Ollscoil na Gaillimhe
University of Galway

Deep Learning

# Topic 04: Deep Neural Networks, Part 1

Prof. Michael Madden

Chair of Computer Science
Head of Machine Learning Group
University of Galway

## Learning Objectives

After successfully completing Topics 4 and 5, you will be able to…

Define key concepts related to Deep Learning, and discuss major advances relative to shallow neural networks

Explain and implement approaches to handling unstructured data and multi-class classification

Explain and implement regularization methods, and algorithms including Mini-Batch Gradient Descent, Momentum, RMSprop and Adam

Explain the principles of operation of Convolutional Networks

Discuss model re-use and transfer learning

Correctly use these features within ML libraries, including selecting hyperparameters

## This Topic: Deep NNs, Part 1

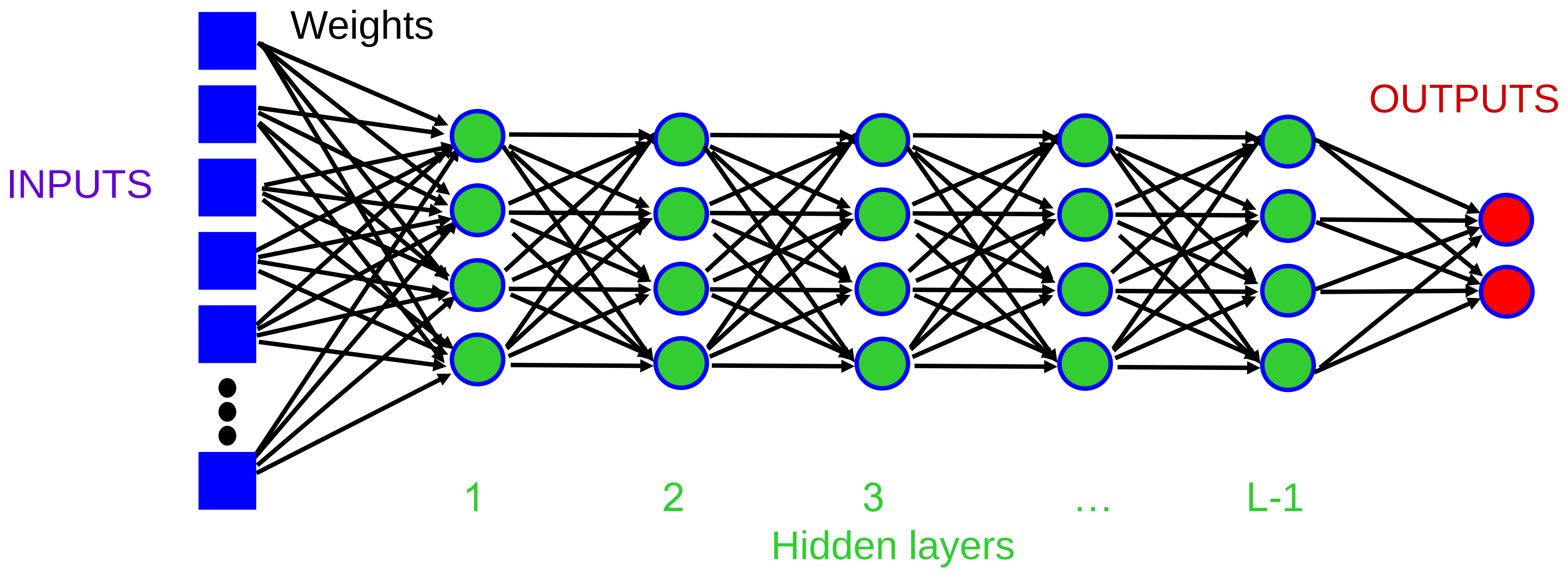
- What is Deep Learning and why is it important?
  - Why depth matters
  - Practical perspective and connectionist computing perspective
- Details of some key Deep Learning techniques:
  - Handling unstructured data
  - Multi-class classification with Softmax
  - Methods to avoid overfitting
- Limitations of standard training algorithms
- Mini-Batch Gradient Descent

# Next Topic: Deep NNs, Part 2

- More Deep NN Training Algorithms:
  - Backprop with Momentum
  - RMSprop
  - Adam
- Locally Connected Networks
- Convolutional Neural Networks:
  - Relationship to classical computer vision operations
  - CNN terminology
  - CNN architectures and training

# What are Deep Neural Networks?

- At basic level, NNs with more than a small number of hidden layers



- Our NN algorithm from last week is already able to handle multiple hidden layers

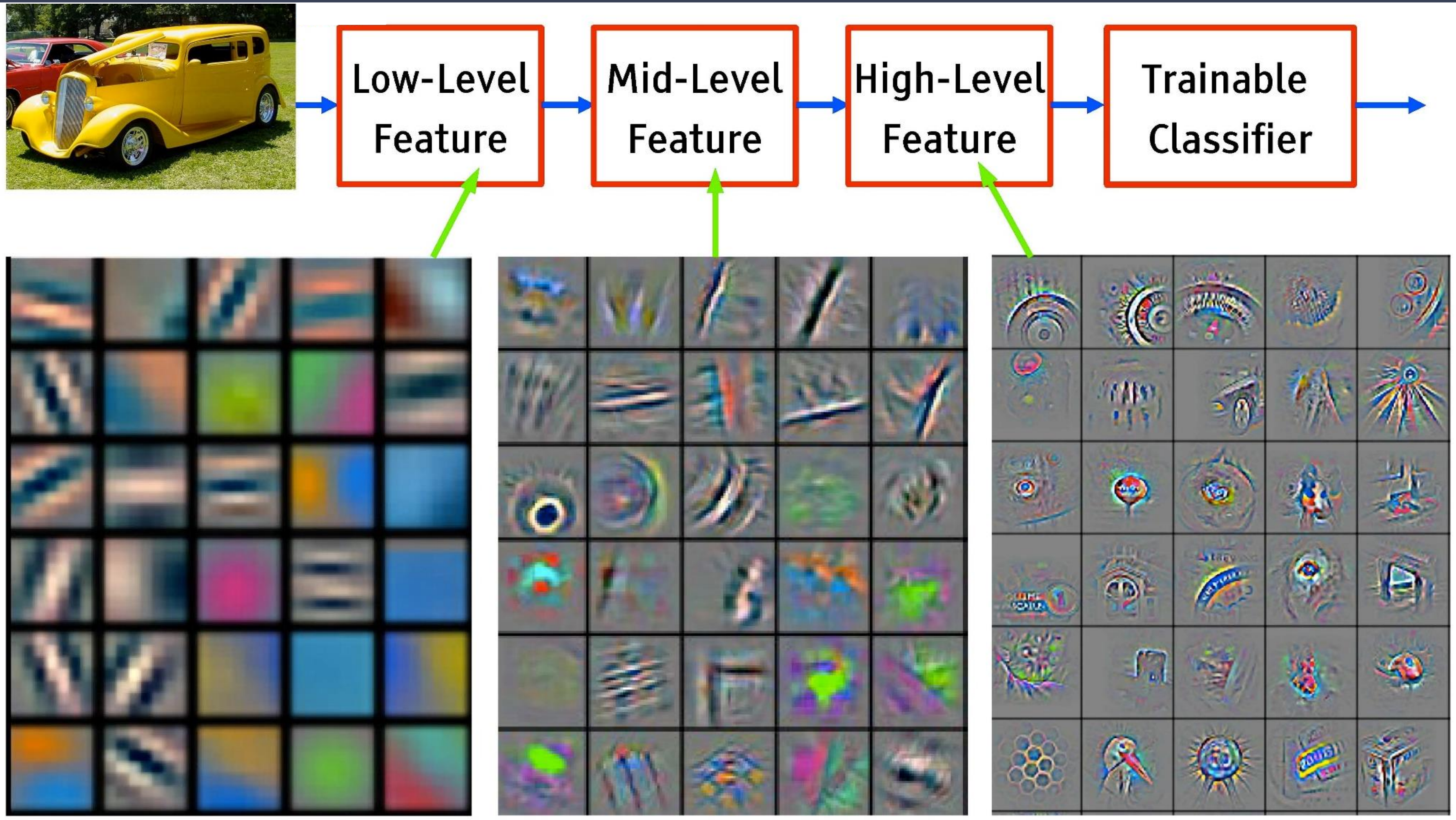
> Hidden Layers: Loop backwards from L-1 to 1:

- However, there are many important advances to consider …

## What are Deep Neural Networks?

- At basic level, NNs with more than a small number of hidden layers
  - Generally large in scale: many input nodes; many hidden layers; many nodes per hidden layer; large amounts of training data
  - The term also refers to the inclusion of many other advances beyond the shallower neural nets of the 1990s
  - The term **Deep Learning** is a ‘rebranding’ to distinguish current-generation NNs from older methods that had fallen out of favour
- Universal approximation theorems:
  - With 1 hidden layer that has an unlimited number nodes, can approximate any continuous function
  - Assumption of unlimited nodes per layer is clearly unrealistic
- With more layers, later layers build on computations of earlier layers
  - Overall size reduced for given overall complexity
  - Analogy: big block of code vs program organised into hierarchy of functions

# More Intuition on Benefit of Depth: Learning Hierarchy of Representations

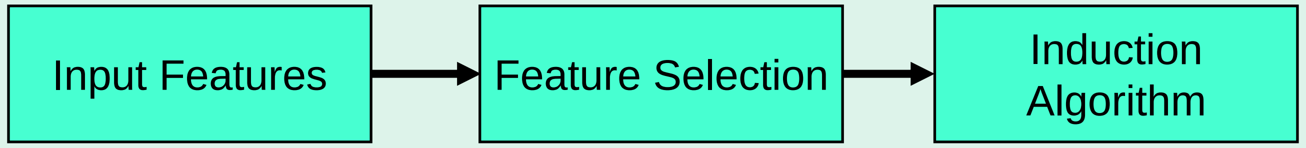


Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

# High-Level Perspectives

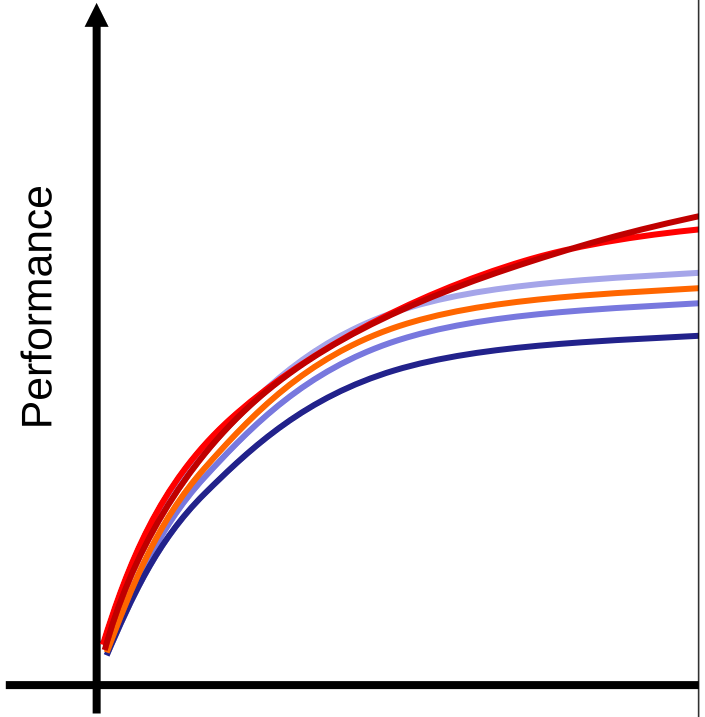
- Practical perspective:
  - Deep architectures achieve many of the same things as shallow ones, but more efficiently, particularly for perception tasks (vision/sound)
  - Basic training techniques were not effective, so we needed to find ways of getting it to work
- Connectionist programming perspective:
  - Deep learning provides an integrated approach to feature engineering and learning, all within a connectionist architecture
  - Nodes in early layers can be considered functions/subroutines that are re-used in later ones
  - Convolutional NNs extend this idea further

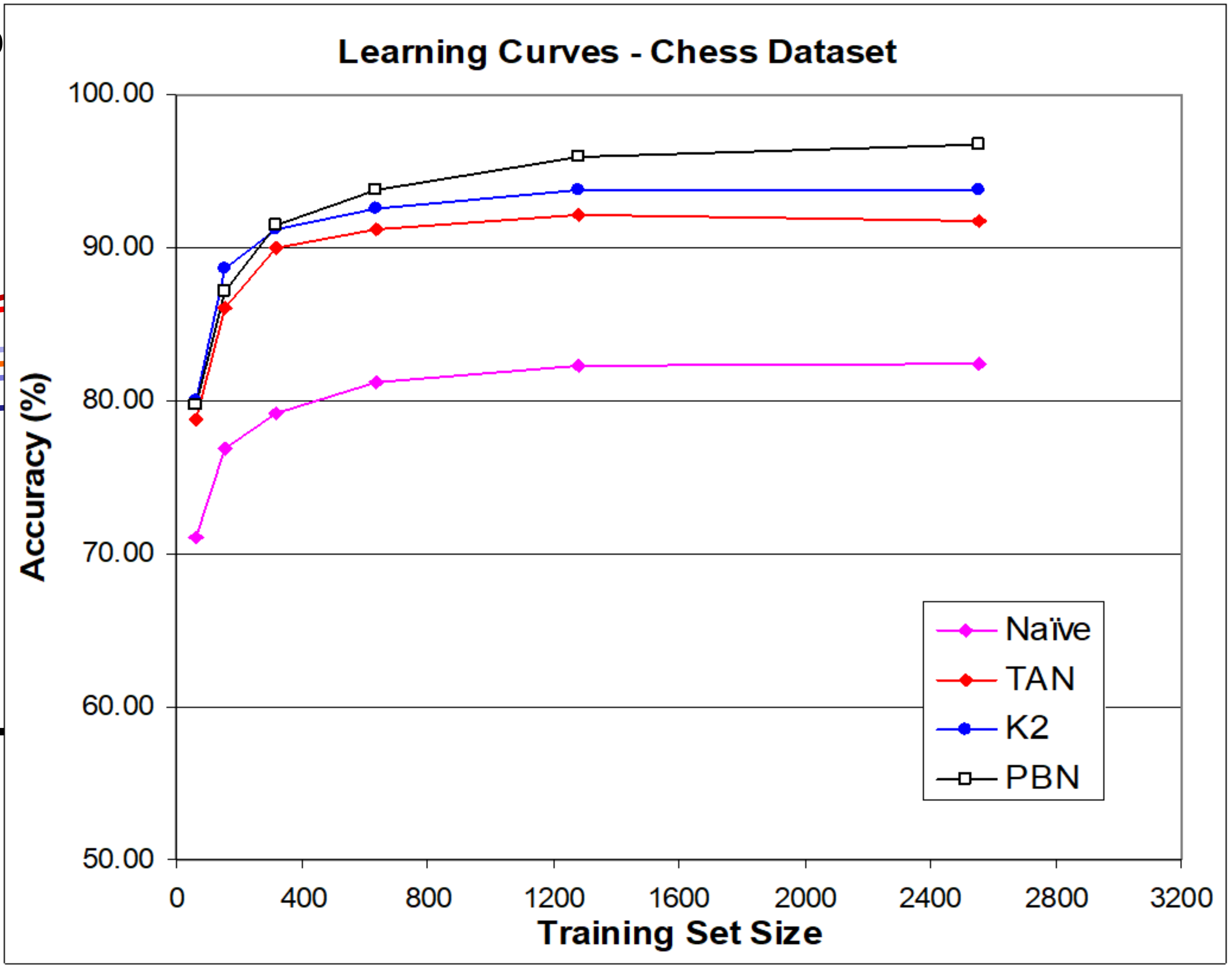
Reminder: typical feature selection process



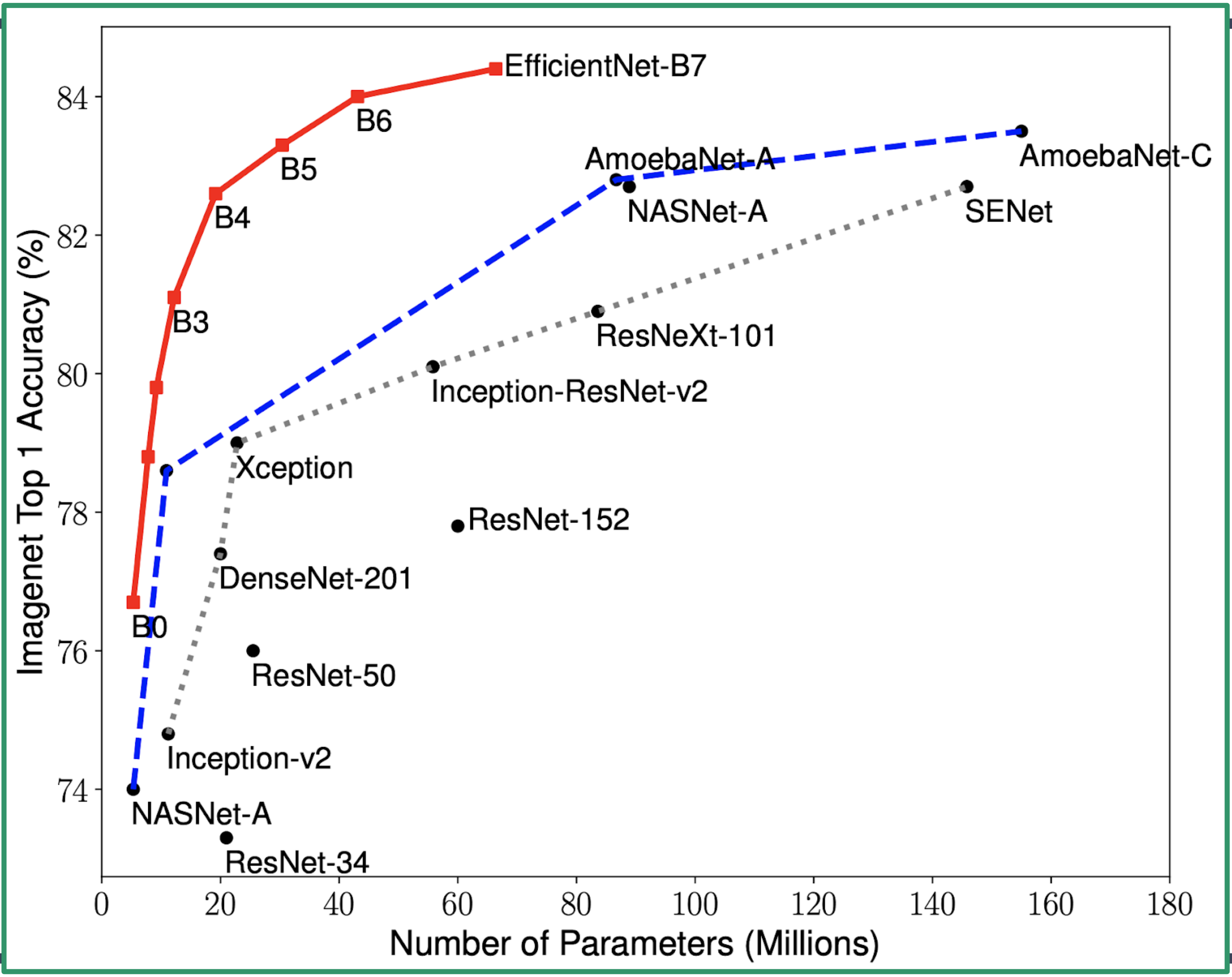
# Representational Power of Deep Networks

Hypo





## Representational Power of Deep Networks

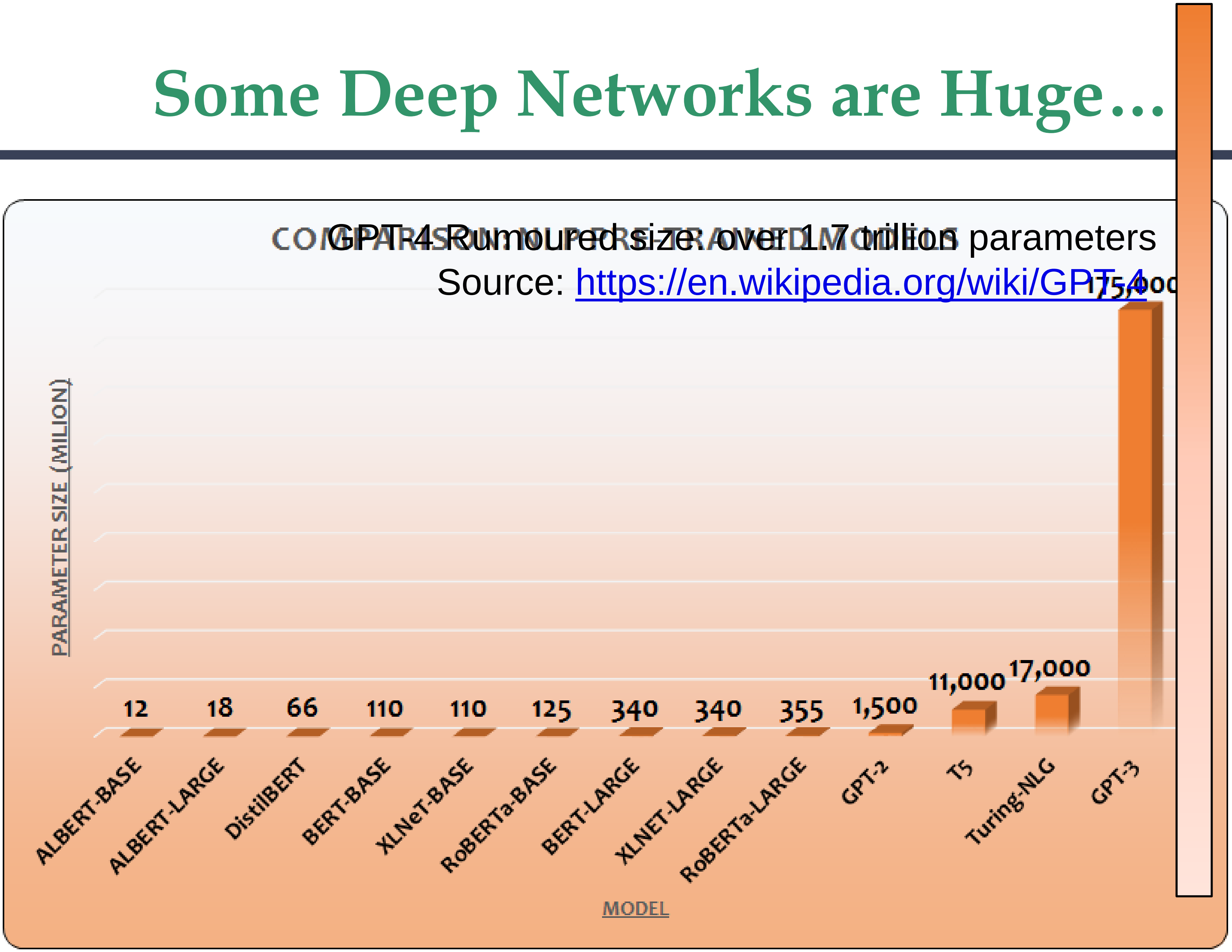


Google AI Blog: EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling (May 2019)

# Representational Power of Deep Networks

- What is the reason for these trends?
  - At some stage when training a ML model, no matter how much data you have, you run up against the limit of the complexity of the hypothesis language
  - Methods with strong bias (low hypothesis language complexity) tend to perform worse than methods with lower bias, provided that the problem requires a complex hypothesis
- With Deep NNs, it is possible to keep adding to the complexity of the hypothesis language
  - Keep adding layers and increasing nodes per layer
  - Worth doing if it's a complex problem and you have very large amounts of data to support the learning process
- Another way to think of this: a neural network is a distributed model of the data that it was trained on
  - With a deep and large network, it is possible to model a large and complex dataset with high fidelity

## Some Deep Networks are Huge…



GPT-4 Rumoured size: over 1.7 trillion parameters
Source: https://en.wikipedia.org/wiki/GPT-4

Source: https://medium.com/analytics-vidhya/openai-gpt-3-language-models-are-few-shot-learners-82531b3d3122

## … And Enable Exciting Applications

- OpenAI’s ChatGPT: conversation, summarisation, code generation, works in many languages
- OpenAI’s DALL-E: image creation
- OpenAI’s Whisper: speech to text
- Google’s Gemini: multi-modal processing of text, images and audio
- Require **vast** amounts of data for training; GPT-4: approximately 500 billion words

Photo of a dolmen in Ireland being used as a portal by aliens
– Michael Madden using DALL-E 3

https://theconversation.com/googles-gemini-is-the-new-ai-model-really-better-than-chatgpt-219526

# Deep NNs and Big Data

- Deep neural nets fit very well with trends in big data
  - Organisations have huge amounts of data that they need to analyse
  - Deep NNs require huge amounts of data to train well

- Traditional ML algorithms were designed for structured data, but deep NNs can also work well with unstructured data
  - These include perception tasks such as vision and speech
  - For many such tasks, it is not easy to come up with a small set of training cases that cover almost all cases
  - Better to provide a large number of examples that each cover a limited number of scenarios

# Training Deep NNs Leads to New Challenges …

- Need efficient computation
  - GPUs
  - Efficient vector calculations (handled by libraries, e.g. Tensorflow and Keras)
- Need regularisation methods to avoid overfitting
  - Have a hypothesis language with high complexity
  - Low bias => helps to avoid underfitting
  - Increased risk of overfitting
- Need new training algorithms
  - Standard backprop runs into problems in deep architectures
  - Credit not always assigned fully back through all layers
  - Early layers in particular tend to converge to 1/0 weights

## Training Deep NNs Leads to New Challenges …

- Need to be able to re-use models
  - Direct model re-use
  - Pre-training
  - Domain adaptation
  - Transfer learning
- Need to consider other architectures beyond fully-connected feed-forward nets
  - Sparsely connected nets
  - Convolutional networks
  - RNNs and LSTMs
- Need very large amounts of data
- Need to be able to handle unstructured data

## We Will Focus On …

- Unstructured Data and Multi-Class Classification
- Methods to Avoid Overfitting
  - Data Augmentation
  - Early Stopping
  - L2 regularisation
- Training Algorithms
  - Mini-Batch Gradient Descent
  - Backprop with Momentum, RMSprop
  - Adam Optimiser
- Model Re-Use
  - Pre-Training
  - Transfer Learning
- Convolutional Networks

# Structured vs Unstructured Data

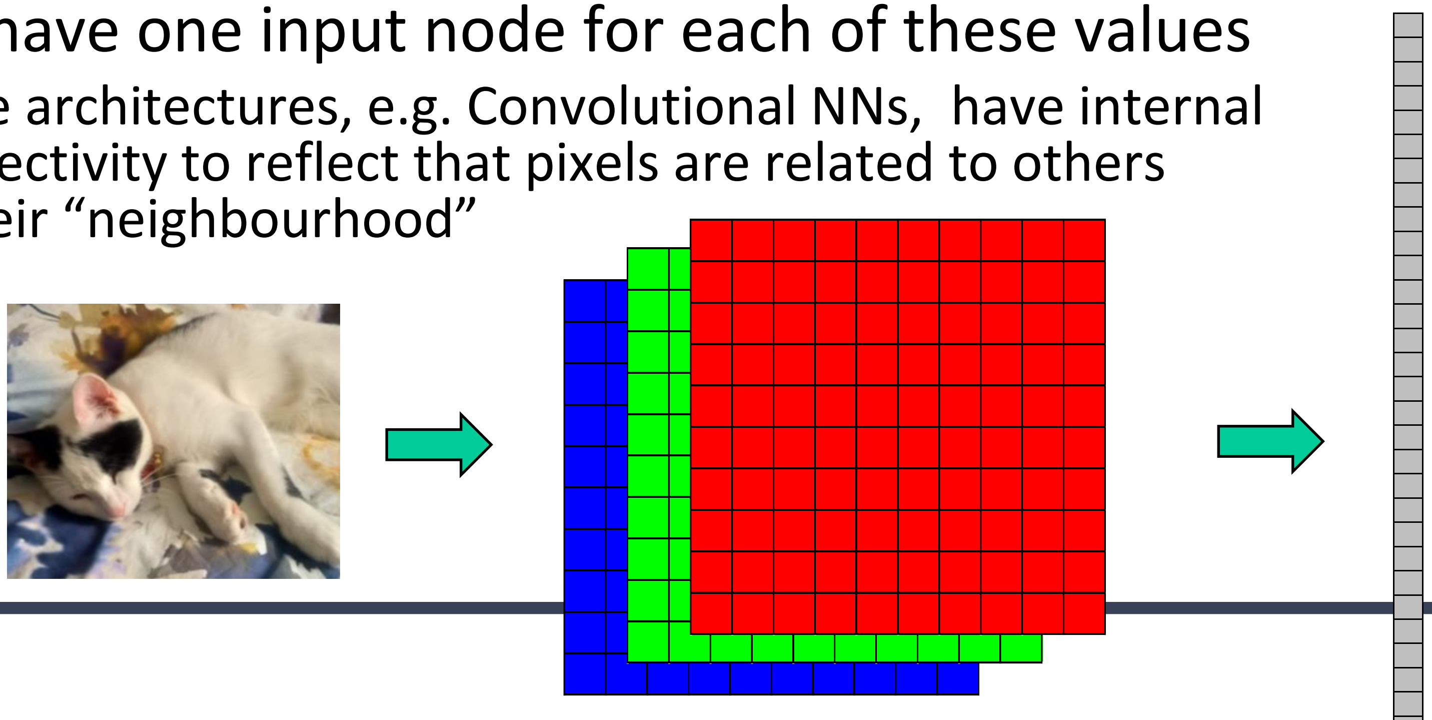
- Structured Data: typically organised in a table

| Movie | Mary | John | I Like |
|---|---|---|---|
| Star Wars 1 | 4.5 | 4 | Yes |
| LOTR 2 | 1 | 5 | No |
| Mov3 | 1 | 4 | No |
| Mov4 | 1.5 | 4 | No |
| Mov5 | 2.5 | 3 | No |

- Unstructured Data:
  - Photos
  - Movies
  - Audio
  - Documents

## Unstructured Data: Processing Images with a Neural Net

- Most neural net architectures require fixed length input vectors
- Before analysis, resize all images to fixed width & height:
  - E.g. 128 x 128 x 3 (RGB)
  - Then map this into a vector of pixels, with 49,512 values in this case
  - In Python, where image is numpy array: image.reshape(width*height*depth,1)
- Practical note: **always** good to normalize NN inputs (0-1 or z-Norm)
- In NN, have one input node for each of these values
  - Some architectures, e.g. Convolutional NNs, have internal connectivity to reflect that pixels are related to others in their “neighbourhood”



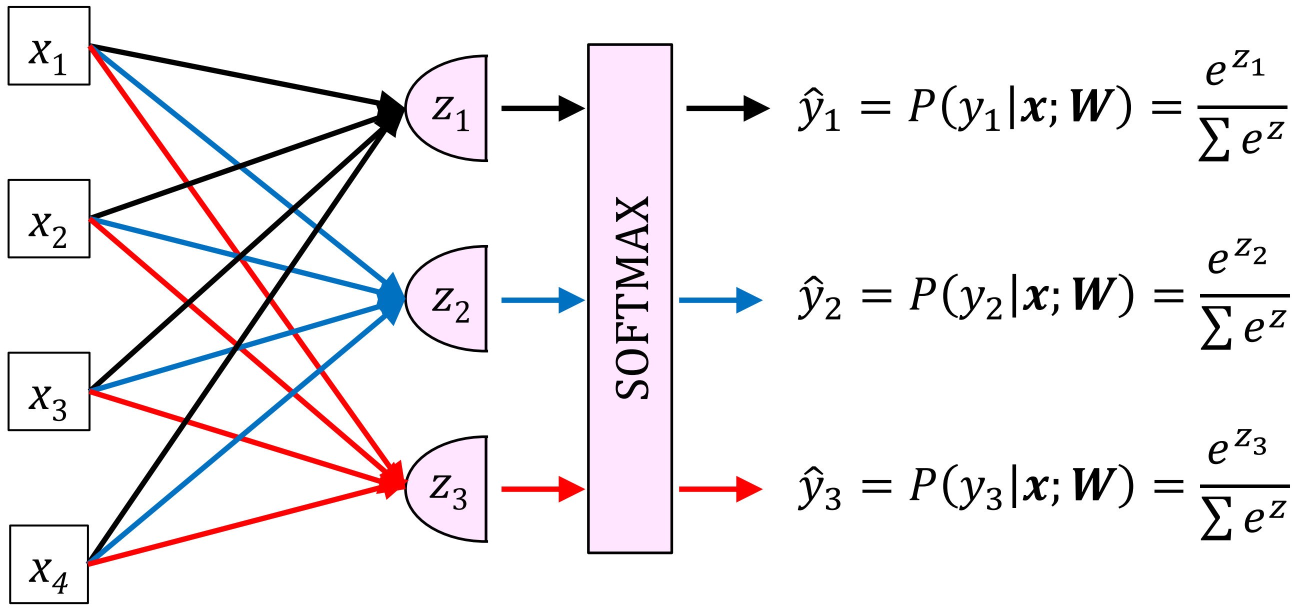
# Multi-Class Classification: Softmax Output

Example: 4 inputs, 3 possible classes.

Example calculations: see compare-softmax-logistic.xlsx

Represent classes as 3 outputs that can be 0 or 1, and only one of them can be 1 at any time ("one-hot encoding").

| **Class 1** | **Class 2** | **Class 3** |
|---|---|---|
| $y_1 = 1$ | $y_1 = 0$ | $y_1 = 0$ |
| $y_2 = 0$ | $y_2 = 1$ | $y_2 = 0$ |
| $y_3 = 0$ | $y_3 = 0$ | $y_3 = 1$ |



$$\hat{y}_1 = P(y_1|\boldsymbol{x};\boldsymbol{W}) = \frac{e^{z_1}}{\sum e^{z}}$$

$$\hat{y}_2 = P(y_2|\boldsymbol{x};\boldsymbol{W}) = \frac{e^{z_2}}{\sum e^{z}}$$

$$\hat{y}_3 = P(y_3|\boldsymbol{x};\boldsymbol{W}) = \frac{e^{z_3}}{\sum e^{z}}$$

The Softmax function replaces the standard Sigmoid function used in binary classification.
It rescales the $z$ values so that the $\hat{y}$ values sum to 1, as required for a probability distribution.

# Multi-Class Classification: Backprop with Softmax

- Since softmax is a direct generalisation of the binary logistic output, the cost function is directly related to what we saw before
- Loss Function for one example, where $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are now vectors of $C$ elements, one for each class (also the Cost Function for Stochastic GD):

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{j=1}^{C} \left( y_j \log(\hat{y}_j) \right)$$

- Cost Function for Batch GD is average loss over all *N* training cases:

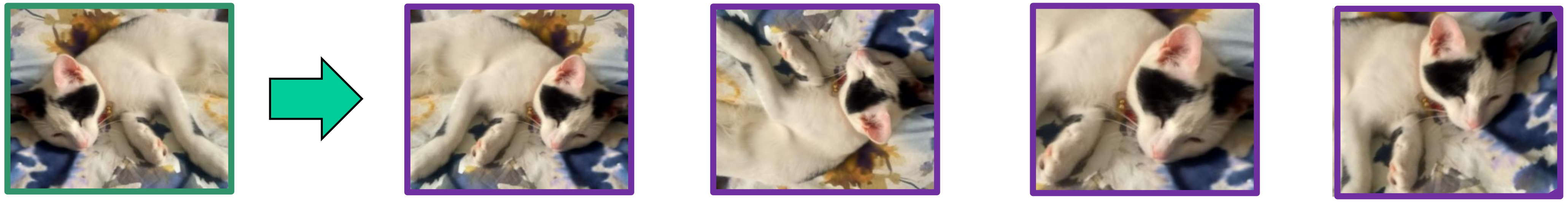
$$J(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} \left( y_c^{(i)} \log\left( \hat{y}_c^{(i)} \right) \right)$$

- The partial derivatives are the same as we had in Topic 3, again just generalised because there are now *C* output nodes:

$$\Delta z_j^{[L]} = a_j^{[L]} - y_j$$

# Methods to Avoid Overfitting: Dataset Augmentation

- The best way to improve generalisation on a ML model is to train it with more data
  - Data is often limited; costly to collect and to label
  - Dataset Augmentation is a strategy to create "fake" data from existing cases: flip, crop, rotate, translate, etc
  - Apply same operations to annotations such as object bounding boxes
  - Not quite as valuable as real new data, but nonetheless helps training



- Make sure not to apply operations that would change the correct class, e.g. in digit recognition, don't rotate 180°

# Methods to Avoid Overfitting: Early Stopping

- A classic sign of overfitting is that the performance on a validation set is much worse than that on the training set
- In early stages of NN training, when it is far from converging, overfitting is never an issue, but it can become an issue as training proceeds, particularly if network is complex relative to dataset size

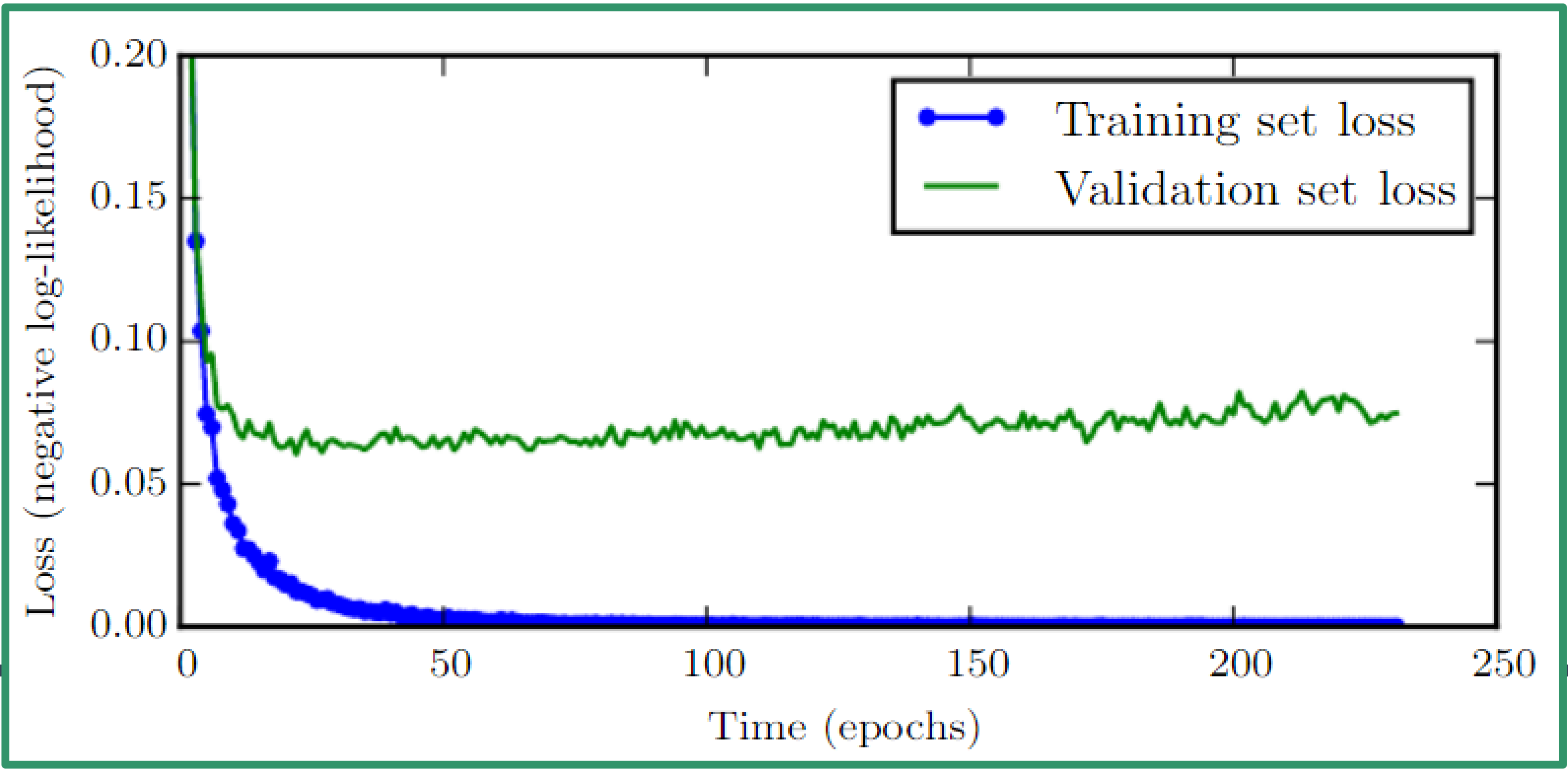


Image from "Deep Learning" by Goodfellow, Bengio & Courville

# Methods to Avoid Overfitting: Early Stopping

How to implement Early Stopping:

- Repeat at regular intervals (e.g. every 100 iterations):
  - Test the current network on a validation set
  - Compare the validation set performance at the current time with the performance from the last time
    - If the current performance is better, save the model parameters
- When finished training, use the last saved parameters instead of the final parameters

# Methods to Avoid Overfitting: $L_2$ Regularization

- The idea behind regularization is that we add an extra penalty term to the cost function to penalise more complex networks
  - A network is effectively smaller if some weights have 0 values
  - Regularization drives weights to low or 0 values
- Various approaches include:
  - $L_1$ Regularization: penalty based on $L_1$ norm of a weight vector ***w***: $\|\boldsymbol{w}\|_1 = \sum|w|$ (Manhattan norm)  Python: numpy.linag.norm(w,1)
  - $L_2$ Regularization: penalty based on $L_2$ norm of a weight vector ***w***: $\|\boldsymbol{w}\|_2 = \sqrt{\sum w^2}$ (Euclidean norm) Python: numpy.linag.norm(w,2) – *note that the penalty used is actually the square of the $L_2$ norm*
  - Dropout Regularization: different approach where a fraction of nodes selected at random in an iteration are dropped out: incoming weights and outgoing weights set to 0

# Methods to Avoid Overfitting: $L_2$ Regularization - Cost Function

- Our cost function *J* is expanded to include the penalty terms:

$$J_{W,b}(\boldsymbol{y},\hat{\boldsymbol{y}}) = \frac{1}{N}\sum_{i=1}^{N} L(\boldsymbol{y},\hat{\boldsymbol{y}}) + \frac{\lambda}{2N}\|\boldsymbol{W}\|_2^2$$

  - Here, λ is the regularization parameter that determines the importance of this penalty to the overall cost, 2 is a convenience term and 1/N means you don't have to change λ if you change training set (or mini-batch) size
  - λ is another hyperparameter that we will have to experiment to find a good value for: could start with 0.1 and adjust up/down in powers of 10
  - I'm using ***W*** to represent a matrix of all weights in our NN, but depending on your data representation, you may have to loop over multiple weight vectors/matrices to compute a single sum of all squared weights
  - Note that the bias terms are **not** normally included in regularization: that would constrain the linear equations to go through origin

## Methods to Avoid Overfitting: L$_2$ Regularization - Backprop

- This is a new cost function to optimise
  - We can calculate the partial derivative of $J$ with respect to $W$ terms
  - This results in new terms added to our equations for $\Delta W$:

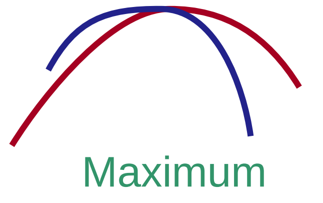
$$\Delta W_{j,i}^{[l]} = \Delta z_j^{[l]} \, a_i^{[l-1]} + \frac{\lambda}{N} W_{j,i}^{[l]}$$

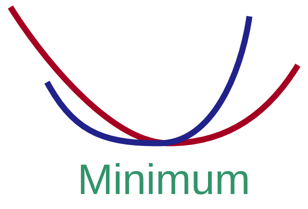
- Reminder: Gradient Descent weight update: $W_{j,i}^{[l]} \mathrel{-}= \alpha \, \Delta W_{j,i}^{[l]}$
- Therefore, in every iteration, every weight is now being reduced by an additional amount: $-\frac{\alpha\lambda}{N} W_{j,i}^{[l]}$

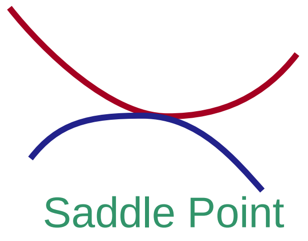
- For this reason, L$_2$ regularization is also called Weight Decay

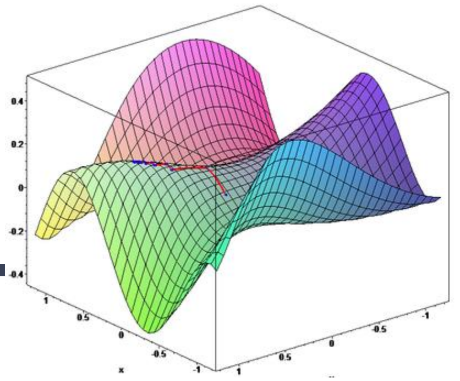
# Training Algorithms: Challenges of Local Minima and Saddle Points

- Objective of Gradient Descent is to reach a place in parameter space where gradient of cost function is 0
  - For convex optimisation problem, this is global best solution
  - But NN training is not a convex optimisation problem
- Three situations in which gradient is 0
  - Local maximum (not an issue for Gradient Descent)
  - Local minimum (minimum simultaneously in all dimensions)
  - Saddle point (minimum in some dims, maximum in others)
- For a large number of dimensions, saddle points are the most likely to arise



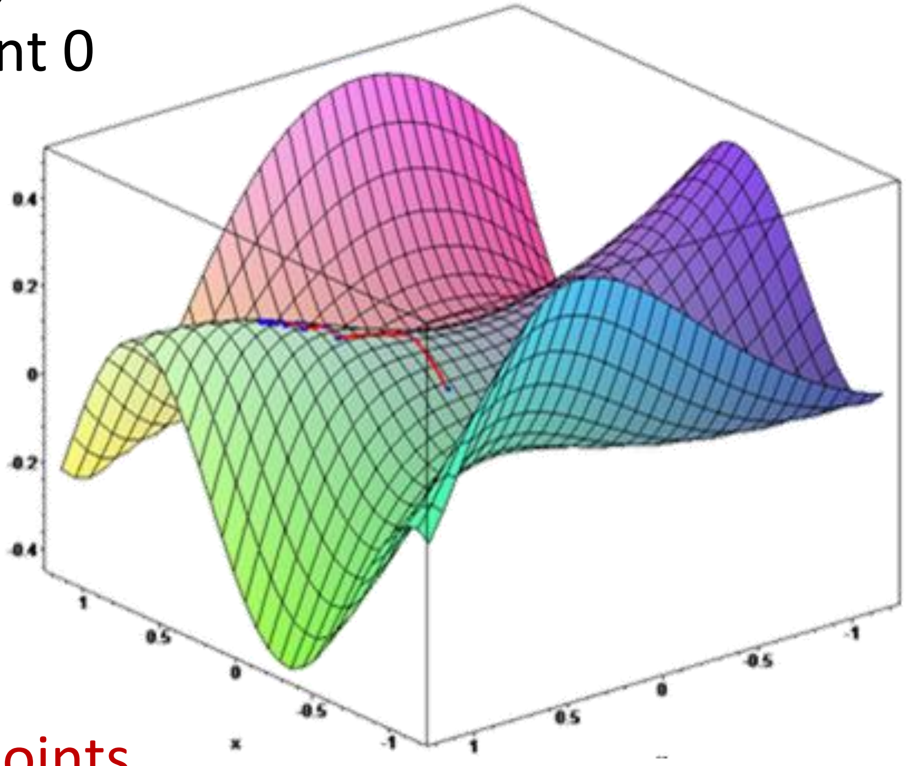






# Training Algorithms: Avoiding Saddle Points

- Overall training goal is to find weights that minimise the cost function, evaluated against all training data
  - Batch Gradient Descent directly optimises this, so it heads directly ‘downhill’ towards a place with gradient 0
  - As such, will terminate at local minima or saddle points
- Stochastic Gradient Descent is less direct:
  - Each step determined by a randomly selected training case
  - As such, less direct & more random in the steps it takes
  - This can actually help to break free of saddle points



# Training Algorithms: Mini-Batch Gradient Descent

- Stochastic GD can’t make efficient use of vectorised calculations
  - Only processing a singe training case at a time
  - Does not make full use of RAM and parallel processing power of GPU/CPU
- Batch GD makes full use of vectorisation
  - However, can get stuck at sub-optimal solutions
  - Can be slower for large datasets: feed-forward all before doing any updates
  - For very large datasets, can’t hold all data in RAM at once, so vectorisation becomes infeasible anyway
- Mini-Batch Gradient Descent is a good compromise, and a common default:
  - Divide full dataset into mini batches
  - Typical batch size *B* = 64, 128 or 256 cases (v. small relative to full training set size, *N*)
  - Loop over mini-batches, and do one iteration of the training algorithm on 1 mini-batch
  - One epoch takes *N*/*B* iterations

# Training Algorithms: Mini-Batch GD - Adjustment to Backprop

For Mini-Batch GD or Batch GD, we slightly adjust the training algorithm from Topic 3

- Here, *N* is the number of cases in the **batch** (not the size of full training set)
- We add an extra subscript *n* for nth training case:
  $a_{(n)}$ is output for case *n*, $y_{(n)}$ is its expected value

**Forward-propagation Step:**

- For each case *n* = 1 to *N*, propagate forward to compute activations for each node *j* in each layer: $a^{[l]}_{(n)j}$

**Backpropagation Step:**

- **In the output layer**, calculate $\Delta z$ terms separately for each node *j* and each case *n*:
  $\Delta z^{[L]}_{(n)j} = a^{[L]}_{(n)j} - y_{(n)j}$
- Then, when calculating $\Delta W$ and $\Delta b$ terms, average over all *N* cases:

$$\Delta W^{[L]}_{j,i} = \frac{1}{N}\sum_{n=1}^{N}\left(\Delta z^{[L]}_{(n)j}\ a^{[L-1]}_{(n)i}\right), \quad \Delta b^{[L]}_{j} = \frac{1}{N}\sum_{n=1}^{N}\left(\Delta z^{[L]}_{(n)j}\right)$$

**Notes:**

- When working with batches, we must be careful not to mix up the Loss function *L* (1 case) and the Cost function *J* (the average loss over all cases)
- Essentially, $\Delta z$ terms are $\partial L/\partial z$ (partial derivatives of Loss function w.r.t. *z*), whereas $\Delta b$ terms are $\partial J/\partial b$ (partial derivatives of Cost function w.r.t. *b*), and likewise for $\Delta W$

# End of Topic 4