



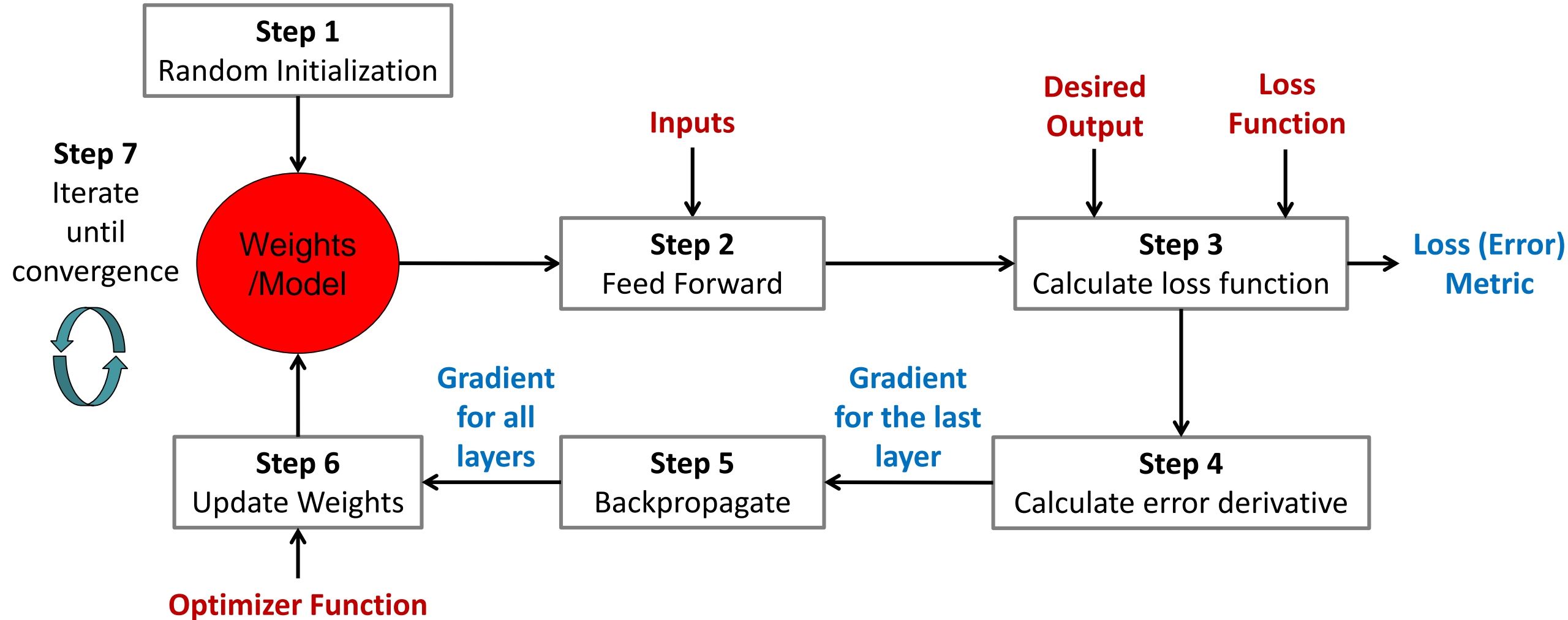
Deep Learning

Feed Forward Neural Networks

Tuesday 12th November

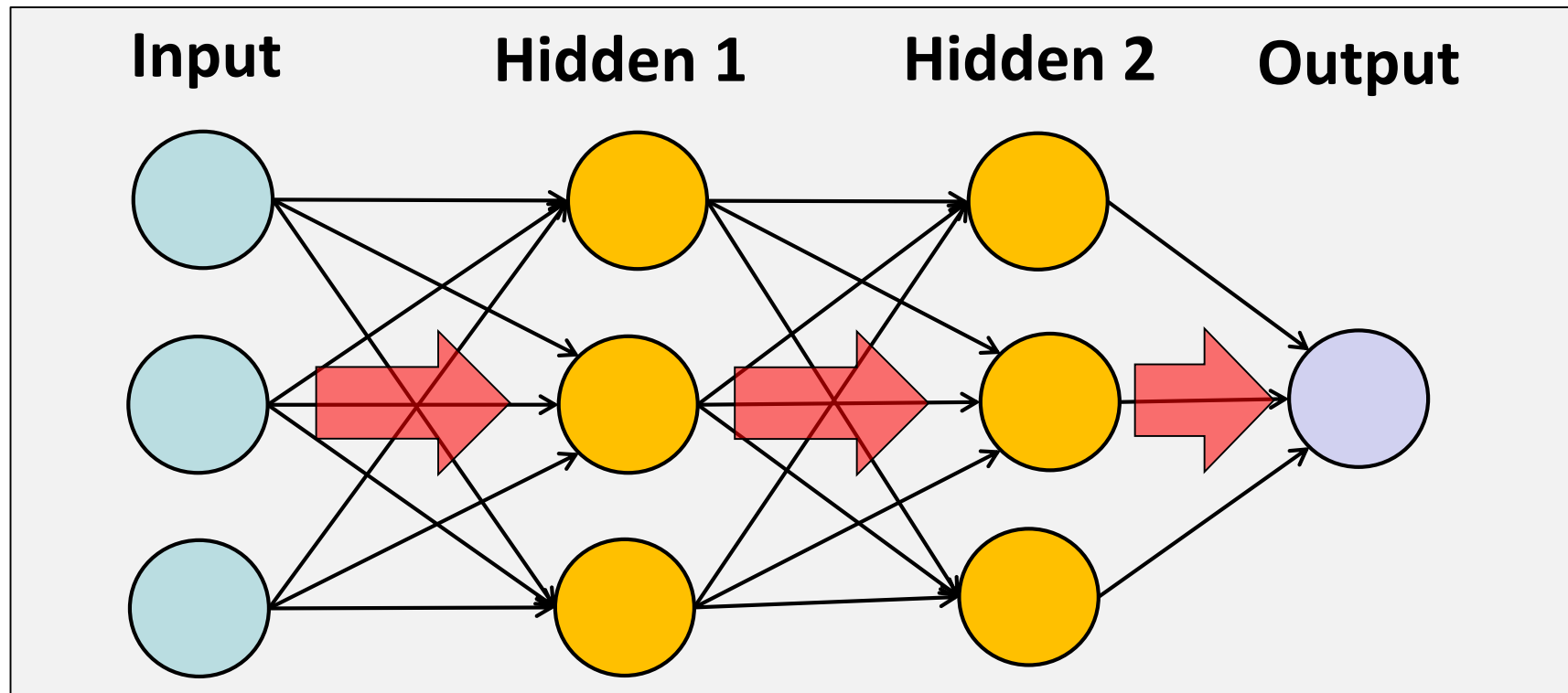
Dr. Nicholas Cummins

- Tutorial sheet and solution sheet from last week will be uploaded sometime this afternoon
- **Exam Dates**
 - Wednesday 12th February at 10:00
 - Wednesday 8th April at 10:00
 - Venue(s): same as lecture and tutorial
 - Languages:
 - Exam will be written in *English*
 - You can answer in *English* or *German*



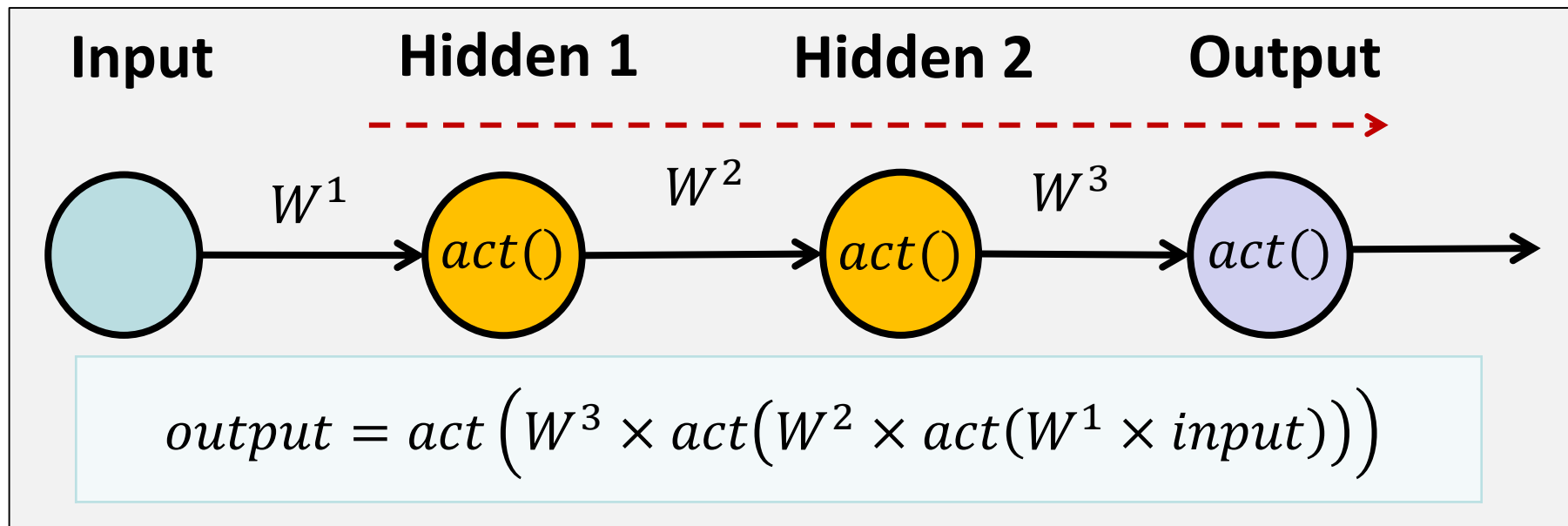
- **Forward Propagation**

- Information flows from input to output to make a predication



- **Forward Propagation**

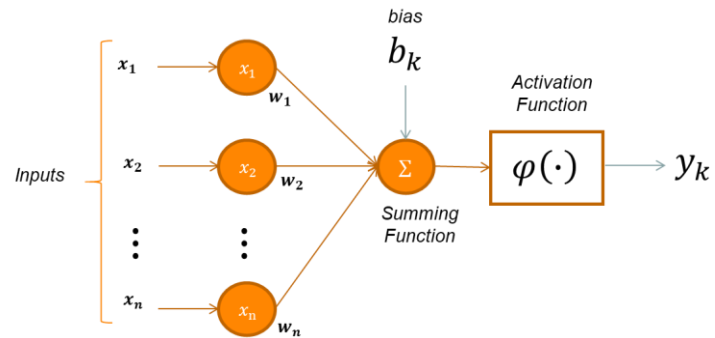
- Each neuron is a function of the previous one connected to it
 - Output is a composite function of the weights, inputs, and activations
 - Change any one of these and ultimately the output will change



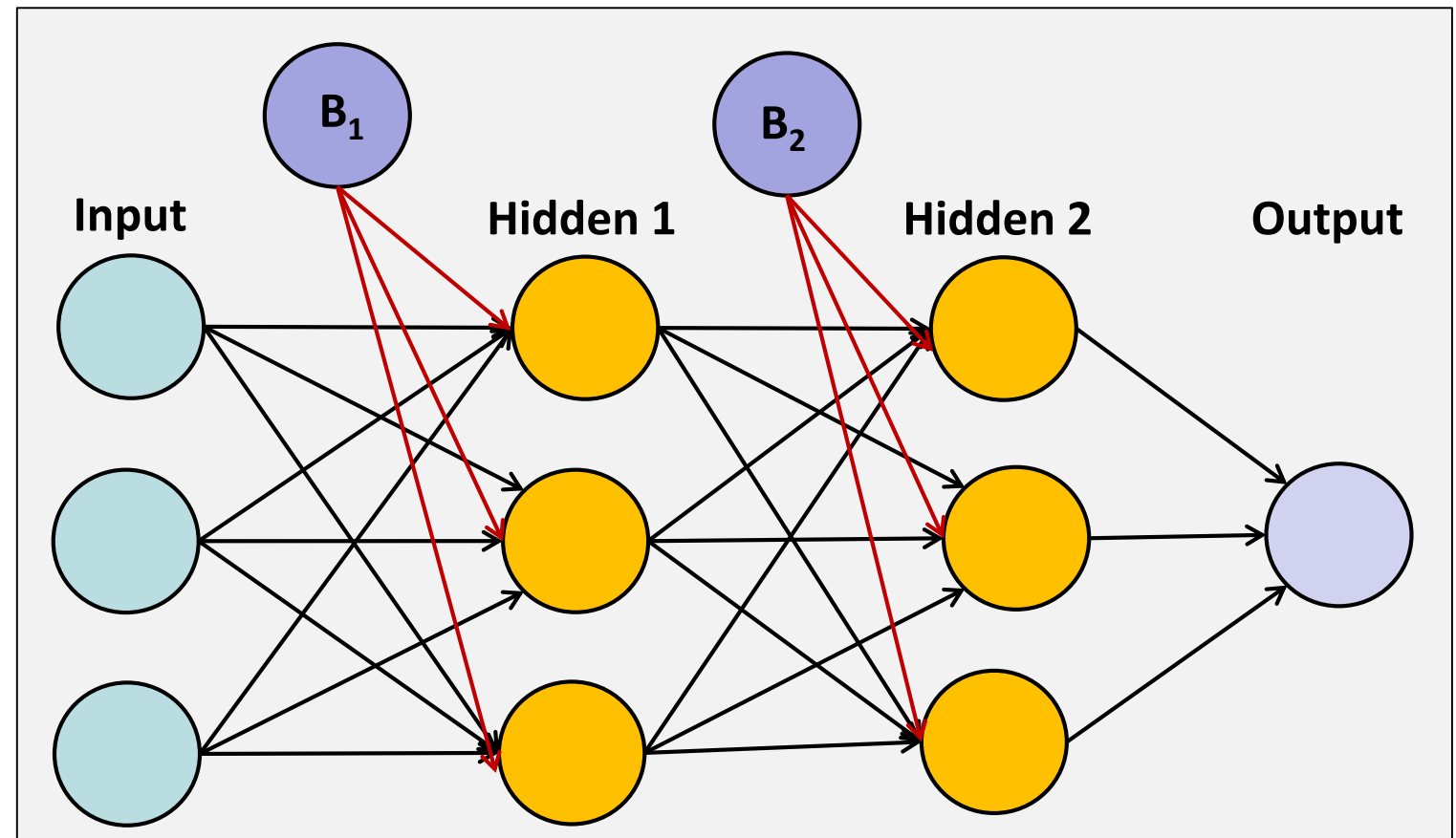
Roles of Bias in a perceptron

• What is Bias?

- Bias is simply a constant value (vector) that is added to the product of inputs and weights



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



Roles of Bias node

- **What is the bias node?**

- Bias is simply a constant value (vector) that is added to the product of inputs and weights

```
output = activation_function(dot_product(weights, inputs) + bias)
```

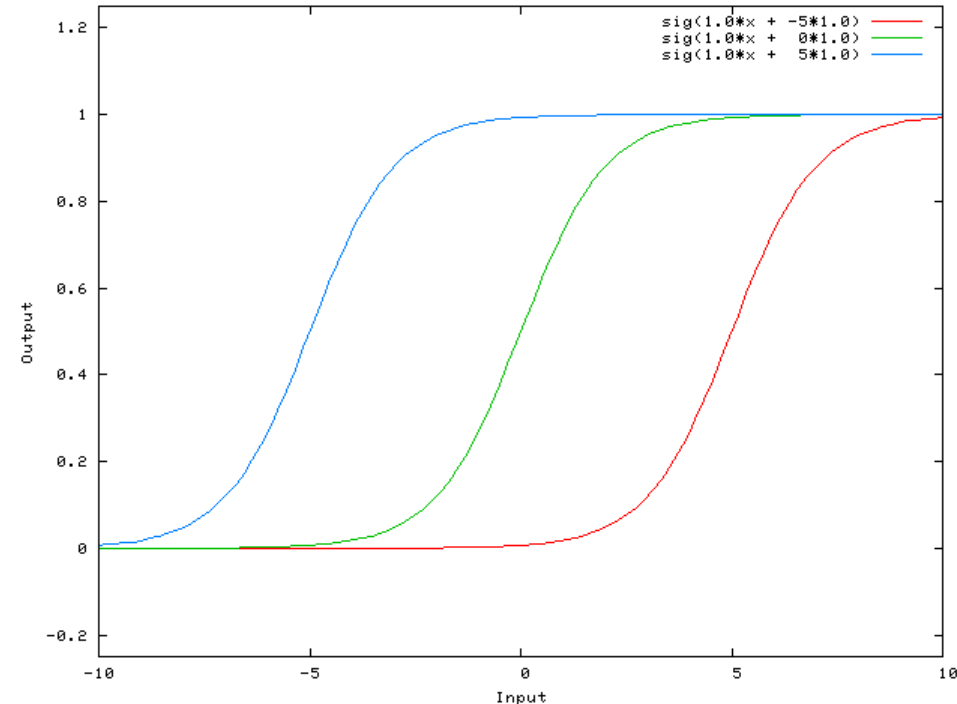
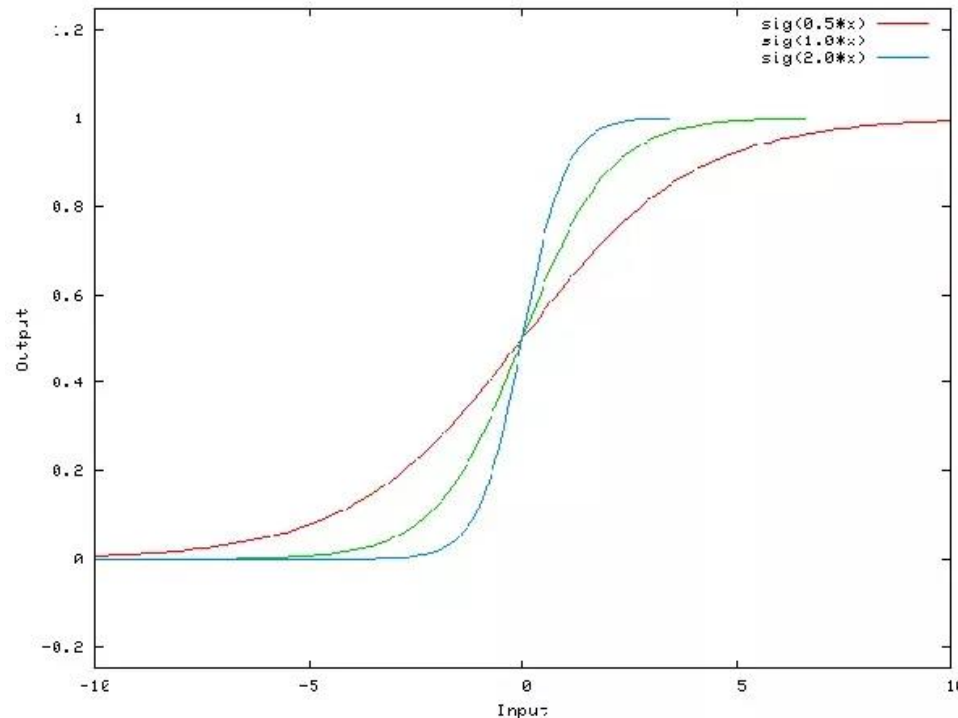
- Allows the activation function to be shifted left or right
- Allows better fit to the data
- Typically initialised to be zero
- **Biases are tuned alongside weights by learning algorithms**
 - E.g. Gradient Descent
 - Note biases differ from weights is that they are independent of the output from previous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

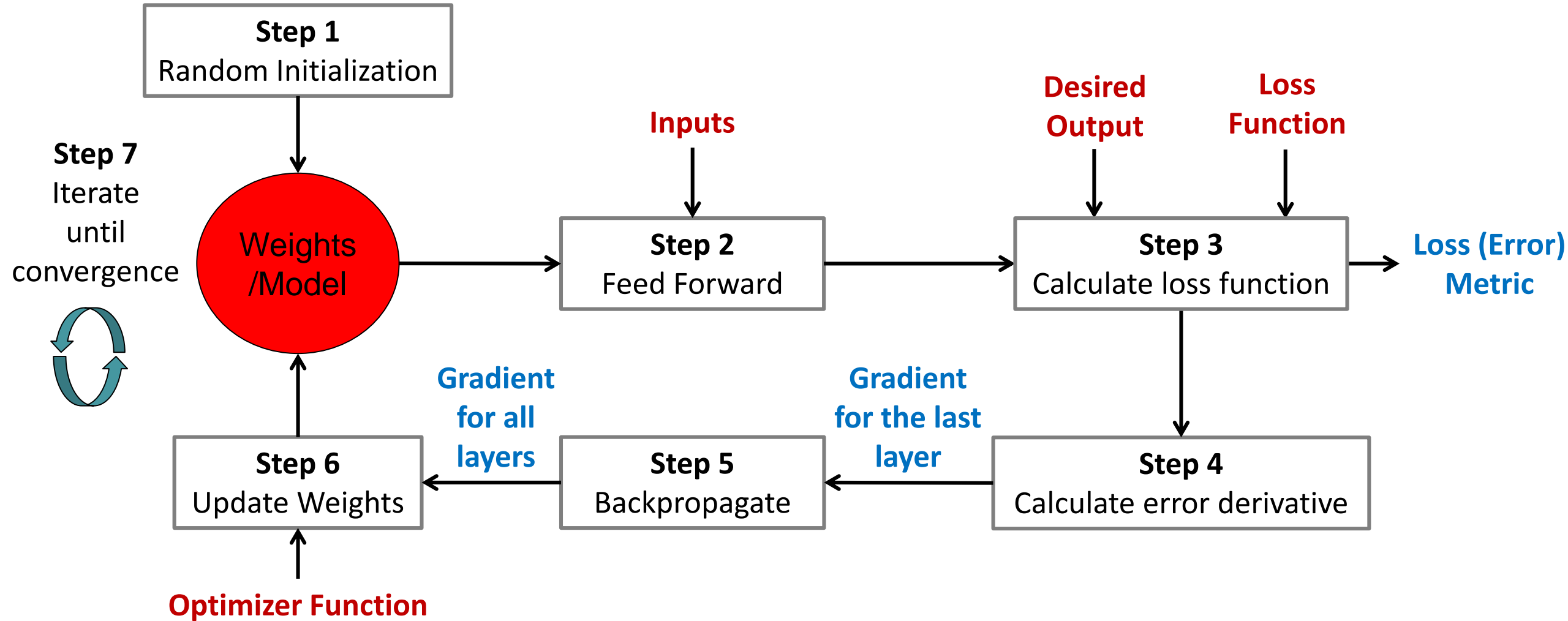
- **What is the bias node?**

- Example sigmoid activation

- Weights will alter steepness of curve
 - Offset will shift the curve left or right

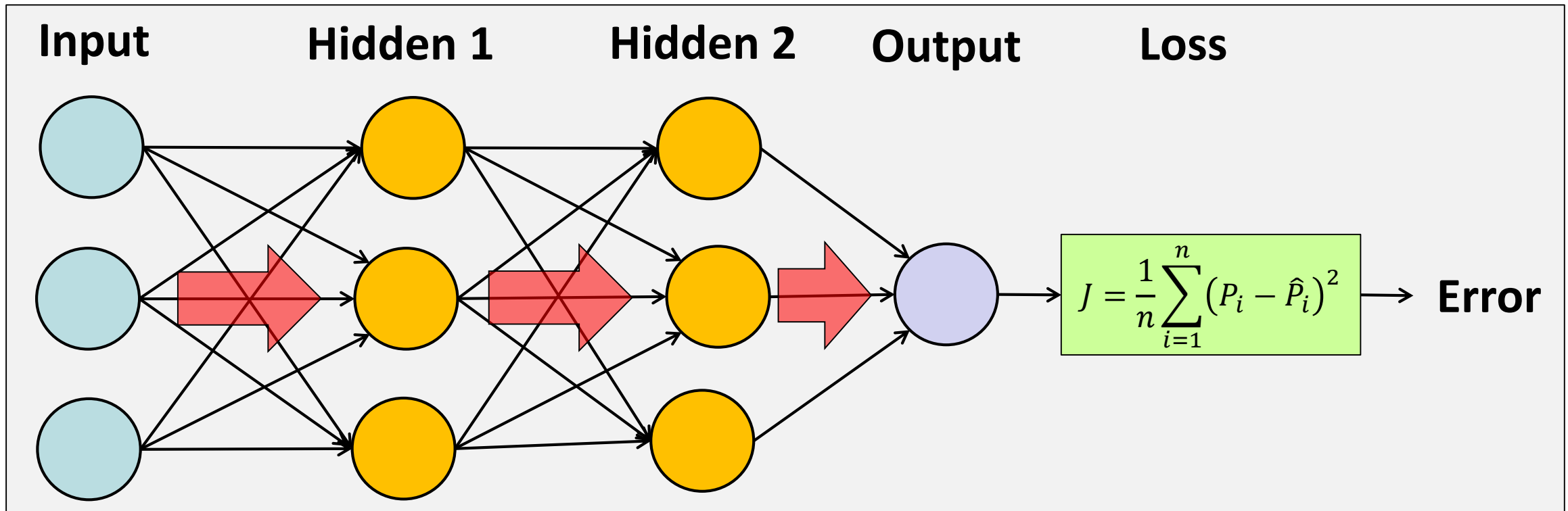
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$





- **Calculate Error/Loss**

- Calculate difference between predication and target



- **The main purpose of machine learning is to be able to predict given data**

- Predict y given some observations x , through function f :

$$f: x \rightarrow y$$

- f is a model and is defined by a set of parameters θ

$$f = f_{\theta}$$

- The goal of machine learning is to find the best parameters
- In supervised learning we know the real value we want to predict y ; i.e., the ground truth or the target

- **Loss function**

- Measures the difference between what we have predicted, \tilde{y} , with the model and what it should predicted y .

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

- Use this information to update the model $f_{\theta+1} = f_{\theta}$
- In order to minimize the cost, we prefer differentiable functions which have simpler optimization algorithms
- Note that a loss function always outputs a scalar value.
 - This value is a measure of fit of the model with the real value.

- **Two most important loss functions**

- **Mean Squared Error (MSE)**

- Usually used for regression

$$\mathcal{L}(\tilde{y}, y) = \sum_i (\tilde{y} - y)^2$$

- **Cross Entropy**

- Used for classification, the model predicts the probability of observation x having a particular label

$$x \rightarrow \{p_c\}_c \text{ with } \sum_c p_c = 1$$

- Cross entropy measure the distance between two probabilities

$$\mathcal{L}(\tilde{p}, p) = \sum_c p_c \log(\tilde{p}_c)$$

- **The choice of cost function is tightly coupled with the choice of output unit**

Problem Type	Output Type	Final Activation Function	Loss function
Regression	Numerical	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple label, multiple classes	Sigmoid	Binary Cross Entropy

- **What is an Activation function?**
 - 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 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

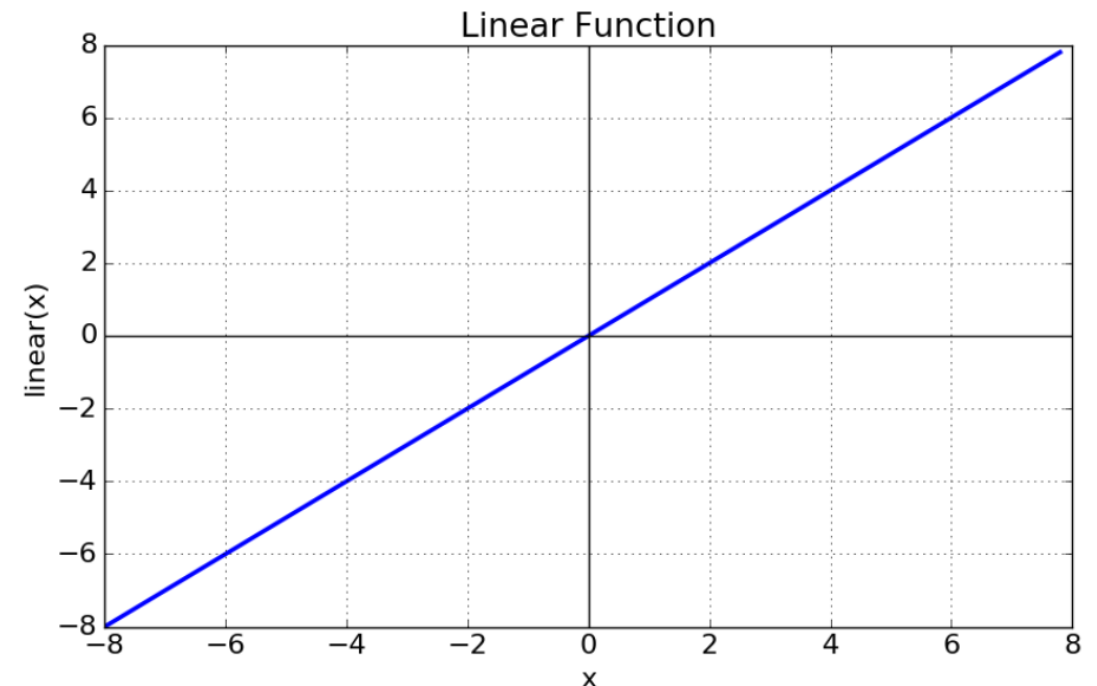
- **Linear**

- A straight line function
 - a is always a constant value
- Values can get large
- Does not capture complex patterns

$$f(x) = ax$$

$$f'(x) = a$$

Range: $-\infty$ to ∞



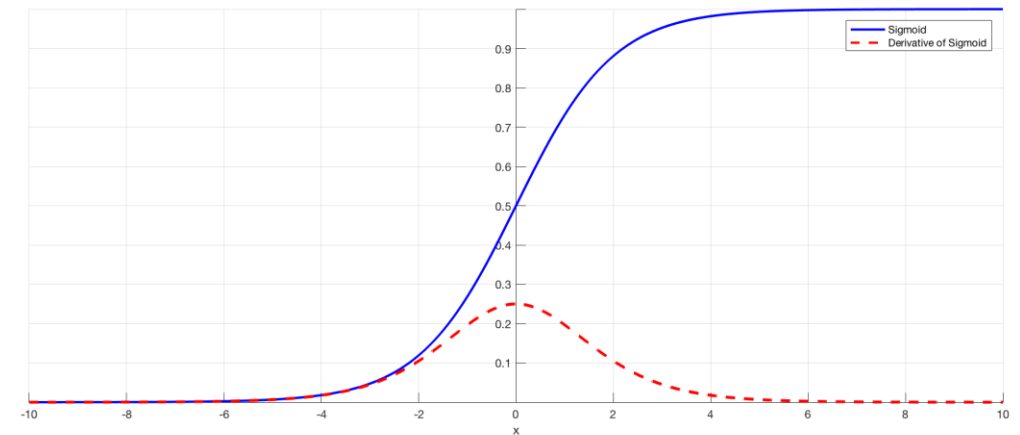
- **Sigmoid**

- Non-linear so can capture more complex patterns
- Output values are bounded so don't get too large can get large
- Can suffer from *vanishing gradient*
 - When $|x|$ becomes large, the derivative becomes close to zero

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

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Range: 0 to 1



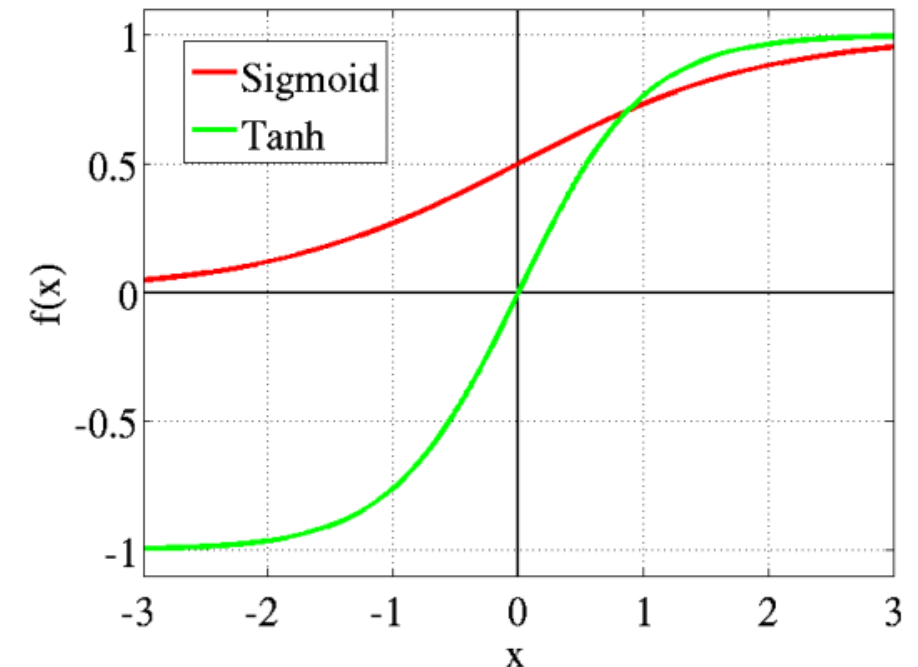
- **Hyperbolic Tangent**

- Non-linear so can capture more complex patterns
- Output values are bounded so cannot get quite large
- Negative inputs mapped to negative outputs
- Zero inputs mapped to zero output
- Can also suffer from vanishing gradient

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\tanh'(x) = 1 - \tanh(x)^2$$

Range: -1 to 1

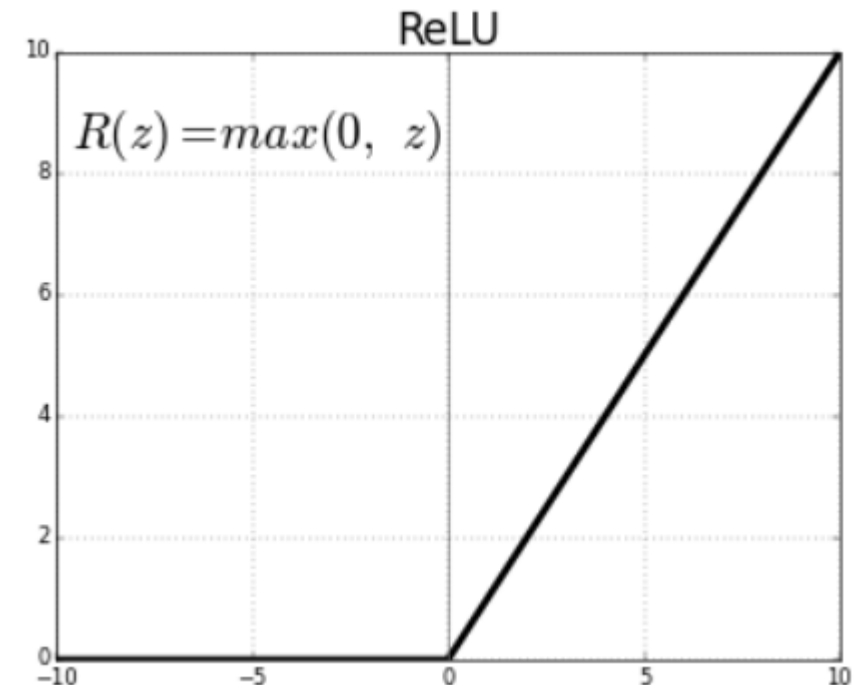


- **Rectified Linear Unit (ReLU)**
 - Non-linear so can capture more complex patterns
 - Positive values can get very large
 - Simple derivative
 - Does not allow for negatives
 - Certain patterns may get missed
 - Dying ReLU: gradient goes to zero and weight is not updated

$$\text{ReLU}(x) = \max(0, x)$$

$$\text{ReLU}'(x) = \begin{cases} 0, & \text{for } x < 0 \\ 1, & \text{for } x \geq 0 \end{cases}$$

Range: 0 to ∞



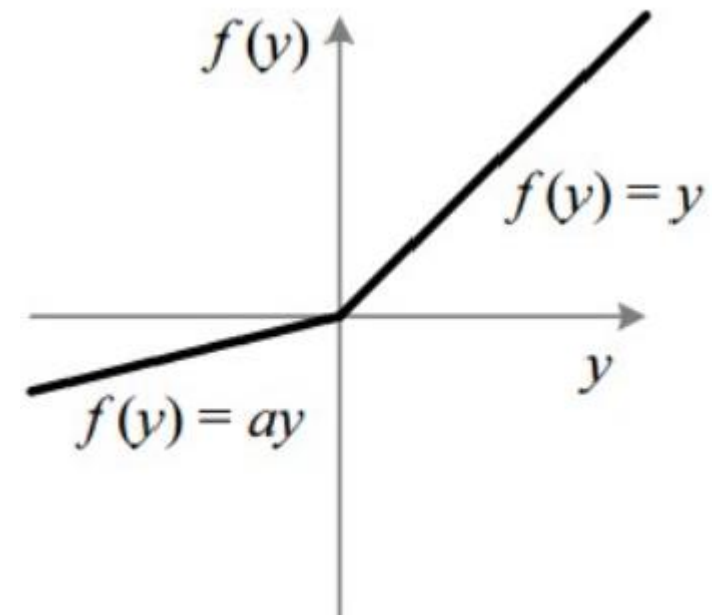
- **Leaky Rectified Linear Unit (LReLU)**

- Non-linear so can capture more complex patterns
- Values can get very large
- Simple derivative
- Attempts to solve for dying ReLU issues

$$LReLU(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

$$LReLU'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Range: $-\infty$ to ∞



- **Softmax**

- Models probability distribution
 - Each value ranges between 0 and 1
 - All values sum to 1
- **Typically only used in the output layer**
- Softmax is a vector function, so the derivative is calculated via partials

$$S(\mathbf{x}): \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix} \rightarrow \begin{bmatrix} S_1 \\ S_2 \\ \dots \\ S_N \end{bmatrix}$$

$$S(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

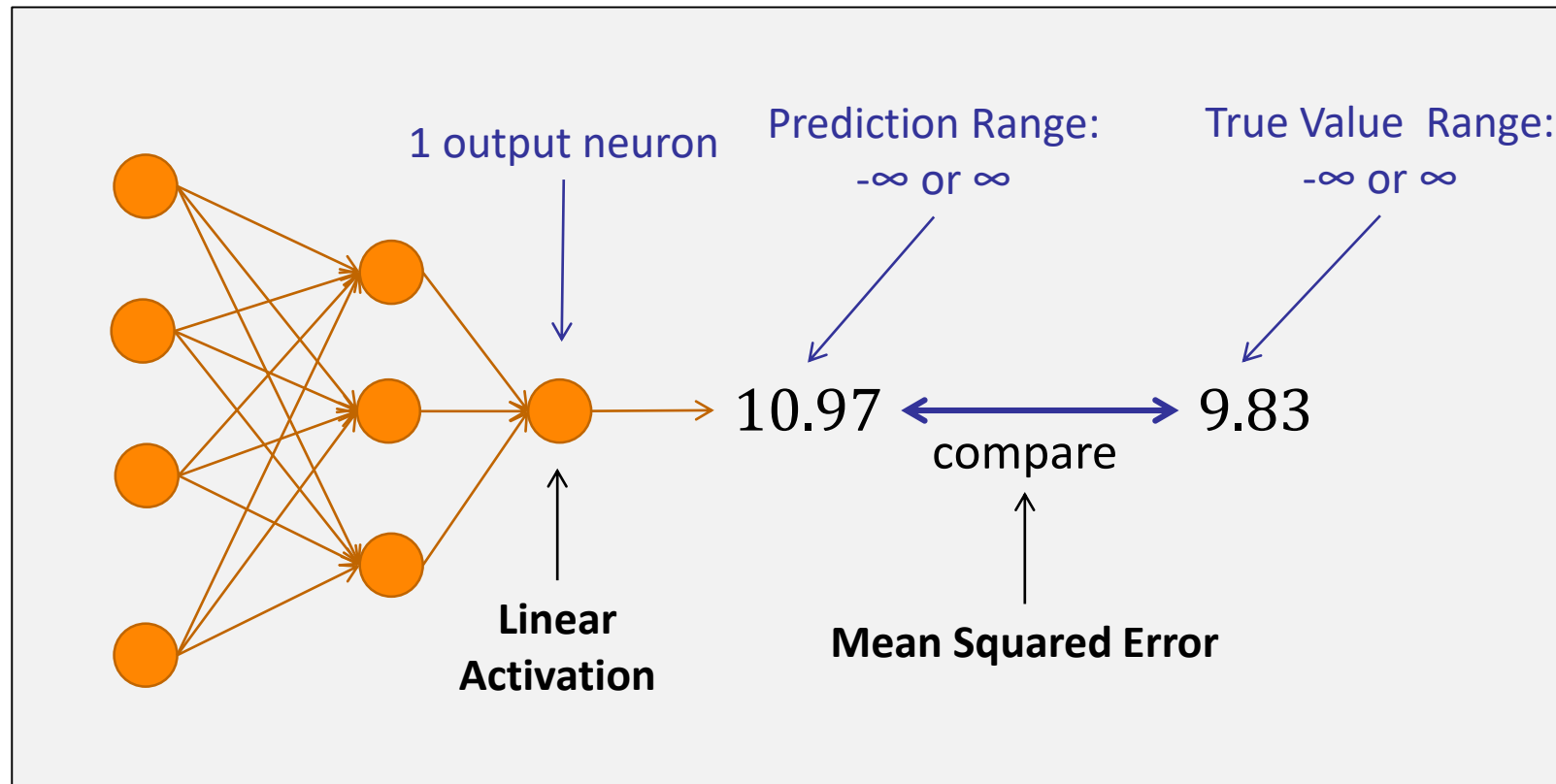
$$\frac{\partial S_i}{\partial x_j} = \begin{cases} S_i(1 - S_i), & \text{for } i = j \\ -S_i S_j, & \text{for } i \neq j \end{cases}$$

Range: 0 to 1

- **The choice of cost function is tightly coupled with the choice of output unit**
 - What are you trying to solve?
 - Are you trying to predict a numerical value → **Regression**
 - Are you trying to predict a categorical outcome?
 - Binary: data is or isn't a class → **Predicting a binary outcome**
 - Multiple classes which are exclusive → **Predicting a single label from multiple classes**
 - If there are multiple labels in your data → **Predicting multiple labels from multiple classes**

Loss \leftrightarrow Activation function relationship

- **Regression**
 - Predicting a single numerical value

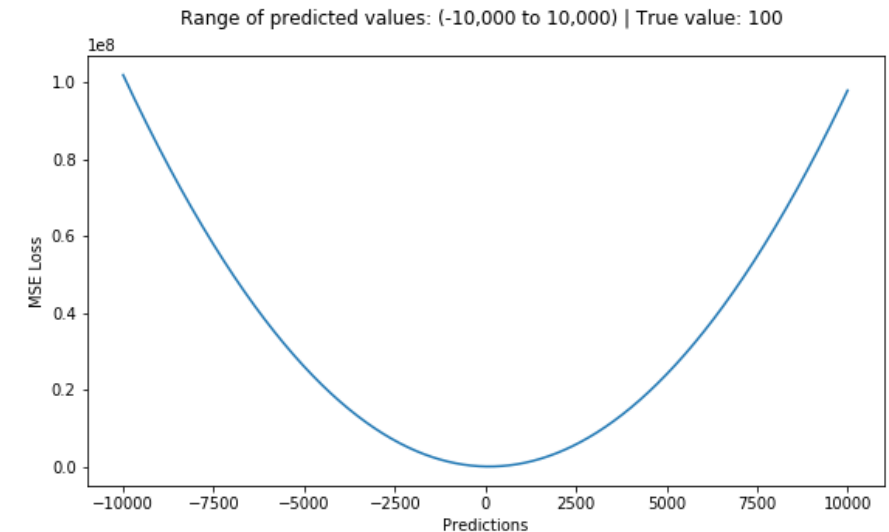


Loss \leftrightarrow Activation function relationship

- **Regression**

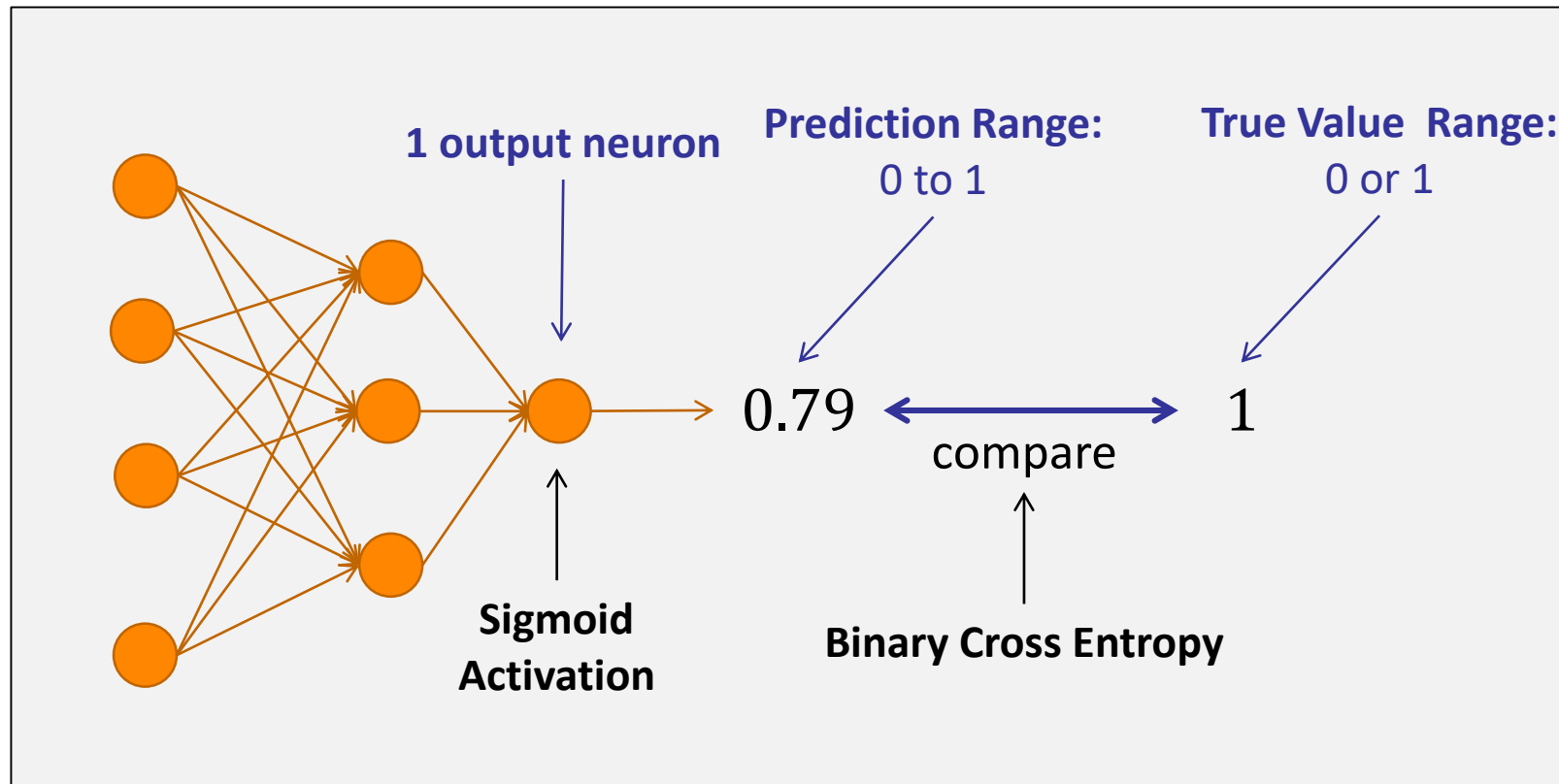
- Predicting a single numerical value
- Final Activation function – **Linear**
- Loss function – **Mean Squared Error**

$$\mathcal{L}(y, \tilde{y}) = \frac{1}{n} \sum_n (y - \tilde{y})^2$$



Loss \leftrightarrow Activation function relationship

- **Predicting a binary outcome**
 - Data is or isn't in a class



Loss \leftrightarrow Activation function relationship

- **Predicting a binary outcome**

- **Data is or isn't a class**

- Final Activation function – **Sigmoid**

- This results in a value between 0 and 1 which we can infer to be how confident the model is of the example being in the class

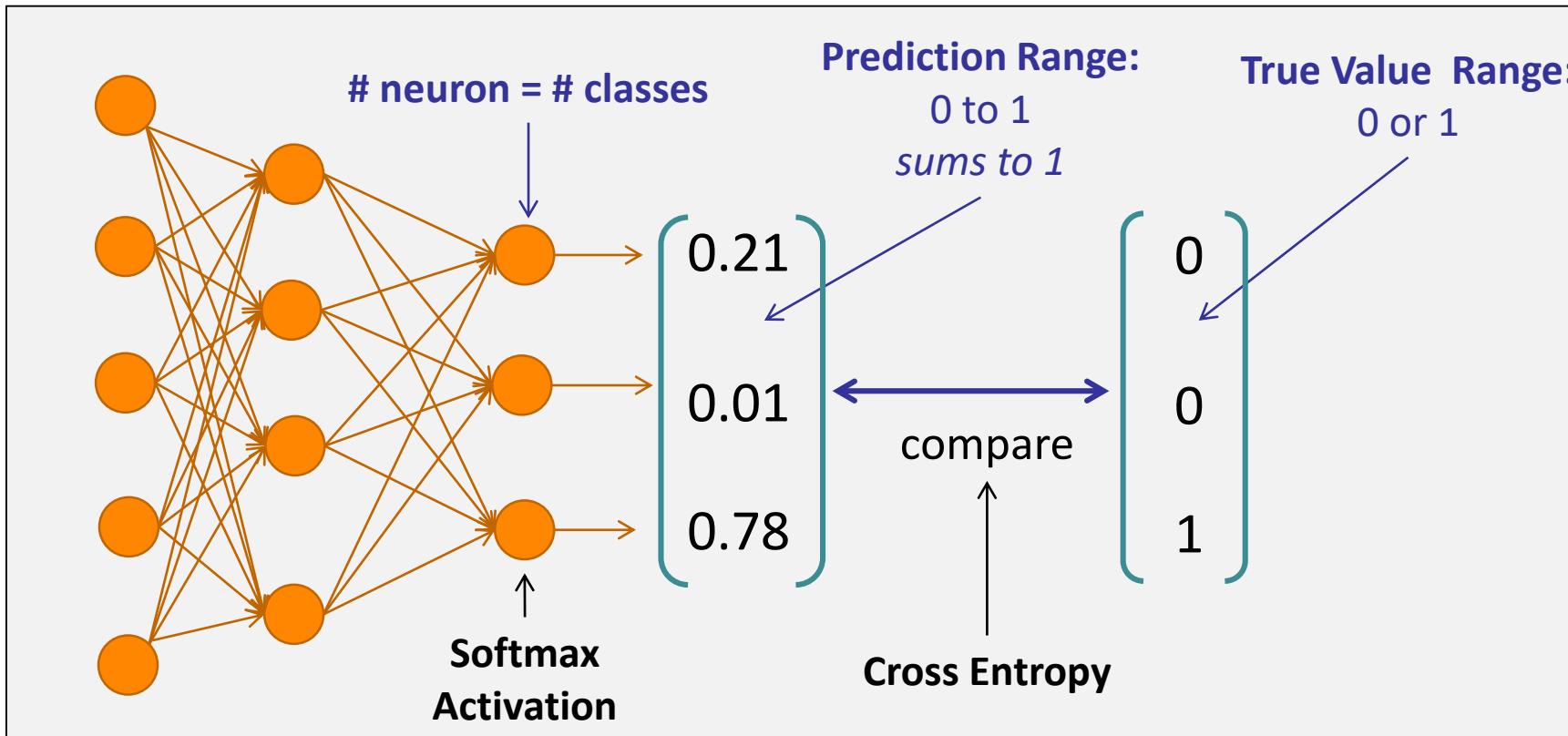
- Loss function – **Binary Cross Entropy**

- Our model predicts a model distribution of $\{p, 1 - p\}$
 - We compare this with the true distribution $\{y, 1 - y\}$

$$BCE = -(y \log(\tilde{y}) + (1 - y) \log(1 - \tilde{y}))$$

Loss \leftrightarrow Activation function relationship

- Predicting a single label from multiple classes
 - Multiple classes which are exclusive



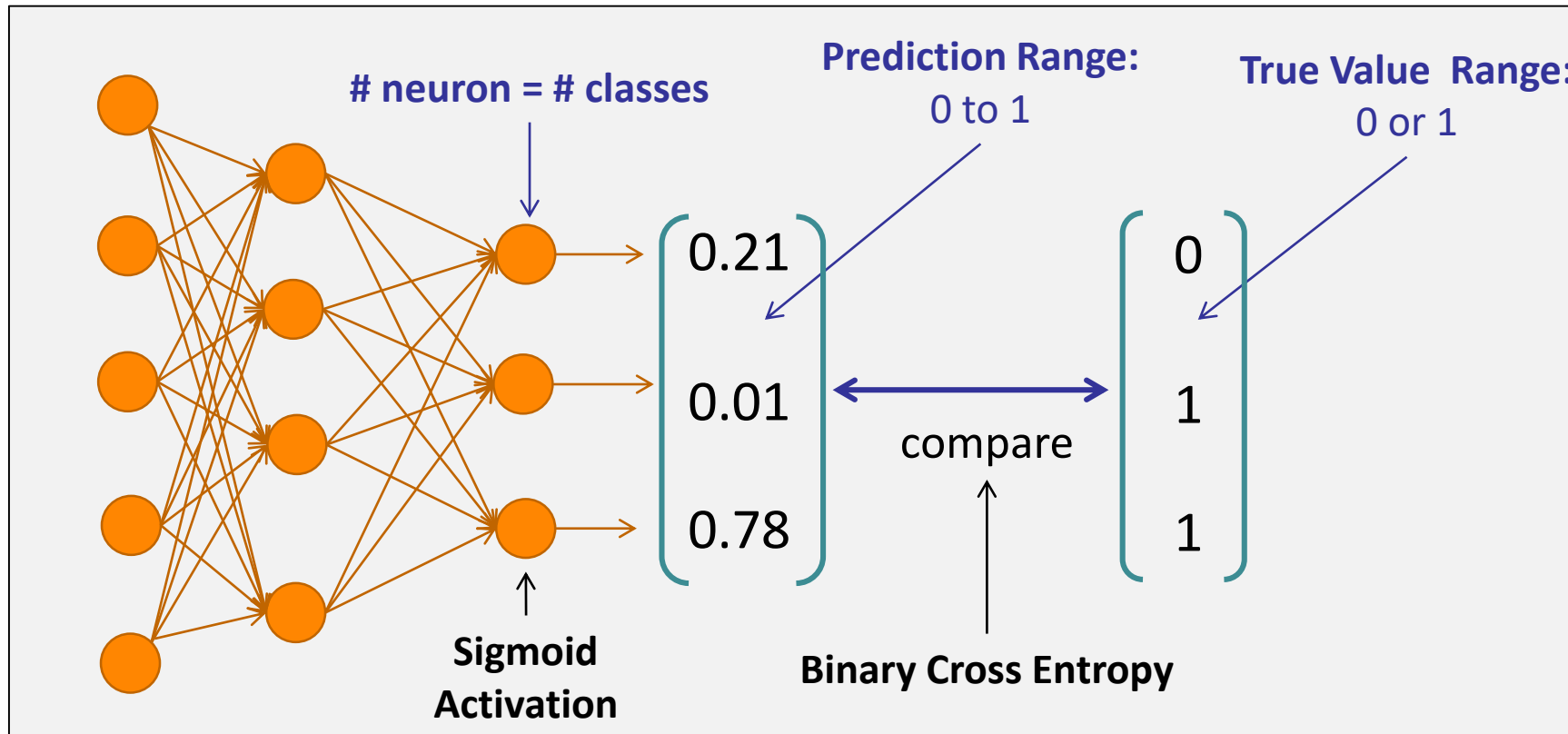
Loss \leftrightarrow Activation function relationship

- **Predicting a single label from multiple classes**
 - Multiple classes which are exclusive
 - Final Activation function – **Softmax**
 - This results in values between 0 and 1 for each of the outputs which all sum up to 1.
 - Consequently, this can be inferred as a probability distribution
 - Loss function – **Cross Entropy**
 - Quantifies the difference between two probability distribution

$$CE = - \sum_{i=1}^M y_i \log(\tilde{y}_i)$$

Loss \leftrightarrow Activation function relationship

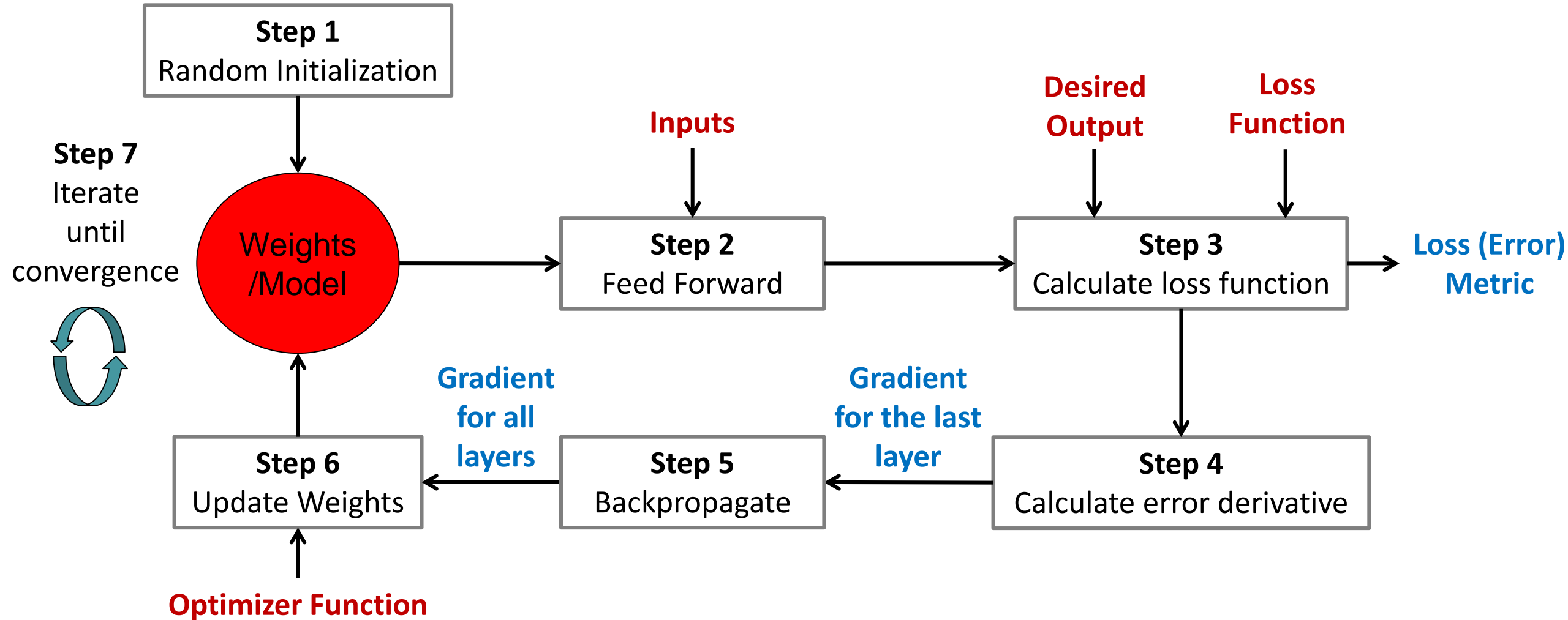
- Predicting multiple labels from multiple classes
 - If there are multiple labels in your data



Loss \leftrightarrow Activation function relationship

- **Predicting multiple labels from multiple classes**
 - **If there are multiple labels in your data**
 - The final layer will have one neuron for each classes
 - This neuron will return a value between 0 and 1
 - This can be inferred as a probably
 - Final Activation function – **Sigmoid**
 - This results in a value between 0 and 1 which we can infer to be how confident it is of it being in the class
 - Loss function – **Binary Cross Entropy**

$$BCE = -(y \log(\tilde{y}) + (1 - y) \log(1 - \tilde{y}))$$



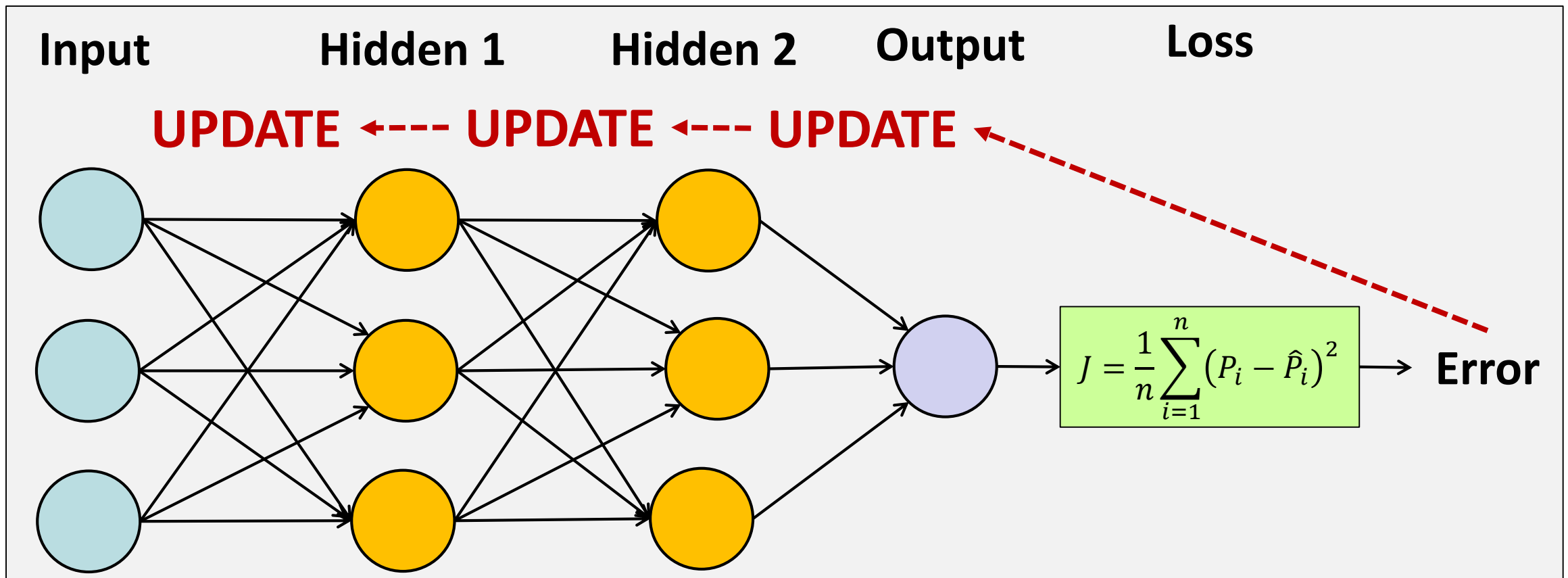
- **Perform Gradient Descent**

- Update weights to minimise loss function
- This is achieved by taking the gradient of loss function with respect to the weights

$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} - \alpha \nabla \mathcal{L}(\mathbf{W}^{(t)}; \mathbf{x}, \mathbf{y})$$

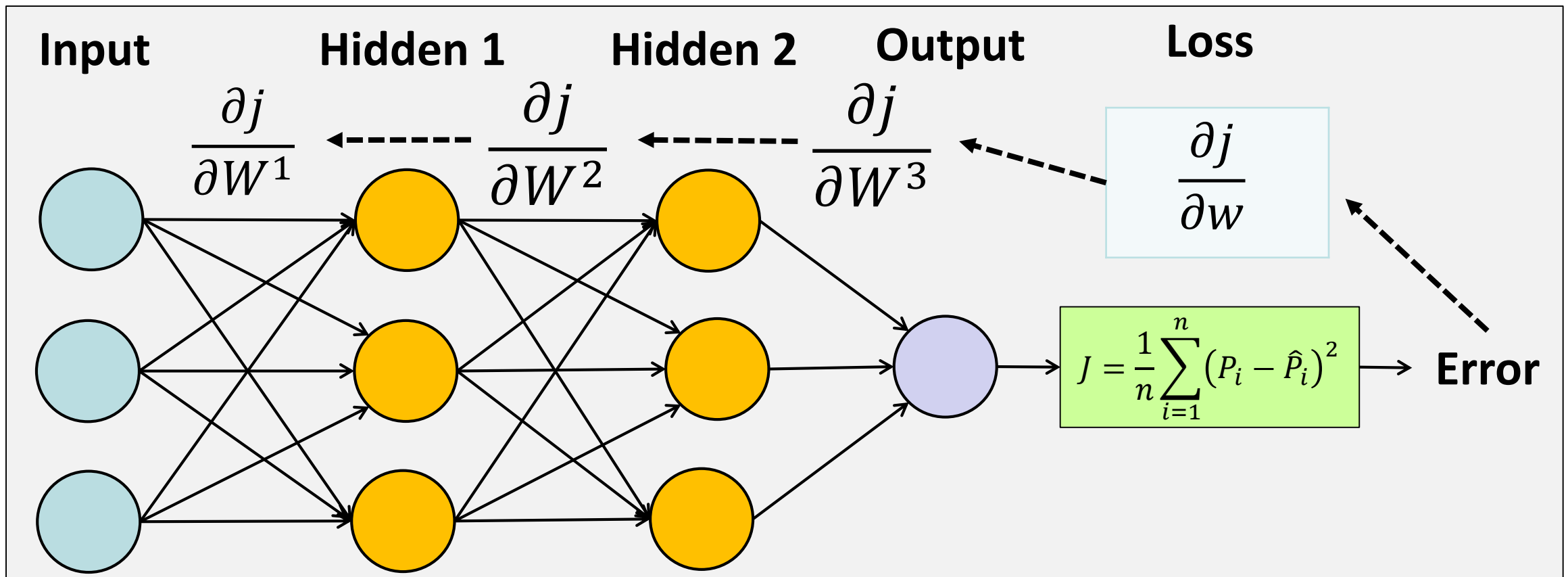
- Not a trivial process as neural networks are structured as a series of layers
 - The output is a composite function of the weights, inputs, and activation function(s)

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

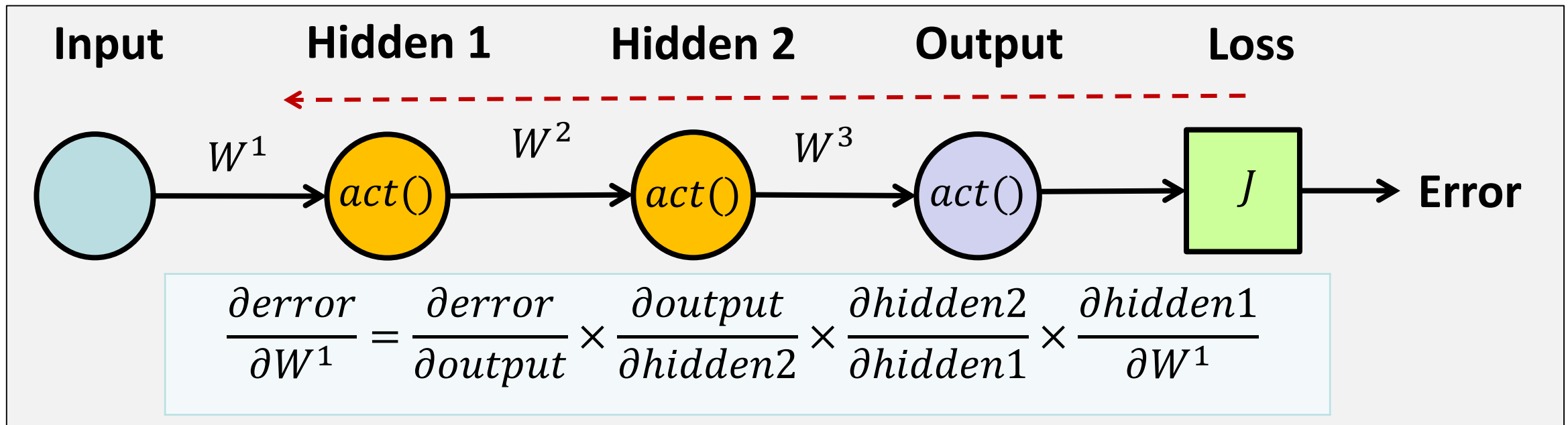


- **Backpropagation**

- Tool to calculate the gradient of the loss function



- **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



- Weights

Feed-Forward \rightarrow

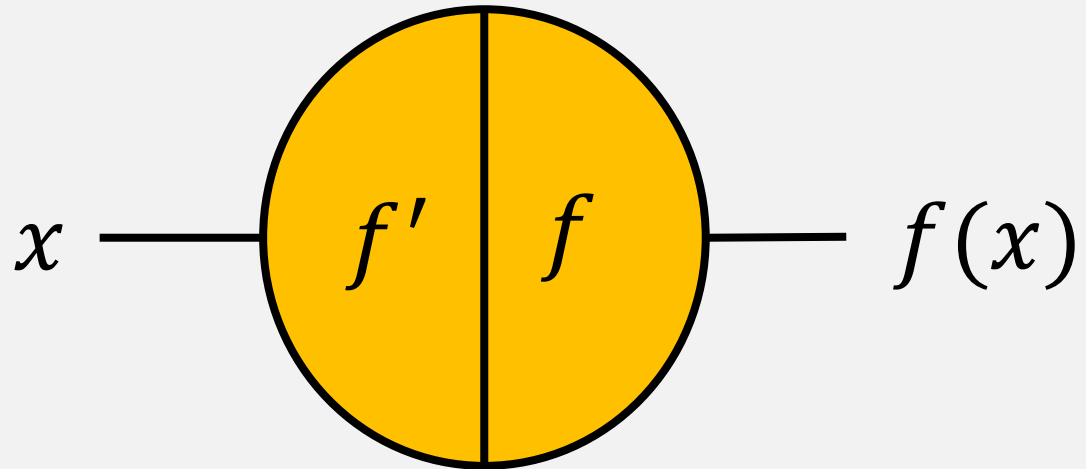
$$x \xrightarrow{w} wx$$

\leftarrow Backpropagation

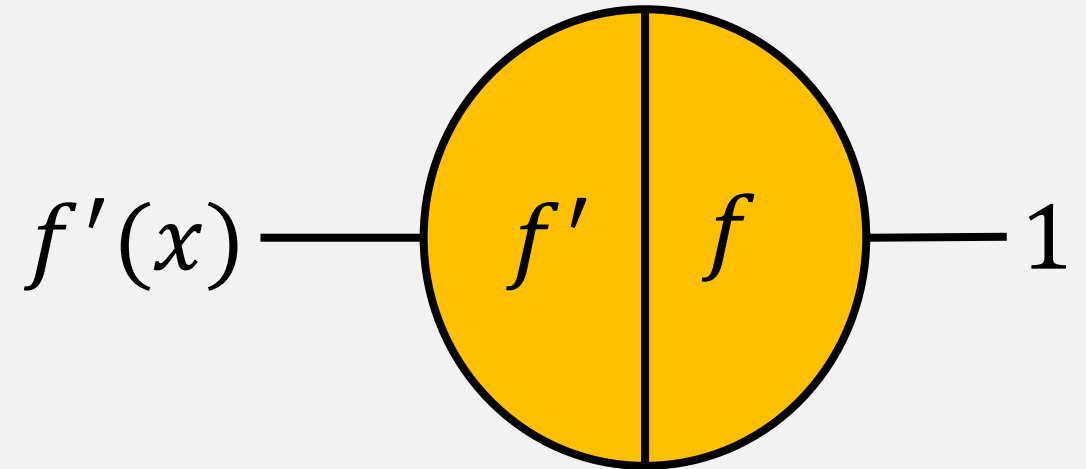
$$w \xleftarrow{w} 1$$

- Activation function

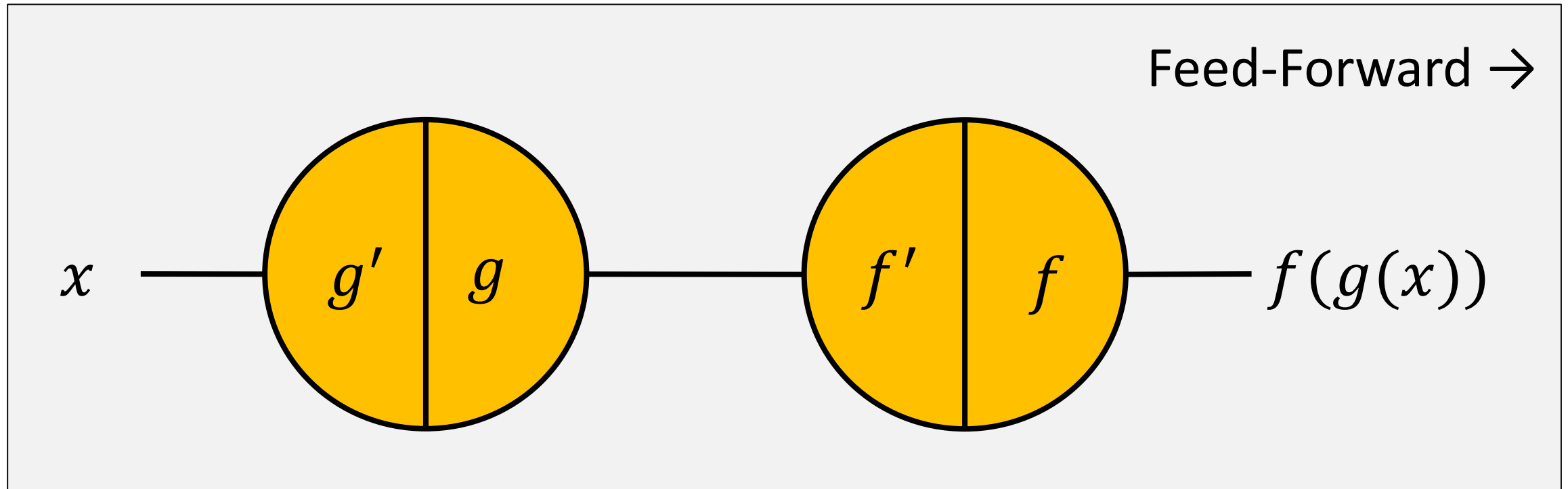
Feed-Forward \rightarrow



\leftarrow Backpropagation

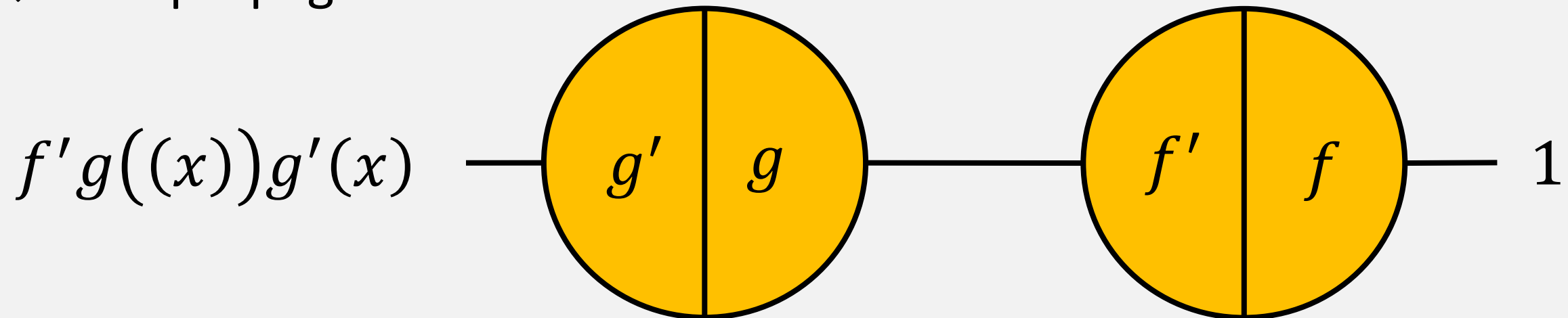


- Composite Function

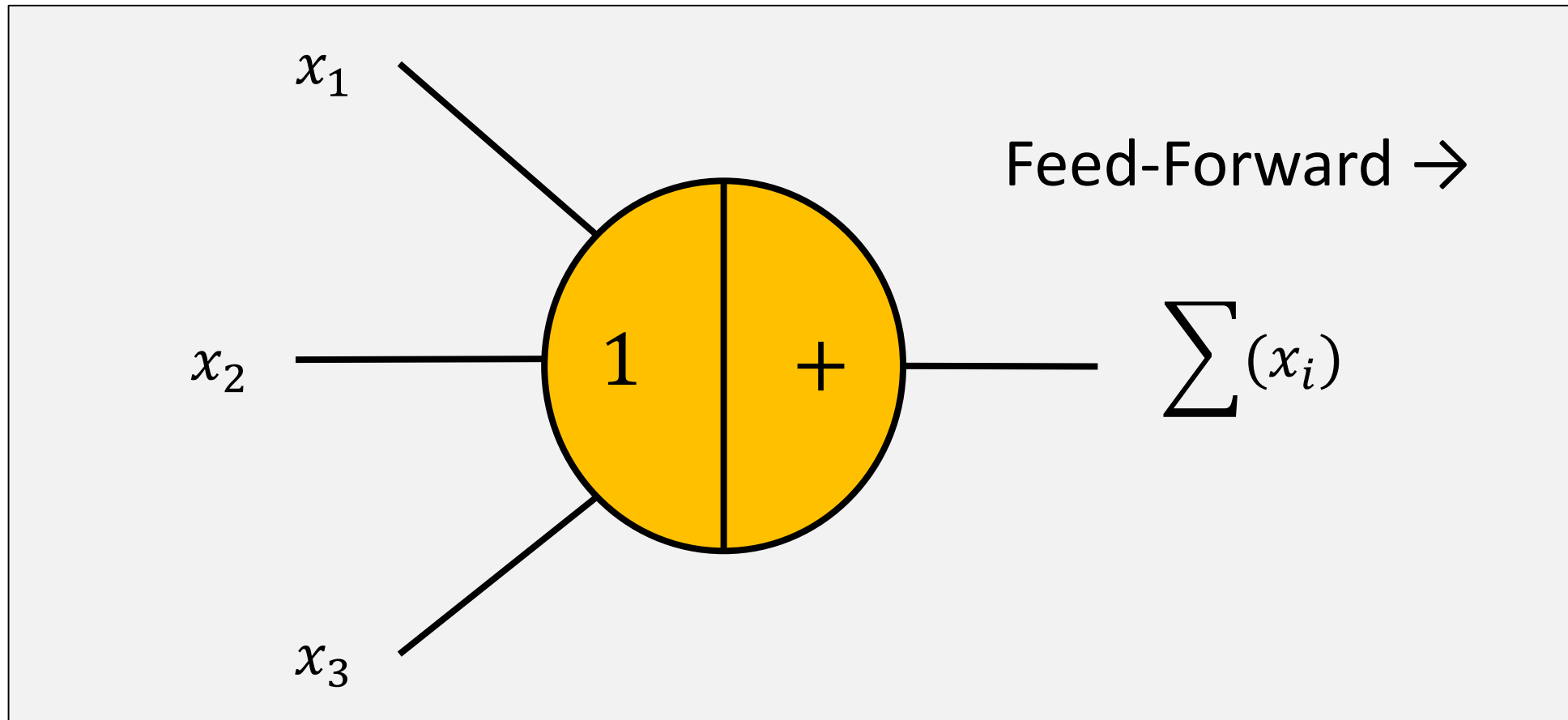


- Composite Function

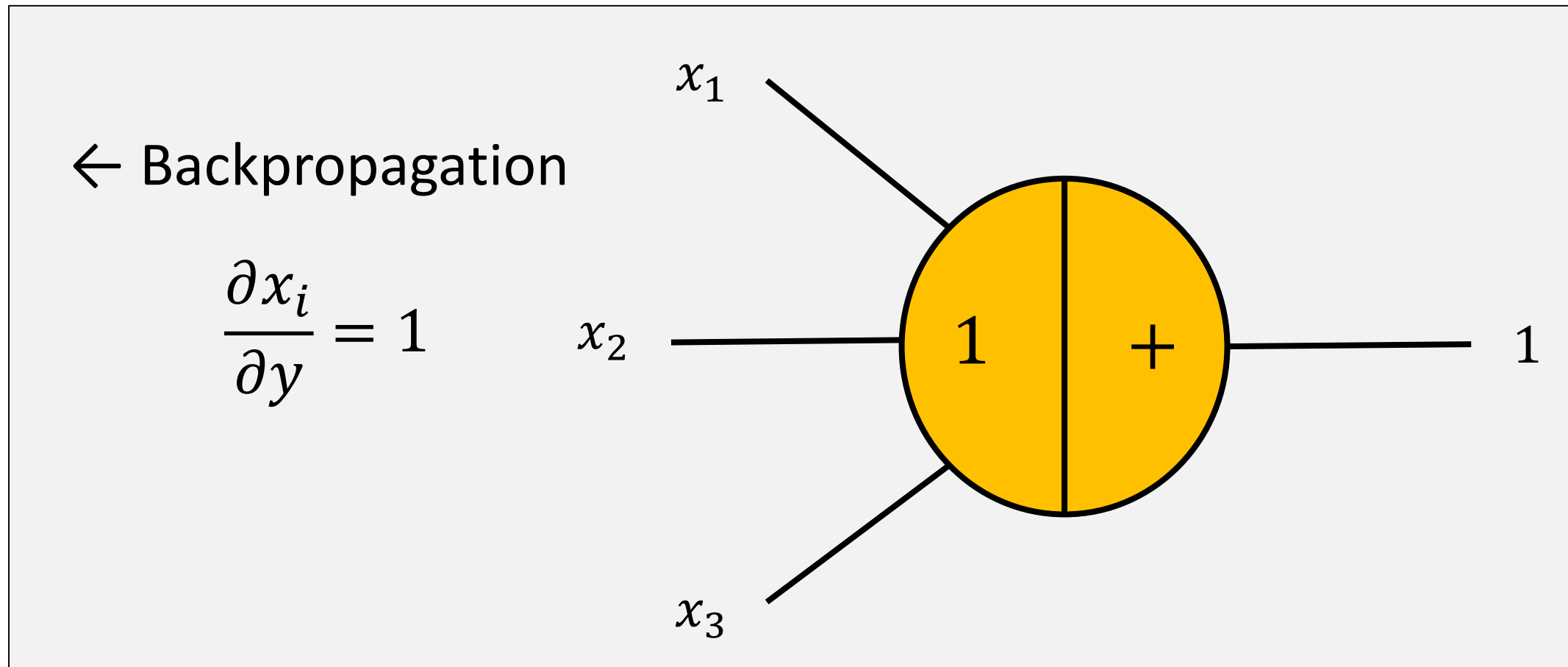
← Backpropagation



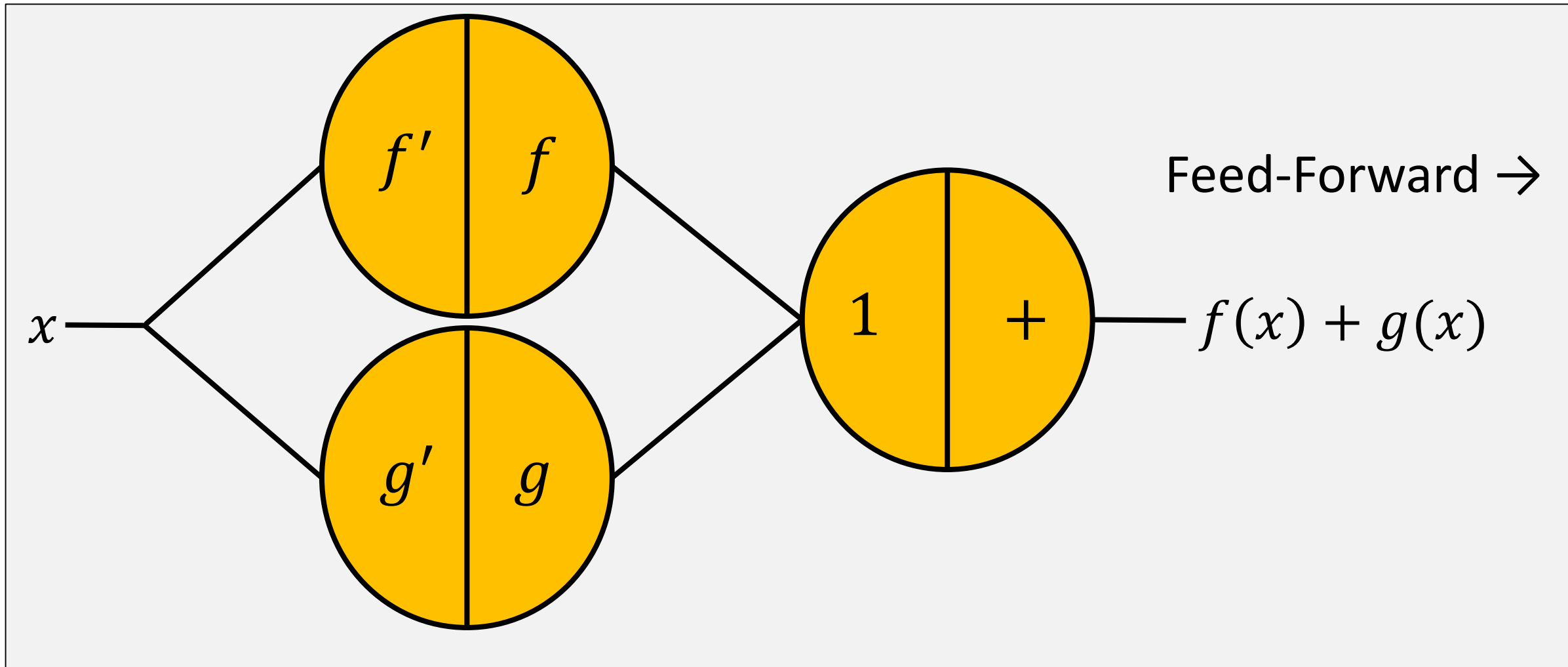
- Summation Function



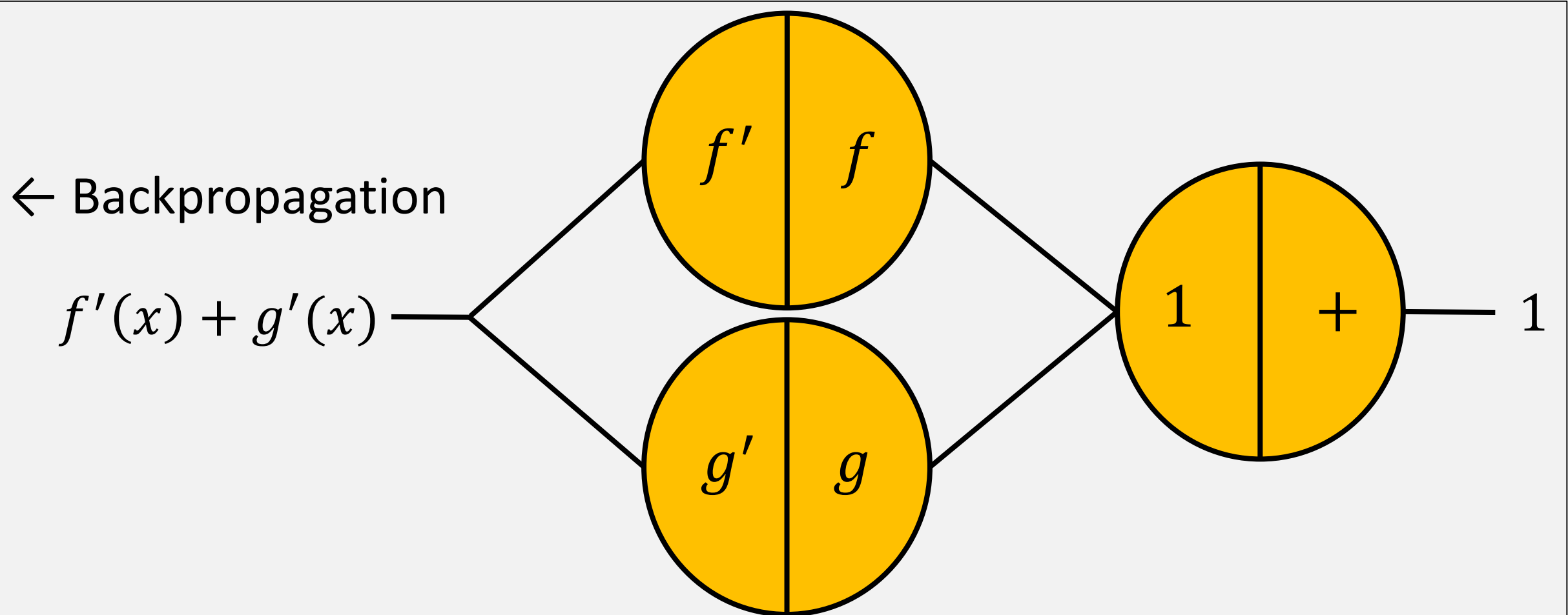
- Summation Function

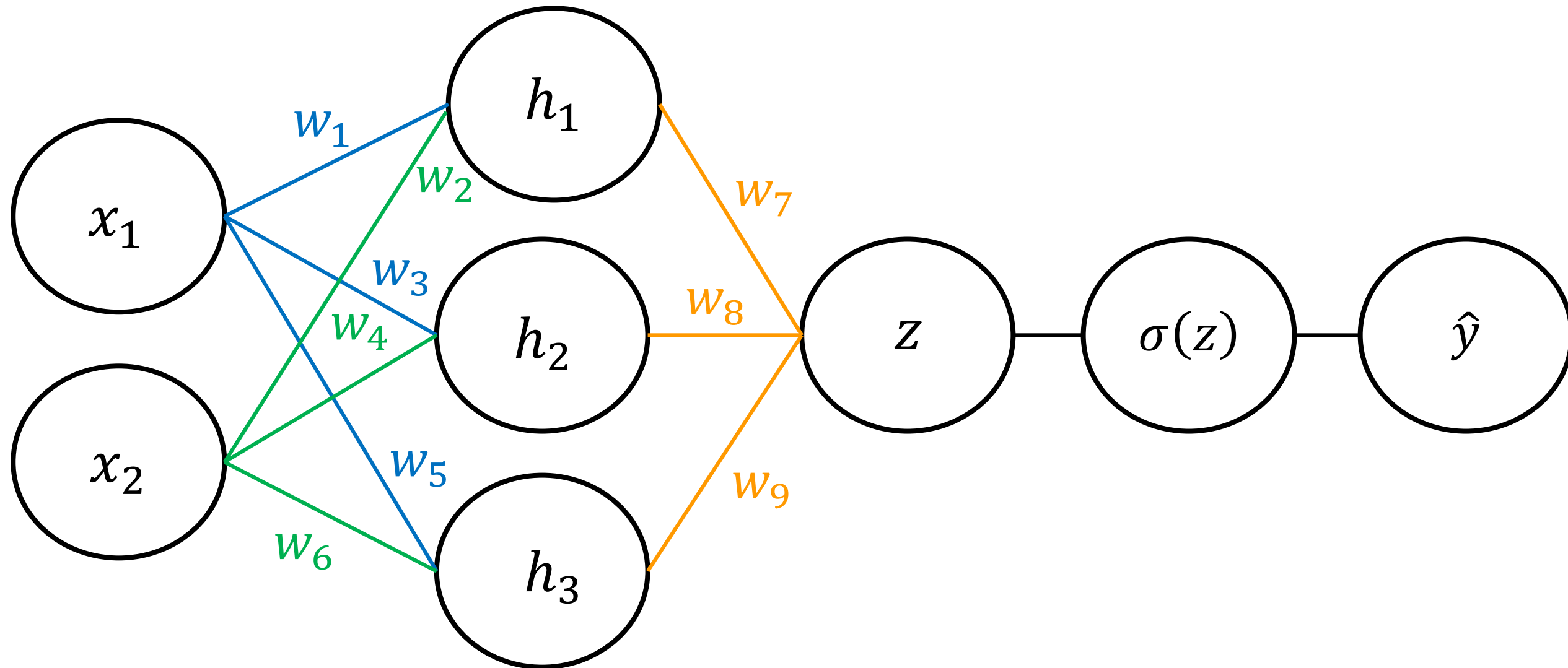


- Addition of functions



- Addition of functions

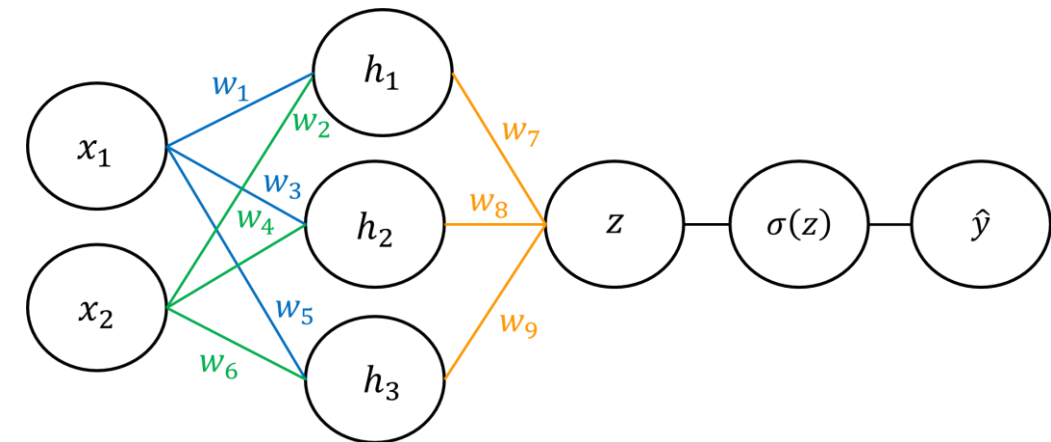


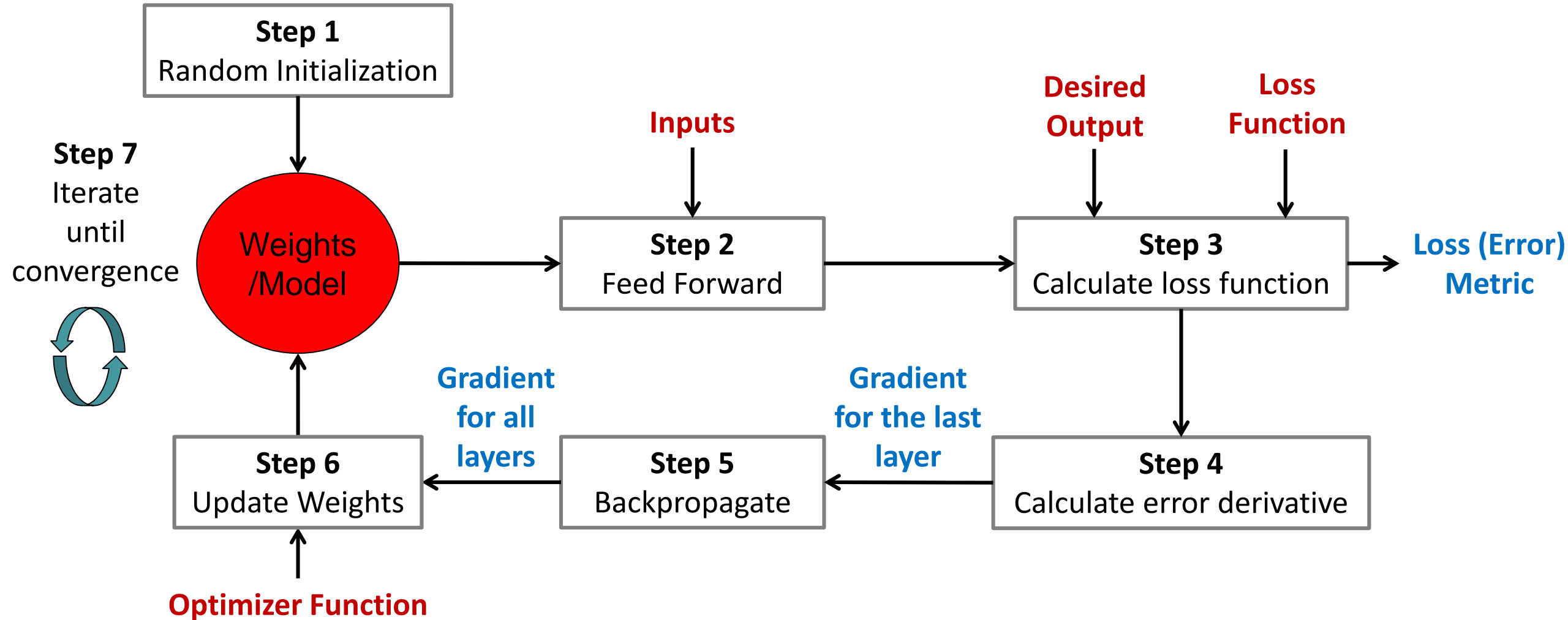


Question:

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
- $L = \frac{1}{2} (y - \hat{y})^2, \frac{dL}{d\hat{y}} = \hat{y} - y$





- **Gradient Descent for neural learning**

```
pred = input * weight  
error = (pred - goal_pred) ** 2  
derivative = input * (pred - goal_pred)  
weight = weight - (alpha * derivative)
```

- During training we use gradient descent to update the parameters (weights) of our network
- Minimising the loss function to make our corrections as correct as possible
- So far we have only discussed basic Gradient Descent, but there are many different algorithms for updating neural networks

- **Gradient Descent**

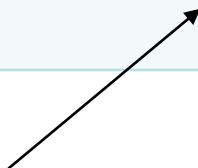
- Iterative optimisation algorithm

- **Core idea:**

- Gradient indicates the direction of increase
 - As we want to find the minimum point in the valley we need to go in the opposite direction of the gradient
 - We therefore update parameters in the negative gradient direction to minimize the loss

$$\theta = \theta - \eta \nabla J(\theta; x, y)$$

Typical value
of η is 0.01



- **Full, batch, and stochastic gradient descent**
 - Stochastic gradient descent
 - Prediction & weight update performed separately for each training sample, see code on previous slide
 - Full gradient descent
 - Calculates the average `weight_delta` over the entire dataset
 - Only updates weights once full average has been calculated
 - Batch gradient descent
 - Choose a batch size (typically between 8 and 256) to calculate `weight_delta` on, then update the weights

- **Gradient Descent strategies**

- **Batch gradient descent**

- Computes the gradient of the cost function with respect to the entire training dataset

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla C(\theta^{(t)}; \mathbf{x}, \mathbf{y})$$

- **Advantages**

- Theoretical analysis of weights and convergence rates are easy to understand

- **Disadvantages**

- Performs many redundant computation
 - Slow and potentially intractable

- **Gradient Descent strategies**

- **Stochastic gradient descent (SGD)**

- Performs a parameter update on each training example
 - Random sampling of training data used to help avoid local minima

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

- **Advantages**

- Quicker learning than Batch Gradient Descent
 - Easy to update weights if new training samples are acquired

- **Disadvantages**

- As we frequently update weights, Cost function fluctuates heavily

- **Gradient Descent strategies**

- **Mini-Batch**

- Performs an update for every mini-batch of n training examples
 - Helps reduce noise in variance of weight updates when compared to SGD

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

- **Advantages**

- Reduction in variance means more stable convergence
 - Learning speed
 - Good approximation of location of actual minima

- **Disadvantages**

- Total loss not accumulated

- **Gradient Descent strategies**

- **Batch gradient descent**

- Computational heavy, smoothest trajectory to convergence, should converge

- **Stochastic gradient descent (SGD)**

- Computational light, noisy convergence trajectory, may not converge

- **Mini-Batch**

- Trade-off between above methods
- Batching adds noise to learning process which can improve generalisation

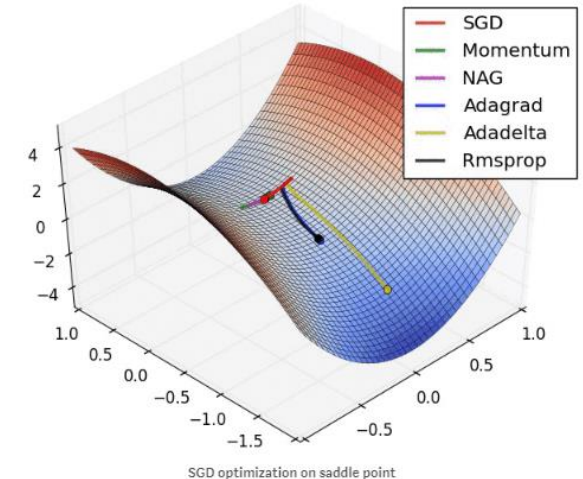
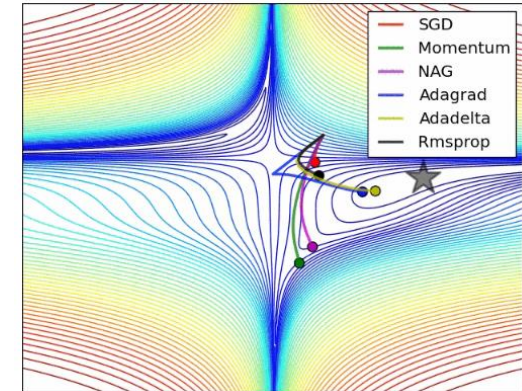


- Batch gradient descent
- Mini-batch gradient Descent
- Stochastic gradient descent

Source: <https://towardsdatascience.com/>

- **Types of Optimizers**

- Momentum
- Nesterov Accelerated Gradient (NAG)
- Adagrad — *Adaptive Gradient Algorithm*
- Adadelata
- RMSProp - *Root Mean Square Propagation*
- Adam — *Adaptive Moment Estimation*



• Momentum

- Used to accelerate convergence of Gradient Descent
 - Analogous to a ball rolling down a hill
- It takes the gradient of the current step as well as the gradient of the previous time steps when updating weights
 - Helps convergence when cost function has a ravine
 - Surfaces which curve more steeply in one dimension than in another

Typical value
of γ is 0.9

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta; x, y)$$

$$\theta = \theta - v_t$$



(a) SGD without momentum



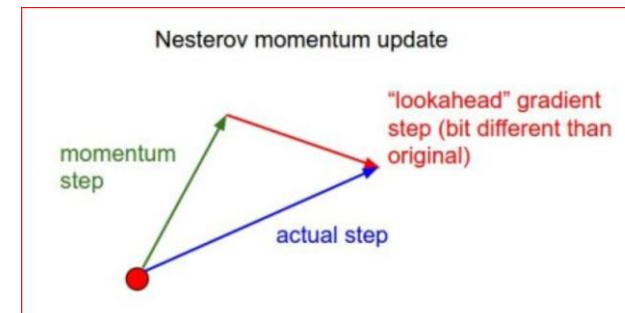
(b) SGD with momentum

- **Nesterov Accelerated Gradient (NAG)**

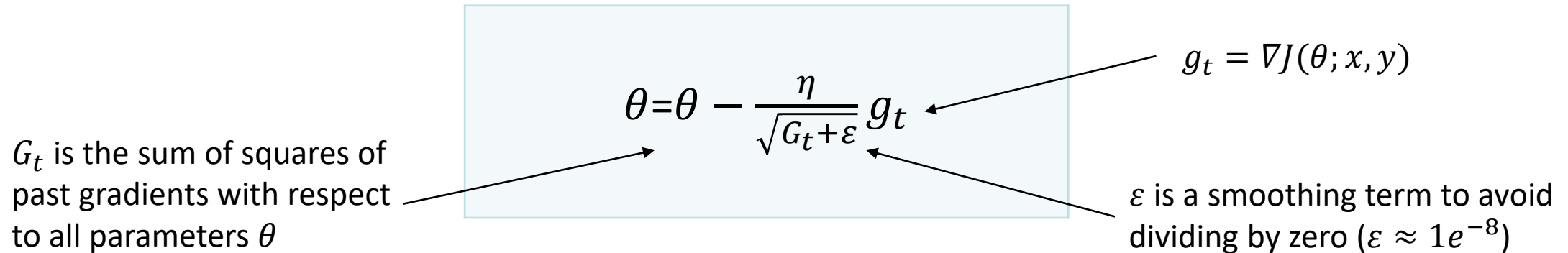
- Momentum is not always good, it is often useful to know when to slow down (approach a rise in the cost function)
 - From momentum: γv_{t-1} will move the parameters θ
 - Computing $\theta - \gamma v_{t-1}$ gives approximation of updated parameters
 - We can effectively look ahead by calculating gradient with respect to this future position
- We calculate the gradient not with respect to the current step but with respect to the future step

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta - \gamma v_{t-1}; x, y)$$

$$\theta = \theta - v_t$$



- **Adagrad — *Adaptive Gradient Algorithm***
 - Expensive to tune learning (and moment) rates
 - Normally done heuristically
 - Adagrad is an adaptive learning rate method
 - We perform larger updates for infrequent parameters and smaller updates for frequent parameters.



The diagram shows the Adagrad update rule:
$$\theta = \theta - \frac{\eta}{\sqrt{G_t + \varepsilon}} g_t$$
 Annotations include:

- An arrow pointing from the text " G_t is the sum of squares of past gradients with respect to all parameters θ " to the G_t term in the denominator.
- An arrow pointing from the text " $g_t = \nabla J(\theta; x, y)$ " to the g_t term in the numerator.
- An arrow pointing from the text " ε is a smoothing term to avoid dividing by zero ($\varepsilon \approx 1e^{-8}$)" to the ε term in the denominator.

- **Adagrad — *Adaptive Gradient Algorithm***

$$\theta = \theta - \frac{\eta}{\sqrt{G_t + \varepsilon}} g_t$$

- **Advantage**

- Eliminates the need to manually tune the learning rate

- **Disadvantage**

- Aggressive monotonically decreasing learning rate
 - Accumulation of squared gradients $\rightarrow G_t$ keeps growing
 - Learning rate shrinks and becomes too small for acquire new updates
 - Adadelta, RMSProp & Adam attempt to address this

- **Adadelta**

- Restricts the window size of accumulated past gradients
 - The sum of gradients is recursively defined as a decaying average of all past squared gradients
- Learning rate is set dynamically as the ratio of the running average of the previous time steps to the current gradient

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

$$\Delta\theta = - \frac{RMS[\Delta\theta]_{t-1}}{RMS[g_t]} g_t$$

RMS denotes the root mean squared error criterion

- **RMSProp - Root Mean Square Propagation**

- Uses a moving average of the squared gradient to adjust the learning rate
 - It divides the learning rate by an exponentially decaying average of squared gradients

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t$$

γ typically set to 0.9

η typically set to 0.001

- **Adam — *Adaptive Moment Estimation***
 - Calculates adaptive learning rate from estimates of first and second moments of the gradients
 - Adam is computationally efficient and has very little memory requirement
 - Adam can be viewed as a combination of Adagrad, which works well on sparse gradients and RMSprop which works well in online and nonstationary settings.

- **Adam — *Adaptive Moment Estimation***

- Algorithm

- The Adam algorithm first updates the exponential moving averages of the gradient (m_t) and the squared gradient (v_t) which are the estimates of the first and second moment
- Hyper-parameters $\beta_1, \beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages as shown below
 - Note: default values are 0.9 for β_1 , and 0.999 for β_2

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- **Adam — *Adaptive Moment Estimation***

- Algorithm

- Moving averages are initialized as 0 leading to moment estimates that are biased around 0 especially during the initial timesteps.
 - This initialization bias can be easily counteracted resulting in bias-corrected estimates

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- **Adam — *Adaptive Moment Estimation***

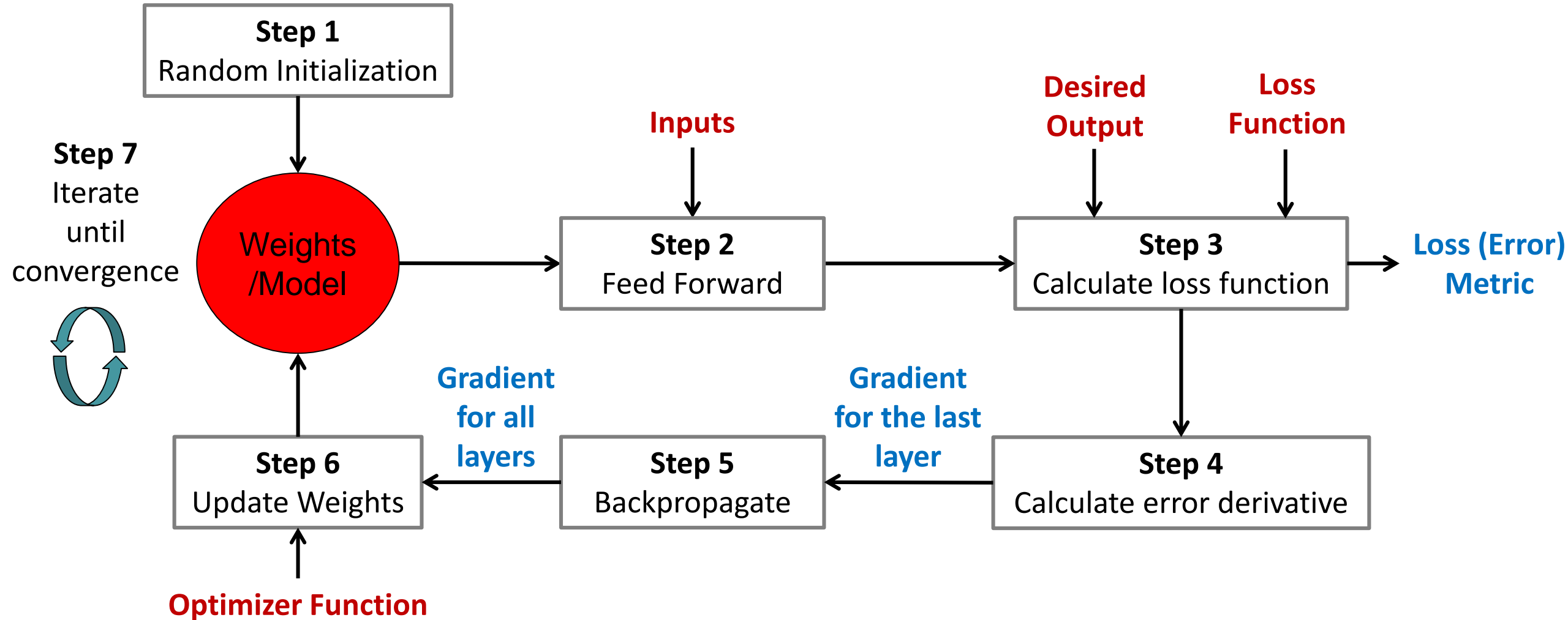
- Algorithm

- Finally, we update the parameter as shown below:

$$\theta_{t+1} = \theta_t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$$

- Adam is a highly popular approach

- Works well in practise, achieves fast convergence, does not suffer from errors faced by other approaches
 - I.e., vanishing learning rate, slow convergence or high variance



- **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 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**

- $LReLU(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
- Range: $-\infty$ to ∞

- **Softmax**

- $S(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$
- Range: 0 to 1

- **Loss function**

- Measures the difference between what we have predicted, \hat{y} , with the model and what it should predicted y .

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

- Use this information to update the model $f_{\theta+1} = f_{\theta}$
- In order to minimize the cost, we prefer differentiable functions which have simpler optimization algorithms
- Note that a loss function always outputs a scalar value.
- This value is a measure of fit of the model with the real value.

- **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**

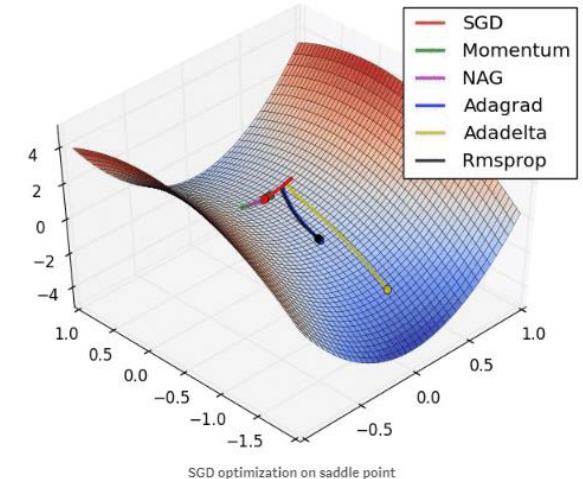
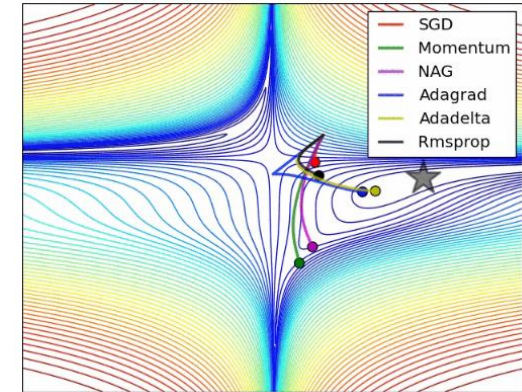
- If there are multiple labels in your data
- Final Activation function: **Sigmoid**
- Loss function: **Binary Cross Entropy**

- **Backpropagation**

- Update weights to minimise loss function
 - This is achieved by taking the gradient of loss function with respect to the weights
- Not a trivial process as neural networks are structured as a series of layers
 - The output is a composite function of the weights, inputs, and activation functions
- **Backpropagation is a tool to calculate the gradient of the loss function with respect to any single weight in the network**

- **Types of Optimizers**

- Momentum
- NesterovAccelerated Gradient (NAG)
- Adagrad—*Adaptive Gradient Algorithm*
- Adadelata
- RMSProp-*Root Mean Square Propagation*
- Adam —*Adaptive Moment Estimation*



- **Gradient Descent**

- Bulk standard method
- Slow to converge, need to heuristically set learning rate

- **Momentum**

- Uses gradient of previous time-step to accelerate convergence of Gradient Descent

- **Nesterov Accelerated Gradient (NAG)**

- Calculates gradient not with respect to the current step but with respect to the future step
- Helps to slow momentum

- **Adagrad Adaptive Gradient Algorithm**
 - An adaptive learning rate method
 - Perform larger updates for infrequent parameters and smaller updates for frequent parameters
 - Disadvantage of a monotonically decreasing learning rate
- **Adadelta and RMSProp**
 - Restrict the window size of accumulated past gradients
 - Weakens effect of decreasing learning rate
- **Adam Adaptive Moment Estimation**
 - Calculates adaptive learning rate from estimates of first and second moments of the gradients