Universität Augsburg University

Deep Learning

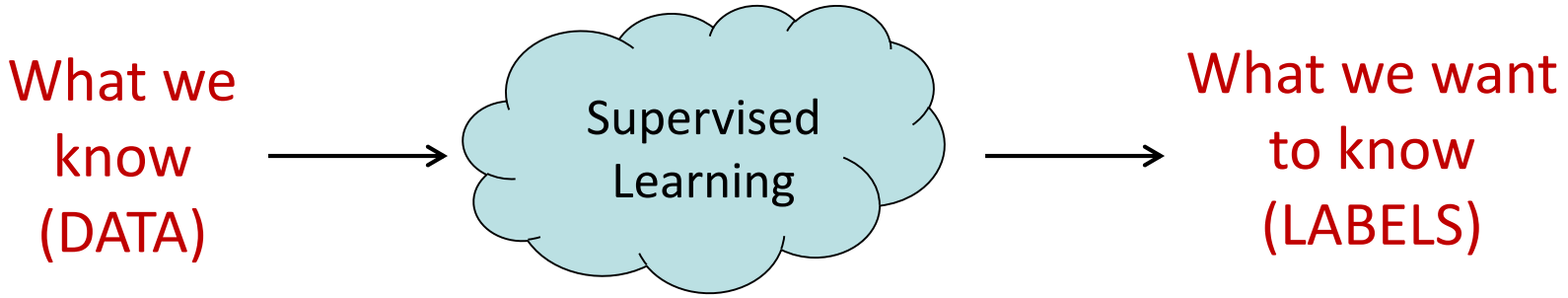
Backpropagation & Regularisation

Tuesday 26th November

Dr. Nicholas Cummins

Revision

- **Supervised Learning**
  - Transforming one dataset into another
  - Taking what you know as input and transforming it into what you want to know at the output
  - Input: observable, recordable, and knowable data
  - Output: data for logical analysis



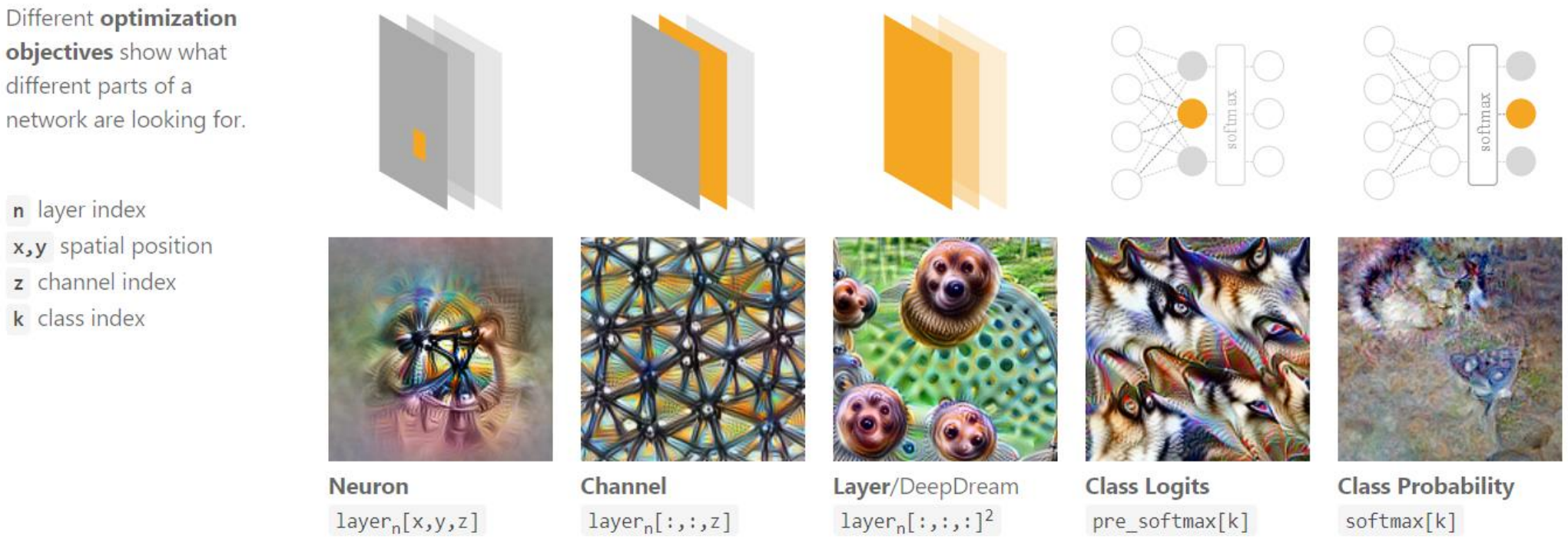
- **Data-Transformation**
  - A machine-learning model transforms its input data into meaningful outputs

$$x \xrightarrow{f(\cdot)} y$$

  - This process that is “learned” from exposure to known examples of inputs and outputs
  - Learning to meaningfully transfer data is the central problem in machine learning
    - Learn useful representations of the input data at hand
    - These representations should get us closer to the expected output

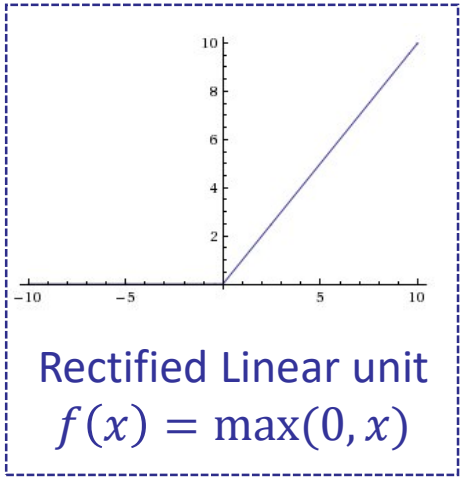
Image Source: https://distill.pub/2017/feature-visualization/

- **Deep learning is a set of multistage techniques for learning successive data representations**
  - A DNN transforms input data into a set of representations that are increasingly informative about the final result

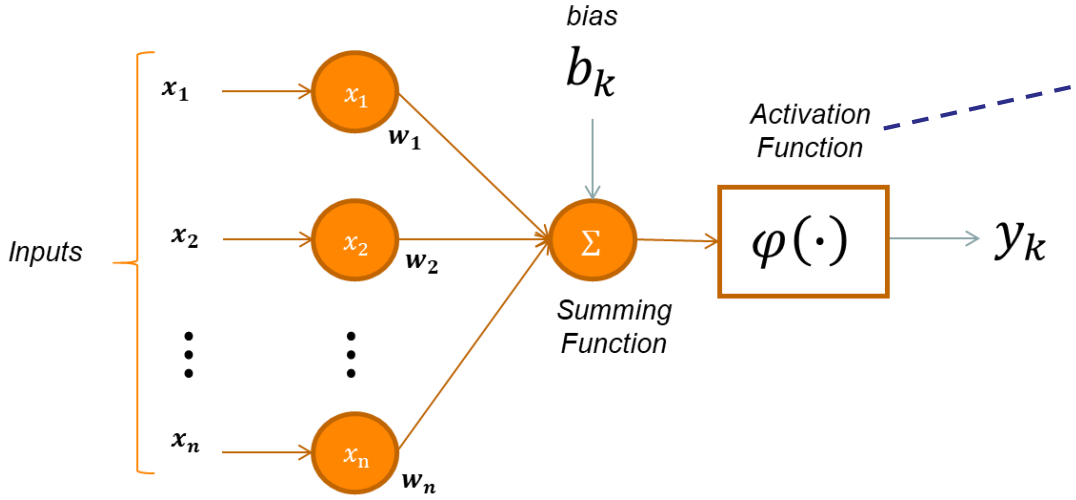


Artificial Neurons

- Building block of neural networks
  - Combines different inputs to make a single output
- Activations Function
  - Inclusion of nonlinearities
  - Enable learning of complex patterns



Rectified Linear unit
$f(x) = \max(0, x)$



$$y_k = \varphi\left(\begin{pmatrix} w_1 & w_2 & \dots & b \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ 1 \end{pmatrix} \right) = \varphi(w_1 x_1 + w_2 x_2 + \cdots + b)$$

- **Activation functions**
  - A function applied to neurons in a layer during prediction.
  - **Core Properties**
    - The function must be continuous and infinite in domain
    - Good activation functions are monotonic, never changing direction
    - There is no point at which two input values of have the same output value
    - Good activation functions are nonlinear
      - Allow for selective correlation: increase or decrease how correlated the neuron is to all the other incoming signals.
    - Good activation functions (and their derivatives) should be efficiently computable

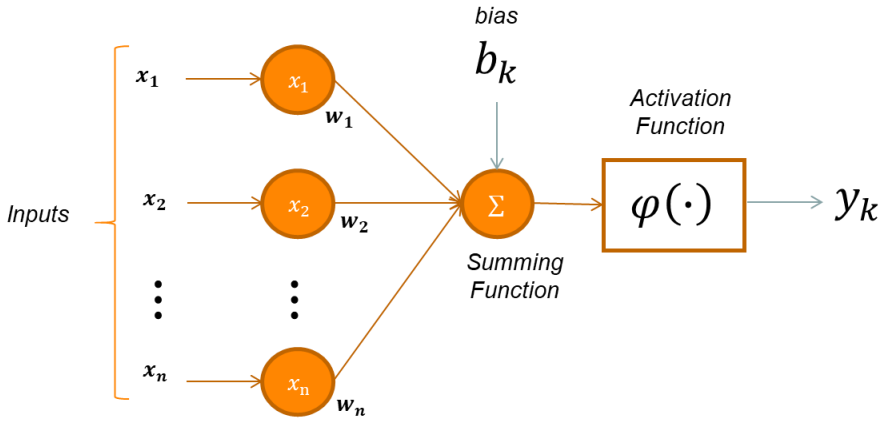
- **Linear**
  - $f(x) = ax$
  - Range: $-\infty$ to ∞
- **Sigmoid**
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - Range: 0 to 1
- **Hyperbolic Tangent**
  - $tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
  - Range: -1 to 1
- **Rectified Linear Unit**
  - $ReLU(x) = \max(0, x)$
  - Range: $-\infty$ to ∞
- **Leaky Rectified Linear Unit**
  - $LReLU(x) = \begin{cases} \alpha x \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$
  - Range: $-\infty$ to ∞
- **Softmax**
  - $S(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$
  - Range: 0 to 1

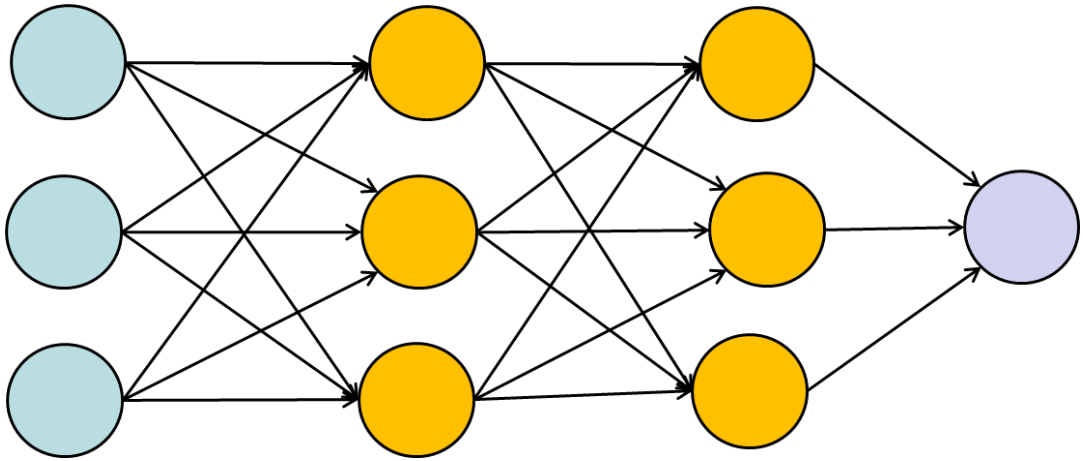
- **What is the bias node?**
  - Bias is simply a constant value (vector) that is added to the dot product of inputs and weights

$$y_k = \varphi\left(\sum_n w_n x_b + b\right)$$

  - Allows the activation function to be shifted left or right
    - Allows better fit to the data
  - Biases are tuned alongside weights by learning algorithms
    - E.g. Via Gradient Descent
    - Typically initialised to be zero
    - Note biases differ from weights is that they are independent of the output from previous layers

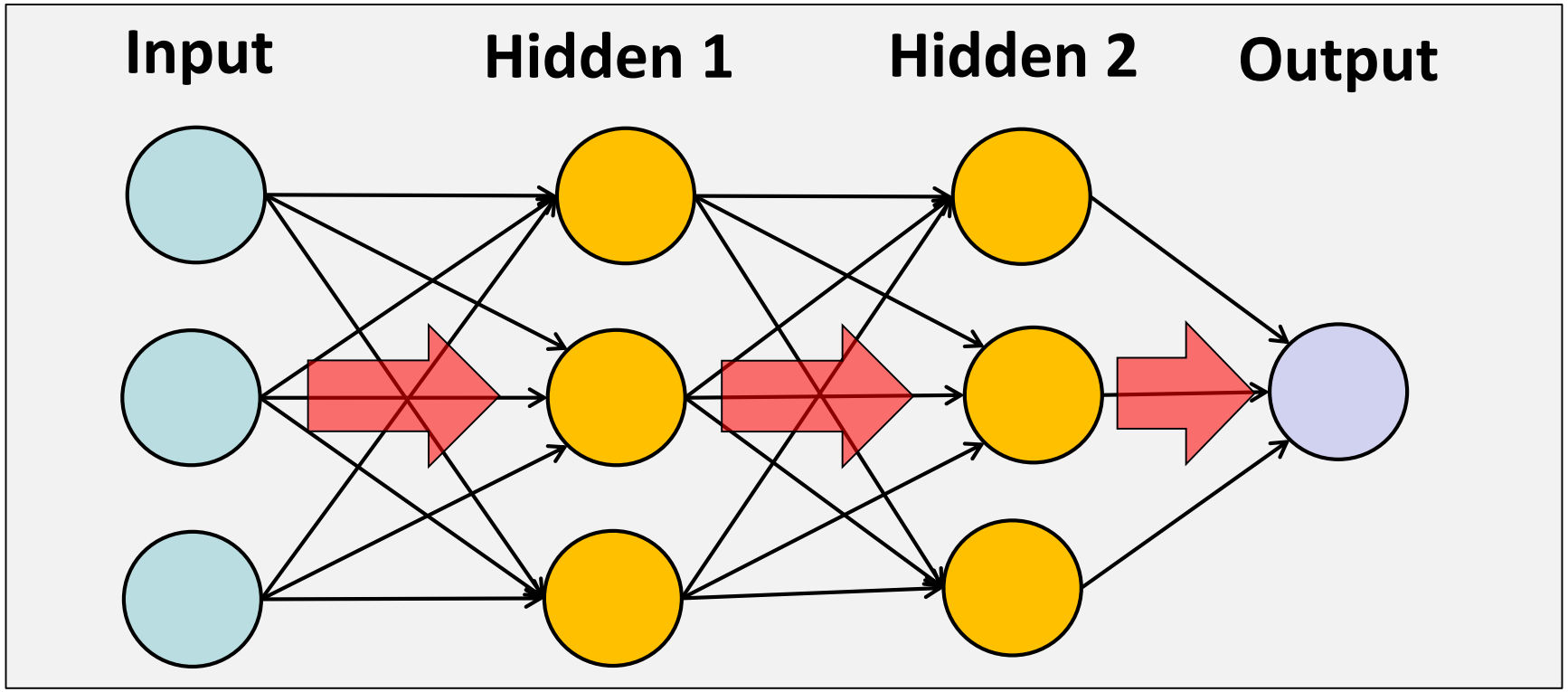
- **Neural networks**
  - In deep learning, the layered representations are (almost always) learned via models called neural networks structured in literal layers stacked on top of each other



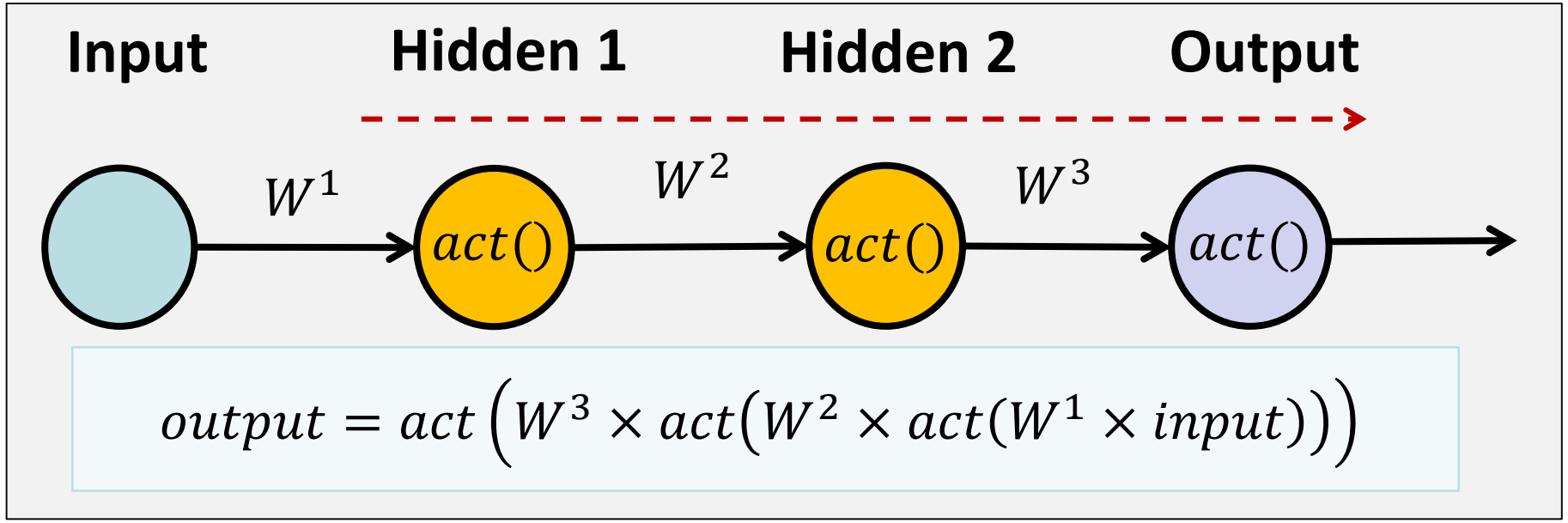


$$\varphi\left((w_1 \quad w_2 \quad \dots \quad b)\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ 1 \end{pmatrix}\right) = \varphi(w_1 x_1 + w_2 x_2 + \cdots + b) = y_k$$

- **Forward Propagation**
  - Information flows from input to output to make a predication



- **Forward Propagation**
  - Each neuron is a function of the previous one connected to it
    - Output is a composite function of the weights, inputs, and activations
      - Change any one of these and ultimately the output well change

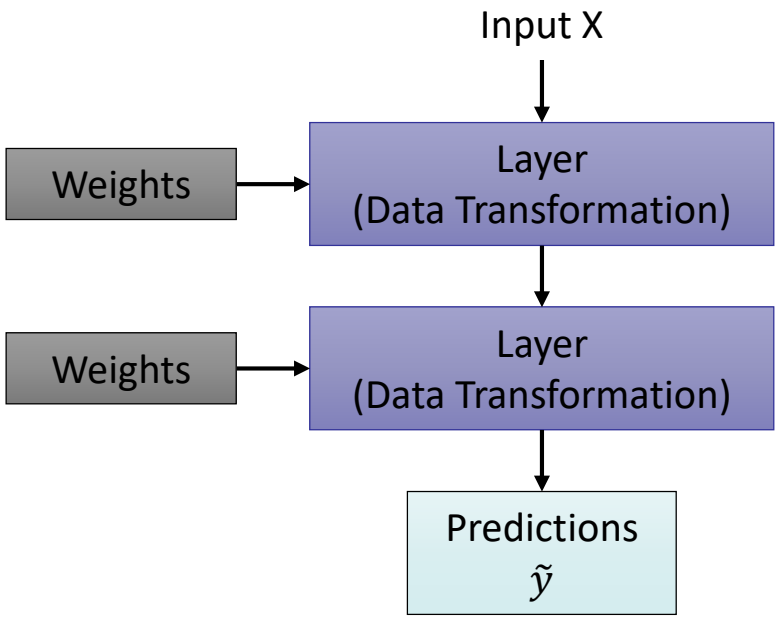


- **Complex Predictions**
  - Concatenation networks and propagation information
    - **Feedforward:** the information flows through the network from the input space $\boldsymbol{x}$, through the intermediary layers, to the output $\boldsymbol{y}$.
    - **Recurrent:** inclusion of looping mechanisms to allow information to flow from one step to the next
  - Weights equal knowledge
    - A networks uses the *knowledge* captured in the weights to *interpret* the input data to make a prediction
    - Learning involves iteratively setting the weight values so that the network can predict accurately

- **Weights Learnt via Gradient Descent**
  - Update weights to minimise loss function
  - This is achieved by taking the gradient of loss function with respect to the weights

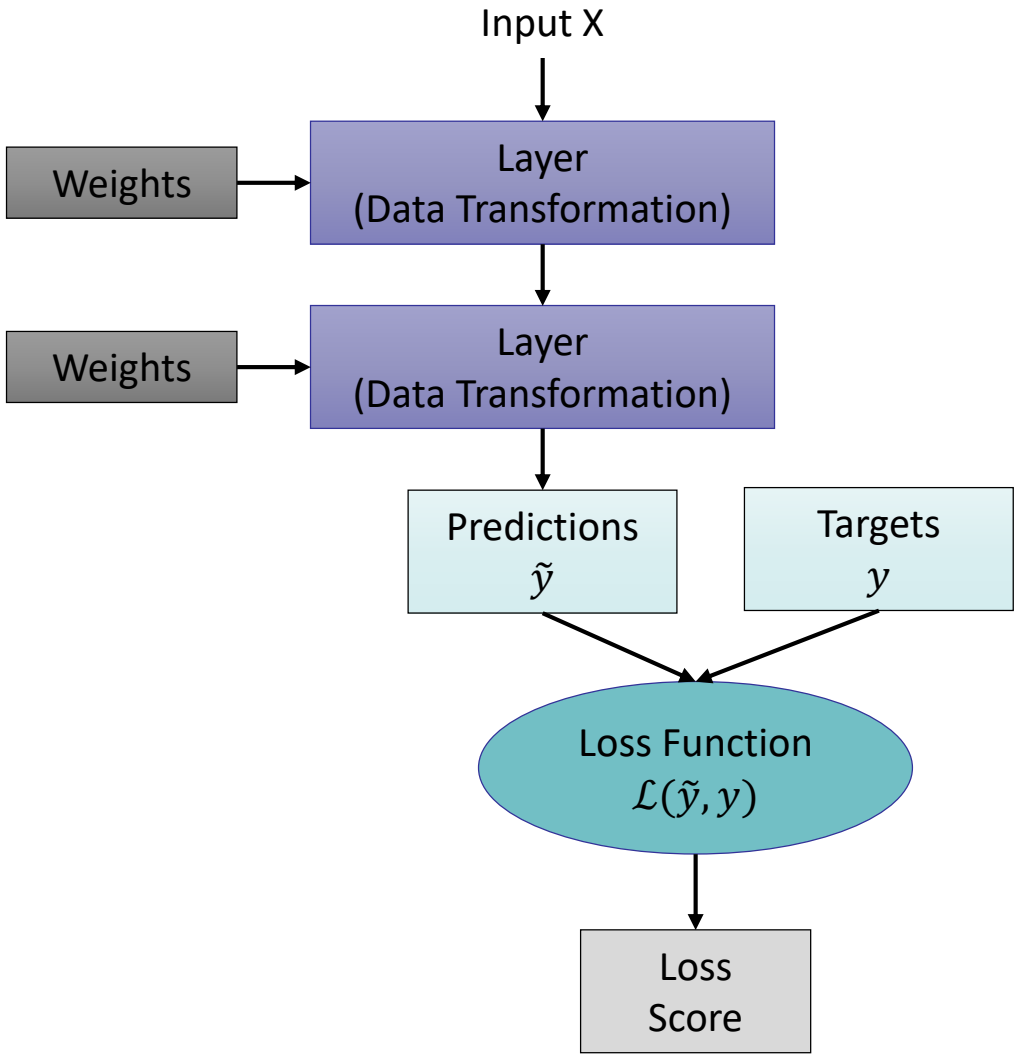
$$W = W - \alpha \frac{\partial \mathcal{L}}{\partial w}$$

  - Not a trivial process as neural networks are structured as a series of layers
  - A single network can contain many millions of weights, and modifying the value of one weight will affect the behaviour of all the others



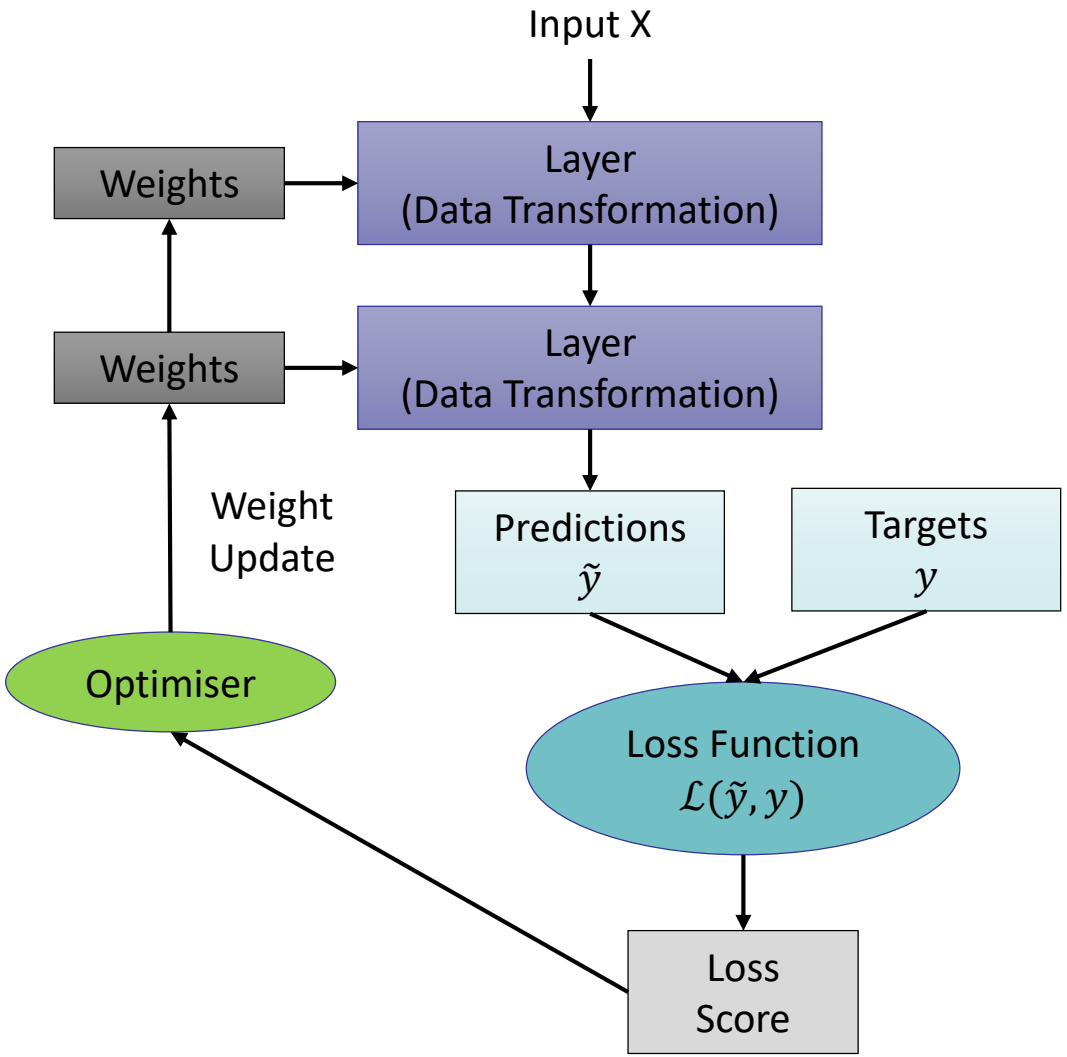
- **Learning in Deep Neural Networks**
  - To control weight updates in neural networks we use a **loss function** to how far an output prediction is from what we expected
  - The loss function computes a single value relating to network performance
  - Measures the difference between what we have predicted, $\tilde{y}$, with the what it should predicted y.

$$\mathcal{L}(\tilde{y}, y)$$

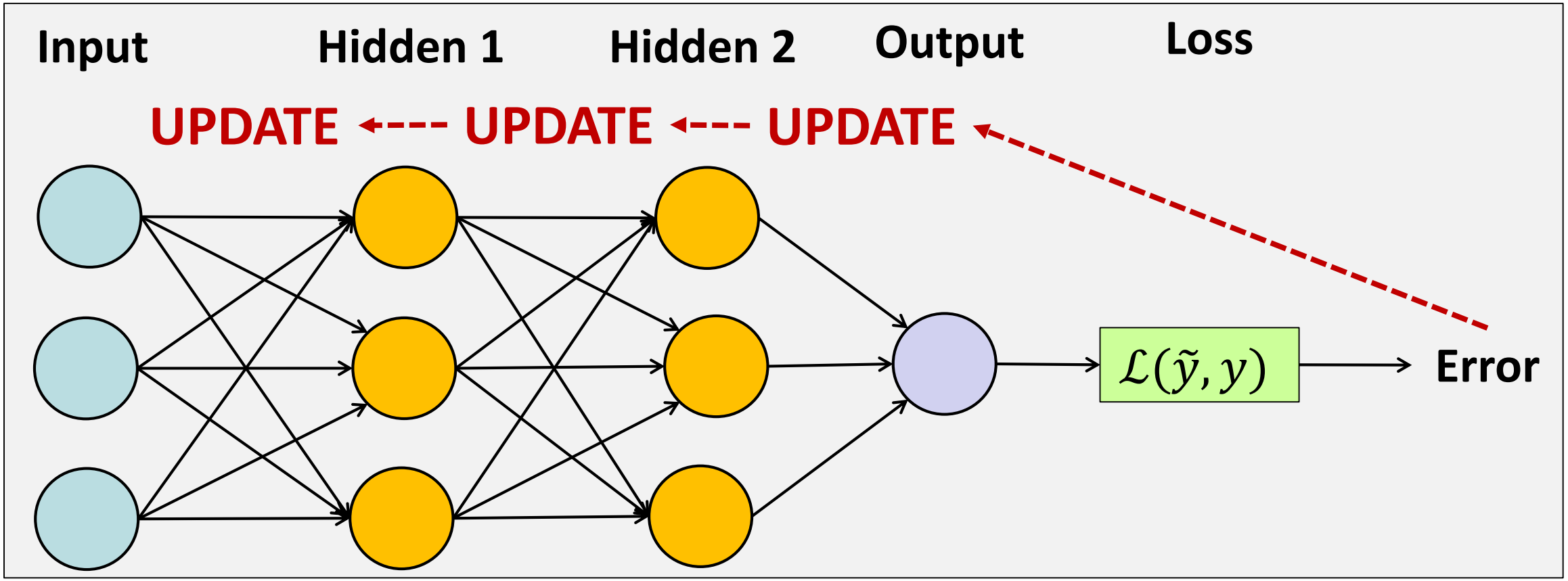


- **Regression**
  - Predicting a single numerical value
  - Final activation: **Linear**
  - Loss function: **Mean Squared Error**
- **Binary outcome**
  - Data is or isn't a class
  - Final Activation function: **Sigmoid**
  - Loss function: **Binary Cross Entropy**
- **Single label from multiple classes**
  - Multiple classes which are exclusive
  - Final Activation function: **Softmax**
  - Loss function: **Cross Entropy**
- **Multiple labels from multiple classes**
  - If there are multiple labels in your data
  - Final Activation function: **Sigmoid**
  - Loss function: **Binary Cross Entropy**

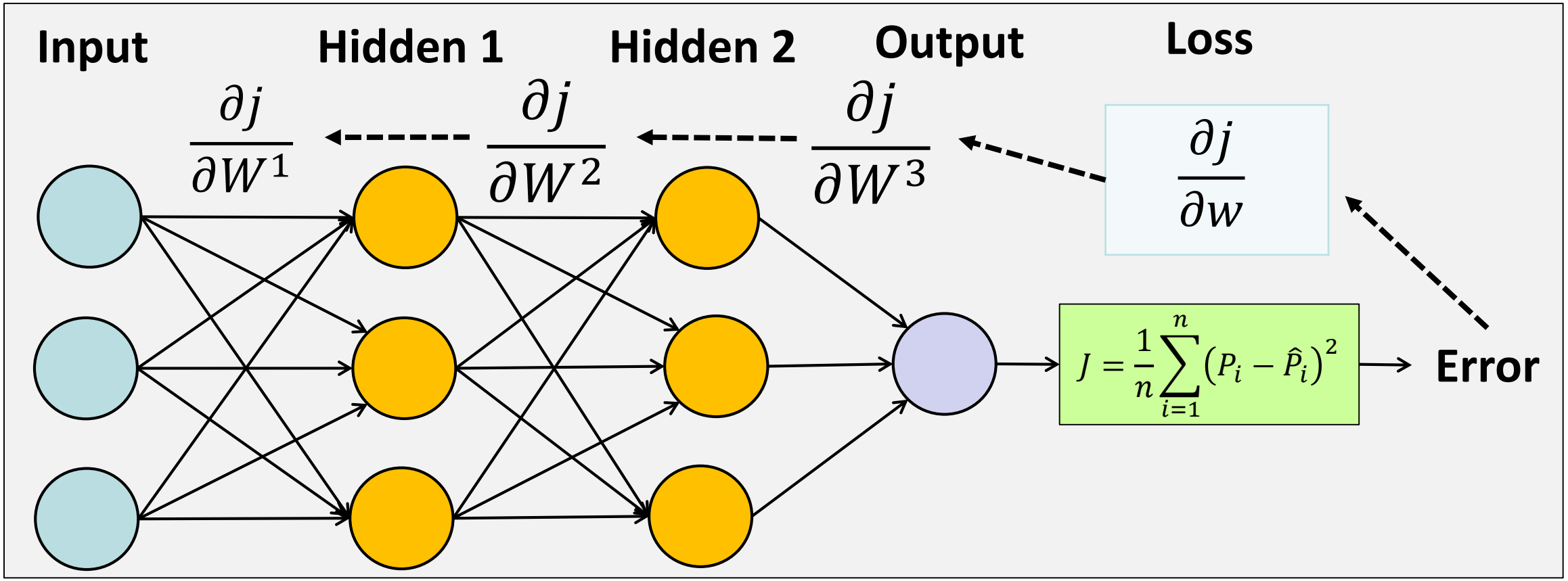
- **Learning in Deep Neural Networks**
  - The loss function provides a feedback signal to adjust the weights by a small amount, in a particular direction that will lower the score
  - This adjustment is performed by an *optimiser*, which implements the **Backpropagation algorithm**
  - **Error attribution**: figuring out how much each weight contributed to the final error by propagating the error back through the network



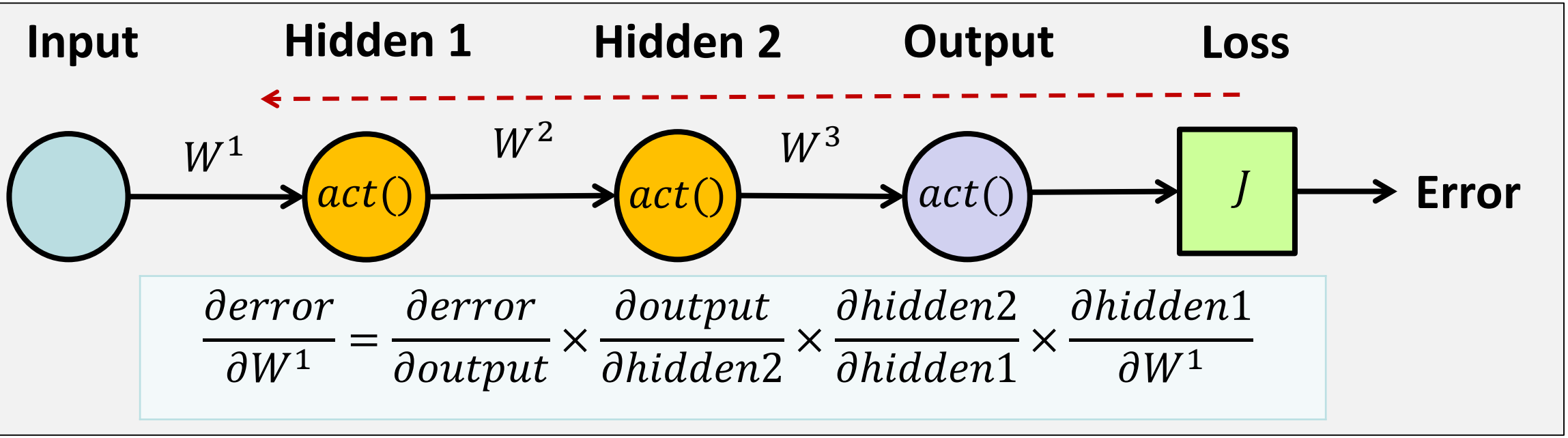
- **Perform Gradient descent**
  - Output value effected by weights at all layers



- **Backpropagation**
  - Tool to calculate the gradient of the loss function for each weight

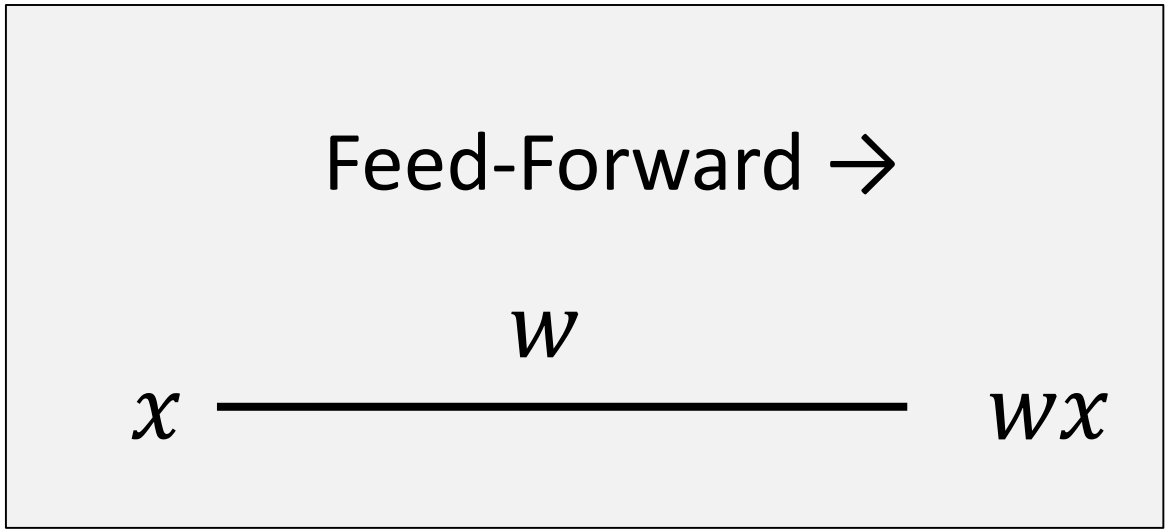


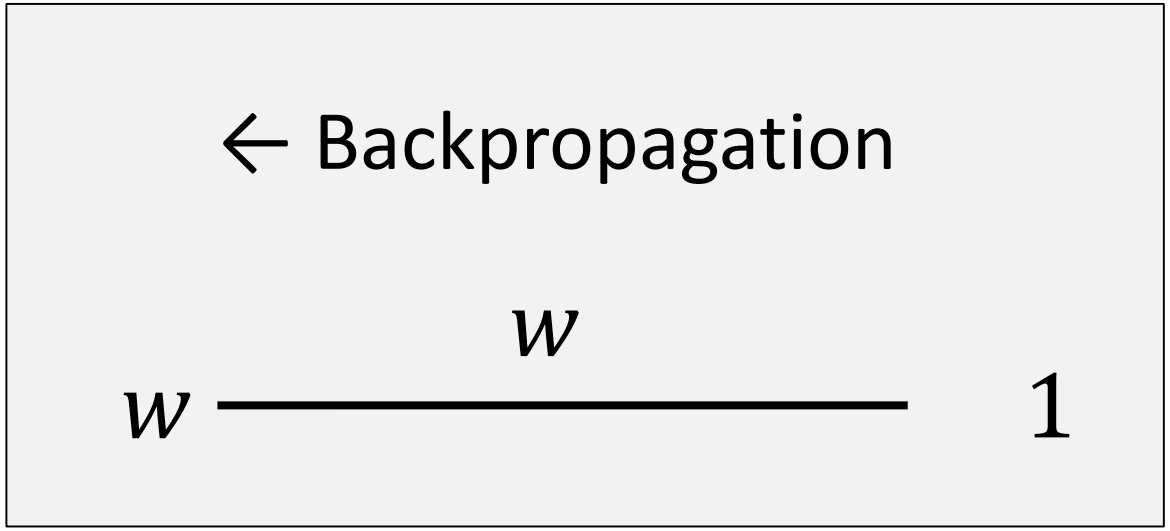
- **Calculating gradient for arbitrary weight**
  - Iteratively apply the chain rule
  - Note: Error is now a function of the output and hence a function of the input, weights, and activation functions



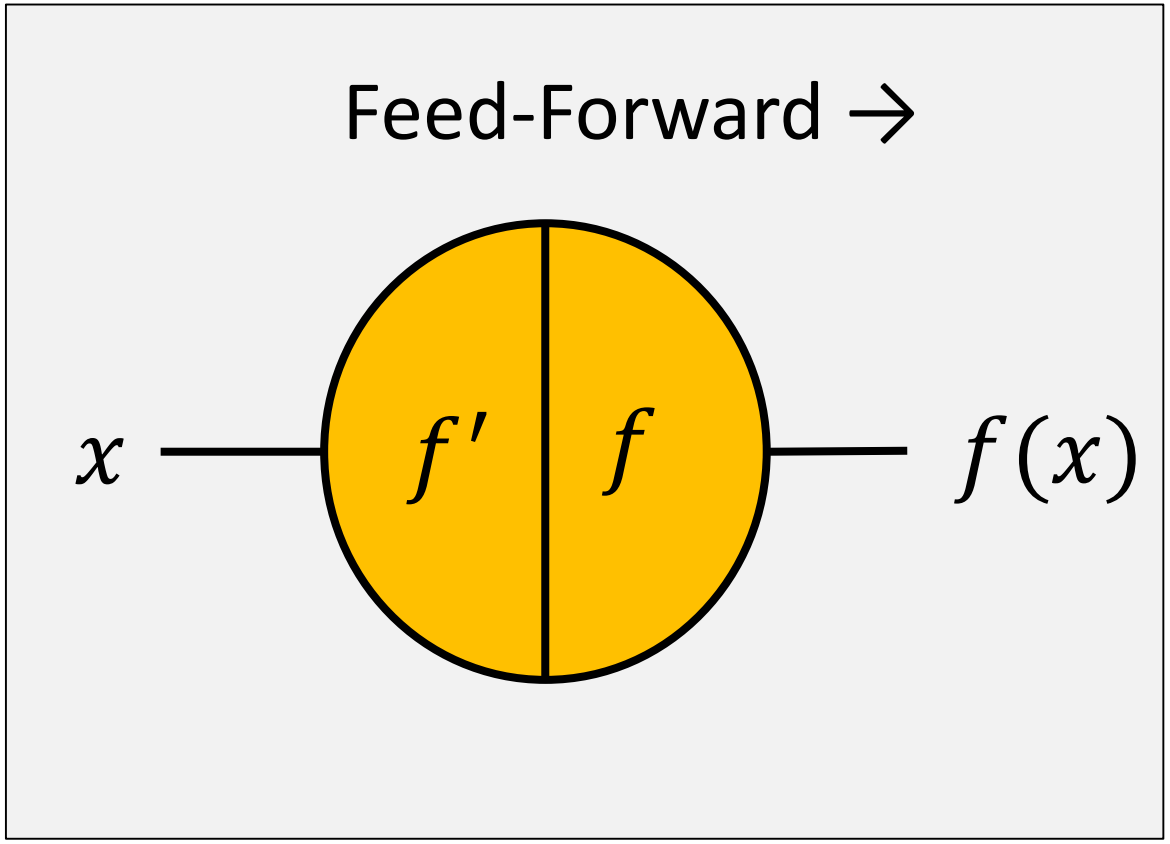
Key Backpropagation Rules

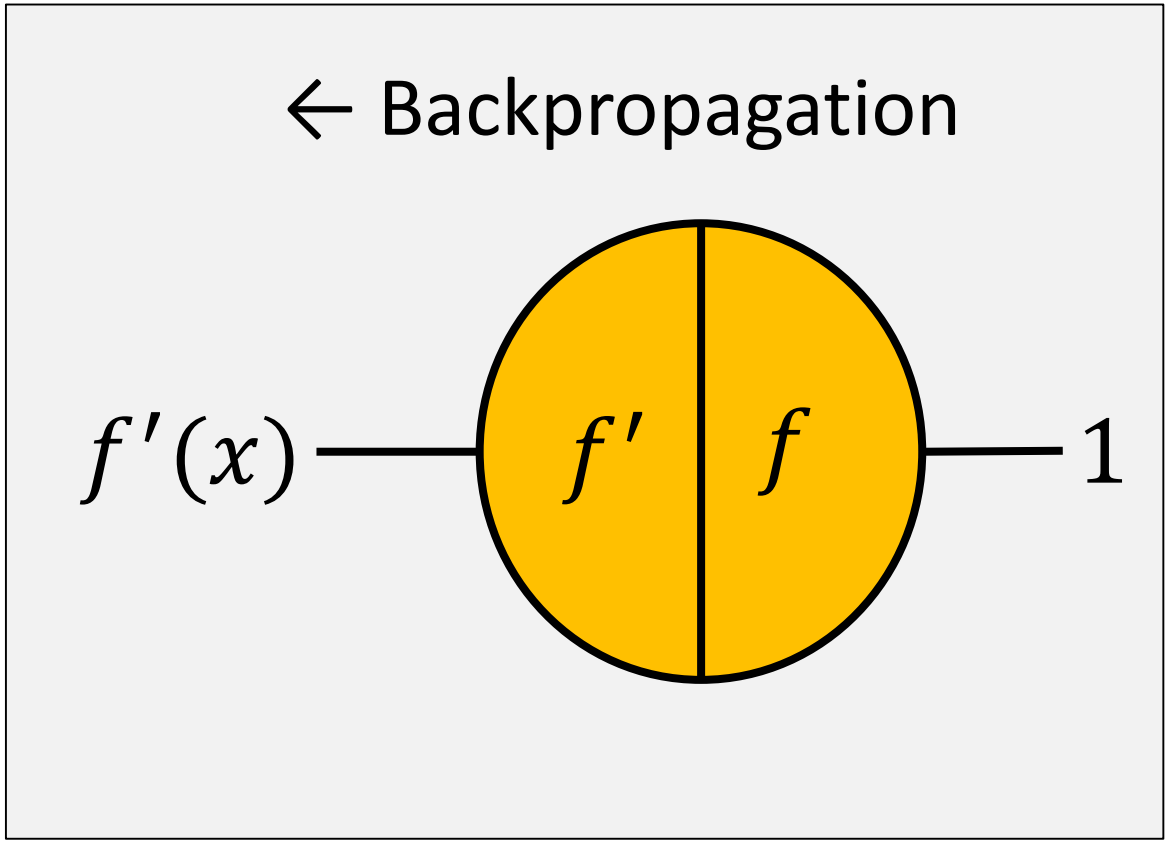
- **Weights**



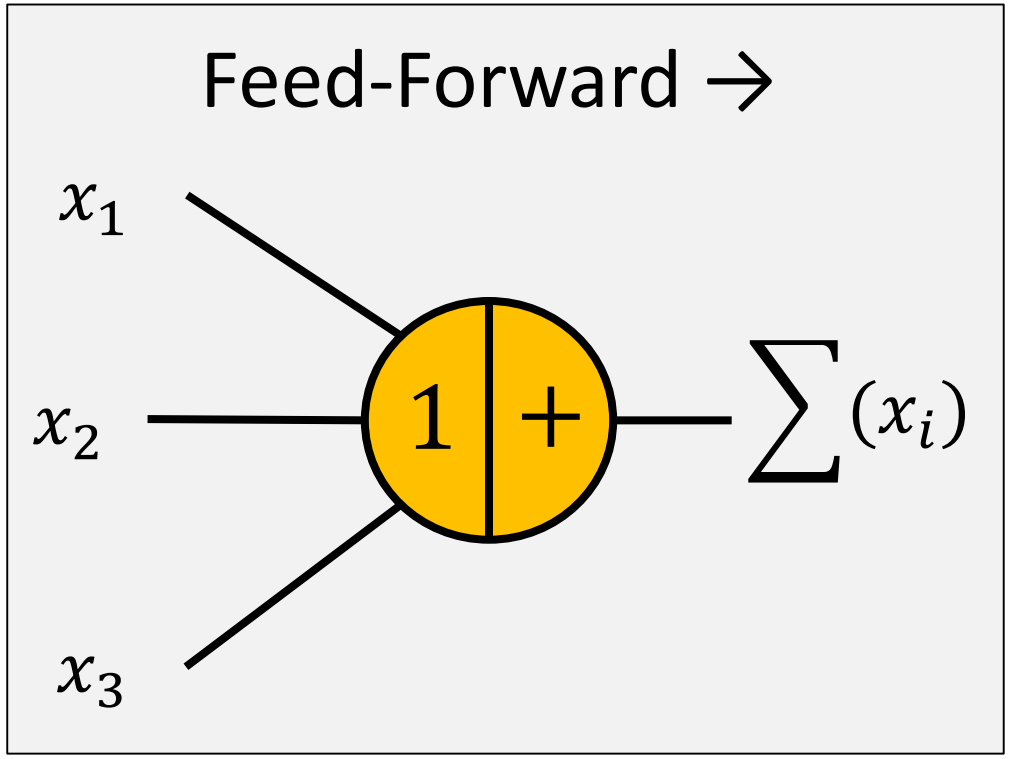


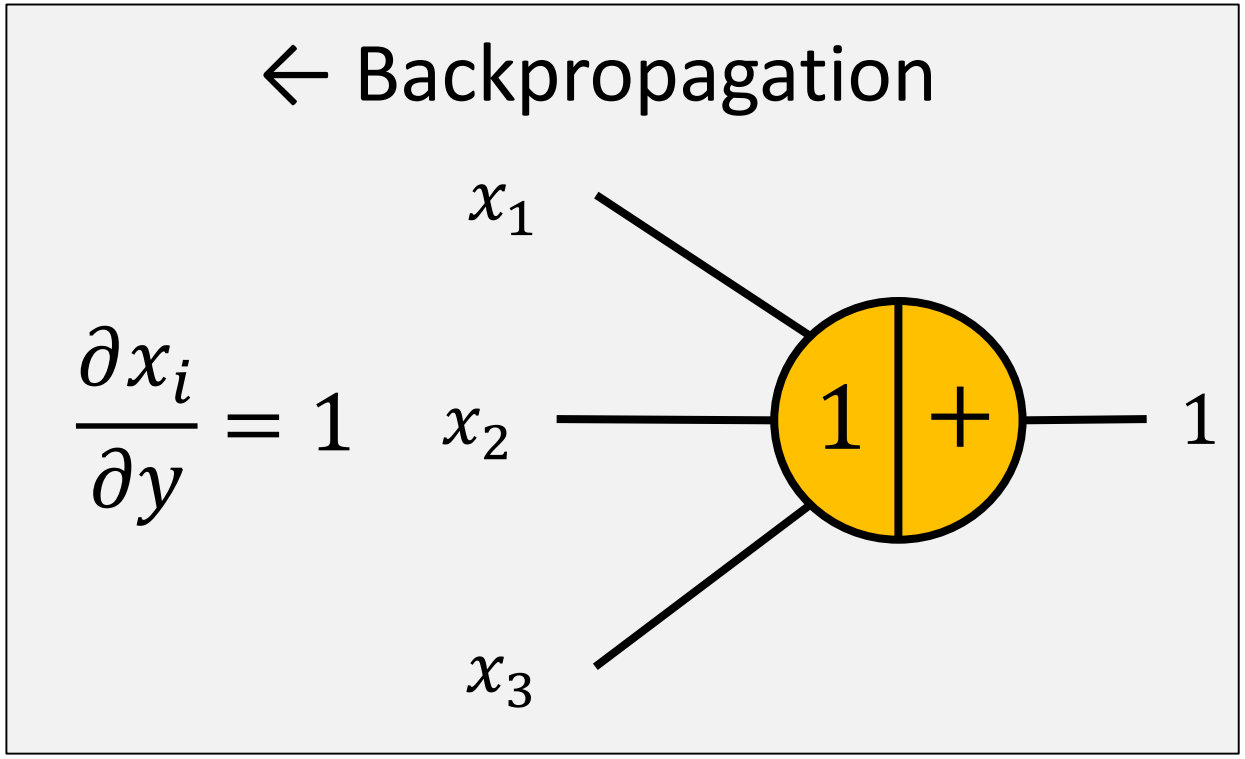
- **Activation function**



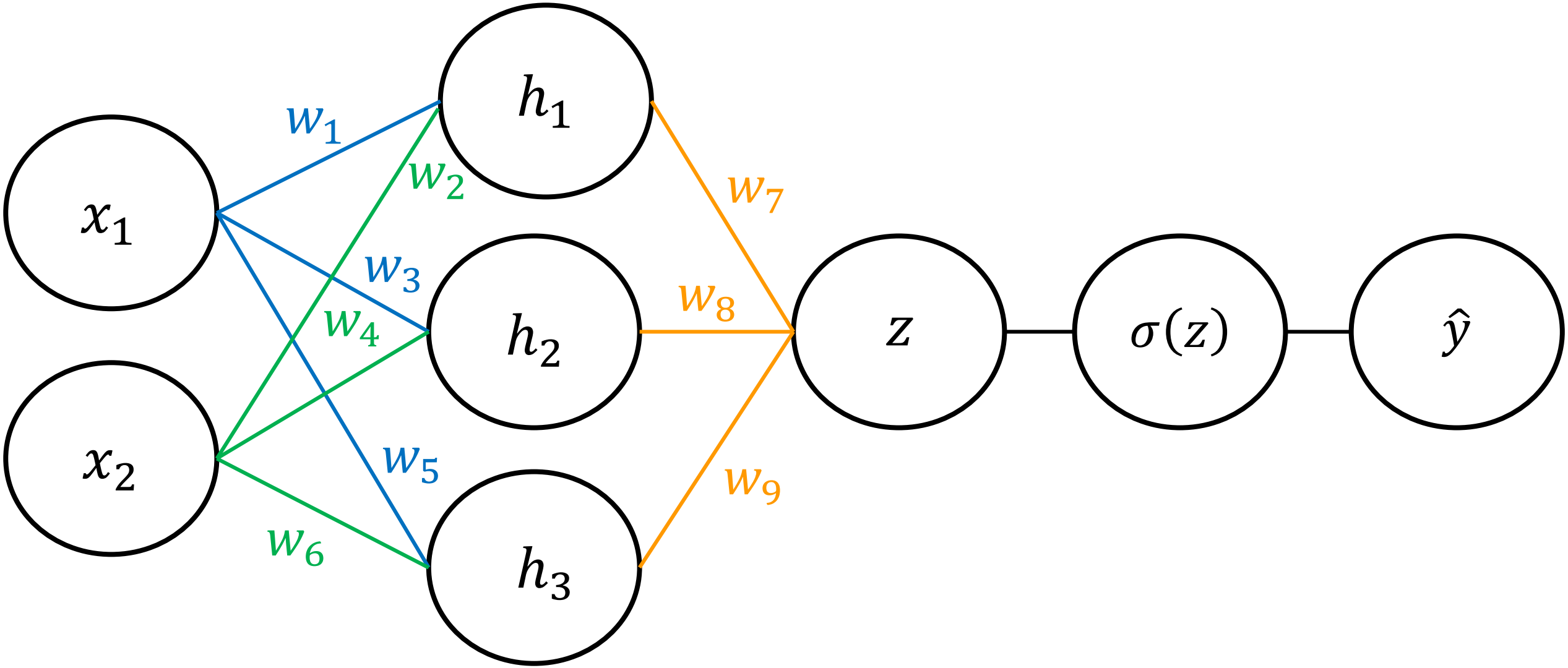


- **Summation Function**



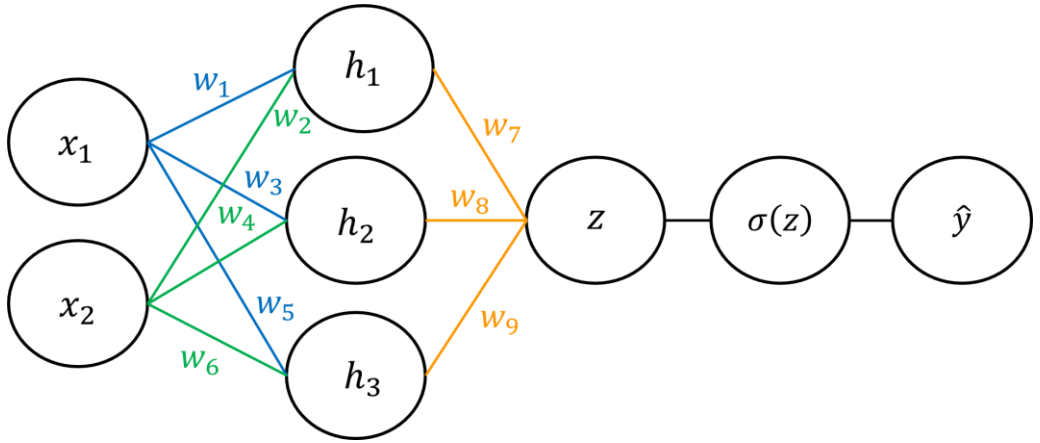


Question from lecture 12.11.19

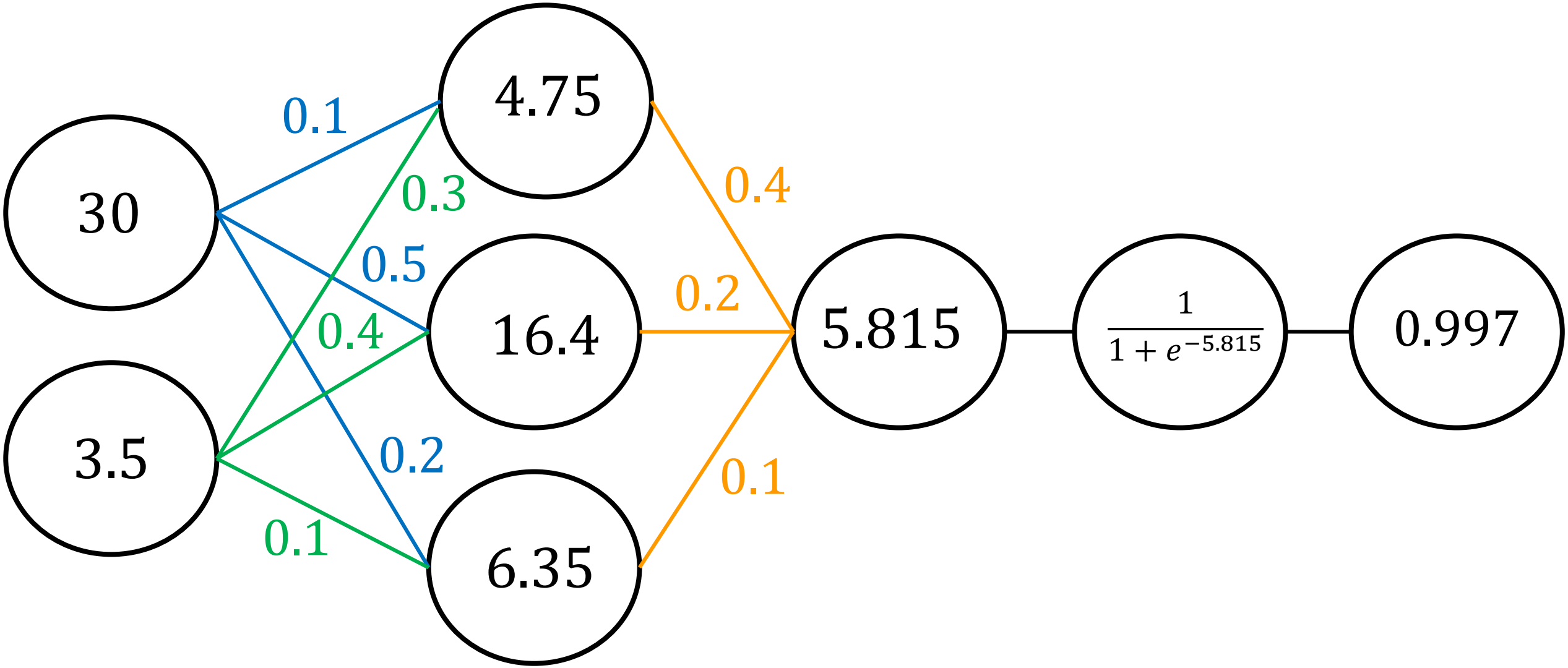


Update w_7

- $x_1 = 30, x_2 = 3.5, y = 1$
- $w_1 = 0.1, w_3 = 0.5, w_5 = 0.2$
- $w_2 = 0.3, w_4 = 0.4, w_6 = 0.1$
- $w_7 = 0.4, w_8 = 0.2, w_9 = 0.1$
- h_1, h_2, h_3 - all linear activation
- $\sigma(z)$ - sigmoid activation
- $L = \frac{1}{2}(y - \hat{y})^2, \frac{dL}{d\hat{y}} = \hat{y} - \mathrm{y}$

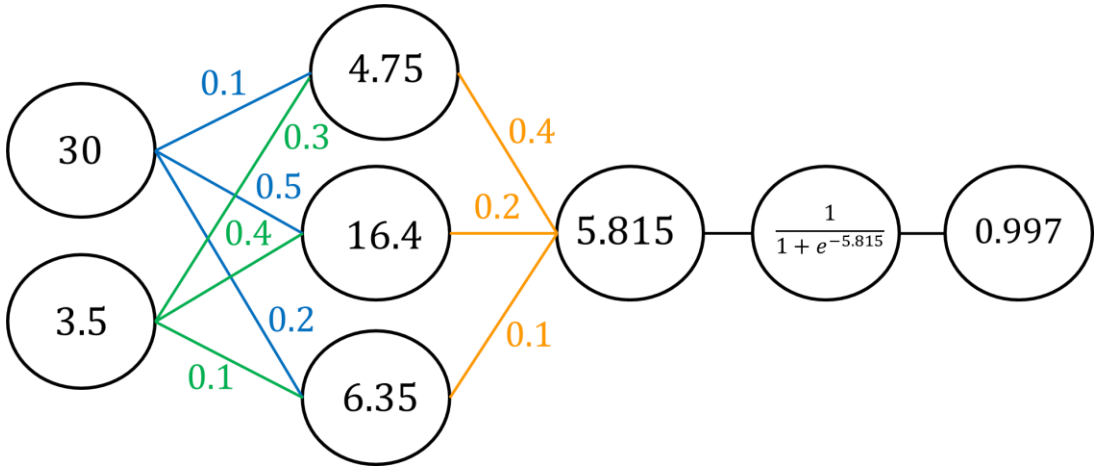


Question:


30
3.5
4.75
16.4
6.35
5.815
$\frac{1}{1+e^{-5.815}}$
0.997
0.1
0.3
0.5
0.4
0.2
0.4
0.2
0.1
0.1

Update w_7

$$\frac{\partial L}{\partial w_7} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{dz}{dw_7}$$



$$\frac{dL}{d\hat{y}} = \hat{y} - \mathrm{y} = 0.997 - 1 = -0.003$$

$$\frac{\partial \hat{y}}{\partial z} = \sigma(z) \times \big(1 - \sigma(z)\big) = \hat{y} \times (1 - \hat{y}) = 0.002991$$

$$\frac{dz}{dw_7} = \frac{d(h_1 w_7 + h_2 w_8 + h_3 w_9)}{dw_7} = h_1 = 4.75$$

Universität Augsburg University

Update w_7

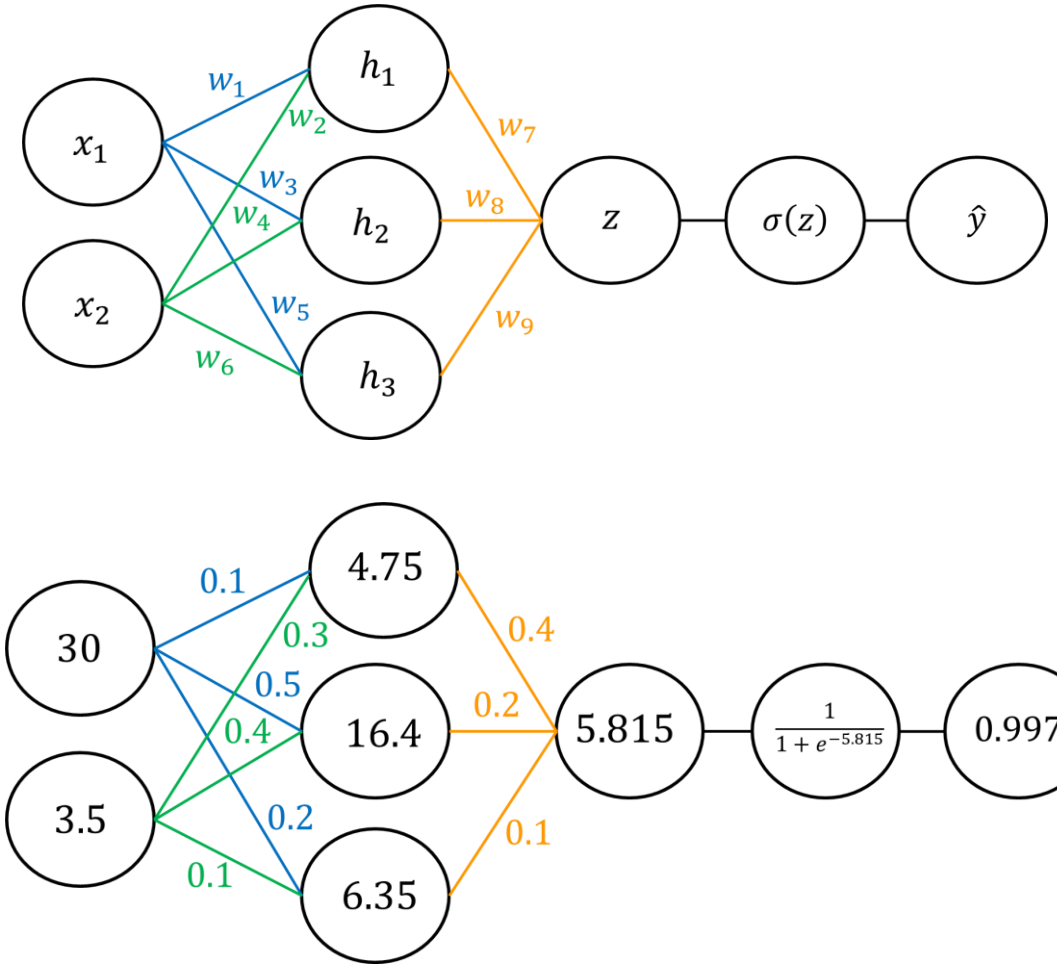
$$\frac{\partial L}{\partial w_7} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{dz}{dw_7}$$

$$\frac{\partial L}{\partial w_7} = -0.003 \times 0.002991 \times 4.75$$

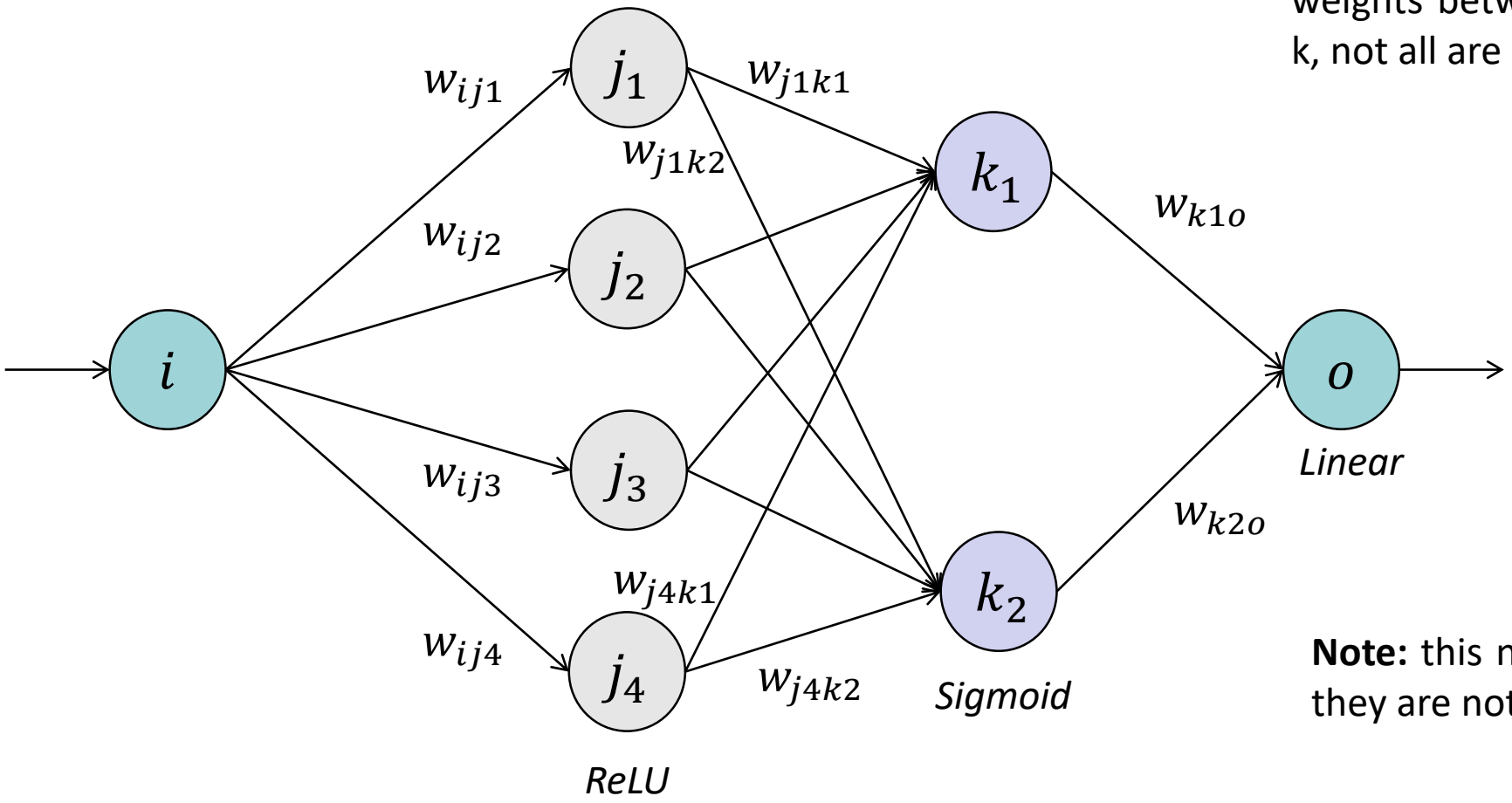
$$\frac{\partial L}{\partial w_7} = -0.0000426$$

$$w_7 = w_7 - \frac{\partial L}{\partial w_7} = 0.4 - (-0.0000426)$$

$$w_7 = 0.4000426$$



- Network Architecture



Note: there are a total of 8 weights between layers j and k, not all are illustrated

Note: this network has biases, they are not illustrated

- Initial Values

$$input = [2.0]; output = [3.0]$$

$$W_{ij} = \begin{bmatrix} w_{ij_1} & w_{ij_2} & w_{ij_3} & w_{ij_4} \end{bmatrix} = \begin{bmatrix} 0.25 & 0.5 & 0.75 & 1.0 \end{bmatrix}$$

$$W_{jk} = \begin{bmatrix} w_{j_1k_1} & w_{j_1k_2} \\ w_{j_2k_1} & w_{j_2k_2} \\ w_{j_3k_1} & w_{j_3k_2} \\ w_{j_4k_1} & w_{j_4k_2} \end{bmatrix} = \begin{bmatrix} 1.0 & 0 \\ 0.75 & 0.25 \\ 0.5 & 0.5 \\ 0.25 & 0.75 \end{bmatrix}$$

$$W_{ko} = \begin{bmatrix} w_{k_1o} \\ w_{k_2o} \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0.5 \end{bmatrix}$$

$$b_{ij} = \begin{bmatrix} b_{ij_1} & b_{ij_2} & b_{ij_3} & b_{ij_4} \end{bmatrix} = \begin{bmatrix} 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix}$$

$$b_{jk} = \begin{bmatrix} b_{jk_1} & b_{jk_2} \end{bmatrix} = \begin{bmatrix} 1.0 & 1.0 \end{bmatrix}$$

$$b_o = [1.0]$$

Backpropagation Example

- Update w_{k1o}, w_{k2o} and b_o
  - Learning rate 0.25

- Key Derivatives

ReLU

$$y = max(0, x)$$

$$\frac{\partial y}{\partial x} = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Sigmoid

$$y = \frac{1}{1 + e^{-x}}$$

$$\frac{\partial y}{\partial x} = \frac{1}{1 + e^{-x}} \times (1 - \frac{1}{1 + e^{-x}})$$

Linear

$$y = x$$

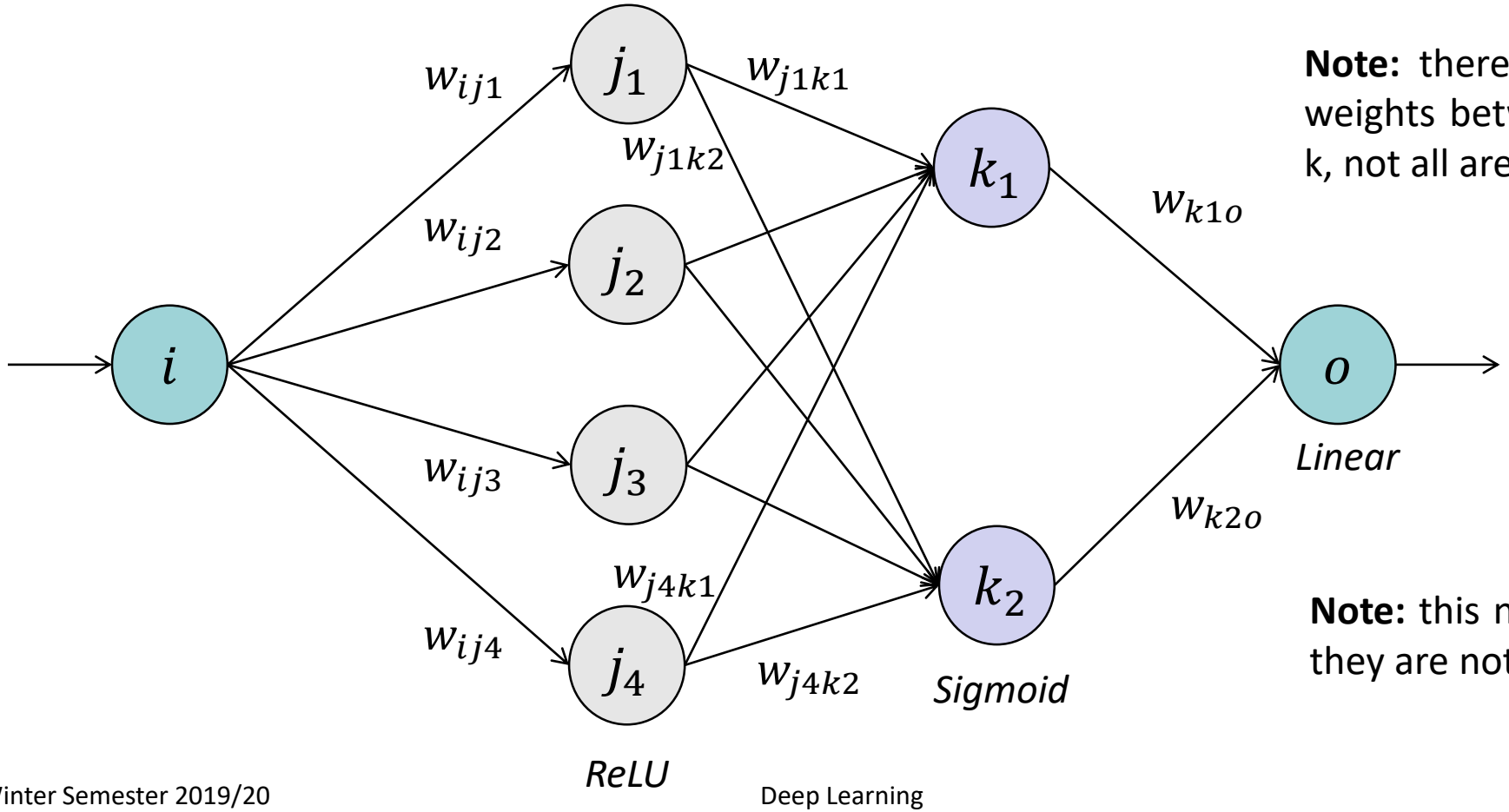
$$\frac{\partial y}{\partial x} = 1$$

Loss

$$Loss = \frac{1}{2}(output - o_{out})^2$$

$$\frac{\partial Loss}{\partial o_{out}} = o_{out} - output$$

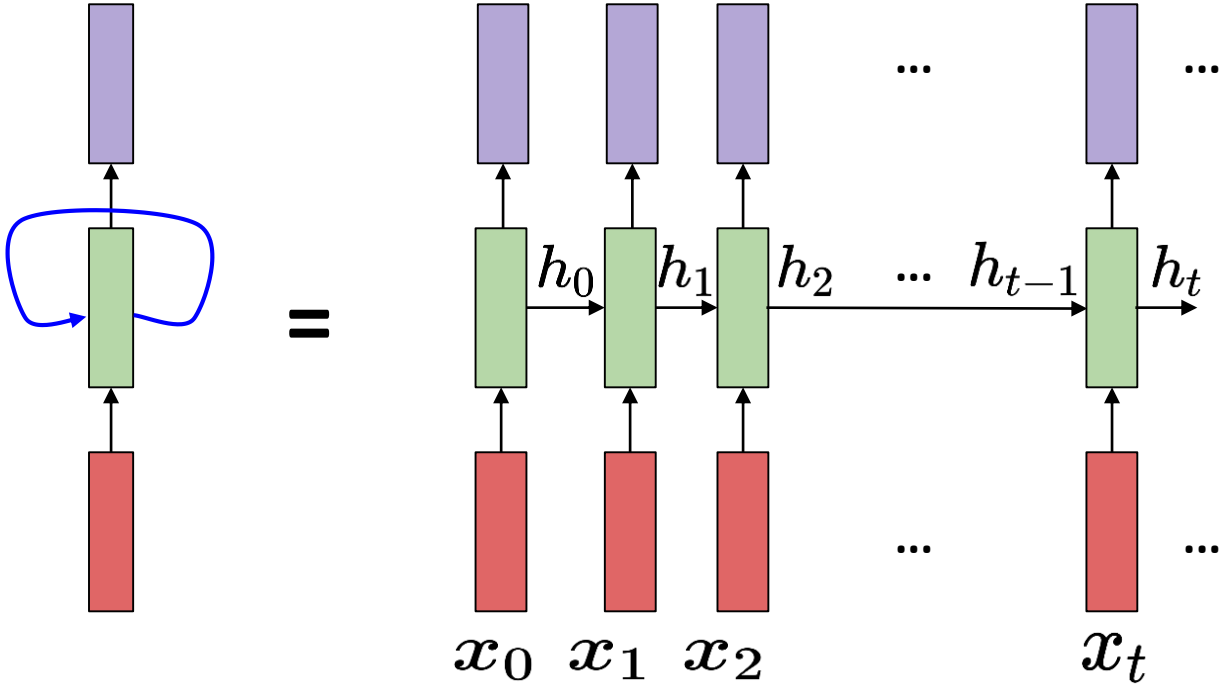
- **Homework:** update the remaining weights



Note: there are a total of 8 weights between layers j and k, not all are illustrated

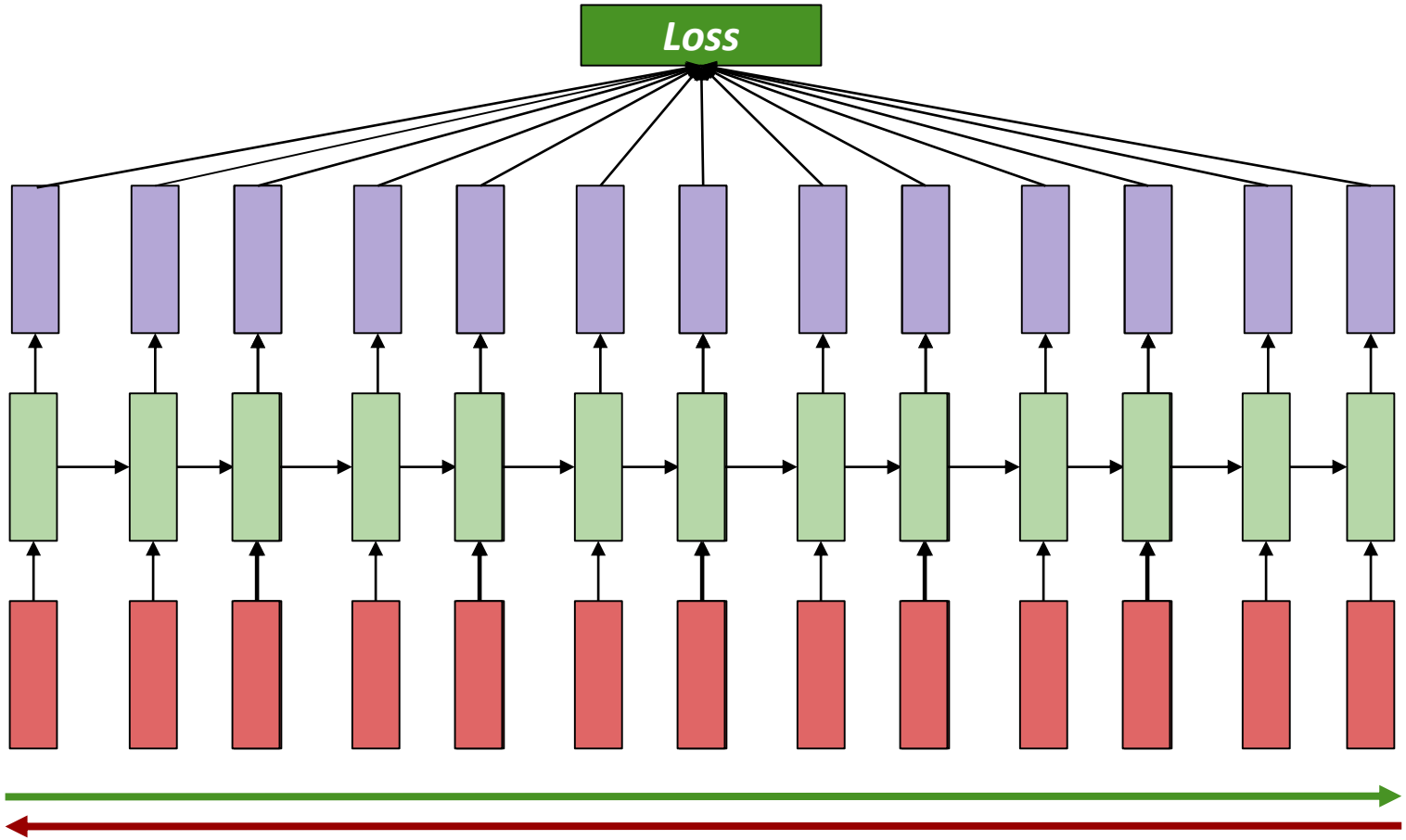
Note: this network has biases, they are not illustrated

Unrolled RNN



- Reuse the same weight matrix at every time step
- Makes the network easier to train

Backpropagation through time (BPTT)



Forward

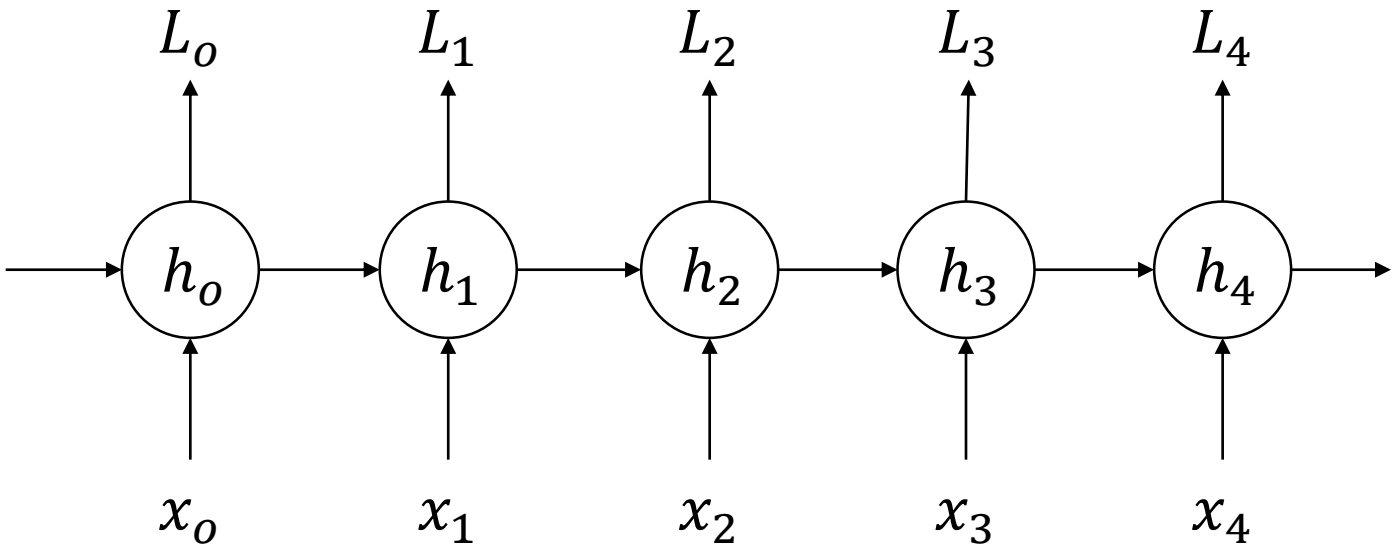
Run through entire sequence to compute the ***loss***

Backward

Run through entire sequence to compute the **gradient** and update the weight matrix:

$$w \leftarrow w - \eta \partial L / \partial w$$

- Vanishing and Exploding Gradients

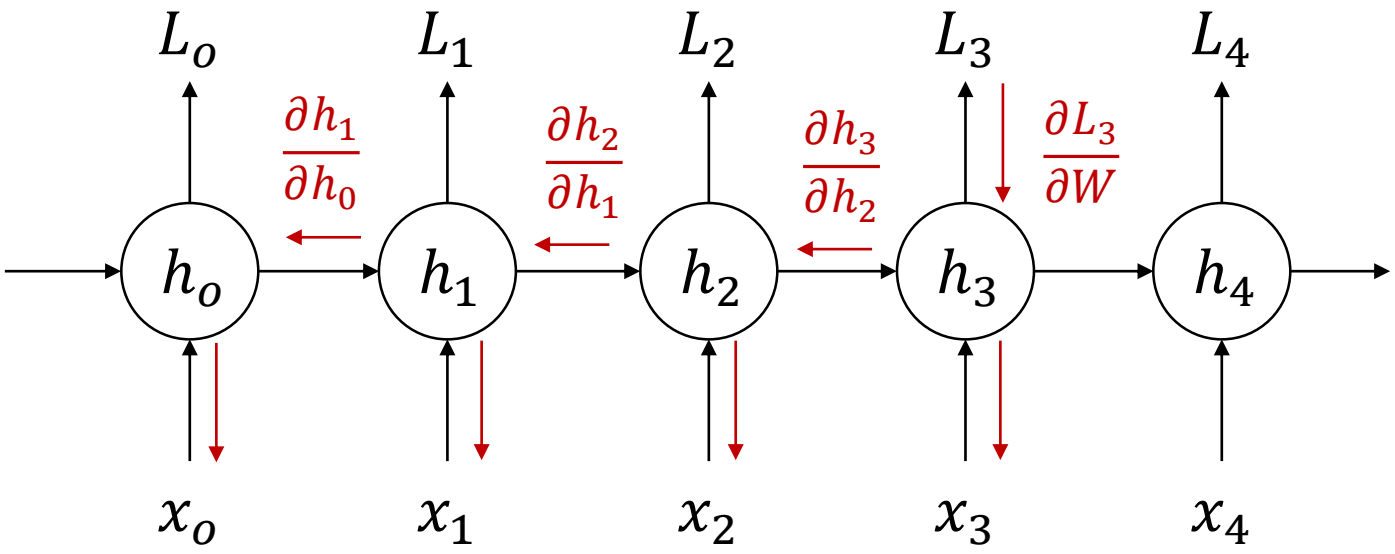


Need to sum up gradients at each time step

$$\frac{\partial L}{\partial W} = \sum_t \frac{\partial L_t}{\partial W}$$

Simple RNN – Backpropagation

- Vanishing and Exploding Gradients



$\hat{y}_j$ is the prediction

For Example:

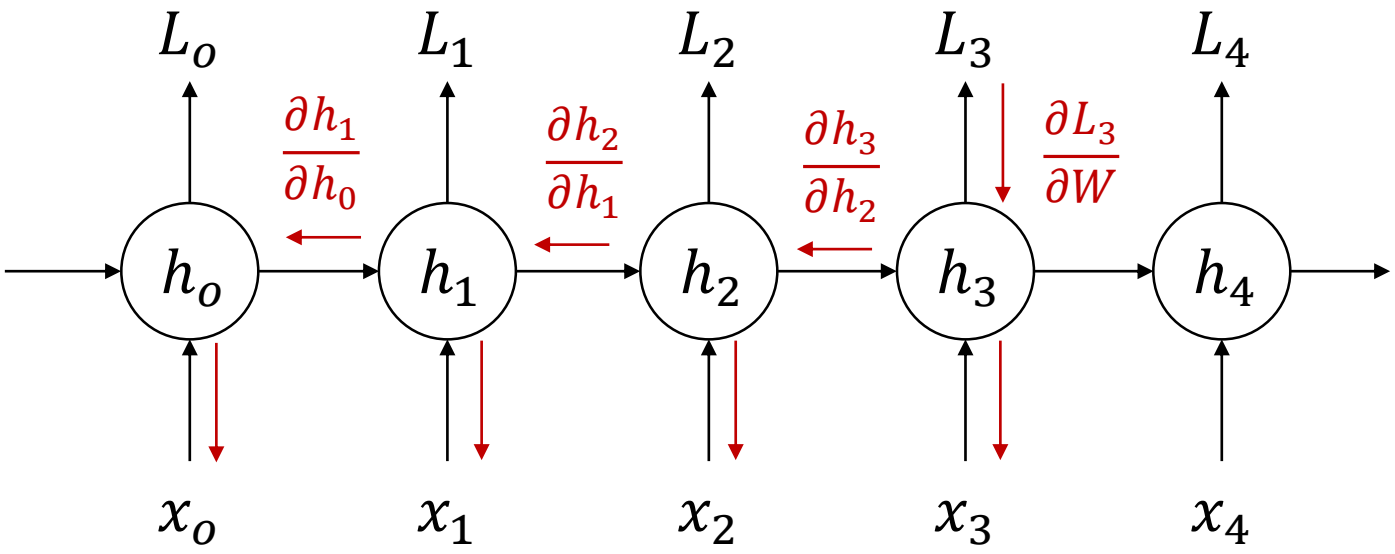
$$\frac{\partial L_3}{\partial W} = \sum_t \frac{\partial L_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial h_3} \frac{\partial h_3}{\partial h_k} \frac{\partial h_k}{\partial W}$$

Note that:

$\frac{\partial h_3}{\partial h_k}$ is found via the chain rule

E.g. $\frac{\partial h_3}{\partial h_1} = \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1}$

- Vanishing and Exploding Gradients



Therefore:

$$\frac{\partial L_3}{\partial W} = \sum_t \frac{\partial L_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial h_3} \left(\prod_{j=k+1}^{3} \frac{\partial s_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W}$$

- These repeated multiplication are the cause of exploding or vanishing gradients in RNNs
- Use gated RNNs to avoid these errors

Generalisation

One of the greatest challenges in deep learning is convincing a neural network to generalise instead of just memorise

Goal

- Learn a *robust* predictive function $f(\cdot)$
- A mapping from the feature space $\mathcal{X}$ to the label space $\mathcal{Y}$

$$\mathcal{X} \xrightarrow{f(\cdot)} \mathcal{Y}$$

- Given a test sample (unknown label), the learnt function maps the test feature vector $\boldsymbol{x}_*$ into a specific label $\boldsymbol{y}_*$

$$\boldsymbol{y}_* = f(\boldsymbol{x}_*)$$

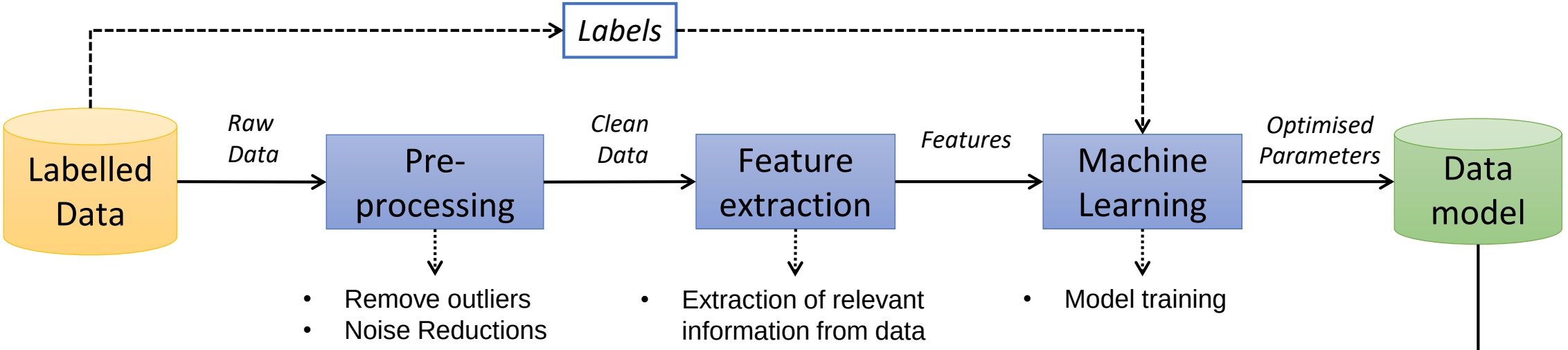
- **Generalisation of a supervised model**
  - In machine learning we want to minimise both the **training error** and the **generalisation error**
  - In deep learning we use a gradient descent algorithm to learning the network weights
  - This iterative processes reduces the training error
  - **Training error** is defined as the difference between the actual and predicted label values in the training data

- **Generalisation of a supervised model**
  - In machine learning we want to minimise both the **training error** and the **generalisation error**
  - When working with real world data it is impossible to gather all possible observations from our domain of interest
    - It may difficult or expensive to make more observations.
    - It may be challenging to gather all observations together.
    - More observations are expected to be made in the future
  - **We only ever have a finite number of training samples**
    - Introduces issues relating to the statistical concept of *sampling error*

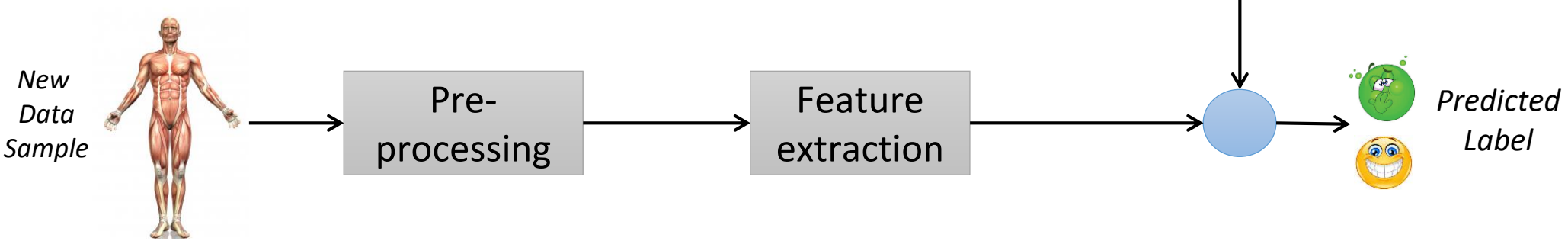
- **Generalisation of a supervised model**
  - In machine learning we want to minimise both the **training error** and the **generalisation error**
  - We only ever have a finite number of training samples
  - There is a need to ensure the **generalisability** of a model
    - The model's ability to adequately label new test data samples
    - Data **not used** during the training/optimisation phase
  - **Generalization Error**: The error on these new data instances

Training a deep neural network that can generalise well to new (test) data is a challenging problem

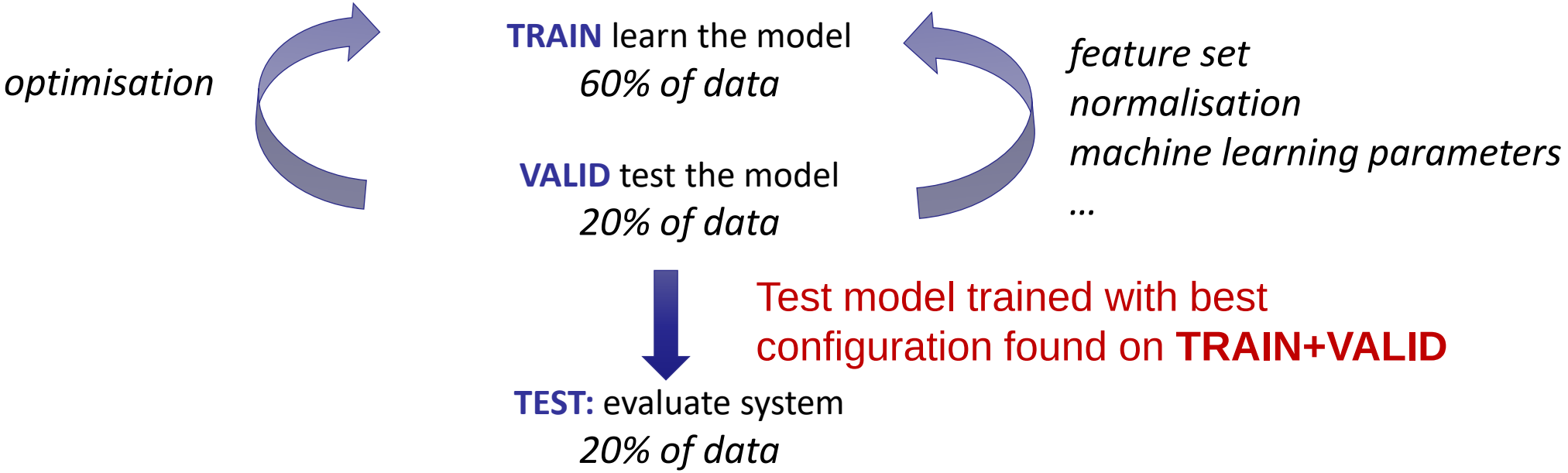
- **Training phase** – *minimise training errors*



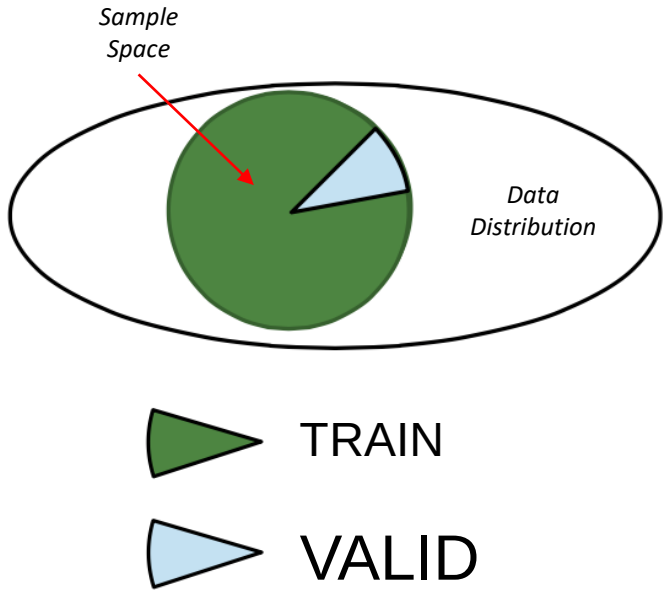
- **Test phase** – *minimise generalisation errors*



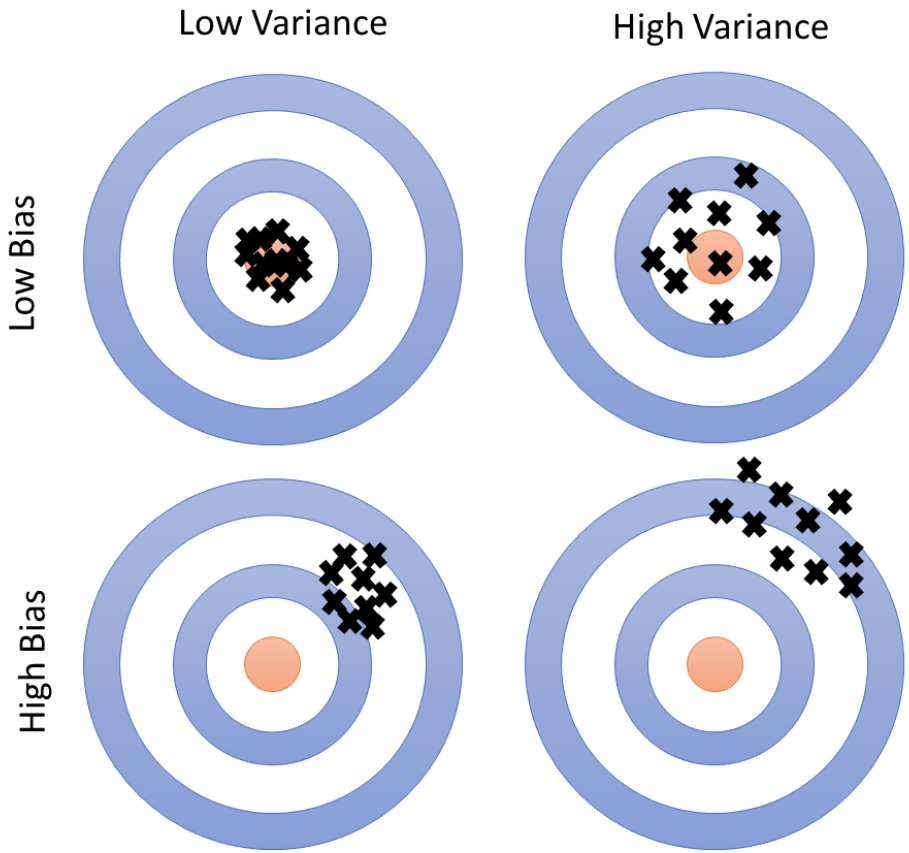
- **Data Partitioning**
  - System must generalise well to unseen data
  - Use 3 non-overlapping partitions



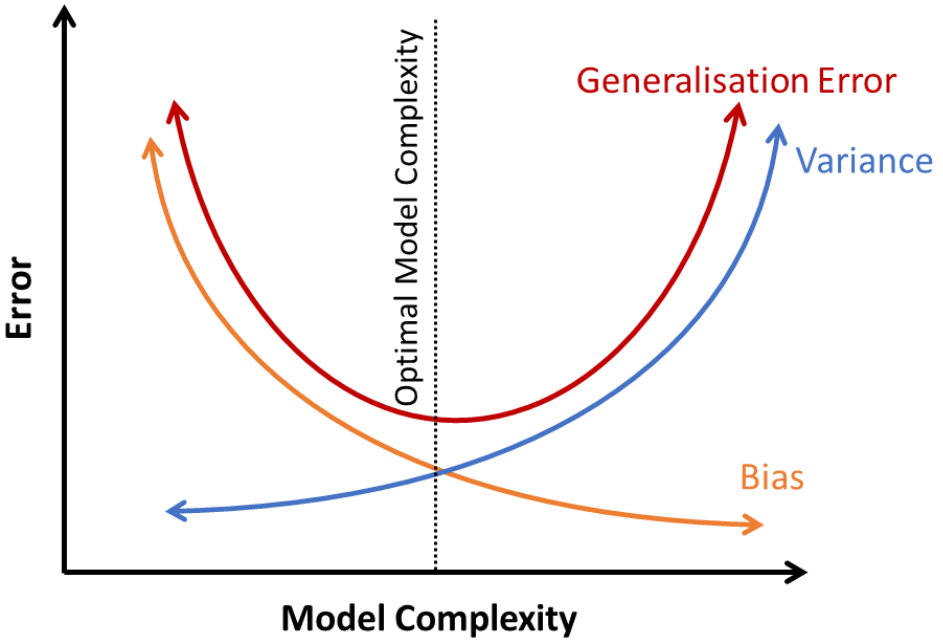
- **Data partitioning**
  - Large dataset: percentage split
    - TRAIN 60%, VALID 20%, TEST 20%
  - Small dataset: cross-validation training
    - Randomise speaker ID or instances
    - Divide dataset into *k* equal folds
    - Train on all (*k* – 1) folds and validate on fold *k*
    - Repeat the procedure *k* times to cover all the data



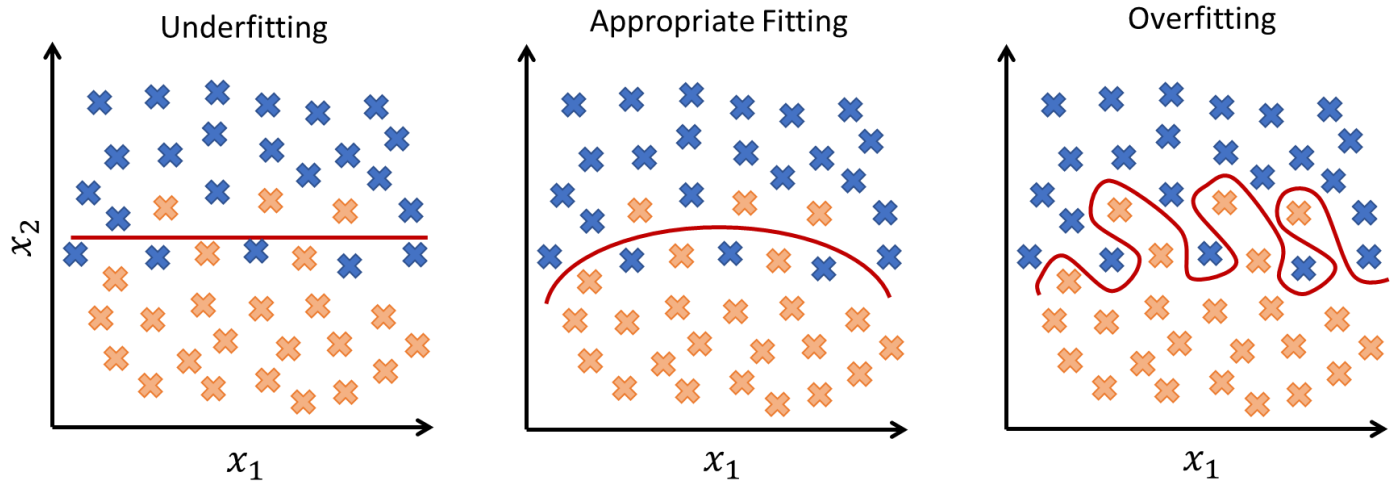
- **Bias and Variance Errors**
  - **Bias:** on average, how much are do the predicted values differ from the actual values
  - **Variance:** how different will the predictions of the model be using different samples taken from the same population



- **Minimising Generalisation Errors**
  - A trade-off of between bias and variance errors and the effect of model complexity
    - Increase in model complexity results in an initial decreases in generalisation error due to a decrease in model bias
    - As model becomes more complex generalisation errors increases due an increase in model variance

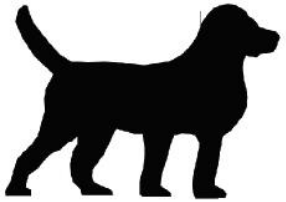


- **Generalisation Errors**
  - **Underfitting** – the model is too simple
    - The model has high bias and lacks sensitivity to the variation in data
  - **Overfitting** – the model is too complex
    - Model attempts to account for all the variation in the training data



- **Underfitting - Too little learning**
  - The model fails to sufficiently learn the problem and performs poorly on a training dataset and does not perform well on the held out test set
  - Underfitting is easy to address: Increasing model complexity
    - More complexity, means that a model can fit more types of functions for mapping inputs to outputs.
    - Increasing the complexity of a model is easily achieved by changing the structure of the model, such as adding more layers and/or more nodes to layers

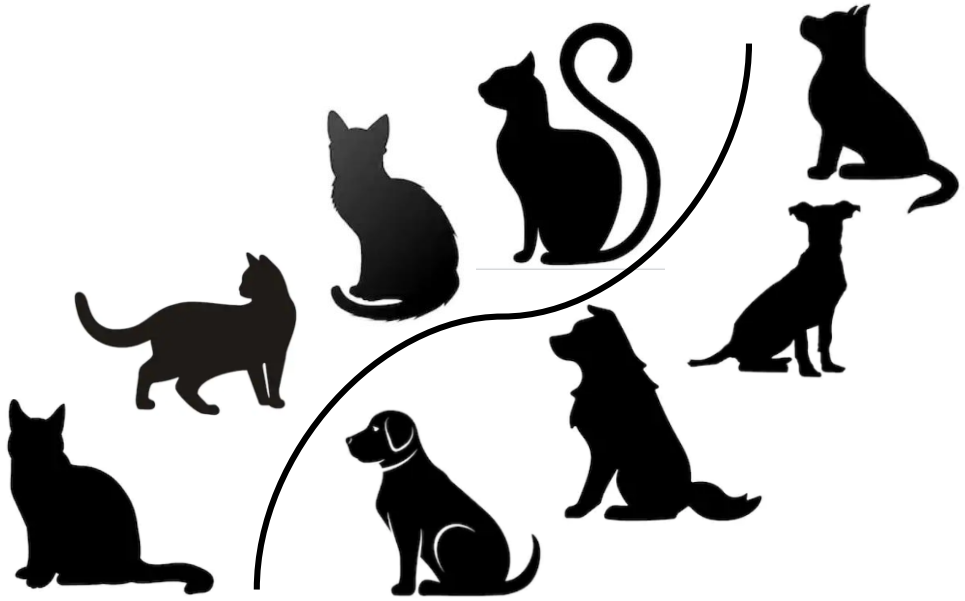
- **Overfitting - Too much learning**
  - Too much learning and the model will perform well on the training dataset and poorly on any new data
  - Detailed information is often not necessary to make generalisable predictions



Example: Everything that makes these pictures unique beyond what captures the essence of a dog can be considered *noise*.

  - **We need to get neural networks to train only on the *signal* and ignore the noise**

- **Overfitting - Too much learning**
  - Detailed information is often not necessary to make generalisable predictions
    - We want to learn the essence of a pattern, not fine details
    - The fine details may be specific, not general



- **Overfitting - Too much learning**
  - Too much learning and the model will perform well on the training dataset and poorly on any new data
  - **Why is this?**
    - Very easy for a network to learn overly detailed information about the training dataset.
    - This causes it to reject samples that are even slightly off the samples seen in the training data
  - The problem of overfitting when training neural networks and how it can be addressed with regularisation methods

Regularisation

Strategies used in machine learning explicitly designed to reduce the generalisation error, possibly at the expense of increased training error

- **What is Regularisation?**
  - A technique to overcome the problem of overfitting resulting from high model variance

- Standard Cost function:

$$C(\theta) = \|e\|_2^2 = \|X\theta - Y\|_2^2$$

- Insert regularisation term to penalizes the overall cost function in case the magnitude of the model parameter vector is high

$$L(\theta) = \|X\theta - Y\|_2^2 + \lambda\|\theta\|_2^2$$

- Taking gradient of $L(\theta)$ and setting to zero gives:

$$\hat{\theta} = (X^T X - \lambda I)^{-1} X^T Y$$

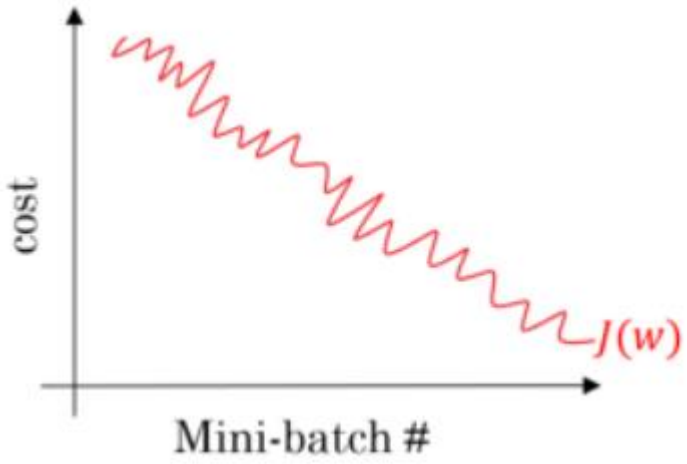
- A higher value for λ would therefore result in smaller values of $\|\theta\|_2^2$ thus making the model simpler and prone to high bias

- **Common Deep Learning regularisation strategies**
  - Mini-Batch Learning
  - Weight Penalties
  - Data Augmentation
  - Training with Noise
  - Dropout
  - Early Stopping

- **Mini-Batch Learning**
  - Performs an update for every batch of n training examples
  - Reduces noise in variance of weight updates

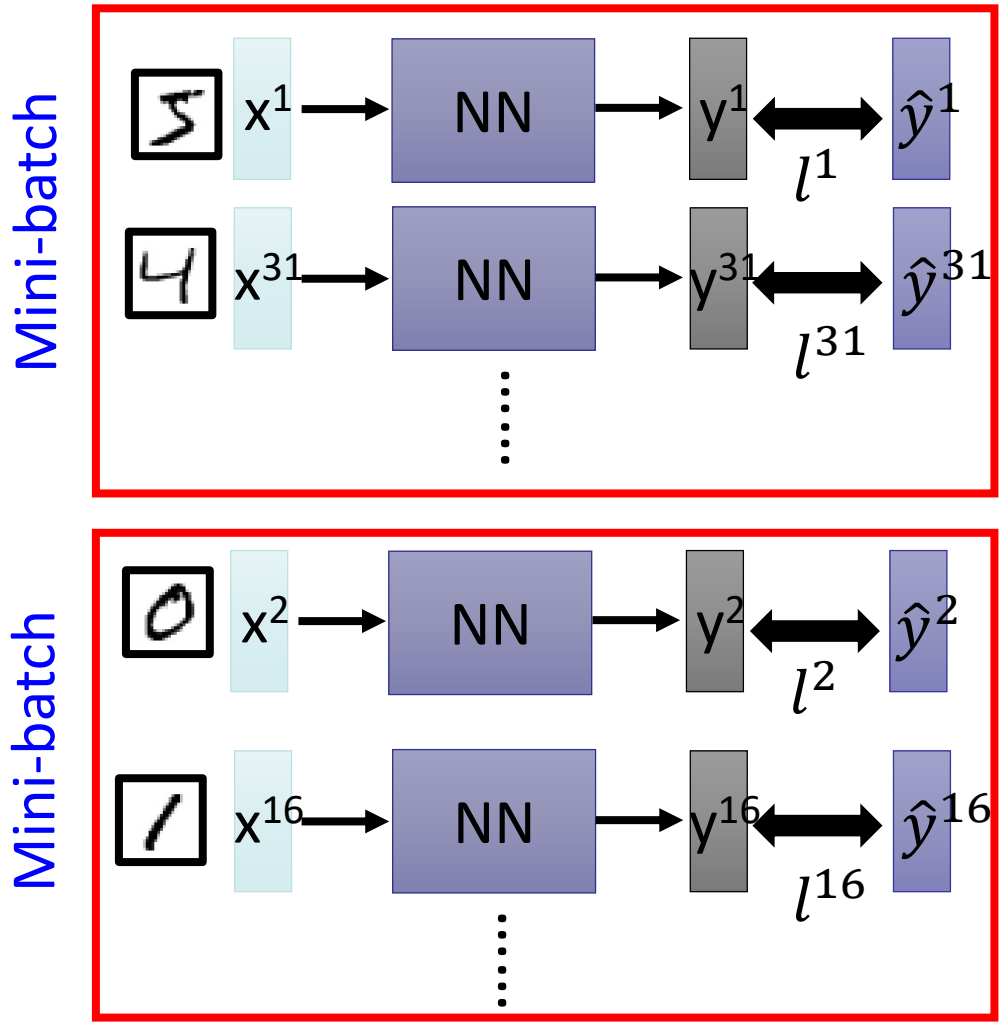
$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla J(\theta^{(t)}; x^{(i:i+n)}; y^{(i:i+n)})$$

  - Advantages
    - Stable convergence
    - Good learning speed
    - Good approximation minima location
  - Disadvantages
    - Total loss not accumulated



- **Mini-Batch Learning – Key Terminology**
  - **Batch size**
    - The number of training data points in each mini batch
      - Large enough to give good estimate of gradient
      - Small enough to provide suitable noise into updates
  - **Number of batches**
    - The total number of mini-batches in the training data
  - **Epoch**
    - One epoch consists of one full pass of training over the dataset. To
    - One epoch would consist of n number of (forward pass + backpropagation) where n denotes the number of batches.

Regularisation – Mini-Batch Learning



- Randomly initialize network parameters

- Pick the 1st batch

  $L' = l^1 + l^{31} + \cdots$

  Update parameters once

- Pick the 2nd batch

  $L'' = l^2 + l^{16} + \cdots$

  Update parameters once

  ⋮

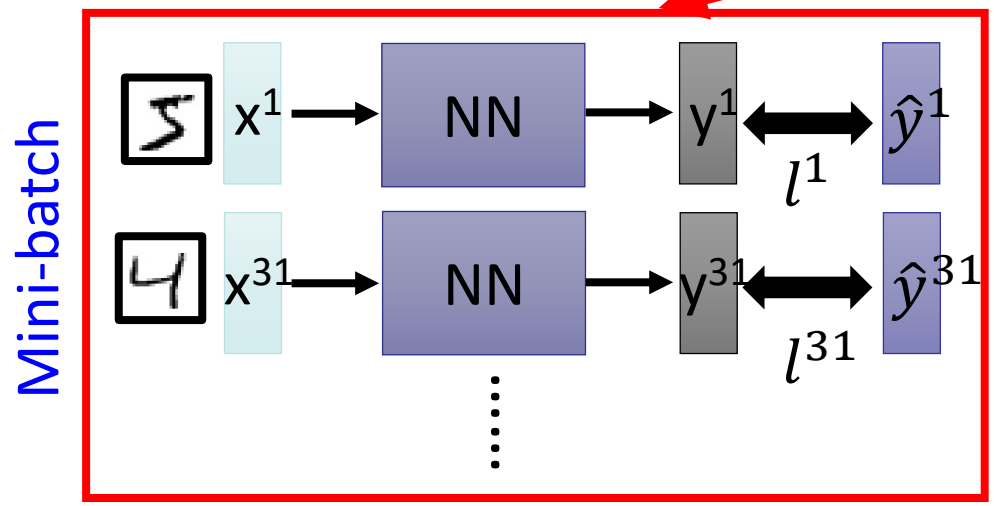
- Until all mini-batches have been picked

Repeat until Convergence

one epoch

Regularisation – Mini-Batch Learning

```
model.fit(x_train, y_train, batch_size=100, nb_epoch=20)
```



100 examples in a mini-batch

- Pick the 1st batch

  $L' = l^1 + l^{31} + \cdots$

  Update parameters once
- Pick the 2nd batch

  $L'' = l^2 + l^{16} + \cdots$

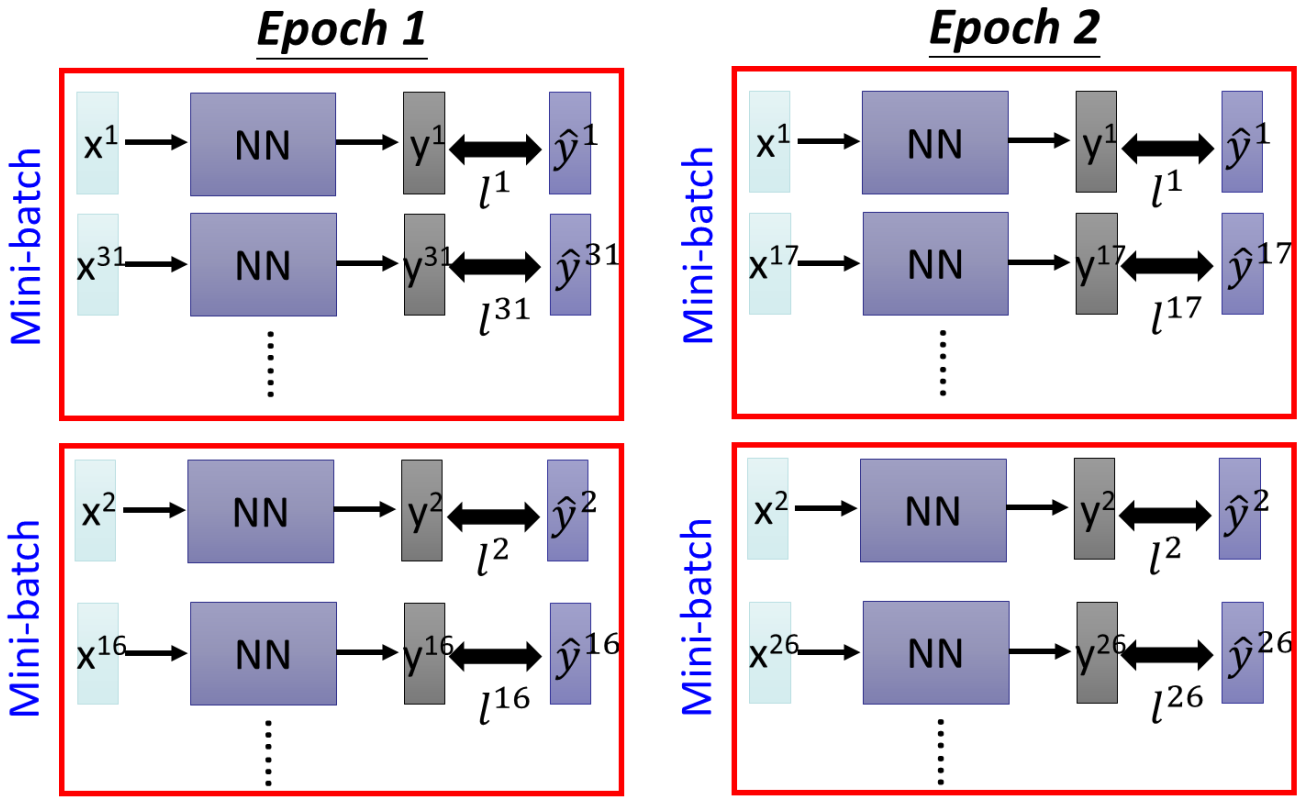
  Update parameters once

  ⋮
- Until all mini-batches have been picked

Repeat 20 times

one epoch

- **Shuffle Training Instances within epochs**
  - Help avoid gradient Descent getting stuck in local minima



- **Weight Penalties**
  - Addition of a weight penalty term to the cost function

$$\tilde{J}(\theta; x, y) = J(\theta; x, y) + \alpha\Omega(\theta)$$

  - Large weights make networks unstable
    - Minor variation or statistical noise on the expected inputs will result in large differences in the output
  - Aim of penalty term is to encourage the model to map the inputs to the outputs of the training dataset in such a way that the weights of the model are kept small

- **Common penalty terms**
  - **L1 Norm**
    - The sum of the absolute values of the weights
    - L1 encourages weights to be zero if possible
    - Resulting in more sparse weights
      - Weights with more zeros values

$$\alpha\Omega(\theta) = \|\theta\|_1 = \sum_n |\theta_n|$$

  - **L2 Norm**
    - The sum of the squared values of the weights
    - Penalizes larger weights

$$\alpha\Omega(\theta) = \frac{1}{2}\|\theta\|_2^2 = \sqrt{\sum_n |\theta_n|^2}$$

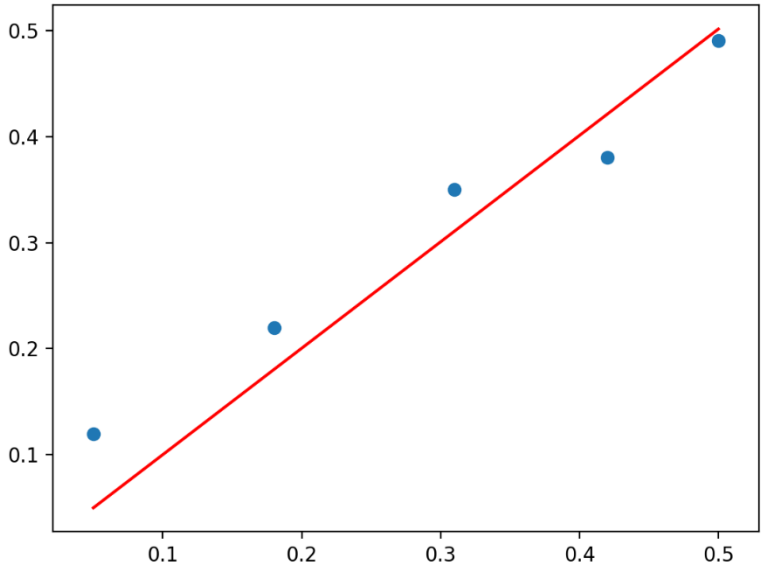
- Simple linear regression as a supervised learning technique

- Actual (observed) values $y = \theta^T X + b$
- Predicted Value: $y_p = \theta^T X + b$
- Wish to learn θ (*and* b)
- Set Cost function to square of l^2norm of error

$$C(\theta) = \|e\|_2^2$$
$$= \|X\theta - Y\|_2^2$$
$$= (X\theta - Y)^T (X\theta - Y)$$

Calculate gradient, set to 0 and solve

$$\nabla C(\theta) = 0$$
$$\hat{\theta} = (X^T X)^{-1} X^T Y$$



- **Insertion of weight penalties**

  - Standard Cost function:

$$C(\theta) = \|e\|_2^2 = \|X\theta - Y\|_2^2$$

  - Insert regularisation term to penalizes the overall cost function in case the magnitude of the model parameter vector is high

$$L(\theta) = \|X\theta - Y\|_2^2 + \lambda\|\theta\|_2^2$$

  - Taking gradient of $L(\theta)$ and setting to zero gives:

$$\hat{\theta} = (X^T X - \lambda I)^{-1} X^T Y$$

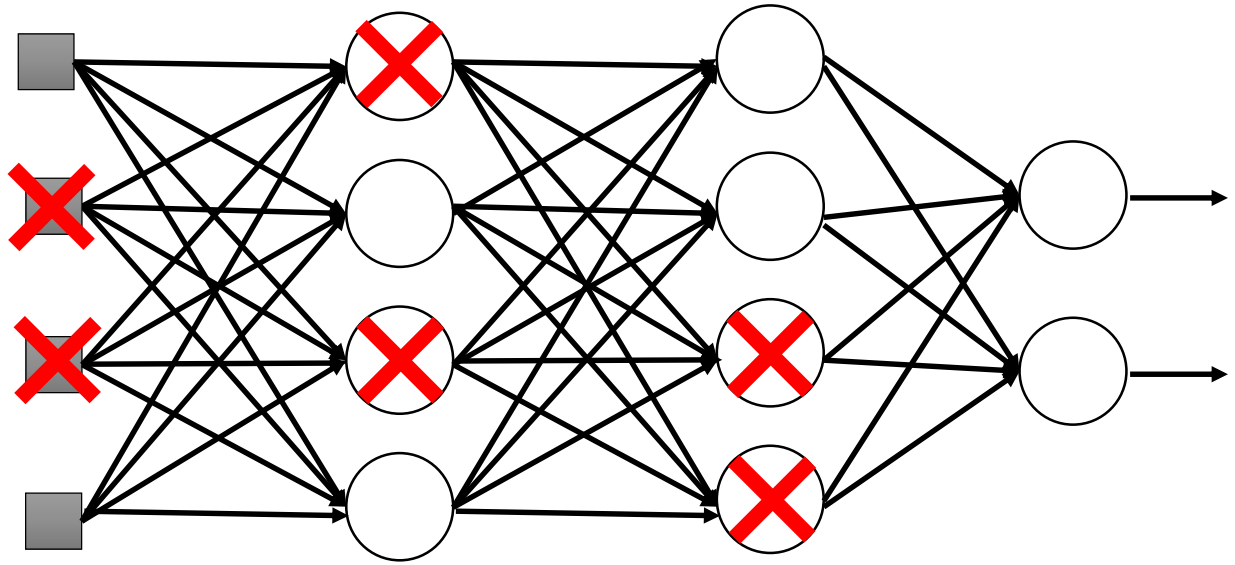
  - A higher value for λ would therefore result in smaller values of $\|\theta\|_2^2$ thus making the model simpler and prone to high bias

- **Data Augmentation**
  - The best way to make a machine learning model generalise better is to train it on more data
  - One way to get around this problem is to create fake data and add it to the training set
    - Trivial with images: shifts, flips, zooms, rotations
    - Adding noise is a form of augmentation
  - One must be careful not to apply transformations that would change the correct class.
    - Flips and rotations not useful when recognising the difference between “b” and “d” or the difference between “6” and “9”,

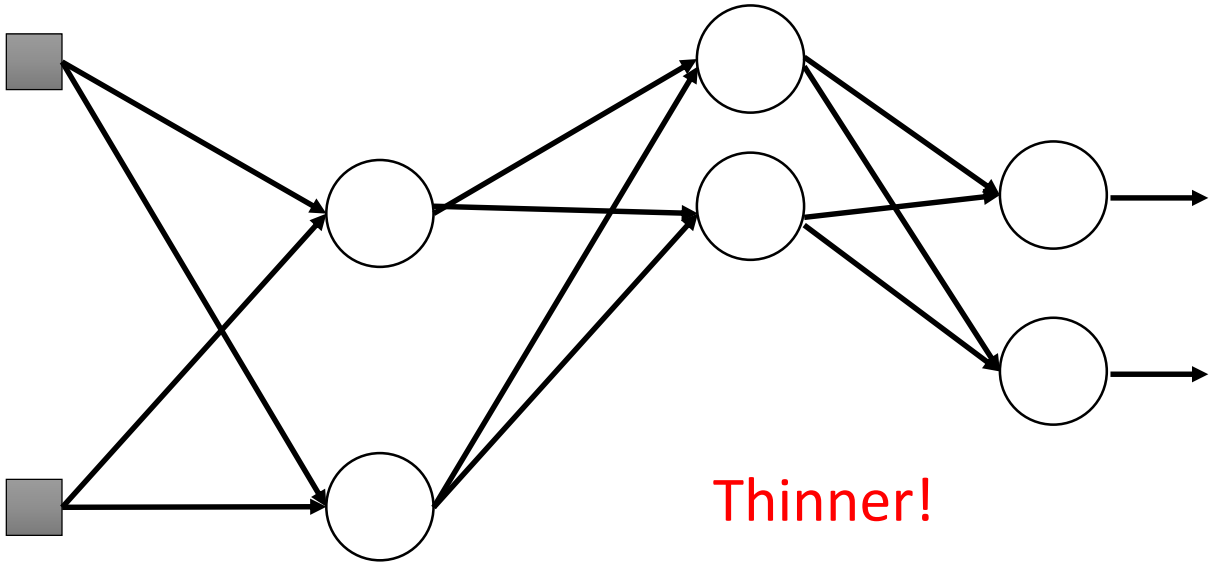
- **Training with noise**
  - **Adding noise means that the network is less able to memorise training samples**
    - Input data is changing all of the time
  - Results in smaller network weights and a more robust network that has lower generalisation error
  - Typical to add noise to input data
    - White Gaussian noise with mean of 0 and a standard deviation of 1
    - Generated as needed using a pseudorandom number generator.
    - We can also add noise activations, to weights, to the gradients and to the to the outputs

- **Dropout**
  - Method for training a ensemble of slightly different networks and averaging them
  - **Underlying concept:**
    - Although it's likely that large, unregularized neural networks will overfit to noise, it's unlikely they will overfit to the *same* noise
      - I.e. they will make slightly different mistakes
    - Averaging will cancel out the *differing* mistakes revealing what they all learned in common: *the signal* properties
  - The ensemble of subnetworks is formed by randomly removing nonoutput units during training

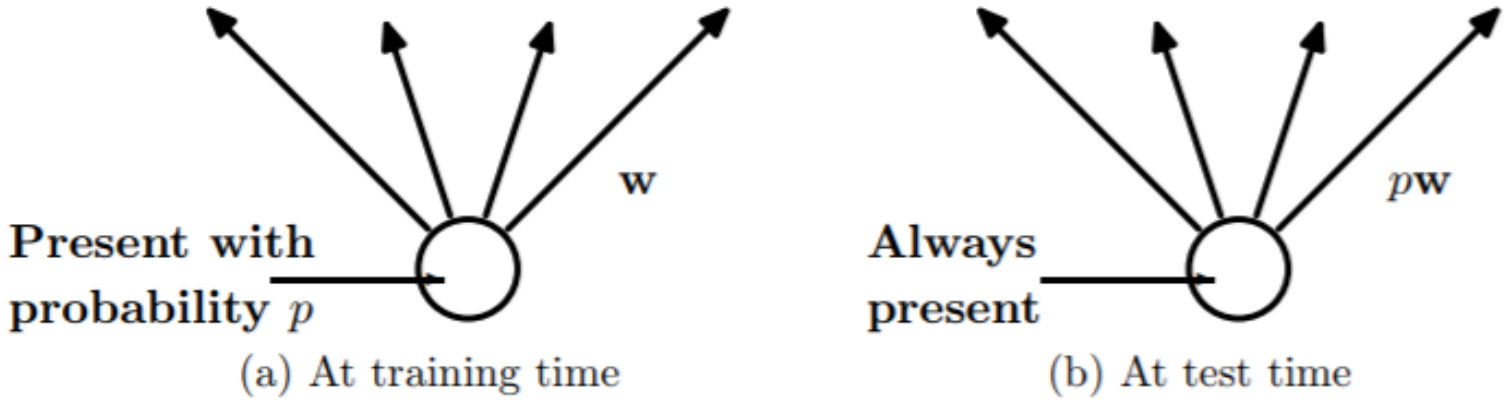
- **Training with Dropout**
  - Random neurons before updating the parameters
  - Each neuron has $p\%$ to dropout
    - p is a hyperparameter chosen before training



- **Training with Dropout**
  - **The structure of the network is changed**
    - Continue training with this new network
  - **For each mini-batch, we resample the dropout neurons**

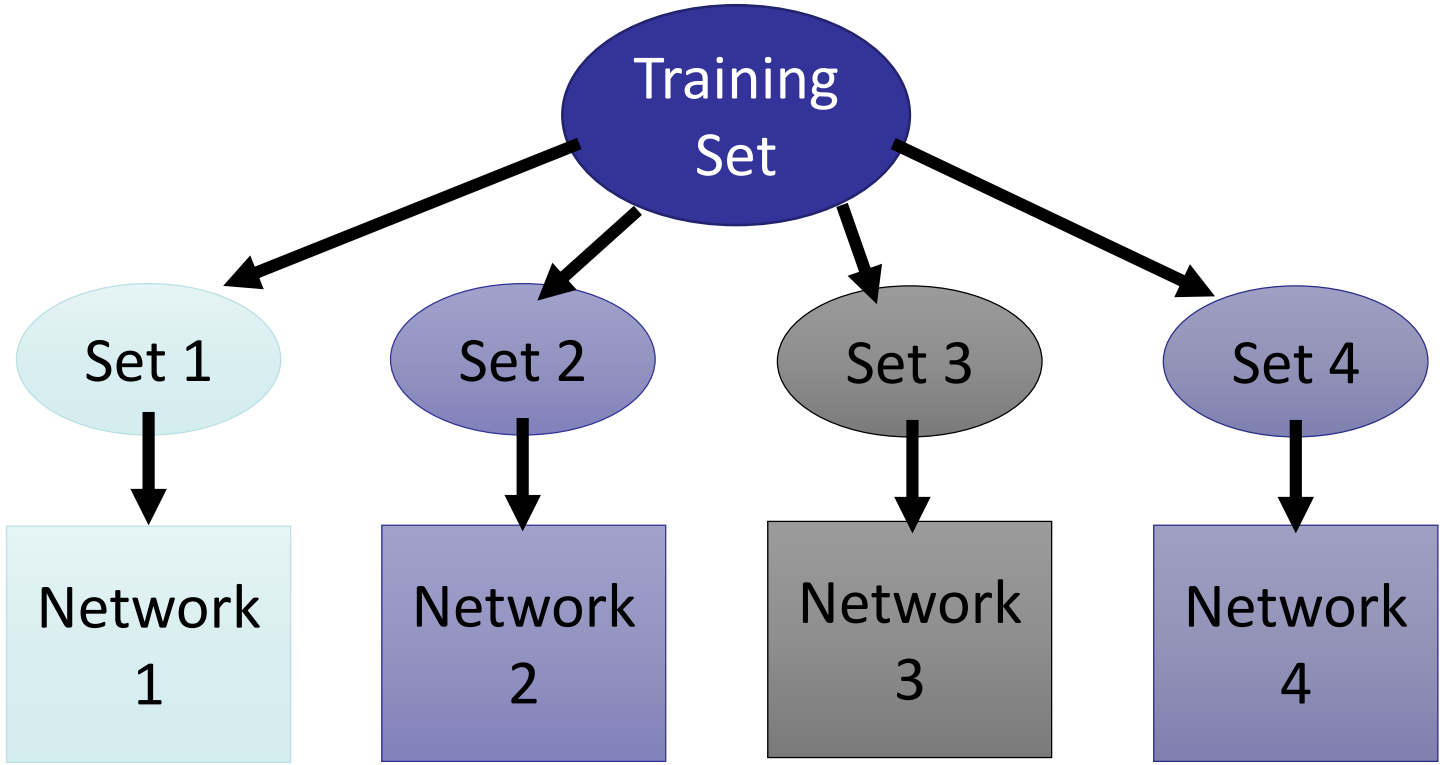


- **Testing with Dropout**
  - Use a single neural without dropout. The weights of this network are scaled-down versions of the trained weights.
  - If a neuron is retained with probability p during training, the outgoing weights are multiplied by p at test time



(a) At training time (b) At test time

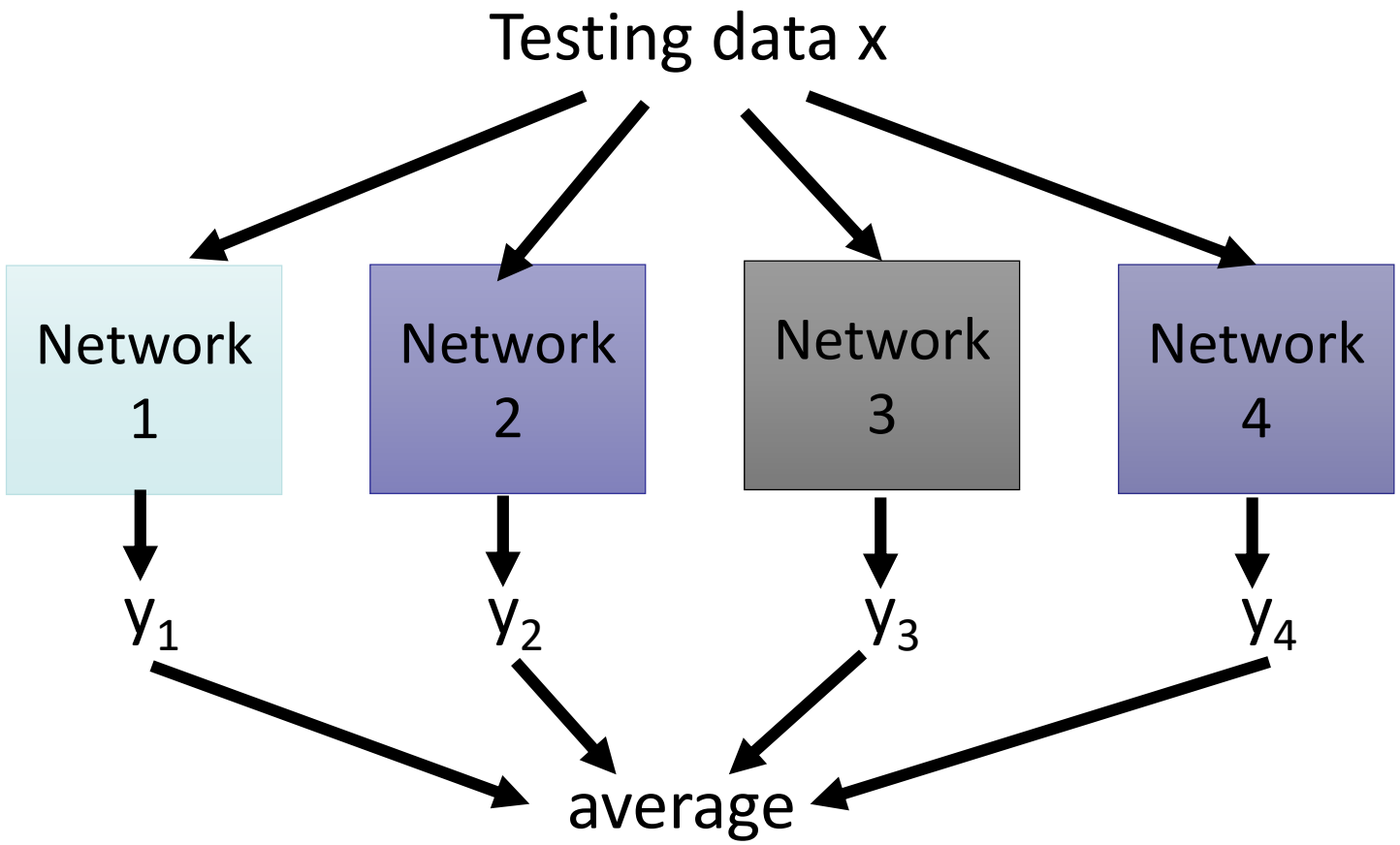
- Dropout is a form of ensemble learning



Train a bunch of networks with different structures

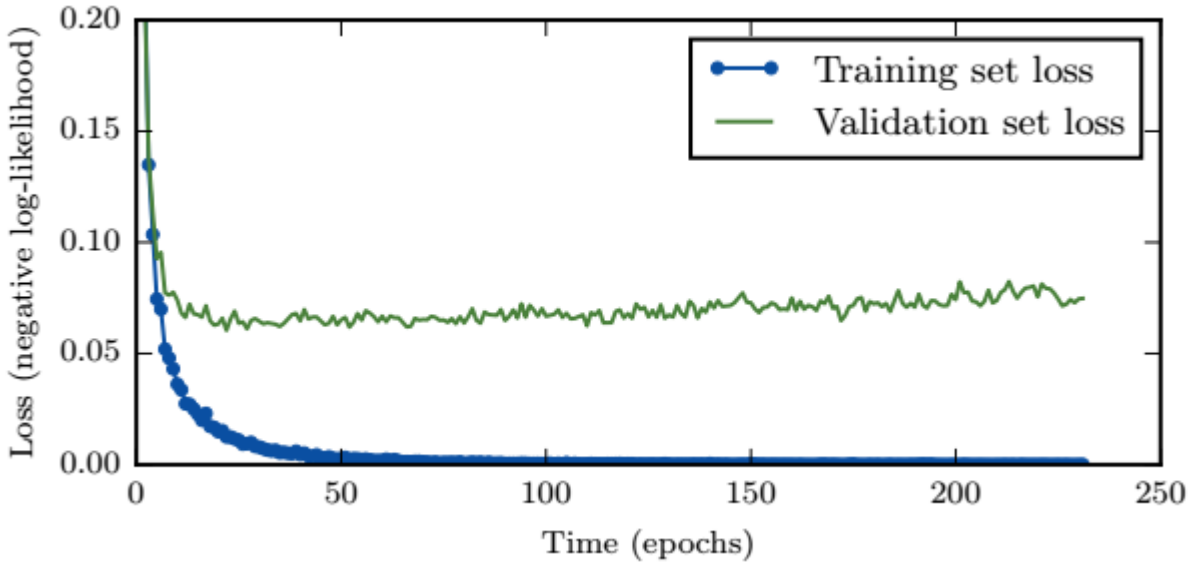
Dropout is a kind of ensemble.

- Averaging of ensemble improves generalisability



- **Early Stopping**
  - Neural networks can get worse if you train them too much
    - When training a large network, there will be a point during training when the model will stop learning the signal and start learning the statistical noise in the training dataset
  - Training a neural network long enough to learn the mapping, but not so long that it overfits the training data.
  - Trivial to monitor performance on a holdout validation dataset can be monitored during training
  - Stop training when generalization error increases

- **Early Stopping**
  - During training, the model is evaluated on a holdout validation dataset after each epoch
  - If the performance of the model on the validation dataset starts to degrade then the training process is stop



- **Regularisation**
  - Processes which adds stability to a learning algorithm by making it less sensitive to the training data and processes.
  - When using regularization, we try to reduce the generalisation error, this may lead to increase the training error which is okay because what we care about is how well the model generalizes.
  - With regularisation, we try to bring back the very complex model that suffers from overfitting to a good model by increasing bias and reducing variance.

Summary

- **Mini-Batch Learning**
  - Perform an update for every batch of n training example
- **Weight Penalties**
  - Addition of a weight penalty term to the cost function to ‘Punish’ large weights
- **Data Augmentation**
  - Create fake data and add it to the training set
- **Training with Noise**
  - Adding small amounts of noise to instances before training
- **Dropout**
  - Create ensemble of networks is formed by randomly removing units during training
- **Early Stopping**
  - Stop training when generalization error increases