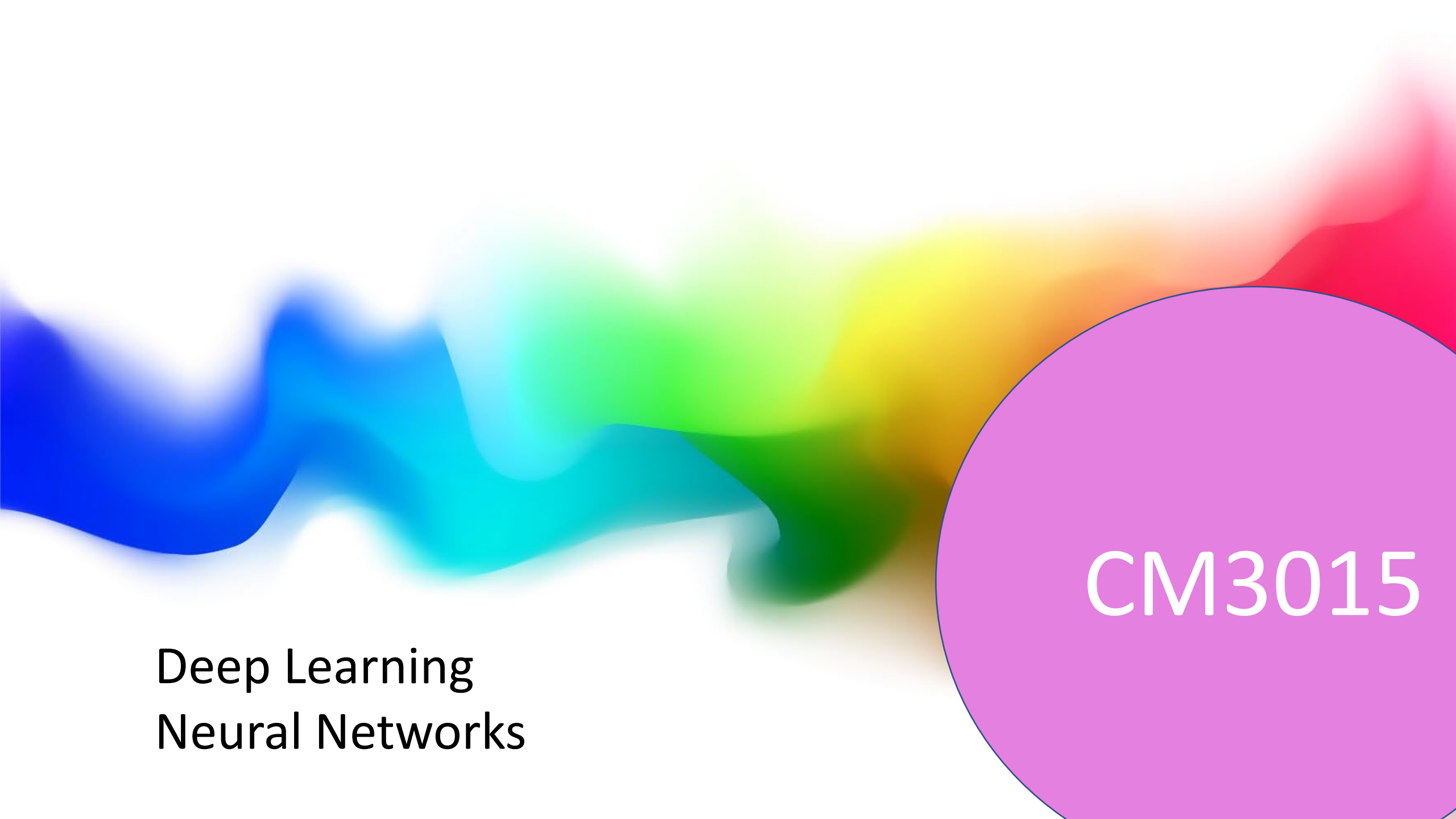


Deep Learning
Neural Networks

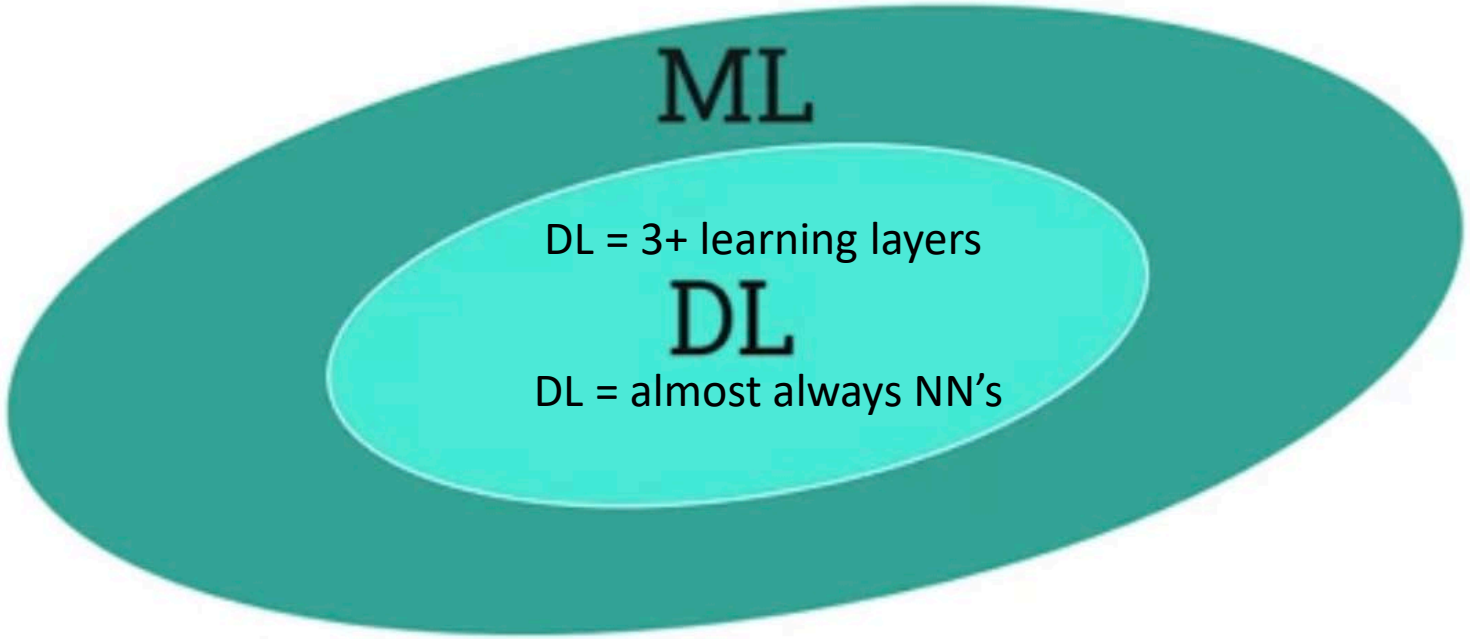
AI = automate tasks which human beings can do

AI

Symbolic AI

Symbolic AI = mechanically automate certain known operations. Thinking comes from programmer, originated by Ada Lovelace in 1830's and 1840's. Peaked in 1980's with Expert Systems.

ML = algorithms are trained rather than specifically programmed. Presented with many examples relevant to a task, the algorithm finds a statistical structure which enables it to formulate rules for automating the task. Usually 1-2 layers.



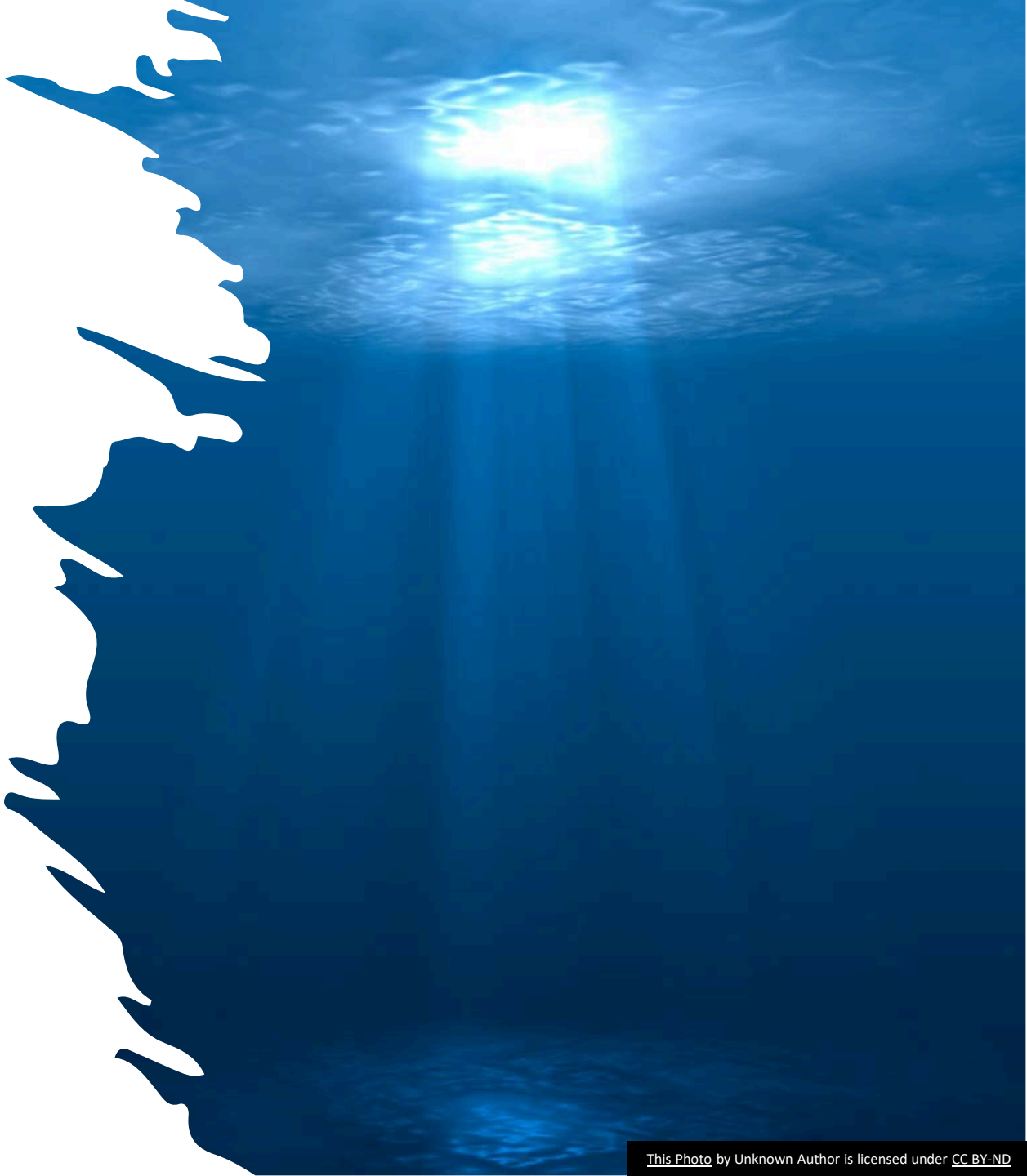
Shallow vs deep learning

Shallow learning = transforming the data into one or two successive representation spaces, usually using a kernel method or decision trees.

Data pipelines have to manually pre-process data, a process called *feature engineering*.

Deep learning completely automates this step. You learn features in one pass. Workflows can then replace a multi-stage pipeline with a simple end-to-end deep-learning model.

F Chollet, “Deep Learning with Python”, Manning Publications, 2018 p. 18



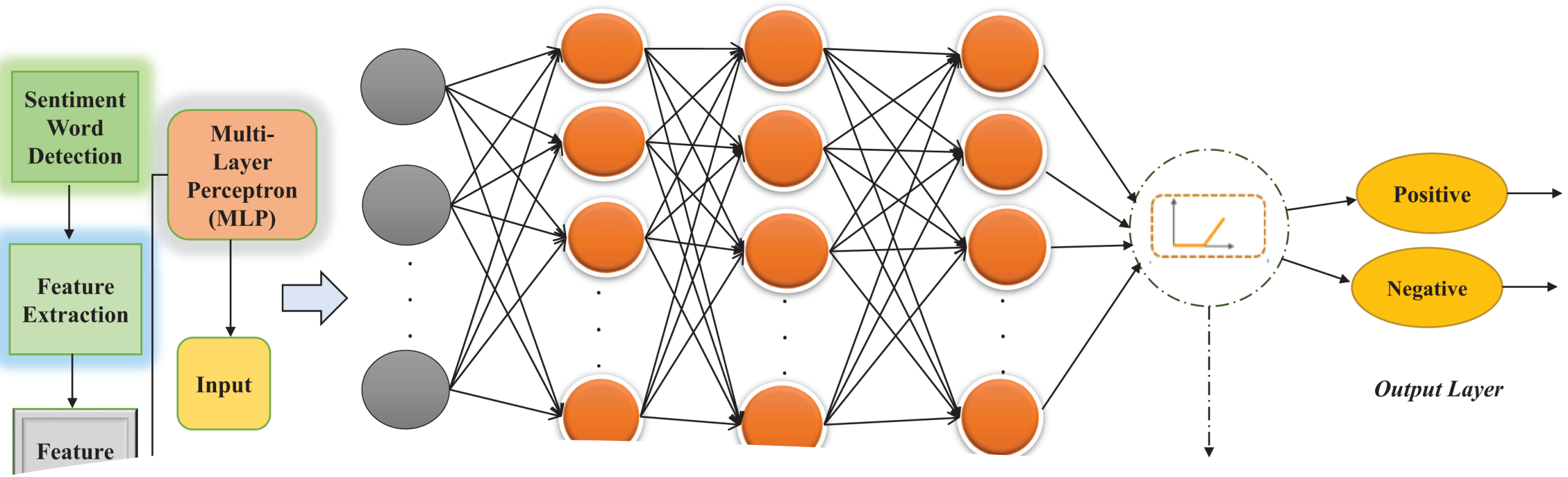
Could we not repeatedly apply a shallow model?

No. A deep learning model learns all layers of representations *jointly* rather than in succession (aka *greedily*).

Everything is supervised by a single feedback signal – every change in the model serves the goal.

This is much more powerful than greedily stacking shallow models. Complex representations are broken down into a long series of spaces. Each space is a simple transformation away from the previous one.

(greedily m= only looking at immediate choices, no long-term strategy)



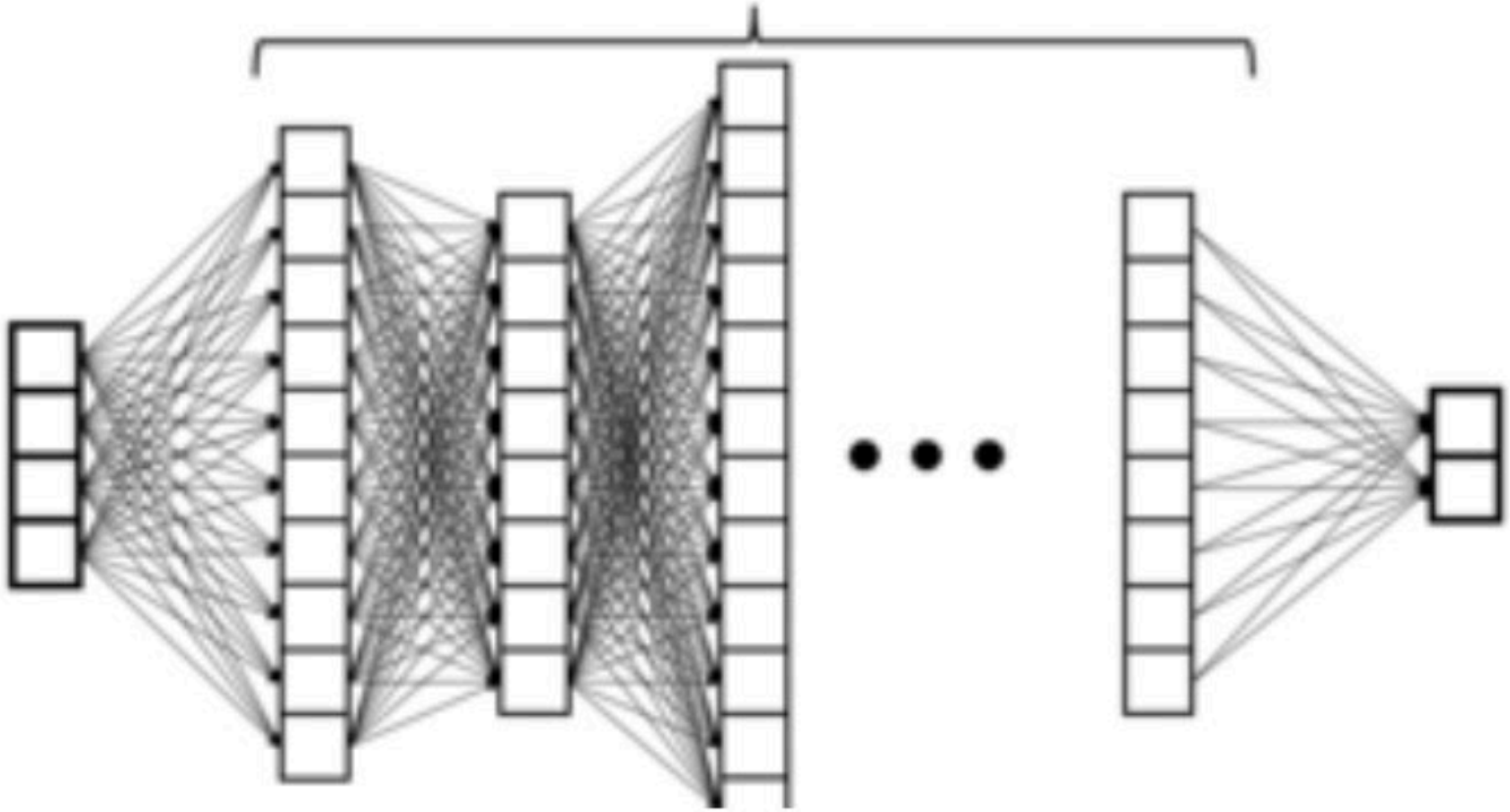
Essence of Deep Learning

1. Incremental layer-by-layer way in which increasingly complex representations are developed

2. Intermediate incremental representations are learned jointly.

Each transformation is performed by a layer. The first input is randomly selected. Then, each layer ends up with a transformation which decreases the loss function between the transformation and the target.

Models with three or more layers are usally called deep.

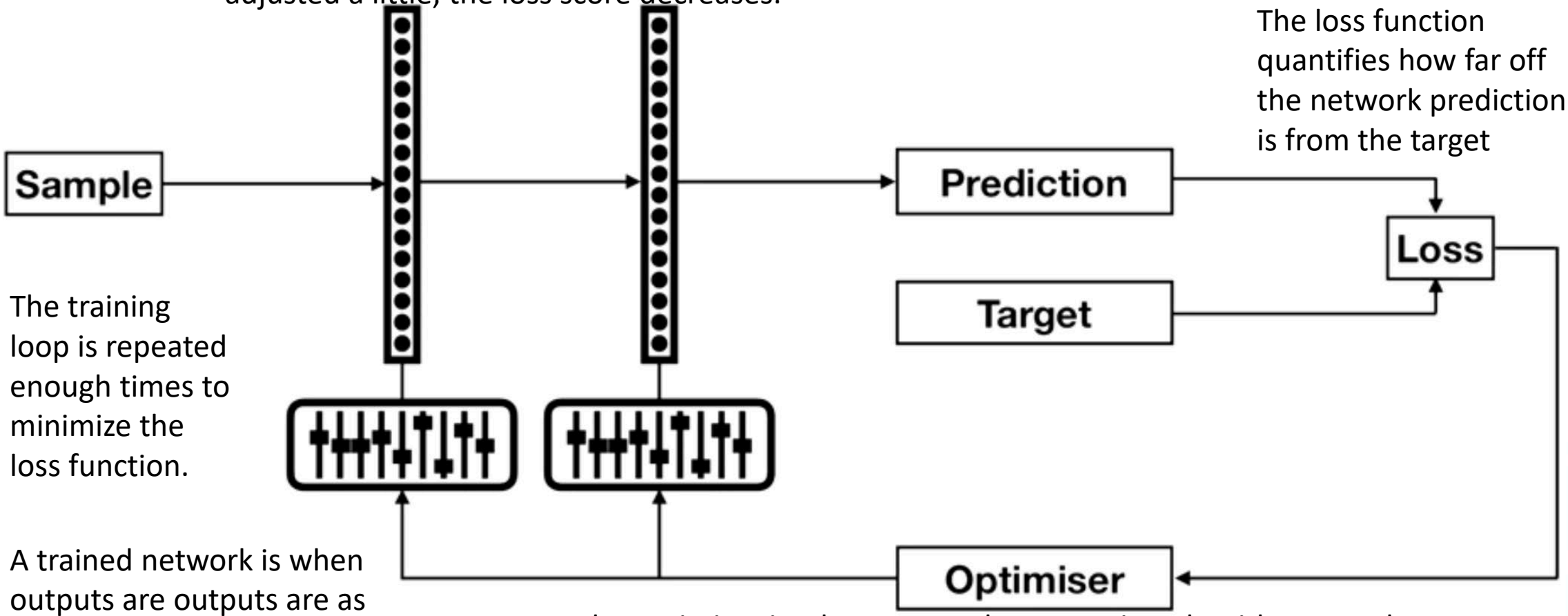


UNIVERSITY OF LONDON

Initially, the weights of the network are assigned random values.
Loss score shows high difference between prediction and target.
With each example the network processes, the weights are adjusted a little, the loss score decreases.

Loss function = objective function

The loss function quantifies how far off the network prediction is from the target

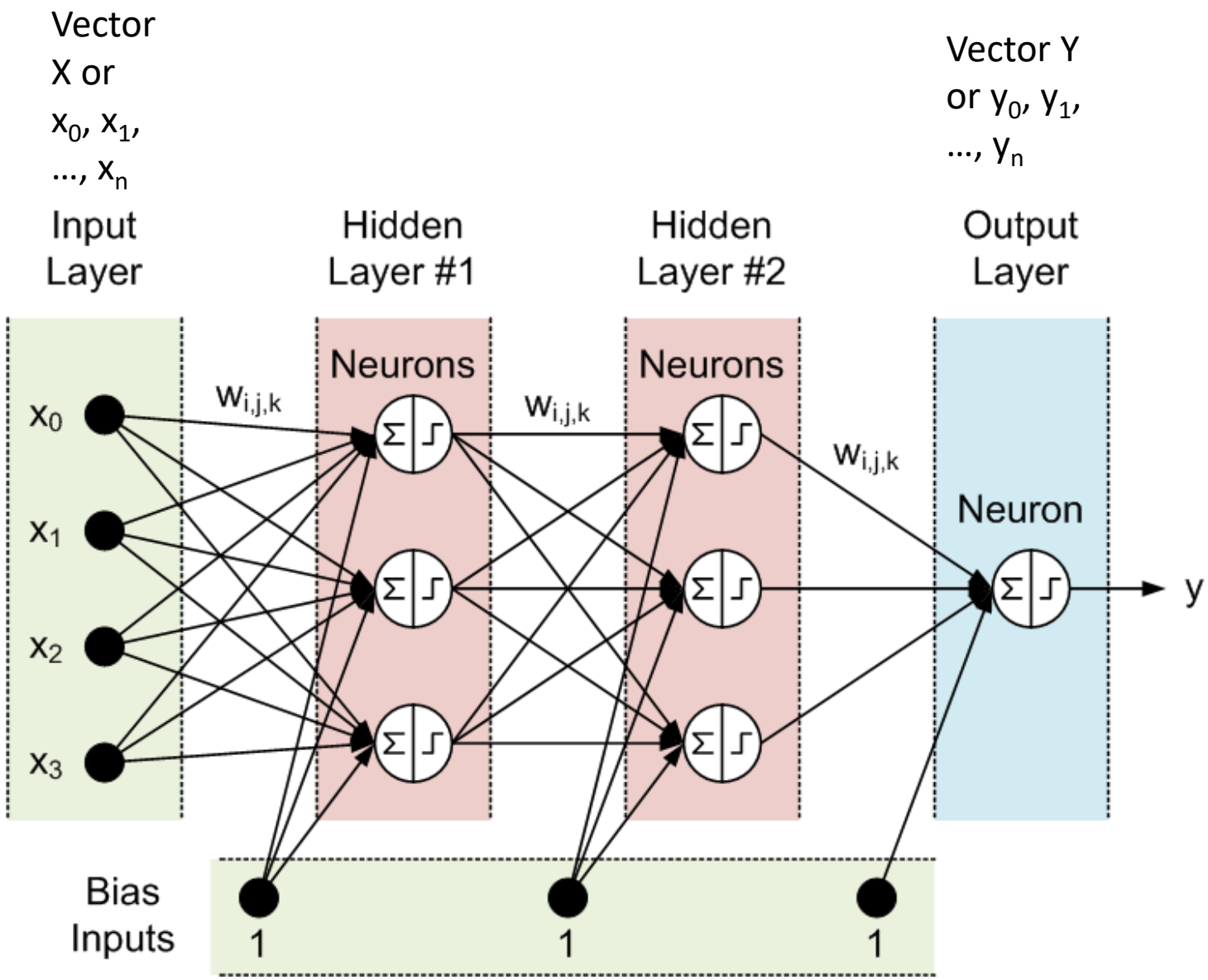


The training loop is repeated enough times to minimize the loss function.

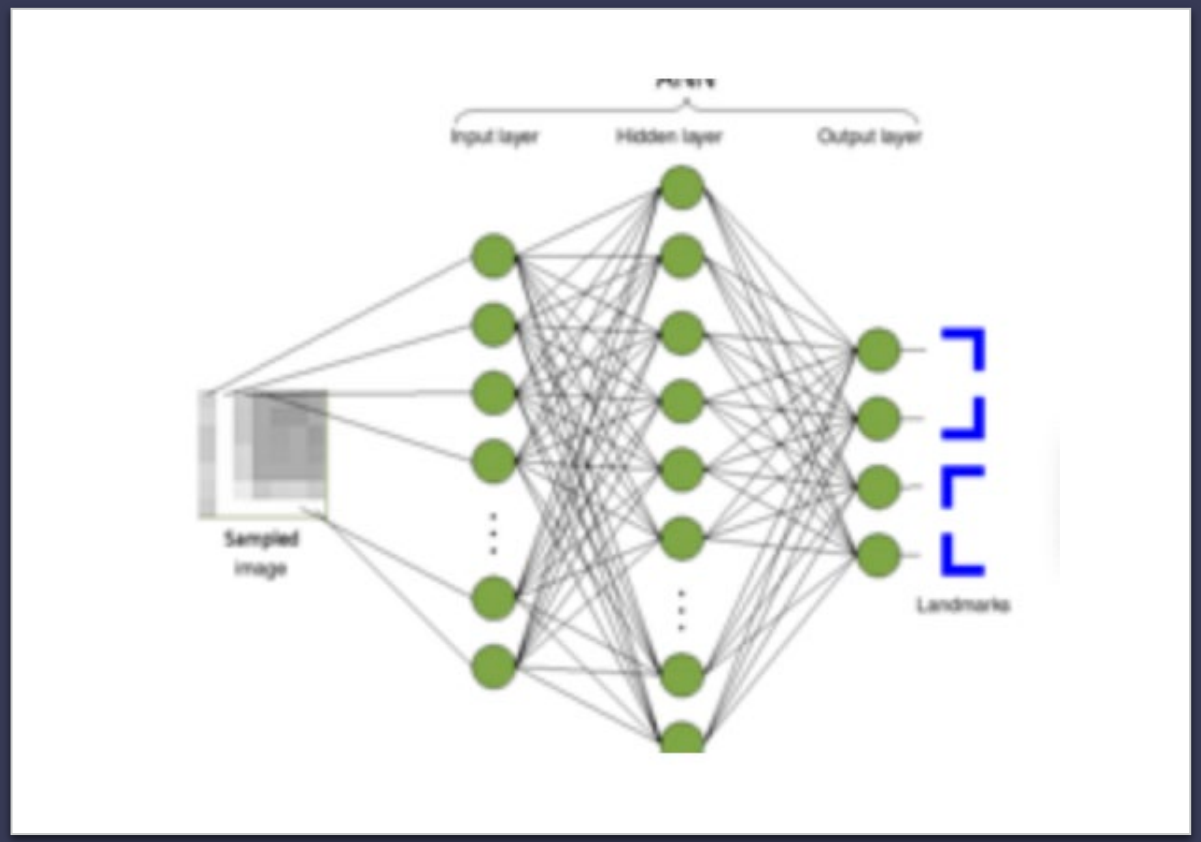
A trained network is when outputs are outputs are as close as they can be to targets

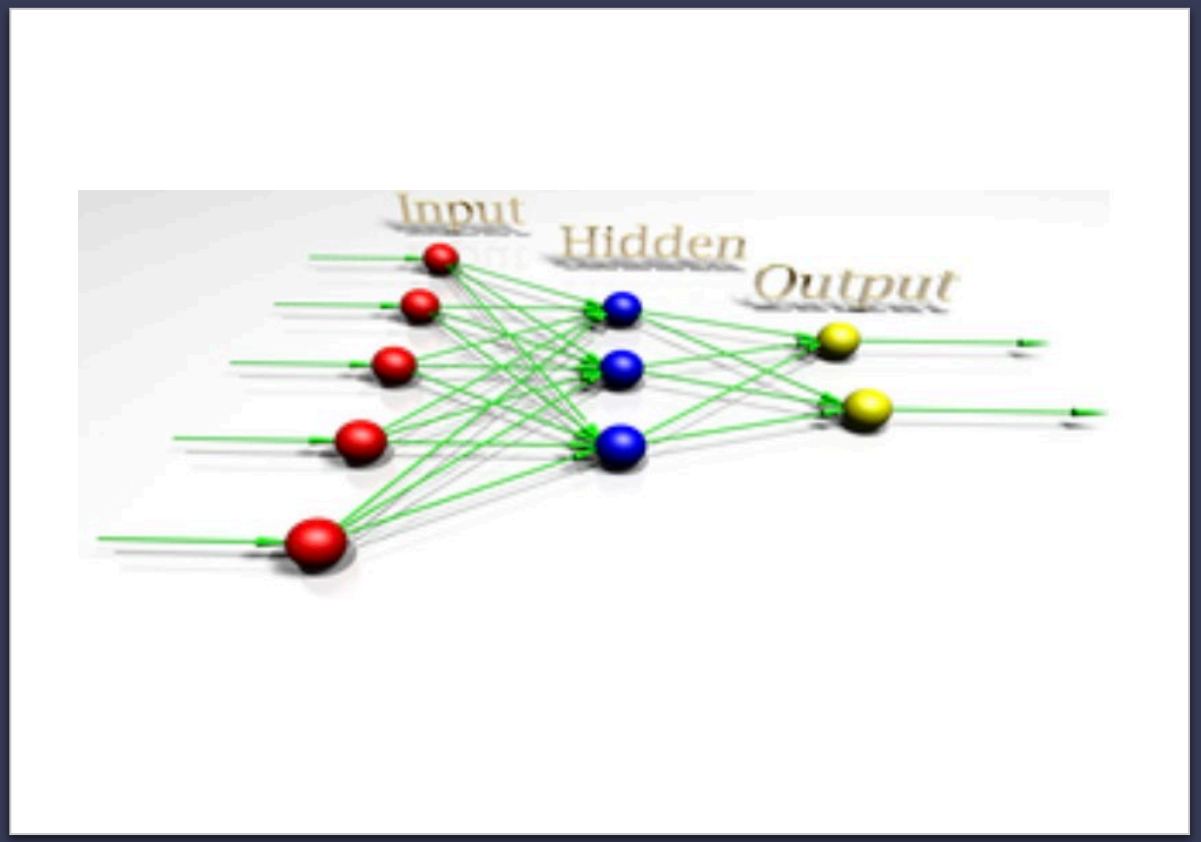
The optimizer implements Backpropagation algorithm. It makes parameter adjustments in the training loop and metrics report on progress

Parametrised by weights (w_{ijk}) and biases



Simple Neural Network = 1 hidden layer between input and output layers

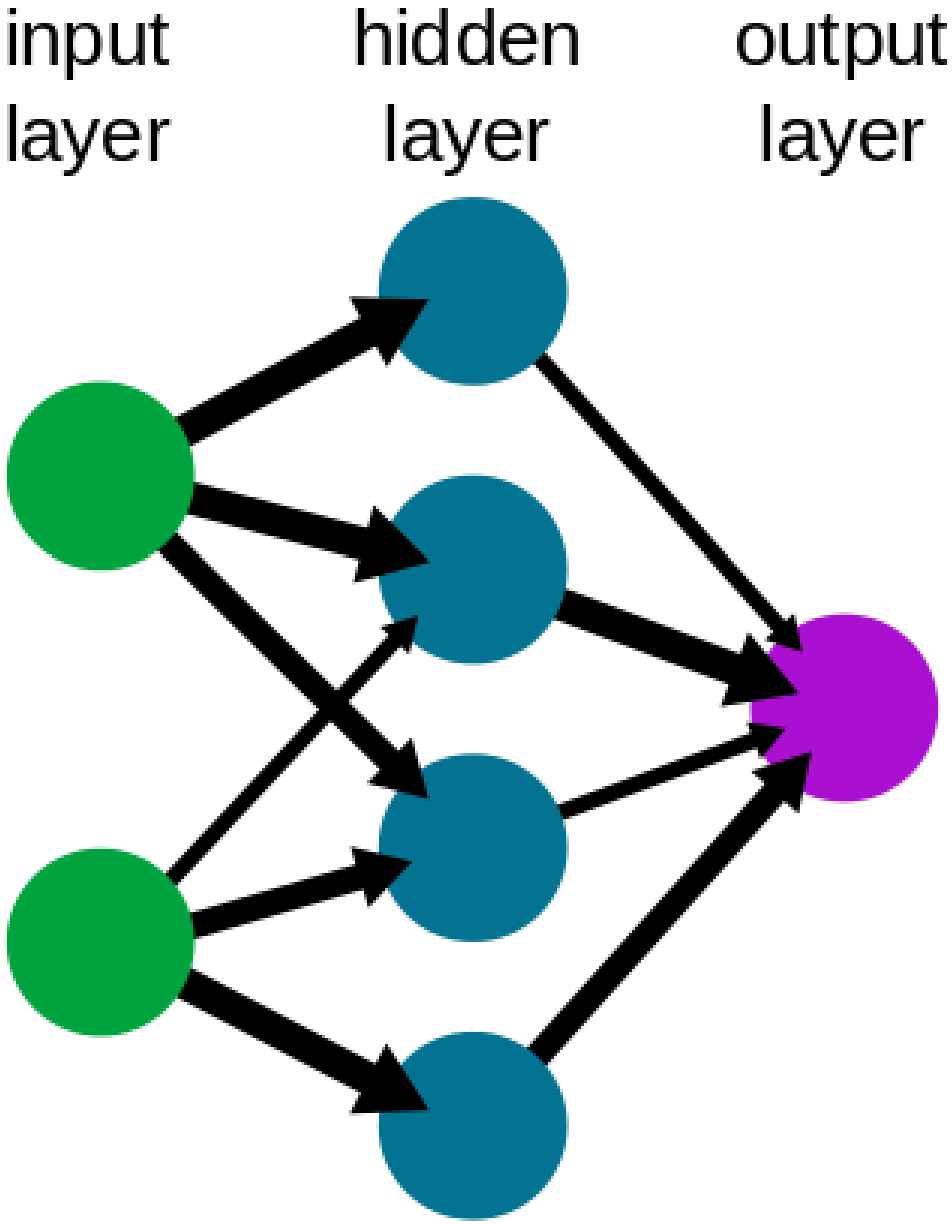




A simple neural network

Adding more hidden layers improves the efficiency of the neural network and reduces the error rate. But the improvement per layer decreases as the hidden layers are added while the computation time increases.

The efficiency of a neural network can be seen as a parallel computation with interconnected components which are “helping” each other. Then, intuitively, the more we add neurons that ‘helps’ the computation, the more accurate we must be.



Directed Acyclic Graph (DAG)

Graph Definition.

A **graph** is an ordered pair $G = (V, E)$ consisting of a nonempty set V (called the **vertices**) and a set E (called the **edges**) of two-element subsets of V.

Some graphs are used more than others, and get special names.

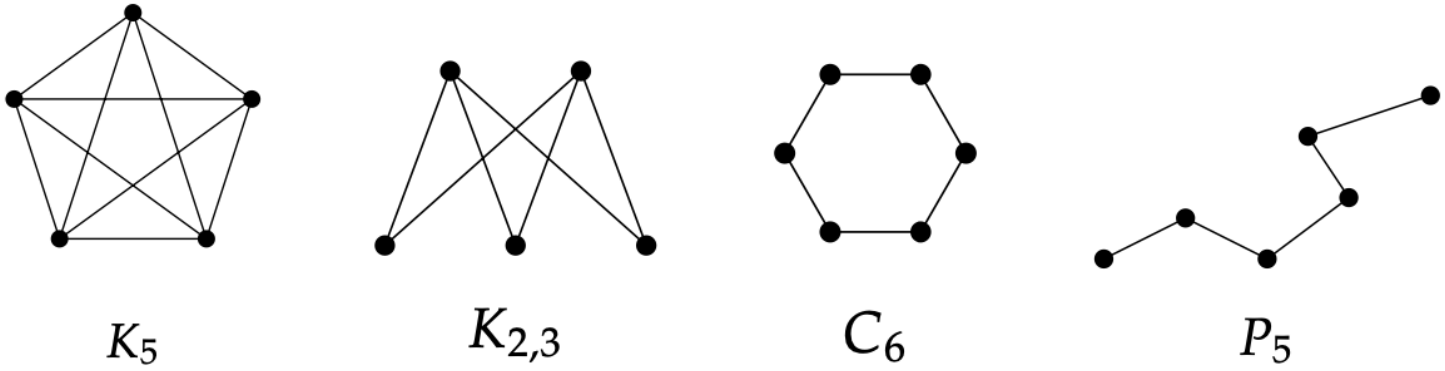
K_n The complete graph on n vertices.

$K_{m,n}$ The complete bipartite graph with sets of m and n vertices.

C_n The cycle on n vertices, just one big loop.

P_n The path on $n + 1$ vertices (so n edges), just one long path.

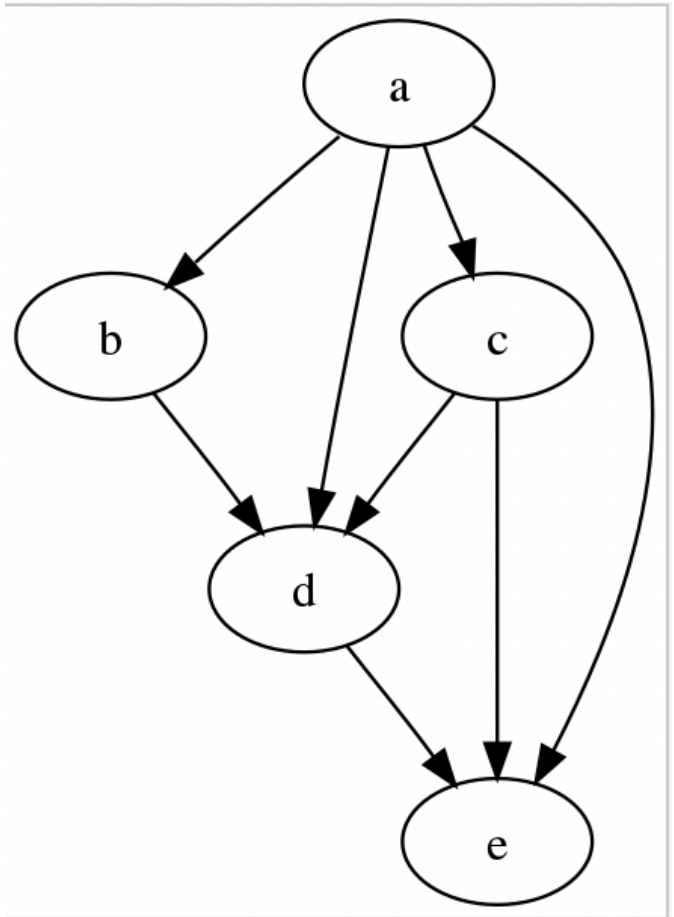
These are all acyclic graphs…



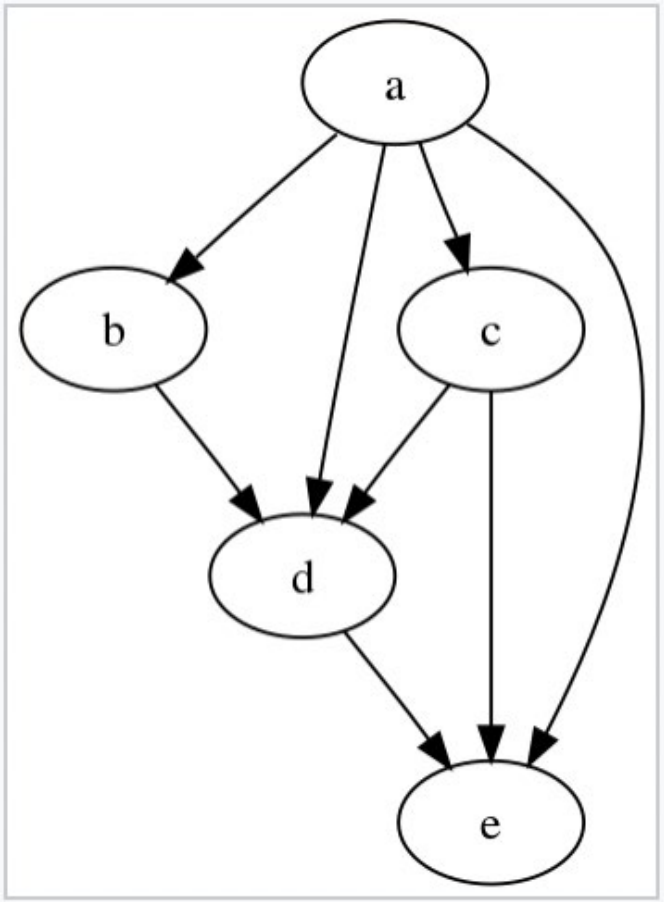
K_5 $K_{2,3}$ C_6 P_5

DAG = directed (arrowed) graph with no loops

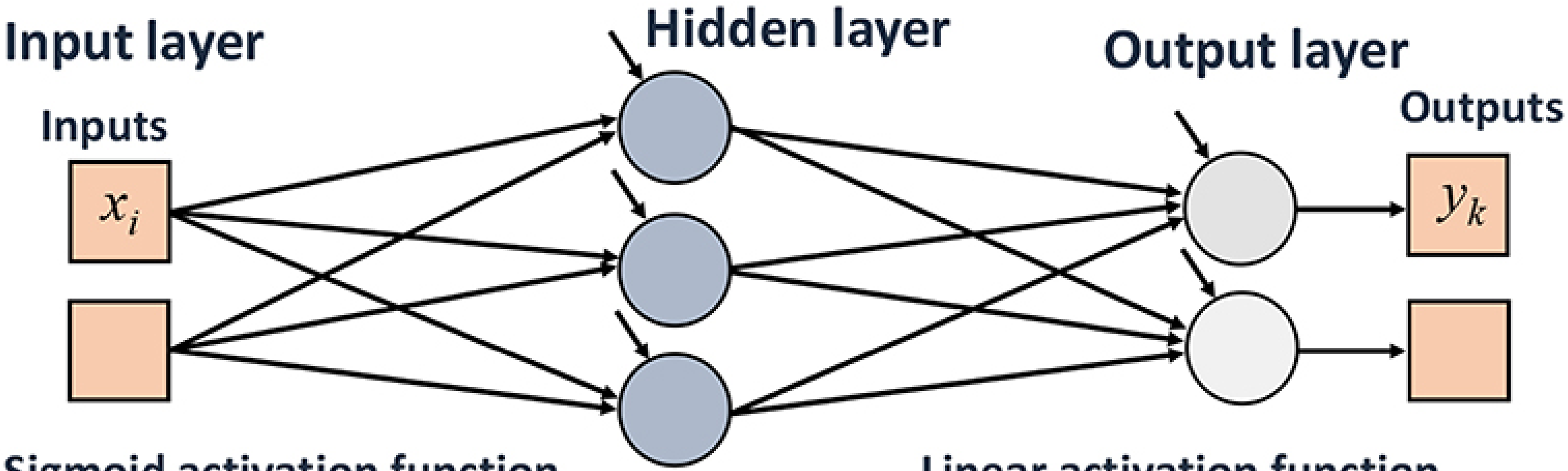
Cycle = loop



Example of a directed acyclic graph



A DAG



Sigmoid activation function

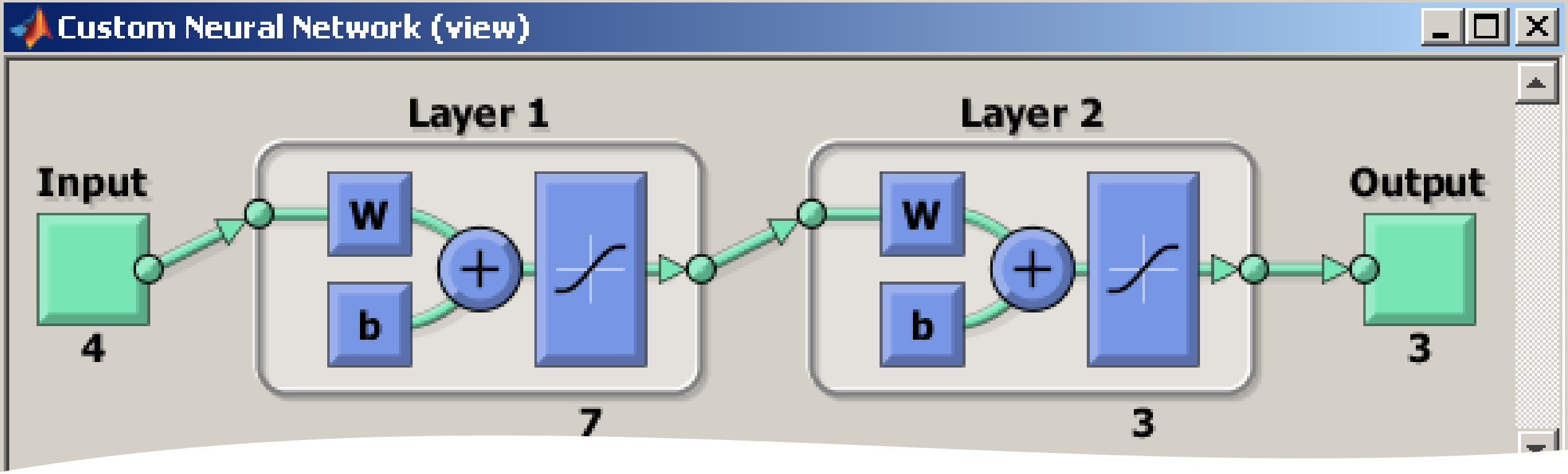
$$\sigma\left(\sum_{i=1}^{N_{\text{in}}} w_{ji}^{(1)} x_i + b_j^{(1)}\right) = h_j$$

Linear activation function

$$\sum_{j=1}^{N_{\text{h}}} w_{kj}^{(2)} h_j + b_k^{(2)} = y_k$$

Tensorflow =
Tensor-manipulation framework for Python that support auto-differentiation, thereby simplifying models, and user-friendly libraries such as Keras , “which makes deep learning as easy as manipulating LEGO bricks”.

F Chollet, “Deep Learning with Python”, Manning Publications, (2018) p. 23



Understanding deep learning requires familiarity with simple mathematical operations:

1. Tensors
2. Tensor operations
3. Differentiation
4. Gradient descent

F Chollet, “Deep Learning with Python”, Manning Publications, (2018) p. 25

What Are Tensors?

- Tensors are the standard way of representing data in TensorFlow (deep learning).
- Tensors are multidimensional arrays, an extension of two-dimensional tables (matrices) to data with higher dimension.

0 axis

Scalar eg

3

0D tensor, in Python a float32 or float64 number

1 axis

'r'
'e'
'n'
's'
'o'
'r'

It is a 1D tensor and a 6D vector.

Tensor of dimension[1]

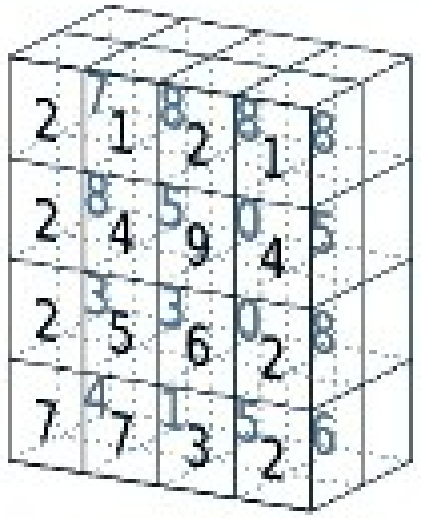
2 axes

3	1	4	1
5	9	2	6
5	3	5	8
9	7	9	3
2	3	8	4
6	2	6	4

Matrix has 2 axes, here 6rows and 4 columns

Tensor of dimensions[2]

2D tensor, in Python ndim == 2



3 axes

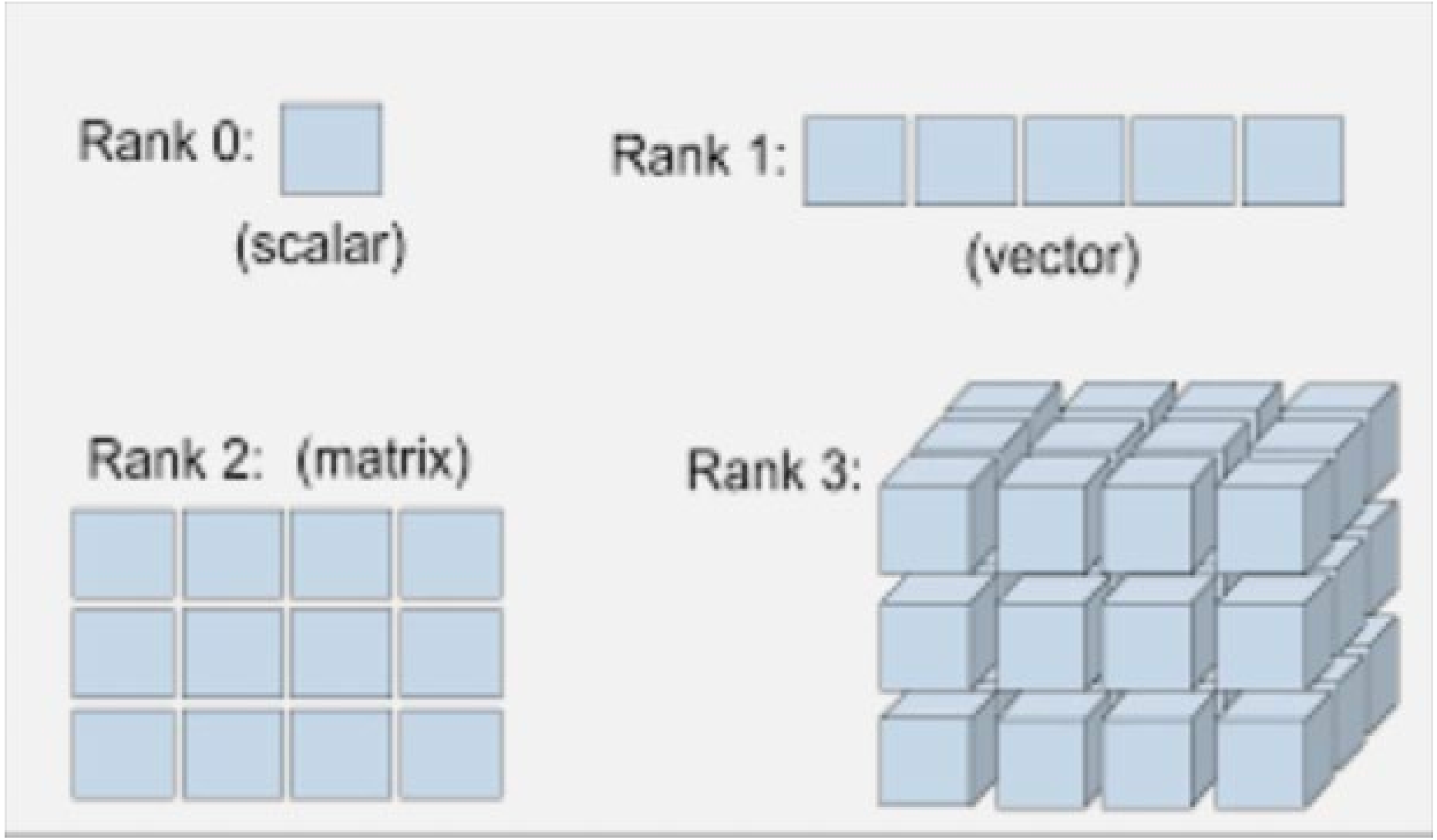
3D tensor, in Python ndim == 3

Tensor of dimensions[3]

TensorFlow Tutorial

edureka!

Another way to describe tensor dimensions is rank:-



3D tensor

```
4]: 1 # the shape of a 3D array
    2 x = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]])
    3 print(x, '\t', x.shape)
```

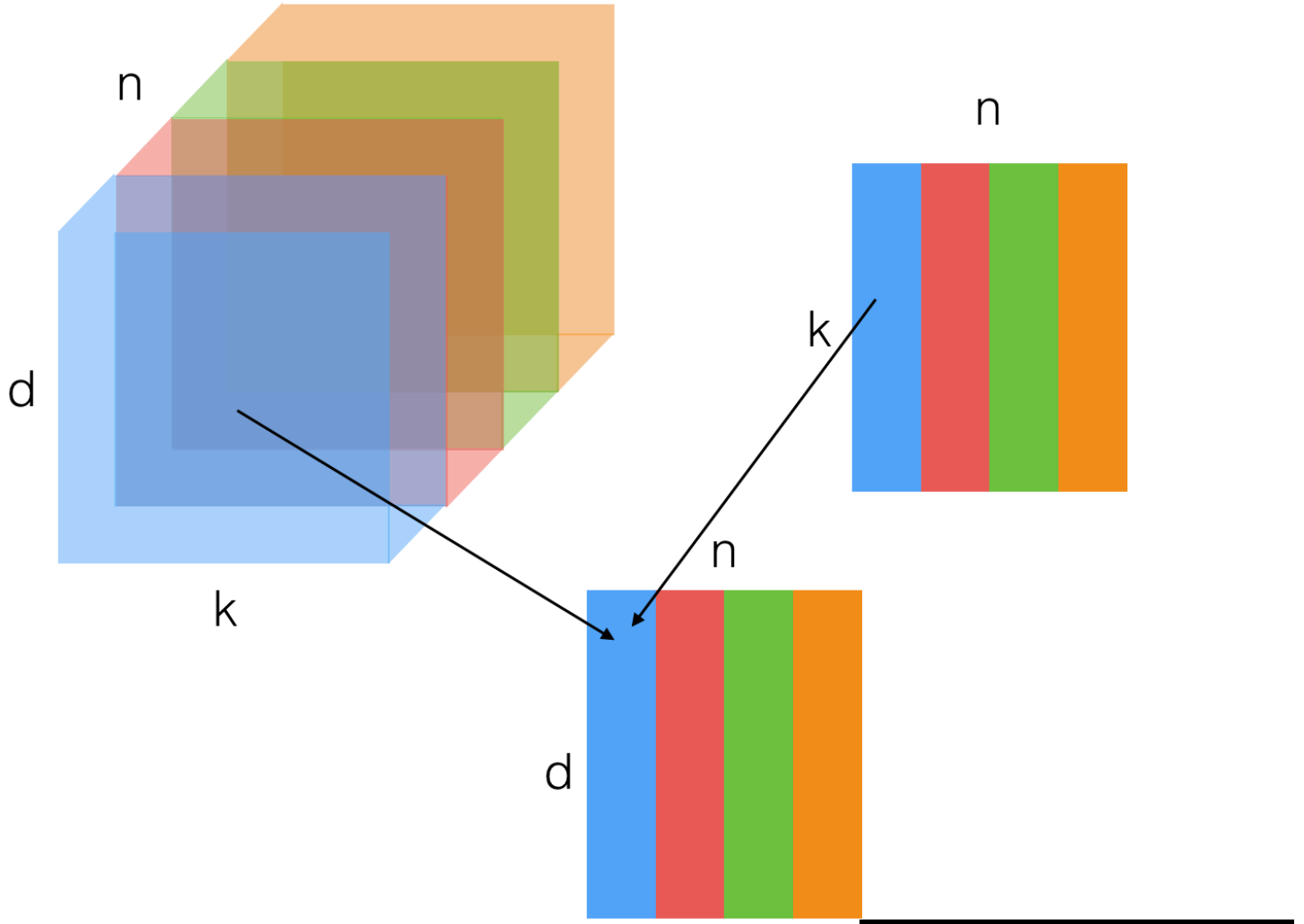
```
[[[ 1  2]
  [ 3  4]]

 [[ 5  6]
  [ 7  8]]

 [[ 9 10]
  [11 12]]]      (3, 2, 2)
```

Tensor operations

- slicing
- arithmetic: addition, multiplication
- broadcasting
- element-wise operations
- tensor dot (akin to matrix multiplication)

Slicing = sub-setting a section of data from the main dataset

Slicing



Basic Boolean Operations

- CONJUNCTION (means ‘AND’), logical product, intersection, conjunction
- x ∧ y
- x ∩ y
- x · y
- DISJUNCTION (means ‘OR’), logical sum, union, disjunction
- x ∨ y
- x ∪ y
- x + y
- NEGATION (means ‘NOT’) = logical complementation or inverter
- x’
- ¬ x or ~x
- x-bar

Addition

Element-wise addition

$$\begin{pmatrix} 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 8 \\ 9 \end{pmatrix} = \begin{pmatrix} 11 \\ 13 \end{pmatrix}$$

In [1]:

```
1 import numpy as np
2 x = np.array([3, 4])
3 y = np.array([8, 9])
4 print(x + y)
```

```
[11 13]
```

Tensor Broadcasting

= transforming tensors of different dimensions/shape to the compatible shape such that arithmetic operations can be performed on them.

In general, if there is no ambiguity, a smaller tensor will be *broadcasted* to match the shape of the larger tensor. There are 2 steps:-

1. Axes (also called *broadcast axes*) added to smaller tensor to match it to larger one.
2. Smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor.

Transpose = change 1st row becomes 1st column, 2nd row becomes 2nd column, etc and vice versa

```
[2]: 1 x = x.reshape(6, 1)
     2 print(x)

[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
```

```
In [3]: 1 x = np.array([[1, 2], [3, 4], [5, 6]])
        2 x = x.transpose()
        3 print(x, '\t', x.shape)

[[1 3 5]
 [2 4 6]]        (2, 3)
```

The General Broadcasting Rule:

- tensor shapes are compared right to left
- dimensions are compatible if they are equal or one of them is 1
- the axis with dimension 1 is copied to match the other

Equivalent to broadcasting

In [1]:

```
1  import numpy as np
2  x = np.array([[2, 3, 4],
3                [5, 6, 7]])
4  y = np.array([0.5, 0, 10])
5  print(x * y)
6  print()
7
8  # equivalent to broadcasting
9  z = np.array([[0.5, 0, 10],
10               [0.5, 0, 10]])
11 print(x * z)
```

```
[[ 1.   0.  40. ]
 [ 2.5  0.  70. ]]

[[ 1.   0.  40. ]
 [ 2.5  0.  70. ]]
```

Example 1

Suppose we wish to element-wise multiply a tensor of shape (2, 3) by a tensor of shape (3,)

A (2, 3) tensor is formed from the (3,) tensor by duplicating along a broadcast axis

Example 2

Suppose you wish to scale the colours of scale a 256 x 256 RGB image by a vector of scale factors

$$\begin{aligned} \text{image tensor} &: 256 \times 256 \times 3 \\ \text{scale vector} &: 3 \\ \text{result} &: 256 \times 256 \times 3 \end{aligned}$$

Example 3

$$x : 5 \times 1 \times 3$$

$$y : 5 \times 2 \times 3$$

$$\text{result} : 5 \times 2 \times 3$$

Tensor Dot

Dot Product

$$\begin{bmatrix} a & b \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax+by \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax+by \\ cx+dy \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{bmatrix}$$

Dot Product

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \bullet \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1\cdot 5)+(2\cdot 7) & (1\cdot 6)+(2\cdot 8) \\ (3\cdot 5)+(4\cdot 7) & (3\cdot 6)+(4\cdot 8) \end{bmatrix}$$

$$= \begin{bmatrix} 5+14 & 6+16 \\ 15+28 & 18+32 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Element-wise Function application

Element-wise function application

$$f\begin{pmatrix}-2\\3\end{pmatrix} = \begin{pmatrix}f(-2)\\f(3)\end{pmatrix}$$

`relu` activation:

$$\texttt{relu}\begin{pmatrix}-2\\3\end{pmatrix} = \begin{pmatrix}\texttt{relu}(-2)\\\texttt{relu}(3)\end{pmatrix} = \begin{pmatrix}0\\3\end{pmatrix}$$

Element-wise operations are highly amenable to massively parallel implementations.

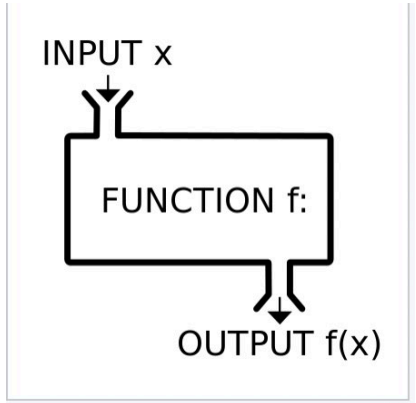
In [2]:

```
# simple definition of a relu function for rank 1 tensors
def relu(x):
    assert len(x.shape) == 1
    y = x.copy()
    for i in range(x.shape[0]):
        y[i] = max(y[i], 0.)
    return y
```

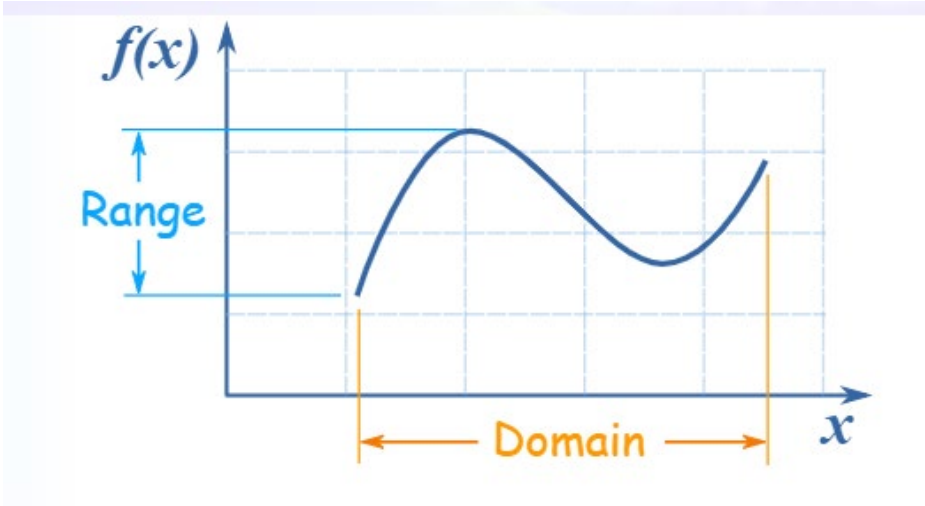
In [3]:

```
x = np.array([-2, 3])
print(relu(x))
```

```
[0 3]
```



Schematic depiction of a function described metaphorically as a "machine" or "black box" that for each input yields a corresponding output



What is a function?

A function is a mathematical ‘rule’ which maps input (X) to ouput (y). It transforms input to output.

In a graph, the x values are transformed into or ‘mapped onto’ the y-values.

Eg $f(x) = x^2$ or $y = x^2$

tells us to take the x-value and square it

Affine function

Function $f(w \cdot x + b)$ is an affine function

= a function composed of a linear function + a constant, $y = Ax + c$.

It is represented by a straight line graph.

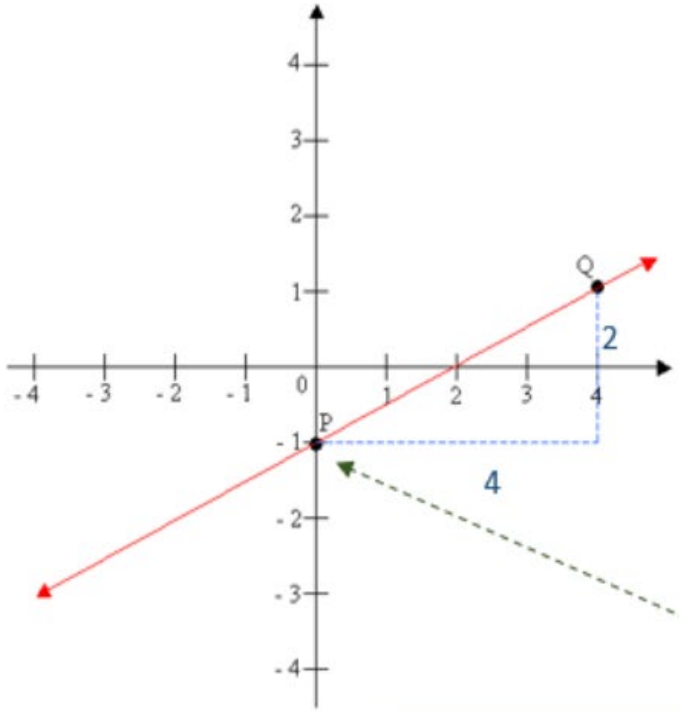
Linear function

A function is linear if

1. $f(x+y) = f(x) + f(y)$
2. $f(ax) = af(x)$

Otherwise it is nonlinear

Find Equation of Straight Line Graph



1. Find the gradient (slope)

$$m = \frac{2}{4} = \frac{1}{2}$$

2. Find the y-intercept

$$c = -1$$

3. Write the equation in slope-intercept form

$$y = mx + c$$

$$y = \frac{1}{2}x - 1$$

A layer performs

```
f(dot(w, x) + b)
```

mapping rank 2 tensors to rank 2 tensors

ie matrix to matrix

```
f(dot(w, x) + b)
```

- `w` is the tensor of weight parameters
- `b` is the tensor of bias parameters
- `x` is the input tensor

```
f(dot(w, x) + b)
```

- `f` is the **activation** function
- `layers.Dense(512, activation='relu')` specifies a `relu` activation
- `relu(x) = max(x, 0.)`

So, we have 3 tensor operations here:-

1. A dot product between input tensor and a tensor named addition
2. + between resulting 2D tensor and vector b
3. relu(x) = max(x,0)

Output = relu(dot(w, x) + b) or Output = f(dot(w, x) + b)

In this expression, w and b are tensors which are attributes of the layer.

They are called *weights* or *trainable parameters* of the layer.

w = kernel and b = bias contain information learned from training the data

The process of learning is to compare the predictions and target and thereby compute a loss gradient. We compare *y_pred* with *y.*

Weights are adjusted to minimize loss gradient, incrementally layer by layer.

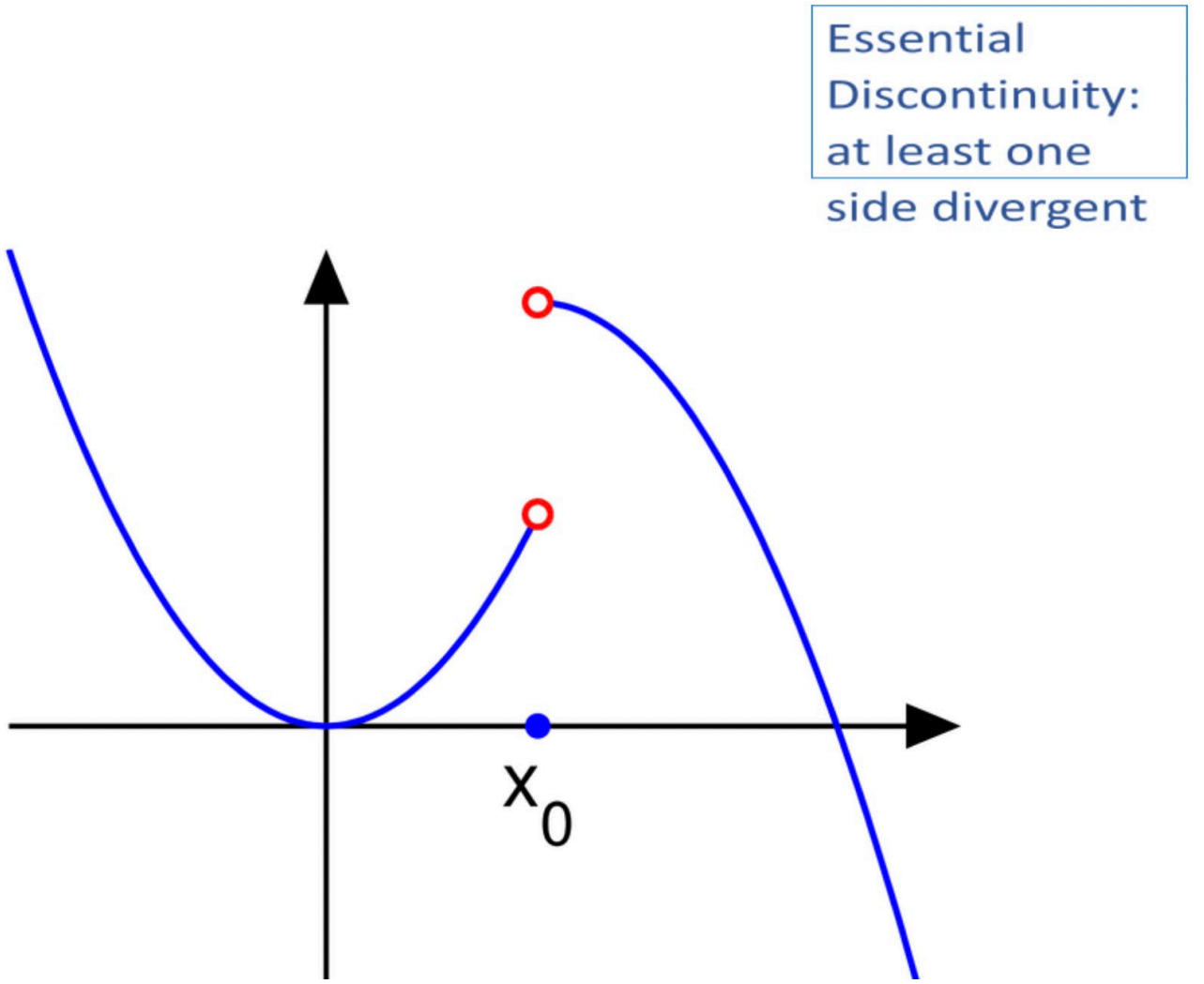
Q: How to evaluate a gradient? A: Differentiation. All operations used in the network are differentiable

---

Differentiation is the process to find the derivative or gradient of the slope of a continuous function.

But what is a continuous function?

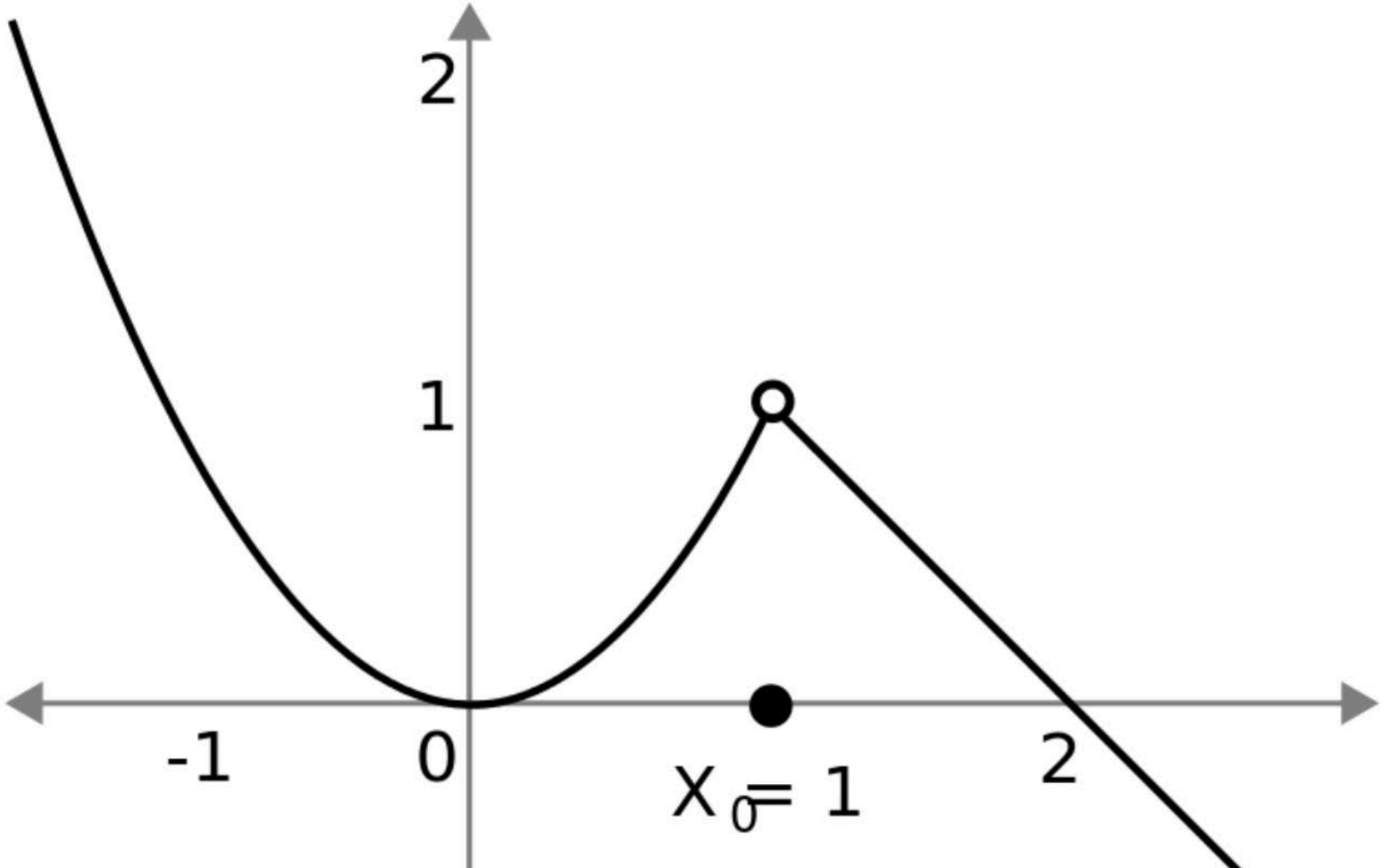
A function which is not discontinuous. It is a smooth function with no breaks.



Removable discontinuity/

Removable singularity

= limit exists but outside the curve of the function

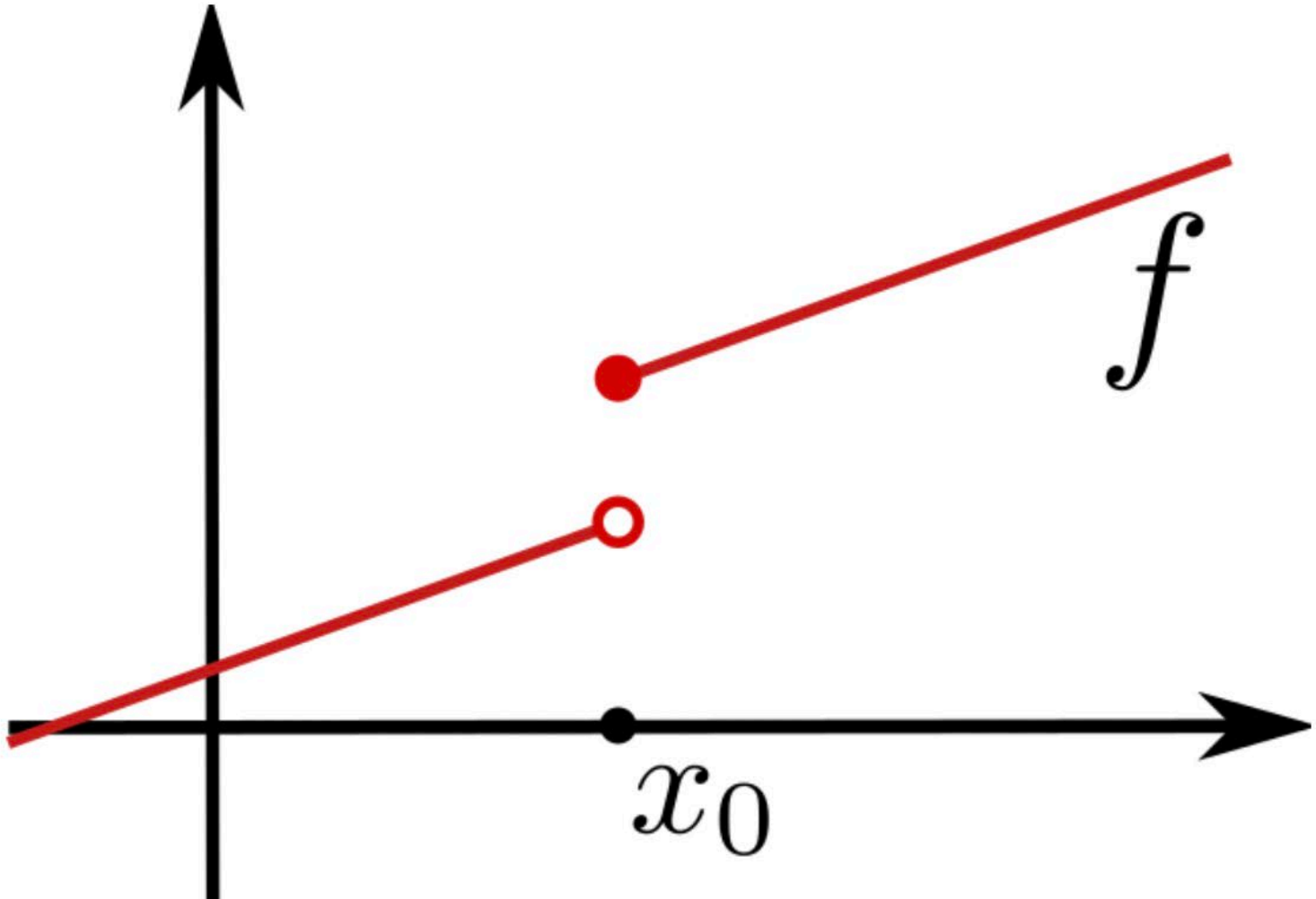


f(x) = { x^2, x < 1

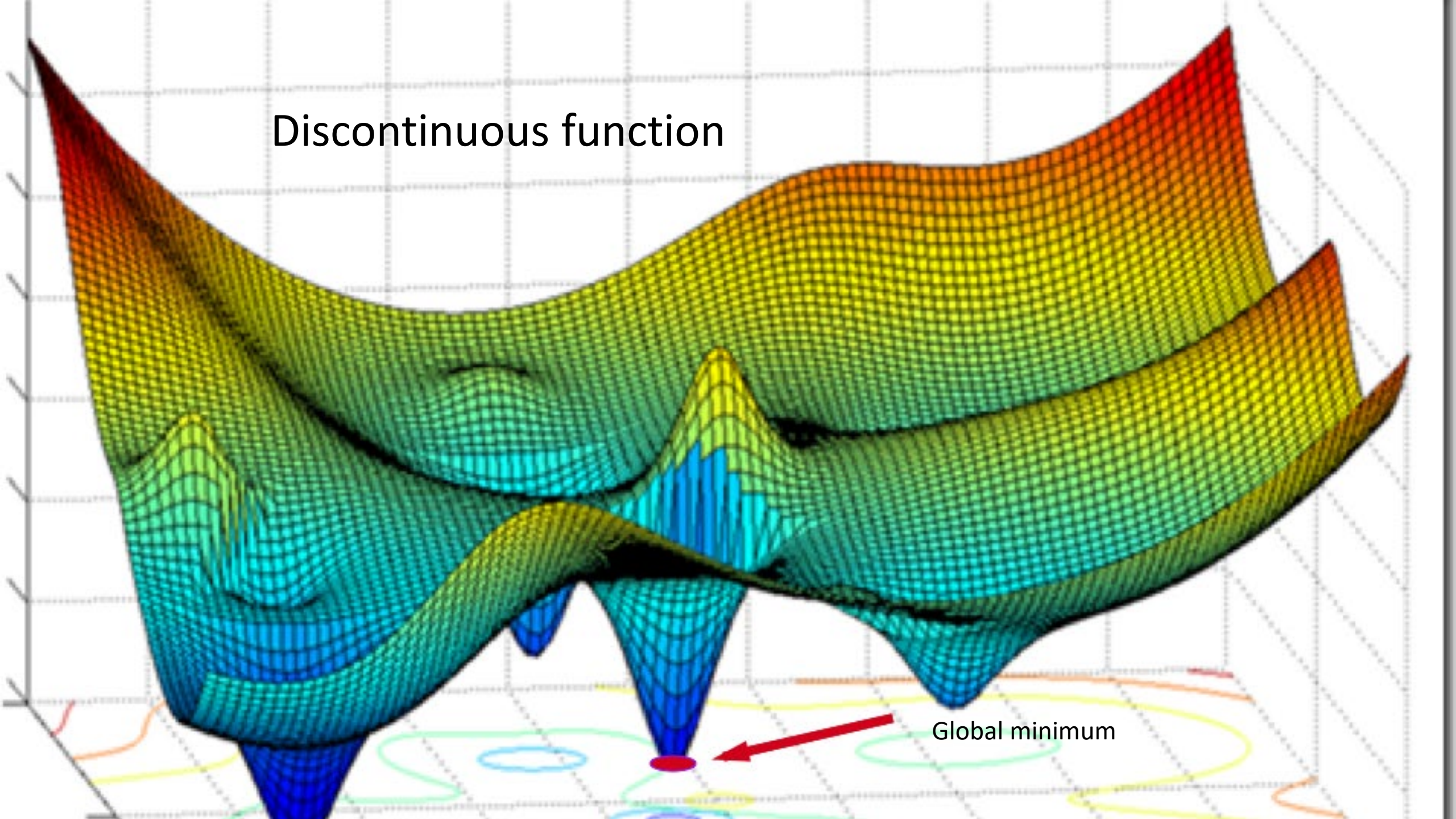
{-x +2, x > 1

h 0

Jump Discontinuity: 2 limits, 1 from left, 1 from right



Discontinuous function

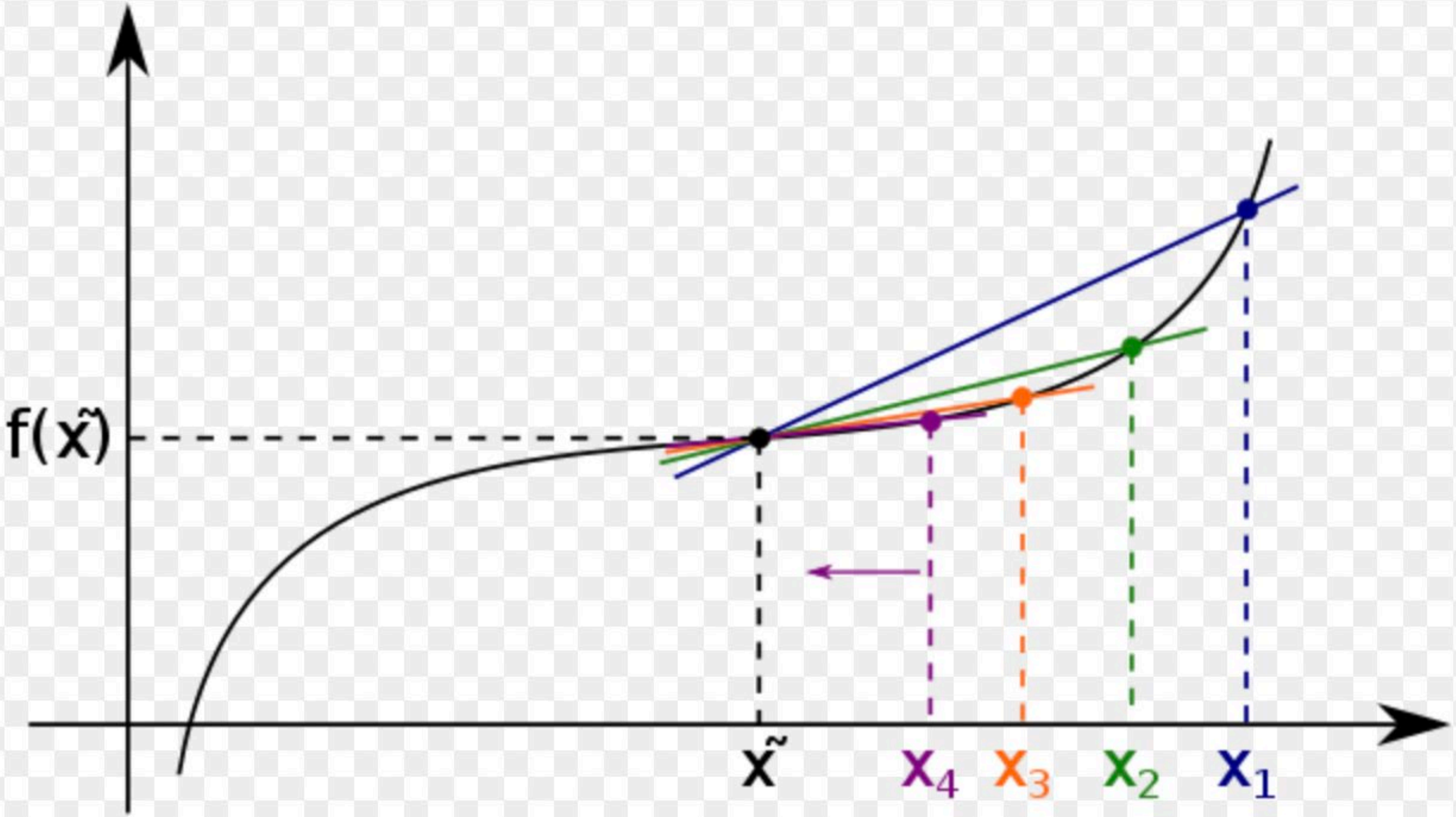


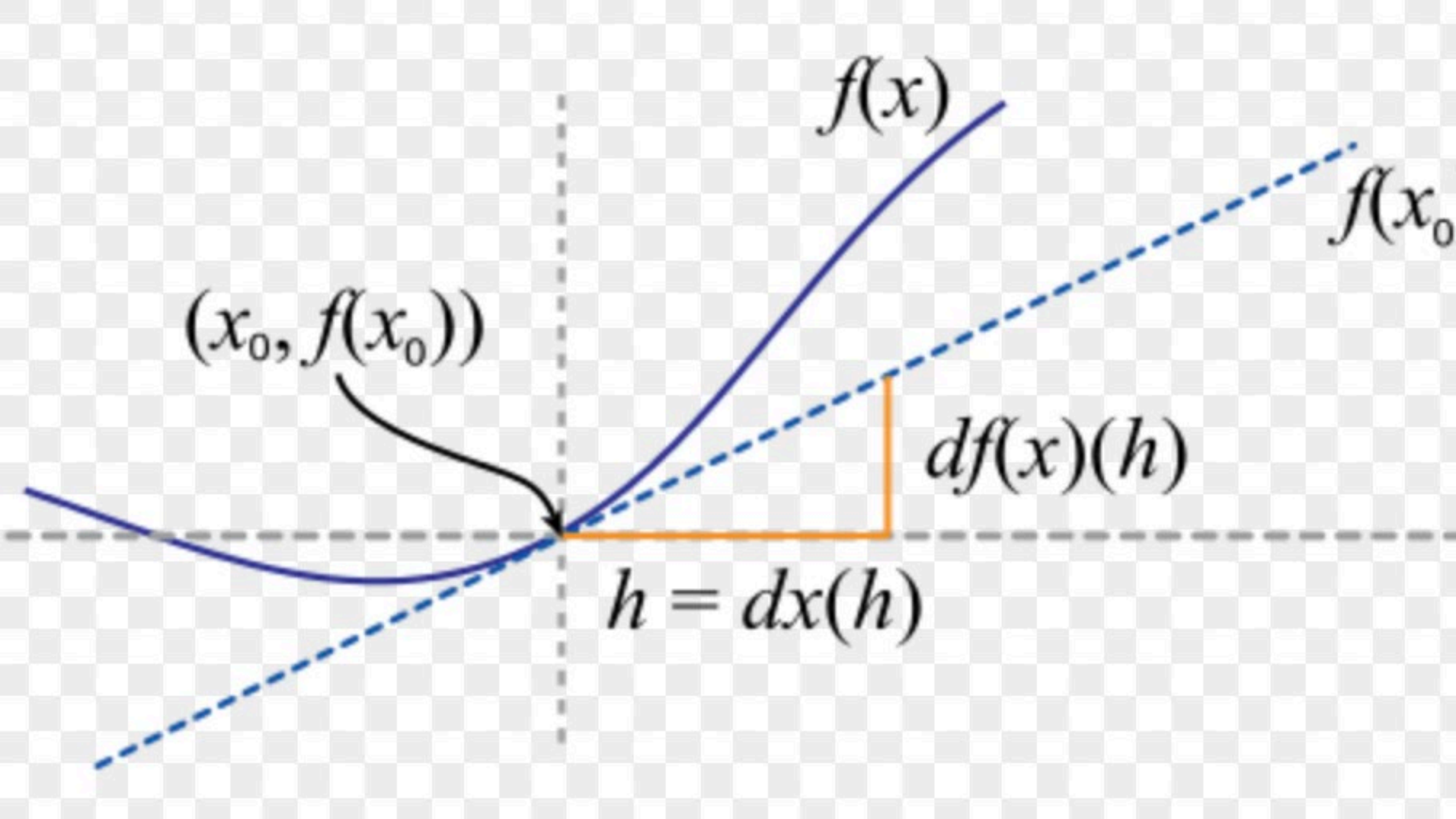
A smooth function f is a function without abrupt changes

f in the vicinity of a point x can always be approximated

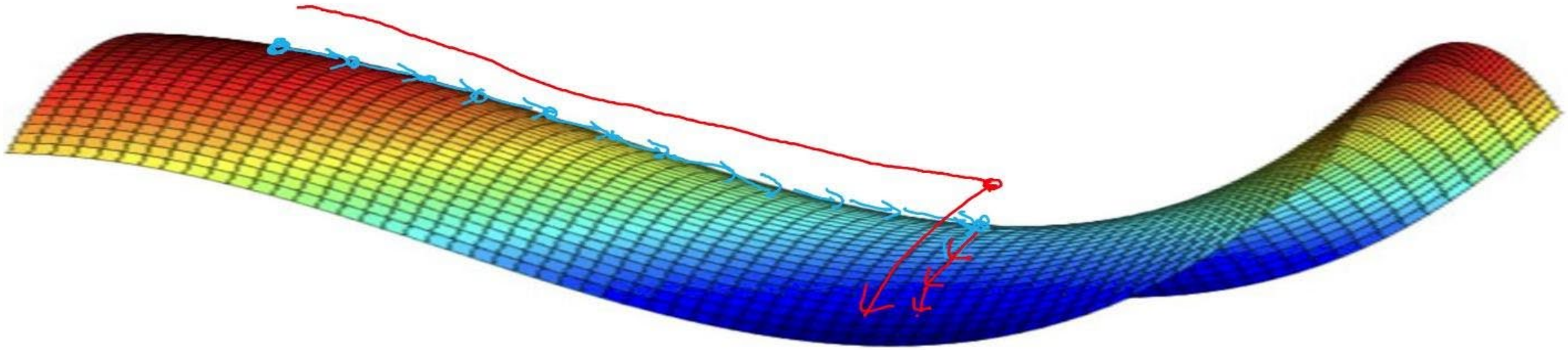
$$f(x + \delta x) \approx f(x) + m\delta x$$

where m is the gradient of the tangent at x and δx is a small step


f(x̃)
x̃
x4
x3
x2
x1


f(x)
f(x0
(x0, f(x0))
df(x)(h)
h = dx(h)

Derivative = how f(x) evolves as you change x.

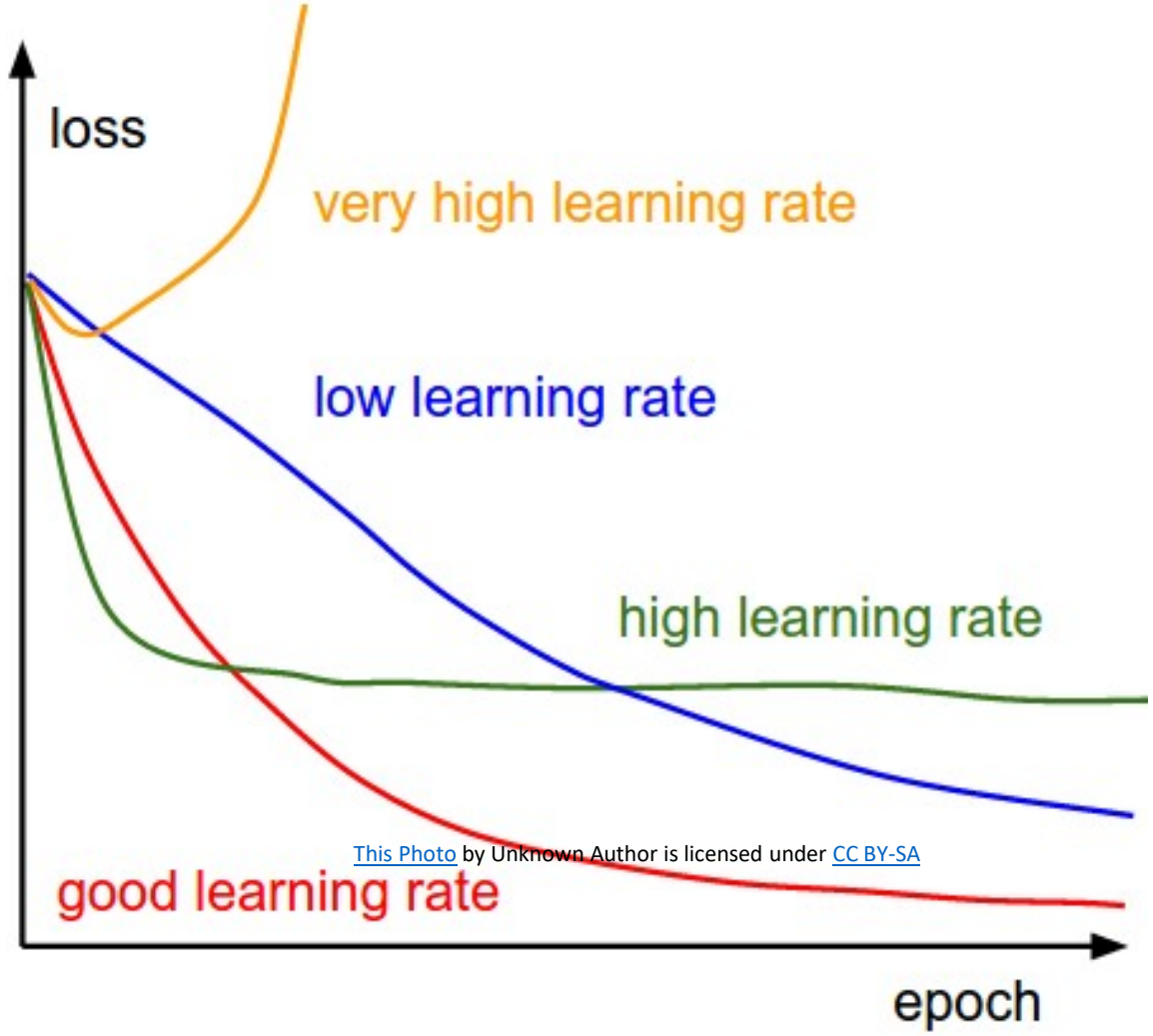


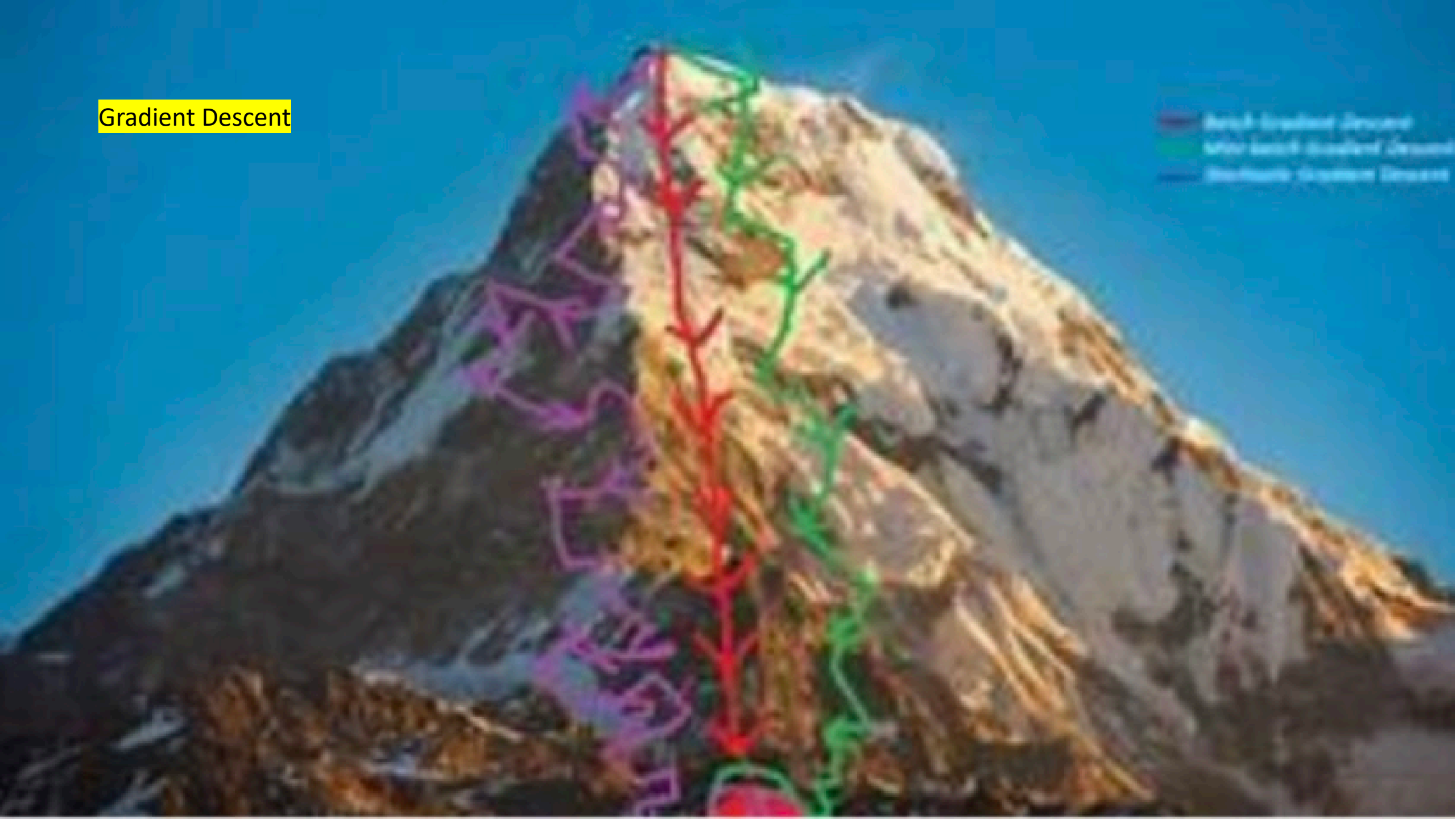
The loss function is similar to ground height in a (multidimensional) landscape

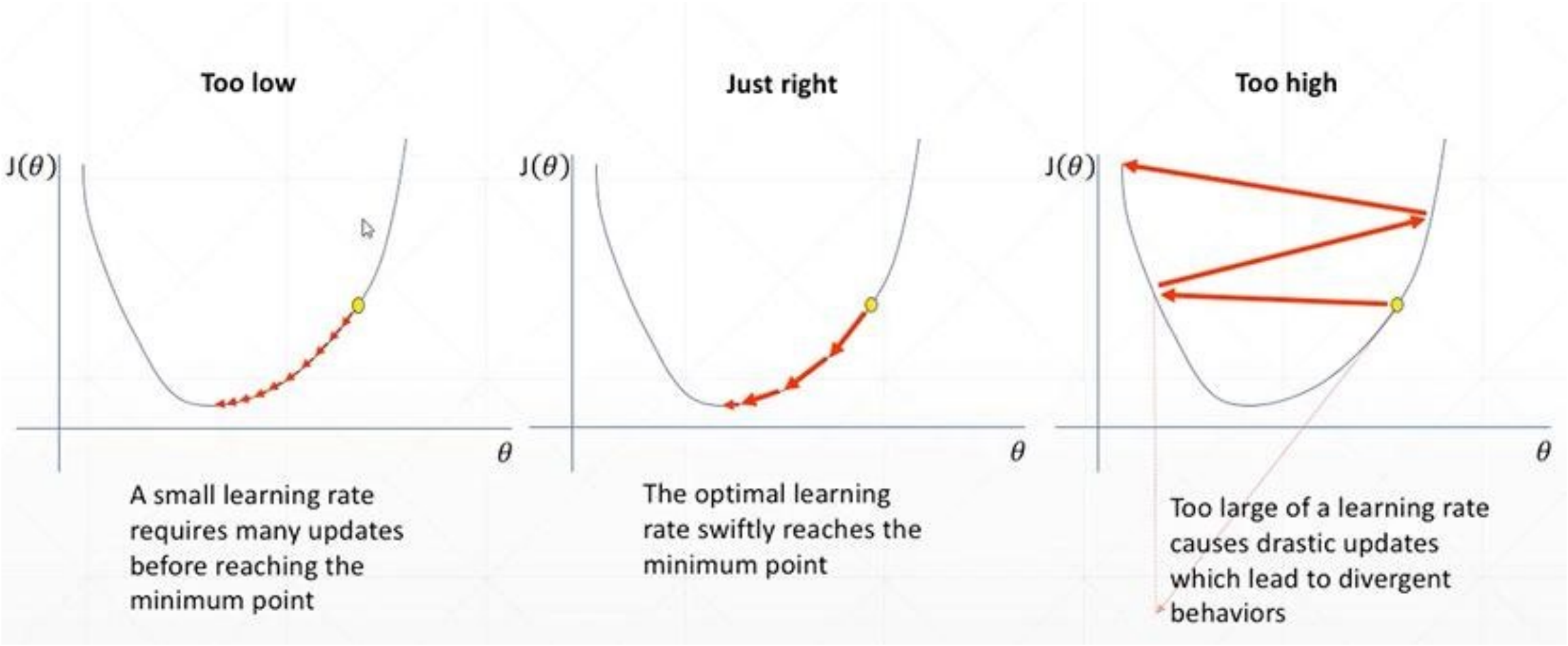
The value of all the weights and biases (all the slider values) define a point on this landscape

The optimiser just needs to move the points in a local downwards direction

A gradient is the derivative of a tensor operation, related to learning rate.

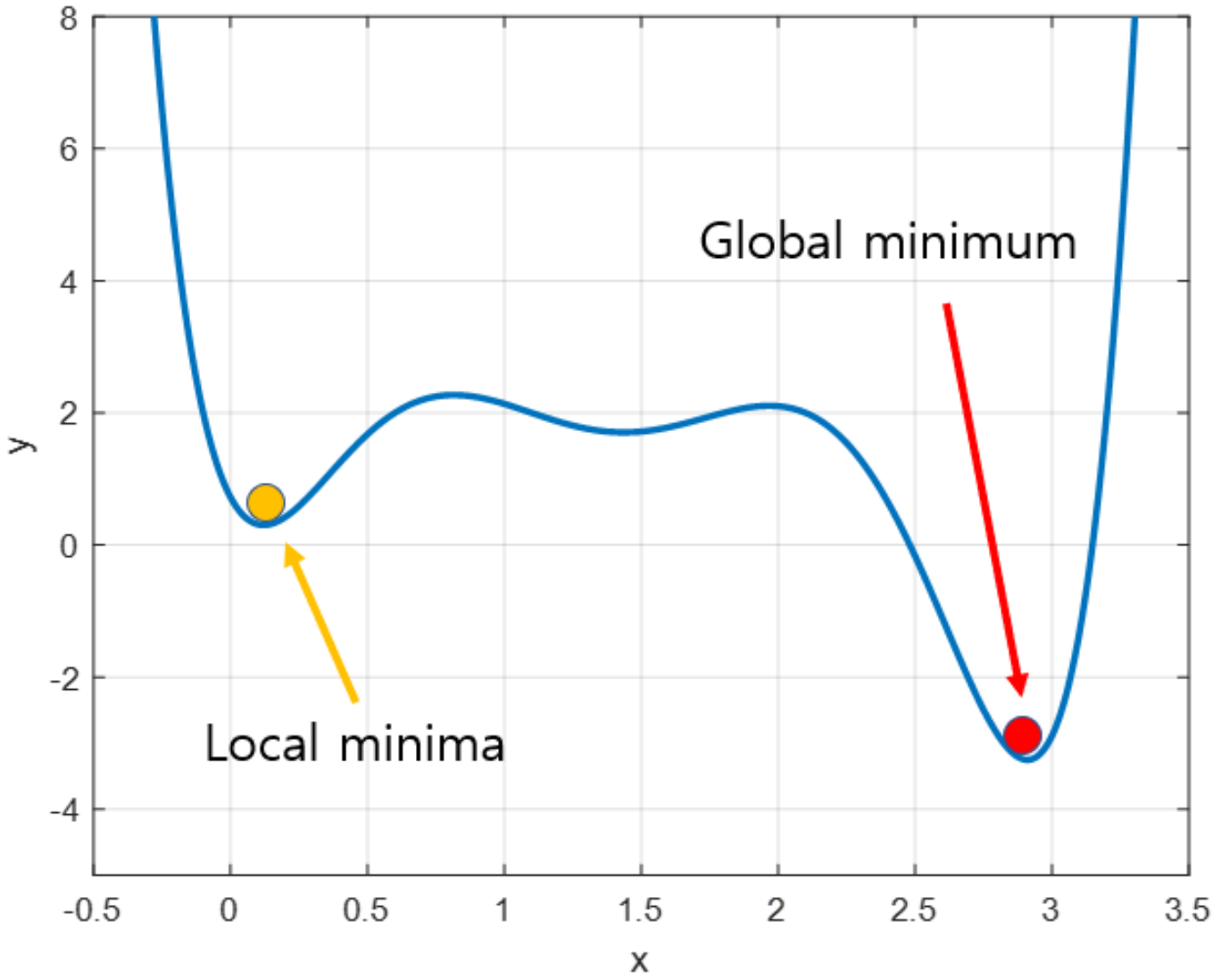



Gradient Descent

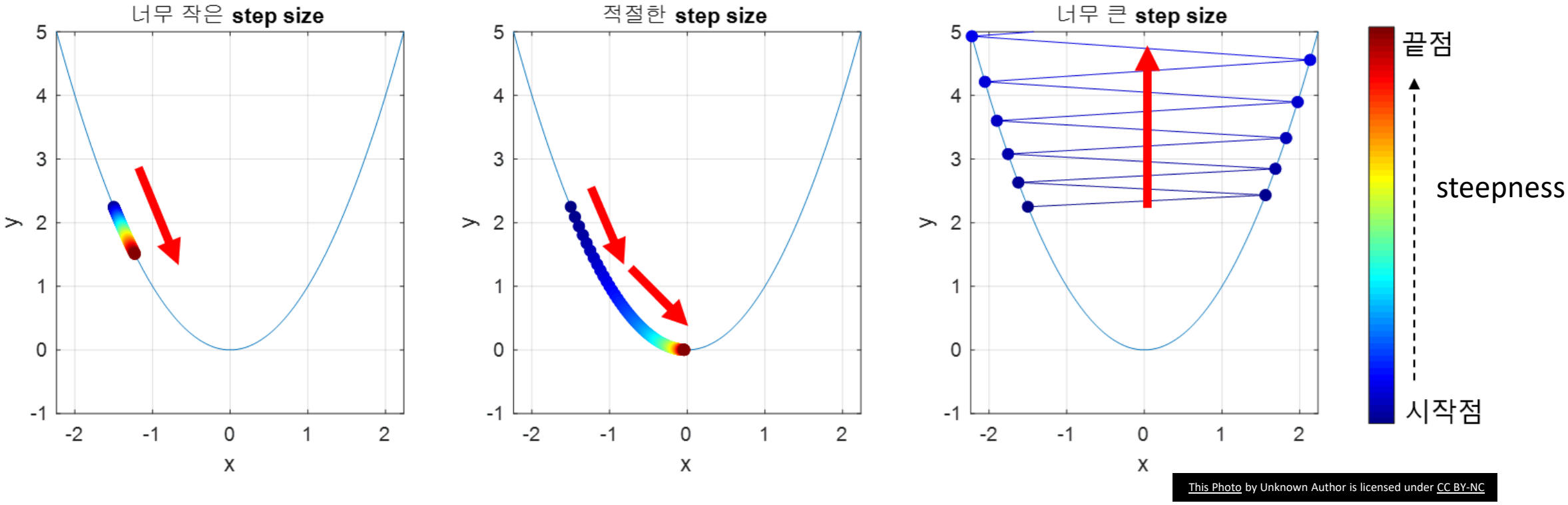


Local vs global minimum

We need enough momentum not to get stuck in a local minimum

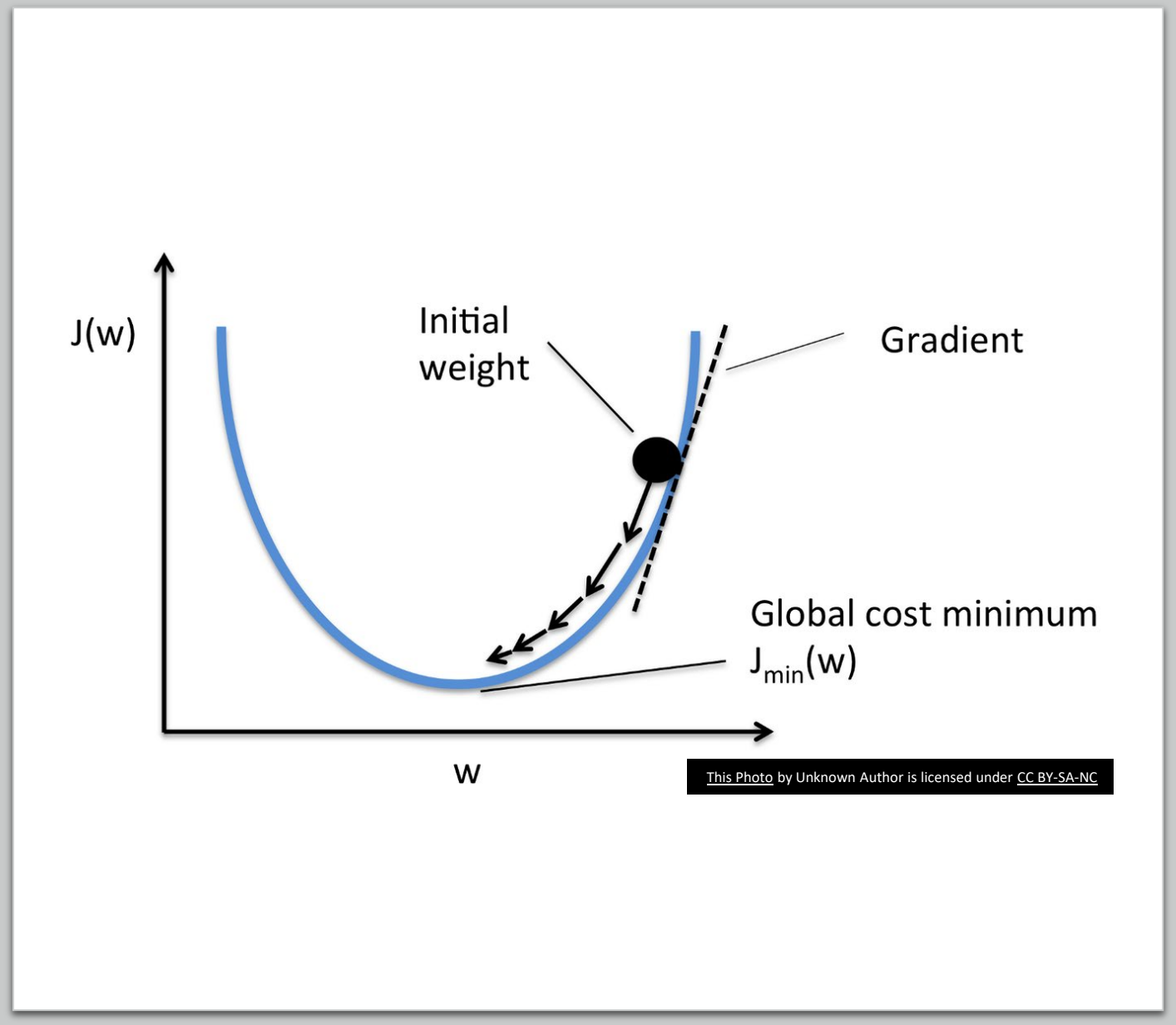


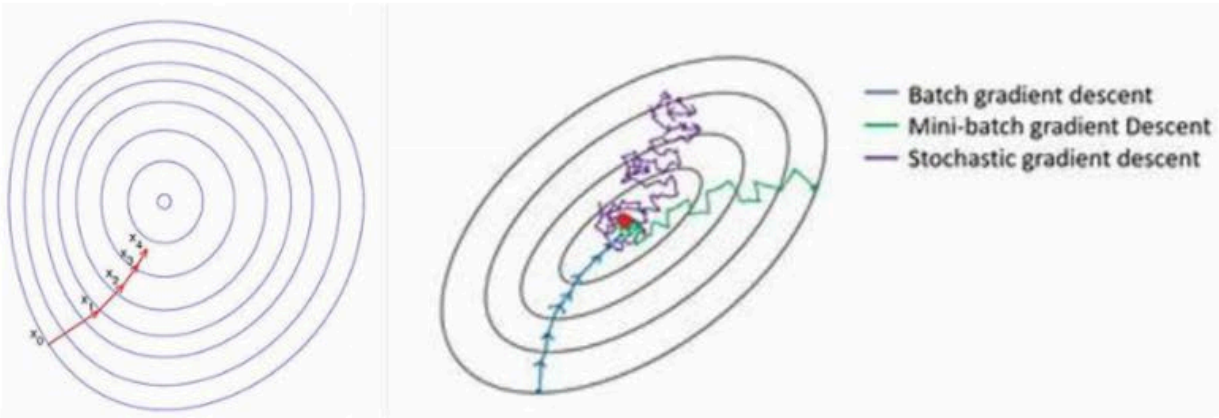
Gradient Descent

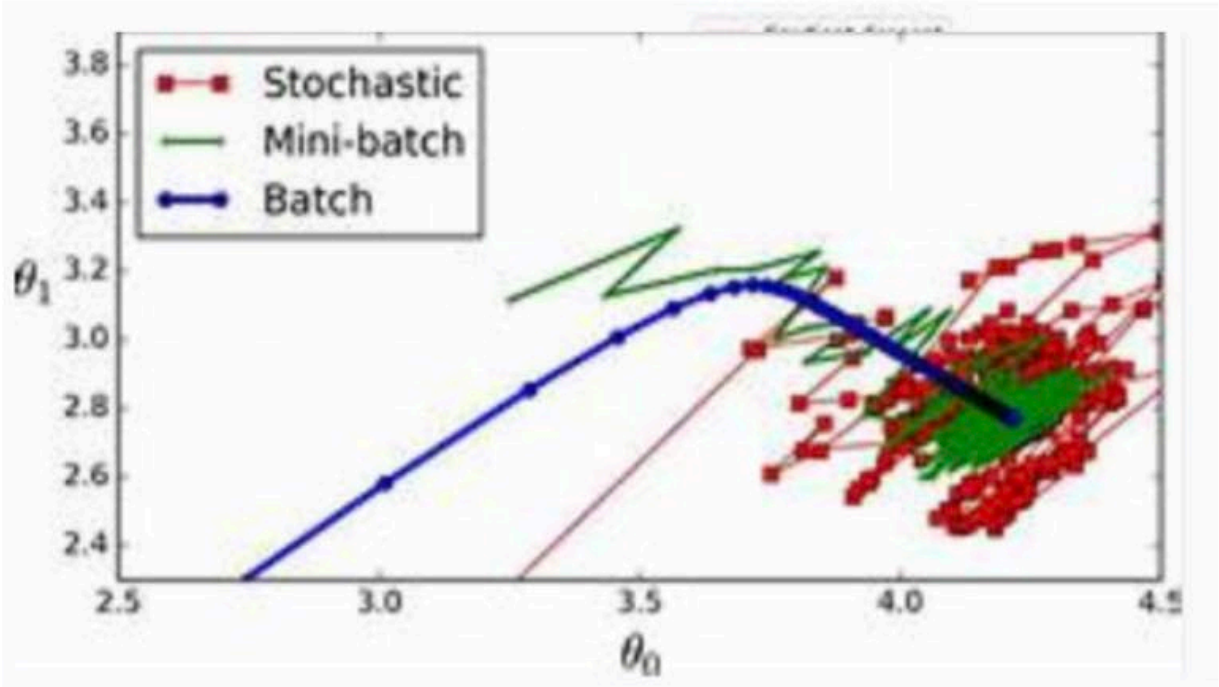


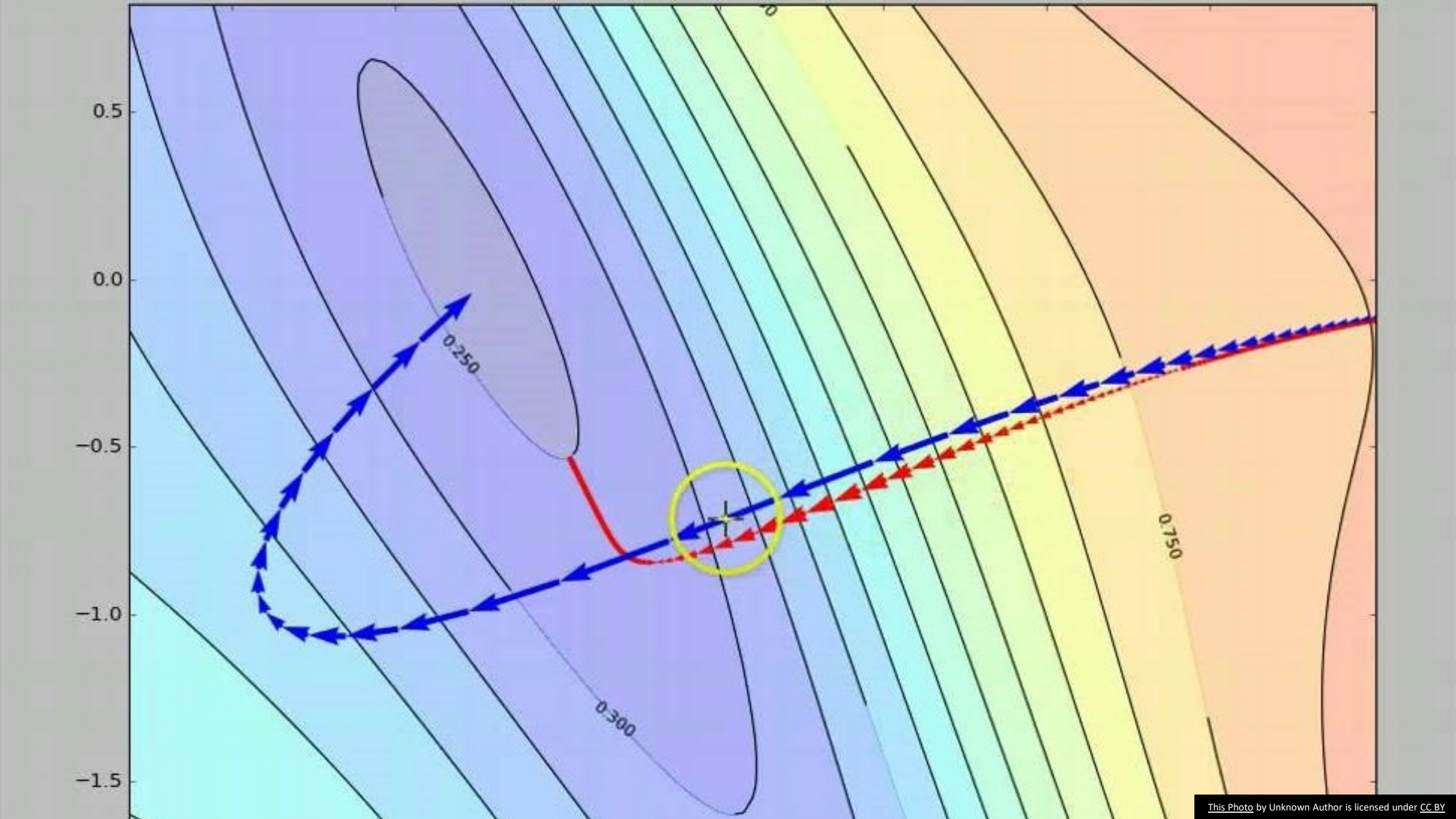
Training the Network

We can back-propagate the result of each training using a simple delta rule, which is gradient descent using a least-square error, eg minimization of the quadratic difference between expected and obtained results.

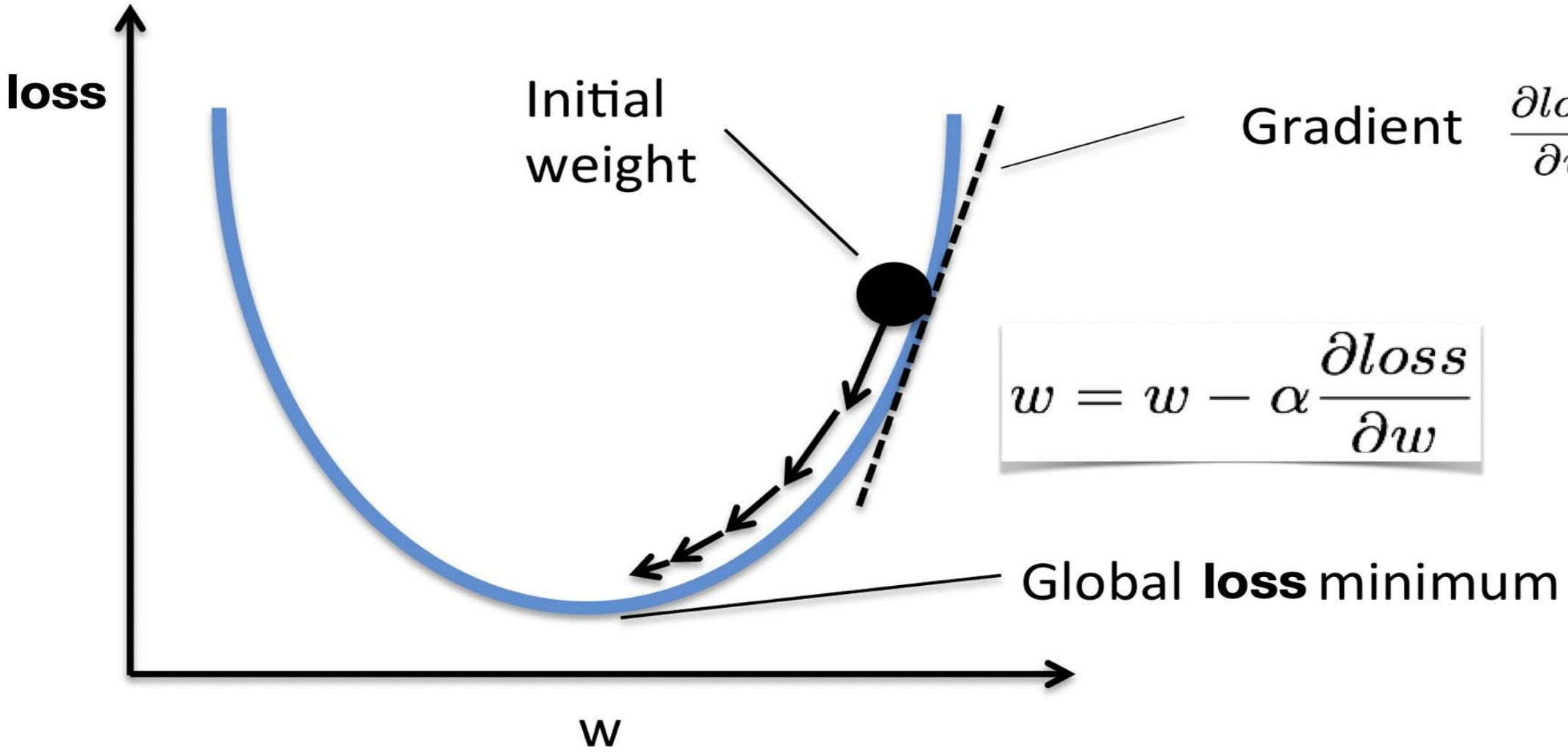



Batch gradient descent
Mini-batch gradient Descent
Stochastic gradient descent


Stochastic
Mini-batch
Batch
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
θ_1
2.5
3.0
3.5
4.0
θ_0



Gradient descent algorithm



$J(\Theta) = \theta_1^2 + \theta_2^2$

假设初始点$\Theta^0 = (1,3)$, 初始的学习率为$\alpha = 0.1$

函数的梯度:

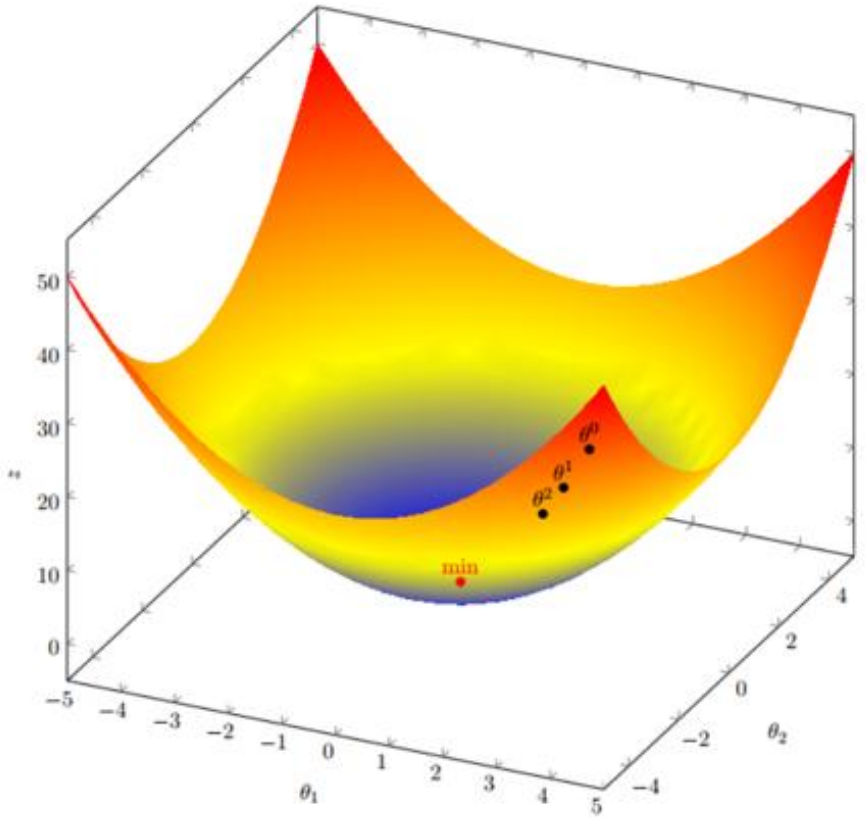
$\nabla J(\Theta) = < 2\theta_1,\ 2\theta_2 >$

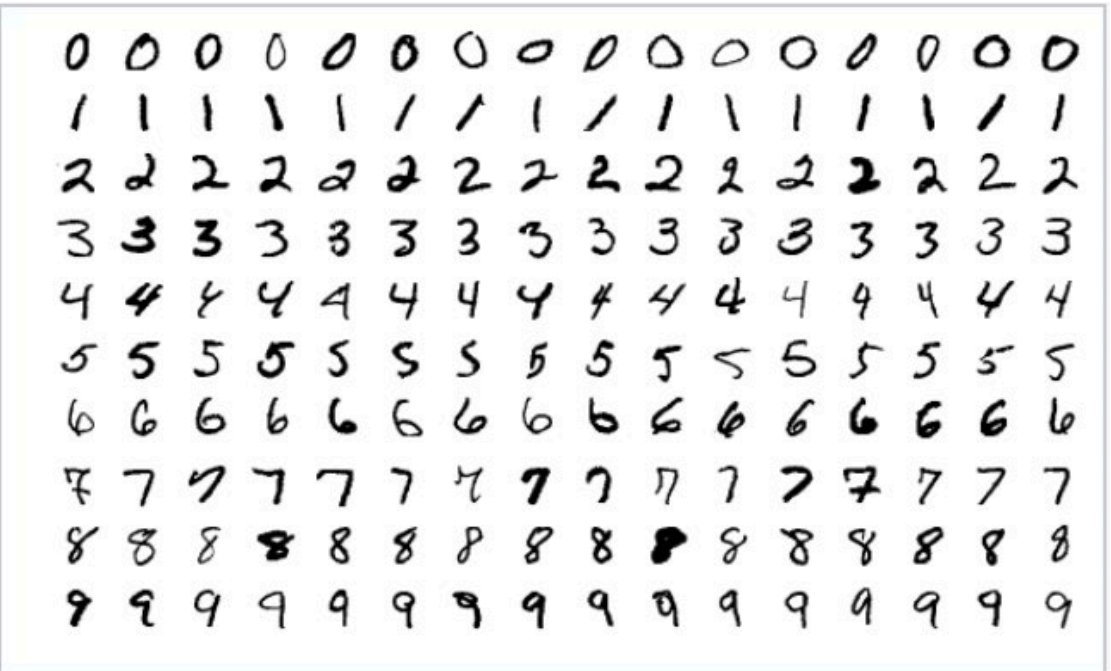
$\Theta^0 = (1,3)$

$$\begin{aligned}\Theta^1 &= \Theta^0 - \alpha\nabla J(\Theta^0)\\ &= (1,3) - 0.1 * (2,6)\\ &= (0.8, 2.4)\end{aligned}$$

$$\begin{aligned}\Theta^2 &= \Theta^1 - \alpha\nabla J(\Theta^1)\\ &= (0.8,2.4) - 0.1 * (1.6,4.8)\\ &= (0.64, 1.92)\end{aligned}$$

$$\begin{aligned}\Theta^3 &= \Theta^2 - \alpha\nabla J(\Theta^2)\\ &= (0.64,1.92) - 0.1 * (1.28,3.84)\\ &= (0.512, 1.536)\end{aligned}$$





Sample images from MNIST test dataset

MNIST is the “Hello World” of deep learning.

MNIST is a set of images of handwritten digits

The problem is to classify each greyscale image into the correct category, namely '0', '1'..., '9'

There are 60,000 **training** images and 10,000 **test** images

Terminology

In a classification problem, category = class

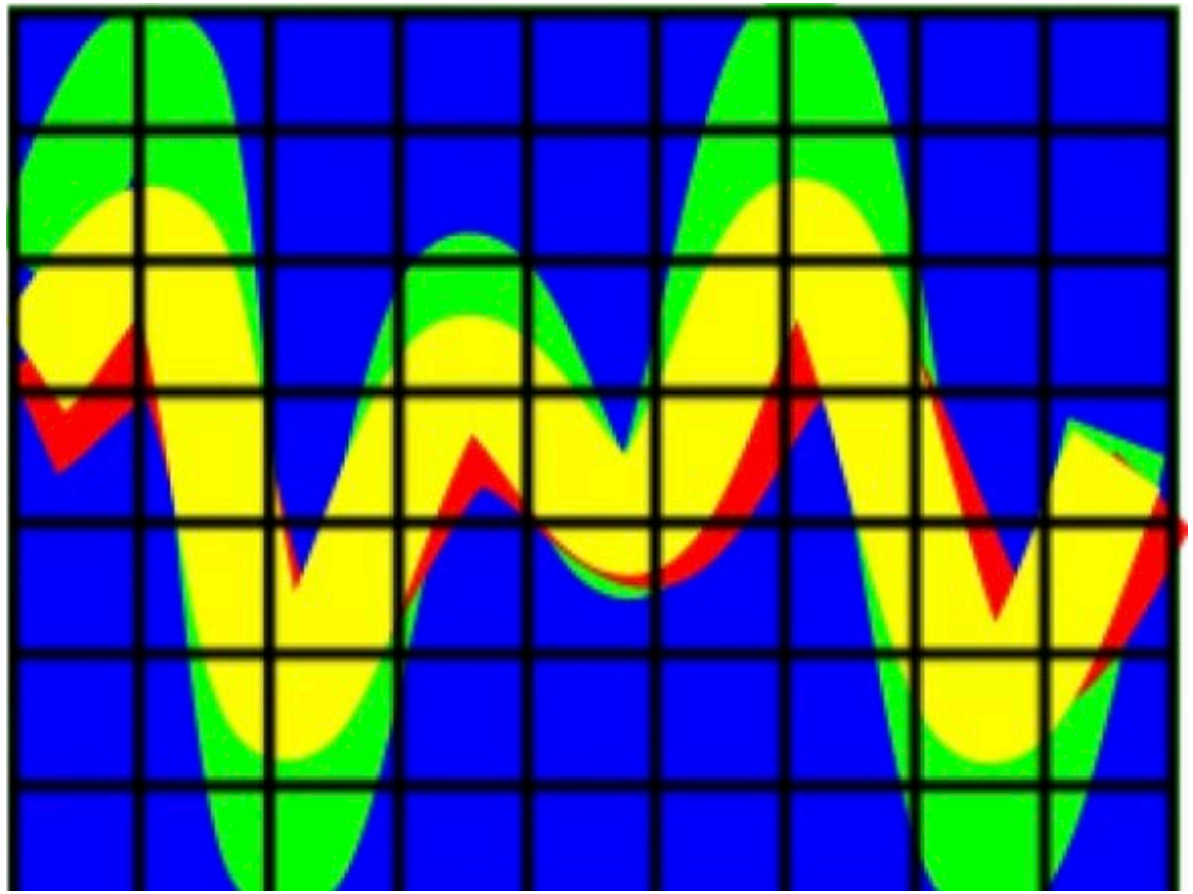
Data points = samples

Class associated with a specific sample = label

Classifying an image

Suppose we have a basic image, with dimension (n x n)

here 7 x 9



Pixel length = 7

Pixel width = 9

Data is stored in special multidimensional arrays - tensors

There are 60000 greyscale images in the training set

Each image is 28 pxl x 28 pxl

- the training set is a data container with 60000 x 28 x 28 elements
- the training labels are stored in a 60,000 element vector

Axis 0 (the first axis) is always the samples axis

- `train_images[0]` - the first 28 x 28 image
- `train_images[1]` - the second 28 x 28 image

Re-shaping

Tensors elements can be redistributed in an operation known as **reshaping**

- the original (60000, 28, 28) MNIST data was reshaped
- `train_images = train_images.reshape((60000, 28 * 28))`

The second **softmax** layer outputs a vector whose elements form a **probability distribution**

- the numbers are nonnegative and sum to one – a probability distribution
- outputs are interpreted as probabilities of membership of each class: the probability that the input sample is labeled '0' or '1' or '2' and so on

Or in other words, this describes the probability that the image has been correctly classified as 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9

In [ ]:

```
# download
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# inspect the first image using the matplotlib graphics library
import matplotlib.pyplot as plt
xmpl_image = test_images[0]
xmpl_label = test_labels[0]
print('sample:')
plt.imshow(xmpl_image, cmap=plt.cm.binary)
plt.show()

# check the label is correct
print('label: ', xmpl_label)
```

In [*]:
```
# imports
from tensorflow.keras import models, layers
```

In [ ]:
```
# an empty network
network = models.Sequential()
```

In [ ]:
```
# add two layers
# don't worry about the complicated arguments - they will be explained later
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28, )))
network.add(layers.Dense(10, activation='softmax'))
```

In [4]:
```
# optimiser, loss and metrics are chosen at compilation
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

We have to decide:

- the number of samples processed in a single pass of the training algorithm – the **mini-batch size**
- the number of complete passes through the entire training set – the number of **epochs**

In [ ]:
```
# training - fit to input data
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

In [ ]:
```
# evaluate on the test set
test_loss, test_acc = network.evaluate(test_images, test_labels)
```

In [ ]:
```
# network prediction for a single sample
# ten numbers, each a probability of class membership
network.predict(test_images[:1])
```

In [ ]:

```
# what is the most probable class?
# the index of the largest element of the output vector
import numpy as np
np.argmax(network.predict(test_images[1]))
```

In [5]:

```
# does the prediction agree with the label?
print(raw_test_labels[0])
```

```
7
```

In [ ]:

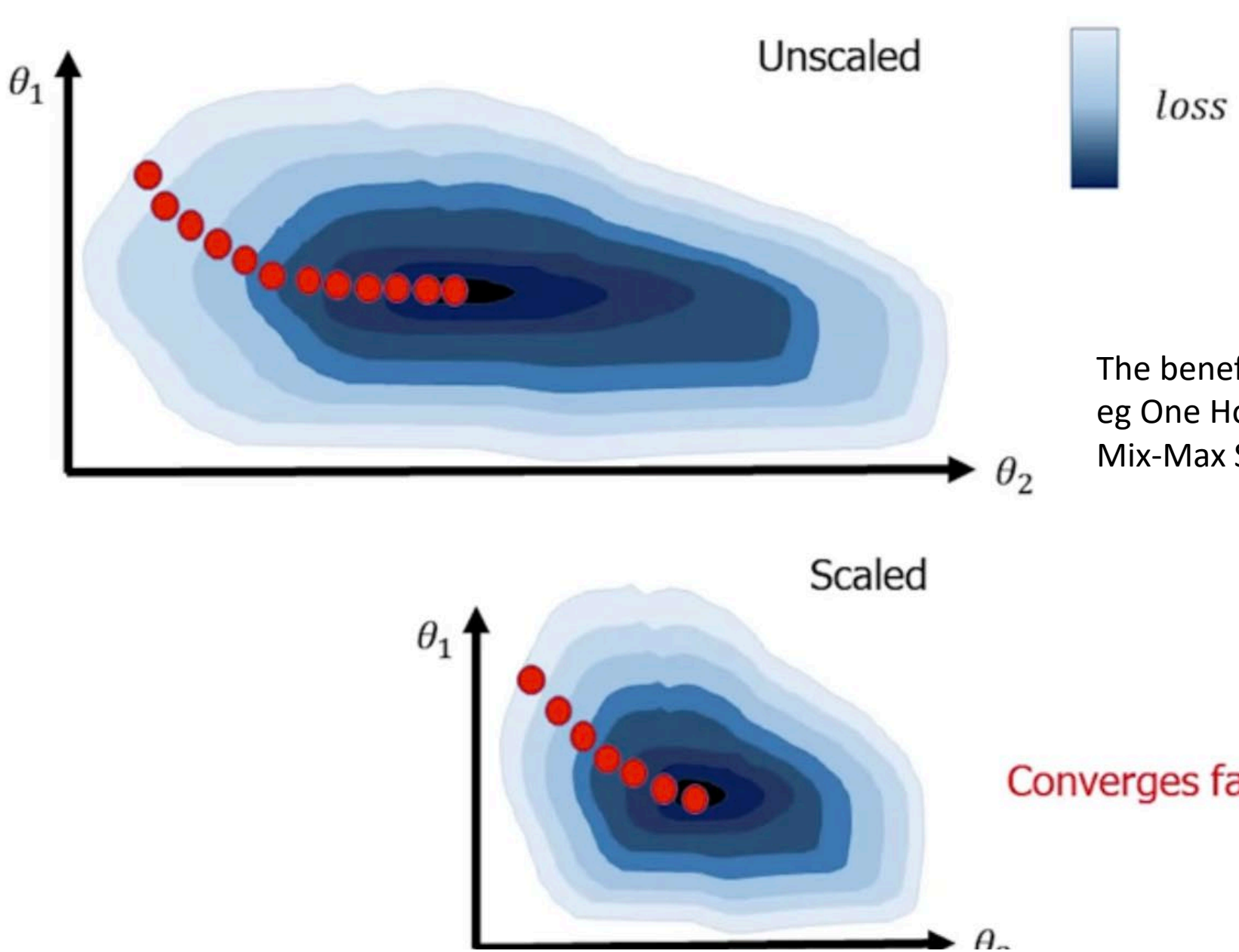
```
import matplotlib.pyplot as plt
plt.imshow(raw_test_images[0], cmap=plt.cm.binary)
plt.show()
```

In [ ]:

```
# reshape flattens 28 x 28 array to a vector of 784 elements
train_images = train_images.reshape((60000, 28 * 28))
test_images = test_images.reshape((10000, 28 * 28))

# cast as floats and rescale from to [0, 1]
train_images = train_images.astype('float32') / 255
test_images = test_images.astype('float32') / 255
```

In [ ]:

```
# encode with the convenient to_categorical function
from tensorflow.keras.utils import to_categorical

orig_label = test_labels[0]
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

# check encoding
print('\'', orig_label, '\'', 'as one-hot vector:\t', test_labels[0], sep='')
```



The benefits of scaling eg One Hot Encoding or Mix-Max Scaling

Converges faster!

Evaluation of Loss

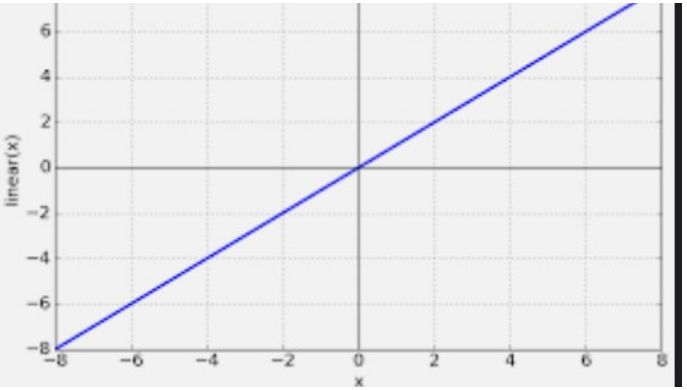
```
In [ ]: 1 test_loss, test_acc = network.evaluate(test_images, test_labels)
```

Glossary

multidimensional array	tensor
dimension	axis
number of dimensions	rank
length	number of elements in a 1D array/along a tensor axis
size	number of matrix rows and cols
shape	lengths of each tensor axis

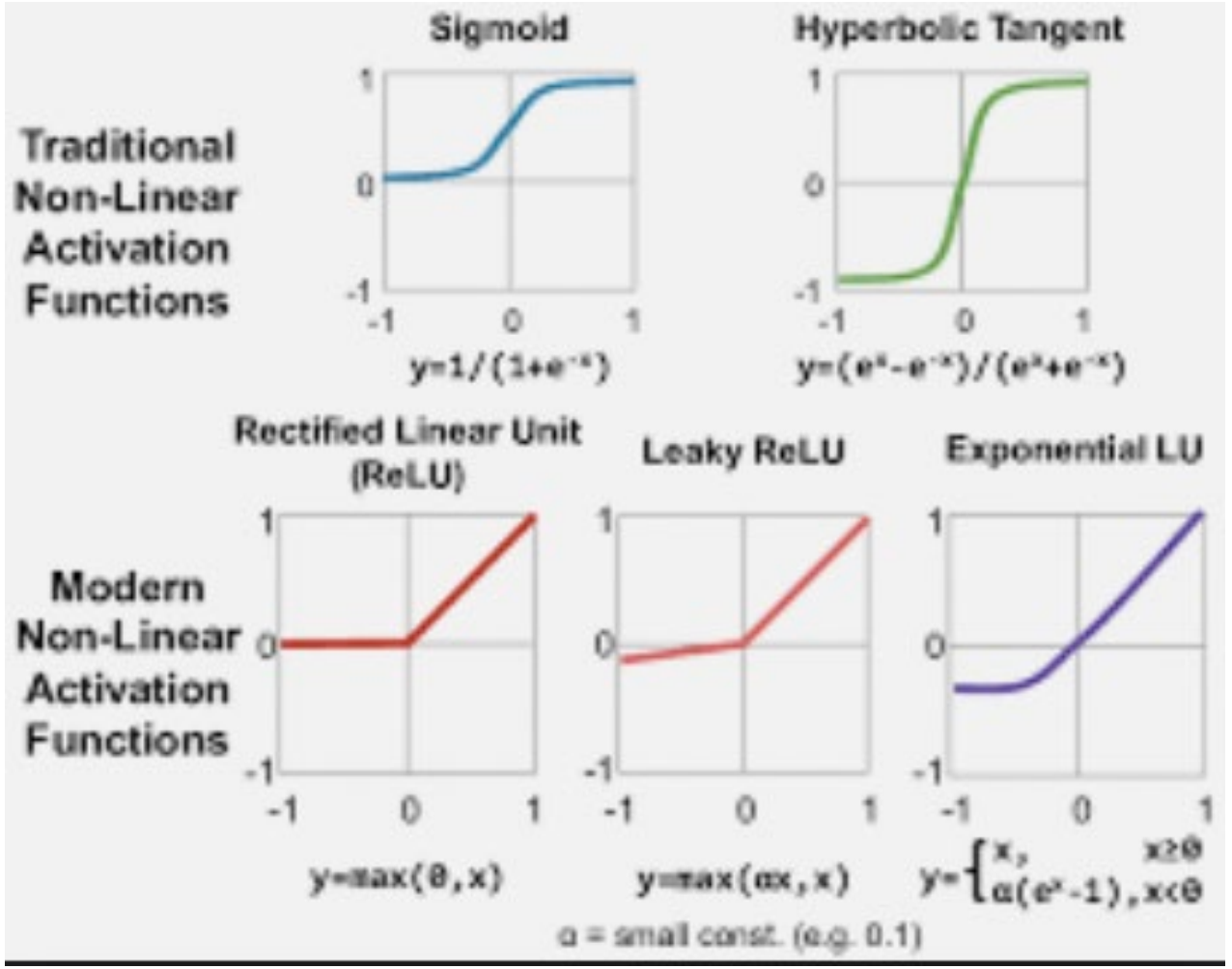
Different activation functions can be used. Traditional activation functions include:-

Linear
Sigmoid (ϕ)
Hyperbolic Tangent (Tanh)



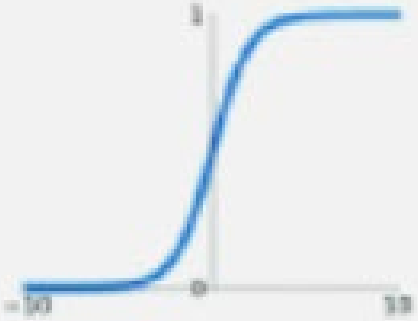
Linear

$y = mx + c$



Sigmoid

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

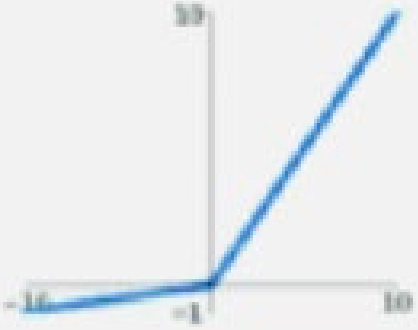


ReLU

$$\max(0, x)$$

Leaky ReLU

$$\max(0.1x, x)$$



Each layer in a neural network performs a non-linear data transformation - $f(w \cdot x + b)$ - where f is the activation function and $w \cdot x + b$ is an affine transformation

The network as a whole, in a series of activations and affine transformations implements a complex transformation on a high dimensional geometric object (the data)

Mini-batch = sub-group

Data is rarely processed in its entirety

Data is broken up into **mini-batches**

- `train_images[:128]` a mini-batch of the first 128 images
- `train_images[128:256]` the second mini-batch
- `train_images[128 * n, 128 * (n + 1)]` the $n+1$'th mini-batch

Training Loop

0. Fill weight and bias tensors with small random values
1. Draw a mini-batch of training samples `x` and corresponding labels `y`
2. The network makes a prediction `y_pred` (**forward pass**)
3. Calculate how much `y_pred` differs from `y` - the loss
4. Update all parameters to lower the loss on this mini-batch
5. Exit or return to 1.

End of the loop

The training loop eventually terminates

The network has been 'trained'

Steps 0 - 3 are simple tensor operations

Step 4: Update all parameters to lower the loss on this mini-batch