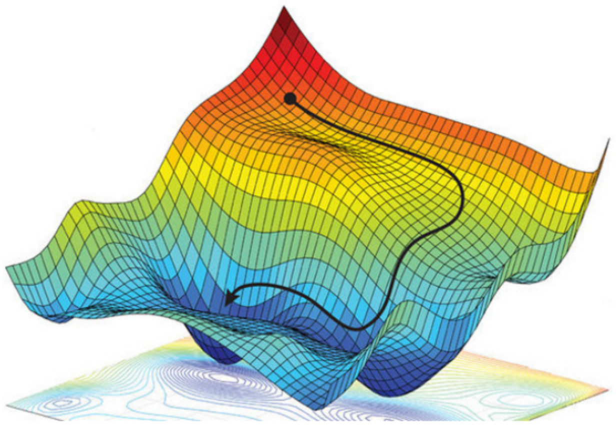


Figure 4. Gradient descent, [2]

To calculate the gradient of a specific weight, we need to note that a small change in this weight will affect all the activations in the next layers. Therefore we can find this derivative in a backward fashion, starting from the last later and finding how they affect C, we can build towards the layer at hand. Put shortly, since all our functions are made up of compositions, derivative at a certain node is dependent on the ones that come after it. This is essentially the backpropagation algorithm.

5. Training and Test

We have a mechanism that can automatically find a nearby local minimum of the cost for a given starting point. This starting point is usually taken to be random, called the ”random initialization”. So we can initialize a few times to avoid noisy results. But how can we decide between the local minima? Well it turns out some minima are better than others!

The choice if minimum depends on the generalization performance as we want to create a model that can generalize well from a sample. We measure this generalization performance by training our model on a dataset, called the training set and compare it’s performance on a dataset that was not seen before, called the test set.

Now, suppose we are training a model on a dataset that consists of 2 different classes. We want our model to be able to differentiate between these classes. Consider the following scenarios:

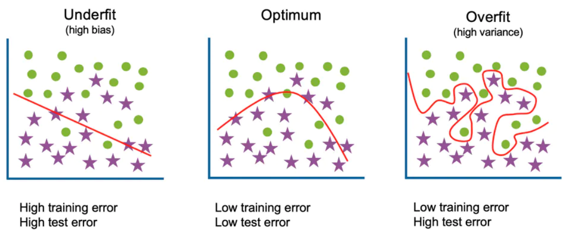


Figure 5. Source: IBM

- Underfit: This model is too simple. It would generalize quite poorly as it ignores the complexity of the underlying pattern.

- Overfit: This model is too complicated and it seems to be more concerned with the specific noise of training data. It would perform poorly on a dataset that has a similar distribution.

- Optimum: This model seems to be capturing the general pattern and it would generalize well.

We want to choose a minimum that would give the optimum behaviour as listed above. This amounts to good performance on both training and test.

6. Regularization

Regularization is the general name used for the methods of reducing overfitting. There are two kinds of regularization methods, implicit and explicit.

6.1 Explicit regularization

Explicit regularization refers to modifying the cost function to avoid certain minima. The most commonly used explicit regularization methods are L_1 and L_2. Both work under the empirical assumption that minima with smaller weights would generalize better.

6.1.1 L_1 regularization (LASSO)

In this method, we add the L_1 norm of weights to the cost function

$$\sum_{l=1} \left\| W^{[l]} \right\|_1 = \sum_{l=1} \sum_{j,k} W_{jk}^{[l]}$$

In effect, L_1 regularization tends to shrink the set of weights by converging to a point in which most of the weights but some are zero. So, L_1 regularization can be used to get a more scarce model.

6.1.2 L_2 regularization (Weight decay)

In this method, we add the L_2 norm of weights to the cost function

$$\left\| W^{[l]} \right\|_2 = \sum_{l=1} \sum_{j,k} {W_{jk}^{[l]}}^2$$

In effect, L_2 regularization tends to reduce the magnitude of all weights and result in a model with small weights.

6.2 Implicit regularization

This is the set of methods that implicitly affect the generalization performance of the overall network. Examples of implicit regularization methods are increasing the training set size and dropout.

References

[1] Michael A. Nielsen. *"Neural Networks and Deep Learning"*. Determination Press, 2015.

[2] Alexander Amini, Ava Soleimany, Sertac Karaman, and Daniela Rus. Spatial uncertainty sampling for end-to-end control. *CoRR*, abs/1805.04829, 2018.