IMPERIAL

Machine Learning for Neuroscience

ML4NS

Artificial Neural Networks

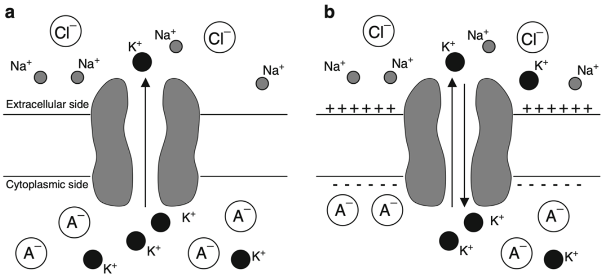
Payam Barnaghi
Department of Brain Sciences &
School of Convergence Science in Human and Artificial Intelligence
Imperial College London

January 2025

IMPERIAL

Hodgkin-Huxley neuron model

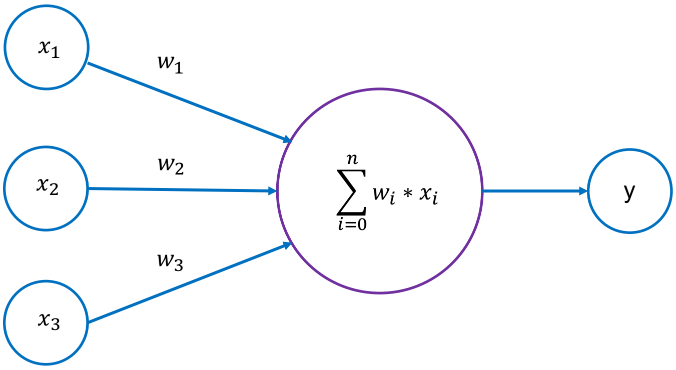
$$E_K = -\frac{RT}{zF}\ln\frac{[K^+]_{\text{in}}}{[K^+]_{\text{out}}}. \qquad (1.1)$$



This is a computational model of a biological neuron; in contrast, in ML, we are interested in modelling an artificial neuron that can help us to solve (machine) learning and decision-making problems.

Image from: http://www.math.pitt.edu/~bdoiron/assets/ermentrout-and-terman-ch-1.pdf

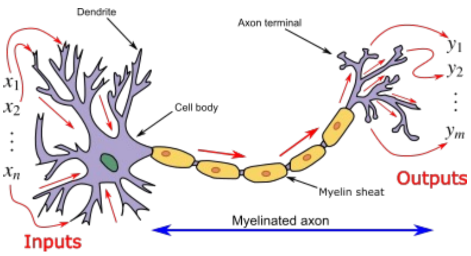
Artificial Neuron

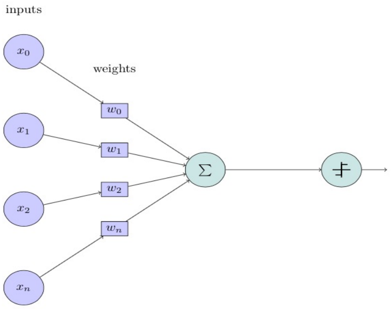


3

3

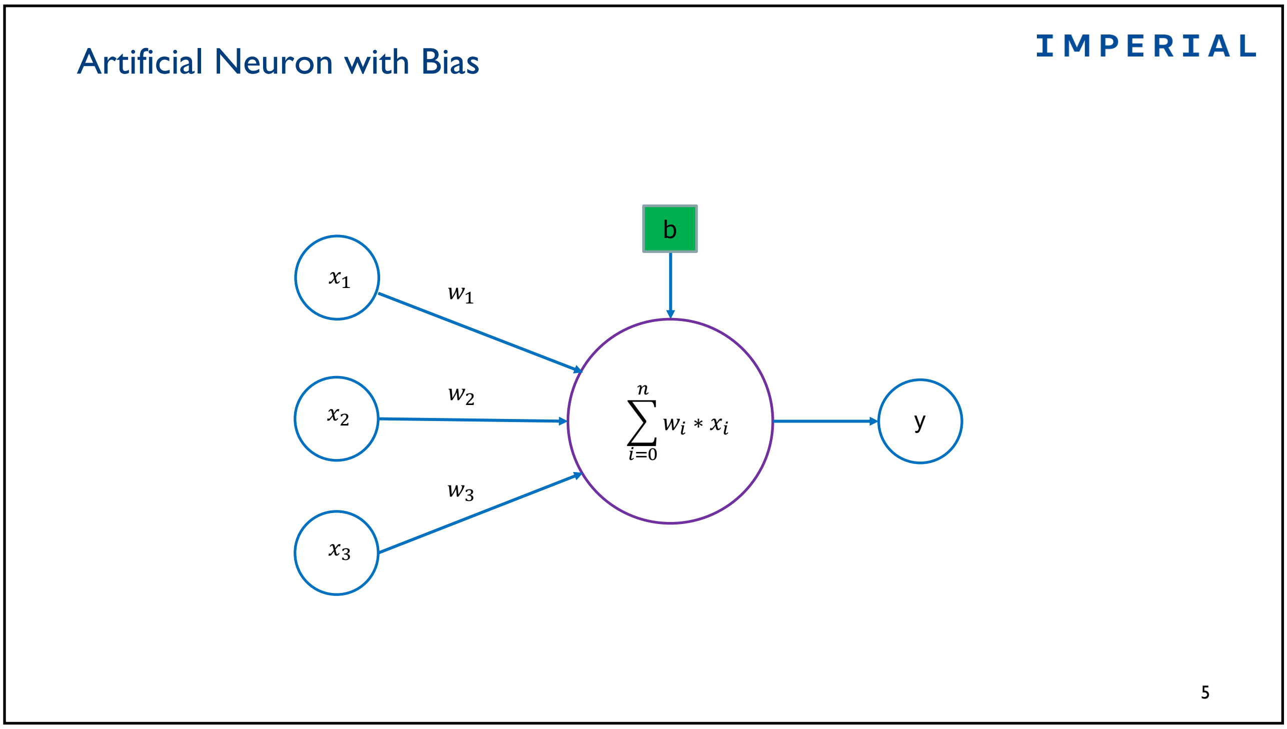
Artificial neuron vs biological neuron





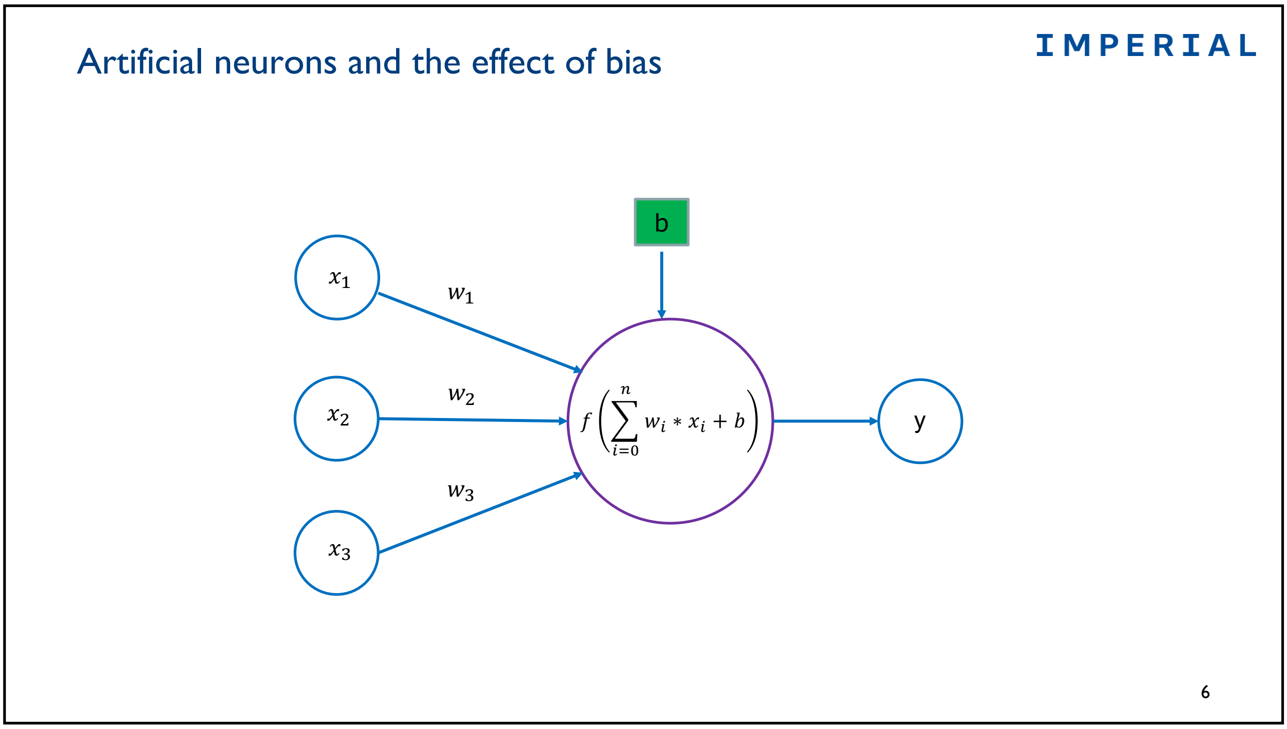
4

4



Artificial Neuron with Bias

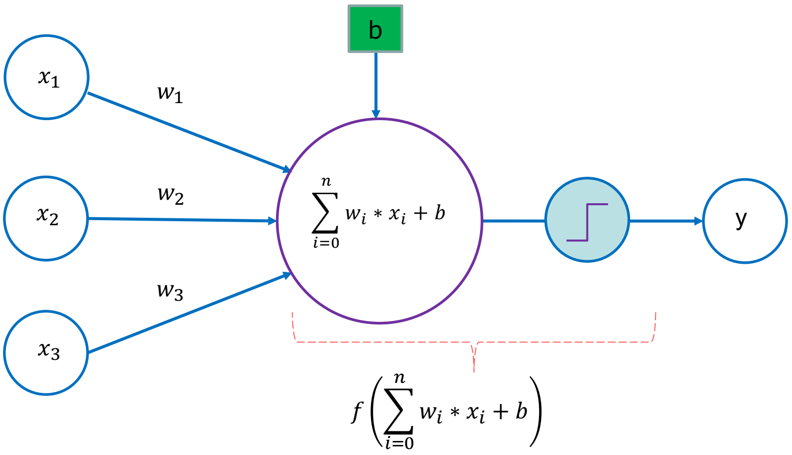
5



Artificial neurons and the effect of bias

6

NN with an Activation Function

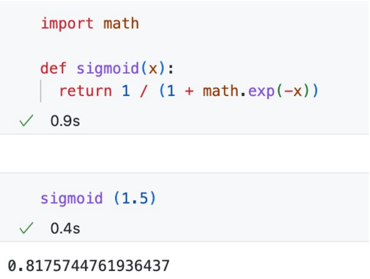

b
x_1
x_2
x_3
w_1
w_2
w_3
$\sum_{i=0}^{n} w_i * x_i + b$
y
$f\left(\sum_{i=0}^{n} w_i * x_i + b\right)$

Limitations of linear models

- Linearity implies the *weaker* assumption of *monotonicity*, i.e., that any increase in our feature must either always cause an increase in our model’s output (if the corresponding weight is positive) or always cause a decrease in our model’s output (if the corresponding weight is negative).

- However, this is not always true or helpful in real-world applications. For individuals with a body temperature above 37C (98.6F), higher temperatures indicate greater risk. However, this risk does not linearly increase when, for example, the temperature drops below 37 or goes above 38.5.

Sigmoid function



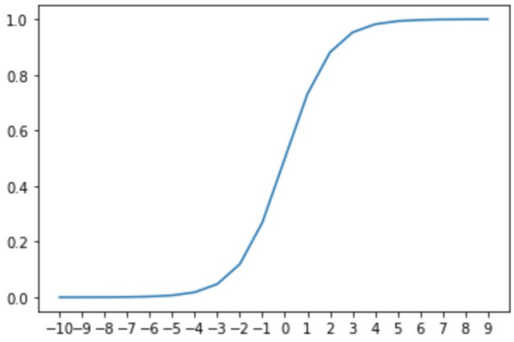
```
import math

def sigmoid(x):
    return 1 / (1 + math.exp(-x))
```

✓ 0.9s

```
sigmoid (1.5)
```

✓ 0.4s

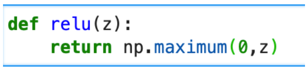
```
0.8175744761936437
```



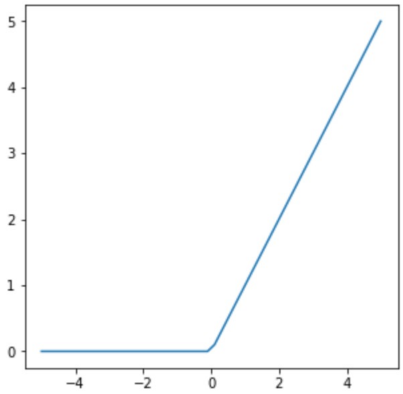
$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

Rectified Linear Units (ReLU)



```
def relu(z):
    return np.maximum(0,z)
```



ReLU

IMPERIAL

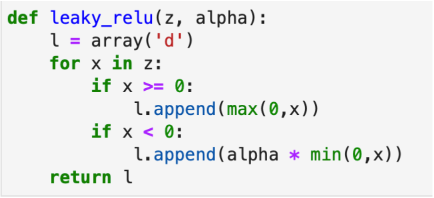
- The half rectifying nonlinearity simulates some of the properties of biological neurons:
  - For some inputs, biological neurons are completely inactive (sparse activation).
  - For some inputs, a biological neuron's output is proportional to its inputs.
  - Most of the time, biological neurons operate in a way that they are inactive (i.e., they have sparse activation).

11

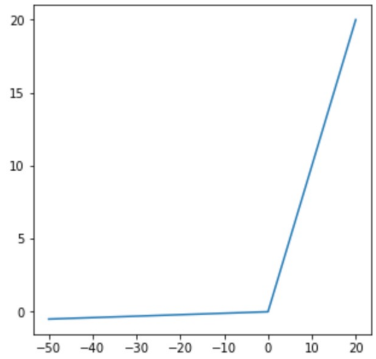
11

Leaky ReLU

IMPERIAL



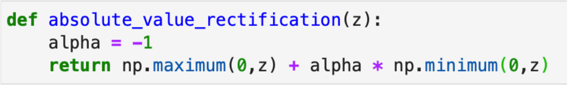
```
def leaky_relu(z, alpha):
    l = array('d')
    for x in z:
        if x >= 0:
            l.append(max(0,x))
        if x < 0:
            l.append(alpha * min(0,x))
    return l
```



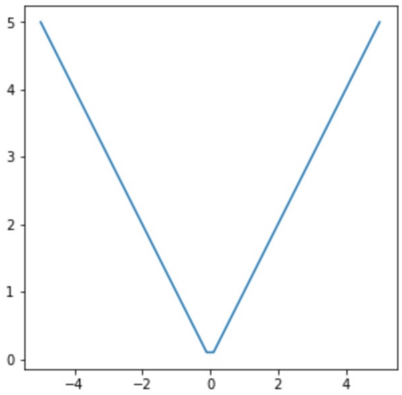
12

12

Absolute Value Rectification



```
def absolute_value_rectification(z):
    alpha = -1
    return np.maximum(0,z) + alpha * np.minimum(0,z)
```



13

13

Nonlinearity in machine problems

- In the past lectures, we have seen that the decision space and feature representation in most machine learning problems is not linear.
- So, the question is how to add nonlinearity to the neural network models.

14

14

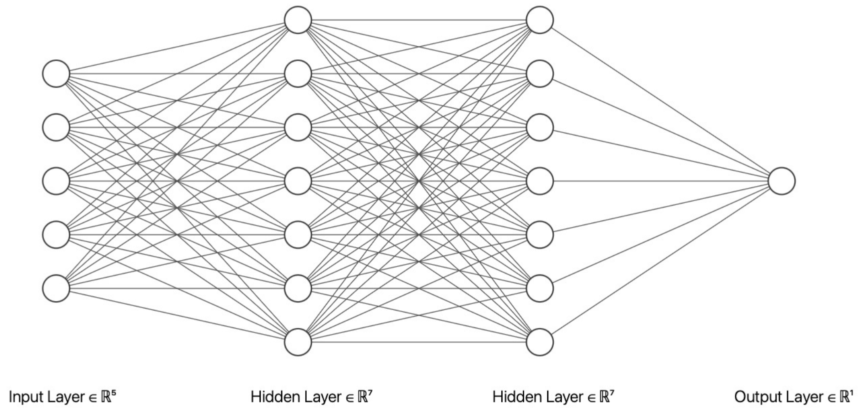
Hidden layers

IMPERIAL

- We can overcome the limitations of linear models by also incorporating one or more hidden layers (with activation functions).
- The easiest way to do this is to stack several fully connected layers on top of each other.
- Each layer feeds into the layer above it until the model generates outputs.
- This architecture is commonly called a Multilayer Perceptron, often abbreviated as `MLP`.

Multilayer perceptron architecture

IMPERIAL


Input Layer ∈ $\mathbb{R}^5$
Hidden Layer ∈ $\mathbb{R}^7$
Hidden Layer ∈ $\mathbb{R}^7$
Output Layer ∈ $\mathbb{R}^1$

The figure is drawn using: https://alexlenail.me/NN-SVG/

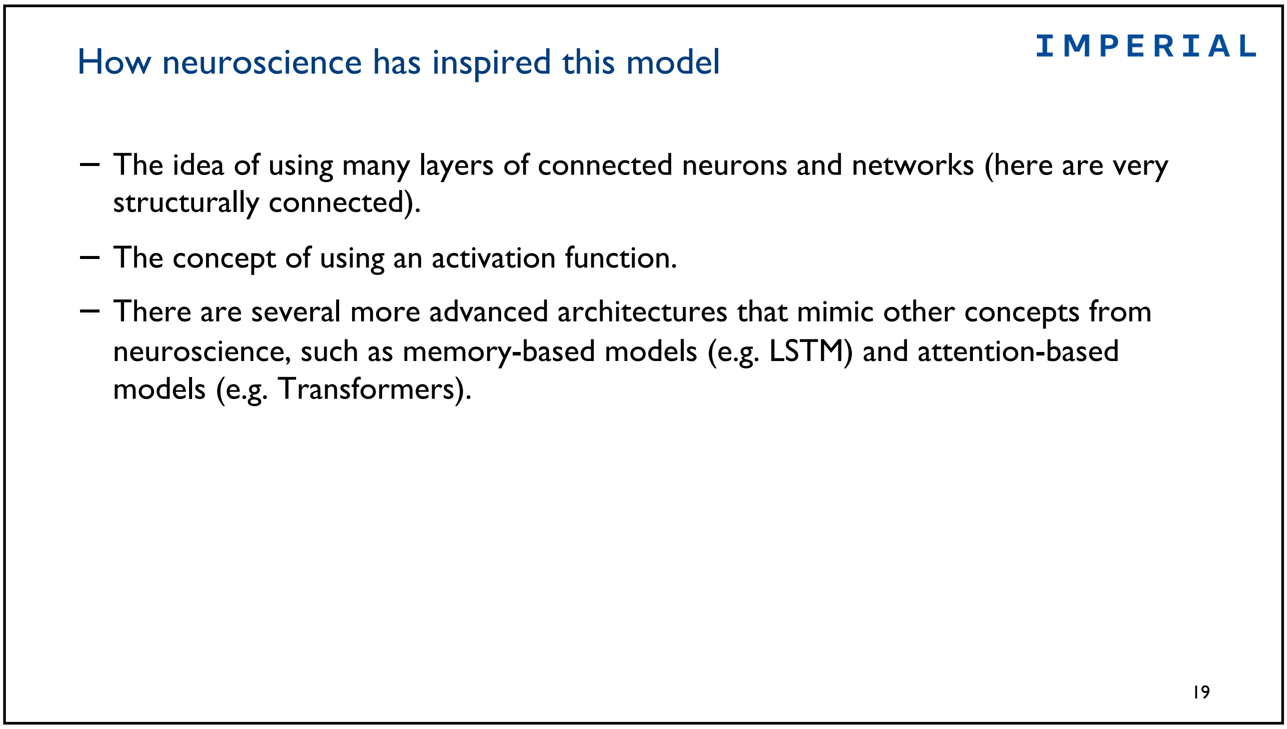
MLP architecture

- We can think of the first `L-1` layers as our representation and the final layer as our linear predictor.

Layers in a neural network

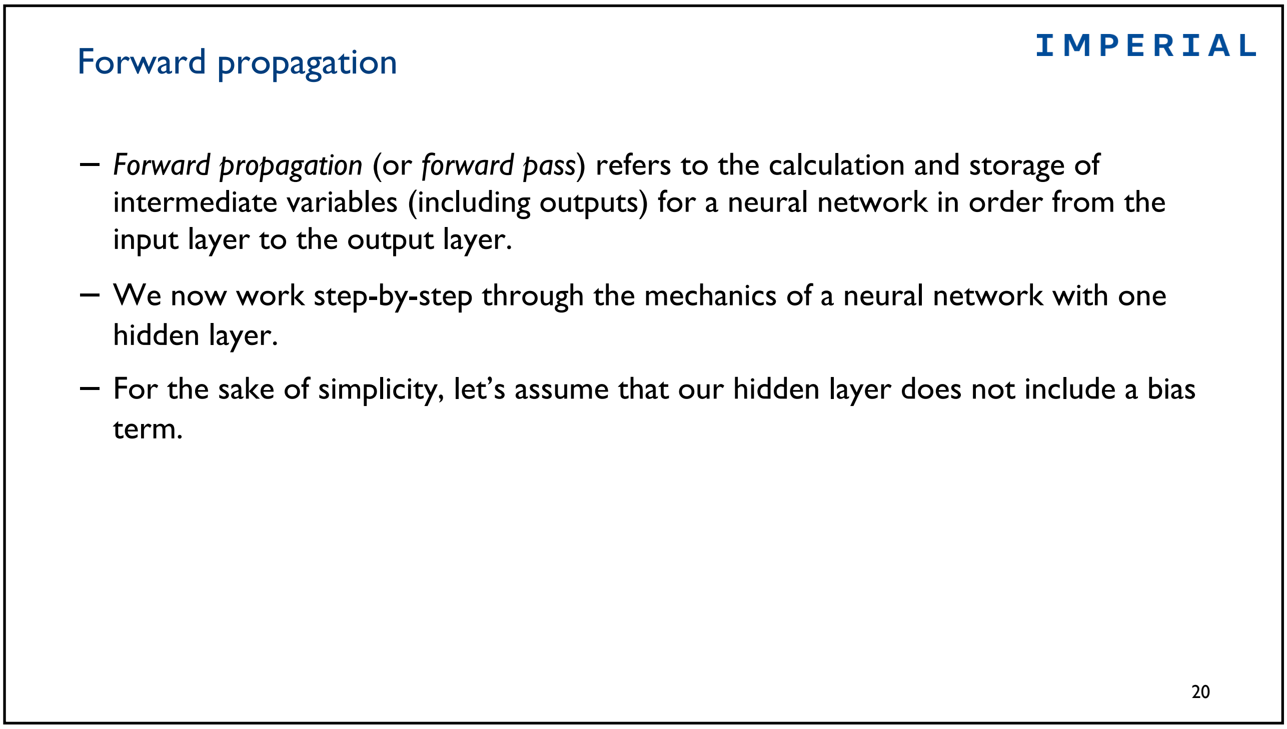
- Input layer
- Hidden layers
- Output layer

- The number of hidden layers determines the depth of the network. The term “deep learning” originates from this terminology.
- The width of the network is determined by the number of nodes (dimensionality) in the hidden layer(s).



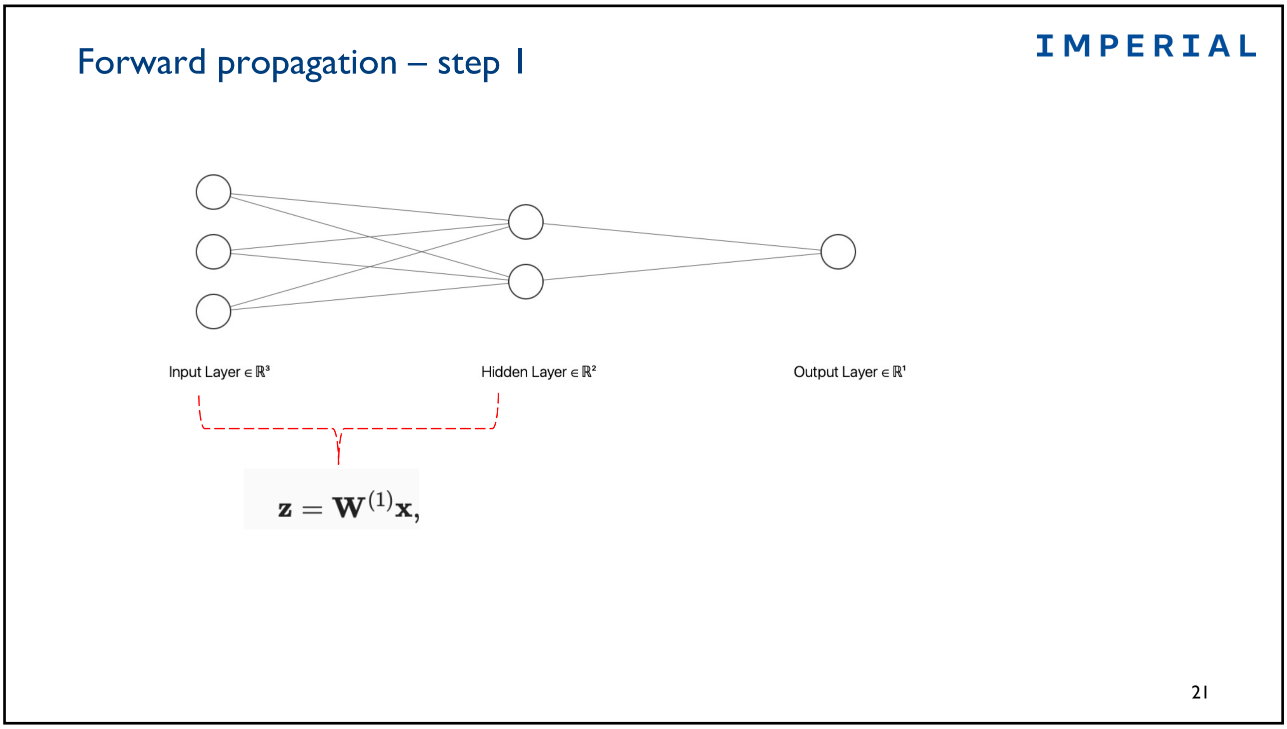
How neuroscience has inspired this model

- The idea of using many layers of connected neurons and networks (here are very structurally connected).
- The concept of using an activation function.
- There are several more advanced architectures that mimic other concepts from neuroscience, such as memory-based models (e.g. LSTM) and attention-based models (e.g. Transformers).



Forward propagation

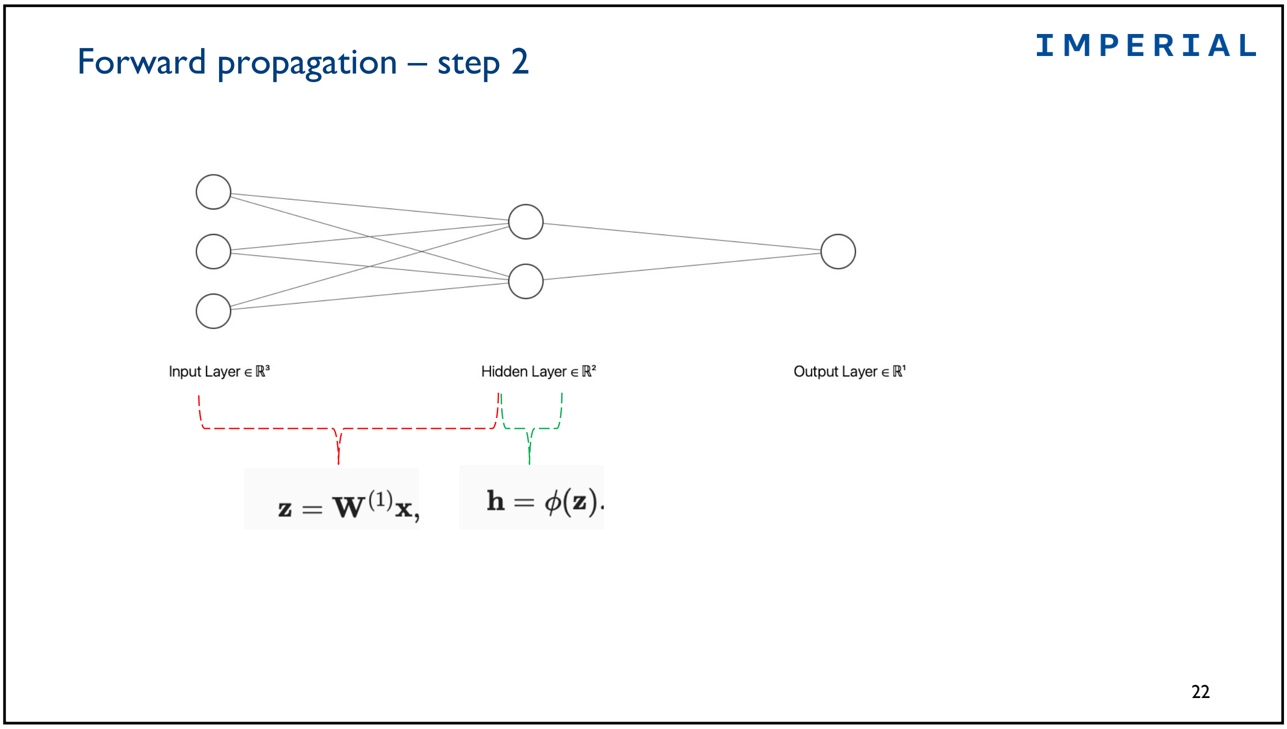
- *Forward propagation* (or *forward pass*) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.
- We now work step-by-step through the mechanics of a neural network with one hidden layer.
- For the sake of simplicity, let’s assume that our hidden layer does not include a bias term.



Forward propagation – step 1

Input Layer $\in \mathbb{R}^3$
Hidden Layer $\in \mathbb{R}^2$
Output Layer $\in \mathbb{R}^1$

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$$



Forward propagation – step 2

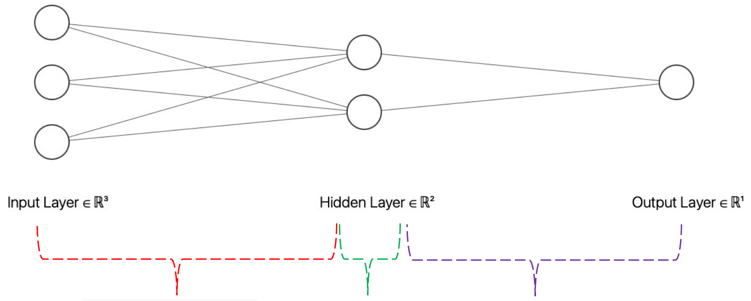
Input Layer $\in \mathbb{R}^3$
Hidden Layer $\in \mathbb{R}^2$
Output Layer $\in \mathbb{R}^1$

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$$

$$\mathbf{h} = \phi(\mathbf{z}).$$

Forward propagation – step 3

IMPERIAL



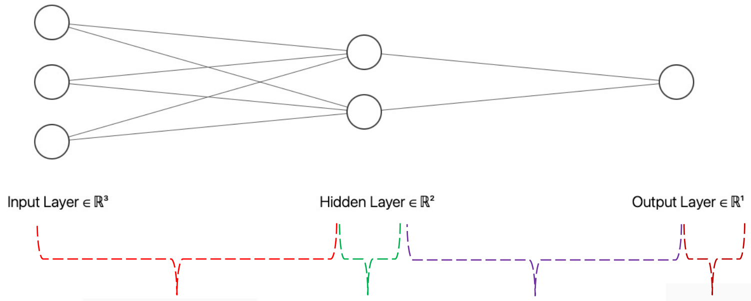
$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$ $\mathbf{h} = \phi(\mathbf{z}).$ $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$

23

23

Forward propagation – Loss function (l)

IMPERIAL



$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$ $\mathbf{h} = \phi(\mathbf{z}).$ $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$ $L = l(\mathbf{o}, y).$

24

24

Back propagation

IMPERIAL

- *Backpropagation* refers to the method of calculating the gradient of neural network parameters.
- The method traverses the network in reverse order, from the output to the input layer, according to the *chain rule* from calculus.
- The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters.

Training a neural network - I

IMPERIAL

- Often, we start with random weight initialisation.
- And then go through multiple iterations applying all the training samples (forward propagation).
- Then, calculate the error/loss based on a loss function.
- After that, we use the error/loss value to update the weights.
- We repeat this training iteration several times; in NN terms, each iteration of trying all the training samples in one round is called an "*epoch*".

Random initialisation**

- If you are interested in the effect of initialisation parameters, see:
- https://stackoverflow.com/questions/49433936/how-do-i-initialize-weights-in-pytorch
- https://pytorch.org/docs/stable/nn.init.html
- *Understanding the difficulty of training deep feedforward neural networks, Glorot, X. & Bengio, Y. (2010)* https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

Training neural networks

- When training neural networks, forward and backward propagation depend on each other.
- In forward propagation, we traverse the computational graph in the direction of dependencies and compute all the variables on its path.
- These are then used for backpropagation, where the compute order on the graph is reversed.

Training a neural network - I

IMPERIAL

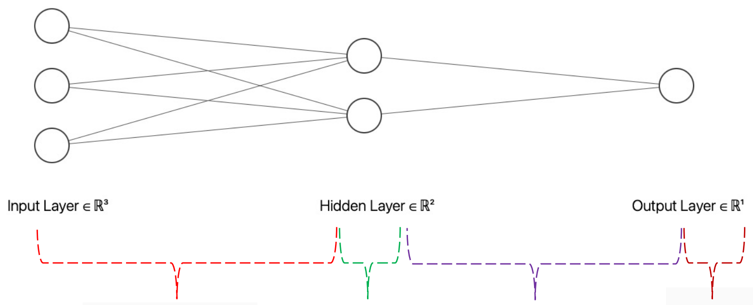
- Now, the question is how to update the weights to optimise the model to obtain less error/loss.
- We need a way to calculate the intermediary values for backpropagation and updating the weights.
- We need a hyperparameter to control the rate of change. This is often done by a hyperparameter called “*learning rate*”.

29

29

Forward propagation – Loss function (I)

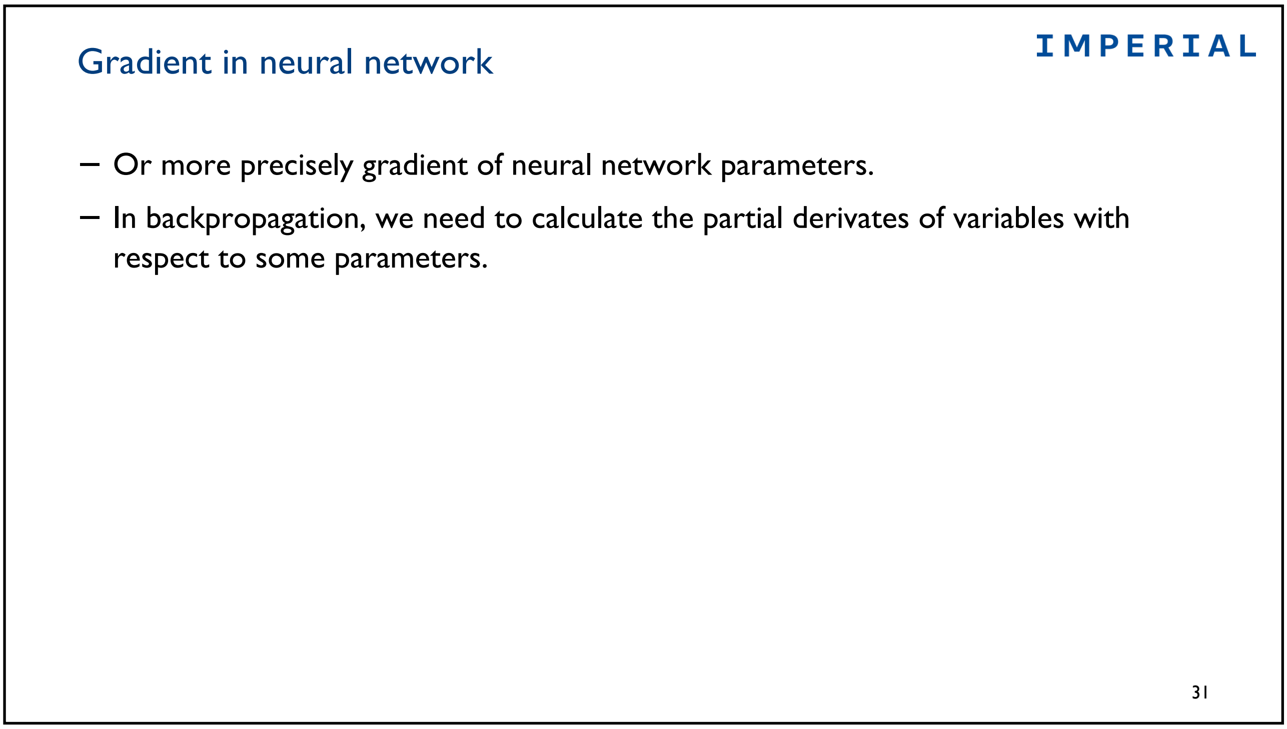
IMPERIAL



$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \quad \mathbf{h} = \phi(\mathbf{z}). \quad \mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \quad L = l(\mathbf{o}, y).$$

30

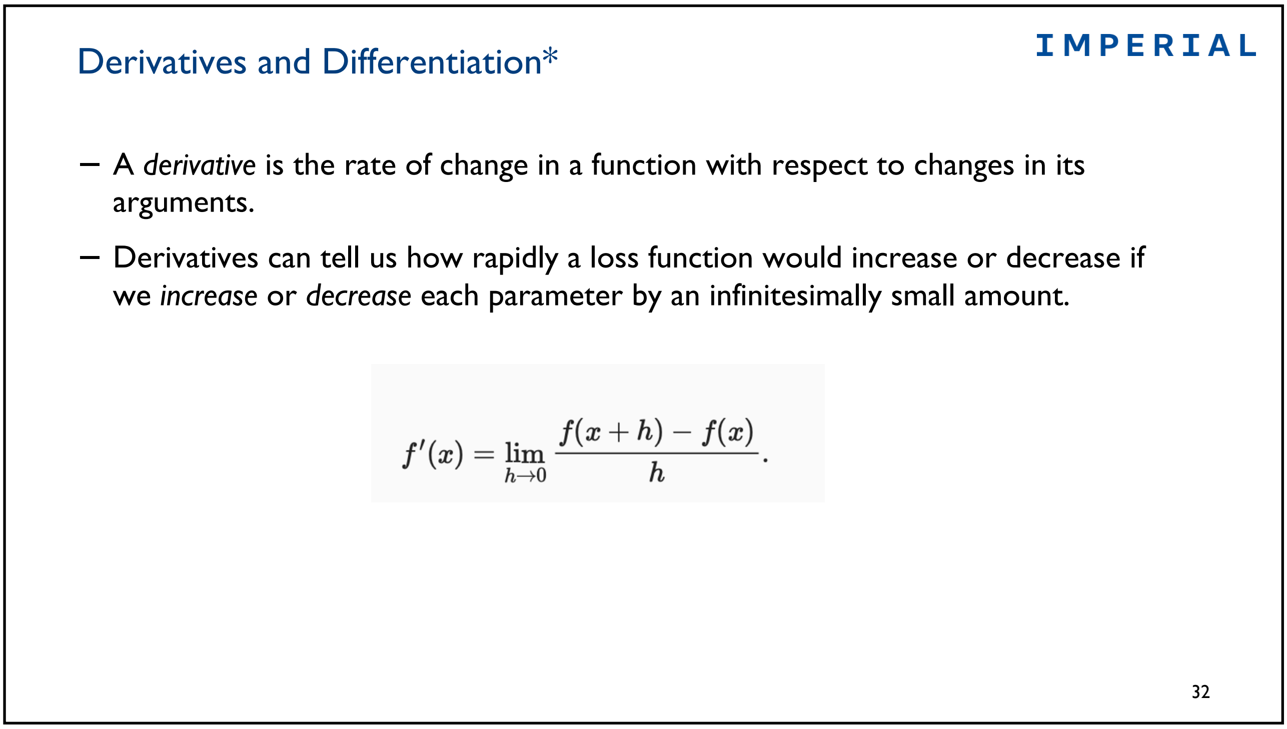
30



Gradient in neural network

- Or more precisely gradient of neural network parameters.
- In backpropagation, we need to calculate the partial derivates of variables with respect to some parameters.

31



Derivatives and Differentiation*

- A *derivative* is the rate of change in a function with respect to changes in its arguments.
- Derivatives can tell us how rapidly a loss function would increase or decrease if we *increase* or *decrease* each parameter by an infinitesimally small amount.

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

32

** Example: Calculate the derivative of f(x) = x^2

IMPERIAL

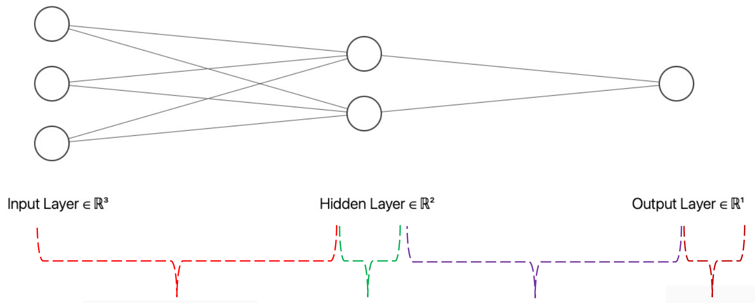
$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

33

33

Example - I

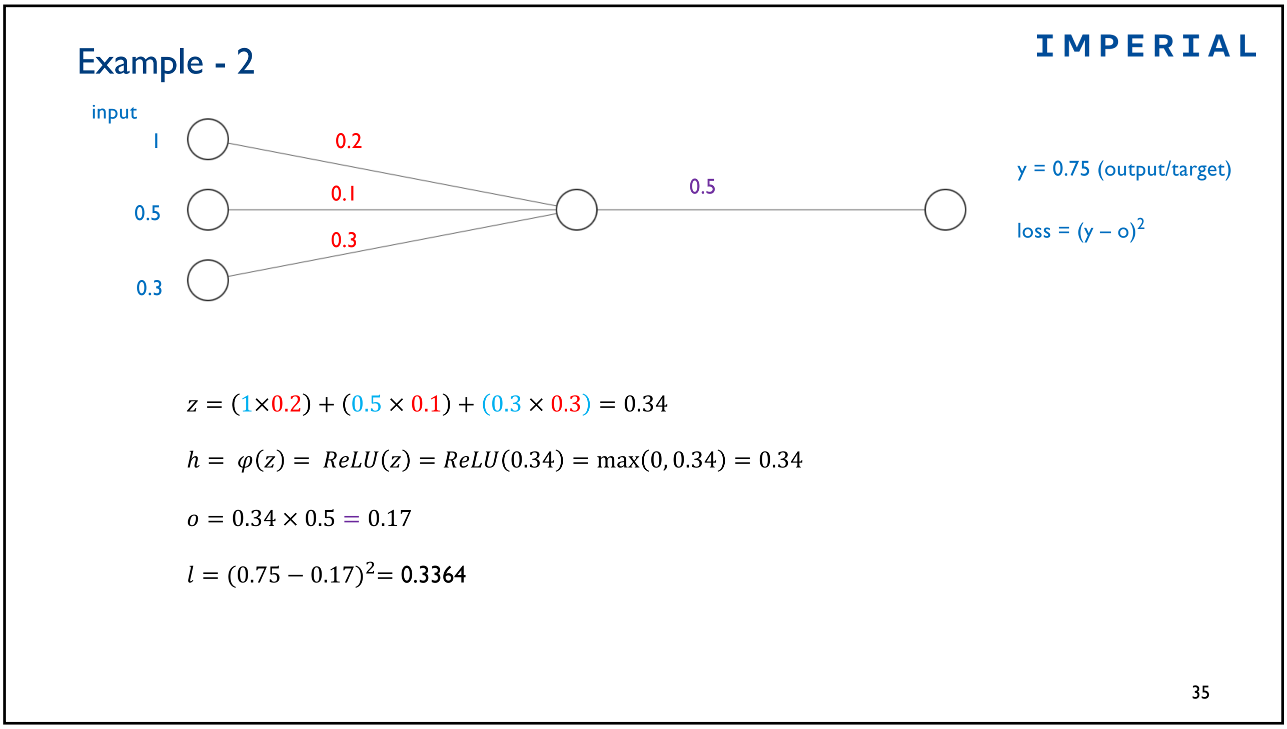
IMPERIAL



$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$ $\mathbf{h} = \phi(\mathbf{z}).$ $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$ $L = l(\mathbf{o}, y).$

34

34



Example - 2

y = 0.75 (output/target)

loss = $(y - o)^2$

$z = (1 \times 0.2) + (0.5 \times 0.1) + (0.3 \times 0.3) = 0.34$

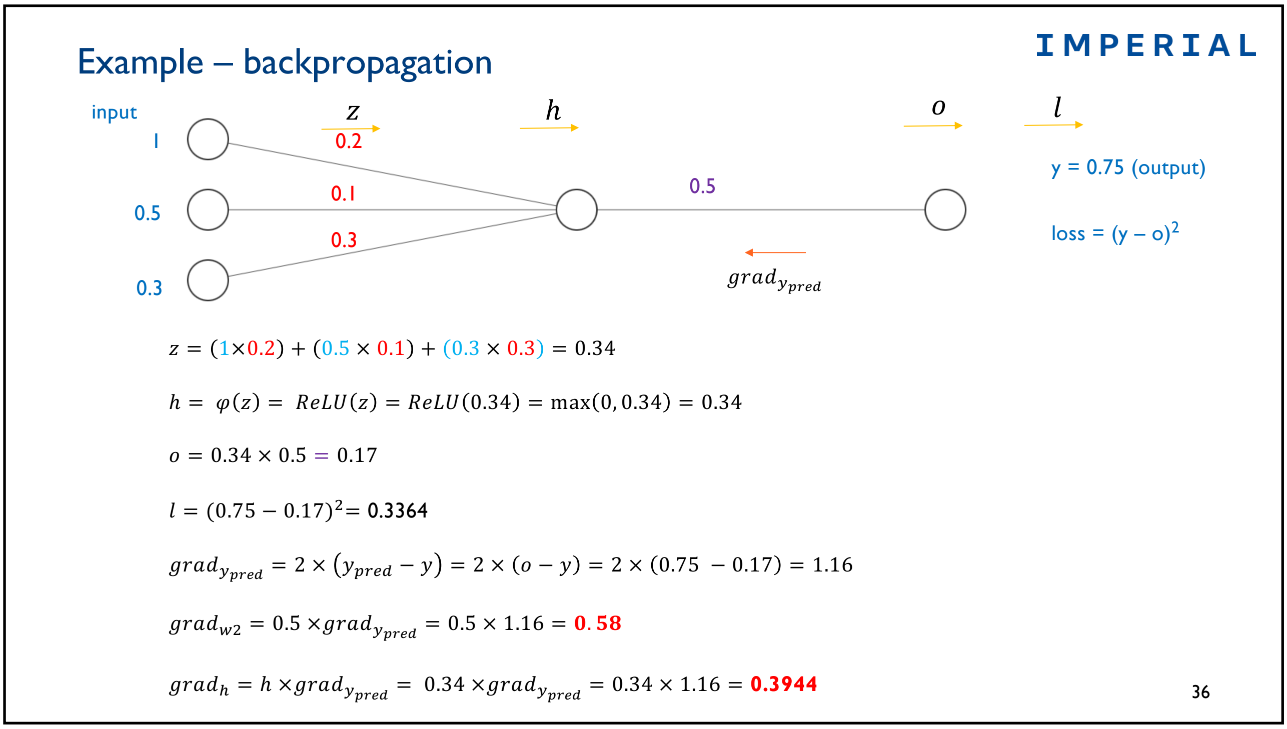
$h = \varphi(z) = ReLU(z) = ReLU(0.34) = \max(0, 0.34) = 0.34$

$o = 0.34 \times 0.5 = 0.17$

$l = (0.75 - 0.17)^2 = 0.3364$

35

35



Example – backpropagation

y = 0.75 (output)

loss = $(y - o)^2$

$z = (1 \times 0.2) + (0.5 \times 0.1) + (0.3 \times 0.3) = 0.34$

$h = \varphi(z) = ReLU(z) = ReLU(0.34) = \max(0, 0.34) = 0.34$

$o = 0.34 \times 0.5 = 0.17$

$l = (0.75 - 0.17)^2 = 0.3364$

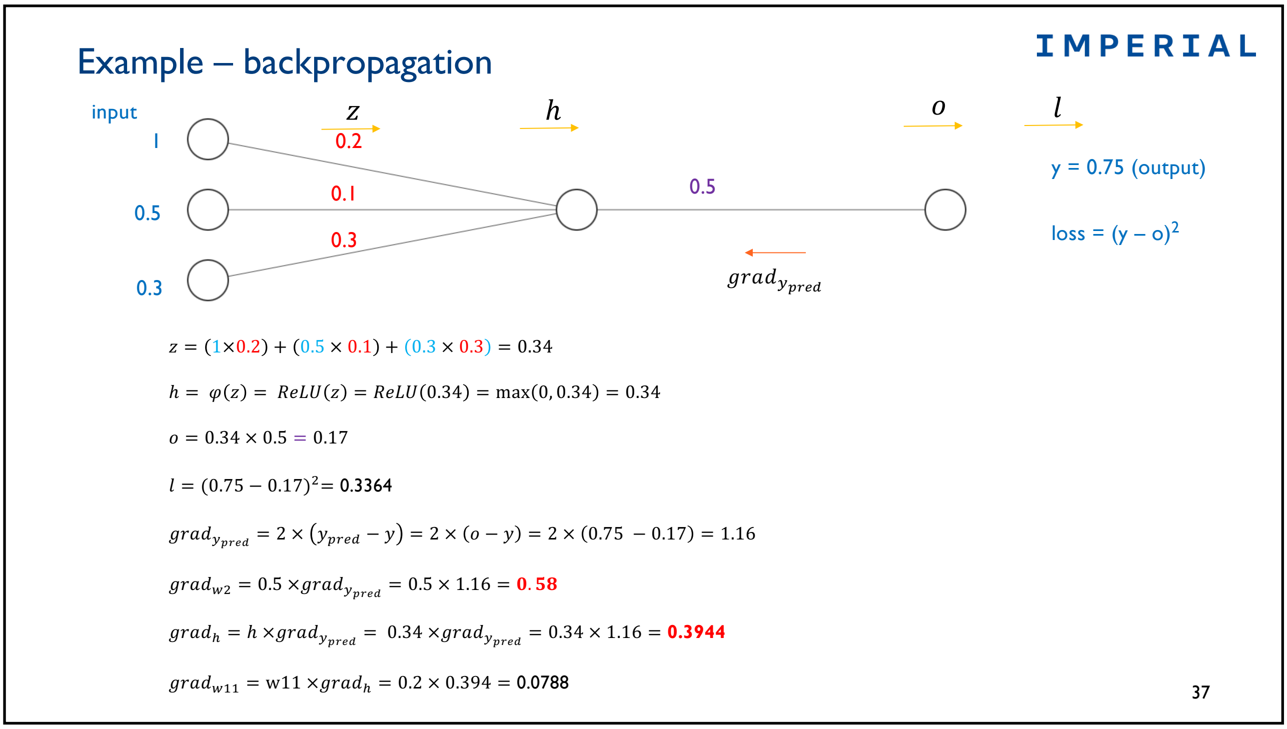
$grad_{y_{pred}} = 2 \times (y_{pred} - y) = 2 \times (o - y) = 2 \times (0.75 - 0.17) = 1.16$

$grad_{w2} = 0.5 \times grad_{y_{pred}} = 0.5 \times 1.16 = \mathbf{0.58}$

$grad_h = h \times grad_{y_{pred}} = 0.34 \times grad_{y_{pred}} = 0.34 \times 1.16 = \mathbf{0.3944}$

36

36



IMPERIAL

Example – backpropagation

y = 0.75 (output)

loss = $(y - o)^2$

$z = (1 \times 0.2) + (0.5 \times 0.1) + (0.3 \times 0.3) = 0.34$

$h = \varphi(z) = ReLU(z) = ReLU(0.34) = \max(0, 0.34) = 0.34$

$o = 0.34 \times 0.5 = 0.17$

$l = (0.75 - 0.17)^2 = 0.3364$

$grad_{y_{pred}} = 2 \times (y_{pred} - y) = 2 \times (o - y) = 2 \times (0.75 - 0.17) = 1.16$

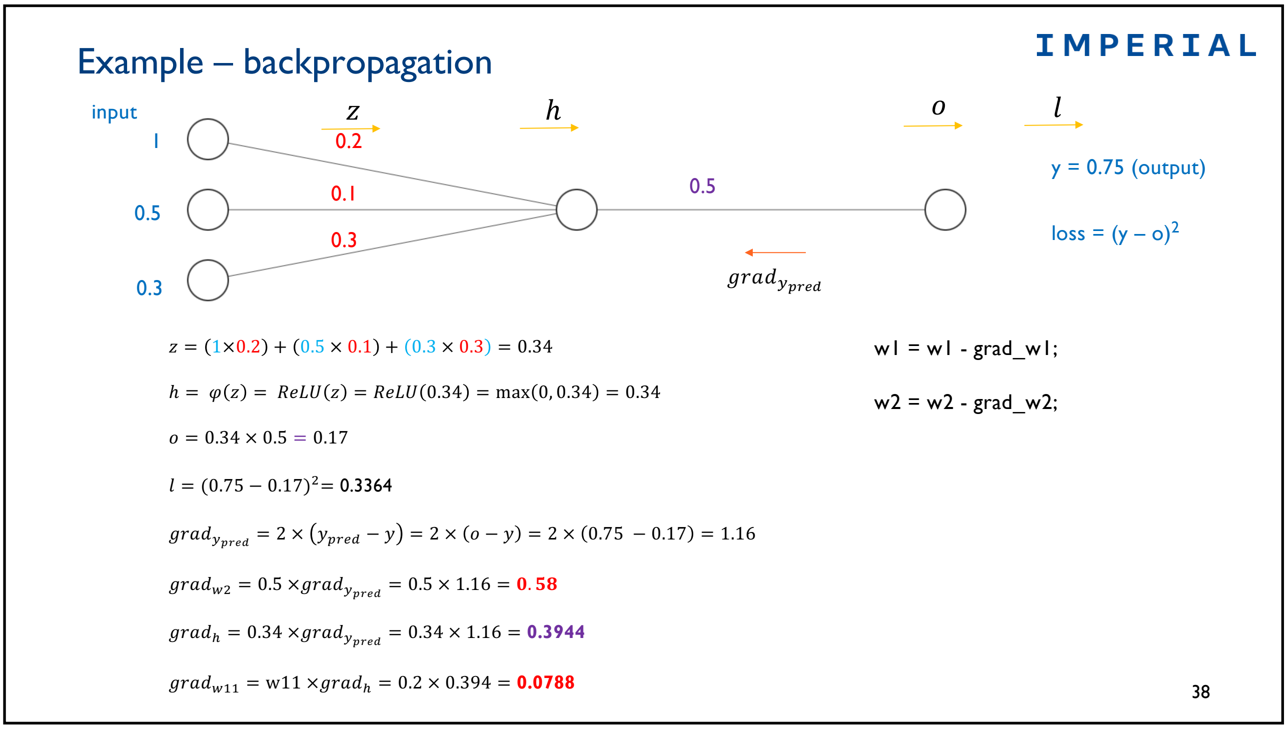
$grad_{w2} = 0.5 \times grad_{y_{pred}} = 0.5 \times 1.16 = \mathbf{0.58}$

$grad_h = h \times grad_{y_{pred}} = 0.34 \times grad_{y_{pred}} = 0.34 \times 1.16 = \mathbf{0.3944}$

$grad_{w11} = \text{w11} \times grad_h = 0.2 \times 0.394 = 0.0788$

37

37



IMPERIAL

Example – backpropagation

y = 0.75 (output)

loss = $(y - o)^2$

$z = (1 \times 0.2) + (0.5 \times 0.1) + (0.3 \times 0.3) = 0.34$

w1 = w1 - grad_w1;

$h = \varphi(z) = ReLU(z) = ReLU(0.34) = \max(0, 0.34) = 0.34$

w2 = w2 - grad_w2;

$o = 0.34 \times 0.5 = 0.17$

$l = (0.75 - 0.17)^2 = 0.3364$

$grad_{y_{pred}} = 2 \times (y_{pred} - y) = 2 \times (o - y) = 2 \times (0.75 - 0.17) = 1.16$

$grad_{w2} = 0.5 \times grad_{y_{pred}} = 0.5 \times 1.16 = \mathbf{0.58}$

$grad_h = 0.34 \times grad_{y_{pred}} = 0.34 \times 1.16 = \mathbf{0.3944}$

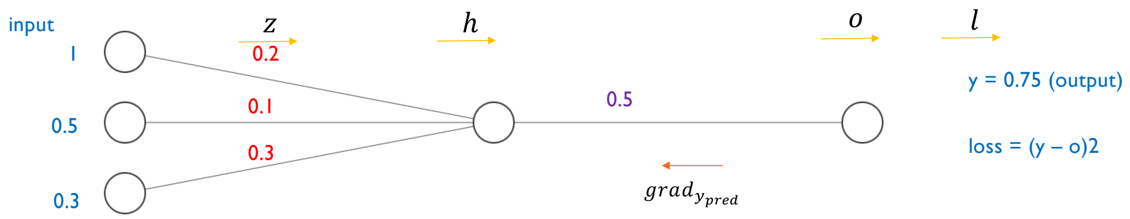
$grad_{w11} = \text{w11} \times grad_h = 0.2 \times 0.394 = \mathbf{0.0788}$

38

38

Example – backpropagation

IMPERIAL



$z = (1{\times}0.2) + (0.5 \times 0.1) + (0.3 \times 0.3) = 0.34$

$h = \varphi(z) = ReLU(z) = ReLU(0.34) = \max(0, 0.34) = 0.34$

$o = 0.34 \times 0.5 = 0.17$

$l = (0.75 - 0.17)^2 = 0.3364$

$grad_{y_{pred}} = 2 \times (y_{pred} - y) = 2 \times (o - y) = 2 \times (0.75\ - 0.17) = 1.16$

$grad_{w2} = 0.5 \times grad_{y_{pred}} = 0.5 \times 1.16 = 0.58$

$grad_h = 0.34 \times grad_{y_{pred}} = 0.34 \times 1.16 = 0.3944$

$grad_{w11} = \text{w11} \times grad_h = 0.2 \times 0.394 = 0.0788$

w_{11} = w_{11} – (learning_rate * grad_w_{11})
w_{12} = w_{12} – (learning_rate * grad_w_{12})
w_{13} = w_{13} – (learning_rate * grad_w_{13})

w_2 = w_2 – (learning_rate * grad_w2)

39

39

Learning rate

IMPERIAL

- Learning rate is a hyperparameter that controls the change of weights in neural networks with respect to the gradient.
- The learning rate scales the magnitude of change for the weights.
- For example:

w1 = w1 – (learning_rate * grad_w1)

w2 = w2 – (learning_rate * grad_w2)

40

40

How to choose a learning rate

IMPERIAL

- There are various ways to choose and optimise the learning rate.
- A simple approach is to start with a relatively large value (e.g. 0.01 or 0.1) and then (exponentially) reduce it over iterations.
- With dynamic learning rates, if the learning decreases too rapidly, it will affect the network convergence (i.e. finding an optimal solution).
- If (a large) learning rate changes too slowly, the network may fail to converge to a good enough solution since noise (noise in samples that causes noisy gradients) could keep on driving the network away from an optimal solution.

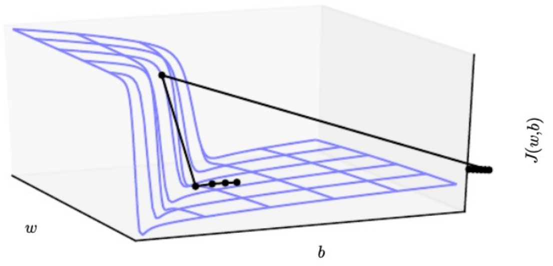
Exploding and vanishing gradients*

IMPERIAL

- The objective function for highly nonlinear deep neural networks or recurrent neural networks often contains sharp nonlinearities in parameter space resulting from the multiplication of several parameters.

- These nonlinearities give rise to very high derivatives in some places.

- When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly losing most of the optimisation work that has been done.



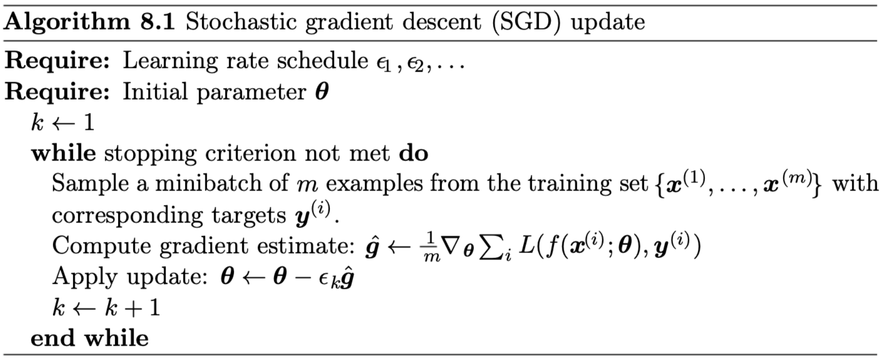
Source: Adapted from Goodfellow *et al.*, Deep Learning, MIT Press, https://www.deeplearningbook.org/contents/optimization.html
Original figure from (Pascanu *et al.*, 2013)

Momentum

- While stochastic gradient descent remains a popular optimisation strategy, learning with it can sometimes be slow. The momentum method (Polyak, 1964) was designed to accelerate learning.
- The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.
- Formally, the momentum algorithm introduces a variable $\mathcal{V}$ that plays the role of velocity - it is the direction and speed at which the parameters move through parameter space.

Goodfellow *et al.*, Deep Learning, MIT Press, https://www.deeplearningbook.org/contents/optimization.html

SGD



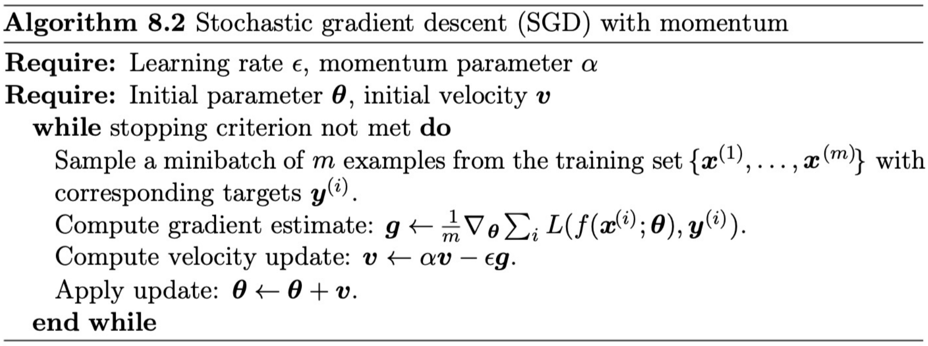
Algorithm 8.1 Stochastic gradient descent (SGD) update

```
Require: Learning rate schedule $\epsilon_1, \epsilon_2, \dots$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
    $k \leftarrow k + 1$
  end while
```

Source: Goodfellow *et al.*, Deep Learning, MIT Press, https://www.deeplearningbook.org/contents/optimization.html

SGD with momentum

IMPERIAL



```
Algorithm 8.2 Stochastic gradient descent (SGD) with momentum
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while
```

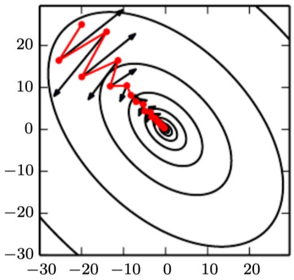
Source: Goodfellow *et al.*, deep Learning, https://www.deeplearningbook.org/contents/optimization.html

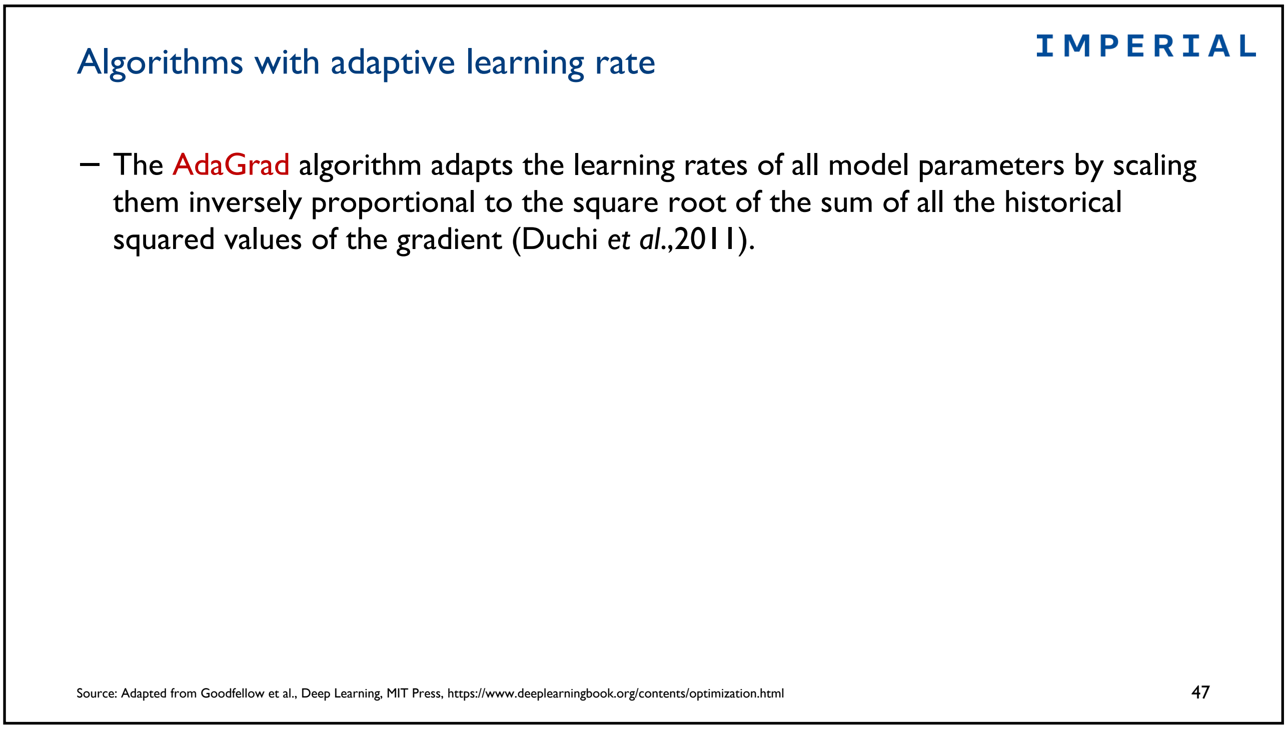
Momentum*

IMPERIAL

- If $\alpha = 0$, the momentum acts like a normal gradient descent:

An example with a non-zero α

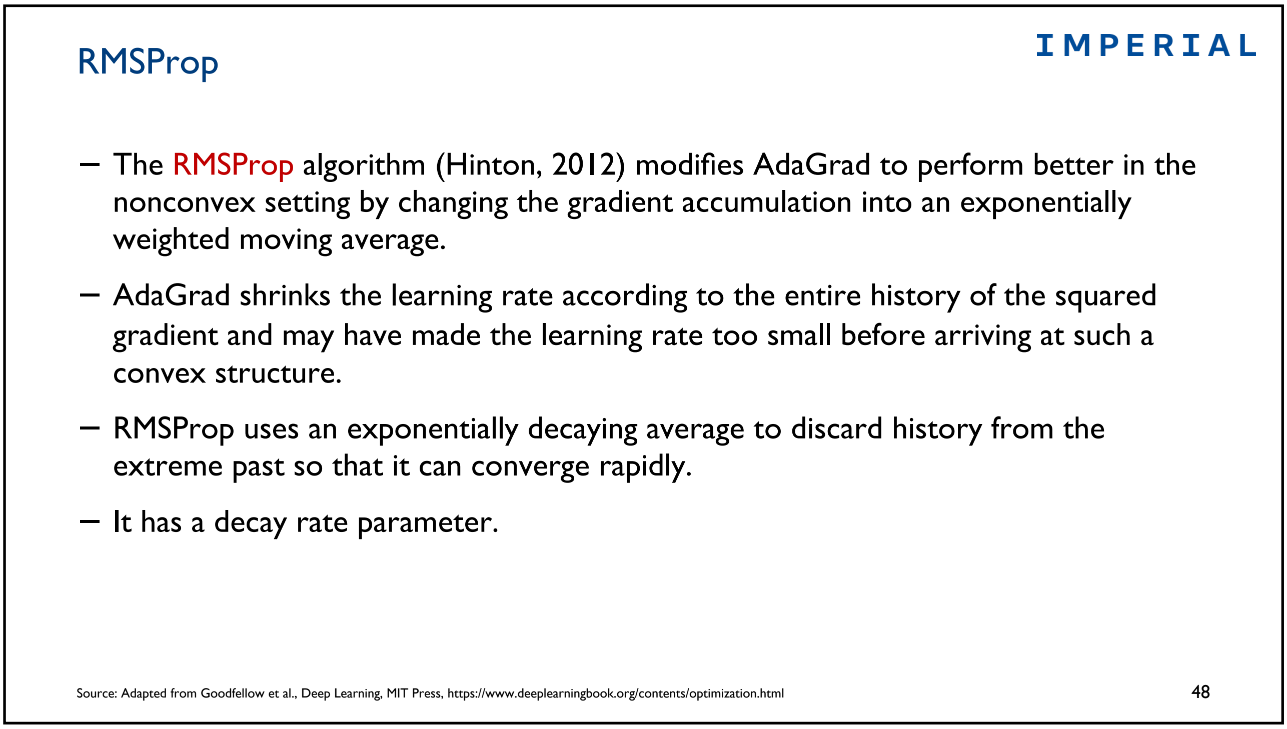




Algorithms with adaptive learning rate

- The AdaGrad algorithm adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient (Duchi *et al.*,2011).

Source: Adapted from Goodfellow et al., Deep Learning, MIT Press, https://www.deeplearningbook.org/contents/optimization.html



RMSProp

- The RMSProp algorithm (Hinton, 2012) modifies AdaGrad to perform better in the nonconvex setting by changing the gradient accumulation into an exponentially weighted moving average.
- AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure.
- RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly.
- It has a decay rate parameter.

Source: Adapted from Goodfellow et al., Deep Learning, MIT Press, https://www.deeplearningbook.org/contents/optimization.html

Adam

- Adam (Kingma and Ba, 2014) is another adaptive learning rate optimisation algorithm.
- The name "Adam" derives from the phrase "adaptive moments."
- In Adam, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient.
- The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients.
- Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialisation at the origin.

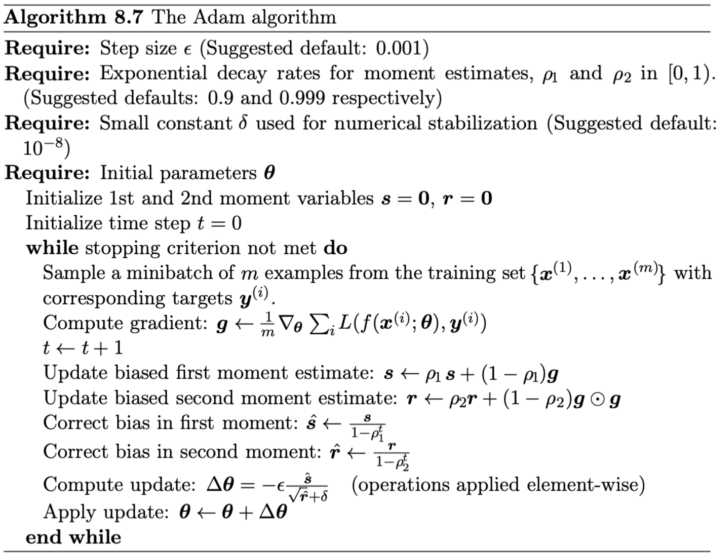
Source: Adapted from Goodfellow et al., Deep Learning, MIT Press, https://www.deeplearningbook.org/contents/optimization.html

Adam*

Adam uses exponential weighted moving averages (also known as leaky averaging) to obtain an estimate of both the momentum and the second moment of the gradient.

Adam**

IMPERIAL



Algorithm 8.7 The Adam algorithm

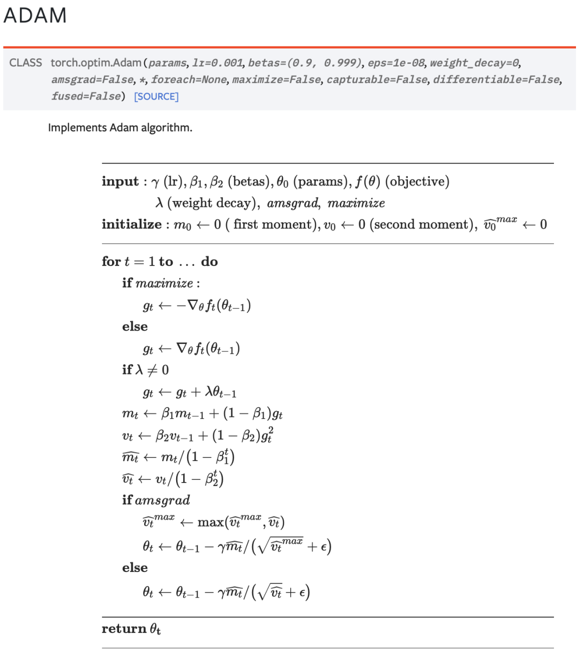
```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
  (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization (Suggested default:
  $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$   (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Source: Adapted from Goodfellow et al., Deep Learning, MIT Press, https://www.deeplearningbook.org/contents/optimization.html

Adam**

IMPERIAL

Same as the previous slide with just different notations to show the variables and parameters.



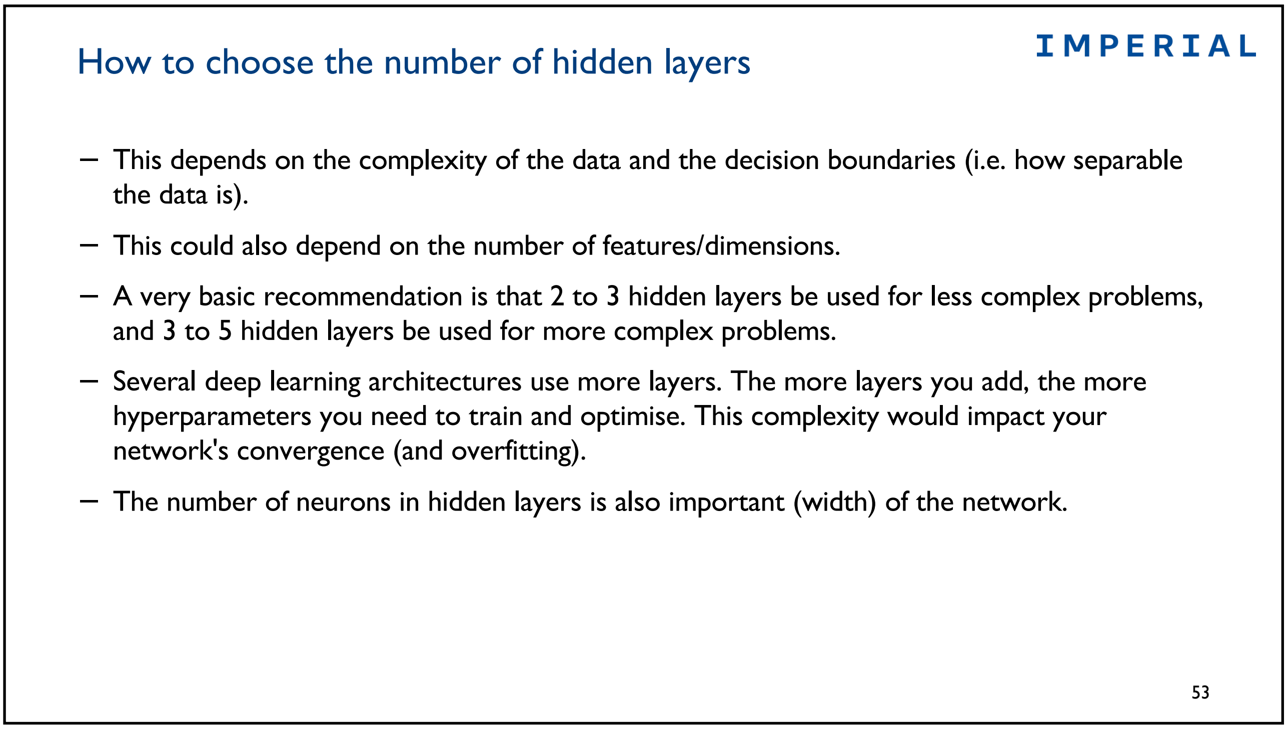
ADAM

CLASS torch.optim.Adam(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False, *, foreach=None, maximize=False, capturable=False, differentiable=False, fused=False*) [SOURCE]

Implements Adam algorithm.

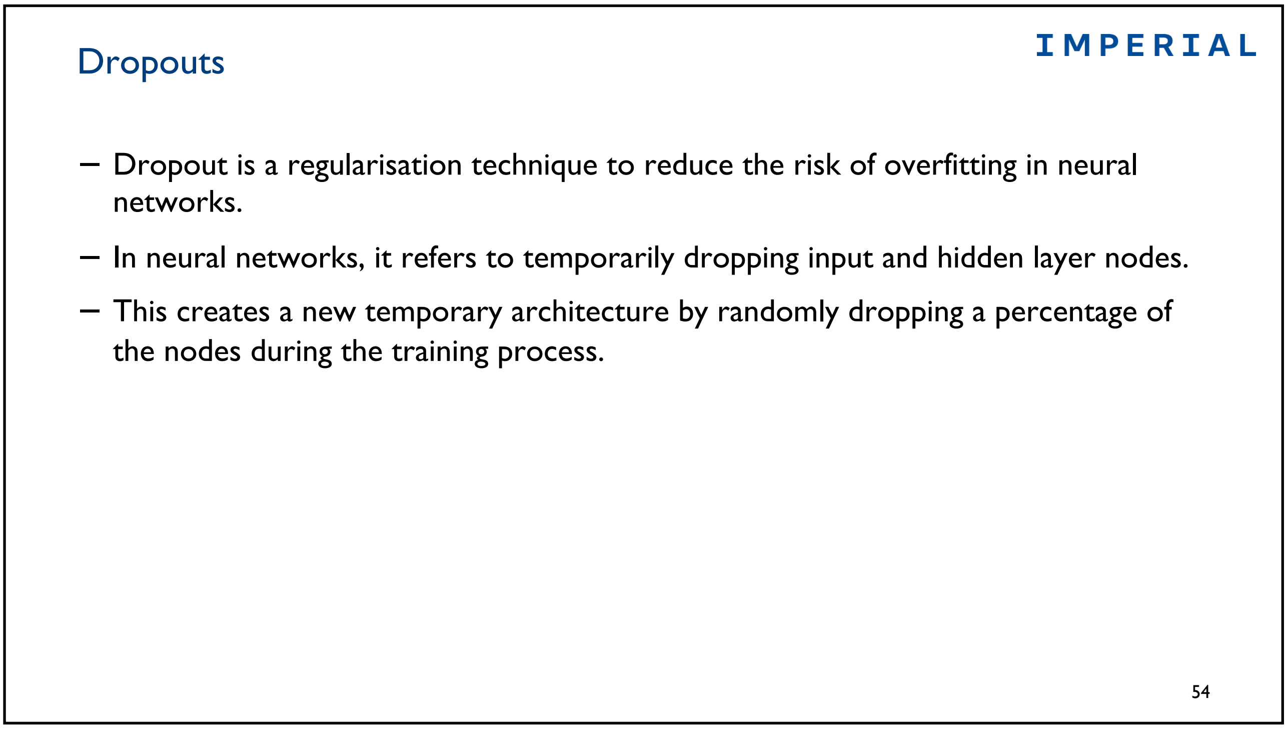
```
input : $\gamma$ (lr), $\beta_1, \beta_2$ (betas), $\theta_0$ (params), $f(\theta)$ (objective)
        $\lambda$ (weight decay), $amsgrad$, $maximize$
initialize : $m_0 \leftarrow 0$ ( first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ to $\ldots$ do
  if $maximize$ :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \big(1 - \beta_1^t\big)$
  $\widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big)$
  if $amsgrad$
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

return $\theta_t$
```

See: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html



How to choose the number of hidden layers

- This depends on the complexity of the data and the decision boundaries (i.e. how separable the data is).
- This could also depend on the number of features/dimensions.
- A very basic recommendation is that 2 to 3 hidden layers be used for less complex problems, and 3 to 5 hidden layers be used for more complex problems.
- Several deep learning architectures use more layers. The more layers you add, the more hyperparameters you need to train and optimise. This complexity would impact your network's convergence (and overfitting).
- The number of neurons in hidden layers is also important (width) of the network.



Dropouts

- Dropout is a regularisation technique to reduce the risk of overfitting in neural networks.
- In neural networks, it refers to temporarily dropping input and hidden layer nodes.
- This creates a new temporary architecture by randomly dropping a percentage of the nodes during the training process.

Dropout – philosophy

- A key idea behind applying dropout is that training a neural network by adding a stochastic behaviour and making predictions, and averaging over multiple stochastic decisions simulates a from of bagging with parameter sharing.

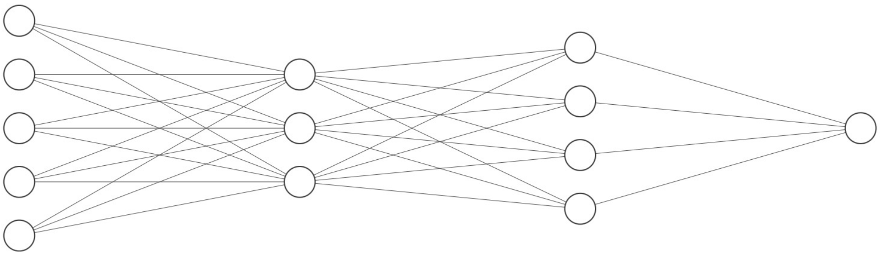
- In Bagging (i.e., bootstrap aggregation) several models are combined to reduce the generalisation error.
- The idea of Bagging is to train multiple models separately and then use all the models to vote on the output for the test example.

Dropouts – practical notes

- When the training sample is very small, the dropout method is not very effective.
- When additional unlabeled samples are available, unsupervised feature learning can improve the performance better than dropout.

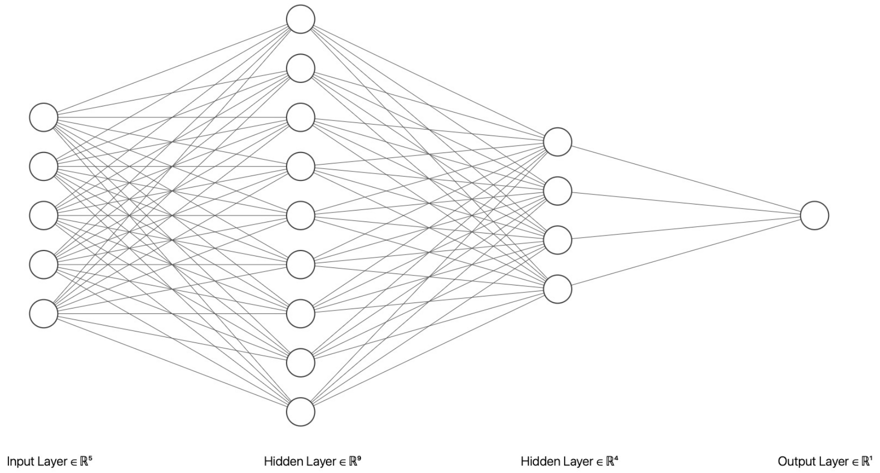
Example - 1

IMPERIAL



Example -2

IMPERIAL



IMPERIAL

Review questions

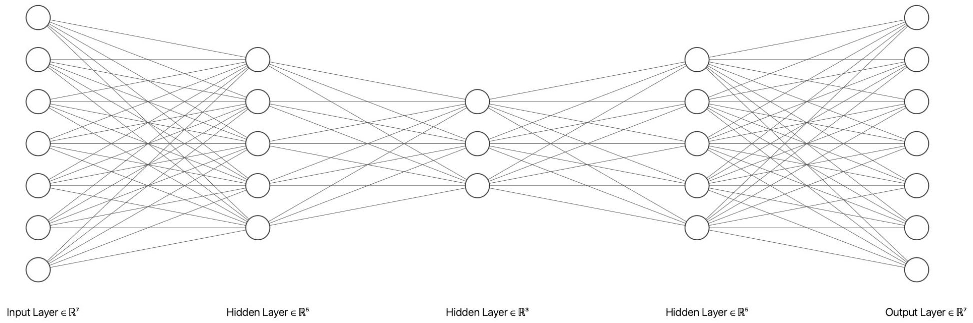
59

59

IMPERIAL

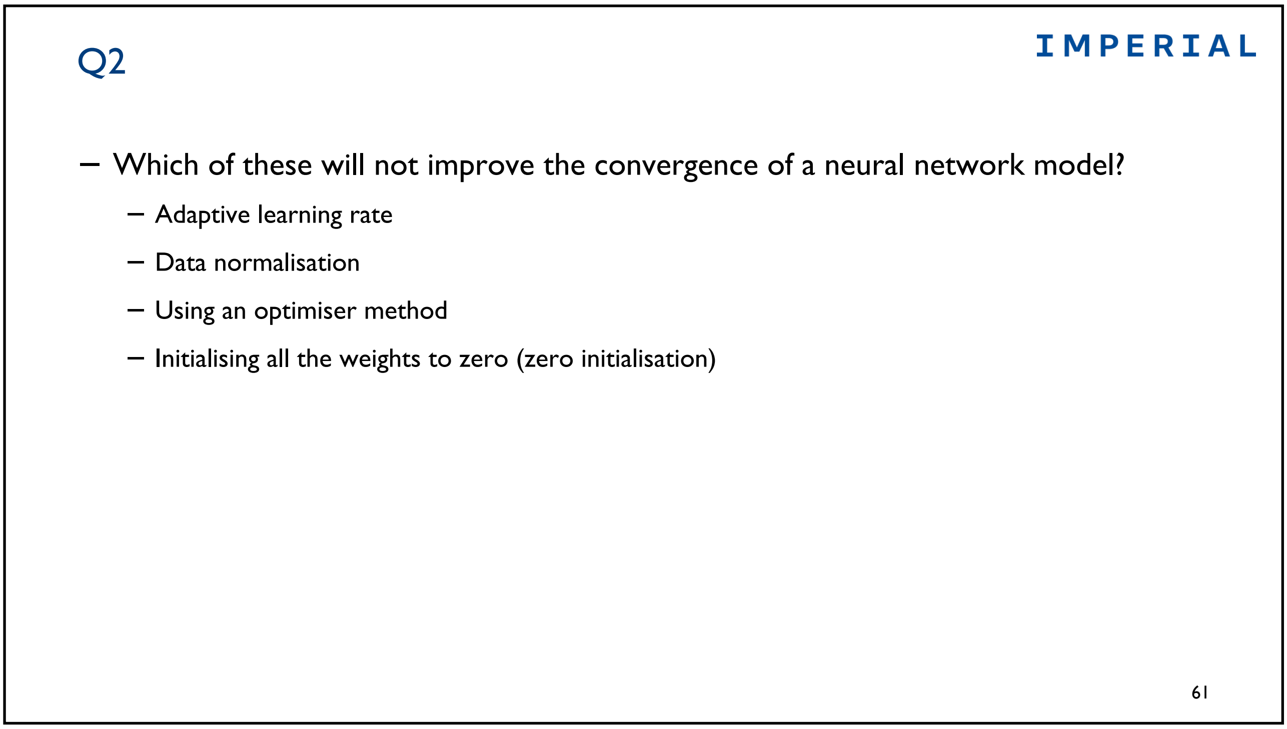
Q1

- What could be the purpose of a NN architecture like this?


Input Layer $\in \mathbb{R}^7$
Hidden Layer $\in \mathbb{R}^5$
Hidden Layer $\in \mathbb{R}^3$
Hidden Layer $\in \mathbb{R}^5$
Output Layer $\in \mathbb{R}^7$

60

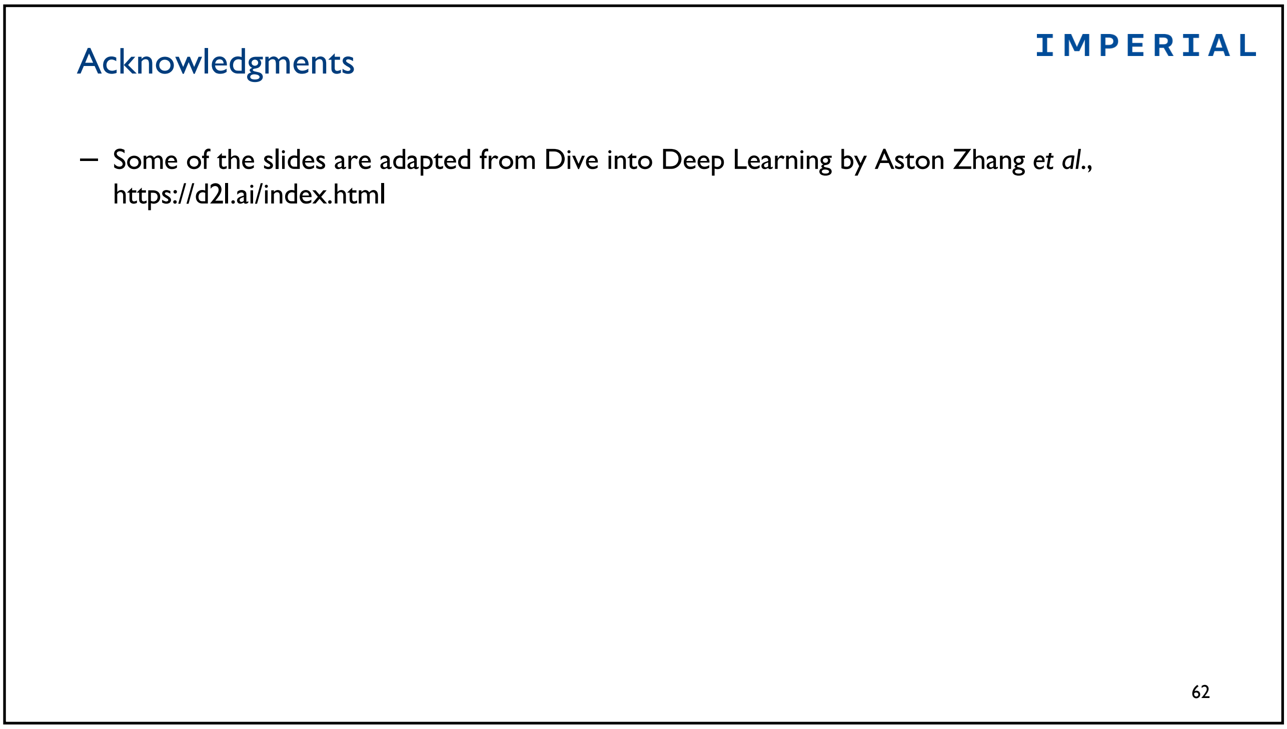
60



Q2

IMPERIAL

- Which of these will not improve the convergence of a neural network model?
  - Adaptive learning rate
  - Data normalisation
  - Using an optimiser method
  - Initialising all the weights to zero (zero initialisation)



Acknowledgments

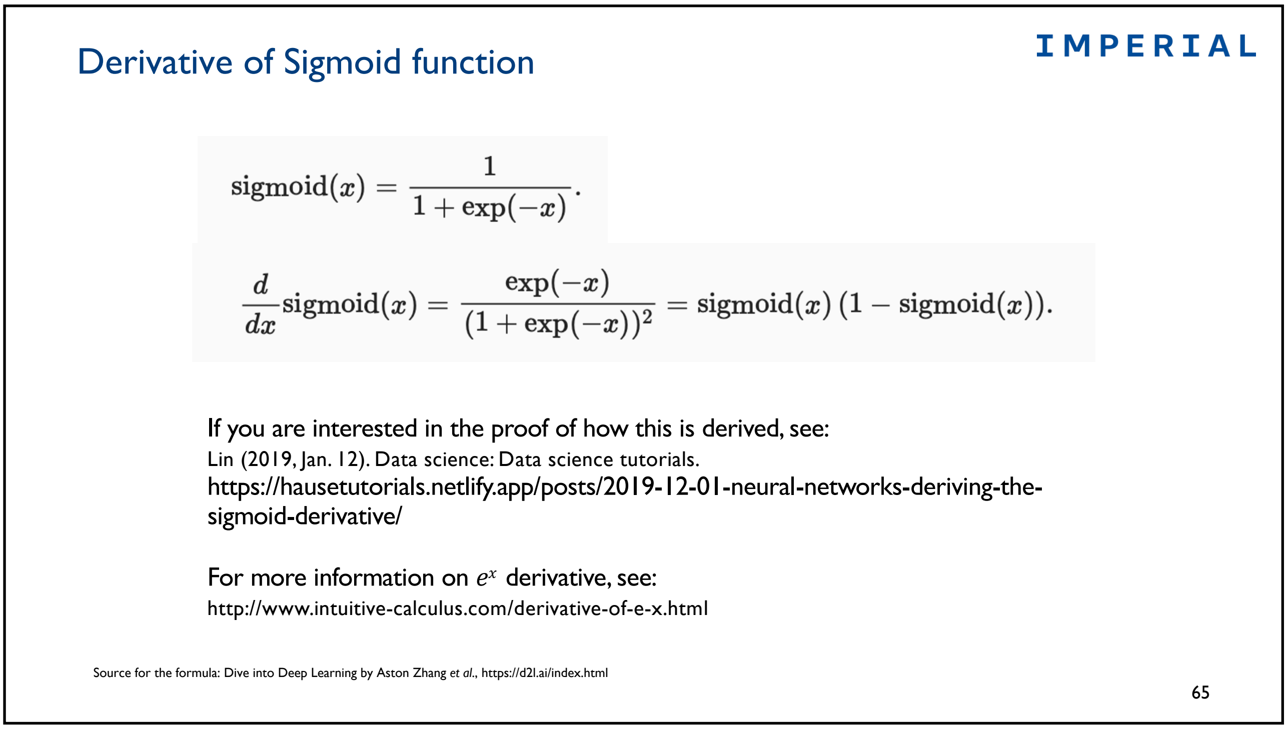
IMPERIAL

- Some of the slides are adapted from Dive into Deep Learning by Aston Zhang *et al.*, https://d2l.ai/index.html

If you have any questions

- Please feel free to arrange a meeting or email (p.barnaghi@imperial.ac.uk).
- My office: 928, Sir Michael Uren Research Hub, White City Campus.

Further reading



Derivative of Sigmoid function

IMPERIAL

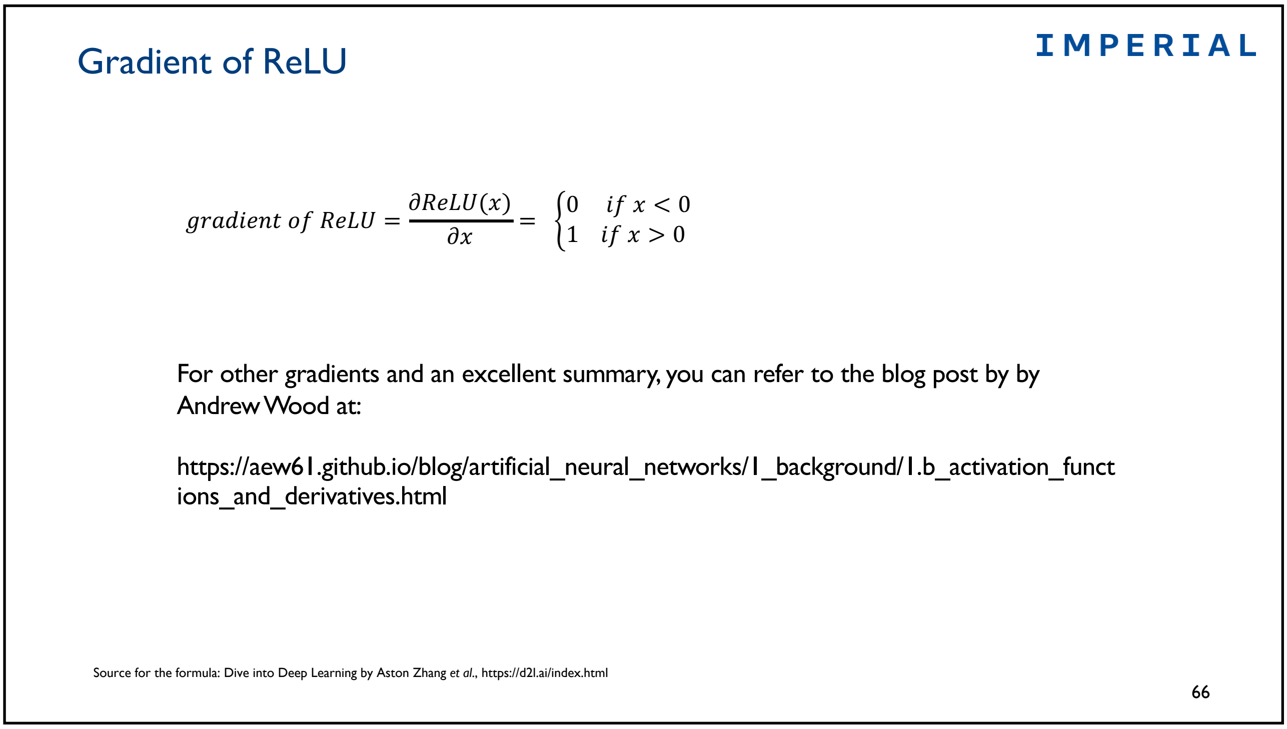
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)\left(1 - \text{sigmoid}(x)\right).$$

If you are interested in the proof of how this is derived, see:
Lin (2019, Jan. 12). Data science: Data science tutorials.
https://hausetutorials.netlify.app/posts/2019-12-01-neural-networks-deriving-the-sigmoid-derivative/

For more information on e^x derivative, see:
http://www.intuitive-calculus.com/derivative-of-e-x.html

Source for the formula: Dive into Deep Learning by Aston Zhang *et al.*, https://d2l.ai/index.html



Gradient of ReLU

IMPERIAL

$$gradient\ of\ ReLU = \frac{\partial ReLU(x)}{\partial x} = \begin{cases} 0 & if\ x < 0 \\ 1 & if\ x > 0 \end{cases}$$

For other gradients and an excellent summary, you can refer to the blog post by by Andrew Wood at:

https://aew61.github.io/blog/artificial_neural_networks/1_background/1.b_activation_functions_and_derivatives.html

Source for the formula: Dive into Deep Learning by Aston Zhang *et al.*, https://d2l.ai/index.html

Random initialisation**

IMPERIAL

– $Pytorch$ return the recommended gain value for the given nonlinearity function.

– According to the method described in *Understanding the difficulty of training deep feedforward neural networks*, Glorot, X. & Bengio, Y. (2010), using a uniform distribution. The resulting tensor will have values sampled from $U(-a, a)$ where (also known as Glorot initialisation):

$$a = \text{gain} \times \sqrt{\frac{6}{\text{fan\_in} + \text{fan\_out}}}$$

$fan\_in\ and\ fan\_out$: https://stackoverflow.com/questions/42670274/how-to-calculate-fan-in-and-fan-out-in-xavier-initialization-for-neural-networks

Source: https://pytorch.org/docs/stable/nn.init.html

67

67

Newton’s method

IMPERIAL

– Newton’s method is an optimisation scheme based on using a second-order Taylor series expansion to approximate $J(\theta)$ near some point θ_0 ignoring derivatives a of higher order:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

– Where H is the $Hessian$ of J with respect to θ evaluated at θ_0 If we then solve for the critical point of this function, we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Source: Adapted from Goodfellow et al., deep Learning, https://www.deeplearningbook.org/contents/optimization.html

For more information see Chapter 8 of Goodfellow et al.’s book.

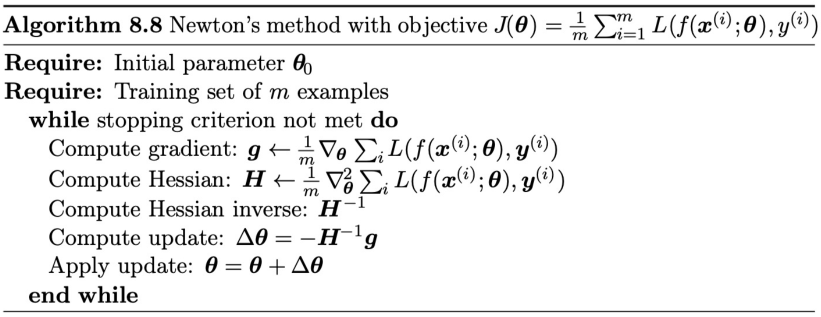
68

68

Newton's method – the algorithm

IMPERIAL

- If the objective function is convex but not quadratic (there are higher-order terms), this update can be iterated, yielding the training algorithm associated with Newton's method, given in algorithm



Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$

```
Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of $m$ examples
  while stopping criterion not met do
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian inverse: $\boldsymbol{H}^{-1}$
    Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Source: Adapted from Goodfellow *et al.*, Deep Learning, MIT Press, https://www.deeplearningbook.org/contents/optimization.html

The *Hessian* Matrix (H)

IMPERIAL

- For more information on the Hessian matrix and how to derive it, see:

https://www.khanacademy.org/math/multivariable-calculus/applications-of-multivariable-derivatives/quadratic-approximations/a/the-hessian