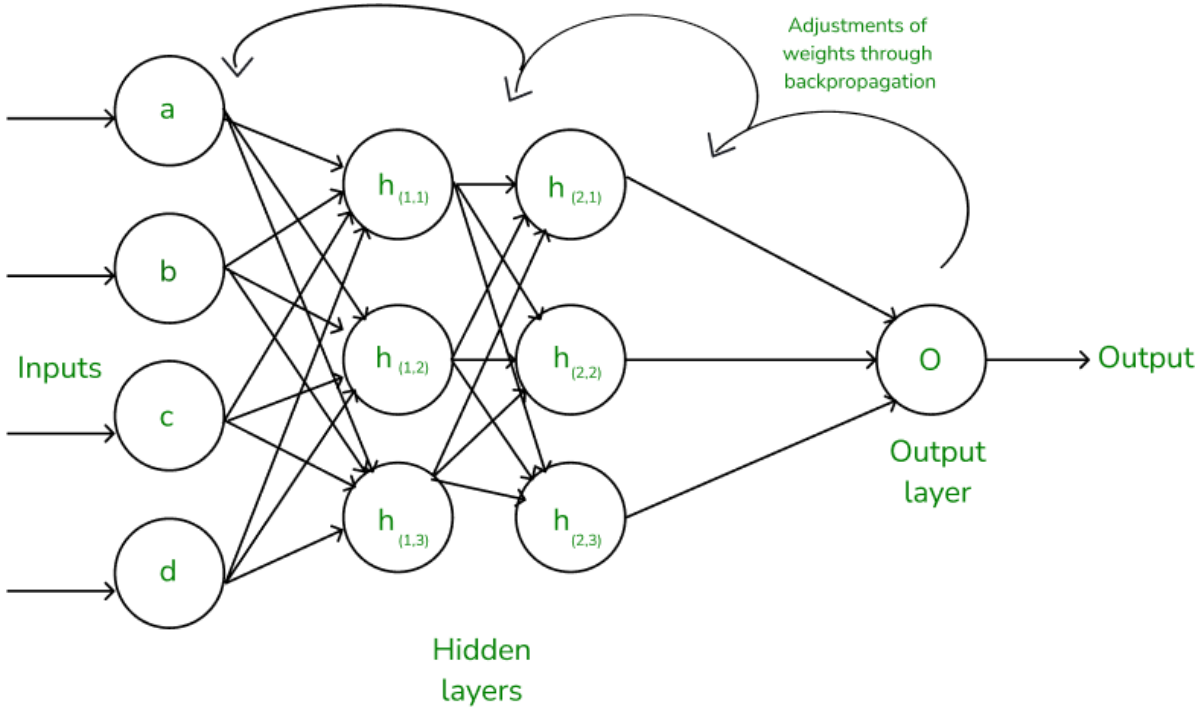
Deep Learning and Applications
UEC630

Backpropagation and Gradient Descent

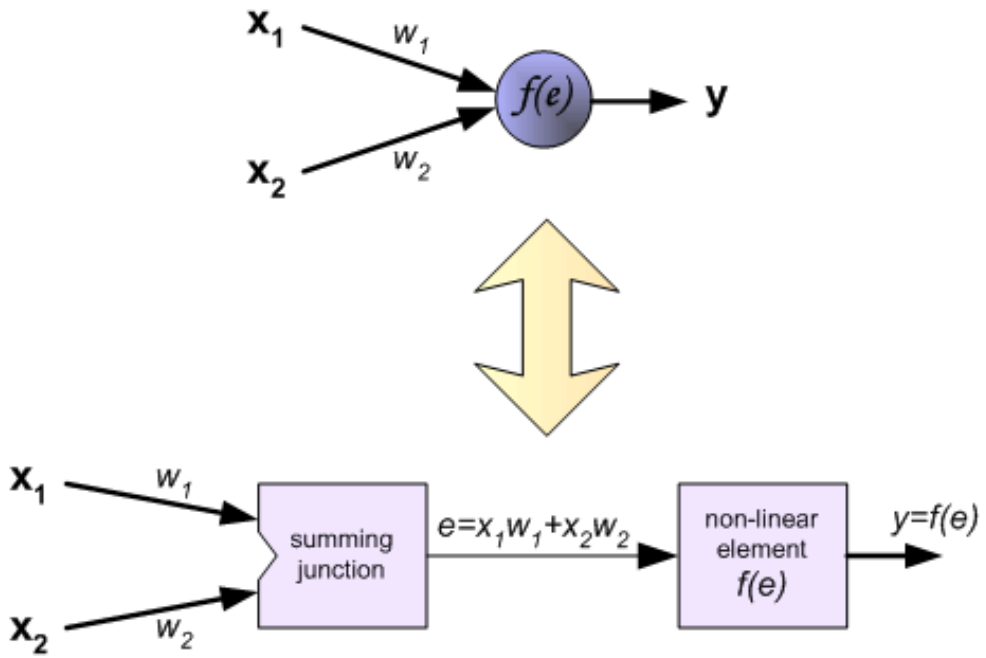
By
Dr. Ashu 

- **What is backpropagation?**

Backpropagation is an iterative algorithm, that helps to **minimize the cost function** by determining which weights and biases should be adjusted.



Backpropagation



Each neuron is composed of two units. First unit adds products of weights coefficients and input signals. The second unit realise nonlinear function, called neuron transfer (activation) function.

Signal *e* is adder output signal, and *y = f(e)* is output signal of nonlinear element. Signal *y* is also output signal of neuron.

Working

- **Forward Pass**: In the forward pass, the input data moves through the network step by step, generating predictions. In each step, a neuron **calculates a weighted sum of its inputs and then uses an activation function to decide its output.** This output gets sent to the next layer for more processing.
- **Loss Computation**: Once the network makes predictions, we compare them with the actual values to compute the loss.

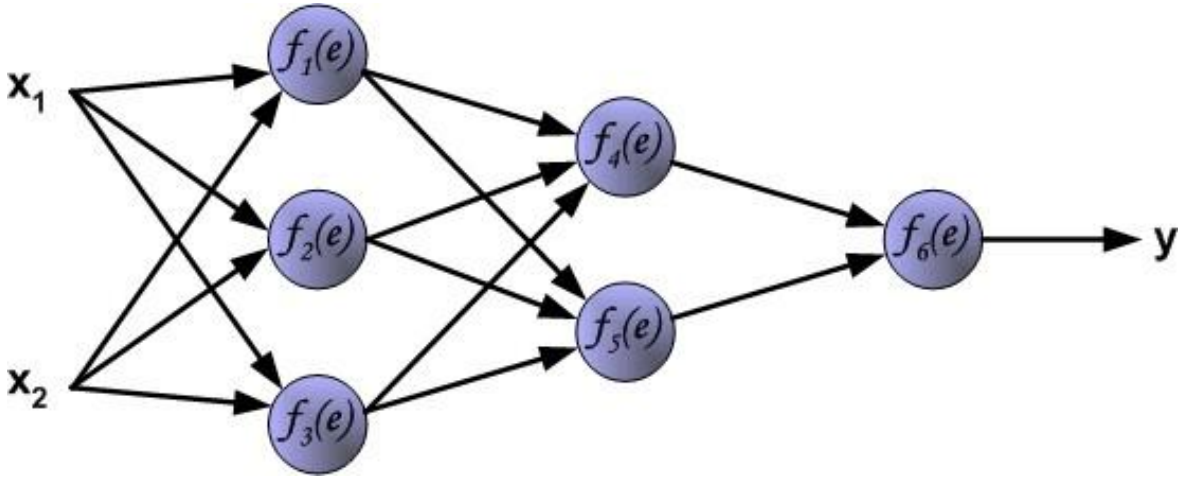
- **Backward Pass**: In the backward pass, we calculate the gradient of the loss function for each parameter **using the chain rule** from calculus.

. Chain rule

$$\frac{\delta f(x)}{\delta x} = \frac{\delta f(x)}{\delta g(x)} \cdot \frac{\delta g(x)}{\delta x}$$

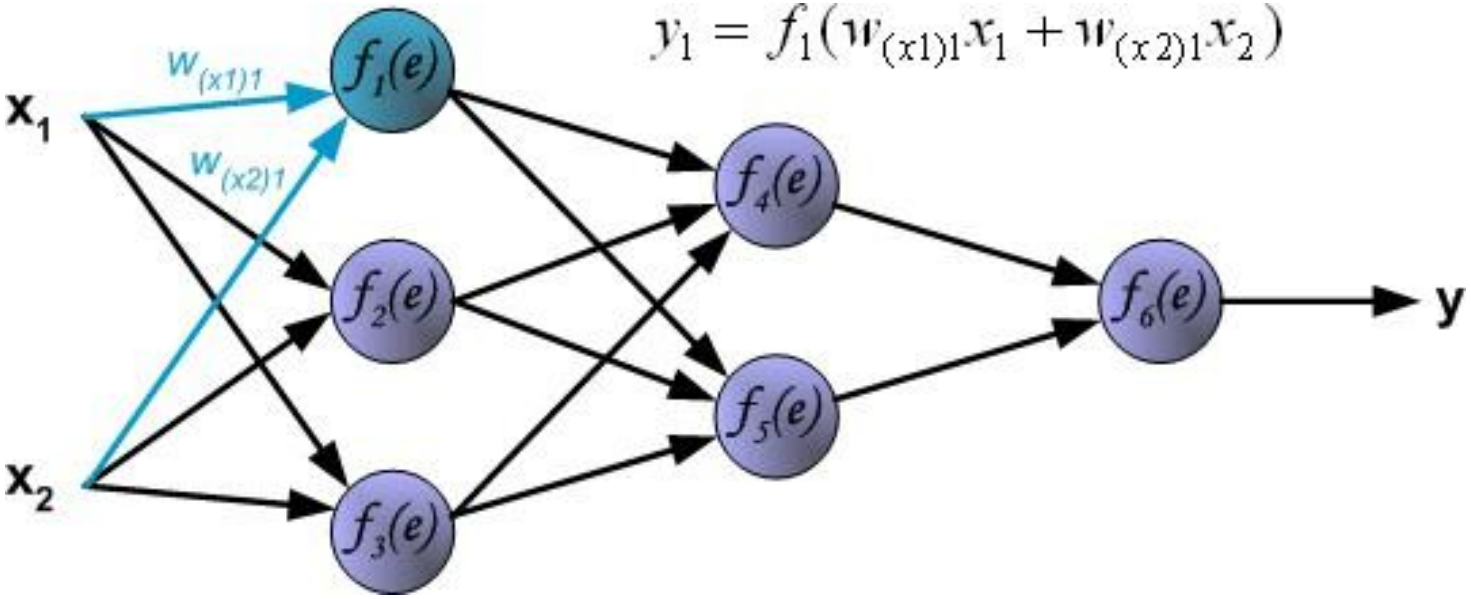
- **Parameter Updates**: With the gradients obtained, we **update the weights and biases** using an optimization algorithm such as stochastic gradient descent (SGD) or its variants.
- **Iteration**: The process of forward pass, backward pass, and parameter updates iterates multiple times (epochs) until the **network converges** where loss is minimized,

Backpropagation



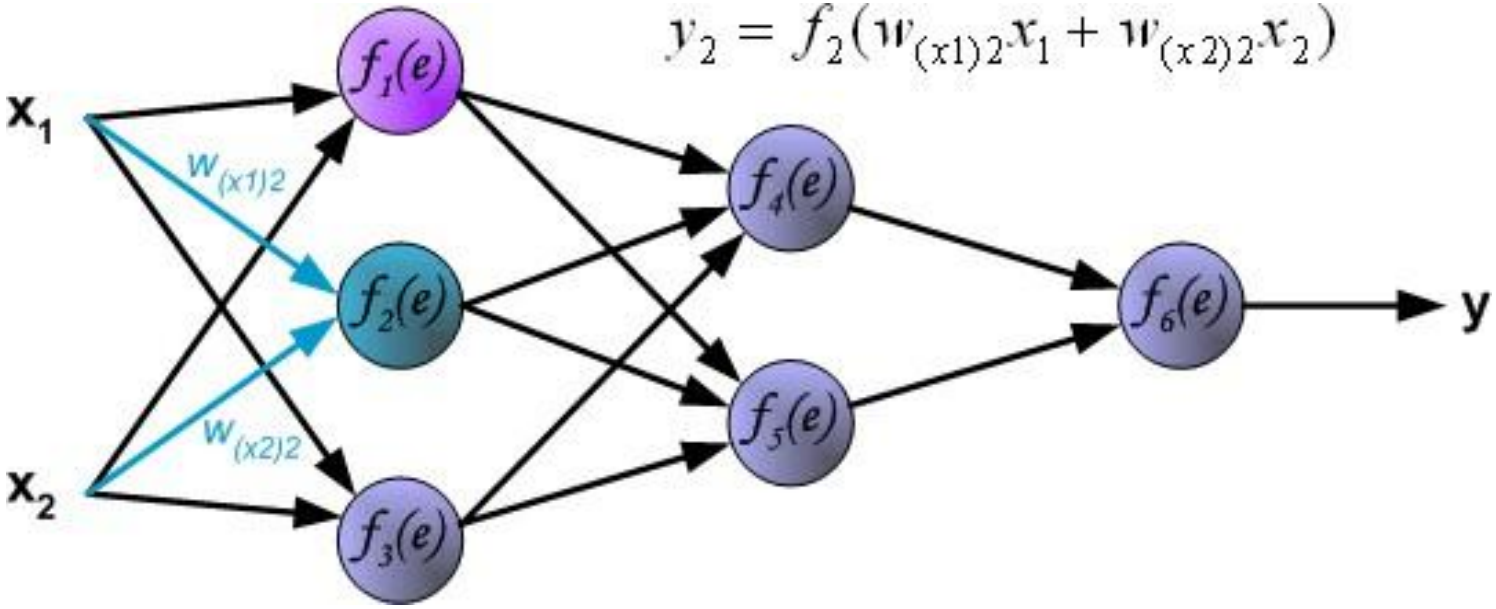
To illustrate this process the three layer neural network with two inputs and one output, which is shown in the picture below, is used:

Backpropagation

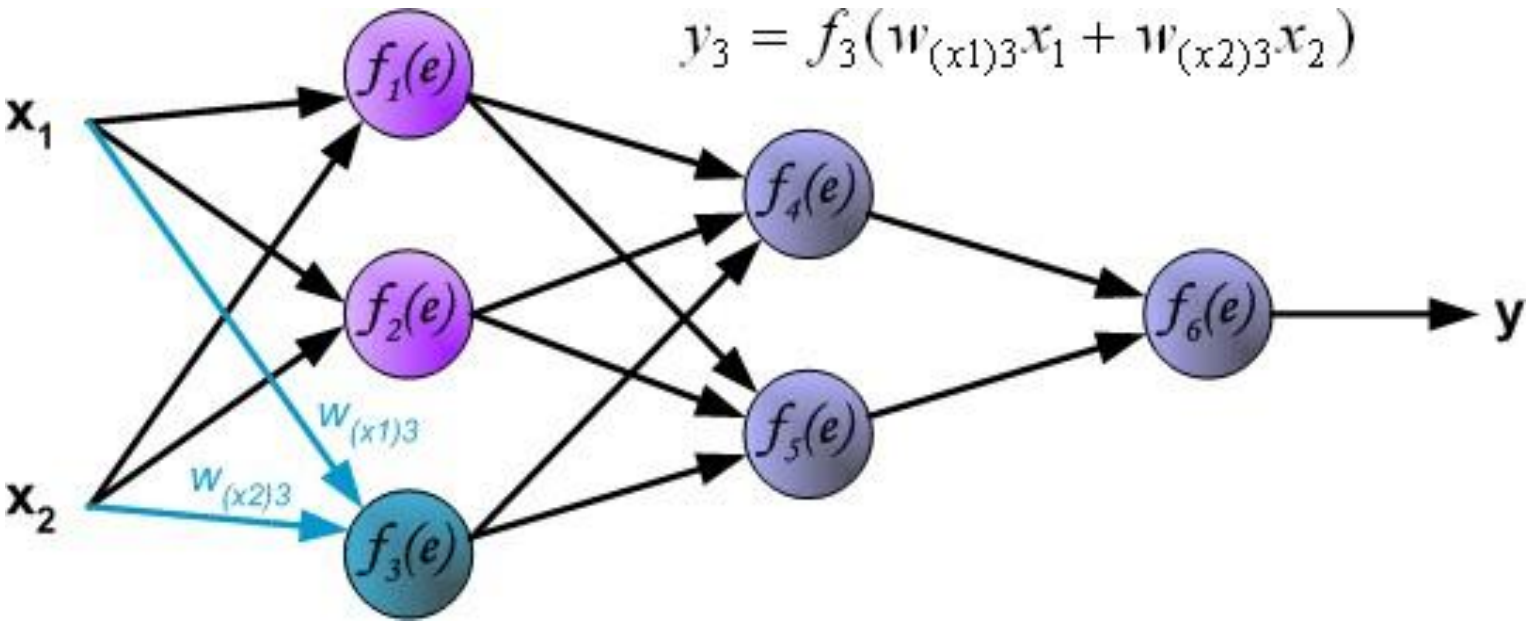


Picture illustrate how signal is propagating through the network, Symbols $w_{(xm)n}$ represent weights of connections between network input x_m and neuron n in input layer. Symbols y_n represents output signal of neuron n.

Backpropagation

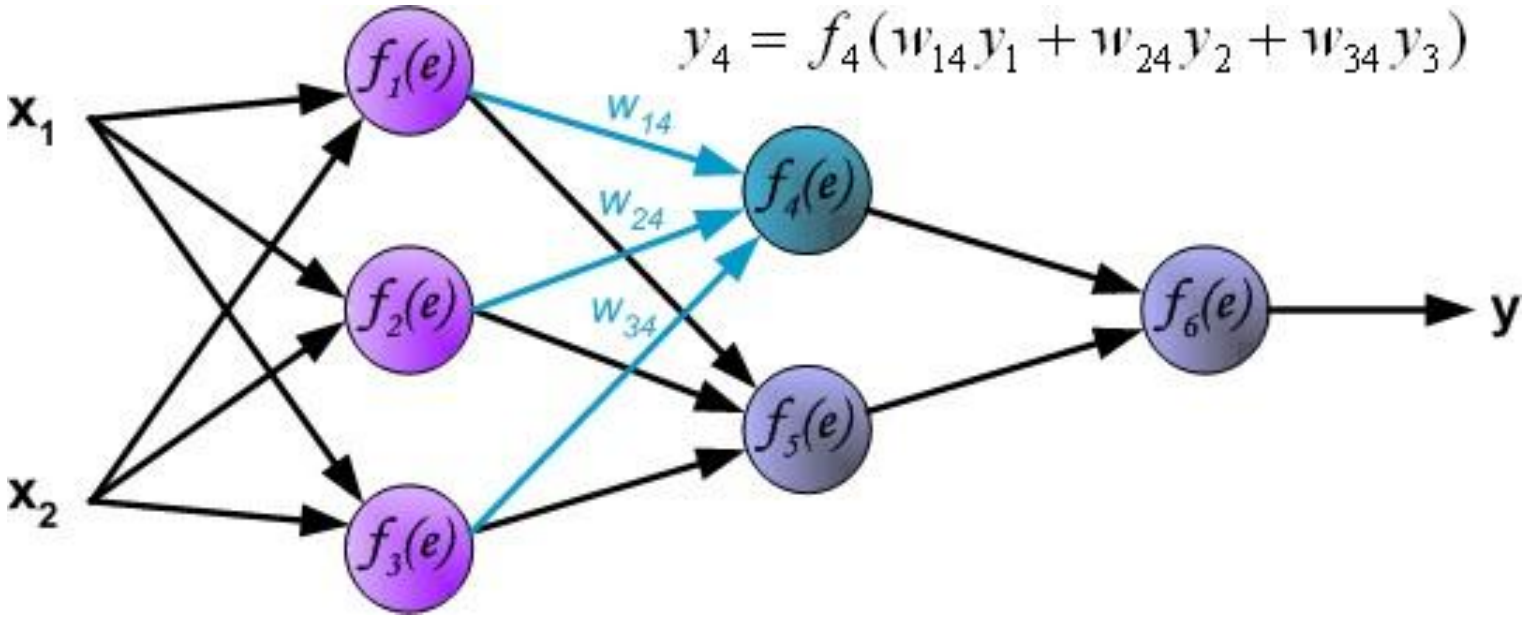


Backpropagation

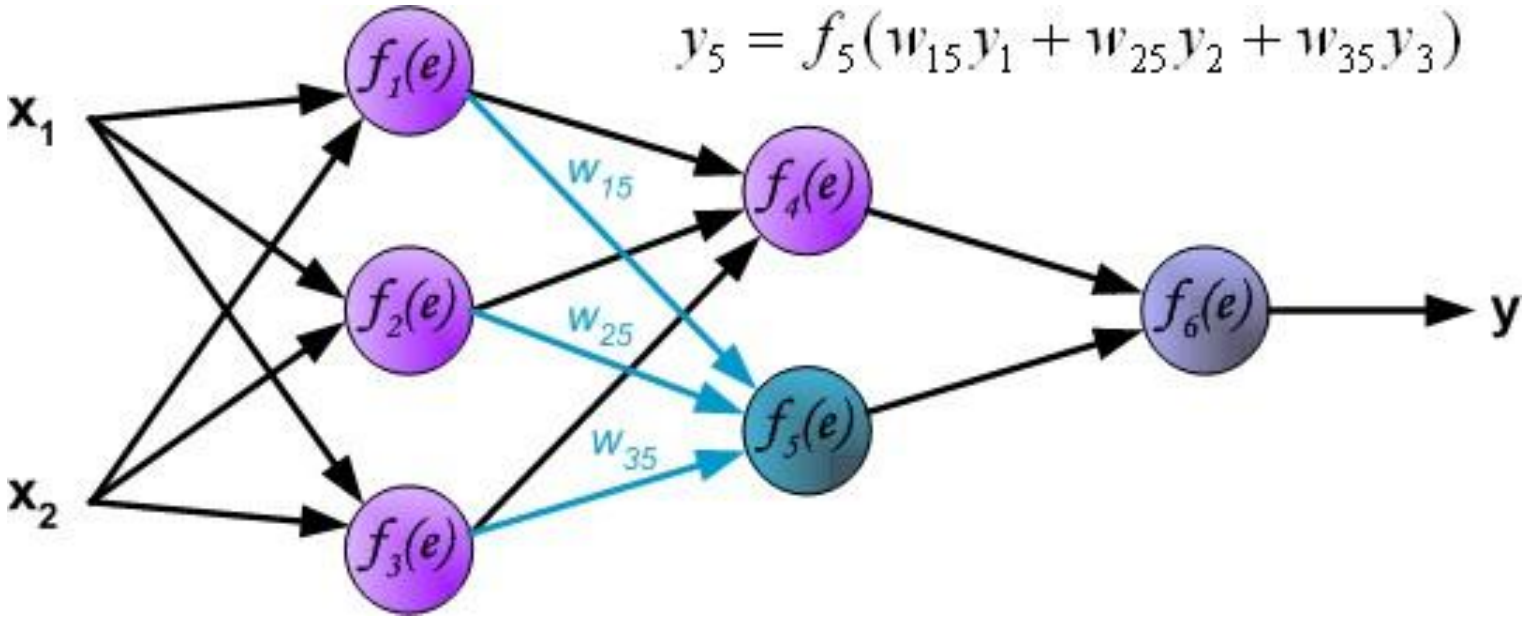


Backpropagation

Propagation of signals through the hidden layer. Symbols w_{mn} represent weights of connections between output of neuron m and input of neuron n in the next layer.

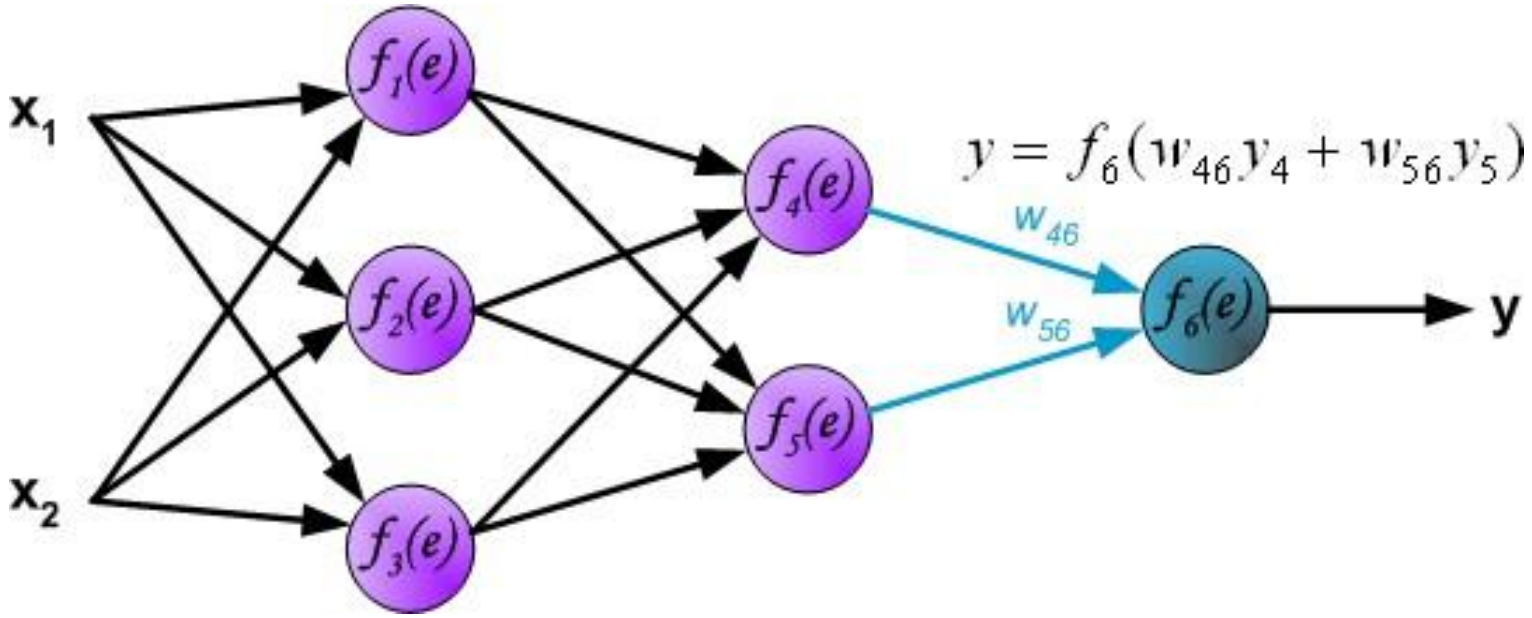


Backpropagation



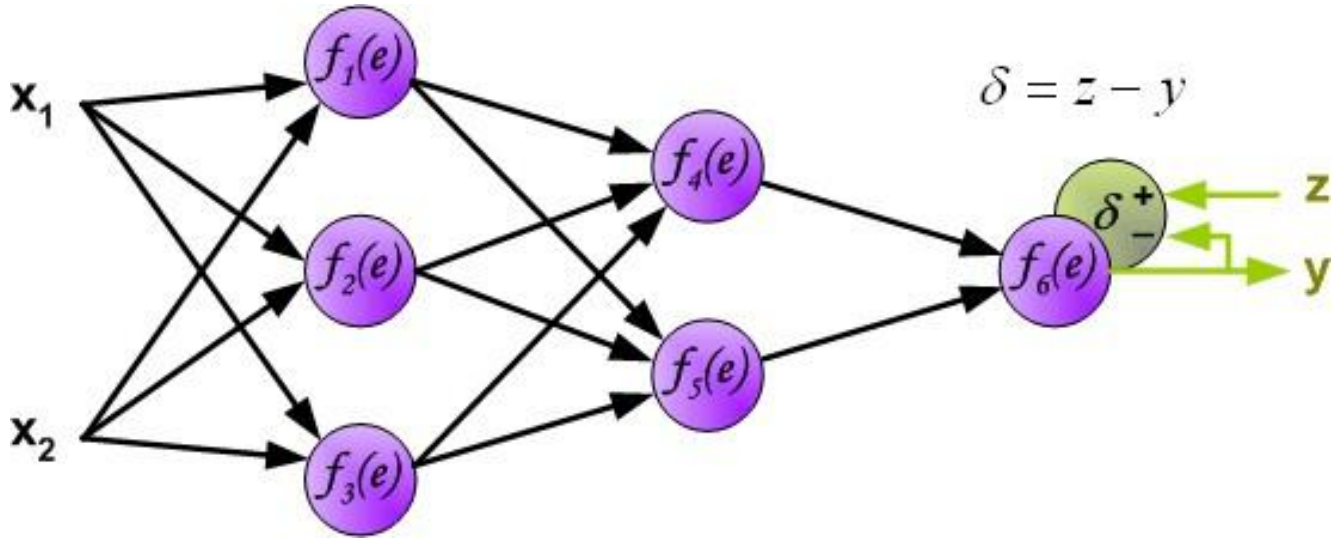
Backpropagation

Propagation of signals through the output layer.

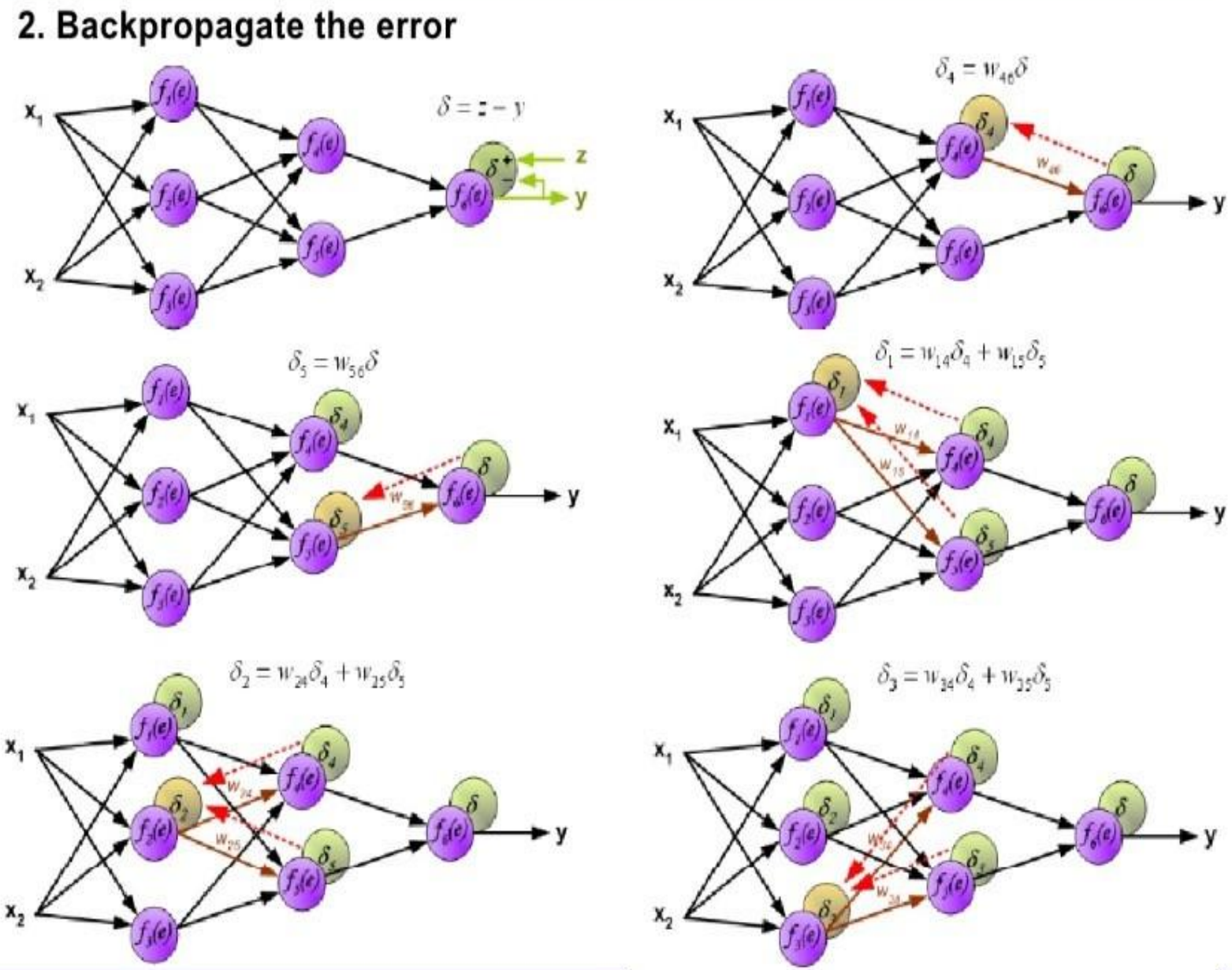


Direction changed Backpropagation

In the next algorithm step the output signal of the network *y* is compared with the desired output value (the target), which is found in training data set. The difference is called error signal *d* of output layer neuron



Back Propagate

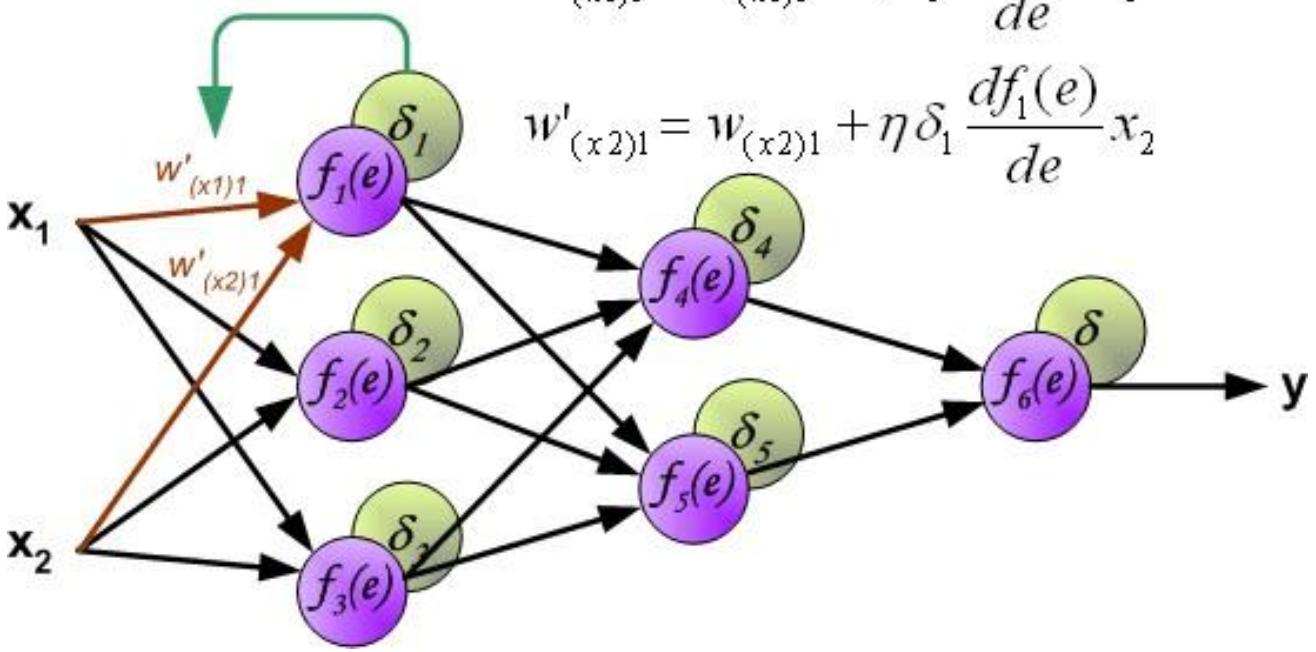


Modify the Weights of Each neuron

When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified).

$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 \frac{df_1(e)}{de} x_1$$

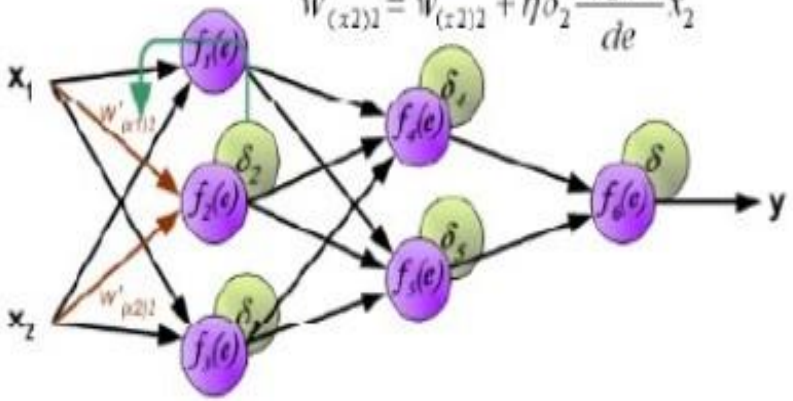
$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 \frac{df_1(e)}{de} x_2$$



3.bis. Do the same of each neuron

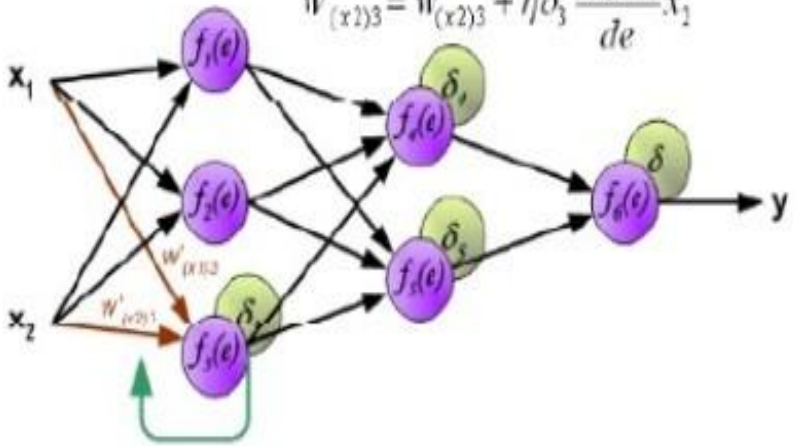
$$w'_{(x1)2} = w_{(x1)2} + \eta\delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta\delta_2 \frac{df_2(e)}{de} x_2$$



$$w'_{(x1)3} = w_{(x1)3} + \eta\delta_3 \frac{df_3(e)}{de} x_1$$

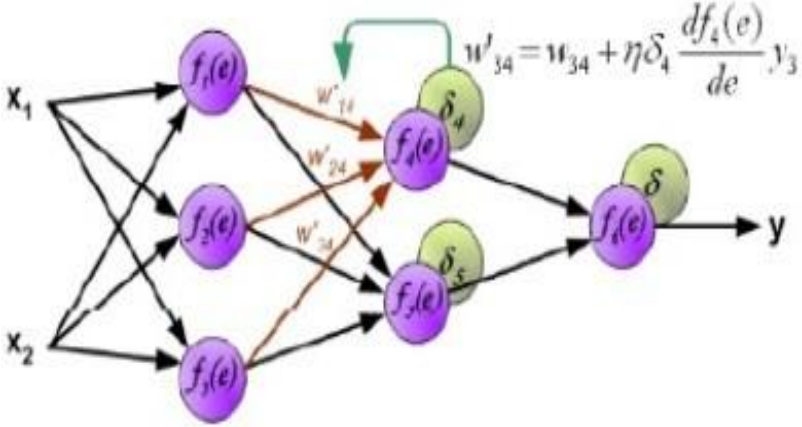
$$w'_{(x2)3} = w_{(x2)3} + \eta\delta_3 \frac{df_3(e)}{de} x_2$$



$$w'_{14} = w_{14} + \eta\delta_4 \frac{df_4(e)}{de} y_1$$

$$w'_{24} = w_{24} + \eta\delta_4 \frac{df_4(e)}{de} y_2$$

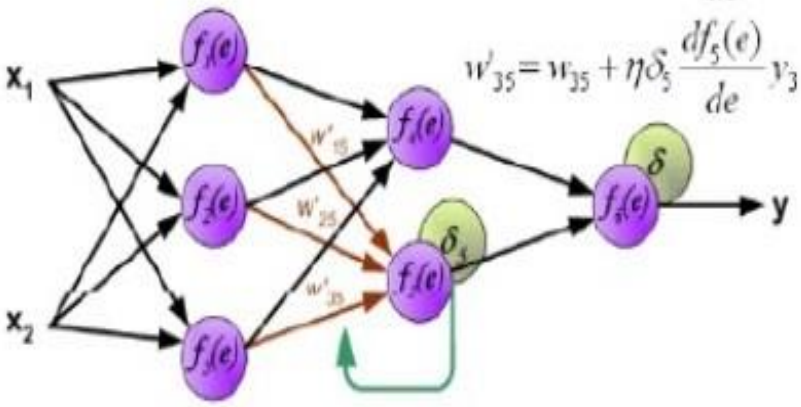
$$w'_{34} = w_{34} + \eta\delta_4 \frac{df_4(e)}{de} y_3$$



$$w'_{15} = w_{15} + \eta\delta_5 \frac{df_5(e)}{de} y_1$$

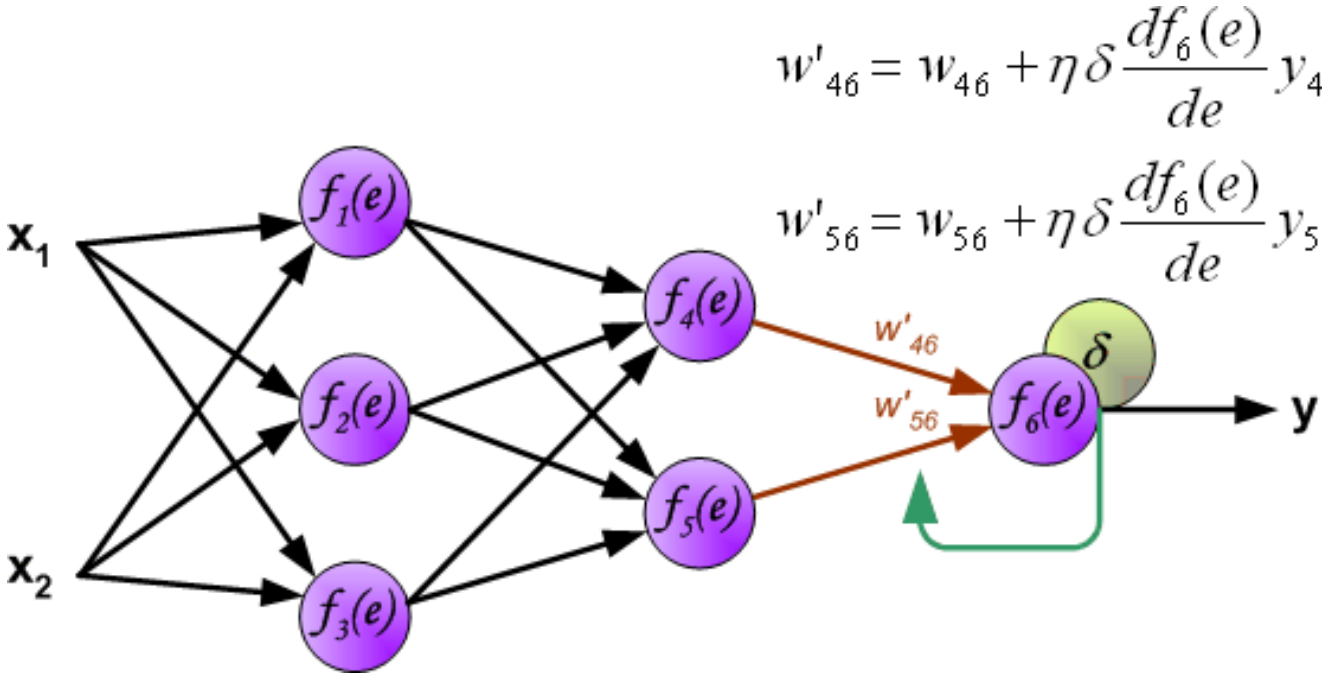
$$w'_{25} = w_{25} + \eta\delta_5 \frac{df_5(e)}{de} y_2$$

$$w'_{35} = w_{35} + \eta\delta_5 \frac{df_5(e)}{de} y_3$$



Cont..

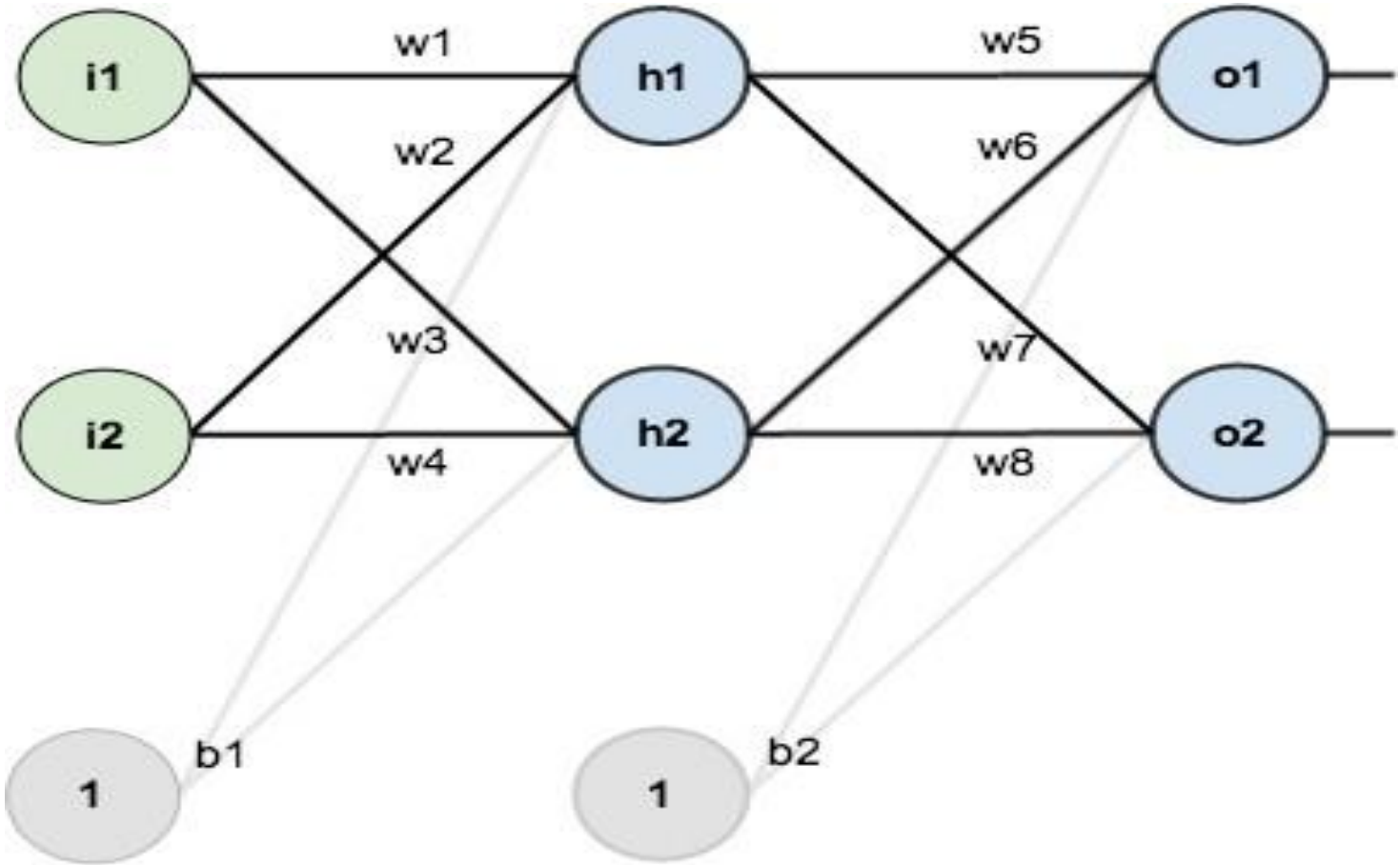
When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified).

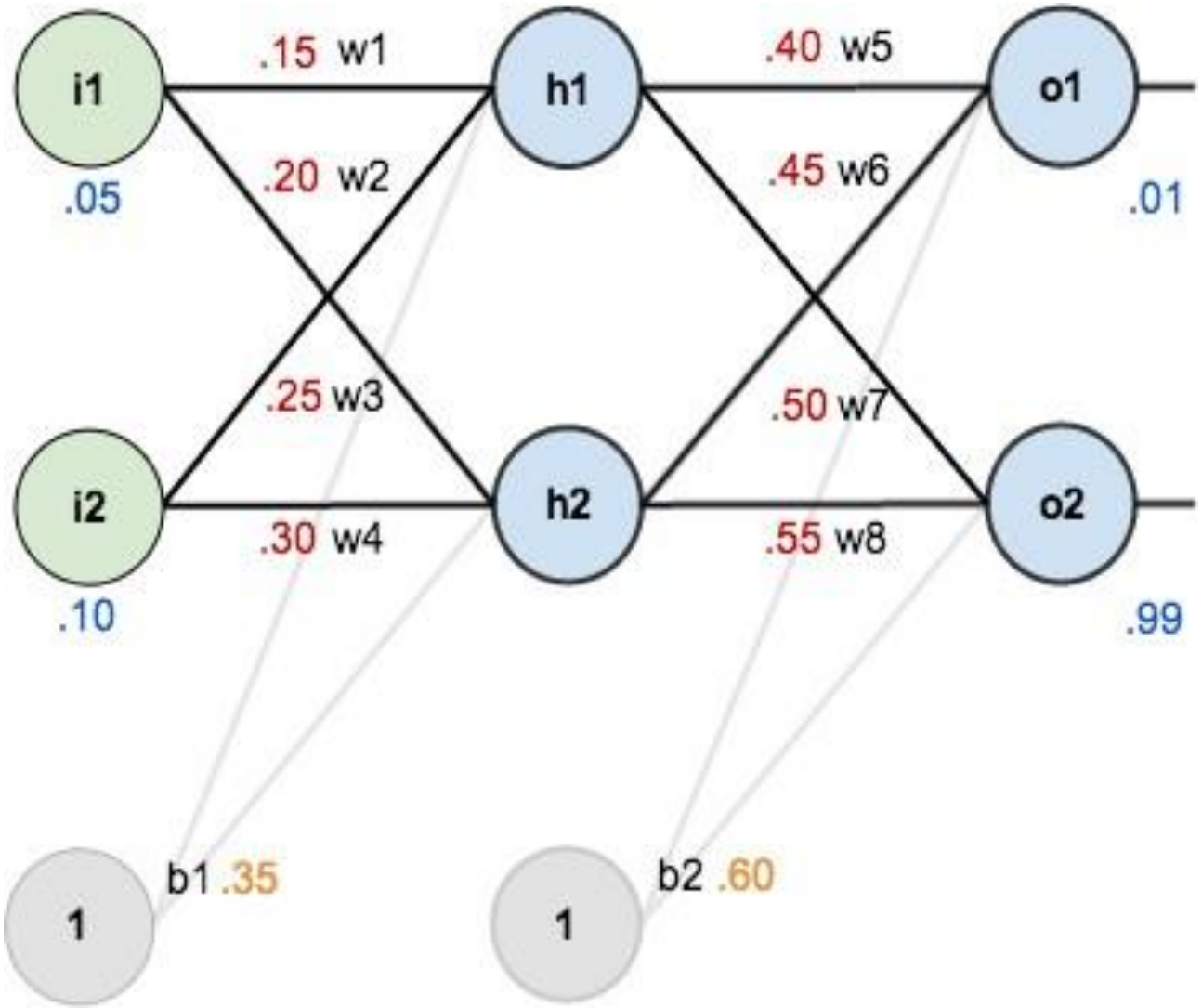


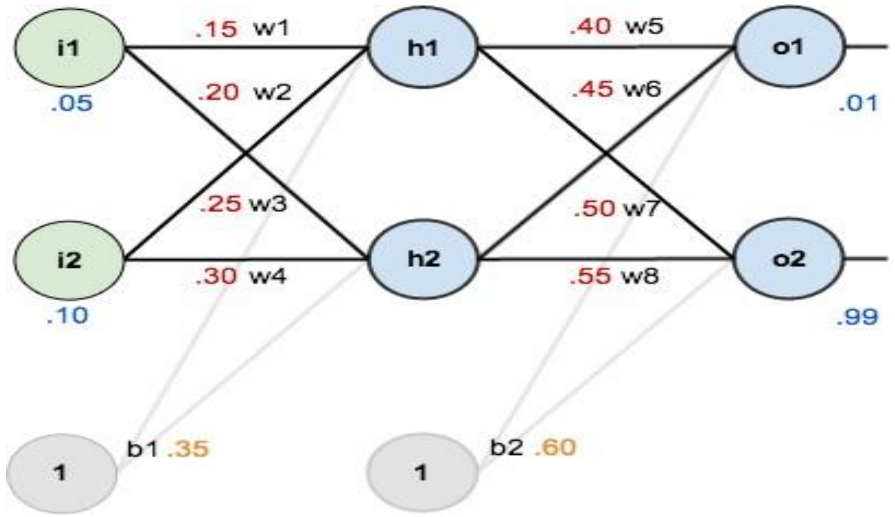
Backpropagation Algorithm

1. Initialize the weights of the network randomly.
2. Forward propagate an input through the network to get the predicted output.
3. Compute the error between the predicted output and the actual output.
4. Backward propagate the error through the network to compute the gradient of the loss function with respect to each weight.
5. Update the weights in the opposite direction of the gradient using an optimization algorithm such as Stochastic Gradient Descent (SGD).
6. Repeat steps 2–5 for multiple iterations until the weights converge.

Example




i1
i2
h1
h2
o1
o2
.15 w1
.20 w2
.25 w3
.30 w4
.40 w5
.45 w6
.50 w7
.55 w8
.05
.10
.01
.99
1
b1 .35
1
b2 .60



$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

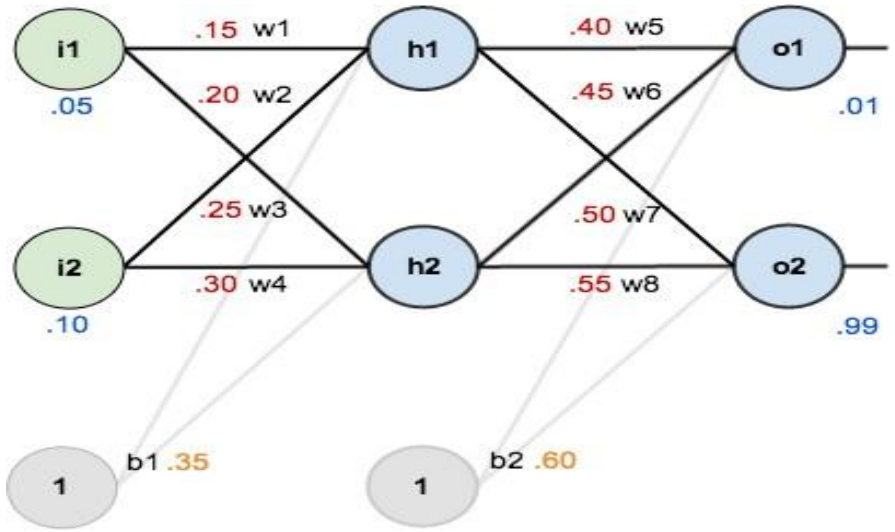
$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

We then squash it using the logistic function to get the output of h_1:

$$out_{h1} = \frac{1}{1+e^{-netₕ1}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

Carrying out the same process for h_2 we get:

$$out_{h2} = 0.596884378$$



Here's the output for o_1:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

And carrying out the same process for o_2 we get:

$$out_{o2} = 0.772928465$$

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

Repeating this process for o_2 (remembering that the target is 0.99) we get:

$$E_{o2} = 0.023560026$$

The total error for the neural network is the sum of these errors:

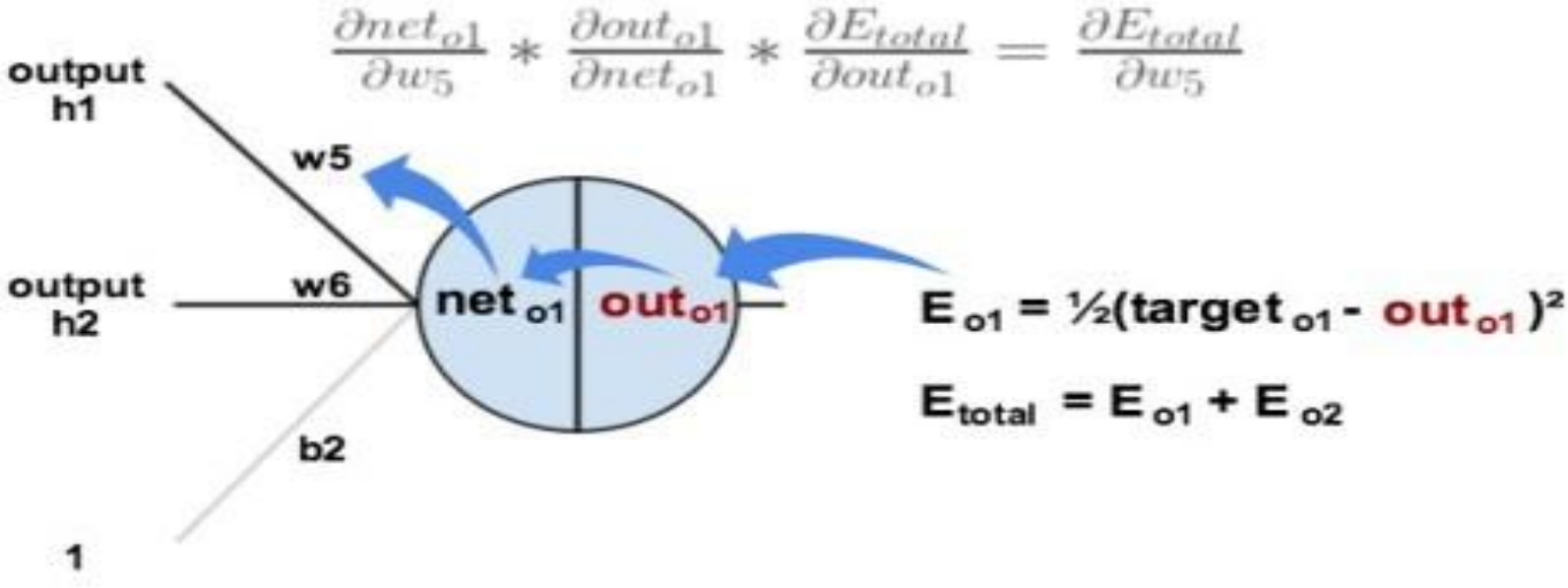
$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

The Backwards Pass, Output Layer

By applying the chain rule we know that:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

Visually, here's what we're doing:



To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5):

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

> Some sources use α (alpha) to represent the learning rate, others use η (eta), and others even use ϵ (epsilon).

We can repeat this process to get the new weights w_6, w_7, and w_8:

$$w_6^+ = 0.408666186$$

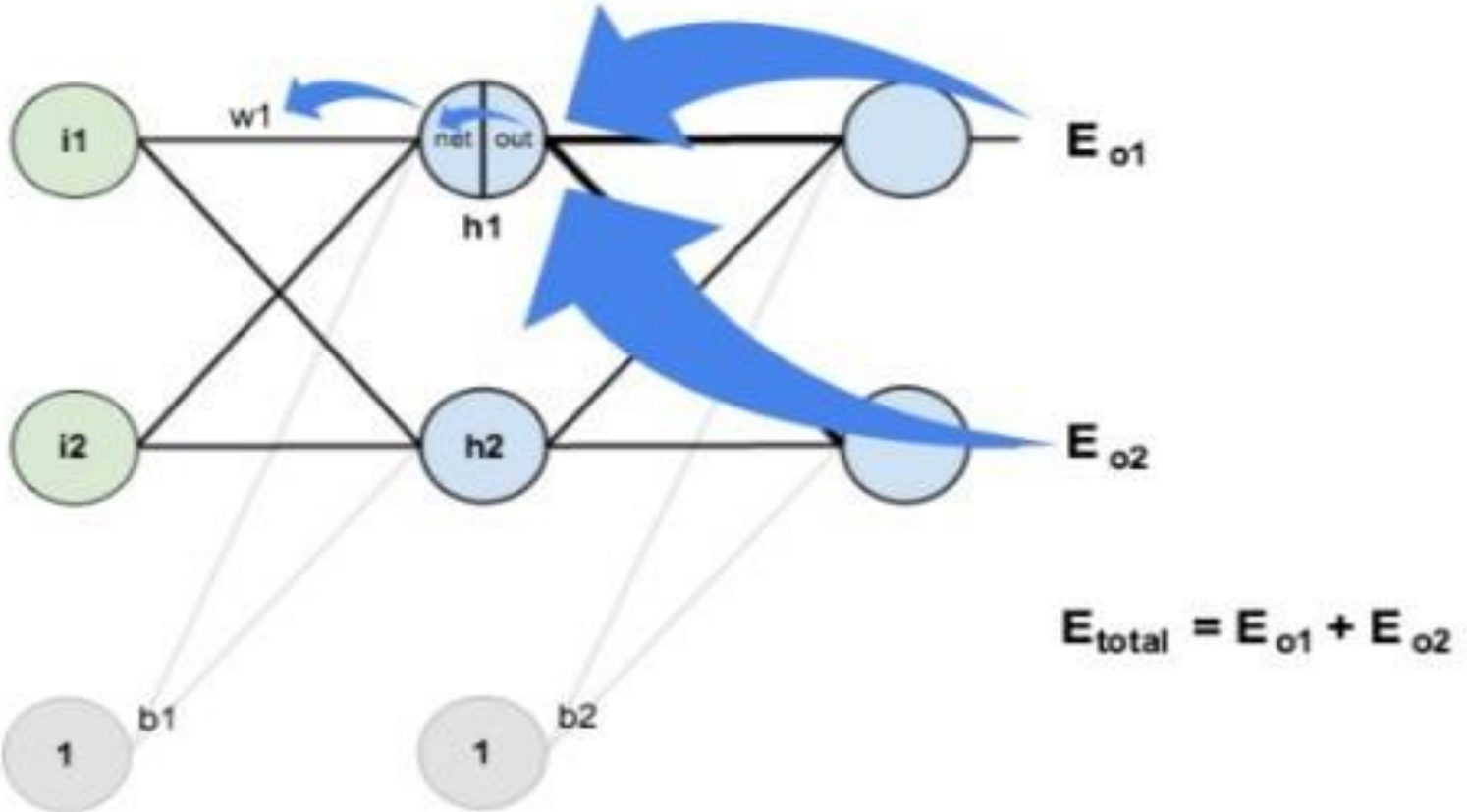
$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

Hidden Layer

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



We can now update w_1:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

Repeating this for w_2, w_3, and w_4

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

Finally, we've updated all of our weights!
Process will cont.. Until error minimization.

Benefits of Backpropagation

- It is a powerful optimization algorithm that can efficiently train complex neural networks.
- It can handle large amounts of data and can learn complex patterns.
- It is flexible and can be applied to various neural network architectures.

Demerits

- Slow and inefficient.
  Can get stuck in local minima resulting in sub-optimal solutions .
- A large amount of input/output data is available, but you're not sure how to relate it to the output.
- The problem appears to have overwhelming complexity, but there is clearly a solution.
- It is easy to create a number of examples of the correct behaviour.
- The solution to the problem may change over time, within the bounds of the given input and output parameters (i.e., today 2+2=4, but in the future we may find that 2+2=3.8).
- Outputs can be "fuzzy", or non-numeric.

Applications of Backpropagation

- Image and speech recognition
- Natural language processing
- Fraud detection
- Medical diagnosis
- Stock market prediction

Gradient-based Learning

Definition

- A gradient is nothing but a derivative that defines the effects on outputs of the function with a little bit of variation in inputs.

- It is a numerical optimization algorithm that aims to find the optimal parameters—weights and biases—of a neural network by minimizing a defined cost function.

- Cost function measures the performance of a machine learning model for a data set. Cost function quantifies the error between predicted and expected values and presents that error in the form of a single real number.

- **Gradient descent** (GD) is an iterative first-order optimisation algorithm, used to find a local minimum/maximum of a given function.

- This method is commonly used in *machine learning* (ML) and *deep learning* (DL) to minimise a cost/loss function .

Gradient Descent

Gradient Descent Algorithm:

- Pick an initial point x_0
- Iterate until convergence

$$x_{t+1} = x_t - \gamma_t \nabla f(x_t)$$

where γ_t is the t^{th} step size (sometimes called learning rate)

Possible Stopping Criteria: iterate until
$\|\nabla f(x_t)\| \leq \epsilon$ for some $\epsilon > 0$

How small should ϵ be?

- In machine learning, it is called **learning rate** and have a strong influence on performance.
- The smaller learning rate the longer GD converges, or may reach maximum iteration before reaching the optimum point
- If learning rate is too big the algorithm may not converge to the optimal point (jump around) or even to diverge completely.

- Gradient Descent method's steps are:

1. choose a starting point (initialisation)
2. calculate gradient at this point
3. make a scaled step in the opposite direction to the gradient (objective: minimise)
4. repeat points 2 and 3 until one of the criteria is met:

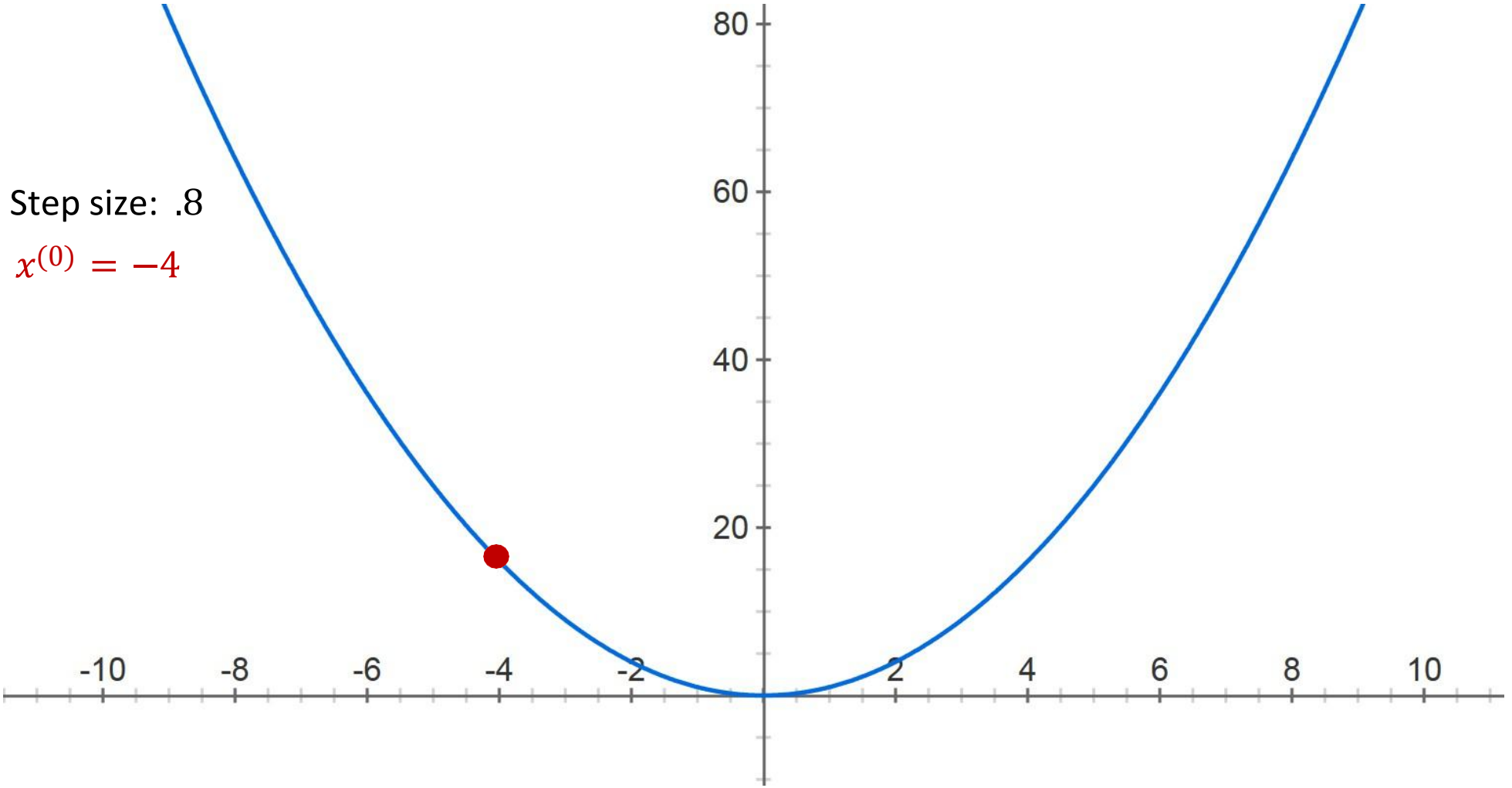
- maximum number of iterations reached
- step size is smaller than the tolerance (due to scaling or a small gradient).

Gradient Descent

$f(x) = x^2$

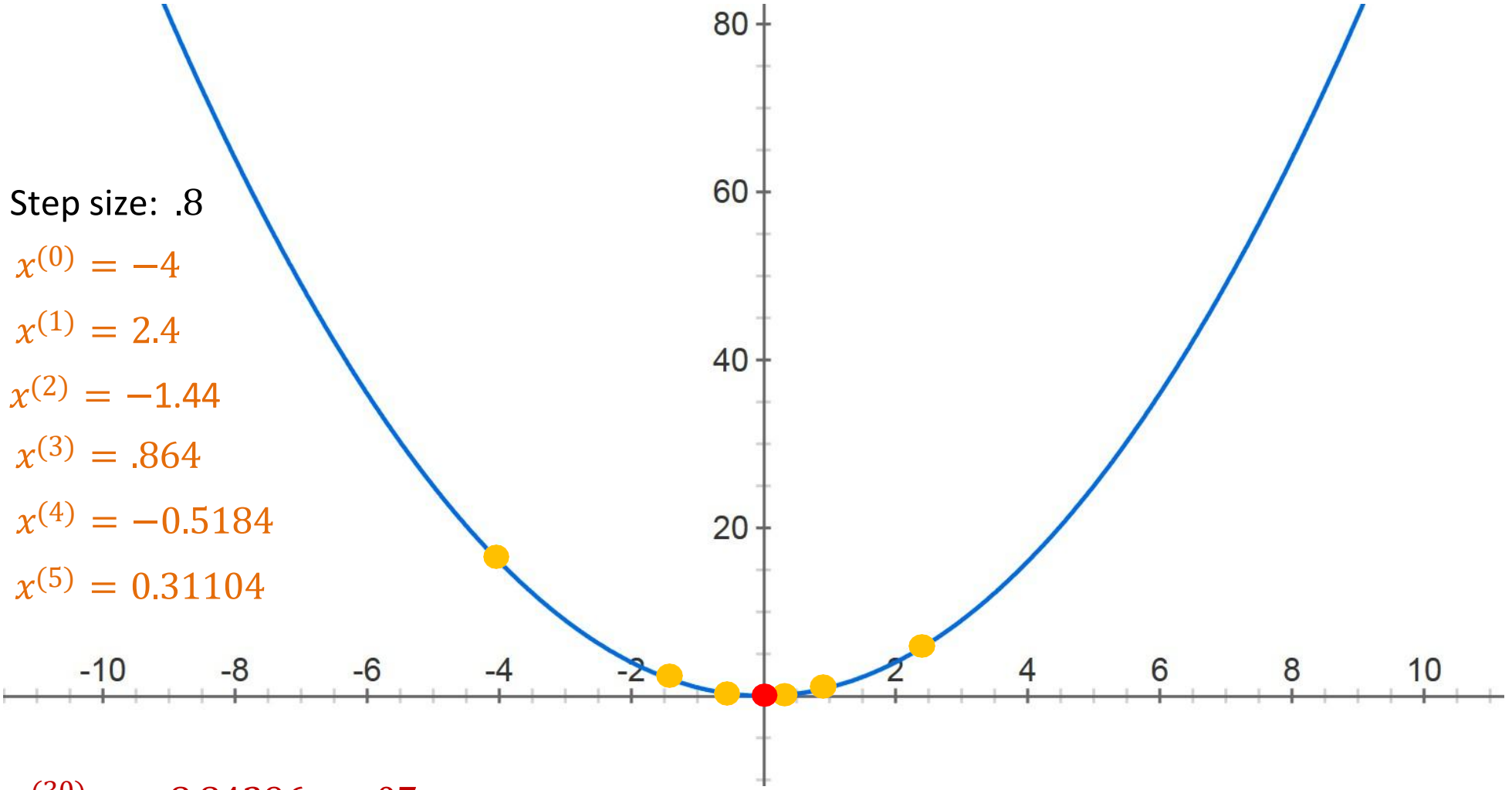
Step size: .8

$x^{(0)} = -4$



Gradient Descent

$f(x) = x^2$



Step size: .8

$x^{(0)} = -4$

$x^{(1)} = 2.4$

$x^{(2)} = -1.44$

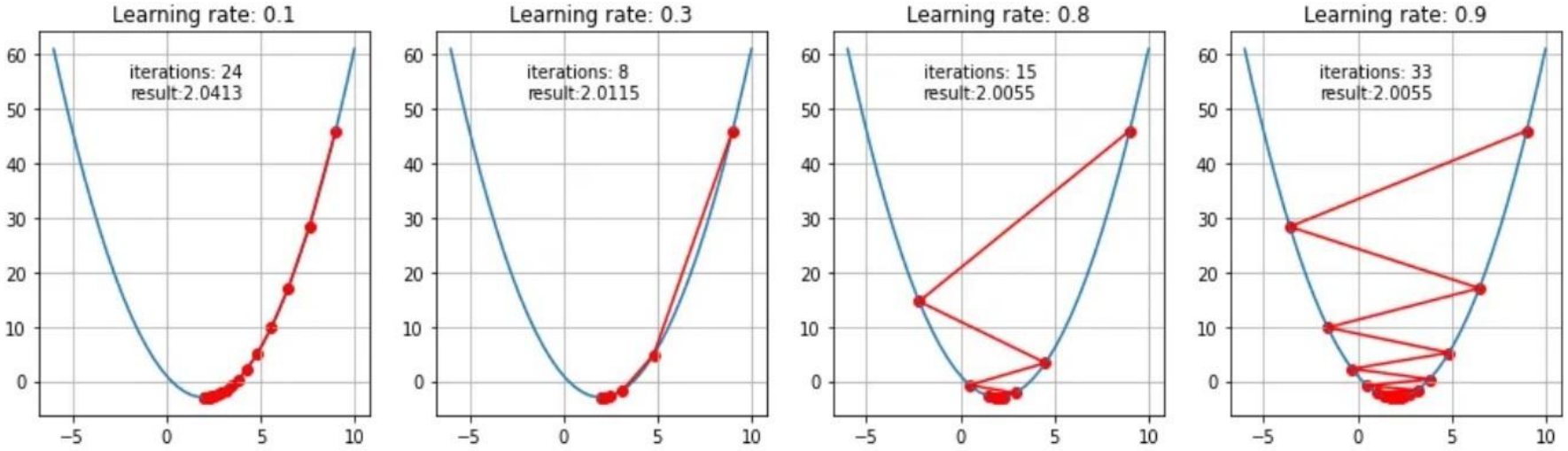
$x^{(3)} = .864$

$x^{(4)} = -0.5184$

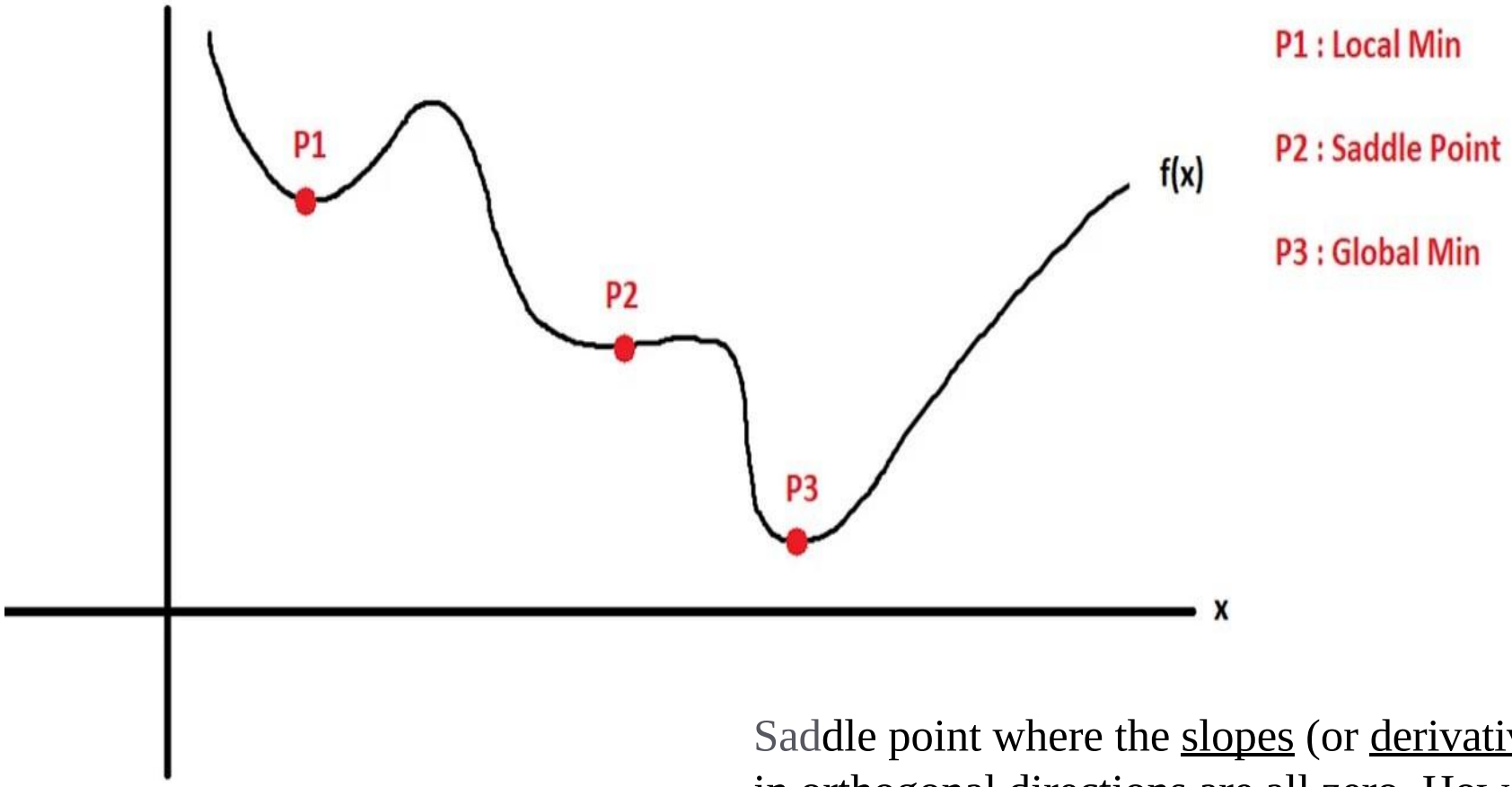
$x^{(5)} = 0.31104$

$x^{(30)} = -8.84296e - 07$

```
import numpy as np



def gradient_descent(start: float, gradient: Callable[[float], float],
learn_rate: float, max_iter: int, tol: float = 0.01):
x = start
steps = [start] # history tracking

for _ in range(max_iter):
diff = learn_rate*gradient(x)
if np.abs(diff) < tol:
break
x = x - diff
steps.append(x) # history tracing

return steps, x
```


Learning rate: 0.1
iterations: 24
result:2.0413
Learning rate: 0.3
iterations: 8
result:2.0115
Learning rate: 0.8
iterations: 15
result:2.0055
Learning rate: 0.9
iterations: 33
result:2.0055

Loss Surface



Saddle point where the slopes (or derivatives) in orthogonal directions are all zero. However, the point is not the highest or lowest point in its neighborhood.

When selecting an optimization algorithm, it is essential to consider whether the loss function is **convex or non-convex.**

A **convex loss function** has only one global minimum and no local minima, making it easier to solve with a simpler optimization algorithm.

However, **a non-convex loss function** has both local and global minima and requires an advanced optimization algorithm to find the global minimum.

Momentum

Momentum in neural networks is a parameter optimization technique that accelerates gradient descent by adding a fraction of the previous update to the current update.

It helps overcome some of the limitations of standard gradient descent optimization methods, such as slow convergence and oscillations around local minima.

Stochastic Gradient(Steepest) Descent (SGD)

Stochastic Gradient Descent (SGD) is a variant of the Gradient Descent algorithm that is used for optimizing machine learning models.

It addresses the computational inefficiency of traditional Gradient Descent methods when dealing with large datasets in machine learning projects.

In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model parameters.

Stochastic Gradient Descent Algorithm

•Initialization: Randomly initialize the parameters of the model.

•Set Parameters: Determine the number of iterations and the learning rate (alpha) for updating the parameters.

•Stochastic Gradient Descent Loop: Repeat the following steps until the model converges or reaches the maximum number of iterations:

- Shuffle the training dataset to introduce randomness.
- Iterate over each training example (or a small batch) in the shuffled order.
- Compute the gradient of the cost function with respect to the model parameters using the current training example (or batch).
- Update the model parameters by taking a step in the direction of the negative gradient, scaled by the learning rate.
- Evaluate the convergence criteria, such as the difference in the cost function between iterations of the gradient.

•Return Optimized Parameters: Once the convergence criteria are met or the maximum number of iterations is reached, return the optimized model parameters.

Batch Gradient Descent

Batch gradient descent (BGD) is used to find the error for each point in the training set

Update the model after evaluating all training examples.

In simple words, it is a greedy approach where we have to sum over all examples for each update.

MiniBatch Gradient Descent

It divides the training datasets into small batch sizes then performs the updates on those batches separately.

Splitting training datasets into smaller batches make a balance to maintain the computational efficiency of batch gradient descent and speed of stochastic gradient descent.

Challenges with the Gradient Descent

Vanishing Gradients

Exploding Gradients

Vanishing Gradient occurs when the gradient is smaller than expected. During backpropagation, this gradient becomes smaller that causing the decrease in the learning rate of earlier layers than the later layer of the network.

it occurs when the Gradient is too large and creates a stable model. Further, in this scenario, model weight increases, and they will be represented as NaN.

Differences between Backpropagation and Gradient Descent

POINT	BACKPROPAGATION	GRADIENT DESCENT
Definition	the algorithm that is used to calculate the gradient of the loss function with respect to parameters of the neural network.	the optimisation algorithm that is used to find parameters that minimise the loss function.
Dependencies	Calculus and Chain rules	Backpropagation
Type of method	This is not a learning method	first order optimizer
Input size	Calculate the gradient for each single input-output example	may be Stochastic, batch or mini-batch
Sequence	come first	must be after backpropagation