Applied Deep Learning - Notes

Dom Hutchinson

October 15, 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.

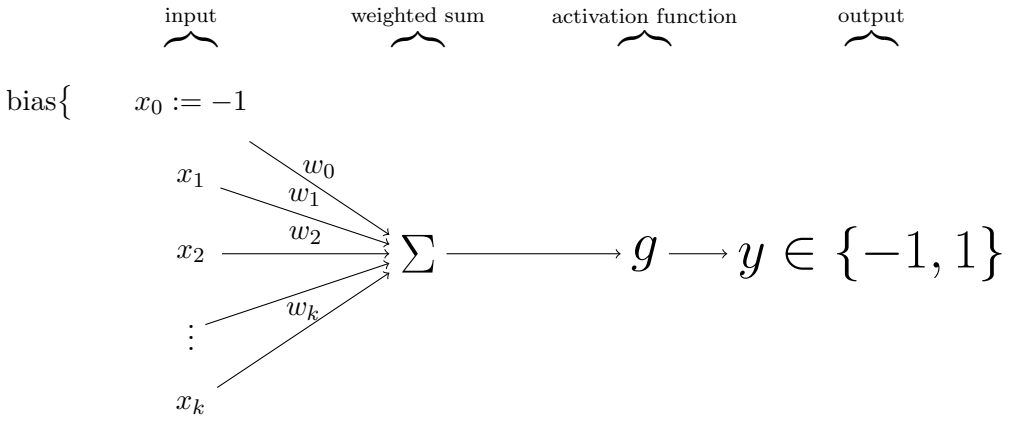
Artificial Neural Networks seek to mimic this structure.

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.

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 2.1 - *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 2.2 - *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.

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

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

Proposition 2.3 - *Training Process for a Single-Layer Perceptron*
Let $\big\{\big(\boldsymbol{x}_1, f^*(\boldsymbol{x}_1)\big), \dots, \big(\boldsymbol{x}_N, f^*(\boldsymbol{x}_N)\big)\big\}$ 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 $\big(\boldsymbol{x}_i, f^*(\boldsymbol{x}_i)\big)$.

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

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

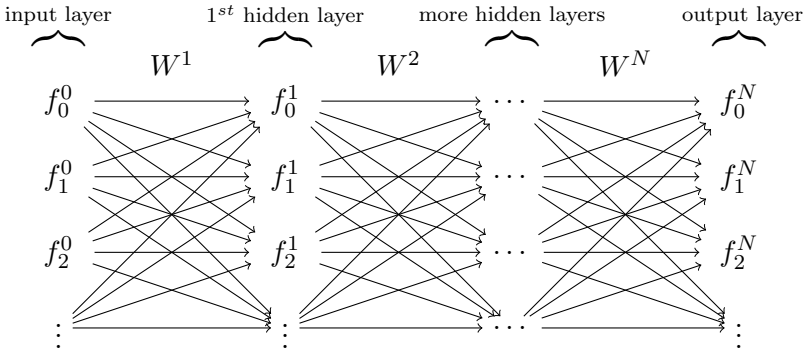
v). Update the weight vector $\boldsymbol{w} = \boldsymbol{w} + \Delta \boldsymbol{w}$ where $\Delta \boldsymbol{w} = \begin{cases} \eta f^*(\boldsymbol{x}) \boldsymbol{x} & \text{if } f^*(\boldsymbol{x}) \neq f(\boldsymbol{x}) \\ 0 & \text{otherwise} \end{cases}$

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 2.4 - *MLPs as Computation Graphs*

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

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

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

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

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` $t = 0, 1, \ldots$ `do`:

(a) `pick` net training sample (x, f^*).

(b) `forward-backward pass` to compute ∇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 <u>all</u> 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 <u>small subset</u> 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) $\eta_k := \big(1 - \frac{k}{\tau}\big)\eta_0 + \frac{k}{\tau}\eta_\tau$

(b) `for` $t = 0, 1, \dots$ `do`:

i. `pick` net training sample (x, f^*).

ii. `forward-backward pass` to compute ∇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-Differentitation

Proposition 3.3 - *Calculating Partial Derivatives*
There are three ways to calculate the partial derivatives required for *Gradient Descent.*

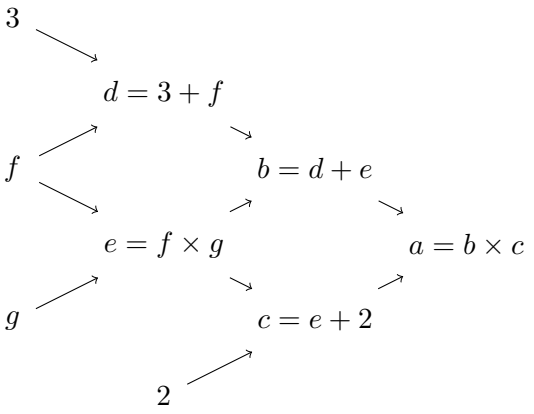
- *Symbolic Differentiation* (i.e. algebra). Hard to define to work in all cases.
- *Numerical Differentiation* (i.e. check values in a neighbourhood and approximate the best direction). Easy to implement but low accuracy and high computational cost.
- *Automatic-Differentiation* using feedforward computation 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.

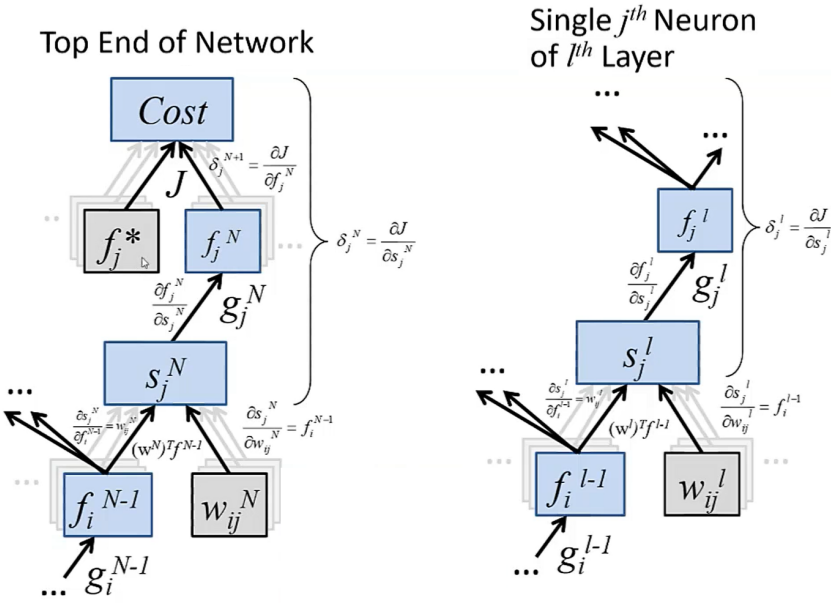
Example 3.1 - *Feedforward Computational Graph*
Consider the following series of equations

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

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 $y \to a \to x$ 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).

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

iii). Applying the chain rule to each path gives

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

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}

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

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

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

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 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 $\alpha, \beta, \varepsilon, \ldots$ 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.

0 Reference

Definition 0.1 - *Hessian Matrix*

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