Machine Learning

Neural Networks

Dariush Hosseini

dariush.hosseini@ucl.ac.uk
Department of Computer Science
University College London

Lecture Overview

Lecture Overview[1]

By the end of this lecture you should:

1. Be familiar with **Neural Networks** and be aware of some of their beneficial properties and empirical successes
2. Understand the **backpropagation** approach to optimisation of neural networks
3. Be aware of the high **capacity** of neural networks, their tendency to **overfit** and some practical remedies for this problem

[1] This lecture is based in large part on Roger Grosse's excellent 'Introduction to Neural Networks' notes

Lecture Overview

Examples

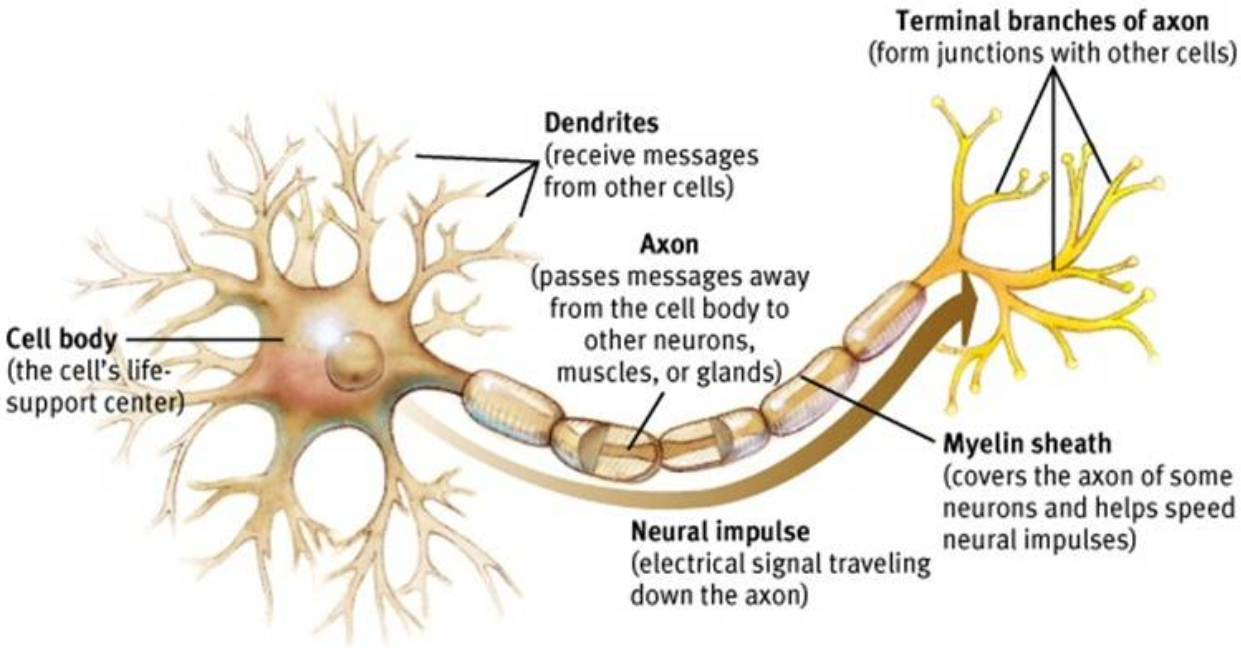
- **Neural Networks** have exhibited great empirical success, and great popular and commercial attention in recent years, for example:
  - **Object Recognition**
    Neural Networks achieved superhuman performance in 2011
    Now used for tasks which were impossible only a few years ago
  - **Speech Recognition**
    All major speech recognition systems are now based on neural networks: Alexa, Siri, Cortana...
  - **Medicine**
    Moorfields Eye Hospital / DeepMind partnership build AI system that matches world-leading experts in diagnosing a range of eye conditions, recommending the correct referral decision for over 50 eye diseases with 94% accuracy (De Fauw et al., 2018).

The Biological Neuron

- **Neural Networks** started life as an attempt to model what goes on in the brain
- The **neuron** is the basic unit of processing in the brain
- It receives chemical signals from other neurons at junctions: the **synapses**
- It converts these signals into electrical signals, combines them, and if the combination is strong enough, generates an **action potential**: an electrical signal that travels along the neuron's **axon**

The Biological Neuron

- The Action potential is binary: either on or off
- This signal then causes the release of chemical signals at synapses with other neurons...



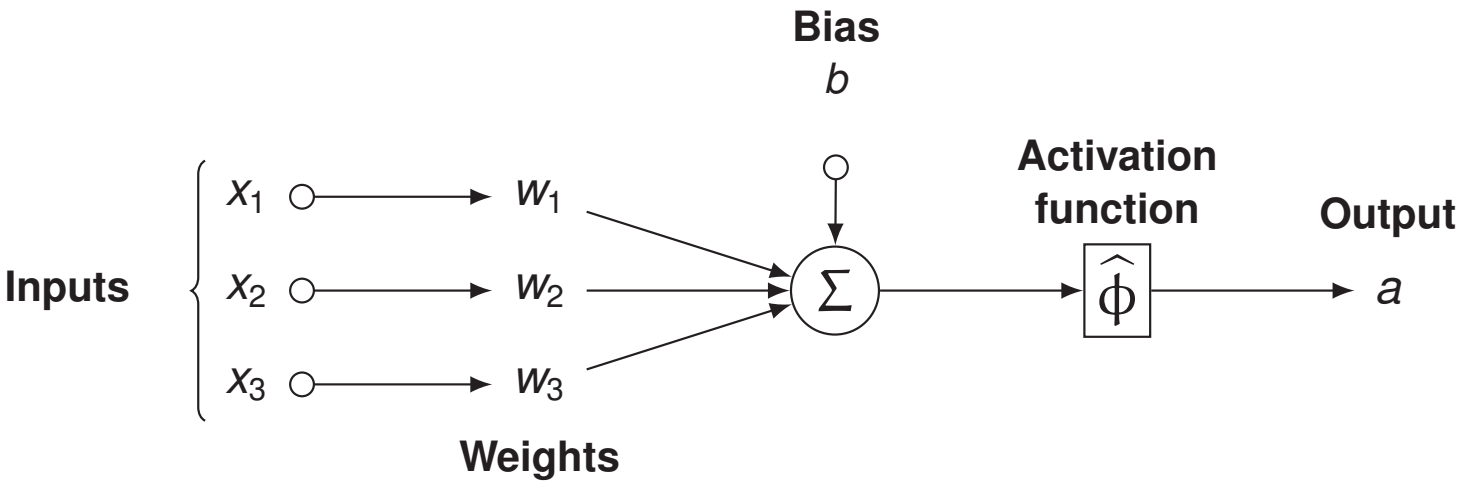
The Artificial Neuron

- An **artificial model** can be viewed as a model of a biological neuron(!)
- It takes in inputs, $\mathbf{x} \in \mathbb{R}^m$, then:
  - Combines them into a weighted sum (and adds a bias term):

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

  - Passes this sum through an activation function, $\widehat{\phi}$
  - Then outputs, $a = \widehat{\phi}(z)$

The Artificial Neuron



- What form does $\widehat{\phi}$ take?
  - The Heaviside function would look most biological...
  - But a common (better behaved) approximation is the logistic sigmoid:

$$\widehat{\phi}(z) = \frac{1}{1 + e^{-z}}$$

  - (Although there are many other forms...)

Artificial Neural Network

- An **Artificial Neural Network** (ANN or NN) is a combination of many of these units
- The units may vary across the network - weights and activation functions might be different
- How the units are connected may be complex and is characterised by the **architecture** of the NN

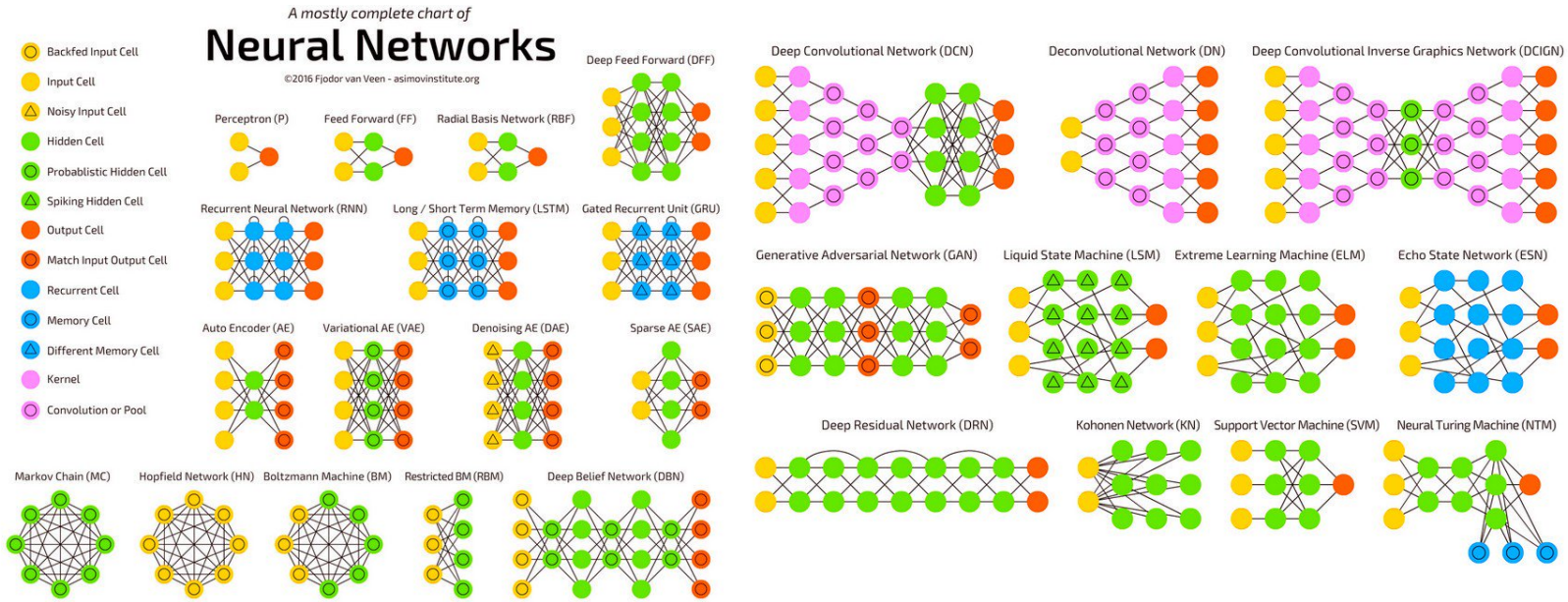
Artificial Neural Network

- Two broad categories are:
  - **Feed Forward Neural Networks** (FFN)
    Where the units are arranged into a graph without any cycles
  - **Recurrent Neural Networks** (RNN)
    Where the graph can have cycles so that the outputs of a unit can loop back to become one of its own inputs

Architecture

- The architecture of a NN can encourage it to 'perceive' different features
- For example a **Convolutional Neural Network** (CNN) exploits the local structure of input data
  - It is particularly effective when applied to datasets such as images where local stucture - edges, sub-images, etc. - are manifest
- Similarly a **Recurrent Neural Network** (RNN) is well placed to perceive time series structure
- How many architectural models are there?

Architecture



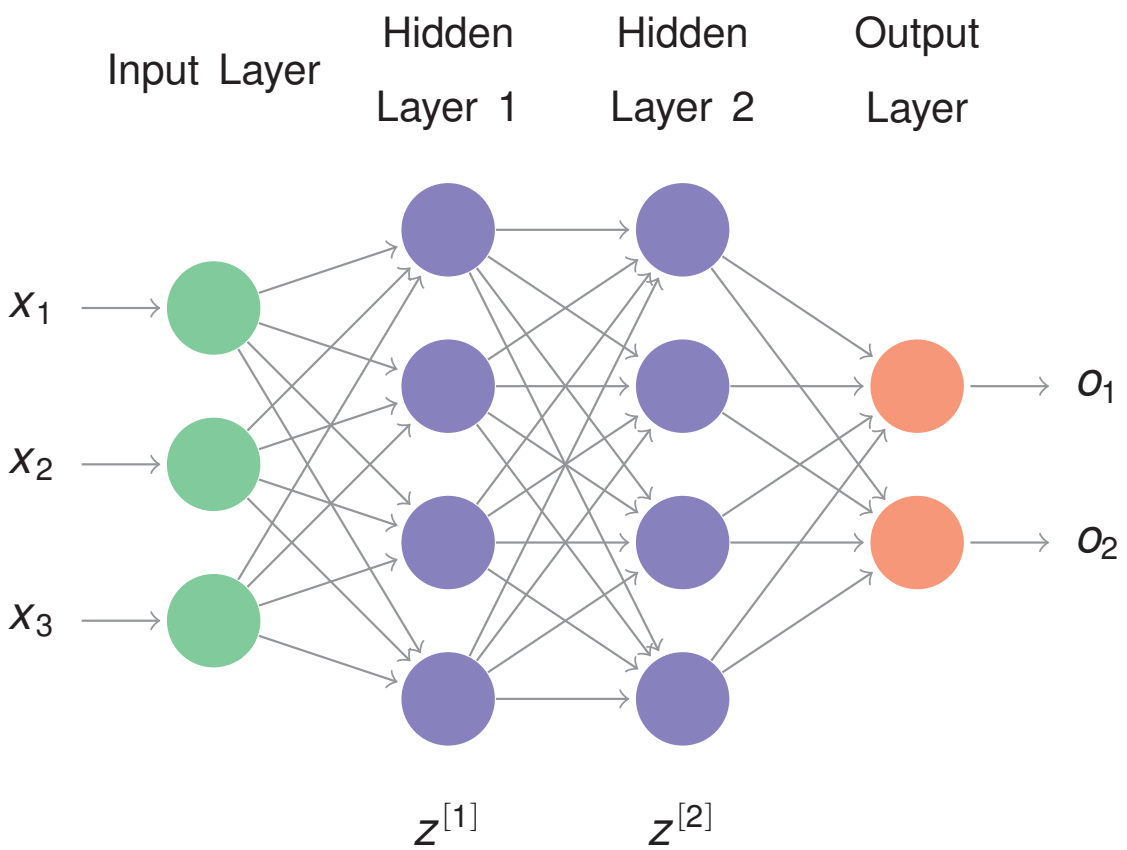
Multilayer Perceptron

- But in this lecture we will focus on the **Multilayer Perceptron** (**MLP**), a particular form of FFN in which each layer contains some number of identical sigmoid units
- The network is **fully connected**
  - Every unit in one layer is connected to every unit in the next
- The first layer is the **input layer**
  - It takes its values from the input attributes, x_i
  - The units in such a layer are known as **input units**

Multilayer Perceptron

- The last layer is the **output layer**
  - It defines values for the output variable, y
  - A slight nuance is that for multiclassification we have a separate y_i for each class i
  - The units in such a layer are known as **output units**
- All the layers in between are the **hidden layers**
  - The units in such a layer are known as **hidden units**
- The number of layers is the **depth** of the MLP
- The number of units in a layer is known as the **width** of the MLP

Multilayer Perceptron: Diagram



Multilayer Perceptron: Diagram

$$z_i^{[1]} = \widehat{\phi}\left(\sum_j w_{ij}^{[1]} x_j + b_i^{[1]}\right)$$

$$z_i^{[2]} = \widehat{\phi}\left(\sum_j w_{ij}^{[2]} z_j^{[1]} + b_i^{[2]}\right)$$

$$o_i = \widehat{\phi}\left(\sum_j w_{ij}^{[3]} z_j^{[2]} + b_i^{[3]}\right)$$

In general:

$$z_i^{[h]} = \widehat{\phi}\left(\sum_j w_{ij}^{[h]} z_j^{[h-1]} + b_i^{[h]}\right)$$

Feature Learning

- Why do we select this representation?
- For 1-layer:

$$o_i = \widehat{\phi}\left(\sum_j w_{ij}^{[2]}\widehat{\phi}\left(\sum_k w_{jk}^{[1]}x_k + b_j^{[1]}\right) + b_i^{[2]}\right)$$

(But in general o_i is more complex)

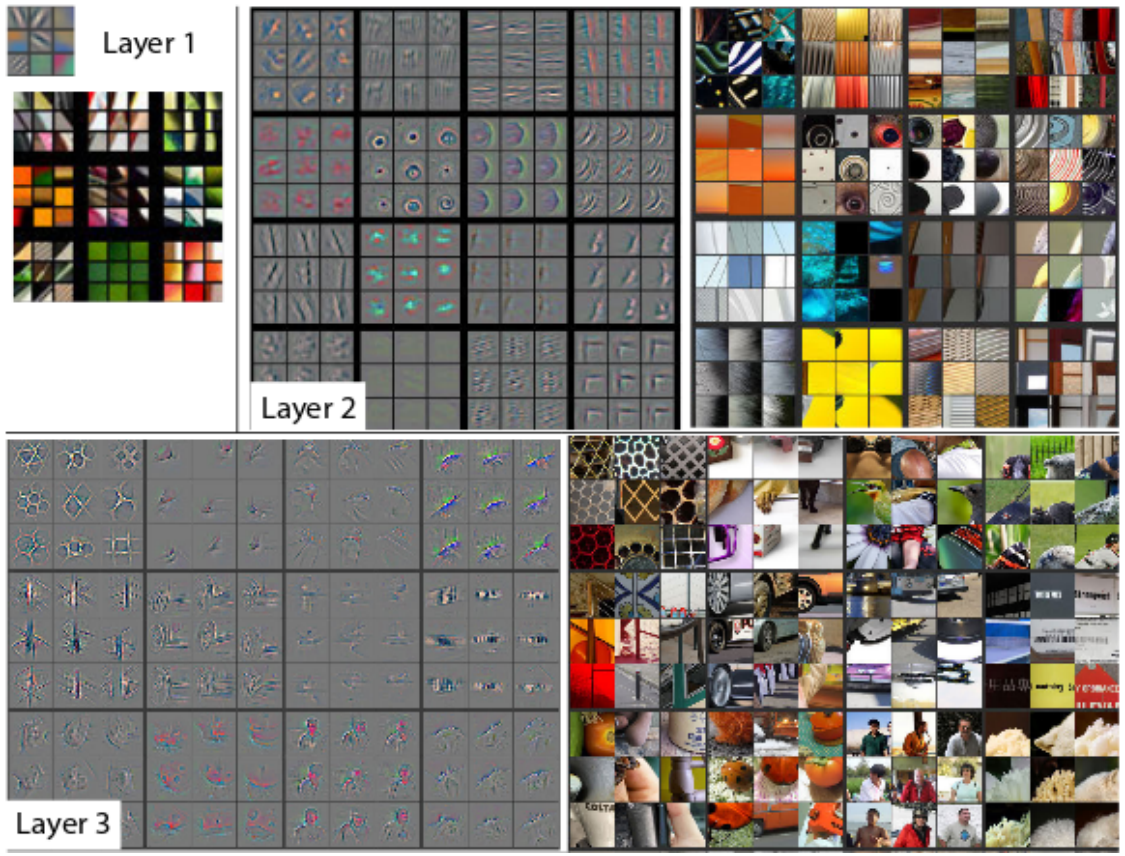
Feature Learning

- So, $o_i : (\mathbf{x}, \mathbf{w}) \mapsto \mathbb{R}$
  - Here $\mathbf{w}$ is the set of all weights and bias parameters
  - So a NN is a nonlinear mapping from a set of input variables, $\mathbf{x}$, to a set of output variables, $\{o_i\}$, with the functional forms controlled by the weights, $\mathbf{w}$, which we are to learn
- The number of weights (and basis functions) are chosen in advance via our architecture
  - c.f. Gaussian Processes or Kernel methods more generally
- But these basis functions are **adaptive** (via the weights)

Feature Learning

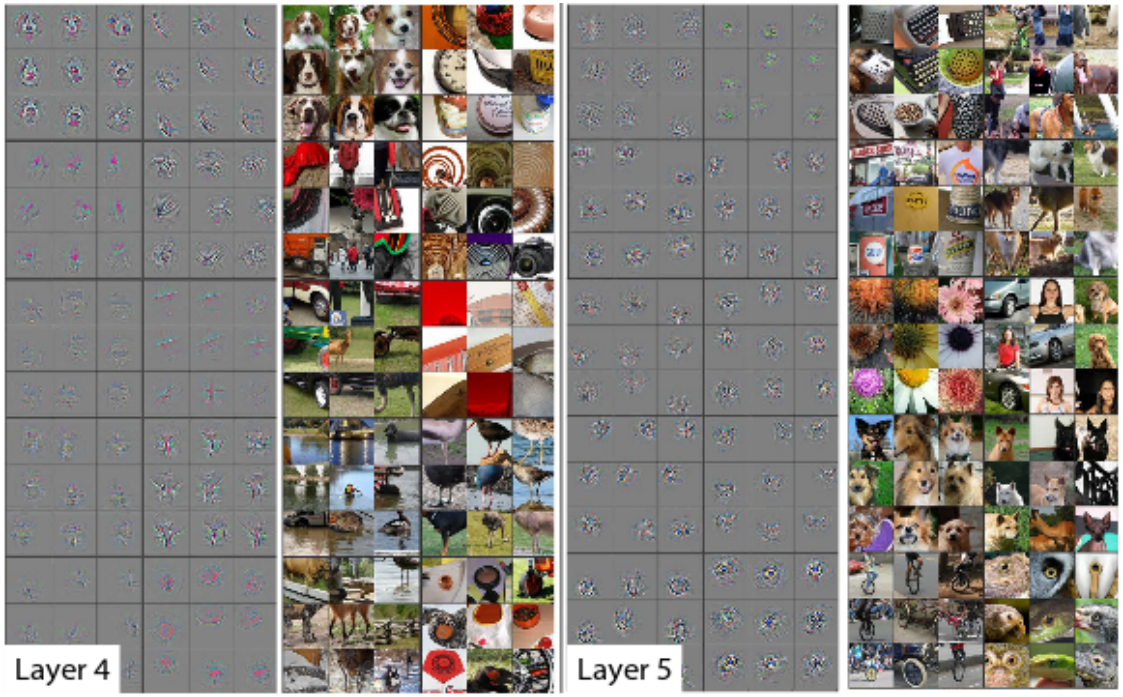
- It turns out that such a structure can be very **expressive**
- Layers of the NN are said to **learn features** implicitly
- And **deeper layers** learn more and more complex features...without any guidance!

Feature Learning: Example[2]



[2] 'Visualizing and Understanding Convolutional Networks' Zeiler & Fergus [2012]

Feature Learning: Example



Universality

- Why should MLP's be so expressive?
- Non-linearity of activation functions
- Linear activation functions can learn little
  - Even a deep set of layers of linear units will just result in a linear unit
- But we can prove that a shallow MLP with just one hidden layer and **non-linear** activation functions can learn any non-linear function of the input
- We say that an MLP is a **universal function approximator**

Universality

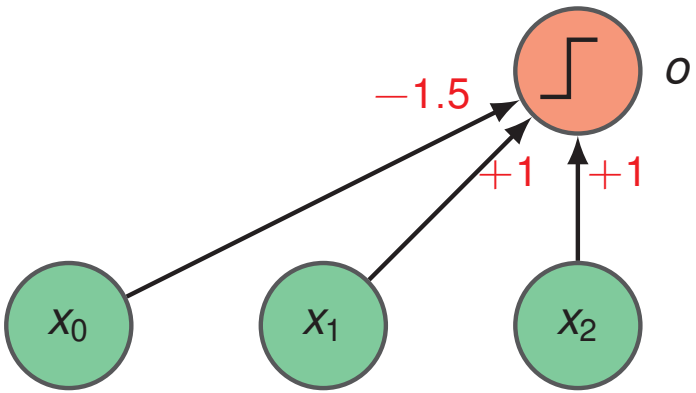
- They are *very* versatile...
  - ...But not necessarily **compact**
  - Sometimes exponentially large NN's are required to achieve function approximators
- This is one reason that **deep learning** is interesting:
  - One can often approximate the same function with a **narrow**, **deep**, network
  - Much more compactly than with a **wide**, **shallow** one

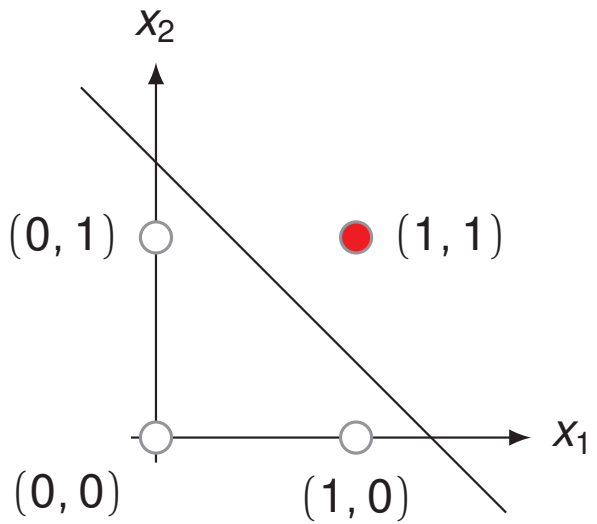
The XOR Problem

- Let's try to get some intuition for why the MLP acts as a Universal Function Approximator
- Consider a **Boolean** function
  - any such function can be represented as a **disjunction** of **conjunctions**...
  - and a disjunction of conjunctions can be implemented by an MLP with 1 hidden layer
    - Each conjunction is implemented by a hidden unit
    - Each disjunction is implemented by the output unit

The XOR Problem

- A perceptron can implement both AND and OR functions:
- For example, with $\mathbf{x} = [1, x_1, x_2]^T$ and $\mathbf{w} = [-1.5, 1, 1]^T$ we have the ingredients of an AND function:





The XOR Problem

- Which yields the following truth table:

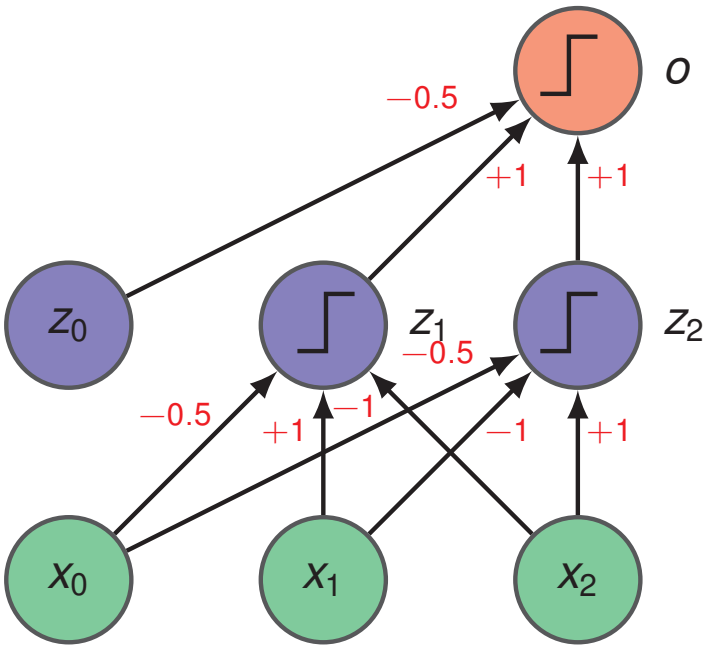
x_1	x_2	$\mathbf{w} \cdot \mathbf{x}$	o
0	0	-1.5	0
0	1	-0.5	0
1	0	-0.5	0
1	1	0.5	1

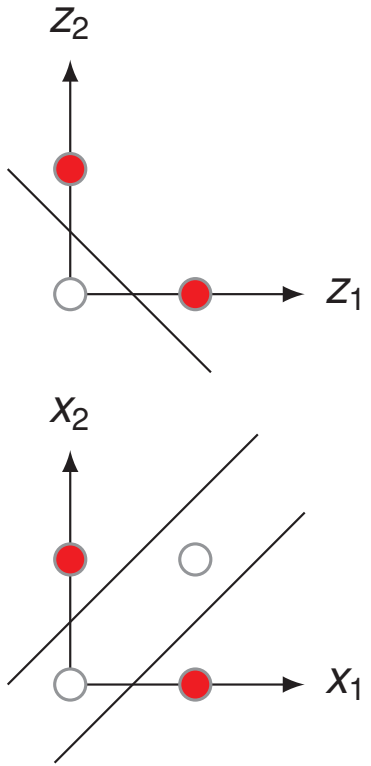
The XOR Problem

- ...But a perceptron can’t implement a XOR function...

The XOR Problem

- However two perceptrons in parallel can implement the two ANDs and another perceptron afterwards can OR these results together:





The XOR Problem

- The first layer maps from **x** to **z**...
- ...And in this space the data is **linearly separable** - because $(x_1 = 0, x_2 = 0)$, $(x_1 = 1, x_2 = 1)$ are both mapped to $(z_1 = 0, z_2 = 0)$
- Which yields the following truth table:

x_1	x_2	z_1	z_2	o
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

Boolean Functions and the MLP

- In general, for Boolean functions with 2 inputs:
  - For each input combination where the output is 1, we define a hidden unit that checks for that particular conjunction of the inputs
  - The next layer then performs a disjunction between these
- But what if there are d inputs?
  - Up to 2^d hidden units may be necessary
  - We can still approximate the function with a two layer MLP....But we end up with a cumbersome lookup table
  - This architecture is not **compact** and not good at **generalisation**

Continuous Functions and the MLP

- How can we approximate continuous functions?
- We'll focus on the 2 hidden layer case:
  - For each input case / region, we can **delimit** that region by hyperplanes on all sides using hidden weights in the first hidden layer
  - A hidden unit in the next layer then ANDs them together to **bound** the region
  - Then the weight of the connection from that hidden unit to the output unit is set equal to the desired function value
  - The results is a **piecewise constant approximation**
  - Its accuracy may be increased by increasing the number of hidden units which is equivalent to placing a finer grid over the inputs

Lecture Overview

Evaluation

- So how do we learn all the weights in the MLP architecture and attempt to learn any function?
- First we need to define an evaluation function
- NN's are very flexible, and can accommodate a variety of evaluation functions - as usual we need to pick one for the task at hand
- Let's take a look at three canonical cases, for a H hidden layer network:

Regression

- Here we have one output, $o^{(i)}$
- And we set this output to be a **linear function** of units in the last hidden layer:

$$o^{(i)} = \sum_j w_j^{[H+1]} z_j^{[H](i)} + b^{[H+1]}$$

- Then set evaluation function to be the squared loss:

$$\mathsf{L} = \frac{1}{2} \sum_{i=1}^{n} (o^{(i)} - y^{(i)})^2$$

Binary Classification

- Here we have one output, $o^{(i)}$
- And we set this output to be a **logistic sigmoid function** of units in the last hidden layer:

$$o^{(i)} = \frac{1}{1 + e^{-\left(\sum_j w_j^{[H+1]} z_j^{[H](i)} + b^{[H+1]}\right)}}$$

- Then set evaluation function to be the cross entropy:

$$\mathrm{L} = -\sum_{i=1}^{n} y^{(i)} \ln(o^{(i)}) + (1 - y^{(i)}) \ln(1 - o^{(i)})$$

Multi-Classification

- Here we have K 'initial' outputs, $\{\widetilde{o}_k^{(i)}\}_{k=1}^K$, from the last hidden layer:

$$\widetilde{o}_k^{(i)} = \sum_j w_{kj}^{[H+1]} z_j^{[H](i)} + b_k^{[H+1]}$$

- And we push these through a **softmax function** to generate the final output units:

$$o_k^{(i)} = \frac{\exp \widetilde{o}_k^{(i)}}{\sum_{j=1}^K \exp \widetilde{o}_j^{(i)}}$$

- Then set evaluation function to be the cross entropy:

$$\mathrm{L} = -\sum_{i=1}^n \sum_{k=1}^K y_k^{(i)} \ln(\widetilde{o}_k^{(i)})$$

Forward Propagation

- In order to calculate each of these evaluation functions then clearly we need to evaluate $o^{(i)}$ for each of our training examples
- $o^{(i)}$ is calculated as a **forward pass** through the network:
  - $\mathbf{x}^{(i)}$ is passed through the first hidden layer where $\{\mathbf{z}^{[1](i)}\}$ is calculated
  - The set $\{\mathbf{z}^{[1](i)}\}$ is passed through the second hidden layer where $\{\mathbf{z}^{[2](i)}\}$ is calculated
  - And so on until the output layer
- This process of evaluation is often termed the **forward propagation** of information through the network

Lecture Overview

Optimisation

- In order to set values on the weights we need to optimise our loss function L
- In common with, for example, logistic regression, in general we cannot optimise L for a NN analytically
- So let's apply a numerical technique, **gradient descent**:

Optimisation

- Recall that for a general representation of the weights, $\mathbf{w}$, gradient descent looks like:

$$\mathbf{w}_{(t+1)} \leftarrow \mathbf{w}_{(t)} - \alpha \nabla_{\mathbf{w}} \mathrm{L}|_{\mathbf{w}=\mathbf{w}_{(t)}}$$

  Here α is the learning rate
  And $\mathbf{w}$ is just a shorthand for *all* the weights and biases in the network

- So what we really need to calculate is:

$$\frac{\partial \mathrm{L}}{\partial w_{ij}^{[h]}}, \frac{\partial \mathrm{L}}{\partial b_{i}^{[h]}} \qquad \forall i, j, h$$

Optimisation

- This amounts to a large amount of derivatives to calculate - how should we proceed?
- **Finite Difference** is one approach...But it's expensive and would require many passes through the network to approach optimality
- Is there a more efficient approach?
- We could try to apply the **Chain Rule**. For $f(u)$ where $u = u(v)$:

$$\frac{\partial f}{\partial v} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial v}$$

- This is the foundation for a key optimisation technique for NN's: the **backpropagation algorithm**

Backpropagation

- Since $\mathrm{L} = \sum_{\widetilde{i}=1}^{n} \mathrm{L}^{(\widetilde{i})}$, where $\mathrm{L}^{(\widetilde{i})}$ is the loss associated with a single example...
- ...Then let's begin by calculating the derivative of $\mathrm{L}^{(\widetilde{i})}$...
- ...We can then sum these later to calculate the aggregate derivative of L
- So, for a network with H hidden layers, let's start by calculating the derivative of $\mathrm{L}^{(\widetilde{i})}$ with respect to the weights in some layer h

Backpropagation[3]

- Let us recall that, for a hidden layer, the activations are given by:

$$z_i^{[h]} = \widehat{\phi}\left(a_i^{[h]}\right) \tag{1}$$

$$a_i^{[h]} = \sum_j \left(w_{ij}^{[h]} z_j^{[h-1]} + b_i^{[h]}\right) \tag{2}$$

- Then, using the chain rule:

$$\frac{\partial \mathsf{L}}{\partial w_{ij}^{[h]}} = \frac{\partial \mathsf{L}}{\partial a_i^{[h]}} \frac{\partial a_i^{[h]}}{\partial w_{ij}^{[h]}} = \delta_i^{[h]} \frac{\partial a_i^{[h]}}{\partial w_{ij}^{[h]}}$$

- Where $\delta_i^{[h]}$ are known as the **errors**:

$$\delta_i^{[h]} = \frac{\partial \mathsf{L}}{\partial a_i^{[h]}} \tag{3}$$

[3] Henceforth we drop the superscript $(\widetilde{i})$ to avoid cluttered notation. Furthermore, we concentrate on taking derivatives w.r.t. $w_{ij}^{[h]}$, a similar analysis obtains for $b_i^{[h]}$

Backpropagation

- Now, from expression(2), $\frac{\partial a_i^{[h]}}{\partial w_{ij}^{[h]}} = z_j^{[h-1]}$, so:

$$\frac{\partial \mathrm{L}}{\partial w_{ij}^{[h]}} = \delta_i^{[h]} z_j^{[h-1]} \tag{4}$$

- In other words the derivative we seek is equivalent to:

(value of δ for the unit at the output end of weight)

$\times$

(value of z for the unit at the input end of the weight)

- Thus we need only calculate $\delta_i^{[h]}$ for each hidden and output unit in the network:

Backpropagation: Calculating $\delta_i^{[h]}$

- For the output units for the loss functions in which we are interested (for example the squared loss or the cross entropy), we have:

$$\delta_i^{[H+1]} = o - y \tag{5}$$

- Why?...Recall the output definition for each of the loss functions:
  - **Regression**:
    $o = a^{[H+1]}$
  - **Binary Classification**:
    $o = \frac{1}{1+e^{-a^{[H+1]}}}$
  - **Multi-Classification**:
    $o_k = \frac{e^{-a_k^{[H+1]}}}{\sum_{j=1}^{K} e^{-a_j^{[H+1]}}}$

Backpropagation: Calculating $\delta_i^{[h]}$

- For the hidden units we use the chain rule:

$$\begin{aligned}\delta_i^{[h]} = \frac{\partial \mathrm{L}}{\partial a_i^{[h]}} &= \sum_k \frac{\partial \mathrm{L}}{\partial a_k^{[h+1]}} \frac{\partial a_k^{[h+1]}}{\partial a_i^{[h]}} \\ &= \sum_k \delta_k^{[h+1]} \frac{\partial a_k^{[h+1]}}{\partial a_i^{[h]}} \end{aligned} \tag{6}$$

  Where k runs over all the units to which unit i sends connections

- Now we need to find an expression for $\frac{\partial a_k^{[h+1]}}{\partial a_i^{[h]}}$

Backpropagation: Calculating $\delta_i^{[h]}$

- Using expressions (1 & 2) and the chain rule:

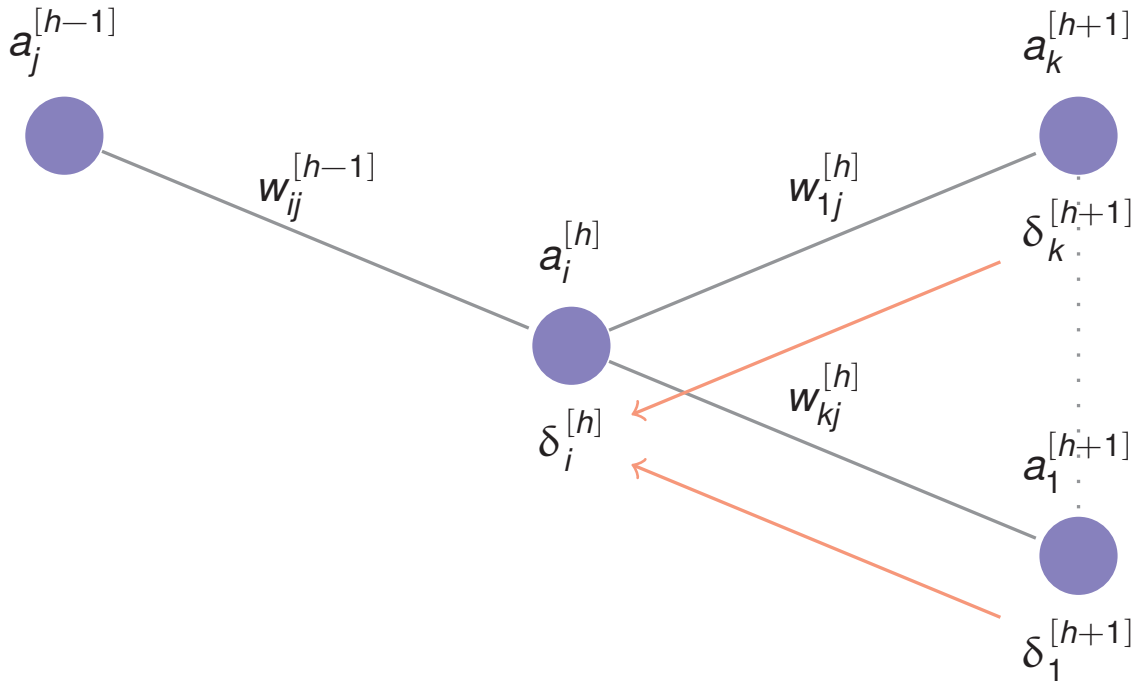
$$\begin{aligned}\frac{\partial a_k^{[h+1]}}{\partial a_i^{[h]}} &= \frac{\partial a_k^{[h+1]}}{\partial z_i^{[h]}} \frac{\partial z_i^{[h]}}{\partial a_i^{[h]}} \\ &= w_{ki}^{[h+1]} \widehat{\phi}' \left(a_i^{[h]} \right)\end{aligned}$$

- Substituting this into expressions (6) gives:

$$\delta_i^{[h]} = \widehat{\phi}' \left(a_i^{[h]} \right) \sum_k w_{ki}^{[h+1]} \delta_k^{[h+1]} \tag{7}$$

Backpropagation: Calculating $\delta_i^{[h]}$

- In other words, the value of δ for a particular hidden unit, h, can be obtained by propagating the δ's backwards from higher up the network:



- If we recursively apply expression (7) we can generate δ's for the entire NN

Backprop Algorithm

Algorithm 1 Backprop

1: **initialise:** $\{w_{ij}^{[h]}, b_i^{[h]}\} \quad \forall i, j, h$

2: **apply:** input vector $\mathbf{x}^{(\widetilde{i})}$ to network and **forward propagate** to generate $o_i^{(\widetilde{i})}, z_i^{[h](\widetilde{i})} \forall i, h$ in network, using:

$$z_i^{[h](\widetilde{i})} = \widehat{\phi}\left(a_i^{[h](\widetilde{i})}\right)$$

$$a_i^{[h](\widetilde{i})} = \sum_j \left(w_{ij}^{[h]} z_j^{[h-1](\widetilde{i})} + b_i^{[h]}\right)$$

3: **evaluate:** $\delta_i^{[H+1](\widetilde{i})}$ for all output units, using:

$$\delta_i^{[H+1](\widetilde{i})} = o_i^{(\widetilde{i})} - y^{(\widetilde{i})}$$

4: **backpropagate:** δ's to obtain $\delta_i^{[h](\widetilde{i})}$ for each hidden unit in the network, using:

$$\delta_i^{[h](\widetilde{i})} = \widehat{\phi}'\left(a_i^{[h](\widetilde{i})}\right) \sum_k w_{ki}^{[h+1]} \delta_k^{[h+1](\widetilde{i})}$$

5: **evaluate:** all derivatives, using:

$$\frac{\partial \mathrm{L}^{(\widetilde{i})}}{\partial w_{ij}^{[h]}} = \delta_i^{[h](\widetilde{i})} z_j^{[h-1](\widetilde{i})}$$

6: **return:**

$$\frac{\partial \mathrm{L}}{\partial w_{ij}^{[h]}} = \sum_{\widetilde{i}=1}^{n} \frac{\partial \mathrm{L}^{(\widetilde{i})}}{\partial w_{ij}^{[h]}}$$

Backpropagation: Flexibility

- Note that Backprop is a generic algorithm that works regardless of the activation function (subject to smoothness) or loss function (subject to structure)
- It can accommodate a variety of activation functions within the same network...
- ...Things just get more fiddly!

Why Backprop?

- Efficiency:
  - 1 forward pass takes $\mathcal{O}(|\mathbf{w}|)$
  - 1 backward pass takes $\mathcal{O}(|\mathbf{w}|)$
- Compare with **finite difference**:
  - Each forward pass takes $\mathcal{O}(|\mathbf{w}|)$ and we need to perform $|\mathbf{w}|$ of these for each unit we perturb
  - This efficiency is $\mathcal{O}(|\mathbf{w}|^2)$

Batch or Stochastic Gradient Descent?

- Training NN's can be expensive, even using Backprop
- Added to this, the capacity of NN's necessitates training on large datasets
- For these reasons we often prefer to use SGD (recall our comparisons of these methods in the *Linear Regression Lecture*)...
- ...Here we apply our updates on an example-by-example basis, rather than across the batch
- This makes for a faster, though locally less stable, convergence

Problems

- Recall the problems associated with numerical optimisation from the *Linear Regression Lecture*
- We have all of these in NN training...and more...
- Much of the work in NN's early in this iteration of their popularity has surrounded identifying and developing workarounds for these problems
- Let's examine some of them:

Non-Convexity

- This is the elephant in the room: NN's, because they are the result of performing both positive and negative summations on convex (or non-convex) activation functions, do not result in convex loss functions
- So Gradient Descent has no reason to converge
- The function which we are attempting to optimise is, in general, riddled with multiple local minima and saddle points

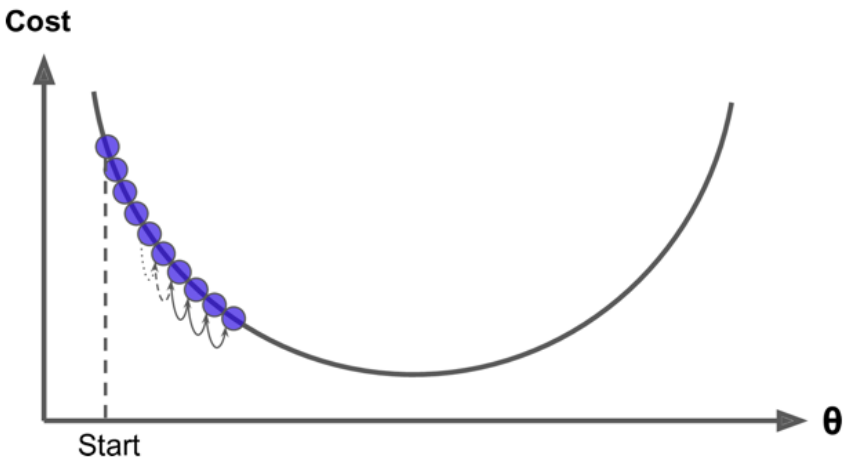
Non-Convexity

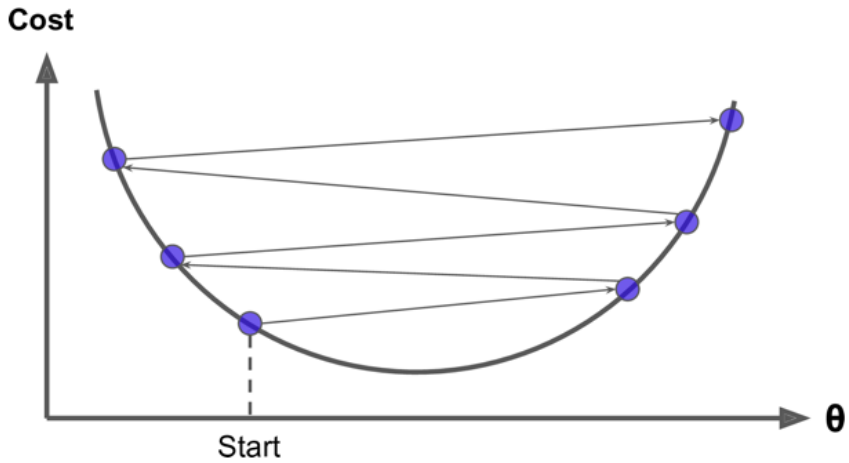
- How can we resolve this?
- We can't - without enumerating all minima we can't know that we're at a globally optimal point rather than just a local one
- One workaround is to re-run Gradient Descent several times for different random intialisations of $\mathbf{w}$ and select the best optimum from amongst them...But it's just a heuristic

Symmetries

- How do we initialise the weights when performing Gradient Descent?
- What if we initialise $\mathbf{w} = 0$?
  - Then all hidden activations will be the same...
  - ...And all weights feeding into a hidden unit will have identical derivatives
  - So then weights will have identical values in the next step
  - The complexity reduces to one neuron!
- We can resolve this problem by symmetry breaking via **random initialisation**

Learning Rate Size[4]





- Recall from the *Linear Regression Lecture*
  - For α too small, convergence is **slow**
  - For α too large, we get **divergence**
- The solution is to treat α as a hyperparameter and then tune it

[4] Geron, 'Hands-on Machine Learning With Scikit-Learn & Tensorflow' [2017]

Fluctuations

- A feature of stochastic gradient descent is that the cost function won't necessarily monotonically decrease
- The stochastic gradients are noisy and sometimes push us in the wrong direction
- This is more manifest as we approach an optimal point - further away we generally take big enough steps to swamp this fluctuation noise
- A solution is **learning rate decay**: we set $\alpha = \alpha_{(t)} = \alpha_{(0)} e^{-t/\tau}$
  - τ is the decay scale
  - Implicitly this will result in an α which exhibits rapid but late decay

Dead Units

- If we have moved to a region where the activation of a unit changes little, i.e. $\widehat{\phi}'\left(a_i^{[h]}\right) \approx 0$, then by expression (7):

$$\delta_i^{[h]} \approx 0 \qquad \Longrightarrow \qquad \frac{\partial \mathsf{L}}{\partial w_{ij}^{[h]}} \approx 0 \quad \forall j$$

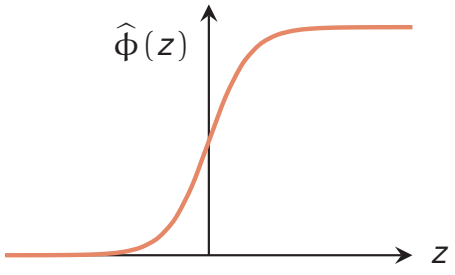
- This means that we have become stuck on a kind of plateau on the loss surface which it will take a long time to escape from
- Learning will slow down
- If we look at a histogram of activations we can see whether we have a big problem

Dead Units

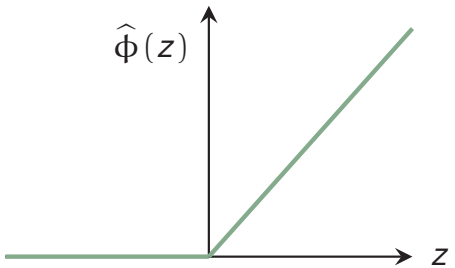
- A solution is to carefully pick random initialisations so that activations are away from their dead points
- An alternative solution is to pick activation units which have fewer dead regions
- So long as our units are non-linear then we can do this!

Activation Units

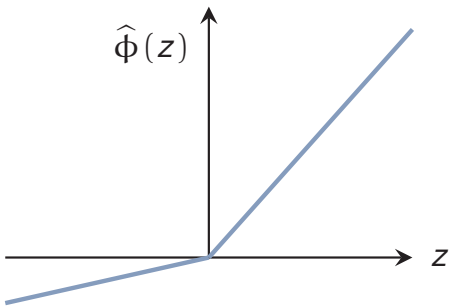
- **Sigmoid**:



- **ReLu**:



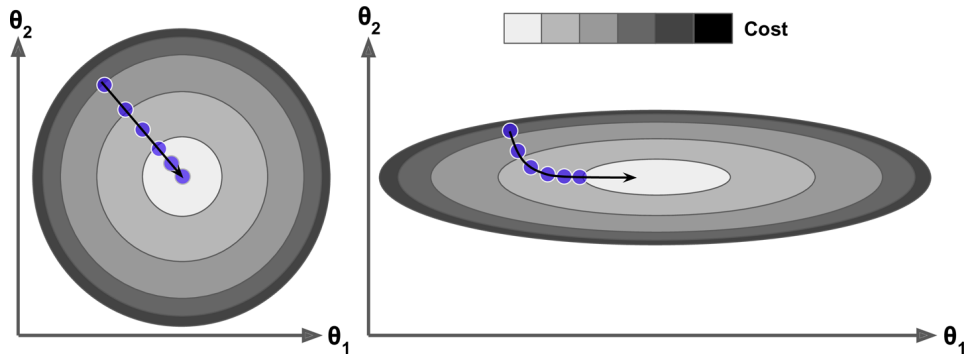
- **Leaky ReLu**:



Bad Conditioning[5]

- Again, recall from the *Linear Regression Lecture* the problem of scaling:

- For unbalanced curvature the gradient of the largest parameter will dominate the update leading to slow convergence



[5] Geron, 'Hands-on Machine Learning With Scikit-Learn & Tensorflow' [2017]

Bad Conditioning

- This is a particular issue for NN's, which exhibit very irregular loss surfaces
- But scaling is at least a partial solution:
  - It can't change a fundamentally highly non-convex surface...
  - ...But it can remove the curvature issues due to inputs with different scales

Recap

- **Representation:**

$$\mathcal{F} = \left\{ \left\{ o_i(\mathbf{x}) = \widehat{\Phi}^{[H+1]} \left(\sum_{j=1}^{n_H} w_{ij}^{[H]} \widehat{\Phi}^{[H]} \left(\sum_{k=1}^{n_{H-1}} w_{jk}^{[H-1]} \widehat{\Phi}^{[H-1]} \left(\sum_{l=1}^{n_{H-2}} w_{kl}^{[H-2]} \dots \right) \right) \right) \right\}_{i=1}^{I} \middle| \left\{ \widehat{\Phi}^h \right\}_{h=1}^{H+1}, \mathbf{w} \right\}$$

  Here I is the number of output units (usually 1 for regression and binary classification)

- **Evaluation:**

$$\mathsf{L} = \frac{1}{2} \sum_{i=1}^{n} \left(o^{(i)} - y^{(i)} \right)^2, \qquad \text{etc.}$$

- **Optimisation:**

Backpropagation
&
(Stochastic) Gradient Descent

Lecture Overview

Overfitting

- NN's, because of their **capacity** are prone to **overfitting**
- In order to combat this we adopt **regularisation** procedures
- As well as the usual regularisation approaches we will see that there are a handful which are unique to NN's

Weight Decay

- This is the name given by the NN community to traditional regularisation
- A penalty term is added to the loss function:
  - For example ℓ_2 regularisation will result in a modified loss function:

$$\widetilde{L} = L + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

  - Where L is the usual empirical loss function

Weight Decay

- If we optimise this using Gradient Descent then an extra term will appear in each update, such that, for all i, j in the network:

$$w_{ij(t+1)} \longleftarrow w_{ij(t)} - \alpha \frac{\partial L}{\partial w_{ij}}\bigg|_{w_{ij}=w_{ij(t)}} - \alpha\lambda w_{ij(t)}$$

$$\longleftarrow (1 - \alpha\lambda) w_{ij(t)} - \alpha \frac{\partial L}{\partial w_{ij}}\bigg|_{w_{ij}=w_{ij(t)}}$$

- In other words the weights experience a decay modulation, $(1 - \alpha\lambda)$, each iteration

Reducing Capacity

- We can treat the number of layers and the number of units/layer in an NN as tunable hyperparameters, and attenpt to regularise on this basis
- In addition we can introduce a **bottleneck**
  - This is a layer with fewer units than its neighbouring layers
  - Linear Units are perfectly adequate for this function
    - They can't increase the ability of the NN to learn more complex functions...
    - ...But they can, in some sense, extract the important information from a layer in the NN and express it more efficiently

Reducing Capacity

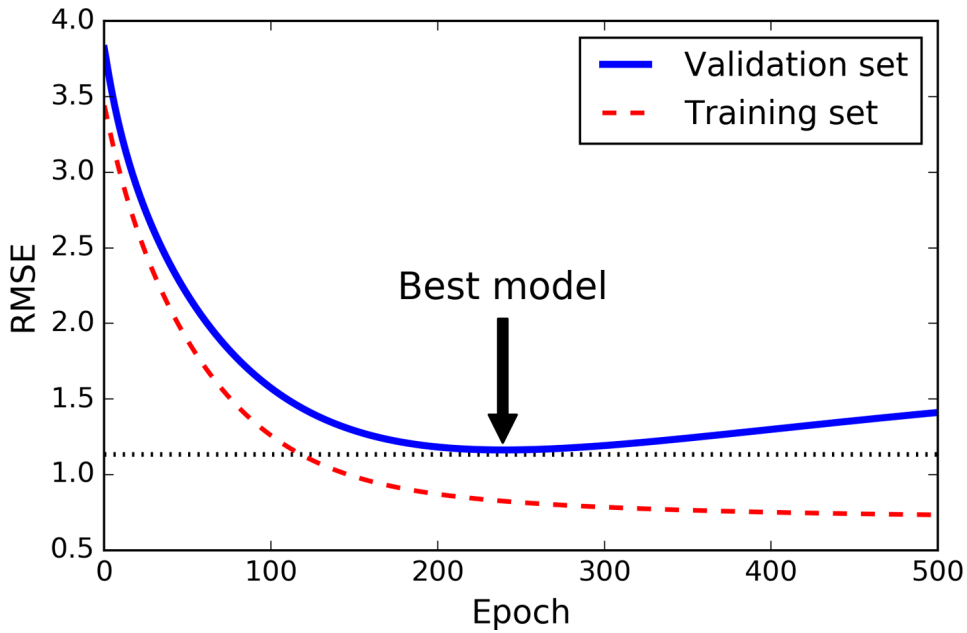
- Bottlenecks significantly reduce the number of parameters

Early Stopping

- Typically when adopting an iterative numerical procedure for optimisation we speak of **epochs**
- An epoch is a full training cycle
  - For example, one update in batch gradient descent...
  - ...or n updates in stochastic gradient descent

Early Stopping[6]

- In NN training we often see the following form of **training curves:**



[6] Geron, 'Hands-on Machine Learning With Scikit-Learn & Tensorflow' [2017]

Early Stopping

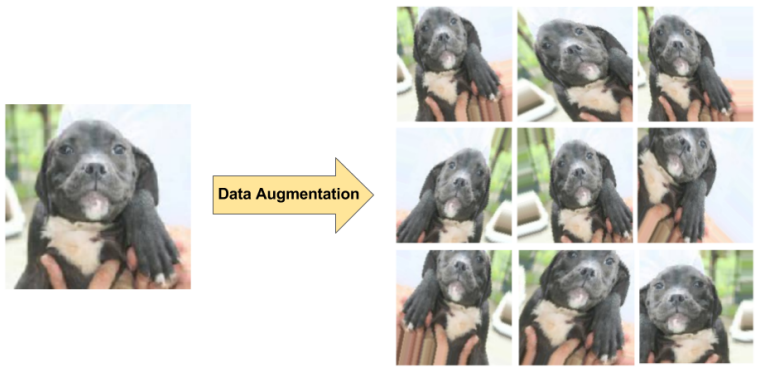
- In other words, as NN's are trained for longer they begin to fit the training data too well, and hence **overfit**
- **Early stopping** is a procedure designed to eliminate this by observing loss v number of epochs on a validation and ending training accordingly

Data Augmentation

- Another way of remedying overfitting is by providing more training examples!
- **Data Augmentation** seeks to generate such additional data
- This is a domain specific technique

Data Augmentation[7]

- In image data for example it is possible to manipulate the original image in a number of ways which result in useful pseudo-novel training data:
  - rotation
  - distortion
  - translation



[7] https://developers.google.com/machine-learning/practica/image-classification/preventing-overfitting

Lecture Overview

Comments

- Neural Networks have been startlingly successful in recent years
- Part of the reason for this is that they allow us to learn almost any function
- But they are also poorly understood
- The representation which they offer is can be off-putting
  - it is far from the intuitive transparency of a decision tree

Comments

- But more significantly they represent a non-convex problem
  - GD and SGD have no reason to work
  - But when combined with the bag-of-tricks which we have discussed they often do...
  - Why this should happen is an open question...

Summary

1. **Neural Networks** are **Universal Function Approximators** which offer access to a rich function class, and which have enjoyed tremendous empirical success on certain tasks in recent years

2. It is possible to train them at scale with a combination of the **backpropagation algorithm** and **stochastic gradient descent** together with a set of tweaks which combine to make learning more efficient

3. Because of their high **capacity** Neural Networks have the potential to **overfit** and this must be controlled via more or less ad hoc **regularisation** techniques