

Using Machine Learning Tools PG

Week 10 – Training Deep Neural
Networks

COMP SCI 7317

Trimester 2, 2024



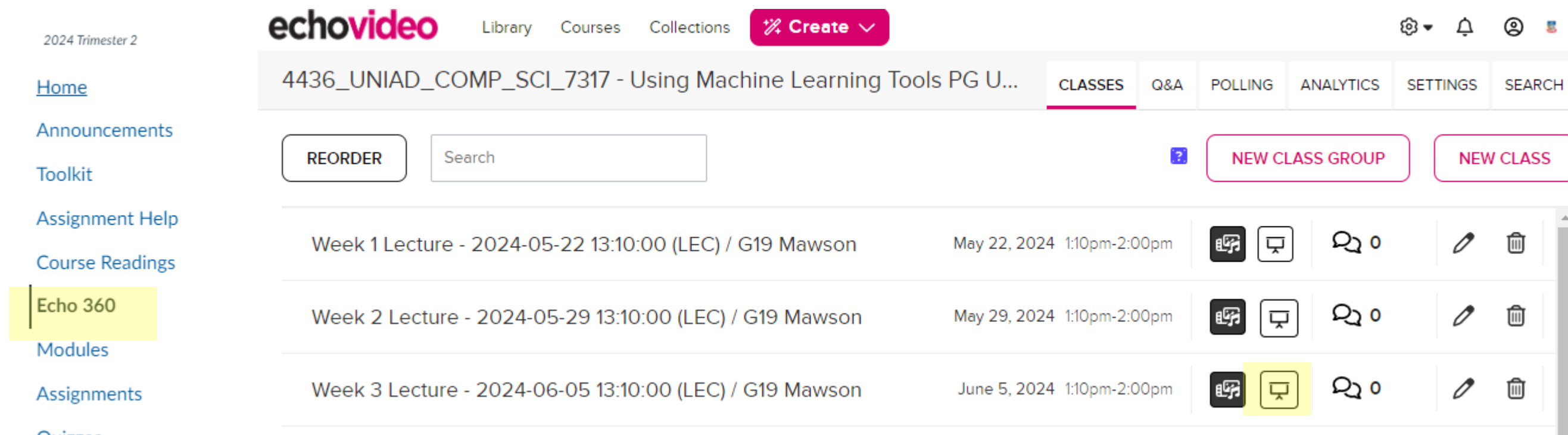
THE UNIVERSITY
of ADELAIDE

150 YEARS

Before we start...

There will be some **interactive elements** throughout this lecture.

Please participate by heading to **MyUni** > **4436_COMP_SCI_7317** > **Echo360** > **Lecture Week 10** > 



The screenshot displays the EchoVideo interface for the course 4436_UNIAD_COMP_SCI_7317 - Using Machine Learning Tools PG U... The interface includes a sidebar with navigation links: Home, Announcements, Toolkit, Assignment Help, Course Readings, Echo 360 (highlighted), Modules, Assignments, and Quizzes. The main content area shows a list of lectures with the following details:

Lecture Title	Date and Time	Interactive Icons
Week 1 Lecture - 2024-05-22 13:10:00 (LEC) / G19 Mawson	May 22, 2024 1:10pm-2:00pm	Video, Echo360, Chat (0), Edit, Delete
Week 2 Lecture - 2024-05-29 13:10:00 (LEC) / G19 Mawson	May 29, 2024 1:10pm-2:00pm	Video, Echo360, Chat (0), Edit, Delete
Week 3 Lecture - 2024-06-05 13:10:00 (LEC) / G19 Mawson	June 5, 2024 1:10pm-2:00pm	Video, Echo360 (highlighted), Chat (0), Edit, Delete

Last week

1. Convolutional Neural Networks
2. Convolution + CNN Structures
3. Pooling + Parameters
4. Applications + Considerations



THE UNIVERSITY
of ADELAIDE

150 YEARS

This week...

1. The process of training neural networks
2. Vanishing and exploding gradients
 - Four ways of dealing with this
3. Optimisers
 - Range of options available
4. Learning rate scheduling and 1cycle method



THE UNIVERSITY
of ADELAIDE

150 YEARS



The process of training

Training algorithm

Training: minimise cost function by adjusting model parameters

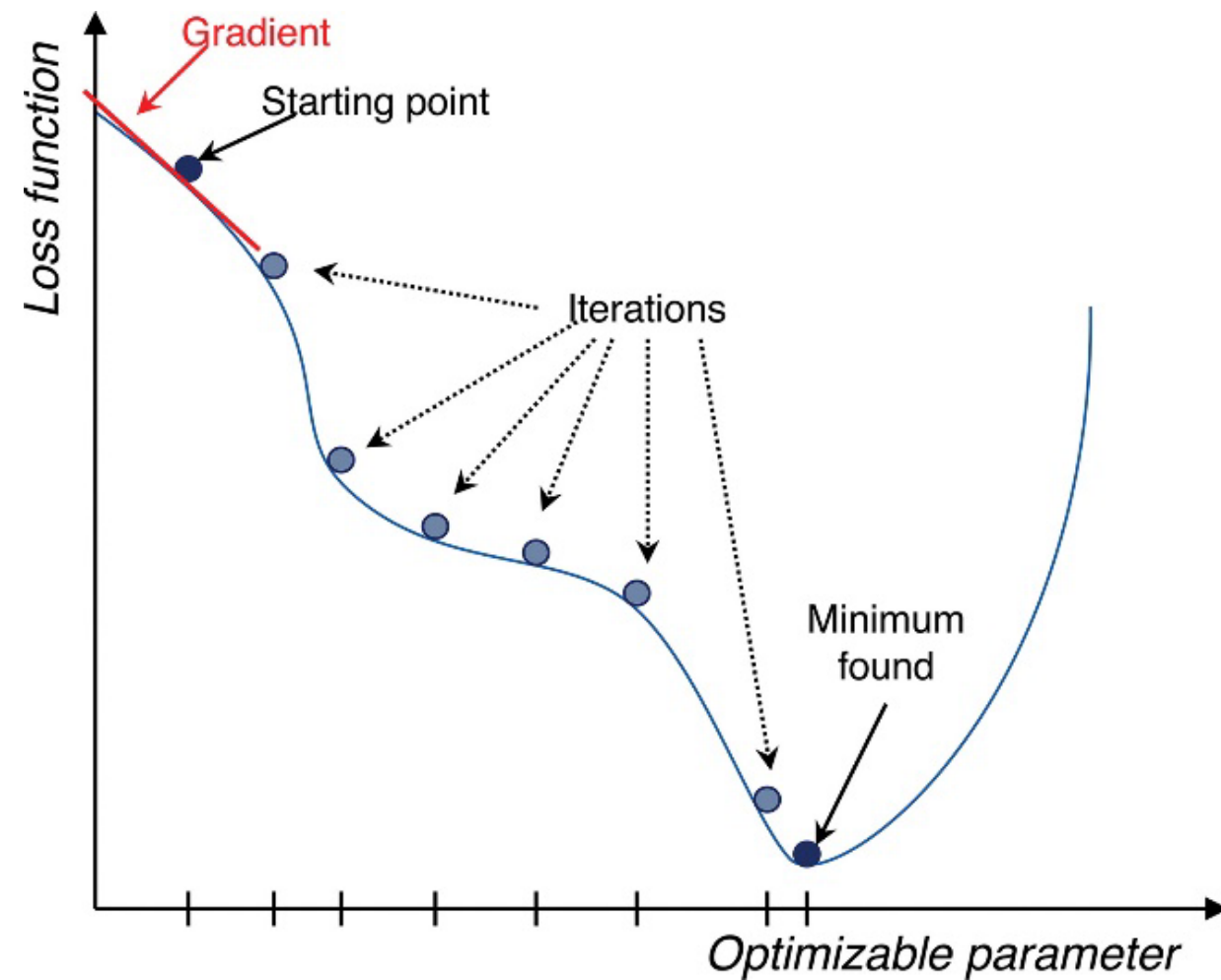
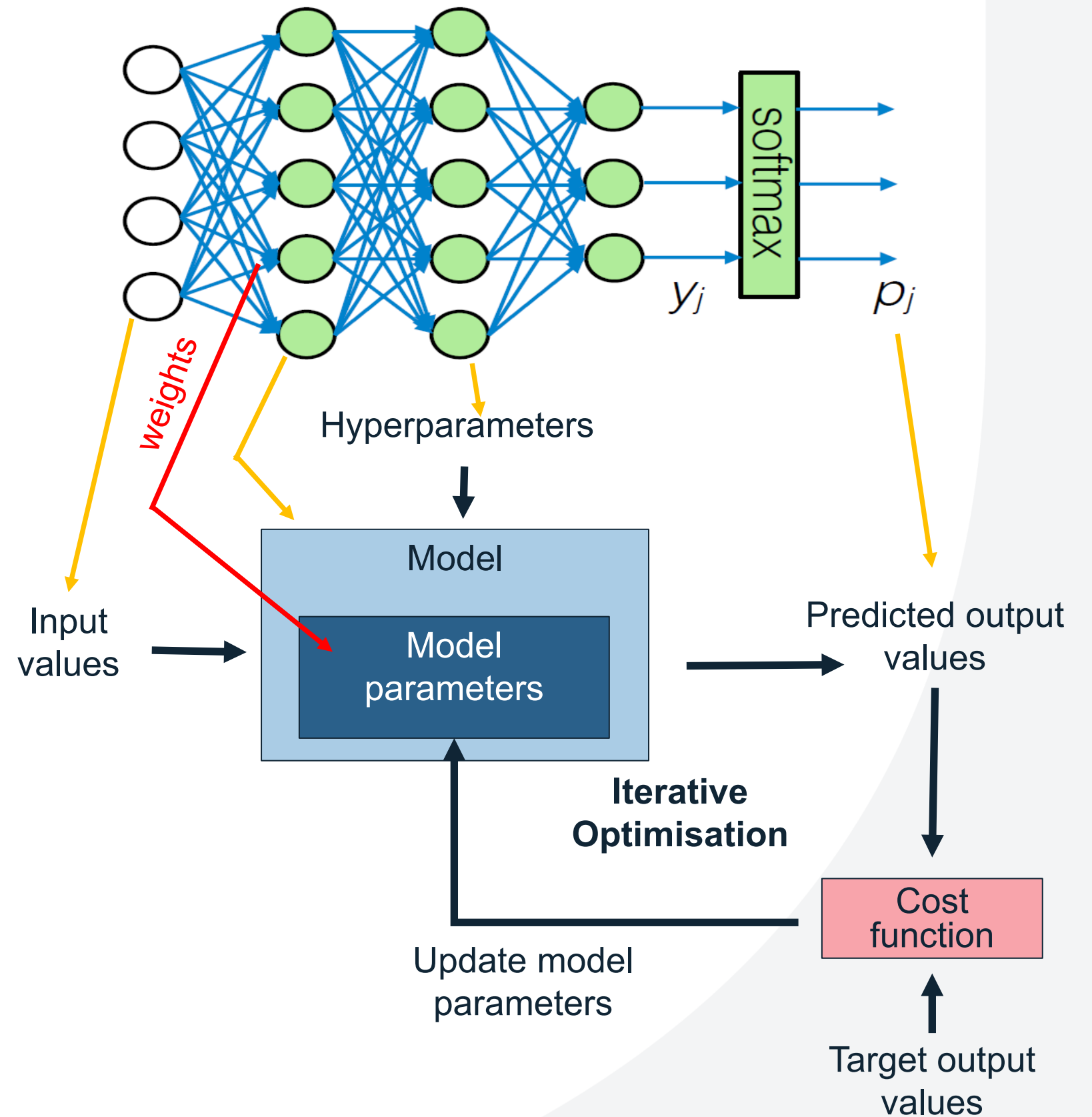
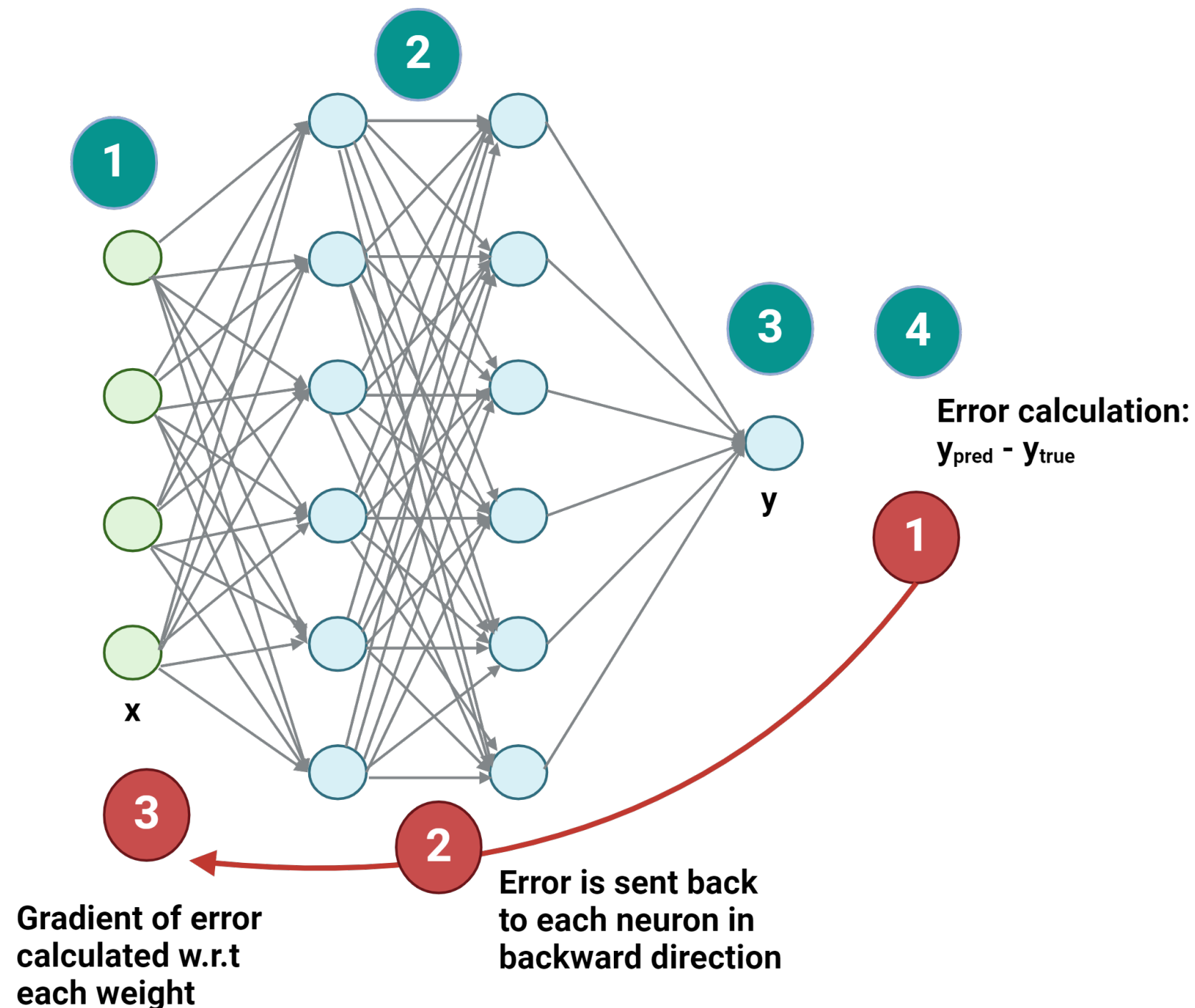


Image: Chartrand et al. RadioGraphics 2017



Training Algorithm: Backpropagation

Backpropagation is used to compute the gradient of the loss function with respect to each weight by the chain rule, essential for gradient descent and to adjust weights to minimise error in predictions.



Forward pass: Compute the output of the network by feeding data into input nodes and passing along layer by layer until output node reached.

Error calculation: Calculate the error (difference between predicted output and actual output).

Backward Pass: Compute the gradient of the loss function with respect to each weight. Update the weights using gradient descent.

Training Algorithm

- **Initialise weights randomly (break symmetry)**
 - If all weights set to the same value, they would do same thing and may just learn the same features from the data. Breaking symmetry = easier to find unique weights/gradients.
- **Forward pass:**
 - Data fed forward and values at each node (or neuron) calculated layer by layer until we get an output.
- **Compute loss function:**
 - Measures how far off our model's output is from the actual target value.
- **Backward pass (Backpropagation):**
 - Work backwards from the output to the input, adjusting the weights to reduce the error. Using chain rule, we can compute contribution of each weight to error.
- **Gradient descent step on weights**
 - Use the gradients to adjust the weights, reducing the error.
- **Repeat for batches of data (mini-batch)**
- **Repeat for multiple epochs**

Training Algorithm: Backpropagation

Aim: Compute the gradient of the loss function with respect to each weight in the network.

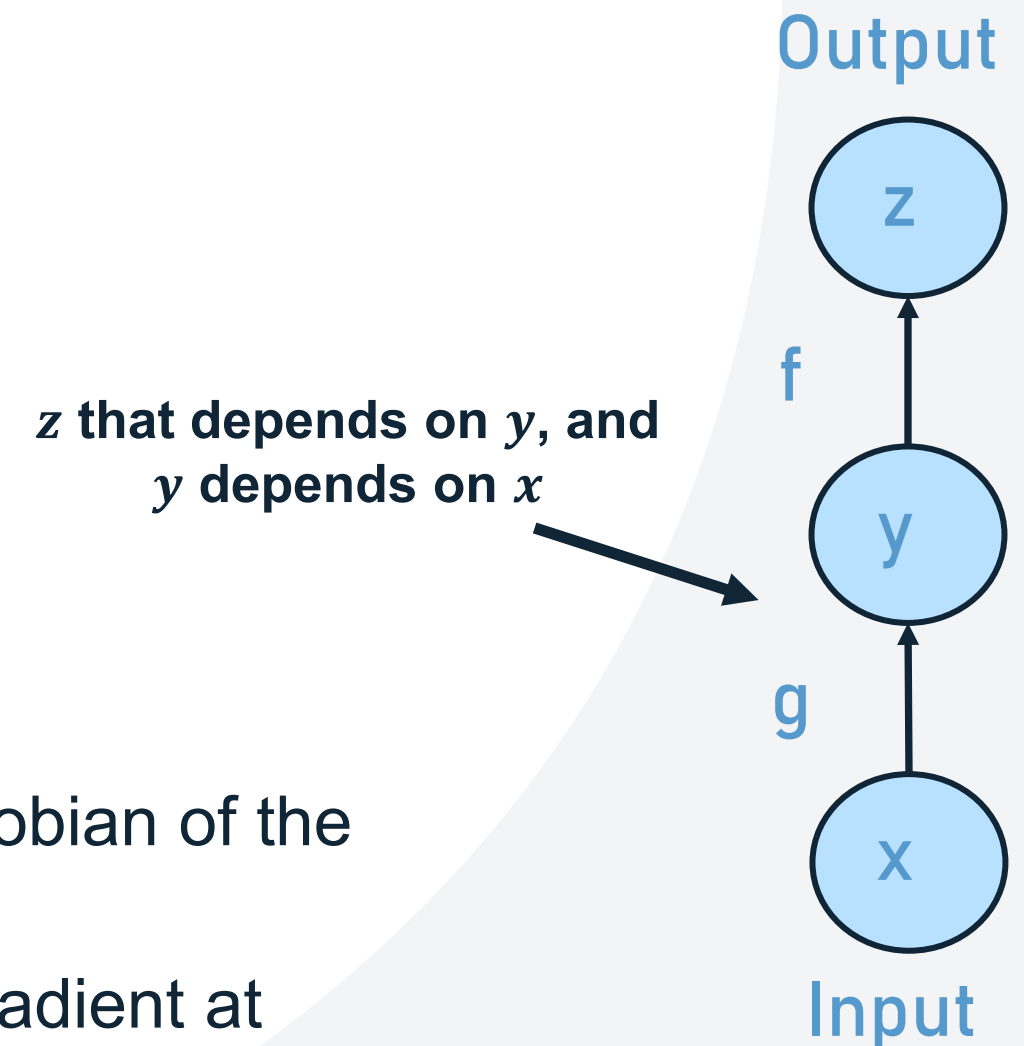
- The chain rule in calculus helps us break down the derivative of a complex function into a series of simpler derivatives:

$$z = f(y) = f(g(x))$$
$$z' = f'(y) * g'(x)$$

Partial derivative (gradient) of z w.r.t $x \longrightarrow \partial z / \partial x = \partial z / \partial y * \partial y / \partial x$

- Travel backwards from outputs to inputs:**

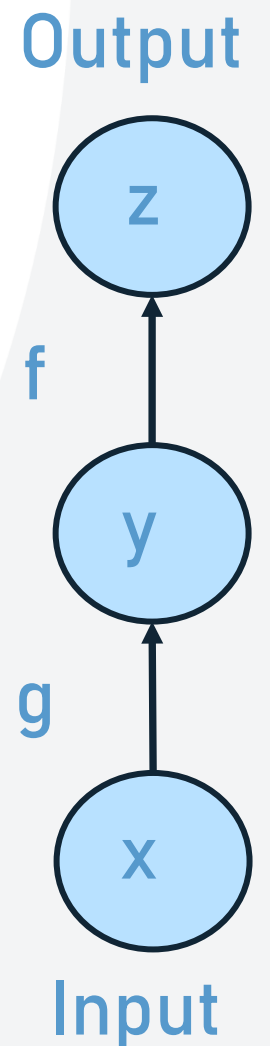
- Start at output layer and move backwards to input layer
- Multiply with local derivatives (Jacobian) each time
 - At each layer, adjust gradients by multiplying with Jacobian of the activation functions and transformation
- Sum up gradients where multiple paths meet to get total gradient at each weight



Training Algorithm: Backpropagation

Aim: Compute the gradient of the loss function with respect to each weight in the network.

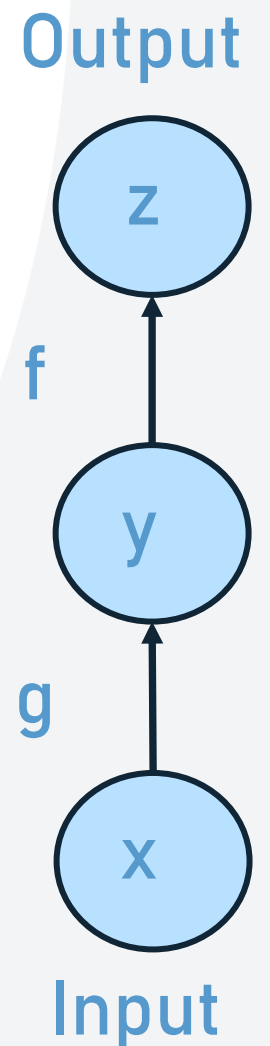
- **Sum up gradients where multiple paths meet to get total gradient at each weight:**
 - Single weight in network can affect the final output through different paths.
 - i.e., weight in the first layer \rightarrow influence multiple neurons in the next layer \rightarrow influence final output
 - During backpropagation, gradient is calculated (i.e., how much the error changes with a small change in the weight) for each of these paths.
 - Each path contributes a partial gradient to the total gradient for that weight



Training Algorithm: Backpropagation

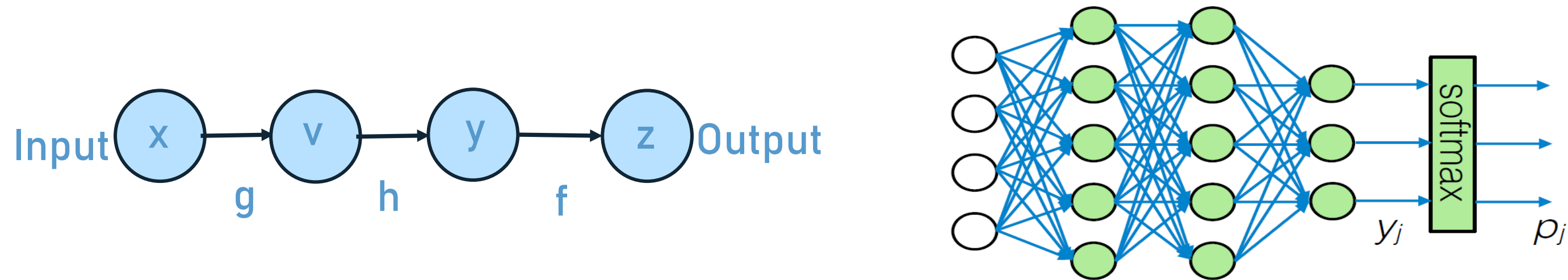
Aim: Compute the gradient of the loss function with respect to each weight in the network.

- **Every operation (or node) in network must be differentiable**
 - Differentiable: We need to be able to calculate the derivative (or gradient) of each function (i.e. f and g) used in the network. This is essential for the backpropagation process.
 - Why it matters: During training, we use these derivatives to understand how changes in weights affect the error, allowing us to update the weights correctly.
 - Backpropagation rule: Each node (or operation) has its own specific way (rule) to compute its derivative during backpropagation.
 - Primarily based on the specific activation function applied to each node, as well as any other operations that the node performs (such as linear transformations).



Training Algorithm: Backpropagation

Aim: Compute the gradient of the loss function with respect to each weight in the network.



- A neural network is composed of multiple layers, each with many nodes (or neurons).
 - We have multiple functions (i.e activation functions), like f and g , at each node.
 - For NN with 3 layers:

Layer 1 (input): Each node in the input layer applies functions g_1, g_2, \dots, g_n to input x_i

Layer 2: Each node in layer 2 applies functions h_1, h_2, \dots, h_n to outputs (v_i) of layer 1

Layer 3: Each node in layer 3 applies functions f_1, f_2, \dots, f_n to y_i to produce output z

Which answer is correct?

Which of the following is NOT a part of the neural network training process?

- Initialising weights
- Forward pass
- Backward pass
- Data normalisation
- Computing the loss



THE UNIVERSITY
of ADELAIDE

150 YEARS

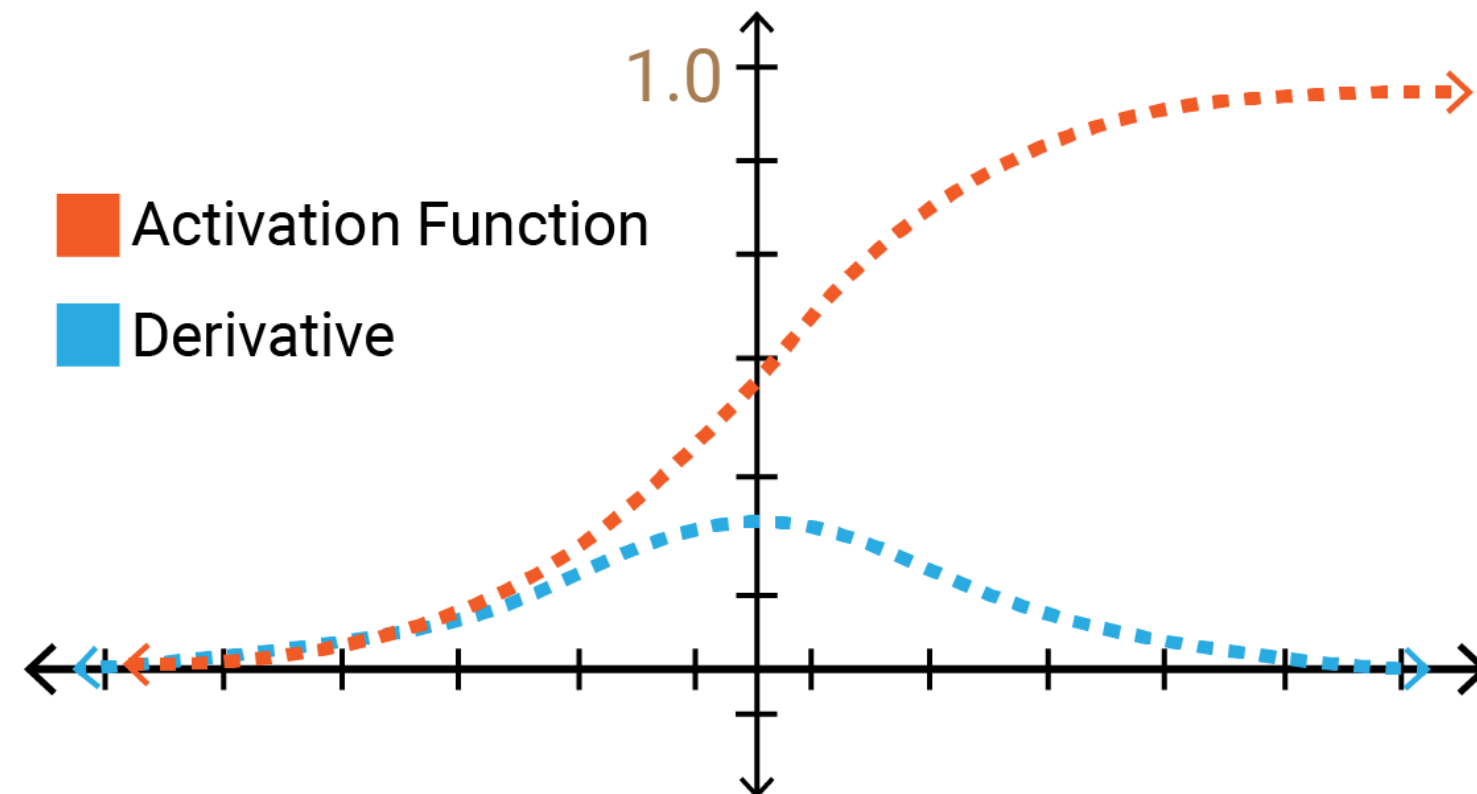


Vanishing & Exploding Gradients

Vanishing Gradients

Vanishing Gradient Problem: As the gradient moves backwards, it gets smaller, causing the following issues:

- **Gradient Shrinks:** The gradient diminishes, especially in the lower layers.
- **Minimal Weight Updates:** Weights in the lower layers change very little during each gradient step. This makes it difficult for these layers to learn effectively.
- **Training Stalls:** Training becomes very slow or never converges due to minimal updates.



Vanishing Gradients

Why does this happen?

- For most activation functions, there is a saturation region where the input values have little or almost no influence on the output.
- If a node has a value in one of these saturation zones, it will not contribute much information when we calculate the gradient.
- Additionally, when many such small contributions from multiple terms are multiplied together, the overall gradient can become very small.

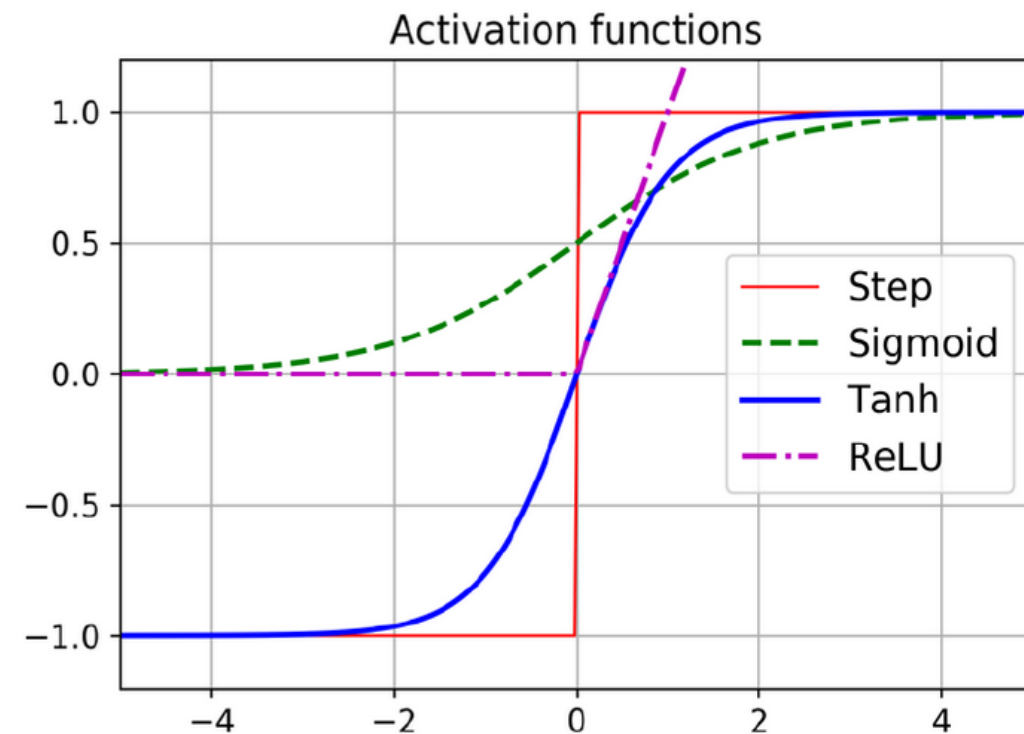


Image: Géron, Hands On ML

- **Sigmoid and Tanh:** Prone to vanishing gradients due to small derivatives.
 - Due to compression of input values into a narrow output range (i.e. 0 - 1 or -1 - 1).
- **ReLU:** Mitigates the problem by maintaining a constant gradient for positive inputs.
 - Prevents the gradient from shrinking as much.

Exploding Gradients

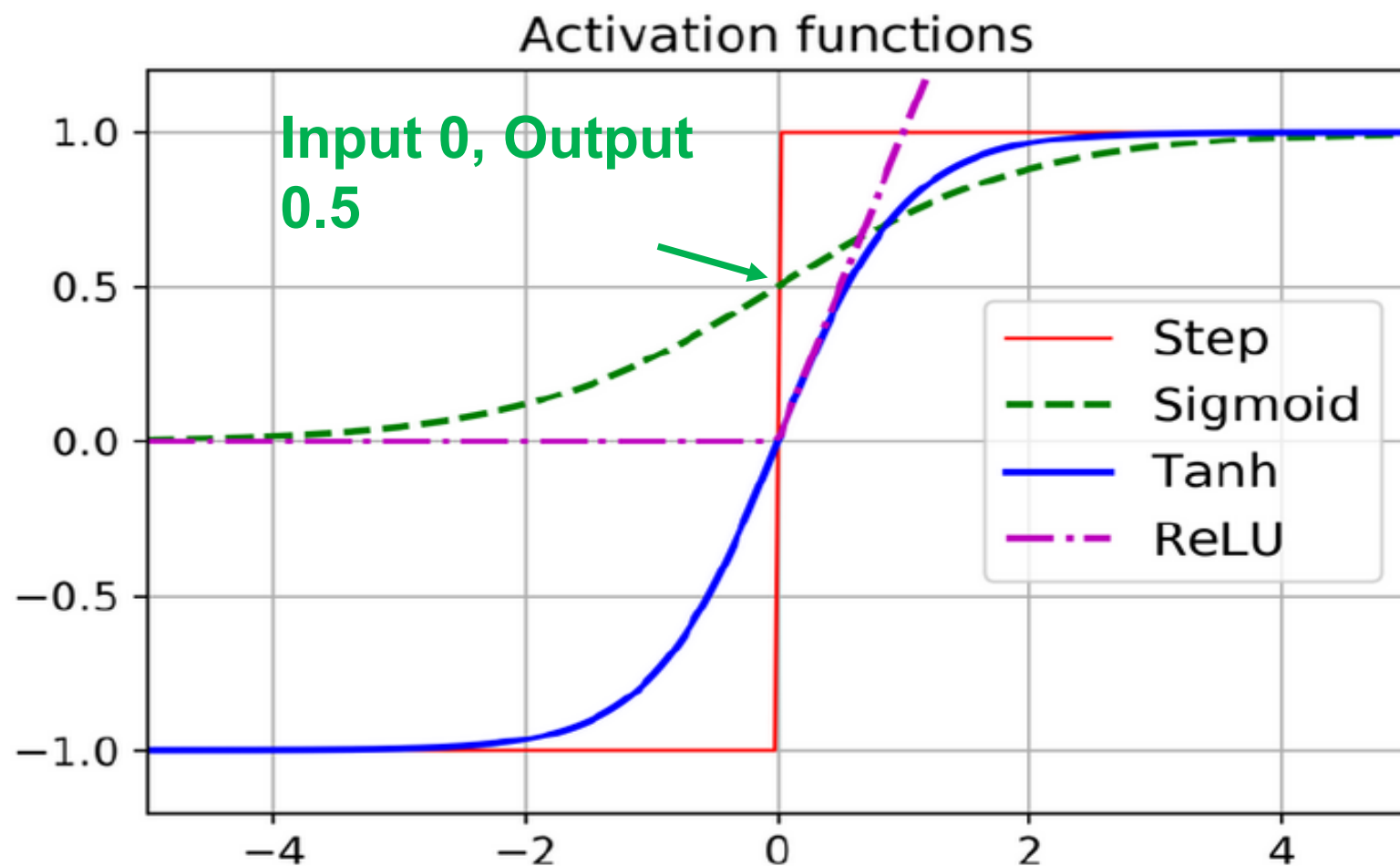
Exploding Gradient Problem: Gradients become excessively large during training (performance fluctuates wildly or fails to converge). Leads to unstable training and failure to converge.

- **Variance Mismatch:** Variance (spread) of the outputs of one layer does not match the variance of the inputs expected by the next layer, it can cause the gradients to increase exponentially during backpropagation.
- **Initialisation Issues:** If weights are initialised with values that are too large, the resulting gradients during backpropagation can also be very large, leading to instability.
- **Input Variability:** Large variability in input can lead to very large gradients during backpropagation, as the network tries to adjust weights to accommodate the wide range of input values.
- **Activation Functions:** Some activation functions, can cause large changes in the gradients for certain input values, leading to instability.

Exploding Gradients

Exploding Gradient Problem: Gradients become excessively large during training (performance fluctuates wildly or fails to converge). Leads to unstable training and failure to converge.

- **Activation Functions:** Some activation functions, can cause large changes in the gradients for certain input values, leading to instability.

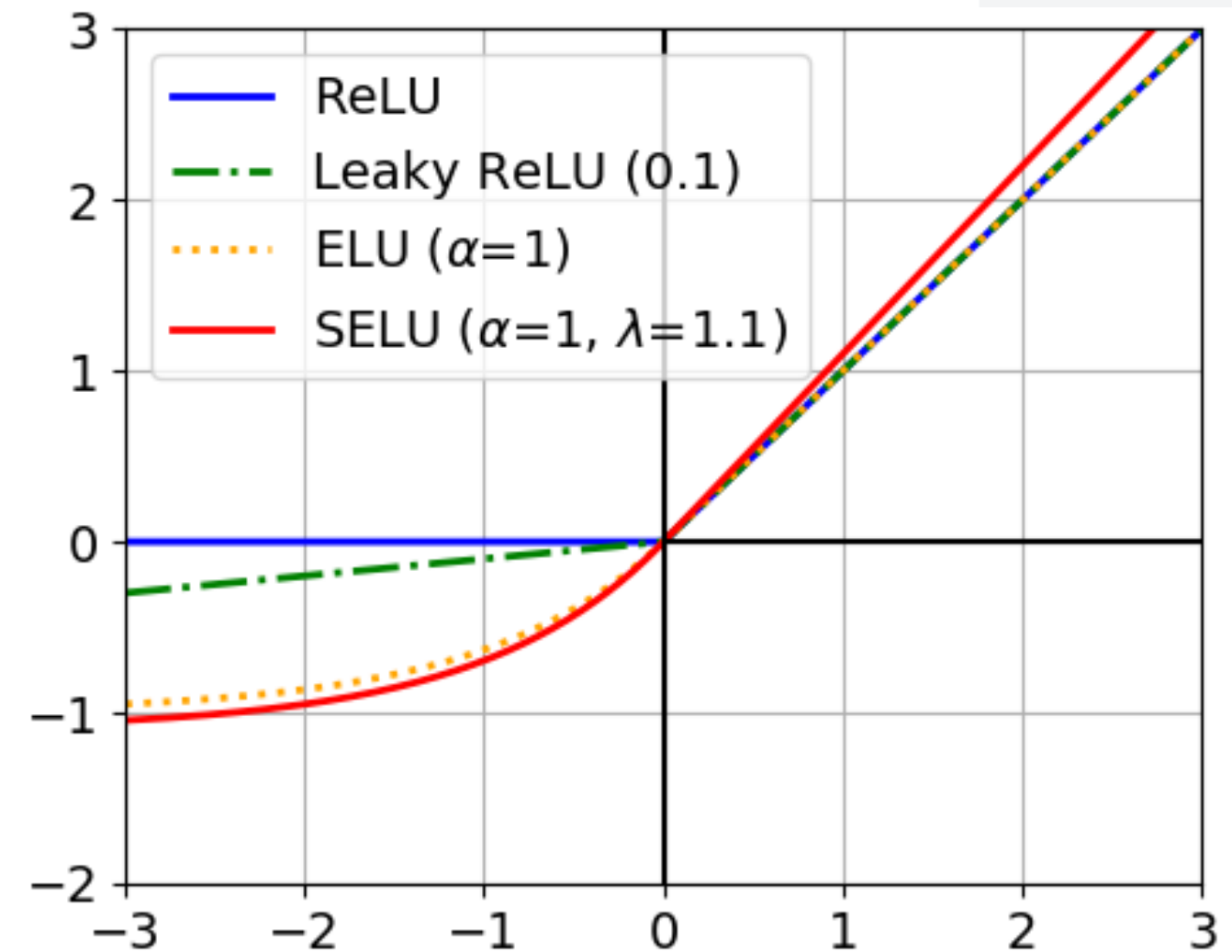


- **Sigmoid and Tanh:** These functions can cause large gradients in certain regions (steep slopes), leading to exploding gradients.
- **ReLU:** ReLU activation function helps mitigate this issue by maintaining a constant gradient (slope of 1) for positive inputs, which prevents the gradients from growing too large.

Mitigation: Non-Saturating Activation Functions

Idea: Try and pick activation function that does not saturate. Many options.

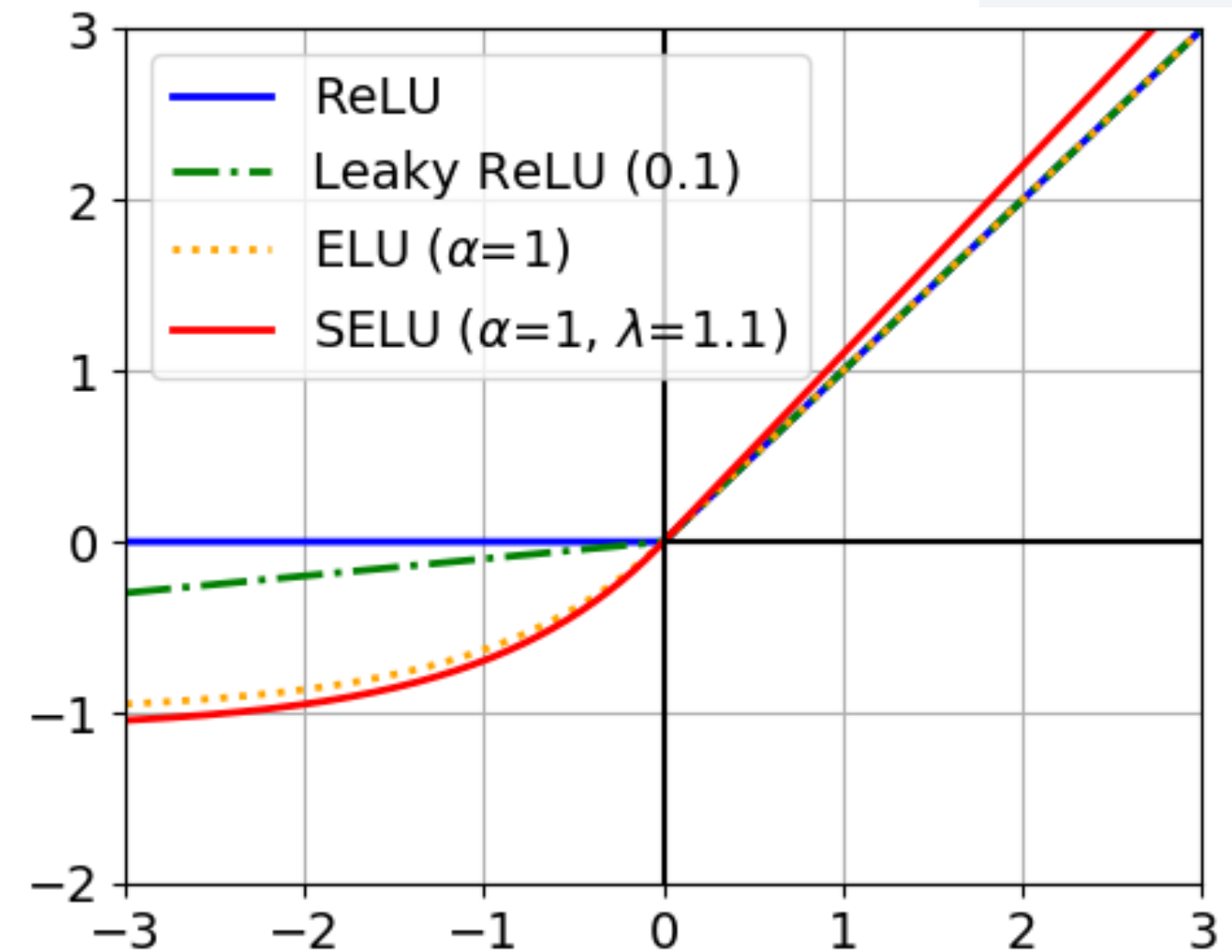
- **Sigmoid** saturates for low/high values
- **ReLU**: for non-negative values, becomes 0 and stays 0 (w/ 0 gradient)
- **Leaky ReLU**: Allows a small gradient when the input is negative
- **Exponential linear unit (ELU)**: Smooths the gradient when the input is negative, which can improve learning, converges faster, slower to compute
- **Scaled exponential linear unit (SELU)**: Self-normalising, which helps maintain a healthy variance throughout the network.



Mitigation: Non-Saturating Activation Functions

Idea: Try and pick activation function that does not saturate. Many options.

- ReLU is still popular – simple and encourages sparseness (sets neurons to have zero activation outputting 0 \rightarrow only subset of neurons active, simpler/more robust model)
- These can be useful, for example to stop vanishing gradients, but not necessarily the best thing to use
 - Be wary of generalisations, as no single thing works best in all situations
 - Many recommendations are based on empirical evidence and opinion



Mitigation: Batch Normalisation

- Adds **normalisation layer** before or after each hidden layer that learns optimal mean and scale of each input layer.

Key concepts:

- **Normalisation:** Ensures inputs have a consistent mean and standard deviation.
- **Learnable Parameters:** γ (scale) and β (shift) are adjusted during training to optimise performance.
- **Stability:** Helps stabilise and accelerate the training process, leading to better model performance.
 - i.e. reducing the change in the distribution of network activations due to the updates in the parameters of the previous layers.



THE UNIVERSITY
of ADELAIDE

150 YEARS

Mitigation: Batch Normalisation

Training for each batch:

1. Standardise to mean 0 and standard deviation 1 across current training batch (z-score normalisation)
2. Scale each input (adjust spread) → New model parameters γ fitted
3. Shift each input (adjusts 'position')/mean → New model parameters β fitted
4. Update moving average across batches of means μ and std. deviations σ to be used during prediction

$$\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$$

γ and β are learnable parameters adjusted during training to optimise performance.

Very commonly used and can be highly useful at improving optimisation, but not always...

Mitigation: Gradient Clipping

Purpose: Prevents gradients from becoming too large during backpropagation, which can destabilise training.

- Limits (clips) the gradient values to a maximum threshold during backpropagation.
- Particularly useful in recurrent neural networks (RNNs) as an alternative to batch normalisation.
- **Keras:** implemented as hyperparameters of optimiser
 - “`clipvalue`”: Limits the absolute value of the gradients per dimension, which may change the direction of the gradient vector.

Example: If `clipvalue` is set to 1.0, any gradient larger than 1.0 or smaller than -1.0 will be clipped to 1.0 or -1.0, respectively.

- “`clipnorm`”: Clips the L2 norm of the gradients, preserving the direction of the gradient vector.

Example: If `clipnorm` is set to 1.0, the gradient vector is scaled down to ensure its L2 norm does not exceed 1.0.

Mitigation: Initialisation strategies

Initialisation Strategies

- **Random Initialisation:** Initialise connection weights randomly with a mean of 0.
- **Statistical Initialisation:** Ensure weights maintain the same signal variance throughout layers (variability of the values/activations in network).

Normal and Uniform Distributions:

- **Normal Distribution:** Weights are initialised based on a normal (Gaussian) distribution. Variance depends on the activation function.
- **Uniform Distribution:** Weights initialised between $-r$ and r , where $r = \sqrt{\frac{3}{n}}$, To ensure that the weights are spread out evenly across a range.
- **Keras Default:** Glorot (Xavier) Initialisation: Uses uniform distribution.

fan_{in} = number inputs of layer/neurons
 fan_{out} = number of neurons/outputs
 $fan_{avg} = (fan_{in} + fan_{out})/2$ (i.e. average)

Initialisation	Activation functions	Variance σ^2 of normal distr.
Glorot	None, tanh, logistic, softmax	$1/fan_{in}$
He	ReLU and variants	$2/fan_{in}$
LeCun	SELU	$1/fan_{in}$

Which answer is correct?

Which activation function is most likely to avoid the vanishing gradient problem?

- Sigmoid
- Tanh
- ReLU
- Softmax



THE UNIVERSITY
of ADELAIDE

150 YEARS

Which answer is correct?

Which of the following is a technique to handle exploding gradients?

- Weight decay
- Gradient clipping
- Learning rate decay
- Weight normalisation



THE UNIVERSITY
of ADELAIDE

150 YEARS



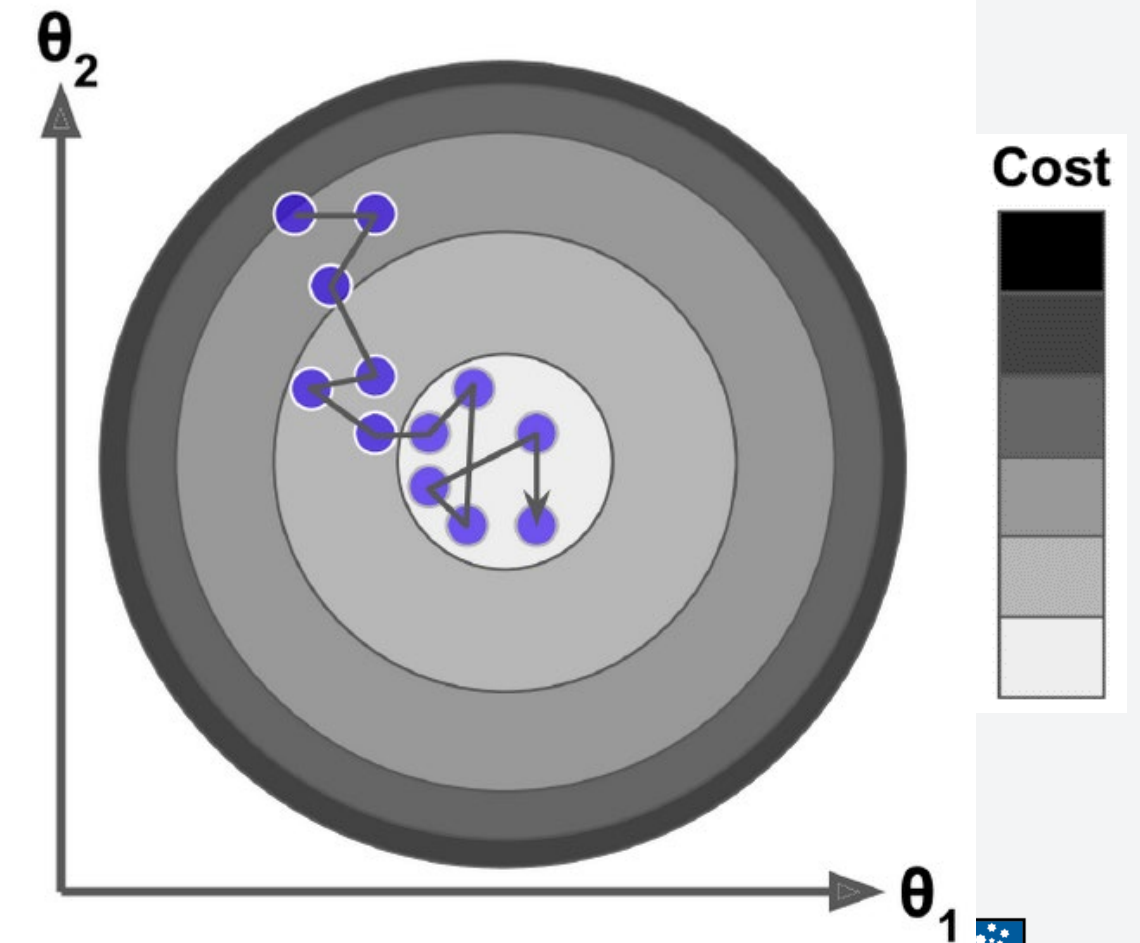
Optimisers

Fast Optimisers: (Baseline) Gradient Descent

Gradient Descent is the foundation of many optimisation algorithms used in ML.

Gradient Descent (recap):

- **Idea:** Move downhill along the gradient to find the minimum cost.
- **Step Size/Learning Rate:** Determines how big each step is during optimisation.
- **Efficiency:** Not always the most efficient method.
- **Stochastic Version:** Uses random samples to estimate gradients, which can be noisy.
- **Variations:** Many optimisers improve upon basic gradient descent.
- **Inspiration:** Optimisation algorithms often inspired by physics (e.g., momentum).



Momentum Optimisation (1964)

Problem of GD:

- If gradient small, then steps small and convergence slow

Idea: Use gradients of previous steps to build 'momentum'

- Helps take bigger steps towards the minimum
- Gradient length = acceleration
- Momentum = Using past gradients to speed up learning (adds fraction of past gradient to current one)

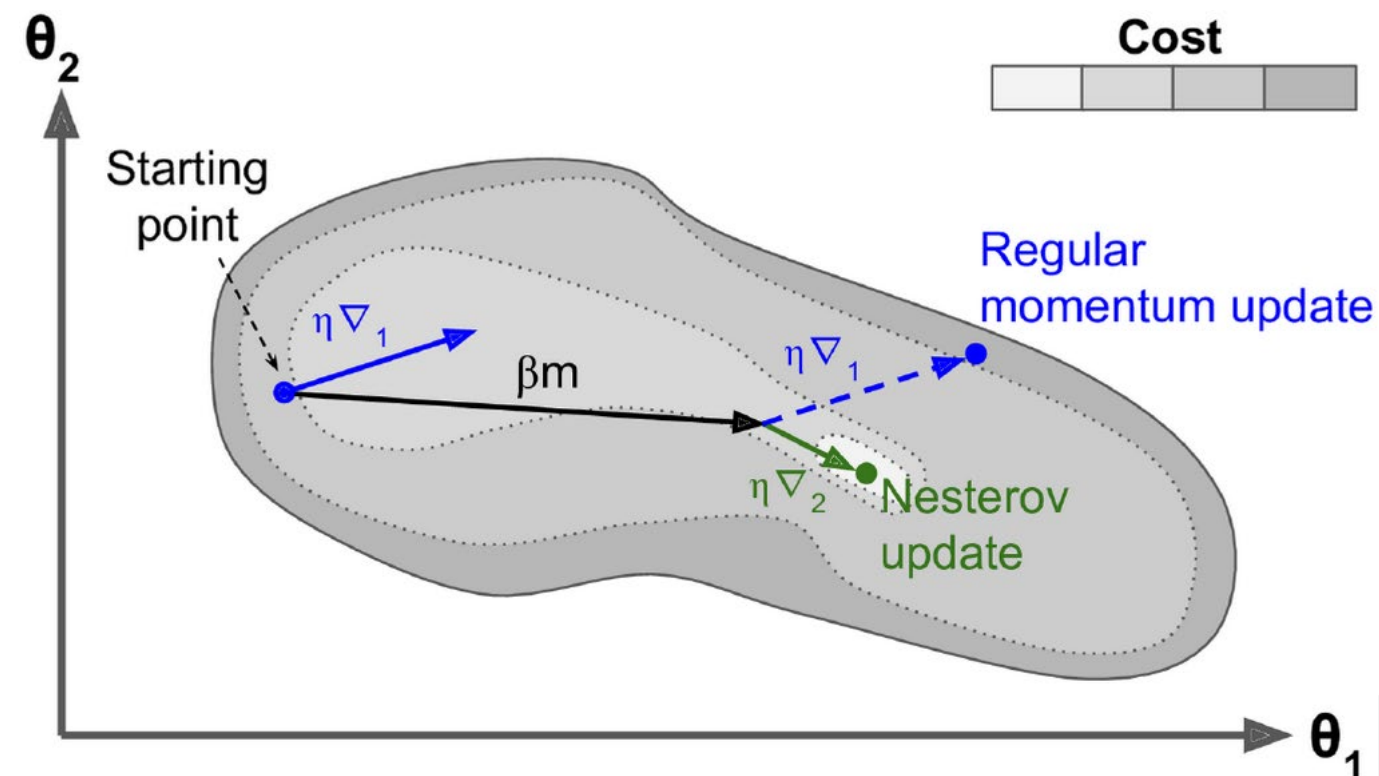
Pros:

- Faster Convergence: Helps escape flat regions and reach the minimum faster
- Reduces Oscillation: Smooths out the path to the minimum

Con: Can overshoot the minimum

Nesterov Accelerated Gradient (1983)

- **Idea:** Predicts the future position based on the current momentum. Computes the gradient after making this prediction
 - More Accurate Updates: By looking ahead, we make more informed updates, leading to better performance.
 - Reduces Oscillations: This method helps to smooth out the path towards the minimum, reducing the back-and-forth oscillations.

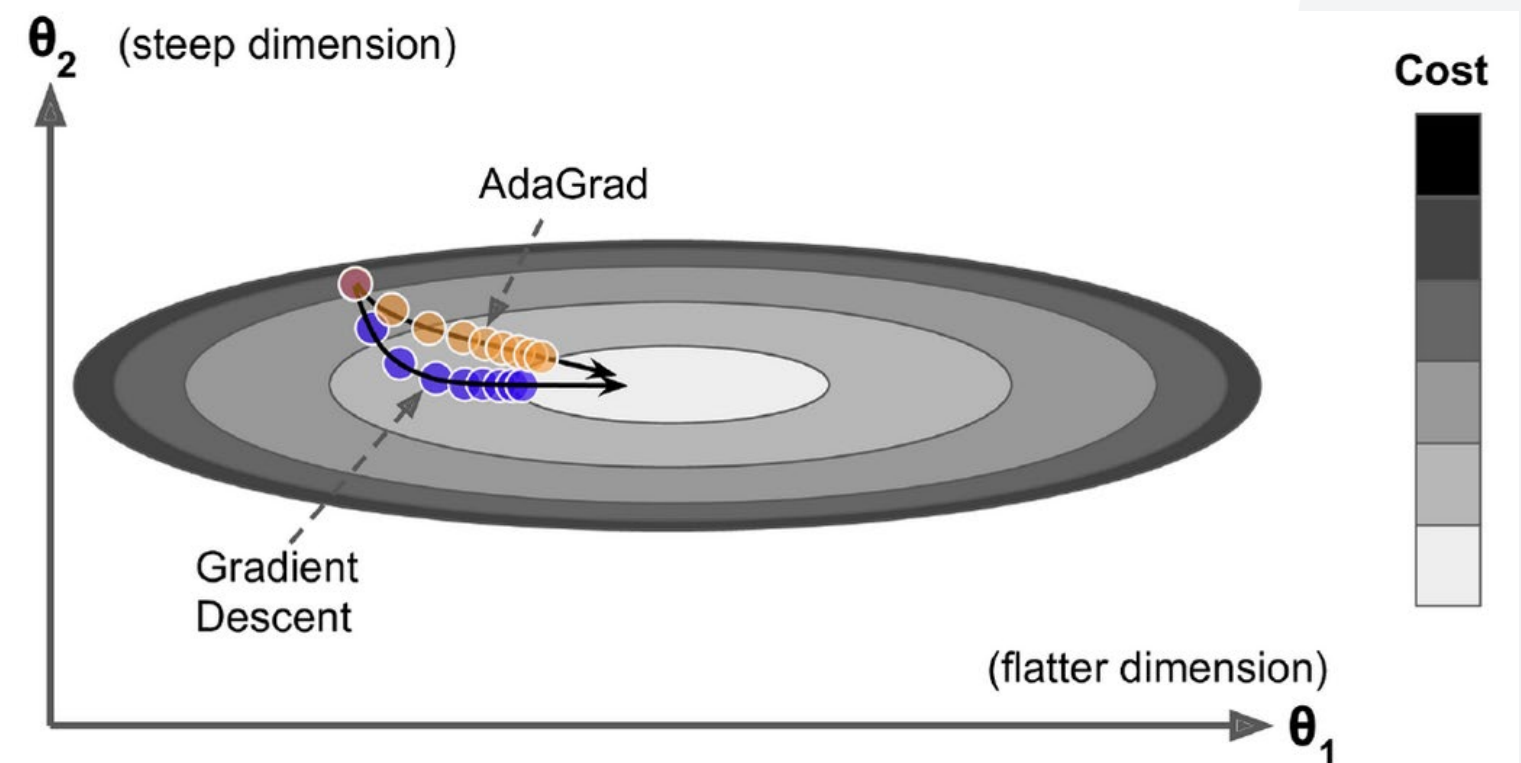


Adaptive Subgradient Opt. (AdaGrad, 2011)

Problem: When the cost function has an elongated bowl shape, the optimisation process can get overly attracted to steep slopes and struggle to make progress in flatter regions.

Idea: Normalise the gradient step by taking into account all previous gradients.

- Adaptive Learning Rate: Each parameter gets its own learning rate that adapts over time = more efficient.
- Helps with Sparse Data: Can adjust learning rates for infrequent features.
- Con: Over time, the learning rate can become very small, stopping/slowing optimisation before reaching minimum.



RMSProp (2012)

- **Fixes AdaGrad:** Decays the cumulative sum of gradient squares.
- **Recent Steps Matter:** Only the most recent preceding steps have significant influence.
- **New Hyperparameter (ρ):** Regulates the influence of previous gradients versus the current gradient.
- **Benefit:** Prevents Learning Rate Decay - Avoids the issue of learning rates becoming too small over time.



THE UNIVERSITY
of ADELAIDE

150 YEARS

Adaptive Moment Estimation (Adam, 2014)

- **Combines Momentum and RMSProp:**
 - Uses momentum with exponential decay.
 - Normalises cumulative sum of squared gradients with separate exponential decay.
- **Iteration Number t :** Gradually slows down optimization over time.
- **Adaptive Learning Rate:** Learning rate is adjusted based on adaptive processes.
- **Benefits:**
 - Efficient and effective optimization
 - Works well with sparse data and large datasets.

Adam Variants: AdaMax, Nadam

AdaMax:

- **Modification:** Uses the infinity norm (maximum) instead of the L2 norm (sum of squares).
 - Takes the maximum absolute value of the gradients, making it more robust to large updates.
- **Benefit:** More stable with larger updates.

Nadam:

- **Modification:** Combines Adam with Nesterov Accelerated Gradient (NAG).
 - Looks ahead to future position (Nesterov), providing smoother and more accurate updates, reducing oscillations

Optimiser Comparison (Géron 2019)

Trade-off between:

- Speed
 - Convergence quality
 - Hyperparameter count
 - Underlying assumptions about cost function landscape
- ❖ Based on empirical tests
 - ❖ May not apply in all cases
 - ❖ Can change as new approaches emerge

Bad - * Average - ** Good - ***

Optimizer (Keras)	Convergence speed	Convergence quality
SGD	*	***
Momentum	**	***
Nesterov	**	***
Adagrad	***	* (stops too early)
RMSprop	***	** or ***
Adam	***	** or ***
Nadam	***	** or ***
AdaMax	***	** or ***

Early Stopping

- Stop if no improvement for X iterations (“patience”) →
- Tolerance (“min_delta”)
- Implemented as a *Callback*

```
tf.keras.callbacks.EarlyStopping(  
    monitor="val_loss",  
    min_delta=0,  
    patience=0,  
    verbose=0,  
    mode="auto",  
    baseline=None,  
    restore_best_weights=False,  
)
```

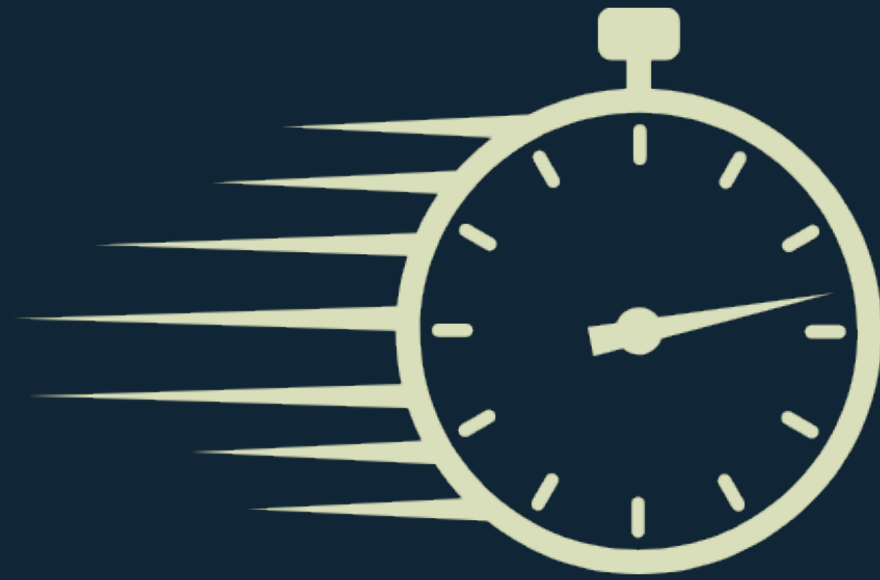
```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)  
# This callback will stop the training when there is no improvement in  
# the validation loss for three consecutive epochs.  
model = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])  
model.compile(tf.keras.optimizers.SGD(), loss='mse')  
history = model.fit(np.arange(100).reshape(5, 20), np.zeros(5),  
                    epochs=10, batch_size=1, callbacks=[callback],  
                    verbose=0)  
len(history.history['loss']) # Only 4 epochs are run.
```

https://keras.io/api/callbacks/early_stopping/



THE UNIVERSITY
of ADELAIDE

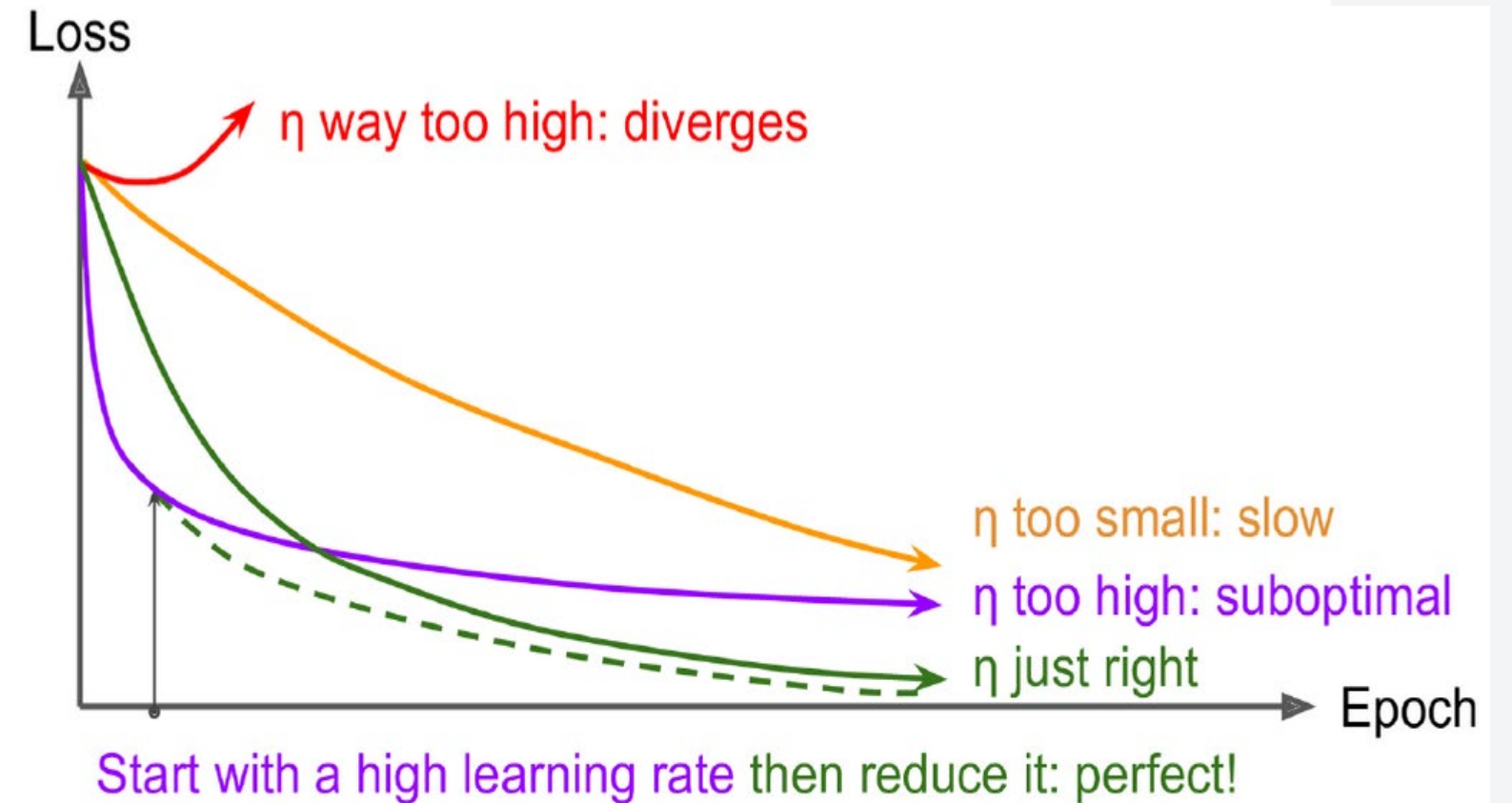
150 YEARS



Learning rate scheduling + 1cycle approach

Learning Rate

- **Simplest choice:** constant learning rate η
- **Best learning rate:**
 - Typically larger at the start, smaller towards the end
 - Influenced by initialisation, optimiser parameters, model, and data
 - Also depends on cost function in landscape high-dimensional weight space, i.e. the data



Learning Rate Scheduling

- Adjust learning rate based on **iteration number t** , error, or a test on data (schedule)
- Very empirically based, often trial and error, as can be influenced by many factors

- **Piecewise constant:**

$$\eta(t) = \begin{cases} 1 - 5 & 0.1 \\ 6 - 10 & 0.01 \\ > 10 & 0.001 \end{cases}$$

- **Power scheduling:**

$$\eta(t) = \frac{\eta_0}{\left(1 + \frac{t}{s}\right)^c}$$

Annotations for the power scheduling formula:

- η_0 : Initial learning rate η_0
- c : Power c (e.g. 1)
- s : Step

- **Exponential scheduling:**

$$\eta(t) = \eta_0 0.1^{\frac{t}{s}}$$

Performance scheduling:

- E.g. if validation error not decreasing, reduce learning rate by factor

Implementation in Keras/ tk.keras:

- Built-in parameter of optimiser
- Callback in model (LearningRateScheduler)
- Schedule object of tk.keras in optimiser



THE UNIVERSITY
of ADELAIDE

150 YEARS

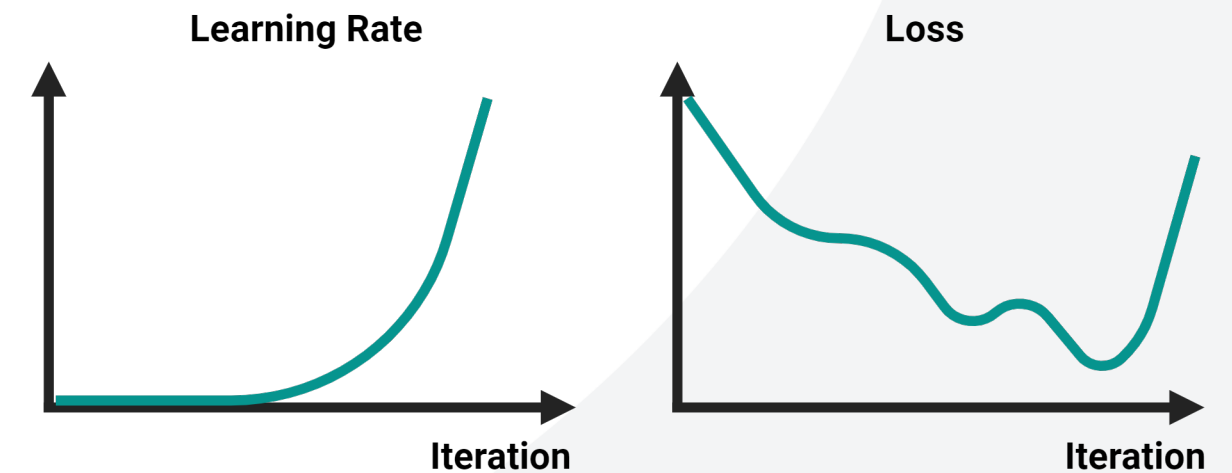
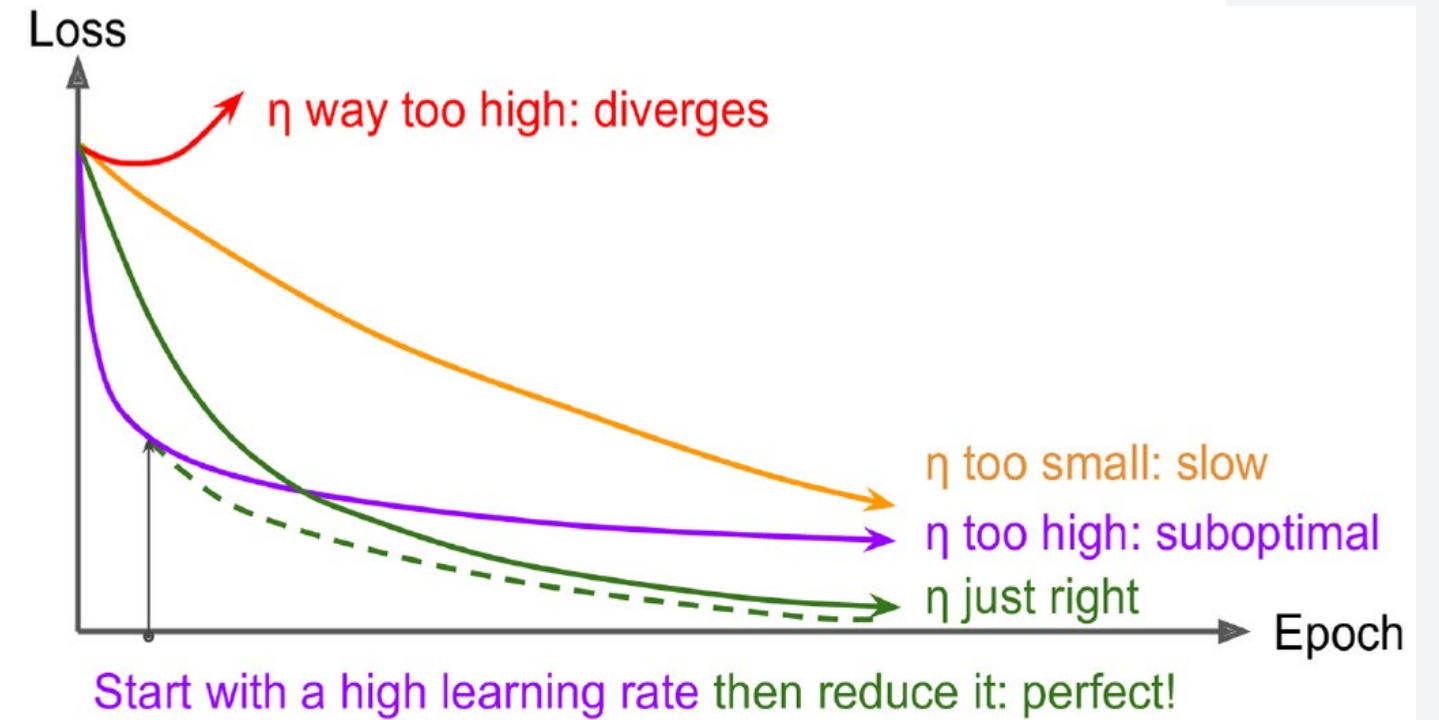
1cycle Approach (Smith 2018)

Cyclical Learning:

- Start with minimum learning rate (LR)
- Increase to a maximum LR
- Decrease back to minimum LR
- Can be repeated (if not 1cycle)

“1cycle” scheduling (Smith 2018)

- LR range test:
 - Increase LR until training starts to diverge (error goes up) = maximum



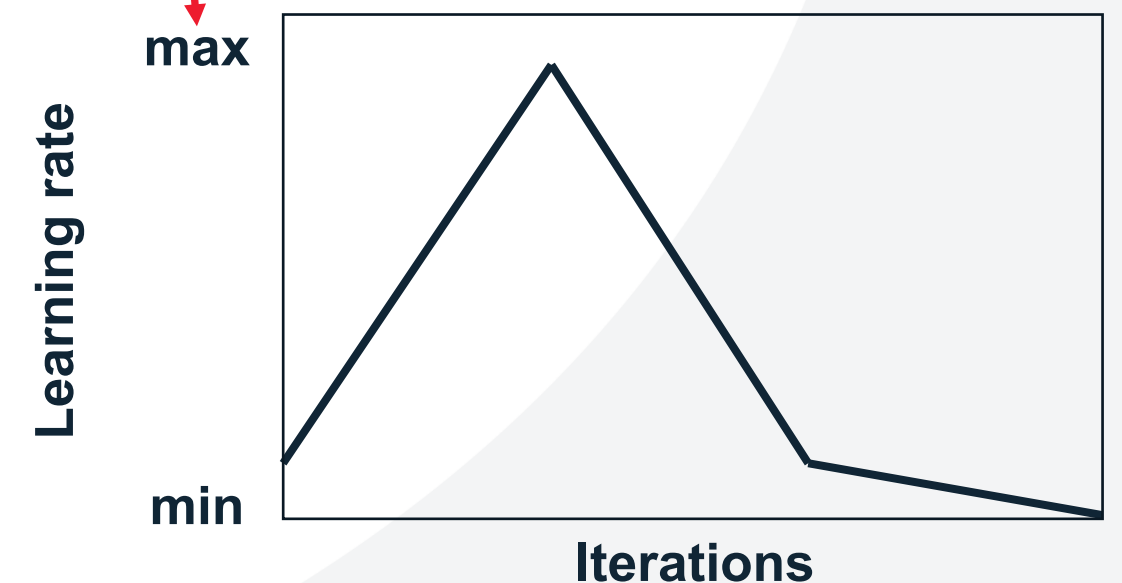
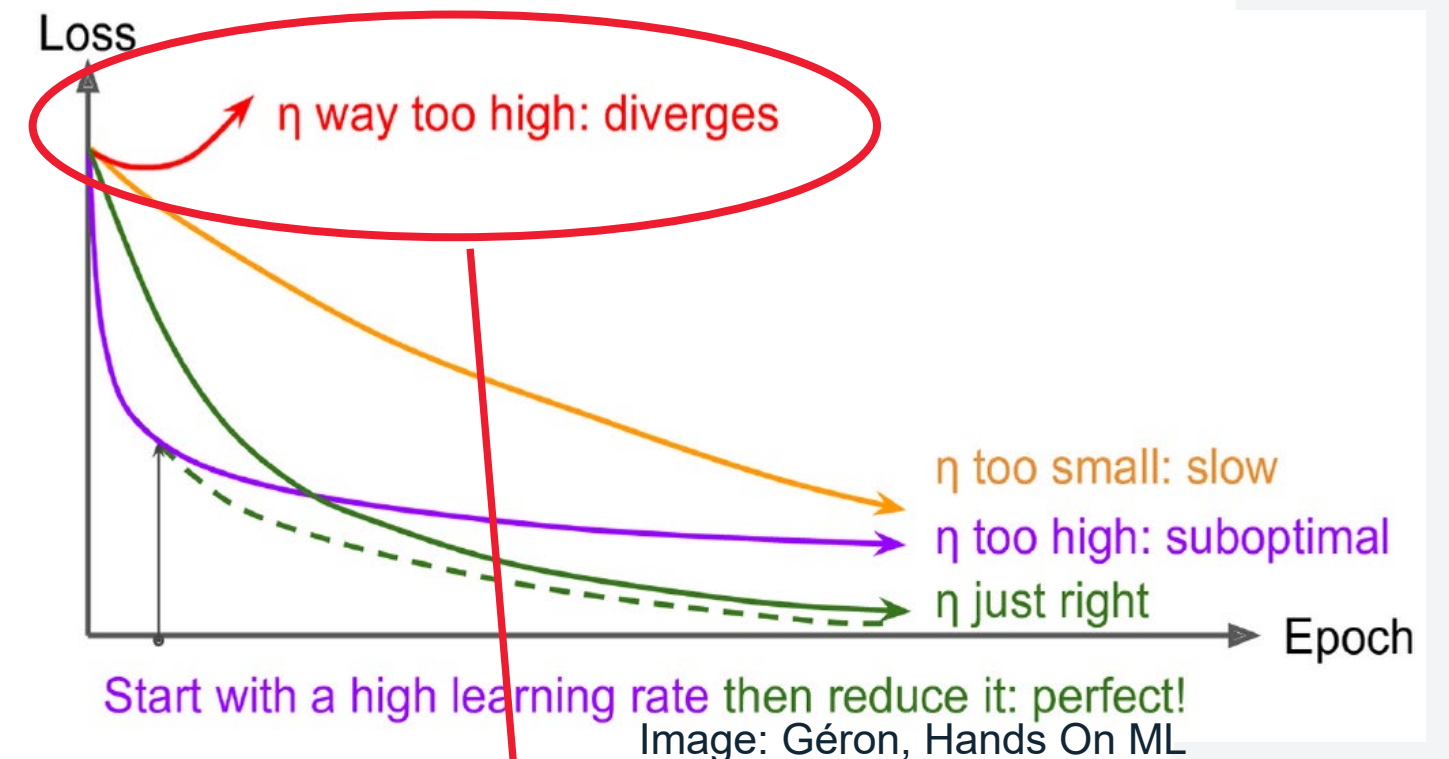
1cycle Approach (Smith 2018)

Cyclical Learning:

- Start with minimum learning rate (LR)
- Increase to a maximum LR
- Decrease back to minimum LR
- Can be repeated (if not 1cycle)

“1cycle” scheduling (Smith 2018)

- LR range test:
 - Increase LR until training starts to diverge (error goes up) = maximum
 - Minimum = maximum/10
- Use 1 cycle with linear increase/decrease of LR
- After that drop LR linearly to very small value



Configurations in Practice (Géron 2019)

Hyperparameter	Dense NN default	Self-normalising DNN default
Kernel initialiser	He ($\sigma^2 = 2/\text{fan}_{\text{in}}$)	LeCun ($\sigma^2 = 1/\text{fan}_{\text{in}}$)
Activation Function	ELU	SELU
Normalisation	None if shallow, batch normalisation if deep	None (self-normalising)
Regularisation	Early stopping (+ L2 norm regularisation if needed)	Alpha drop out if needed
Optimiser	Momentum (or RMSProp or Nadam)	Momentum (or RMSProp or Nadam)
Learning rate schedule	1cycle	1cycle

- *Field and libraries under active development, defaults may change!*

Which answer is correct?

What is the primary benefit of the 1cycle learning rate policy? (Select all that apply)

- Helps escape local minima
- Maintains constant learning rate
- Increases learning rate to maximum at the start
- Balances exploration and fine-tuning



THE UNIVERSITY
of ADELAIDE

150 YEARS

Summary

- 1. The process of training neural networks**
- 2. Vanishing and exploding gradients**
 - Four ways of dealing with this
- 3. Optimisers**
 - Range of options available
- 4. Learning rate scheduling and 1cycle method**



THE UNIVERSITY
of ADELAIDE

150 YEARS

Questions?

dhani.dharmaprani@adelaide.edu.au