Applied Deep Learning - Notes

Dom Hutchinson

November 21, 2020

Contents

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.

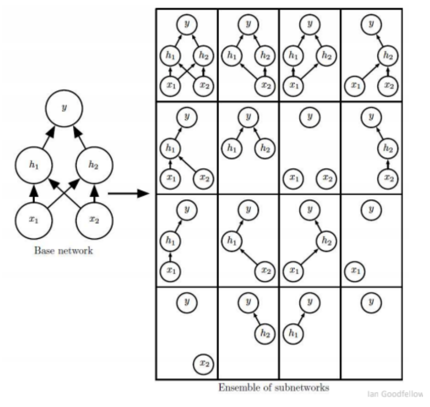
iii). *Axon Terminals & Synapses* = Send information to other neurons.

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.

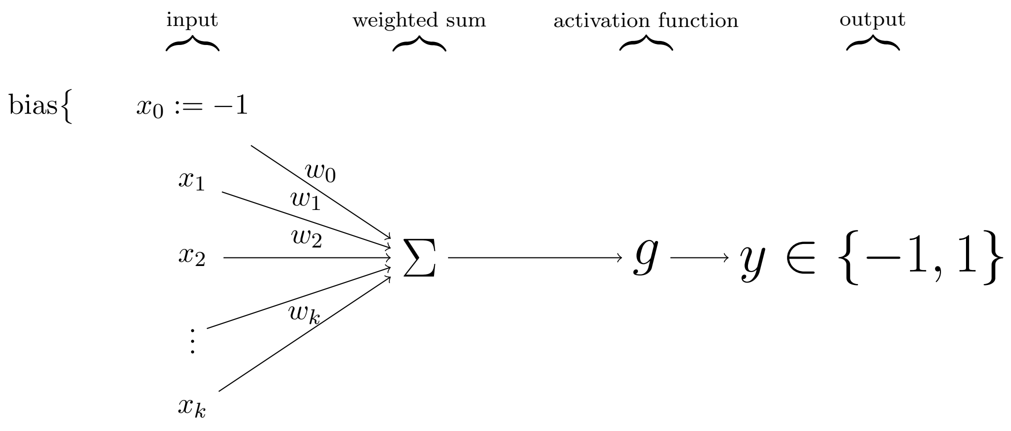
Proposition 2.1 - *Neural Networks as an Ensemble of Sub-Networks*
A *Neural Network* can be considered to represent many *sub-networks.* These sub-networks are switched between depending on which components are picked and how they are defined.



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.

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

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

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 from Σ to $\{-1, 1\}$, effectively performing a binary classification. The user has several options for how to define this. (n.b. $g : \mathbb{R} \to \{-1, 1\}$)

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 2.2 - *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.

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

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 $\boldsymbol{w}$. A simple learning rule is to update the current weights by a certain proportion of the error made.

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

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

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

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

iii). Calculate prediction $f(\boldsymbol{x})$.

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

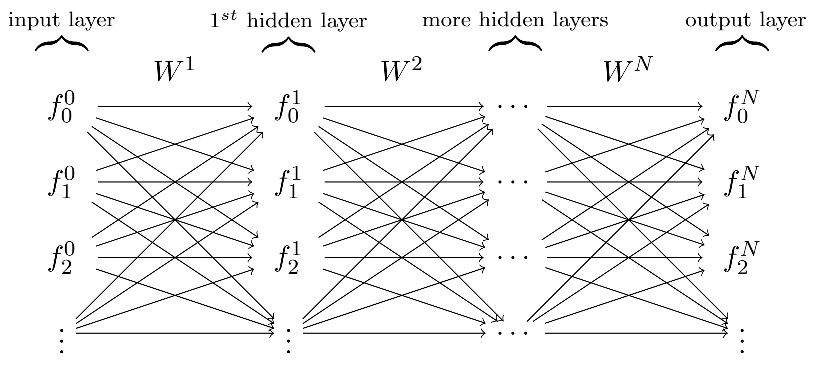
v). Update the weight vector $\boldsymbol{w} = \boldsymbol{w} + \Delta\boldsymbol{w}$ where $\Delta\boldsymbol{w} = \begin{cases} \eta f^*(\boldsymbol{x})\boldsymbol{x} & \text{if } f^*(\boldsymbol{x}) \neq f(\boldsymbol{x}) \\ 0 & \text{otherwise} \end{cases}$

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

2.2 Multi-Layer Perceptron

Remark 2.3 - *Learning Arbitrary Decision Boundaries*
To lean an arbitrary decision boundary (i.e. anything non-linear) can be done by using a *Multi-Layer Preceptron* 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 occuring 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 $\boldsymbol{f}^i, \boldsymbol{f}^j$ (of widths n_i, n_j respectively) there is an associated weight matrix $W \in \mathbb{R}^{n_i \times n_j}$ st $\boldsymbol{f}^j = W^T \boldsymbol{f}^i$. VarThe 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, althought the layer may be exponentially wider than the input.

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

Proposition 2.5 - *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 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

$$\begin{array}{rcl} g'^N_j(s_j^N) & = & f_j^N(1 - f_j^n) \\ g'^N_j(s_{i \neq j}^N) & = & -f_j^N f_i^N \end{array}$$

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

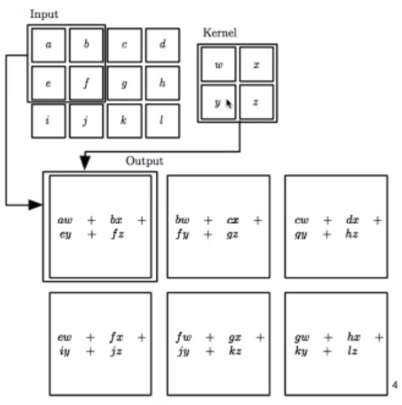
2.3 Types of Deep Neural Networks

2.3.1 Convolution DNN (CNNs)

Definition 2.5 - *Convolution* $*$
The *Convolution* operation takes an input x and a kernel ω. 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 st its the 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 is starts again but one cell lower. This means that $(X * \omega)$ is likely smaller in all dimensions that X.



[4]

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)*
Convoltuon DNN expected 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.

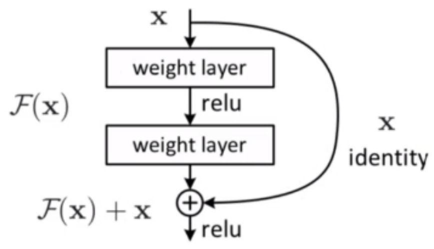
Remark 2.7 - *C*
are needs to be taken when backpropagating CNNs with zero padding or stride greater than 1.

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 mergered 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.


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

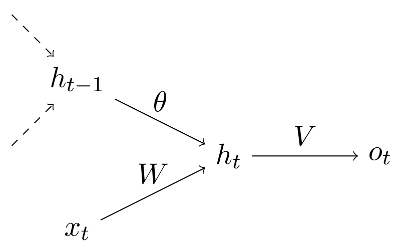


Figure 1: Diagram of how the prediction o_t from step t is calculated. Note that this process Has a *Hidden-to-Hidden* recursive step (ie h_{t-1} to h_t). **Note** if x_t or h_t are multi-dimensional then the arrows are considered as a fully-connected layer. x_t is t^{th} input, h_{t-1} is the hidden value of the previous step, θ is the hidden-to-hidden weights, W is the input-to-hidden weights and V is the hidden-to-output weights.

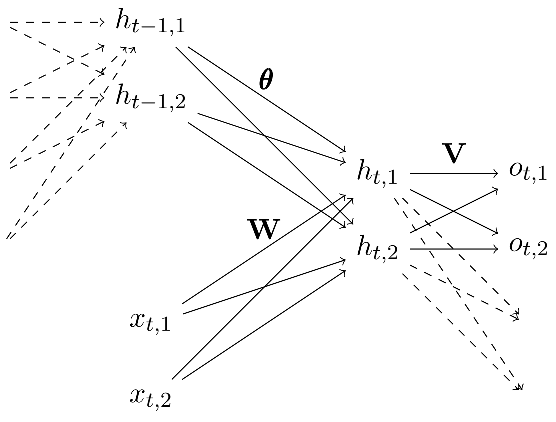


Figure 2: Diagram of an RNN which accepts a sequence $\mathbf{x}_1, \dots, \mathbf{x}_T$ of 2D data (ie $\mathbf{x}_t = (x_{t,1}, x_{t,2})$). $\mathbf{W}, \boldsymbol{\theta}, \mathbf{V} \in \mathbb{R}^{2\times 2}$ are the weight matrices between each pair of data. **Note** that this graph is recurrent with the dashed arrows denoting how $\mathbf{h}_{t-1}$ is receiving inputs from the previous step in the graph, and $\mathbf{h}_t$ is sending its value to the next step of the graph.

2.3.2 Recurrent DNN (RNNs)

Definition 2.8 - *Recurrent DNN (RNN)*
Recurrent DNNs are designed for processing sequential data (similar to how CNNs are designed for grid-like data). Most *RNN* architectures are designed to process sequences of variable length.

Each hidden layer of an RNN acts on a different element in the sequence (ie layer t acts on x_t) and takes the output of the previous layer h_{t-1} as an input too.

Every layer uses the same weight values, and these weights can be considered as three categories: *Input-to-Hidden* W which interact with the inputs to the layer x_t; *Hidden-to-Hidden* $\theta)t$ which interacts with the hidden value of the previous layer hidden layer h_{t-1}; and, *Hidden-to-Output* V when produces the outputted prediction o_t from the calculated hidden value h_t. Traditionally θ denotes the set of all weights (ie $\theta := (\theta, W)$)

$$h_t = f(h_{t-1}, x_t; \theta)$$

To calculate the outputted prediction o_t we apply the *Hidden-to-Output* weights V to the hidden value h_t.

$$o_t = g(h_t; V)$$

`Figure 1` provides an abstract diagram for a layer of a recurrent neural network.

Remark 2.10 - *1D CNN vs RNN*
Here are some differences between a 1D CNN and an RNN

- A 1D CNN allows for parameter sharing across time, but is shallow.

  An RNN shares parameters will all previous members of the output. This means RNNs share parameters through a very deep computation graph.

- In a 1D CNN, the output is a function of neighbouring members of the input.

Remark 2.11 - *Properties of RNNs*
RNNs scale to much longer sequences of data than other networks which are not specialised to sequence-based data.

Proposition 2.10 - *Unrolling the RNN*
We can unroll the function for an RNN over finite time steps

$$\begin{aligned} h_3 &= f(h_2, x_3; \theta) \\ &= f(f(h_1, x_2; \theta), x_3; \theta) \end{aligned}$$

This can be done as the learnt model f and the parameters θ are shared for all temporal steps, and have the same size.

Remark 2.12 - *RNN Models are Lossy*
The function $f(\cdot)$ at time t can be considered as a *lossy summary* of the steps $x_1, \dots, x_{t-1}$ (as the function is on lower dimension than the input data). We can specify which parts of the data to keep or discard when defining the training criteria.

$$\begin{tikzcd}
 & y_{t-1} \arrow[d] & y_t \arrow[d] & y_{t+1} \arrow[d] & \\
 & L & L & L & \\
 & o_{t-1} \arrow[u] & o_t \arrow[u] & o_{t+1} \arrow[u] & \\
{} \arrow[r, dashed] & h_{t-1} \arrow[u, "V"'] \arrow[r, "\theta"] & h_t \arrow[u, "V"'] \arrow[r, "\theta"] & h_{t+1} \arrow[u, "V"'] \arrow[r, dashed] & {} \\
 & x_{t-1} \arrow[u, "W"'] & x_t \arrow[u, "W"'] & x_{t+1} \arrow[u, "W"'] &
\end{tikzcd}$$

Figure 3: Diagram of relationship between data (x, y) and the loss function L, for an RNN which makes predictions are every time step and has a hidden-to-hidden recursive relationship between steps (h_{t-1} to h_t).

Proposition 2.11 - *Types of RNN*
There are several types of *RNN*, these can be summarised as below

	Predictions	**Recursive Relationship**	**Figure**
I	Every Step	Hidden-to-Hidden	`Figure 3`
II	Every Step	Output-to-Hidden	`Figure 4`
III	Single Step	Hidden-to-Hidden	`Figure 5`

$$\begin{tikzcd}
& y_{t-1} \arrow[d] & y_t \arrow[d] & y_{t+1} \arrow[d] & \\
& L & L & L & \\
{} \arrow[dr, dashed] & o_{t-1} \arrow[u] \arrow[dr, "\theta"] & o_t \arrow[u] \arrow[dr, "\theta"] & o_{t+1} \arrow[u] \arrow[dr, dashed] & \\
& h_{t-1} \arrow[u, "V"'] & h_t \arrow[u, "V"'] & h_{t+1} \arrow[u, "V"'] & {} \\
& x_{t-1} \arrow[u, "W"'] & x_t \arrow[u, "W"'] & x_{t+1} \arrow[u, "W"'] &
\end{tikzcd}$$

Figure 4: Diagram of relationship between data (x, y) and the loss function L, for an RNN which makes predictions are every time step and has an output-to-hidden recursive relationship between steps (o_{t-1} to h_t).

$$\begin{tikzcd}
& & & y_{t+1} \arrow[d] & \\
& & & L & \\
& & & o_{t+1} \arrow[u] & \\
{} \arrow[r, dashed] & h_{t-1} \arrow[r, "\theta"] & h_t \arrow[r, "\theta"] & h_{t+1} \arrow[u, "V"'] \arrow[r, dashed] & {} \\
& x_{t-1} \arrow[u, "W"'] & x_t \arrow[u, "W"'] & x_{t+1} \arrow[u, "W"'] &
\end{tikzcd}$$

Figure 5: Diagram of relationship between data (x, y) and the loss function L, for an RNN which only makes a prediction at time step $t + 1$ and has a hidden-to-hidden recursive relationship between steps (h_{t-1} to h_t).

Type I) is the most common as it is computing complete. Type II) is less powerful as it only the information captured in the prediction o_t is passed to future steps (and o_t tends to be low-dimensional). Type III) is only used to produced summaries (e.g. classify a sentence from its constituent words).

Training an RNN

Proposition 2.12 - *Training a Type I) RNN*
Consider the Type I) RNN defined in `Proposition 2.12`. The value for each hidden value h_t and predicted value o_t can be considered as the following functions

$$\begin{aligned} h_t &= g(\theta h_{t-1} + W x_t + b) \\ o_t &= g(V h_t + c) \end{aligned}$$

where g are activation functions and $b, c \in \mathbb{R}$ are bias terms. The total loss is the sum of all losses at all time steps.

$$L := \sum_{t \in T} L(o_t, y_t)$$

To calculate the gradient of L the generalised back-propagation algorithm is applied to the unrolled network. (The parameters are V, θ, W, b, c and the nodes are x_t, h_t, o_t, L_t for all $t \in T$).

This gives us gradients

$$\begin{aligned}\frac{\partial L}{\partial L(o_t,y_t)} &= 1\\ \left[\frac{\partial L}{\partial o_t}\right]_i &= \left[\frac{\partial L}{\partial L(o_t,y_t)}\frac{\partial L(o_t,y_t)}{\partial o_t}\right]_i = [o_t]_i - [y_t]_i\end{aligned}$$

$\frac{\partial L}{\partial o_t}$ is calculated using softmax and cross-entropy. To work out the gradients wrt each prediction o_t we start at the end of the sequence (o_T, y_T) and towards (o_1, y_1).

Due to unrolling the model, the instances of the weights W, θ, V at each time step are treated separately $(W_1, \theta_1, V_1, \dots, W_T, \theta_T, V_T)$ when calculating gradients. Thus, once the partial derivatives have been calculated for each of these instances we merge them to get the gradient wrt each of W, θ, V by taking averages.

Proposition 2.13 - *Gradients for RNN - Type 1*
Let T be the size of the sequence $(x_1, \dots, x_T)$ (not transpose).

$$\begin{aligned}\nabla_c L &= \sum_{t=1}^T \nabla_{o_t} L\\ \nabla_c L &= \sum_{t=1}^T \left(\frac{\partial h_t}{\partial b_t}\right)^T \nabla_{h_t} L\\ \nabla_V L &= \sum_{t=1}^T \sum_i \left(\frac{\partial L}{\partial o_t}\right)^T \nabla_{V_t}[o_t]_i &= \sum_t (\nabla_{o_t} L) h_t^T\\ \nabla_W L &= \sum_{t=1}^T \sum_i \left(\frac{\partial L}{\partial h_t}\right)^T \nabla_{W_t}[h_t]_i &= \sum_t \text{diag}(1-h_t^2)(\nabla_{h_t} L) h_{t-1}^T\\ \nabla_U L &= \sum_{t=1}^T \sum_i \left(\frac{\partial L}{\partial h_t}\right)^T \nabla_{U_t}[h_t]_i &= \sum_t \text{diag}(1-h_t^2)(\nabla_{h_t} L) x_t^T\end{aligned}$$

Remark 2.13 - *Computing the Gradient in an RNN is expensive*
Computing the gradient of L in an RNN is very expensive as it requires performing a forward propagation pass of the unrolled network (even though the weights V, W, θ at each time step are shared they are considered as separated when unrolling), followed by a back-propagation pass through time (BPTT). The run-time of this cannot be reduced by parallelisation as the forward pass is sequential (and, all values computed n the forward pass need to be stored for reuse by the backward pass). The run-time and memory cost are both $O(T)$.

More Types of RNN

Definition 2.9 - *Bi-Directional RNN*
Bi-Directional RNN are able to use information from the past and the future. Effectively, a *Bi-Directional RNN* is actually two RNNs: one moves forward through the sequence; the other moves backwards through the sequence, both have their own weight sets. These two RNNs are joined to make the final decision by the prediction step (ie calculating o_t) takes into account the hidden values of both RNNs (for the same input). Suppose the forward direction RNN has hidden values $h_1, \dots, h_T$ and the backwards direction RNN has hidden values $k_1, \dots, k_T$, then

$$o_t = g(V_h h_t + V_k k_t + b)$$

where V_h and V_k are the *Hidden-to-Output* weights for the forward and backwards RNN respectively.

Bi-Directional RNN are very popular in speech recognition and translation

Definition 2.10 - *Encoder-Decoder RNN*
An *Encoder-Decoder RNN* map from one variable-length sequence to another variable-length sequence. This is useful in translation where a sentence in one language can have more words than in another.

An *Encoder-Decoder RNN* reads in all the information at once and "encodes" it into a fixed sized vector c (The *context*). This c is then used to produce an output (ie decode) and keeps doing so until a stopping criteria is reached. At each time step t whilst decoding, the previously produced outputs $x_1, \dots, x_{t-1}$ and the context c are known and used as inputs.

Remark 2.14 - *Length of Dependecies*
Bi-Directional and *Encoder-Decoder* RNNs are goof at learning short-term dependencies in variable-length sequences. However, gradients propagated over long periods tend to either vanish or explode! This persists even after using mitigation techniques, such as limiting the parameter space.

A *Gated RNN* attempts to fix this issue.

Definition 2.11 - *Gated RNN*
Gated RNNs creates several possible paths through the input sequence. This allows for the network to *accumulate* information over a long duration and allows the network to *forget* (gate) old states when needed. Practically, this is achieved by having connection weights which vary at each time step.

Long Short-Term Memory (LSTMs) are the most popular form of *Gated RNN.*

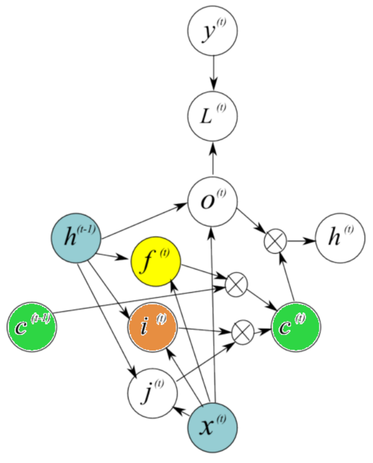


Figure 6: A layer of an LSTM

Proposition 2.14 - *Features of an LSTM*
`Figure 6` provides part of an LSTM. Here are some features of it

- *Forget Gate* (Yellow) - The function $\mathbf{f}_t(x_t, h_{t-1}; \cdot)$ is known as the *Forget Gate* as it encodes the ability to forget the past. It calculates a value between $[0, 1]$ which weights whether to forget what has been learnt upto that point.

$$\begin{aligned}[f_t]_i &= \sigma\left([b_{\mathbf{f}}]_i + \textstyle\sum_j [U_{\mathbf{f}}]_{i,j}[x_t]_j + \sum_j [W_{\mathbf{f}}]_{i,j}[h_{t-1}]_j\right) \\ \mathbf{f}_t &= \mathbf{b_f} + U_{\mathbf{f}}\mathbf{x}_t + W_{\mathbf{f}}\mathbf{h}_{t-1}\end{aligned}$$

where $\sigma(\cdot)$ is a sigmoid function, $U_{\mathbf{f}}$ is the *input-to-hidden* weights learnt for $\mathbf{f}$ and $W_{\mathbf{f}}$ is the *hidden-to-hidden* weights for $\mathbf{f}$.

- *External Input Gate* (Orange)- The function $\mathbf{i}_t(x_t, h_{t-1}; \cdot)$ is used to gate another input. It is computed in the same was as the *Forget Gate* (using a sigmoid)

$$\mathbf{i}_t = \mathbf{b_i} + U_{\mathbf{i}}\mathbf{x}_t + W_{\mathbf{i}}\mathbf{h}_{t-1}$$

  where $U_{\mathbf{i}}$ is the *input-to-hidden* weights learnt for $\mathbf{i}$ and $W_{\mathbf{i}}$ is the *hidden-to-hidden* weights for $\mathbf{i}$.

- *State Gate* (Green) - The function $\mathbf{c}_t(\mathbf{f}, \mathbf{i}, \mathbf{j}, \mathbf{c}_{t-1})$ is a *State Gate* which updates the *memory* of the LSTM.

$$\mathbf{c}_t = \mathbf{f}_i \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \sigma\left(\mathbf{b}_j + U_j\mathbf{x}_t + W_j\mathbf{h}_{t-1}\right)$$

  where $\circ$ is element-wise multiplication

- *Output Gate* - The function $\mathbf{o}_t(\mathbf{x}_t, \mathbf{h}_{t-1}; \cdot)$ calculates the about using the previous hidden value (before any forgetting occurs).

$$\mathbf{o}_t = \sigma\left(\mathbf{b}_o + U_o\mathbf{x}_t + W_o\mathbf{h}_{t-1}\right)$$

- *Hidden State* - The hidden state of this layer h_t is calculated by updating the value of the *Output Gate* $\mathbf{o}_t$ with the memory $\mathbf{c}_t$

$$\mathbf{h}_t = \tanh(\mathbf{c}_t) \circ \mathbf{o}_t$$

Remark 2.15 - *Simpler LSTMs*
GRUs are simpler versions of LSTMs which are deemed to be just as powerful. A GRU just have an *Update Gate*, which gates future information, and a *Reset Gate*, which control which parts of the state get used. A single equation can be used to calculate the hidden value for a GRU.

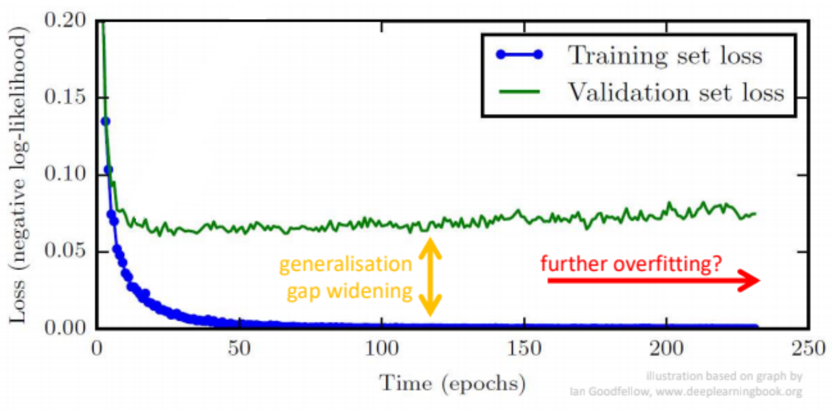
Remark 2.16 - *For Revision*
Focus on the notion of RNNs; and the power & use-cases of the different flavours. Don't worry about derivatives. See chapter 10.

2.4 Problems

2.4.1 Overfitting

Proposition 2.15 - *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 2.16 - *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 2.17 - *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 2.17 - *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 2.12 - *Regularisation*
A *Regularisation* is any modification made to a learning algorithm which is intended to reduced its *generalisation error*, <u>but</u> not its *training error*.

Definition 2.13 - *L-Regularisation*
L-Regularisation constrains the weight space (i.e. makes certain weight values more likely to be learned by the system).

Definition 2.14 - *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 2.15 - *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 2.16 - *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 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 2.17 - *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 a more fine-grained approach to ensemble learning than *Dropout*.

2.5 Usefulness

Remark 2.18 - *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 $\to$ 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 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 2.18 - *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 2.19 - *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

$$\begin{aligned}\text{Expected Loss}\quad J(X;\boldsymbol{w}) &= \mathbb{E}[L(f(\boldsymbol{x},\boldsymbol{w}),f^*(\boldsymbol{x}))]\\ \text{Empirical Risk}\quad J(X;\boldsymbol{w}) &= \frac{1}{|X|}\sum_{\boldsymbol{x}\in X}L(f(\boldsymbol{x},\boldsymbol{w}),f^*(\boldsymbol{x}))\end{aligned}$$

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: $|x-x^*|$, $(x-x^*)^2$ & $\mathbb{1}\{x=x^*\}$

3.1 Gradient Descent

Definition 3.2 - *Gradient Descent*
Gradient Descent aims to learn 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

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

$\nabla J(X;\boldsymbol{w}_t)$ 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^{th} component of $\Delta\boldsymbol{w}$ after observing $(\boldsymbol{x},f^*(\boldsymbol{x}))$

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

Definition 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 η is set too *large* then we may overshoot the global minimum. If the learning rate η 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 ∇J. There are two approaches

- *Deterministic Gradient Descent* (DGD) where all training samples (X, F^*) are used. 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 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 ∇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 η*
Setting the learning rate η can be hard so a process called *Simulated Annealing* is used to test out several learning rates.

Let η_0 be an initial (high) learning rate and η_τ be a final (smaller) learning rate. *Simulated 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 $t = 0, 1, \dots$ 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.

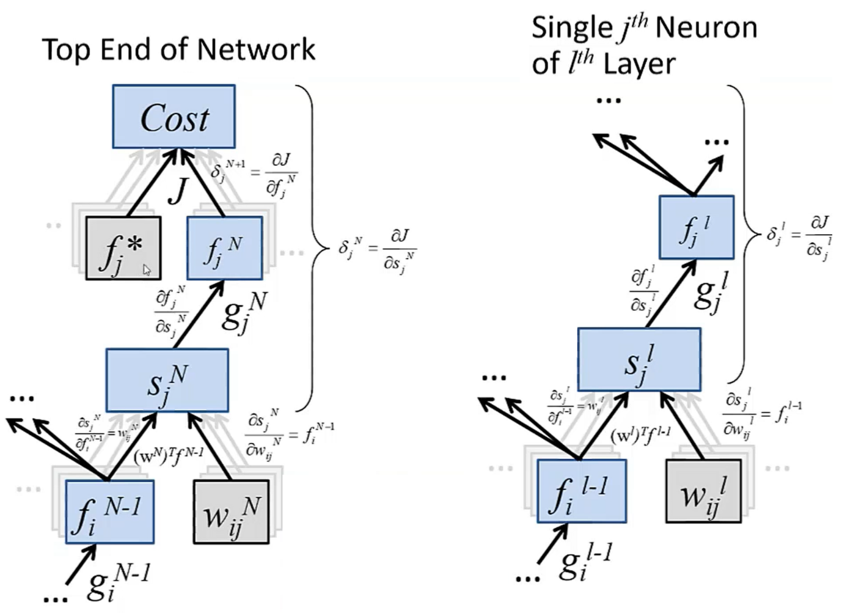
Example 3.1 - *Feedforward Computational Graph*
Consider the following series of equations

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

We can construct the following *Computational Graph*

$$\begin{tikzcd}
3 \arrow[dr] & & & \\
& d=3+f \arrow[dr] & & \\
f \arrow[ur] \arrow[dr] & & b=d+e \arrow[dr] & \\
& e=f\times g \arrow[ur] \arrow[dr] & & a=b\times c \\
g \arrow[ur] & & c=e+2 \arrow[ur] & \\
& 2 \arrow[ur] & &
\end{tikzcd}$$

Proposition 3.4 - *Feedforward Computational Graph - Neural Network*


Top End of Network
Single j^{th} Neuron of l^{th} Layer
Cost
J
$\delta_j^{N+1} = \frac{\partial J}{\partial f_j^N}$
f_j^*
f_j^N
$\delta_j^N = \frac{\partial J}{\partial s_j^N}$
$\frac{\partial f_j^N}{\partial s_j^N}$
g_j^N
s_j^N
$\frac{\partial s_j^N}{\partial f_i^{N-1}} = w_{ij}^N$
$(w^N)^T f^{N-1}$
$\frac{\partial s_j^N}{\partial w_{ij}^N} = f_i^{N-1}$
f_i^{N-1}
w_{ij}^N
g_i^{N-1}
f_j^l
$\delta_j^l = \frac{\partial J}{\partial s_j^l}$
$\frac{\partial f_j^l}{\partial s_j^l}$
g_j^l
s_j^l
$\frac{\partial s_j^l}{\partial f_i^{l-1}} = w_{ij}^l$
$(w^l)^T f^{l-1}$
$\frac{\partial s_j^l}{\partial w_{ij}^l} = f_i^{l-1}$
f_i^{l-1}
w_{ij}^l
g_i^{l-1}

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 $y \to a \to x$ 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 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) $f \to d \to b \to a$; (2) $f \to e \to b \to a$; and, (3) $f \to e \to c \to a$.

ii). We need to calculate the following 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).

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

iii). Applying the chain rule to each path gives

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

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 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 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)

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 δ_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 δ_j^N of the cost function wrt the signals of the last layer

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

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

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

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

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

Proof 3.1 - *Derivation of δ_i^{l-1}*

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

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 $\delta_j^N := g_j^{N\prime}(s_j^N)\cdot\frac{\partial J}{\partial f_j^N}$. [ii)-iii)]

  (d) `backward pass` compute all deltas $\delta_i^{l-1} := g_i^{l-1\prime}(s_i^{l-1})\sum_{j=1}^{d(l)} w_{ij}^l\delta_j^l$ [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 δ_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

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

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 introduce. 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

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

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 lookahead 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

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

NAG is consistently better that *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* η) 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

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

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

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

NOTE perform element-by-element squaring and ε 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 β.

Formally we now define weight updates as

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

NOTE perform element-by-element squaring and ε 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 α

ii). Correcting the impact of bias which is introuced by *initialising* the two smoother measures.

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

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 $\alpha, \beta, \varepsilon, \dots$ 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 define as

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

The steepness of the cost function derivative $\frac{\partial J}{\partial f_j^N}$ <u>exactly</u> 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 δ_i^N for Cross-Entropy Cost Function*

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

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

4 Implementation

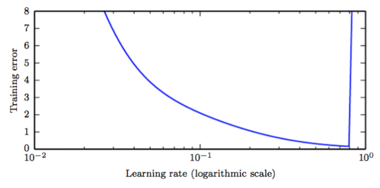
4.1 Training

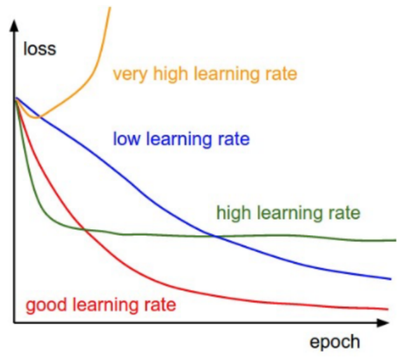
Remark 4.1 - *This Section*
In this subsection I discuss some ways to avoid overfitting during training.

Remark 4.2 - *Tuning*
When tuning hyper-parameters we are seeking to understand the relationship between the value of the hyperparameter and the following quantities

- Training Accuracy & Training Loss;
- Testing Accuracy; and,
- Computational Resources (e.g. time and memory)

Remark 4.3 - *Learning Rate is the most important hyper-parameter to tune*
When tuning the *Learning Rate* there is a trade-off between speed (not too small) and actually converging (not too big).





The plot on the left shows a typical relationship between *Learning Rate* value and training accuracy. The plot on the right shows how different learning rates affect the training error after each training epoch.

Batch Normalisation allows for greater learning rates to be used.

Remark 4.4 - *Batch Size*
Batch Size is the number of examples propagated through the network at any one time, with members of each batch chosen randomly.

Smaller batches are better able to utilise a GPU's parallel processing capability. The risk with smaller batch sizes is that they do not sufficiently represent the gradient of the entire dataset, so the parameter updates overfit to that batch. Further, very small batches make the gradient too noisy to be useful.

Larger batch sizes generally make the execution quicker, due to few passes being completed. The risk with larger batch sizes is they result in fewer weight updates as fewer epochs are completed, and the increases in training accuracy are not linear with the size.

Batch size is often limited by the GPU's memory capacity and typically are powers of 2. For small-sized data batches of 32-256 are common, for large-sized data batches of 8-16 are common.

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 the input to the 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, μ_c and σ_c^2 are the mean and variance for each training batch; ϵ is a small constant for numerical stability (ie when σ_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 γ_c and β_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 allows for greater learning rates to be used. *Batch Normalisation* is typically added after each convolution or fully connected layer, and before the activation function. *Batch Normalisation* should not be applied to the output layer for classification.

Remark 4.5 - *Parameter Initialisation*
Since the cost function of a DNN is non-convex, optimisation algorithms on the cost function depend on the initial parameters.

Random initialisations are easy, but in practice it is better to start with a pre-trained model from a relevant problem which has larger data sets. (For images *ImageNet* is a common pre-trained model.)

Remark 4.6 - *Other Ways to Optimise*
Momentum can be added to SGD. This adds another hyper-parameter to be tuned.

Proposition 4.1 - *Per-Class Accuracy*
Per-Class Accuracy is an extension of the *Accuracy* measurement which calculates the proportion of correct classifications for each class.

$$\text{Acc}_C = \frac{\sum_i [\hat{y}_i == y_i == C]}{\sum_i [y_i == C]}$$

Remark 4.7 - *Dropout*
Dropout is a *regularisation* technique which prevents overfitting by randomly removing inputs to a layer. This means the layer has to rely on a small combination of inputs for detection. *Dropout* decreases training accuracy but should help the model generalise.

Without any *dropout* it is possible for a model to perfectly classify the training data. Generally $p = .5$ is a good balance.

Proposition 4.2 - *Model Checkpointing*
Model Checkpointing is the process of periodically saving model parameters so that if the script crashes then training can be resumed. Further, we can evaluate the trained network separately from the training process itself. Note that a *Checkpoint* typically only contains the parameter

values and does not have a description of the computation the model is performing (thus it is only useful with the source code of the model being saved).

Checkpointing is particularly important in models which take a long time to train.

4.2 Data Augmentation

Proposition 4.3 - *Invariant Transformation*
The following is a table of transformation which can be applied to data for different problem types without affect the training label

Problem	**Invariant To**
Object Recognition	Translation, Rotation, Scaling/Cropping, Viewpoint
Number Plate Recognition	Translation, Rotation
Action Recognition	Translation, Rotation, Scaling/Cropping, Viewpoint, Speed

Other common invariances include: Random Noise, Occulusion, Lighting Conditions, Colour Variations.

Definition 4.2 - *Data Augmentation*
Data Augmentation is the process of applying transform to existing data in order to increase the size of the training validation set. The transformation is done in such a way that the training label is invariant (ie it is not invalidated). This can be done either *online* or *offline*.

Offline Augmentation involves transforming the data and saving it before then training on the larger data set. *Offline Augmentation* is typically used when transformation operations are expensive.

Online Augmentation involves starting the algorithm with unaltered data, but when each new training sample is read in a transformation is applied with some given probability. *Online Augmentation* leads to greater diversity as a sample is likely to be different each time it is used.

It is common to chain transformations

Proposition 4.4 - *Popular Augmentations*
See `torchvision.transforms`

- *Flipping/Rotating* - An image can be flipped or rotated on either of its axes and keep its training label.
- *Brightness* - Making an image uniformly brighter or darker. There can be an issue if values are clipped at 0 or 255.
- *Scaling/Cropping* - Removing parts of the image. Be careful to not cut the object out of the image.
- *Padding* - Pad the image with random noise. Learns to classify regardless of position.
- *Viewpoint* - Minor geometric transformation to an object

0 Reference

Definition 0.1 - *Hessian Matrix*

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