

# Applied Deep Learning - Notes

Dom Hutchinson

October 30, 2020

## Contents

<b>1</b>	<b>Machine Learning</b>	<b>2</b>
<b>2</b>	<b>Artificial Neural Networks</b>	<b>2</b>
2.1	Perceptron . . . . .	2
2.2	Multi-Layer Perceptron . . . . .	4
2.3	Types of Deep Neural Networks . . . . .	5
2.3.1	Fully Connected DNN . . . . .	5
2.3.2	Convolution DNN (CNNs) . . . . .	5
2.3.3	Recurrent DNN . . . . .	7
2.4	Problems . . . . .	7
2.4.1	Overfitting . . . . .	7
2.5	Usefulness . . . . .	9
2.6	Possible Extensions . . . . .	10
<b>3</b>	<b>Training Algorithms</b>	<b>10</b>
3.1	Gradient Descent . . . . .	10
3.1.1	Auto-Differentiation . . . . .	11
3.2	Backpropagation Algorithm . . . . .	14
3.3	Momentum . . . . .	16
3.4	Function Adaptive Optimisation Algorithms . . . . .	17
3.5	Cost Functions for Classification . . . . .	18
<b>4</b>	<b>Implementation</b>	<b>19</b>
<b>0</b>	<b>Reference</b>	<b>21</b>

# 1 Machine Learning

## 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.

# 2 Artificial Neural Networks

## Remark 2.1 - 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 2.1 - Neuro-Plasticity

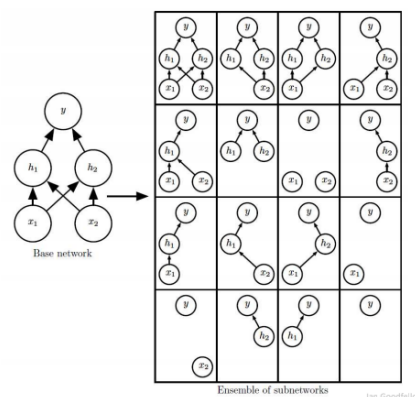
*Neuro-Plasticity* is the ability of a neural system to adapt its structure to accommodate new information (i.e. Learn). This can take several forms including growth & function changes.

## Definition 2.2 - 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.

## Proposition 2.1 - Neural Networks as an Ensemble of Sub-Networks

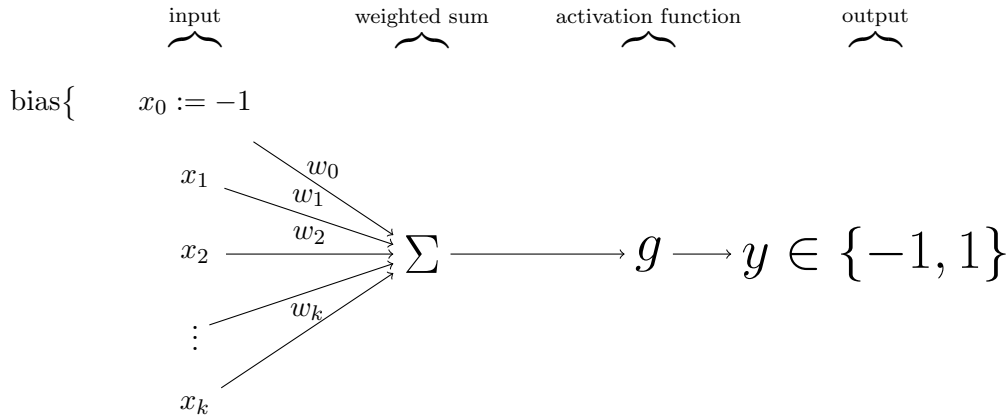
A *Neural Network* can be considered to represent many *sub-networks*. These sub-networks are switched between depending on which components are picked and how they are defined.



## 2.1 Perceptron

### Definition 2.3 - 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  $w_0$ .

$\mathbf{x} = (x_0, \dots, x_k)$  is the input.  $(x_1, \dots, x_k)$  are the inputs for the item we wish to classify

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

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

$g$  is the *Activation function* which maps from  $\Sigma$  to  $\{-1, 1\}$ , effectively performing a binary classification. The user has several options for how to define this. (n.b.  $g : \mathbb{R} \rightarrow \{-1, 1\}$ )

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

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

### Remark 2.2 - Limitations of Perceptron

A *Perceptron* can only perform linear binary classification so is not useful when two classes are not linearly separable. See **Section 2.2** for how to learn arbitrary decision boundaries.

### Proposition 2.2 - Activation Function

There are several choice for the *Activation Function* including:

**sign** binarily assigns values depend on whether they are positive or negative.

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

### Proposition 2.3 - Perceptron (Supervised) Learning Rule

We need a way for a perceptron to learn when it makes a misclassification. This is done by adjusting the weight vector  $\mathbf{w}$ . A simple learning rule is to update the current weights by a certain proportion of the error made.

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

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

**Proposition 2.4 - Training Process for a Single-Layer Perceptron**

Let  $\{(\mathbf{x}_1, f^*(\mathbf{x}_1)), \dots, (\mathbf{x}_N, f^*(\mathbf{x}_N))\}$  be a set of training data. To learn a good set of weights  $\mathbf{w}$  do the following process.

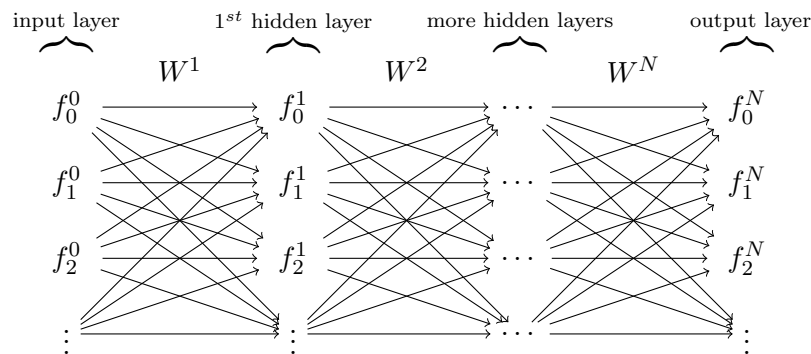
- i). Initialise the weight vector  $\mathbf{w} = \mathbf{0}$
- ii). Consider next training datum  $(\mathbf{x}_i, f^*(\mathbf{x}_i))$ .
- iii). Calculate prediction  $f(\mathbf{x})$ .
- iv). Compare prediction  $f(\mathbf{x})$  and ground truth  $f^*(\mathbf{x})$ .
- v). Update the weight vector  $\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$  where  $\Delta\mathbf{w} = \begin{cases} \eta f^*(\mathbf{x})\mathbf{x} & \text{if } f^*(\mathbf{x}) \neq f(\mathbf{x}) \\ 0 & \text{otherwise} \end{cases}$
- vi). Repeat ii)-v) until the training set is exhausted.

**2.2 Multi-Layer Perceptron****Remark 2.3 - Learning Arbitrary Decision Boundaries**

To learn an arbitrary decision boundary (i.e. anything non-linear) can be done by using a *Multi-Layer Perceptron* with non-linear activation functions.

**Definition 2.4 - Multi-Layer Perceptron**

A *Multi-Layer Perceptron* has the same general structure as a perceptron but with multiple calculations occurring and multiple output values. 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 *width* (i.e. number of nodes in the layer). For each consecutive pair of layers  $\mathbf{f}^i, \mathbf{f}^j$  (of widths  $n_i, n_j$  respectively) there is an associated weight matrix  $W \in \mathbb{R}^{n_i \times n_j}$  st  $\mathbf{f}^j = W^T \mathbf{f}^i$ . The values from the output layer are then passed to an *activation function* to make a classification.

**Remark 2.4 - Using MLPs**

An MLP with a *single* hidden layer is sufficient to represent any boolean or continuous function, although the layer may be exponentially wider than the input.

An MLP with *two* hidden layers is sufficient to represent any mathematical function.

**Proposition 2.5 - MLPs as Computation Graphs**

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

### Proposition 2.6 - Output Layer for Classification Problem

To use an MLP for classification we require the output layer to represent a probability distribution for the possible classes (i.e. each node has a value in  $[0, 1]$  and the sum of all nodes is 1).

We can force outputs to reflect this distribution using a *Softmax Neuron Group* in the last layer with activation function

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

This has gradients

$$\begin{aligned}
g_j'^N(s_j^N) &= f_j^N(1 - f_j^N) \\
g_j'^N(s_{i \neq j}^N) &= -f_j^N f_i^N
\end{aligned}$$

All outputs  $f_i^N := g_i^N(s_i^N)$  range between 0 and 1, while the  $\sum_{i \in \text{Group}} f_i = 1$ .

## 2.3 Types of Deep Neural Networks

### 2.3.1 Fully Connected DNN

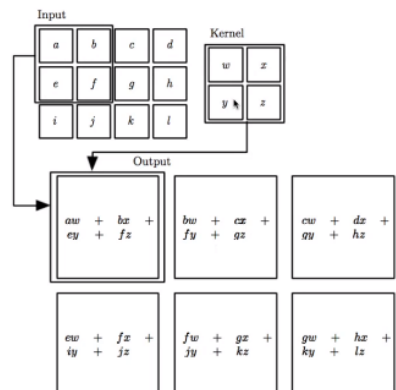
### 2.3.2 Convolution DNN (CNNs)

#### Definition 2.5 - Convolution \*

The *Convolution* operation takes an input  $x$  and a kernel  $\omega$ . The kernel is then applied to each element of the input. The actual operation depends on the set up of the input and the kernel (generally on their dimension).

$$(X * \omega)(i, j) = \sum_m \sum_n X_{i+m, j+n} \omega(m, n)$$

The kernel is first placed so its top-left entry overlaps the top-left entry of the input; the operation is applied; the kernel is then shifted right by one cell; this repeats until the right-side of the kernel surpasses the right-side of the input at which point it starts again but one cell lower. This means that  $(X * \omega)$  is likely smaller in all dimensions than  $X$ .



**Remark 2.5** - *This is actually the Cross-Correlation operation, but is what is used in practice. Convolution flips the kernel.*

**Definition 2.6** - *Convolution DNN (CNNs)*

*Convolution DNN* expects an  $n$ -dimensional grid-like topology for its input (ie an  $nD$ -array of data). This means operations can be applied to individual or groups of cells and this *CNNs* use *convolution* in place of general matrix multiplication (in at least one of its layers). The kernels used by a CNN are typically  $nD$ -arrays of kernels (AKA a *Tensor*) which are learnt from the data (rather than hand-crafted).

**Remark 2.6** - *Attraction of CNNs*

CNNs have properties which distinguish them from Fully connected networks

- 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 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). This means that nodes in later layers can have a greater *receptive field of inputs*.
- 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 (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.

**Definition 2.7** - *Pooling Operation*

A *Pooling Operation* replaces the output of a network at a certain location with a summary of the outputs in nearby positions. This is applied after convolution and the activation function, and reduces the output size. *Pooling* is typically similar to down sampling.

*Max Pooling* is a popular pooling operation where the output is changed to be the maximum value within a rectangular neighbourhood.

Other pooling operations include: average pooling; weighted average pooling;  $L^2$  norm etc.

**Proposition 2.7** - *Zero Padding*

The *kernel* shrinks the dataset, which we don't want as eventually the dataset would disappear. To avoid this *Zero Padding* is used. *Zero Padding* adds zeros to the outside of the input st the input and output are the same size after the kernel is applied to the original input data.

**Proposition 2.8** - *Higher-Dimensional Data*

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. (ie a single convolution of an image is a 3D tensor). An extra dimension is added if we want to apply multiple filters/convolutions. The number of filters applied is equal to the number of channels in the next layer.

**Proof 2.1 - Convolution in Practice**

In practice we do not convolve densely (ie every pixel), rather we skip certain pixels. The number of pixels skipped is referred to as the *stride* of the layer. This results in downsampled convolutions.

**Remark 2.7 - C**

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

**Remark 2.8 - T**

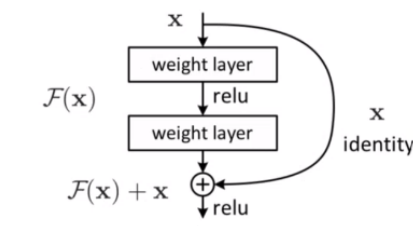
ypically CNNs start with convolution layers and then end with a couple of fully connected layers.

**Remark 2.9 - 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.

**Proposition 2.9 - Residual Networks (ResNet)**

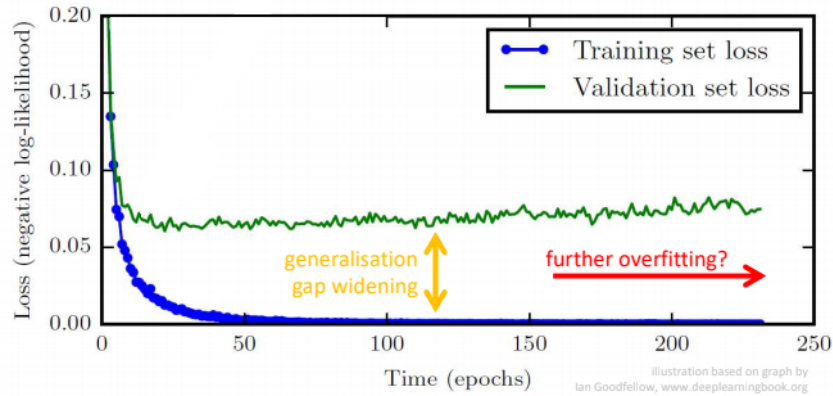
*ResNets* are a new innovation in CNNs where filters are applied to the input and then merged back (by addition) with the input before be passed to the activation function. This leads to faster convergence by searching for weights which deviate only slightly from the identity. This allows for deeper networks.

**2.3.3 Recurrent DNN****2.4 Problems****2.4.1 Overfitting****Proposition 2.10 - High Parameter Space Overfitting**

Multi-Layer perceptrons often have millions of neurons, with millions of parameters & connections. These models have very high degrees of freedom and thus are prone to overfitting.

**Proposition 2.11 - Identifying Overfitting**

When training a neural network we can plot the *Loss Function* for the training data against that for the testing data, values after each epoch. Overfitting occurs if the distance between these lines increases over time.



A natural idea from this is to keep a copy of the model with the smallest generalisation width, and then reverting to it after all the training.

**Remark 2.10** - *Overfitting can always be addressed by using more data, more representative data and/or strategically sampling data.*

Deep Learning techniques are particularly good at extract new information from new data. Whereas some older algorithms reach a ceiling where they can no longer learn form new information.

**Proposition 2.12** - *Obtaining More Data (Data Augmentation)*

Data for deep learning is expensive to collect as it requires ground truth annotations. *Data Augmentation* is a se of techniques used to increase the size of our data pool, without having to collect more data, by slightly modifying existing data. This can be done as an *online process* during training.

For images these techniques include: cropping, adding noise, rotating, translation, hue shift etc.

**Definition 2.8** - *Regularisation*

A *Regularisation* is any modification made to a learning algorithm which is intended to reduced its *generalisation error*, but not its *training error*.

**Definition 2.9** - *L-Regularisation*

*L-Regularisation* constrains the weight space (i.e. makes certain weight values more likely to be learned by the system).

**Definition 2.10** - *L1-Regularisation*

*L1-Regularisation* targets a local minimum with *sparse* weights to combat overfitting (ie using very few nodes). This is done by introducing a penalty to the cost function for every weight, based on the *absolute value* of the weight. This penalises non-zero weight values. This gives a cost function of the form

$$J(X; W) = \underbrace{\frac{1}{|X|} \sum_{x \in X} L(f(x; W), f^*(x))}_{\text{normal cost function}} + \frac{1}{2} \sum_{w \in W} \mu |w|$$

During training this gives us the update rule

$$W_{t+1} = W_t - \eta \cdot \left[ \underbrace{\nabla J_{\text{plain}}(X; W^t)}_{\text{normal cost function}} + \mu \text{sign}(W^t) \right]$$



**Definition 2.11 - L2-Regularisation**

*L2-Regularisation* targets a local minimum with *small-magnitude* weights to combat overfitting (ie lots of nodes making small contributions). This is done by introducing a penalty to the cost function for every weight, based on the *squared value* of the weight. This penalises higher weights more. This gives a cost function of the form

$$J(X; W) = \underbrace{\frac{1}{|X|} \sum_{x \in X} L(f(x; W), f^*(x))}_{\text{normal cost function}} + \frac{1}{2} \sum_{w \in W} \lambda w^2$$

During training this gives us the update rule

$$W_{t+1} = W_t - \eta \cdot \left[ \underbrace{\nabla J_{\text{plain}}(X; W^t)}_{\text{normal cost function}} + \lambda W^t \right]$$

**Definition 2.12 - Dropout**

*Dropout* uses the idea that Neural Networks are in fact an ensemble of sub-networks (**Proposition 2.1**) and tries to define a training procedure so that each sub-network can learn (somewhat) independently by dropping out nodes.

This is done by, during each training loop, setting the incoming weights to a random set of nodes to 0 with probability  $1 - p$  (effectively immobilising a random set of neurons) and then training the network. This is particularly effective on fully connected networks.

$p$  is often set to .5 before tuning on the validation set. 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.  $p$  is then tuned.

**Definition 2.13 - DropConnect**

*DropConnect* is a variation on *Dropout* where connections are dropped, rather than nodes. This is done by setting a random set of weights to zero, with probability  $1 - p$ , during each training cycle. This is a more fine-grained approach to ensemble learning than *Dropout*.

**2.5 Usefulness****Remark 2.11 - Advantages of Deep Neural Networks**

- *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  $\rightarrow$  edges  $\rightarrow$  corners  $\rightarrow$  object parts  $\rightarrow$  object class). These modular layers are generated automatically during training.
- *Practical Performance.* Greater depth gives greater performance than greater width. Note that large networks require more training time and larger data sets.
- *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.

## 2.6 Possible Extensions

### Proposition 2.13 - Network Distillation

Is it possible to learn an ensemble of deep networks, but then *compress* these deep networks into a single shallow (or more efficient) network? Sometimes.

### Proposition 2.14 - Mixture of Experts

We could learn a series of networks which each deal with a specific subtask of a problem and then use another network to decide which of these networks (or order of networks) to use in each instance.

## 3 Training Algorithms

### Definition 3.1 - Cost/Loss Function, $J$

A *Cost Function*  $J(\cdot; \cdot)$  is a real-valued measure of how inaccurate a classifier is for a given input configuration (test data & weights). Greater values imply the classifier is less accurate. Here are some common cost functions

Expected Loss  $J(X; \mathbf{w}) = \mathbb{E}[L(f(\mathbf{x}, \mathbf{w}), f^*(\mathbf{x}))]$

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

Here  $L(x, x^*)$  is a measure of loss (distance) between two values. This is defined by the user on a case by case basis. Popular definitions are:  $|x - x^*|$ ,  $(x - x^*)^2$  &  $\mathbb{1}\{x \neq x^*\}$

### 3.1 Gradient Descent

#### Definition 3.2 - Gradient Descent

*Gradient Descent* aims to learn a set of weight values  $\mathbf{w}$  which produce a local minimum for a given cost function  $J$ . The update rule for gradient descent is

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

$\nabla J(X; \mathbf{w}_t)$  is the partial derivative of the cost function wrt to the weights and gives the direction of the greatest descent. We can calculate the  $i^{\text{th}}$  component of  $\Delta \mathbf{w}$  after observing  $(\mathbf{x}, f^*(\mathbf{x}))$

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

#### Definition 3.3 - Online Gradient Descent

- i). **initialise** all weights  $W$  randomly.
- ii). **for**  $t = 0, 1, \dots$  **do**:
  - (a) **pick** net training sample  $(x, f^*)$ .
  - (b) **forward-backward pass** to compute  $\nabla J$ .
  - (c) **update** weights  $W \leftarrow W - \eta \nabla J$ .

- (d) **if** (stopping criteria met) **break loop**.
- iii). **return** final weights  $W$ .

**Remark 3.1 - Using Single Samples**

Using single samples to find the minimum point of the cost function will only roughly approximate aspects of the cost function gradient in online mode, leading to a very noisy gradient descent which may not find the global minimum at all.

Thus, it is not good to do online learning. And if the learning rate  $\eta$  is set too *large* then we may overshoot the global minimum. If the learning rate  $\eta$  is set too *small* then we takes a very long time to find a minimum.

**Proposition 3.1 - Using Multiple Samples**

As using a single sample is bad, we try using multiple samples at once and using the average  $\nabla J$ . There are two approaches

- *Deterministic Gradient Descent* (DGD) where all training samples  $(X, F^*)$  are used. Given a small enough learning rate  $\eta$  this will process to the true local minimum, but at high computational cost.
- *Stochastic Gradient Descent* (MiniBatch) where a small subset of training samples  $(X, F^*)$  are used. This is still good at finding a minimum, and much less computationally costly.

For the average of  $\nabla J$  we use

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

**Proposition 3.2 - Setting the Learning Rate  $\eta$**

Setting the learning rate  $\eta$  can be hard so a process called *Simulated Annealing* is used to test out several learning rates.

Let  $\eta_0$  be an initial (high) learning rate and  $\eta_\tau$  be a final (smaller) learning rate. *Simulated Annealing* transitions from  $\eta_0$  to  $\eta_\tau$ .

- i). **initialise** all weights  $W$  randomly.
- ii). **for**  $k = 0, \dots, \tau$  **do**:
  - (a)  $\eta_k := (1 - \frac{k}{\tau})\eta_0 + \frac{k}{\tau}\eta_\tau$
  - (b) **for**  $t = 0, 1, \dots$  **do**:
    - i. **pick** net training sample  $(x, f^*)$ .
    - ii. **forward-backward pass** to compute  $\nabla J$ .
    - iii. **update** weights  $W \leftarrow W - \eta_k \nabla J$ .
    - iv. **if** (stopping criteria met) **break loop**.
  - (c) **return** final weights  $W$ .

### 3.1.1 Auto-Differentiation

**Proposition 3.3 - Calculating Partial Derivatives**

There are three ways to calculate the partial derivatives required for *Gradient Descent*.

- *Symbolic Differentiation* (i.e. algebra). Hard to define to work in all cases.

- *Numerical Differentiation* (i.e. check values in a neighbourhood and approximate the best direction). Easy to implement but low accuracy and high computational cost.
- *Automatic-Differentiation* using feedforward computation graphs. See below

**Definition 3.4 - Feedforward Computational Graph**

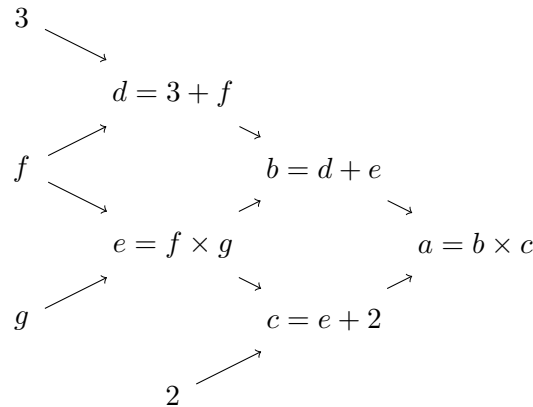
Given a series of equations we can construct a *feedforward computational graph*. *Feedforward computational graphs* have a node for each variable or constant, and then an edge between nodes which are dependent. Once values are defined for all variables at a given depth, values can easily be calculated for variables higher up the tree.

**Example 3.1 - Feedforward Computational Graph**

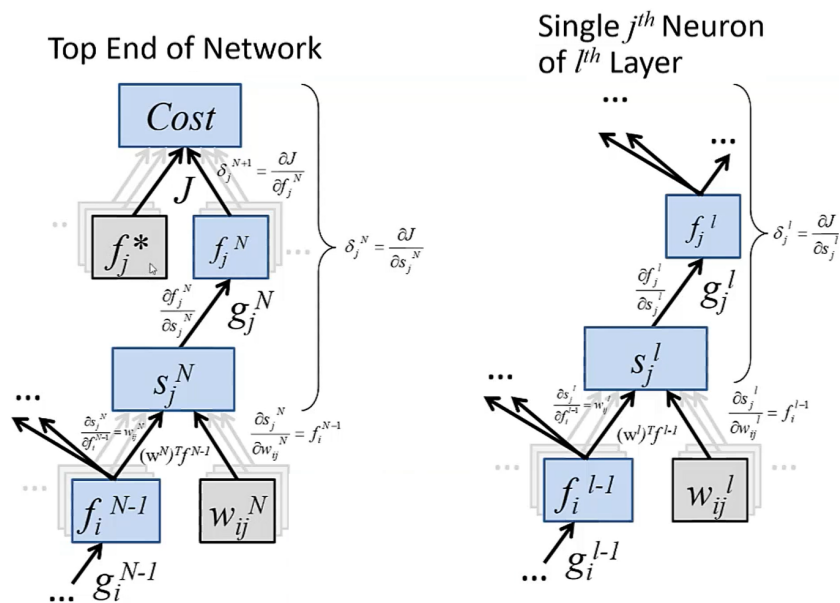
Consider the following series of equations

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

We can construct the following *Computational Graph*



**Proposition 3.4 - Feedforward Computational Graph - Neural Network**



Remember that  $J(\cdot, \cdot)$  is the cost function;  $s_j^l := (w^l)^T f^{l-1}$  is the signal of a layer;  $g_j^l(\cdot)$  is the activation function of a layer;  $f_j^l := g_j^l(s_j^l)$  is the output of the layer;

**Definition 3.5 - Auto-Differentiation using a Feedforward Computational Graph**

Consider two nodes in a computational graph  $x, y$  and suppose you want to find 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  $y \rightarrow a \rightarrow x$  calculate  $\frac{\partial a}{\partial y}, \frac{\partial x}{\partial a}$ ).
- iii). Apply the chain rule along each path (i.e. For  $y \rightarrow a \rightarrow x$  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 3.2 - Auto-Differentiation using a Feedforward Computational Graph**

Consider the graph in **Example 3.1** and wanting to calculate  $\frac{\partial f}{\partial a}$ .

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

	(1)		(2)		(3)
$\frac{\partial d}{\partial f}$	$= 1$	$\frac{\partial e}{\partial f}$	$= g$	$\frac{\partial e}{\partial f}$	$= g$
$\frac{\partial b}{\partial d}$	$= 1$	$\frac{\partial b}{\partial e}$	$= 1$	$\frac{\partial c}{\partial e}$	$= 1$
$\frac{\partial a}{\partial b}$	$= c$	$\frac{\partial a}{\partial b}$	$= c$	$\frac{\partial a}{\partial c}$	$= b$

- iii). Applying the chain rule to each path gives

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

- 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 3.5 - 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 3.2 - Usefulness of Auto-Differentiation**

*Auto-Differentiation* allows us to mathematically 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 computationally hard. (*Hierarchical Dependence* can be used to mitigate this)

## 3.2 Backpropagation Algorithm

**Remark 3.3 - Backpropagation Algorithm - Intuition**

The *Backpropagation Algorithm* combines *reverse auto-differentiation* with *gradient descent*. Reverse auto-differentiation is used to find the relationship between the cost function and each weight; and gradient descent to perform stepwise adjustments on weights.

The *Backpropagation Algorithm* seeks to compute the discrepancy between the network's output and the target value; then propagate this discrepancy backwards through the network to determine the influence of each weight on this discrepancy, by considering the influence of each path.

**Proposition 3.6 - Backpropagation Algorithm - Overall Strategy**

- i). Read the input & perform a forward pass through the network. (This will calculate all  $s_j^l, f_j^l$ .)
- ii). Calculate the cost function between each final layer neuron and its target  $J(f_j^*, f_j^N)$ .
- iii). Calculate the error derivatives  $\delta_j^{N+1}$  of the cost function  $J$  wrt each final layer neuron  $f_j^N$

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

- iv). Compute the error derivative  $\delta_j^N$  of the cost function wrt the signals of the last layer

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

- v). Layer-by-layer calculate the *error derivatives*  $\delta_i^{l-1}$  of the cost function wrt the signal each neuron in the next layer, using the error derivatives  $\delta_j^l$  of the layer above

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

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

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

**Proof 3.1 - Derivation of  $\delta_i^{l-1}$** 

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

**Proposition 3.7 - Backpropagation Algorithm**

- i). **initialise** all weights randomly (typically to small values).
- ii). **for**  $t = 0$  **do**
  - (a) **pick** next training sample  $([f_1^0, f_2^0, \dots], [f_1^*, f_2^*, \dots])$ .
  - (b) **forward pass** compute all layer outputs  $s_j^l := \sum_{i=1}^{d(l-1)} w_{ij}^l f_i^{l-1}$  and  $f_j^l := g_j^l(s_j^l)$ . [i]
  - (c) **compute** derivative of cost function wrt final layer  $\delta_j^N := g_j^{N'}(s_j^N) \cdot \frac{\partial J}{\partial f_j^N}$ . [ii]-iii]
  - (d) **backward pass** compute all deltas  $\delta_i^{l-1} := g_i^{l-1'}(s_i^{l-1}) \sum_{j=1}^{d(l)} w_{ij}^l \delta_j^l$  [iv)-vi]
  - (e) **update** all weights based on deltas and neuron activities  $w_{ij}^l \leftarrow w_{ij}^l - \eta f_i^{l-1} \delta_j^l$  [gradient descent]
  - (f) **if** (stopping criteria met): **break** loop
- iii). **return** final weights  $w_{ij}^l$

**Remark 3.4 - Issues with Backpropagation Algorithm**

The *Backpropagation Algorithm* was known for 30 years before deep learning began. There are a few factors which prevented deep learning starting earlier:

- The *Vanishing Gradient Problem*. Gradients are unstable/noisey when you backpropagate gradients in a very deep network.
- 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.
- The number of parameters explode in deep networks (every node in one layer is connected to every node in the next layer, for all layers!). This can be addressed by sharing parameters (e.g. CNNs) or reuse parameters (e.g. RNNs).
- Regularisation techniques are critical to achieve good generalisation beyond the training data available (avoid overfitting).

**Remark 3.5 - Activation Functions need to be Differentiable & Non-Linear**

For the *Back-Propagation Algorithm* the derivative of each activation function is used, so each activation function must be differentiable.

The step-function does not fulfil this.  $\tanh$  was consider as an alternative, however the gradient of its derivative is vanishingly small on the tails. This causes  $\delta_i^{l-1}$  to become close to 0 (are saturated), significantly slowing down learning of early layers (if any learning occurs at all).

This is addressed (usually) by forwarding signal via a residual neural network (ResNet), or using specially robust neuron layouts.

**Definition 3.6 - Rectifying Linear Unit (ReLU)**

*ReLU* is an activation function which combines high speed of evaluation with a non-saturating non-linear 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 3.6 - Usefulness of ReLU**

Using *ReLU* may reach network convergence 5-10 times faster than using  $\tanh$ .

However, 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.

**3.3 Momentum****Proposition 3.8 - Learning via Momentum**

Learning via *Momentum* is an extension of gradient descent. A velocity term  $v$  for ‘current descent speed’ is introduced. The velocity term defines the step sizes, depending upon how large & how aligned to previous gradients a new gradient is.

Formally we now define weight updates as

$$W_{t+1} = W_t + \underbrace{v_{t+1}}_{\text{momentum}} \quad \text{where} \quad v_{t+1} = \underbrace{\alpha}_{\text{momentum parameter}} \cdot v_t - \eta \nabla J(X; W_t)$$

Momentum can overshoot.

**Proposition 3.9 - Nesterov Accelerated Gradient (NAG)**

*Nesterov Accelerated Gradient* is an extension of *Learning via Momentum*, where instead of calculating the gradient at the current position you look 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

$$W_{t+1} = W_t + v_{t+1} \quad \text{where} \quad v_{t+1} = \alpha v_t - \eta \nabla J(X; \underbrace{W_t + \alpha v_t}_{\text{location perview}})$$

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

**Remark 3.7 - When Momentum Struggles**

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 improving the learning rate.

**Proposition 3.10 - Newton's Method**

*Newton's Method* removes all hyperparameters (inc. *Learning Rate*  $\eta$ ) and instead uses curvature to rescale the gradient, by multiplying the gradient by the inverse Hessian of the current cost function  $H(J(X; W_t))$ . This leads to an optimisation that takes aggressive steps in directions of shallow curvature, and shorter steps in directions of steep curvature.

Formally we now define weight updates as

$$W_{t+1} = W_t - H(J(X; W_t))^{-1} \nabla J(X; W_t)$$

Computing and inverting the Hessian is computationally and space expensive. Newton's method is attracted to saddle points (bad!).

**Remark 3.8 - 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 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 values  $J$  should be saddle points, which we can escape using symmetry-breaking descent methods.

### 3.4 Function Adaptive Optimisation Algorithms

**Definition 3.7 - Adaptive Gradient Algorithm (AdaGrad)**

The *AdaGrad* algorithm keeps track of per-weight learning rates to force evenly spread learning speeds across the weights. This means that weights with a high gradient have their learning rate decreases, whilst those with low gradients have it increased.

Formally we now define weight updates as

$$W_{t+1} = W_t - \eta \frac{\nabla J(X; W_t)}{\sqrt{A_{t+1} + \varepsilon}} \quad \text{where} \quad A_{t+1} = A_t + (\nabla J(X; W_t))^2$$

NOTE perform element-by-element squaring and  $\varepsilon$  is used to avoid division by zero.

$A_t$  is an accumulator vector, which accumulates the changes so far in each dimension.

**Remark 3.9 - Limitations of Monotonic Learning**

*Monotonic Learning* is very aggressive and lacks the possibility of late adjustments, meaning learning usually stops too early.

**Proposition 3.11 - Root-Mean-Square Propagation (RMSProp)**

*RMSProp* combats the aggressive reduction in *AdaGrad*'s learning speed by propagation of a smooth running average, using a smoothing parameter  $\beta$ .

Formally we now define weight updates as

$$W_{t+1} = W_t - \eta \frac{\nabla J(X; W_t)}{\sqrt{A_{t+1} + \varepsilon}} \quad \text{where} \quad A_{t+1} = \beta A_t + (1 - \beta) (\nabla J(X; W_t))^2$$

NOTE perform element-by-element squaring and  $\varepsilon$  is used to avoid division by zero.

**Proposition 3.12 - Adaptive Moment Estimation (AdaM)**

The *AdaM* algorithm is an extension of *RMSProp* with two new additions.

- i). Smoothing *RMSProp's* (usually noisy) incoming gradient by using a new parameter  $\alpha$
- ii). Correcting the impact of bias which is introduced by *initialising* the two smoother measures.

$$\begin{aligned}
 G_{t+1} &= \alpha G_t + (1 - \alpha) \nabla J(X; W_t) \\
 \bar{G} &= \frac{G_{t+1}}{1 - \alpha^t} \\
 A_{t+1} &= \beta A_t + (1 - \beta) [\nabla J(X; W_t)]^2 \\
 \bar{A} &= \frac{A_{t+1}}{1 - \beta^t} \\
 W_{t+1} &= W_t - \eta \frac{\bar{G}_{t+1}}{\sqrt{\bar{A}_{t+1} + \epsilon}}
 \end{aligned}$$

**Remark 3.10 - Using AdaM**

Applying AdaM to a ReLU-based network is sufficient to perform deep learning but there is no guarantee of success for a few reasons

- i). We have hyperparameters  $\alpha, \beta, \epsilon, \dots$  which need to be set
- ii). The size of each mini-batch is not certain.
- iii). How do we initialise the network?
- iv). How do we avoid overfitting? Especially with so many parameters.
- v). Which loss function to use.

Achieving top-end results in deep learning often involves lots of parameter tuning, testing and trial-and-error.

### 3.5 Cost Functions for Classification

**Remark 3.11 - Cost Functions for Classification Problem**

In classification problems it is not obvious how to define the distance between classifications, and thus how to define a cost function. Using the setup of activation functions proposed in **Proposition 2.5** it is common to use the *Cross-Entropy Cost Function*

**Definition 3.8 - Cross-Entropy Cost Function**

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* is defined as

$$\begin{aligned}
 J &= - \sum_{j \in \text{Group}} f_j^* \ln(f_j^N) \\
 \delta_i^N &= \sum_{j \in \text{Group}} \frac{\partial J}{\partial f_j^N} \frac{\partial f_j^N}{\partial s_i^N} \\
 &= f_i^N - f_i^*
 \end{aligned}$$

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*  $\delta + i^N$  which is propagated backwards from layer  $N$ .

**Proposition 3.13 - Derivation of  $\delta_i^N$  for Cross-Entropy Cost Function**

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

where  $f_j^N := \frac{e^{s_j^N}}{\sum_{i \in \text{Group}} e^{s_i^N}}$

## 4 Implementation

### Remark 4.1 - Tuning

When tuning hyper-parameters we want to maximise test accuracy (not training accuracy) and minimise computational load (ie how long the execution takes)

### Proposition 4.1 - Order of Hyper-parameter Tuning

- i). Learning Rate
- ii). Batch Size

### Remark 4.2 - Learning Rate

When tuning the *Learning Rate* there is a trade-off between speed (not too small) and actually converging (not too big).

### Remark 4.3 - Batch Size

*Batch Size* is the number of examples propagated through the network at any one time. The risk with smaller batch sizes is that they do not sufficiently represent the entire dataset, so the parameter updates overfit to that batch. The risk with larger batch sizes result in fewer weight updates as fewer epochs are completed

Larger batch sizes generally make the execution quicker, due to few passes being completed. Smaller batches are better able to utilise a GPU's parallel processing capability.

**Definition 4.1 - Batch Normalisation**

*Batch Normalisation* normalises layer inputs so the distribution of these inputs does not vary while training and adjusting parameters. This speeds up convergence by changing each input distribution to have zero mean and unit variance. The equation used is

$$\hat{x}_{i,c,x,y} = \frac{x_{i,c,x,y} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$$

where  $x_{i,c,x,y}$  is an output from the previous layer,  $\hat{x}_{i,c,x,y}$  is the transformed value which will be used as an input,  $i, c, x, y$  define the batch, channel, width & height position of the input,  $\mu_c$  and  $\sigma_c^2$  are the mean and variance for each training batch;  $\epsilon$  is a small constant for numerical stability (ie when  $\sigma_c^2$  is 0).

*Batch Normalisation* limits the functions a neural network can represent (bad!). To restore this representation power we learn two new parameters per channel  $\gamma_c$  and  $\beta_c$ . Giving

$$\hat{x}_{i,c,x,y} = \gamma_c \cdot \frac{x_{i,c,x,y} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}} + \beta_c$$

*Batch Normalisation* is typically added after each convolution or fully connected layer, and before the activation function.

**Remark 4.4 - Other Ways to Optimise**

Momentum can be added to SGD. This adds another hyper-parameter to be tuned.

## 0 Reference

**Definition 0.1** - *Hessian Matrix*

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