Applied Deep Learning - Reviewed Notes

Dom Hutchinson

November 7, 2020

Contents

1 Introduction

Definition 1.1 - *Deep Representation Learning*
Representation Learning is a set of techniques in machine learning where a system can automatically learn representations needed for feature detection from the raw data without the need for hand-designed feature descriptions. *Deep Representation Learning* is then learning to classify using this feature detection.

Remark 1.1 - *Biological Inspiration*
In the natural world *Neurons* are the basic working units of the brain. *Neurons* can be split into three main areas

i). *Dendrites* - Receives inputs from other neurons.

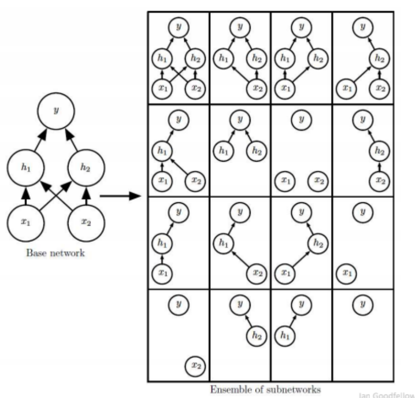
ii). *Axon* - Carries information.

iii). *Axon Terminals & Synapses* = Send information to other neurons.

Artificial Neural Networks seek to mimic this structure.

Definition 1.2 - *Neuro-Plasticity*
Neuro-Plasticity is the ability of a neural system to adapt its structure to accommodate new information (i.e. Learn). This can take several forms including growth & function changes.

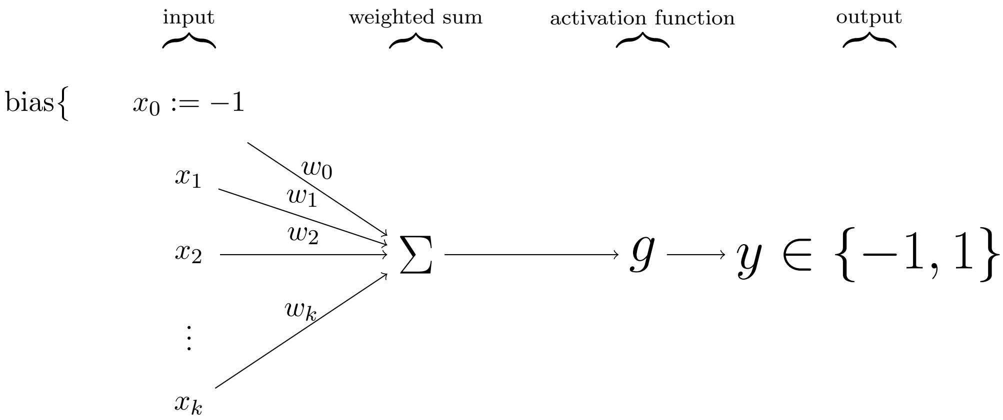
Proposition 1.1 - *Neural Networks as an Ensemble of Sub-Networks*
A *Neural Network* can be considered to represent many *sub-networks*. These sub-networks are switched between depending on which components are picked and how they are defined.



2 Structures

2.1 Perceptron

Definition 2.1 - *Perceptron*
A *Perceptron* is an algorithm for supervised learning of a binary classifier. A *Perceptron* defines a hyperplane which acts as a decision boundary which linearly separates the input-state space. These two regions correspond to the two-classes. A perceptron has the following structure.



x_0 is the bias element. It is always set to -1 in the input and the actual value is defined by its weight w_0.

$\boldsymbol{x} = (x_0, \dots, x_k)$ is the input. Note that x_0 is the bias and $(x_1, \dots, x_k)$ are the observed inputs.

$\boldsymbol{w} = (w_0, \dots, w_k)$ is the weights assigned to each input (including the bias).

Σ is the weighted sum of the bias & inputs. $\Sigma := \sum_{i=0}^{k} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}$

g is the *Activation Function* which maps the result from the weighted sum Σ to $\{-1, 1\}$, performing a binary classification.

y is the output of the *Activation function.* (i.e. the classification). Typically denoted as $f(\boldsymbol{x}; \boldsymbol{w})$

$$y = g\left(\sum_{i=0}^{k} x_i w_i)\right) = g(\boldsymbol{w}^T \boldsymbol{x})$$

Remark 2.1 - *Limitations of Perceptron*
A *Perceptron* can only perform linear binary classification so is not useful when two classes are not linearly separable.

Proposition 2.1 - *Perceptron Supervised Learning Rule*
To make a perceptron learn from misclassifications, we adjust the weight vector $\mathbf{w}$. One learning rule for this is to update the current weights by a certain proportion of the error made $\Delta\mathbf{w}$.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \Delta\boldsymbol{w} \quad \text{where} \quad \Delta\boldsymbol{w} = \begin{cases} \eta f^*(\boldsymbol{x})\boldsymbol{x} & \text{if} \quad \overbrace{f^*(\boldsymbol{x})}^{\text{ground truth}} \neq \overbrace{f(\boldsymbol{x})}^{\text{prediction}} \\ 0 & \text{otherwise} \end{cases}$$

where $\eta \in \mathbb{R}^+$ is the *Learning Rate.* Remember that $f^*(\cdot) \in \{1, -1\}$.

Remark 2.2 - *Learning Rate* η
The *Learning Rate* ν defines how big the steps learning rule makes towards a minimum. The *Learning Rate* needs to be tuned as too small a value will mean it takes a long time (and a lot of data) to reach a minimum, whereas too great a value means the minimum can be overstepped and convergence is unlikely.

Proposition 2.2 - *Training Process for a Single-Layer Perceptron*
Let $\{(\boldsymbol{x}_1, f^*(\boldsymbol{x}_1)), \dots, (\boldsymbol{x}_N, f^*(\boldsymbol{x}_N))\}$ be a set of training data.

The following is an algorithm for learning a good set of weights $\boldsymbol{w}$ for a *Perceptron*

i). Initialise the weight vector $\boldsymbol{w} = \mathbf{0}$

ii). Consider next training datum $(\boldsymbol{x}_i, f^*(\boldsymbol{x}_i))$.

iii). Calculate prediction $f(\boldsymbol{x}) := g(\mathbf{w}^T\mathbf{x})$.

iv). Compare prediction $f(\boldsymbol{x})$ and ground truth $f^*(\boldsymbol{x})$.

v). Update the weight vector $\boldsymbol{w} = \boldsymbol{w} + \Delta\boldsymbol{w}$ where $\Delta\boldsymbol{w} = \begin{cases} \eta f^*(\boldsymbol{x})\boldsymbol{x} & \text{if } f^*(\boldsymbol{x}) \neq f(\boldsymbol{x}) \\ 0 & \text{otherwise} \end{cases}$

vi). Repeat ii)-v) until the training set is exhausted.

Remark 2.3 - *Arbitrary Decision Boundaries*
Multiple preceptrons can be connected to form networks which are able to define arbitrary decision boundaries, rather than just a linear boundary. See `Section 4` for details on these network architectures.

2.1.1 Activation Functions

Proposition 2.3 - *Activation Function*
There are several choice for the *Activation Function* including:

- The *Sign* activation function binarily assigns values depend on whether they are positive or negative. *Sign* is not differentiable.

$$g_{sign}(x) := \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases} = \frac{x}{|x|}$$

- The *Rectifying Linear Unit* (ReLU) function is a differentiable activation function.

$$\begin{aligned} g_{ReLU}(s) &:= \max\{0, s\} \\ g'_{ReLU}(s) &:= \begin{cases} 1 & \text{if } s \geq 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Remark 2.4 - *Limitation of ReLU*
Using *ReLU* introduces a problem of *Dying Neurons* where a large gradient flowing through *ReLU* may force the neuron never to activate again (as it pushes the incoming signal to 0). This is bad, as these neurons will no longer contribute to learning anymore.

2.2 Artificial Neural Network

Definition 2.2 - *Artificial Neural Network*
An *Artificial Neural Network* is a graph which loosely models the neural network of a brain and is used for classification/regression.

An *Artificial Neural Network* is structured as a set of layers with each layer only takes input from the layer immediately before, and passes its signal to the layer immediately after. There are three groups of layers:

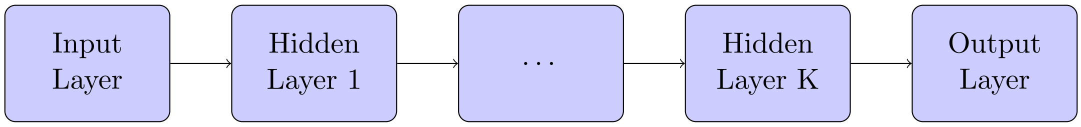


Figure 1: Abstract Diagram of the Structure of an Artifical Neural Network with K hidden layers.

i). Input Layer. Observed values.

ii). Hidden Layers. A collection of perceptrons which are grouped into layers. How the layers pass their output/take an input is defined by the architecture of the neural network

iii). Output Layer. The classification of the network. This may be a single node which represents a numerical value being predicted, or a collection of N nodes used to classify to one of N classes.

See `Figure 1` for an diagram of the structure of an ANN.

Definition 2.3 - *Forward Propagation*
Forward Propagation is the process a neural network calculating its output, once it is given a set of inputs.

This is done by inserting the observed values into the correct nodes of the input layer. This values are then passed forwards, throught the network, one hidden layer at a time. The flow of data is unidirectional.

Several values are calculated during *Forward Propagation*

- Signal $s_j^l := (w^l)^T f^{l-1}$. The weighted sum calculated by the perceptron from its inputs, before it is passed to the activation function.
- Output $f_j^l := g_j^l(s_j^l)$. The result from applying the activation function g_l^j to the signal s_j^l.

Here l denotes a layer and j the specific node in that layer.

3 Deep Neural Networks

Definition 3.1 - *Deep Neural Network*
A *Deep Neural Network* is an *Artifical Neural Network* with multiple hidden layers. These multiple hidden layers allow *Deep Neural Networks* to learn complex non-linear relationships as later layers can compose features identified by earlier layers.

3.1 General

3.1.1 Usefulness

Remark 3.1 - *Advantages of Deep Neural Networks*
Here are some advantages *DNNs* offer over neural networks with a single hidden layer.

- *Hierarchical Automatic Modularisation.* A deep neural network has many layers, and the information of each layer is available to the succeeding layer. This means each layer can be considered to extract slightly more precise features (e.g. pixel colours $\to$ edges $\to$ corners $\to$ object parts $\to$ object class). These modular layers are generated automatically during training.

- *Practical Performance.* Greater depth gives greater performance than greater width. Note that large networks (either by width or depth) require more training time and larger data sets.
- *The Oscillation Argument.* There are functions f that can be represented by a deep ReLU network with a polynomial number of neurons, whereas a shallow network would require exponentially many units.

3.1.2 Overfitting

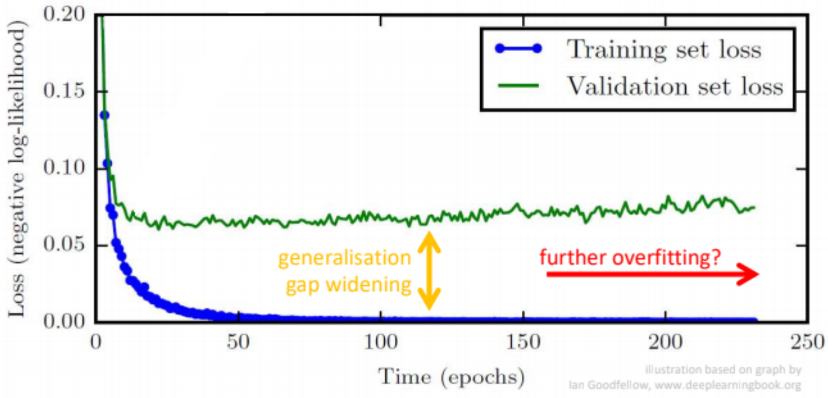


Figure 2: An example of a bad relationship between training and testing loss

Proposition 3.1 - *Overfitting*
Deep Neural Networks have lots-and-lots of parameters, meaning they have high degrees of freedom and thus prone to *Overfitting* (ie the model is fitted too closely to the training data and does not fit unseen data well).

`Figure 2` provides an example of how *Overfitting* can be identified graphically as the distance between training and testing loss increases with each time step, and the training loss converges to zero. Ideally the lines for training and testing loss would be fairly similar.

Overfitting can be combatted by: Using more data; Using data which represents the full sample space better; And, strategic sampling of data.

Definition 3.2 - *Data Augmentation*
Collecting data can be expensive and hard. *Data Augmentation* is a set of techniques to increase the available set of data, without having to collect more data, by slightly modifying already observed data. For images this may involve: cropping; translating; adding noise; shifting the hue...

Definition 3.3 - *Regularisation*
A *Regularisation* is any modification made to a learning algorithm which is intended to reduced its *generalisation error*, but not its *training error*. This is typically done by introducing more information and making changes to the *Loss Function*.

L-Regularisation places constraints on the weight space, thus reducing the searchable area for the model.

L_1-Regularisation extends the cost function $J(\cdot,\cdot)$ to include a sum of the absolute values of all the weights, with a dampening factor λ.

This gives the following cost function J_{L1} and weight update rule

$$\begin{array}{rcl} J_{L1}(X;W) & := & J(X,W)+\lambda\sum_{w\in W}|w| \\ W^l & \leftarrow & W^l-\eta\left(\nabla J+\lambda\cdot\texttt{sign}(W^l)\right) \end{array}$$

L_2-Regularisation extends the cost function $J(\cdot,\cdot)$ to include a sum of the square value of each weights, with a dampening factor λ.

This gives the following cost function J_{L2} and weight update rule

$$\begin{array}{rcl} J_{L2}(X;W) & := & J(X,W)+\frac{\lambda}{2}\sum_{w\in W}w^2 \\ W^l & \leftarrow & W^l-\eta\left(\nabla J+\lambda W^l\right) \end{array}$$

Remark 3.2 - *L_1 Regularisation vs L_2 Regularisation*
Both L_1 *Regularisation* and L_2 *Regularisation* encourage the model to learning the smallest-valued set of weights which minimise the normal cost function. Moreover, they discourage extreme weight values.

The key difference is that, in L_1 *Regularisation* the less important features are shrunk to zero, removing that feature altogether.

Definition 3.4 - *Dropout*
Dropout is a training procedure which builds on the idea that *Neural Networks* are an ensemble of sub-networks (See `Proposition 1.1`) and seeks to allow each sub-network to train somewhat independently.

Dropout does this by, each training loop, choosing a random set of nodes and setting the inbound weight to each of these nodes to 0. This effectively immobilises this set of nodes during this training loop

During validation all weights are turned on, so the output of the network is significantly greater. To combat this weights are set to pW in order to reduce their magnitude (where p is the probability of a particular node being immobilised in a particular training loop).

Definition 3.5 - *DropConnect*
DropConnect is a variation on *Dropout* where <u>connections</u> are dropped, rather than nodes. This a more fine-grained approach to ensemble learning than *Dropout*.

3.1.3 Extensions

Proposition 3.2 - *Extensions of Neural Networks*
Here I suggest some areas which could be explored to extend implementation of *Neural Networks*

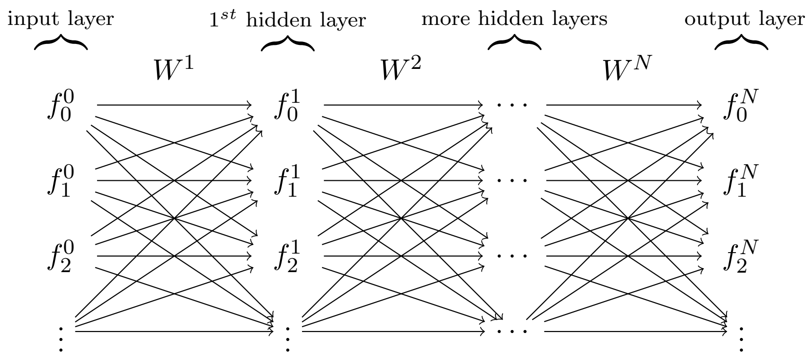
- Learn an ensemble of deep networks and then compress them into a single shallow-network. (This is sometimes possible).
- Learn a set of networks which each deal with a specific subtask of the problem and then learn another network whcih decides which of these specialised networks (or which combination) to use.

3.2 Fully Connected DNN

Definition 3.6 - *Fully Connected DNN*
A *Fully Connected DNN* is an *Artificial Neural Network* where each node in a layer takes an input from every node in the preceding layer and passes its output to every node in the next layer.

The output of each perceptron is the weighted sum of its inputs, passed through an activation function.

Below is a diagram of a MLP of *depth N* (i.e. there are N layers of computation)



Note that each layer can have a different number of nodes (AKA *width*).

For each consecutive pair of layers $\boldsymbol{f}^i, \boldsymbol{f}^j$ (of widths n_i, n_j respectively) there is an associated weight matrix $W \in \mathbb{R}^{n_i \times n_j}$ st $\boldsymbol{f}^j = W^T \boldsymbol{f}^i$.

Remark 3.3 - *Discriminating Power of Differt Depth Fully Connected DNNs*
A *Fully Connected DNN* with a single hidden layer is sufficient to represent any boolean or continuous function, althought the layer may be exponentially wider than the input.

A *Fully Connected DNN* with *two* hidden layers is sufficient to represent any mathematical function.

Proposition 3.3 - *MLPs as Computation Graphs*

$$
\begin{array}{rrcll}
 & s_i^j & := & (W^j)^T f^{j-1} & \text{weighted sum of the } i^{th} \text{ node of the } j^{th} \text{ hidden layer} \\
\implies & \dfrac{\partial s_i^j}{\partial w_{ii}^j} & = & f_i^{j-1} & \\
 & f_i^j & := & g_i^j(s_i^j) & i^{th} \text{ output vajue of } j^{th} \text{ hidden layer} \\
\implies & \dfrac{\partial f_i^j}{\partial s_i^j} & = & \text{depends on def of } g_i^j &
\end{array}
$$

Proposition 3.4 - *Output Layer for Classification Problem*
When using a *Fully Connected DNN* for classification the output layer represents a probability distribution for each possible class. This means, in the output layer, the node values are in $[0, 1]$ and they sum to 1.

This distribution is achieved by using a *Softmax Neuron Group* in the last layer with activation function. Defined as

$$g_j^N(s_j^N) := \frac{e^{s_j^N}}{\sum\limits_{i \in \text{Group}} e^{s_i^N}}$$

This has gradients

$$\begin{array}{rcl} g_j'^N(s_j^N) & = & f_j^N(1 - f_j^n) \\ g_j'^N(s_i^N) & = & -f_j^N f_i^N \quad \text{for } i \neq j \end{array}$$

3.3 Convolution Connected DNN

Remark 3.4 - *This is actually the Cross-Correlation operation, but is what is used in practice.* Convolution flips the kernel.

Definition 3.7 - *Channels*
When we have multiple data readings per instance (e.g. for each pixel of an RGB image) we consider each of these data fields to be a *channel*. When we apply a convolution they must have the same dimension as the number of channels, and they can applied to both space & channels.

Definition 3.8 - *Convolution DNN (CNNs)*
A *Convolution DNN* is an *Artificial Neural Network* contains a *Convolutional Layer* (it may also contain fully connected layers). *Pooling Layers* are common in CNNs in order to reduce dimensionality.

- A *Convolutional Layer* takes an n-dimensional grid-like topology (e.g. an image or video) as an input and applies several convolutions to this input (rather than matrix multiplication). The output from the convolution in each position of the input is passed through an *Activation Funtion* and then to the node in the layer at the equivalent position. The convolutional layer is three dimensional with X, Y representing spatial data and Z the convolution applied.

  A *Convolutional Layer* has two possible additions *Zero Padding* and *Stride Length*. See `Definition 4.4` and `Definition 4.5`.

- A *Pooling Layer* takes an n-dimensional grid-like topology as an input and for each position it outputs a summary of the values in the neighbourhood of that position. This summary is typically the: mean, min or max value; and the size of the neighbourhood is user defined.

  Pooling is applied after the convolution and the activation function, and reduces the output size.

Rather than fitting/learning a weights matrix, here we are learning the *Kernel* (AKA *Tensor*) of the convolution.

Definition 3.9 - *Zero Padding*
The output from *Convolution* is smaller than the original input. This is bad as eventually the dataset would disappear. To avoid this *Zero Padding* is used. *Zero Padding* adds rings zeros to the outside of the input st the input and output are the same size after *Convolution* is applied.

The number of rings added depends on the size of the Kernel. For a Kernel of size $N \times M$: $M - 1$ rows are added to the top & bottom of the input; and, $N - 1$ columns are added to the left and right of the input.

Definition 3.10 - *Stride Length*
Stride Length defines how far the convolution operation steps each time. This applies to both horizontal and vertical steps. The greater the *Stride Length*, the smaller the output will be.

In practice *Stride Length* is rarely set to one as this requires a lot more weights to be fitted and often little is gained by looking at values which are adjacent.

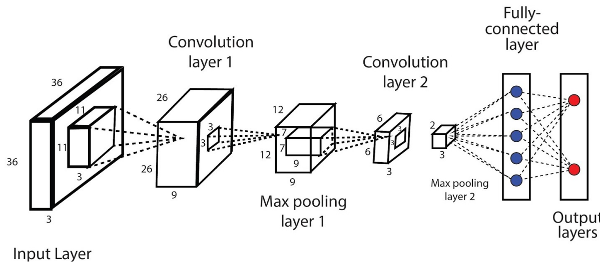


Figure 3: Example of an Architecture for a Convolution DNN

Example 3.1 - *CNN Architecture*
`Figure 3` gives an example of an architecture for a convolution DNN. In this example

- *Input Layer* has dimension $36 \times 36 \times 3$. This could be an RGB image with 36×36 pixels.
- *Convolution Layer 1* applies 9 different convolutions, each with a kernel of dimension $11 \times 11 \times 3$. In this example a stride of 1 is used and no zero-filtering. This means the output has dimension 26×26 and a depth of 9 (due to 9 convolutions being used).
- *Max Pooling Layer 1* applies the max pooling operation to each $3 \times 3 \times 1$ set of values in *Convolution Layer 1*. In this example a stride of 2 is used, meaning the output has dimension $12 \times 12 \times 9$.
- *Convolution Layer 2* applies 3 different convolutions, each with a kernel of dimension $7 \times 7 \times 9$.
- *Max Pooling Layer 2* applies the max pooling operation to each $3 \times 3 \times 1$ set of values in *Convolution Layer 2*.
- A *Fully-Connected Layer* is the used before the final classification.

Remark 3.5 - *Attraction of CNNs*
Here are some features which distinguish *Convolution Layer*s away from *Fully Connect Layer*s

i). *Sparse Interactions.* In CNNs a node in one layer does not necessarily connected to every node in the next layer. This means that changing this node will not affect every node in the next layer. This is ideal due to the grid structure of the input, where there is implicit relationships between adjacent cells/cell groups (e.g. adjacent pixels in an image).

ii). *Parameter Sharing.* The same parameter/weight is used for more than one function in the network. This can be considered as tying two parameters together st that have the same value. We use prior knowledge to decide which nodes to tie together (rather than doing it randomly). This reduces the number of parameters which need to be optimised (which means less data is required for training). The number of reduced parameters increases for later layers. This does not affect the runtime of the forward pass, but significantly reduces the memory requirements of the model.

iii). *Equi-variant Representations.* If the input changes/shifts in a certain way (e.g. translation), then the output changes in exactly the same way. This is as a result of the previous two properties. However, CNNs are not equivariant to rotation or scaling.

Remark 3.6 - *Implementing CNNs*
In practice there are several difficulties in implementing CNNs, including

- Care needs to be taken when backpropagating CNNs with zero padding or stride greater than 1.

Remark 3.7 - *Training CNNs*

- The most expensive part of training CNNs is training the convolutional layers. The fully-connected layers at the end are relatively inexpensive as they have a small number of features.
- When performing gradient descent, every gradient step requires a complete run of feed-forward propagation and backward propagation through the entire network.

3.3.1 Residual Networks (ResNets)

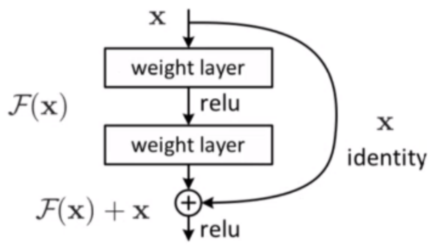


Figure 4: Example of a Residual Network Layer

Proposition 3.5 - *Residual Networks (ResNet)*
ResNets are a version in CNNs where filters are applied to the input and then merged back (using addition) with the input before being passed to the activation function. This leads to faster convergence by searching for weights which deviate only slightly from the identity. This allows for deeper networks.

See `Figure 4` for an example of the architecture of a *ResNet* layer.

3.4 Recurrent DNN

TODO

4 Algorithms

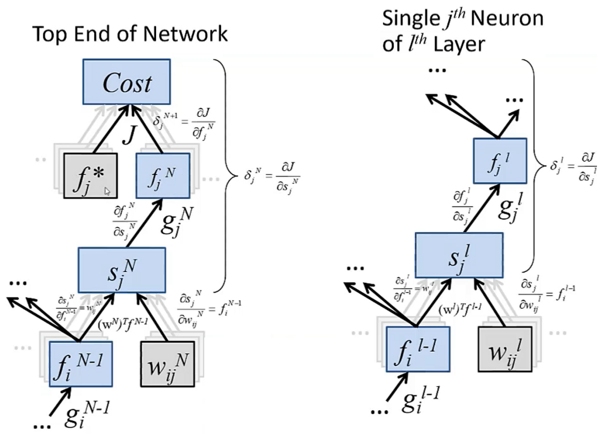


Figure 5: Computational Graph of an ANN

Remark 4.1 - ***Figure 5***

- $J(\cdot,\cdot)$ is the cost function.
- $s_j^l := (w^l)^T f^{l-1}$ is the signal of the j^{th} node in the l^{th} layer.
- $g_j^l(\cdot)$ is the activation function of the j^{th} node in the l^{th} layer.
- $f_j^l := g_j^l(s_j^l)$ is the output of the j^{th} node in the l^{th} layer.

4.1 Backpropagation

Definition 4.1 - *Backpropagation*
Backpropagation is an algorithm for training feedforward neural-networks. *Backpropagation* uses *Reverse Auto-Differentiation* (See `Section 0.2`) o calculate the derivative of the loss function wrt each weight. The value of the derivative then defines how much to update each weight by.

Proposition 4.1 - *Backpropagation Algorithm*
For each training sample $(\mathbf{f}^0, \mathbf{f}^*)$ where $\mathbf{f}^0$ is the input and $\mathbf{f}^*$ is the true output.

i). Read the input & perform a forward pass through the network to calculate signals s_j^l and outputs f_j^l for all nodes in all layers.

$$s_j^l := (\mathbf{w}_j^l)^T \mathbf{f}^{l-1} \quad f_j^l := g_j^l(s_j^l)$$

ii). Calculate the cost function value $J(f_j^*, f_j^N)$ between each output layer neuron f_j^N and its target value f_j^*.

iii). Calculate the error derivatives δ_j^{N+1} of the cost function J wrt each output of the last hidden layer f_j^N (ie the values of the output layer).

$$\delta_j^{N+1} := \frac{\partial J}{\partial f_j^N}$$

iv). Compute the error derivative δ_j^N of the cost function wrt the signals of the last hidden layer s_j^N.

$$\delta_j^N := \frac{\partial J}{\partial s_j^N} = g_j^{N\prime}(s_j^N) \cdot \delta_j^{N+1}$$

v). Layer-by-layer: Calculate the *error derivatives* δ_i^{l-1} of the cost function wrt the signal s_j^{l-1} of each neuron in the next layer, using the error derivatives δ_j^l of the layer above.

$$\delta_i^{l-1} := \frac{\partial J}{\partial s_i^{l-1}} = g_i^{l-1\prime}(s_i^{l-1}) \sum_{j=1}^{d(l)} w_{ij}^l \delta_j^l$$

where $d(l)$ is the width of the l^{th} layer.

vi). Calculate the error derivates $\frac{\partial J}{\partial w_{ik}^l}$ wrt to the weights of each neuron w_{ik}^l using the error derivatives of the neuron activities δ_j^l.

$$\frac{\partial J}{\partial w_{ij}^l} := \frac{\partial J}{\partial s_j^l} \frac{\partial s_j^l}{\partial w_{ij}^l} = \delta_j^l f_i^{l-1}$$

vii). Update all weights w_{ij}^l based on the deltas and neuron activities using the Gradient Descent learning rule

$$w_{ij}^l \leftarrow w_{ij}^l - \eta f_i^{l-1} \delta_j^l$$

where η is a pre-defined learning rate.

Proof 4.1 - *Derivation of* δ_i^{l-1}

$$\begin{aligned}
\delta_i^{l-1} &:= \frac{\partial J}{\partial s_i^{l-1}} \\
&= \sum_{j=1}^{d(l)} \underbrace{\frac{\partial J}{\partial s_j^l}}_{\delta_j^l} \underbrace{\frac{\partial s_j^l}{\partial f_j^{l-1}}}_{w_{ij}^l} \underbrace{\frac{\partial f_i^{l-1}}{\partial s_i^{l-1}}}_{g_i^{l-1\prime}(s_i^{l-1})} \\
&= \sum_{j=1}^{d(l)} \delta_j^l w_{ij}^l g_i^{l-1\prime}(s_i^{l-1}) \\
&= \underbrace{g_i^{l-1\prime}(s_i^{l-1})}_{\text{independent}} \sum_{j=1}^{d(l)} w_{ij}^l \delta_j^l
\end{aligned}$$

Definition 4.2 - ∇J

The derivative for layer l defined below

$$\nabla J^l := \begin{pmatrix} f_1^{l-1}\delta_1^l & f_1^{l-1}\delta_2^l & \dots \\ f_2^{l-1}\delta_1^l & f_2^{l-1}\delta_2^l & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

Remark 4.2 - *Limitations with Backpropagation Algorithm*

Here are some limitations to the *Backpropagation Algorithm*. These are part of the reason why the practice of deep learning did not start earlier.

- The *Vanishing Gradient Problem.* Gradients are unstable/noisy when you backpropagate gradients in a very deep network, meaning that when the true value of the gradient gets very small it is lost in noise.
- Descent-based optimisation techniques need to work accurately and fast in practice, despite large training data sets. This was not possible before GPU parallelisation and improved optimisers.
- Regularisation techniques are critical to achieve good generalisation beyond the training data available (avoid overfitting).

Remark 4.3 - *Activation Functions need to be Differentiable & Non-Linear*
In the *Backpropagation Algorithm* the derivative of each activation function is used, so each activation function must be differentiable.

The step-function does not fulfil this. This is addressed (usually) by using the *ReLU* activation function.

4.2 Gradient Descent

Definition 4.3 - *Gradient Descent*
Gradient Descent is an iterative algorithm for finding a local minimum of a differentiable function. In an ANN this is done by learning a set of weight values $\boldsymbol{w}$ which produce a local minimum for a given cost function J. The update rule for gradient descent is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \underbrace{\eta \cdot \nabla J(X; \boldsymbol{w}_t)}_{\Delta \boldsymbol{w}}$$

where η is a specified learning rate and $\nabla J(X; \boldsymbol{w}_t)$ is the partial derivative of the cost function wrt to the weights (ie the direction of fastest descent). We calculate the i^{th} component of $\Delta \boldsymbol{w}$ after observing as $(\boldsymbol{x}, f^*(\boldsymbol{x}))$

$$[\Delta \boldsymbol{w}]_i = \eta x_i(\underbrace{\boldsymbol{w}_t^T \boldsymbol{x}}_{f(\boldsymbol{x};\boldsymbol{w}_t)} - f^*(\boldsymbol{x}))$$

Definition 4.4 - *Online Gradient Descent*
Below is an online algorithm for *Gradient Descent* (ie you can pass it new data without having to restart the process).

i). Initialise all weights W randomly.

ii). `for each training sample` (x, f^*) `do`:

  (a) Forward propagate to calculate network output values.
  (b) Back propagate to calculate ∇J^l for each layer l
  (c) Update weights for each layer l $W^l \leftarrow W^l - \eta \nabla J^l$.
  (d) `if` (stopping criteria met) `break loop`.

iii). `return` final weights W^l for all layers.

Definition 4.5 - *Simulated Annealing*
Simulated Annealing is a process for setting the learning rate η by testing τ learning rates in the interval $[\eta_\tau, \eta_0]$. Annealing transitions from η_0 to η_τ.

i). Initialise all weights W randomly.

ii). `for` $k = 0, \dots, \tau$ `do`:

  (a) $\eta_k := \left(1 - \frac{k}{\tau}\right)\eta_0 + \frac{k}{\tau}\eta_\tau$

  (b) `for each training sample` (x, f^*) `do`:

    i. Forward propagate to calculate network output values.

    ii. Back propagate to calculate ∇J^l for each layer l

    iii. Update weights $W^l \leftarrow W^l - \eta_k \nabla J^l$.

    iv. `if` (stopping criteria met) `break loop`.

  (c) `return` final weights W^l.

Remark 4.4 - *Limitations of Online Gradient Descent*

- *Sample Size.* Using single samples at a time to find the minimum point of the cost function will only roughly approximate aspects of the cost function gradient. This leads to a very noisy gradient descent which may not find the global minimum at all. This is exasperated if the learning rate η is set too *high* as the minimum may be overshot.

  This addressed by *Deterministic Gradient Descent* and *Stochastic Gradient Descent.*

- *Constant Learning Rate.* Having the same learning rate η for the whole learning process is bad as you cannot skip over shallow gradient, nor focus more iterations in areas of steep gradient. We would rather have weights with a shallow gradient have a greater learning rate, and weights with a steeper gradient have a lesser learning rate.

  This is addressed by *Learning via Momentum.*

- *Monotonic Learning Rate.* Having the same learning rate η for all weights is problematic as weights have different gradients and thus should be learning at different rates as they will hit shallow/steep areas at different times.

  This is addressed by *Adaptive Gradient Algorithm* and *Root-Mean-Square Propagation.*

4.2.1 Stochastic & Deterministic Gradient Descent

Definition 4.6 - *Alternatives to Online Gradient Descent*
As *Online Gradient Descent*'s use of a single sample at a time is bad, multiple samples can be used by using mean ∇J. Here are two approaches

- *Deterministic Gradient Descent* (DGD) where <u>all</u> training samples (X, F^*) are used at once. Given a small enough learning rate η this will process to the true local minimum, but at high computational cost.

- *Stochastic Gradient Descent* (MiniBatch) where a <u>small subset</u> of training samples (X, F^*) are used each iteration. This is still good at finding a minimum, and much less computationally costly.

For the average of ∇J we use

$$\nabla J = \frac{1}{|X|} \nabla_W \sum_j L(\underbrace{f(\mathbf{x}_j, W)}_{\text{prediction}}, f^*)$$

4.2.2 Momentum

Definition 4.7 - *Learning via Momentum*
Momentum is a extension to the *Gradient Descent* weight update rule. Rather than a fixed learning rate η, a velocity term v_t is introduced which defines the step length and direction. The closer the direction of the previous step is aligned to the direction of the current step, the longer the step length is.

$$W^l \to W^l + \underbrace{v_{t+1}^l}_{\text{momentum}} \quad \text{where} \quad v_{t+1}^l := \alpha \cdot v_t^l - \eta \nabla J(X; W_t^l)$$

where α, η are hyperparameters for momentum and learning rate, respectively.

Proposition 4.2 - *Nesterov Accelerated Gradient (NAG)*
Nesterov Accelerated Gradient is an extension of *Learning via Momentum* which, instead of calculating the gradient at the current position, looks-ahead at the gradient of the target. This is since *Momentum* will carry us towards the next location anyway.

Formally we now define weight updates as

$$W^l \leftarrow W^l + v_{t+1}^l \quad \text{where} \quad v_{t+1}^l = \alpha v_t^l - \eta \nabla J(X; \underbrace{W^l + \alpha v_t^l}_{\substack{\text{location} \\ \text{perview}}})$$

NAG is consistently better that *Learning via Momentum* in practice.

Remark 4.5 - *Limitations of Momentum Methods*
Methods which use momentum progress very slowly in shallow plateau regions of the cost function state space as momentum is not able to build up. This can be rectified by tuning the learning rate.

Proposition 4.3 - *Newton's Method*
Newton's Method removes all hyperparameters (η and α) and instead uses curvature to rescale the gradient by multiplying the gradient by the inverse Hessian of the current cost function $H(J(X; W_t))$.

Newton's Method takes aggressive steps in directions of shallow curvature, and shorter steps in directions of steep curvature, however it is attracted to saddle points (bad!).

$$W^l \leftarrow W^l - H(J(X; W^l))^{-1} \nabla J(X; W^l)$$

Computing and inverting the Hessian is computationally and space expensive.

Remark 4.6 - *The more parameters there are the more likely saddle points are*
Saddle points occur when the hessian has both positive & negative eigenvalues. This is more likely when we have more parameters (as the probability of all eigenvalues being positive is low).

Random Matrix Theory states that the lower the cost function J is (ie the closer it is to the global minimum), the more likely it is to find positive eigenvalues. This means that if we find a minimum it is likely to be a good one (i.e. low cost).

Thus, most critical points with higher cost function values are likely to be saddle points, which we can escape using symmetry-breaking descent methods.

4.2.3 Adaptive Gradient Algorithm (AdaGrad)

Definition 4.8 - *Adaptive Gradient Algorithm (AdaGrad)*
Adaptive Gradient Algorithm (AdaGrad) is an extension to the *Gradient Descent* weight update rule st each weight can have a different learning rate. Scaling the learning rate η for each weight wrt the past gradients for that weight.

For each weight w_i the following update rule is used

$$w_i \leftarrow w_i - \frac{\eta g_t}{\sqrt{M_t} + \varepsilon} \quad \text{where} \quad M_t := \sum_{i=1}^{t} g_i^2$$

where η is a hyper-parameter for the learning rate; g_t is the gradient for this weight in this iteration; M_t is the square-sum of all observed gradients for this weight; and, $\varepsilon > 0$ is a small constant to prevent dividing by zero.

4.2.4 Root-Mean-Square Propagation (RMSProp)

Definition 4.9 - *Root-Mean-Square Propagation (RMSProp)*
Root-Mean-Square Propagation (RMSProp) is an extension to the *AdaGrad* weight update rule which introduces a smoothing parameter β to combat the aggressive reduction of learning speed.

For each weight w_i the following update rule is used

$$w_i \leftarrow w_i - \frac{\eta g_t}{\sqrt{M_t} + \varepsilon} \quad \text{where} \quad M_t := (1-\beta)g_t^2 + \beta M_{t-1}$$

where η is a hyper-parameter for the learning rate; g_t is the gradient for this weight in this iteration; M_t is an exponentially decaying average[1] of the square of all previously observed gradients of this weight; and, $\varepsilon > 0$ is a small constant to prevent dividing by zero.

4.2.5 Adaptive Moment Estimation (AdaM)

Definition 4.10 - *Adaptive Moment Estimation (AdaM)*
Adaptive Moment Estimation (AdaM) which aims to smooth the incoming gradient g_t in *RMSProp*. This is done by introducing an exponentially decaying average of all previously observed gradients for this weight G_t (Not just their squares M_t)[2].

For each weight w_i the following update rule is used

$$\begin{array}{rcl} G_t & := & (1-\alpha)g_t + \alpha G_t \\ \bar{G} & := & \dfrac{G_t}{1-\alpha^{t-1}} \\ M_t & := & (1-\beta)g_t^2 + \beta M_{t-1} \\ \bar{M} & := & \dfrac{M_t}{1-\beta^{t-1}} \\ w_i & \leftarrow & w_i - \dfrac{\eta \bar{G}}{\sqrt{\bar{A}} + \varepsilon} \end{array}$$

where α, β, η are hyper-parameters for the learning and decay rates; g_t is the gradient for this weight in this iteration; G_t, M_t are an exponentially decaying averages for previously observed

[1]The influence of each gradient in M_t decays over time. The closer β is to $\frac{1}{2}$ the quicker this occurs.

[2]G_t is an estimate of the first moment (*ie* mean) of the gradients. M_t is an estimate of the second moments (*ie* un-centred variance) of the gradients.

gradients of w_i, and their squared value; and, $\varepsilon > 0$ is a small constant to prevent dividing by zero.

Remark 4.7 - *$\bar{G}$ and $\bar{M}$*
$\bar{G}$ forces fast startups.

Using the square-root of $\bar{M}$ in the update to w_i amplifies shallow gradients (since $\bar{M}$ is small) and dampens steep gradients (since $\bar{M}$ is large).

Remark 4.8 - *Using AdaM*
Applying AdaM to a ReLU-based network is sufficient to perform deep learning but there are other features which need to be decided.

i). What to set hyperparameters $\alpha, \beta, \varepsilon$ and the size of each training batch.

ii). How to initialise the network.

iii). How to avoid overfitting.

iv). Which loss function to use.

Tuning and trial-and-error of these features is often required to achieve good results from deep learning.

4.3 Cost Function

Definition 4.11 - *Cost Function, $J(\cdot,\cdot)$*
A *Cost Function* is a real-valued measure of how inaccurate a classifier is for a given input configure (*ie* test data $[X, f^*(X)]$ and weights W). Greater values imply the classifier is less accurate.

Neural networks us training to learn a set of weights which minimise the *Cost Function*, given the training data.

4.3.1 Numerical

Proposition 4.4 - *Distance Measures for Numerical Quantities $L(\cdot,\cdot)$*
Let $x, x^* \in \mathbb{R}$ with x representing an estimate and x^* the true value.

Here are some popular measures of distance between real-valued quantities.

$$\begin{aligned}
\textit{Identity Distance}\ \ & L(x, x^*) := |x - x^*| \\
\textit{Absolute Distance}\ \ & L(x, x^*) := (x - x^*)^2 \\
\textit{Quadratic Distance}\ \ & L(x, x^*) := \{x \equiv x^*\} = \begin{cases} 1 & \text{if } x \equiv x^* \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Proposition 4.5 - *Cost Functions for Numerical Quantities J*
Let $[X, f^*(X)]$ be a set of training data, $f_W(\cdot)$ be the values our model predicts when using weight set W and $L(\cdot,\cdot)$ be some *Loss Function*.

Here are some *Cost Functions* for classifiers which classify real-valued quantities.

$$\textit{Expected Loss}\ \ J(X;W)=\mathbb{E}[L(f_W(X),f^*(X))]$$

$$\textit{Empirical Risk}\ \ J(X;W)=\frac{1}{|X|}\sum_{\mathbf{x}\in X}L(f_W(\mathbf{x}),f^*(\mathbf{x}))$$

4.3.2 Classification

Remark 4.9 - *Non-Numerical Cost Measures*
In non-numerical classification there is no obvious way to define the distance between classification. Thus we cannot define a cost function.

How we combat this depends on how the output layer is defined. When using the setup of the output layer defined in `Proposition 4.2`, which uses *Softmax*, the *Cross-Entropy Cost Function Function*

Definition 4.12 - *Cross-Entropy Cost Function J*
Let f_j^* be the ground truth for output node j and f_j^N be our predicted value for the ground truth for output node j.

The *Cross-Entropy Cost Function* J and its derivative δ_i^N wrt signal i is defined as

$$\begin{array}{rcl} J & := & -\sum\limits_{j\in\text{Group}} f_j^*\ln(f_j^N) \\ \delta_i^N & = & f_i^N-f_i^* \end{array}$$

The cost function derivative δ_i^N is just the difference between the predicted value and the true value.

Proof 4.2 - *Derivation of Derivative δ_i^N of Cross-Entropy Cost Function*
The steepness of the cost function derivative $\frac{\partial J}{\partial f_j^N}$ exactly cancels the shallowness of the softmax derivative $\frac{\partial f_j^N}{\partial s_i^N}$, leading to an *MSE-Style Delta* δ_i^N which is propagated backwards from layer

N.

$$\begin{aligned}
\delta_i^N &:= \frac{\partial J}{\partial s_i^N} \\
&= -\sum_{j\in\text{Group}} f_j^* \underbrace{\frac{\partial \ln(f_j^N)}{\partial s_i^N}}_{\text{apply chain rule}} \\
&= -\sum_{j\in\text{Group}} f_j^* \frac{1}{f_j^N}\frac{\partial f_j^N}{\partial s_i^N} \quad \text{where } f_j^N := \frac{e^{s_j^N}}{\sum_{i\in\text{Group}} e^{s_i^N}} \\
&= -\sum_{j=i} f_j^* \frac{1}{f_j^N} \underbrace{f_j^N(1-f_j^N)}_{\frac{\partial f_j^N}{\partial s_i^N} \text{ for } i=j} - \sum_{j\neq i} f_j^* \frac{1}{f_j^N} \underbrace{(-f_j^N f_i^N)}_{\frac{\partial f_j^N}{\partial s_i^N} \text{ for } i\neq j} \\
&= -f_i^*(1-f_i^N) + \sum_{j\neq i} f_j^* \frac{f_j^N f_i^N}{f_j^N} \\
&= -f_i^* + \underbrace{f_i^* f_i^N + \sum_{j\neq i} f_j^* f_i^N}_{=f_i^N \sum_{j\in\text{Group}} f_j^*} \\
&= \underbrace{f_i^N\left(\sum_{j\in\text{Group}} f_j^*\right)}_{=1} - f_i^* \\
&= f_i^N - f_i^*
\end{aligned}$$

0 Reference

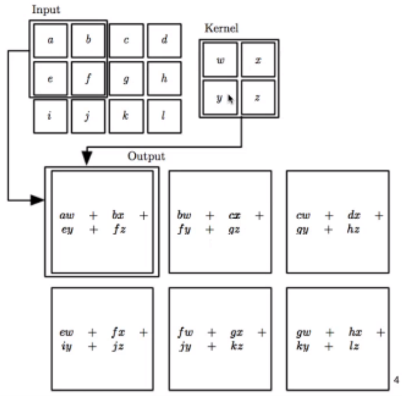
Definition 0.1 - *Convolution* $*$
Convolution is an operation which takes two functions f & g and produces a third function $(f * g)$ which shows how one function affects the other.

Consider the 2D discrete space where f, g are real valued 2D matrices (often representing images). *Convolution* in this case is defined as

$$(f * g)(x, y) = \sum_{i \in I} \sum_{j \in J} f(x - i, y - j) \cdot g(i, j)$$

where I is the horizontal index of g and J is the vertical index of g.

Typically a *Convolution* is then shift right until the RHS of the kernel surpasses the RHS of the input, it will then shift down a row. This will continue until the bottom row has been completed



Note that *Convolution* reduces the size of the input. To avoid this *Zero-padding* is used, where a ring on zeros is placed around the input.

Definition 0.2 - *Hessian Matrix* $H(\cdot)$

$$H(J) = \begin{pmatrix} \frac{\partial J^2}{\partial w_1^2} & \frac{\partial J^2}{\partial w_1 \partial w_2} & \cdots & \frac{\partial J^2}{\partial w_1 \partial w_n} \\ \frac{\partial J^2}{\partial w_2 \partial w_1} & \frac{\partial J^2}{\partial w_2^2} & \cdots & \frac{\partial J^2}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J^2}{\partial w_n \partial w_1} & \frac{\partial J^2}{\partial w_n \partial w_2} & \cdots & \frac{\partial J^2}{\partial w_n^2} \end{pmatrix}$$

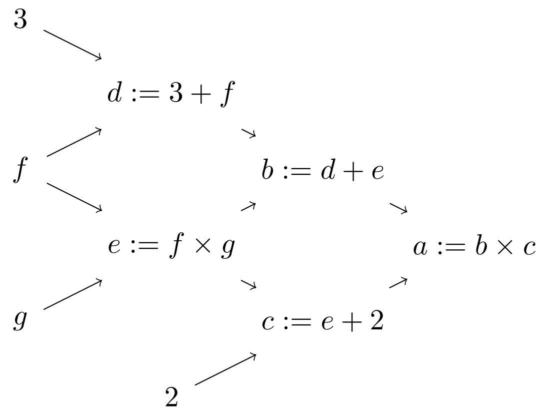
0.1 Feed-Forward Network

Definition 0.3 - *Feed-Forward Network*
A *Feed-Forward Network* is an artificial neural network where the connections between nodes are uni-directional. Data is provided to the input layer and then an output is returned from the output layer, no layers are visited twice.

0.2 Auto-Differentiation

$$
\begin{aligned}
a &= b \times c \\
b &= d + e \\
c &= e + 2 \\
d &= 3 + f \\
e &= f \times g
\end{aligned}
$$

(a) Equations



(b) Graph

Figure 6: Example of a Feedforward Computational Graph

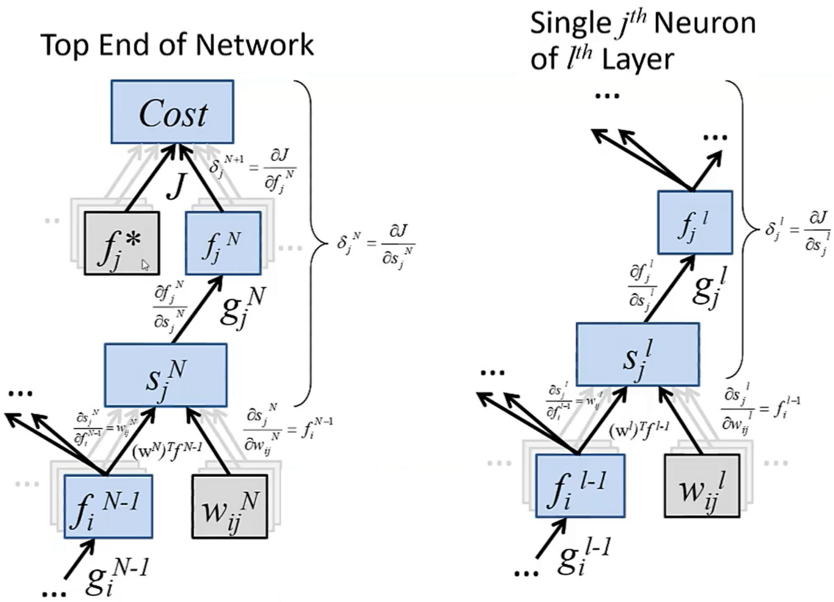


Figure 7: A Neural Network as a Feedforward Computational Graph.
J is the cost function, s_j^l, $g_j^l(\cdot)$, f_j^l are the signal, activation function and output of the j^{th} node in the l^{th} layer respectively.

Definition 0.4 - *Feedforward Computational Graph*
A *Feedforward Compurational Graph* is a uni-directional graph which represents a series of equations. In a *Feedfoward Computational Graph* nodes represent variables or constants and edges represent mathematical operations (and thus dependence between nodes).

If values are set for all leaves of the graph, then values can be calculated for all nodes in the graph. Moreover, if values are defined for all nodes at a given depth then values can be calculate for all variables higher up the tree.

`Figure 6` provides an example of a *Feedforward Computational Graph.*

Definition 0.5 - *Auto-Differentiation*
Auto-Differentiation is a technique for calculating partial derivatives of a *Feedforward Computational Graph* and is used by *Gradient Descent.*

Consider two nodes in a computational graph x, y. Use the following process to calculate the partial derivative $\frac{\partial x}{\partial y}$.

i). Establish all the paths from y to x in the graph.

ii). Calculate the partial derivatives of each step of these graphs. (i.e. if there is a path $y \to a \to x$ calculate $\frac{\partial a}{\partial y}, \frac{\partial x}{\partial a}$).

iii). Apply the chain rule along each path (i.e. For $y \to a \to x$ calculate $\frac{\partial a}{\partial y} \cdot \frac{\partial x}{\partial a}$).

iv). Sum these calculations together to get the final result $\frac{\partial x}{\partial y}$.

v). Substitute variables to make computation easier.

Example 0.1 - *Auto-Differentiation using a Feedforward Computational Graph*
Consider calcualte $\frac{\partial f}{\partial a}$ for the graph in `Figure 6`.

i). There are three paths from f to a in the graph: (1) $f \to d \to b \to a$; (2) $f \to e \to b \to a$; and, (3) $f \to e \to c \to a$.

ii). We need to calculate the following sets of partial derivatives: $\frac{\partial d}{\partial f}, \frac{\partial b}{\partial d}, \frac{\partial a}{\partial b}$ for (1); $\frac{\partial e}{\partial f}, \frac{\partial b}{\partial e}, \frac{\partial a}{\partial b}$ for (2); and, $\frac{\partial e}{\partial f}, \frac{\partial c}{\partial e}, \frac{\partial a}{\partial c}$ for (3).

$$\begin{array}{ccccccccc} & (1) & & & (2) & & & (3) & \\ \frac{\partial d}{\partial f} & = & 1 & \frac{\partial e}{\partial f} & = & g & \frac{\partial e}{\partial f} & = & g \\ \frac{\partial b}{\partial d} & = & 1 & \frac{\partial b}{\partial e} & = & 1 & \frac{\partial c}{\partial e} & = & 1 \\ \frac{\partial a}{\partial b} & = & c & \frac{\partial a}{\partial b} & = & c & \frac{\partial a}{\partial c} & = & b \end{array}$$

iii). Applying the chain rule to each path gives

$$\begin{array}{cccl} (1) & \frac{\partial d}{\partial f}\frac{\partial b}{\partial d}\frac{\partial a}{\partial b} & = & 1 \cdot 1 \cdot c = c \\ (2) & \frac{\partial e}{\partial f}\frac{\partial e}{\partial f}\frac{\partial b}{\partial e} & = & g \cdot 1 \cdot 1 \cdot c = gc \\ (3) & \frac{\partial e}{\partial f}\frac{\partial c}{\partial e}\frac{\partial a}{\partial c} & = & g \cdot 1 \cdot b = gb \end{array}$$

iv). Summing the terms together we get

$$\frac{\partial a}{\partial f} = c + gc + gb$$

v). By substitution we get a final expression

$$\frac{\partial a}{\partial f} = 2 + 5g + 2fg + 2fg^2$$

So when $f = 4, g = 2$ we have that $a = 150$ and $\frac{\partial a}{\partial f} = 60$.

Proposition 0.1 - *Using Hierarchical Dependency*

By the chain rule we have that $\frac{\partial x}{\partial z}=\frac{\partial x}{\partial y}\frac{\partial y}{\partial z}$. So, if $\frac{\partial x}{\partial y}$ is already known then we just need to multiply that value by $\frac{\partial y}{\partial z}$ to get $\frac{\partial x}{\partial z}$.
This can be utilised to ease the computational load of a calculation. In particular, calculating the derivatives one layer at a time is a good strategy.

Remark 0.1 - *Usefulness of Auto-Differentiation*
Auto-Differentiation allows us to mathematical quantify the affect one variable has on another, which is good. However, the number of paths in a network grows exponentially with the number of nodes, thus this can be computational hard. (*Hierarchical Dependence* can be used to mitigate this)