Introduction to Artificial Neural Networks

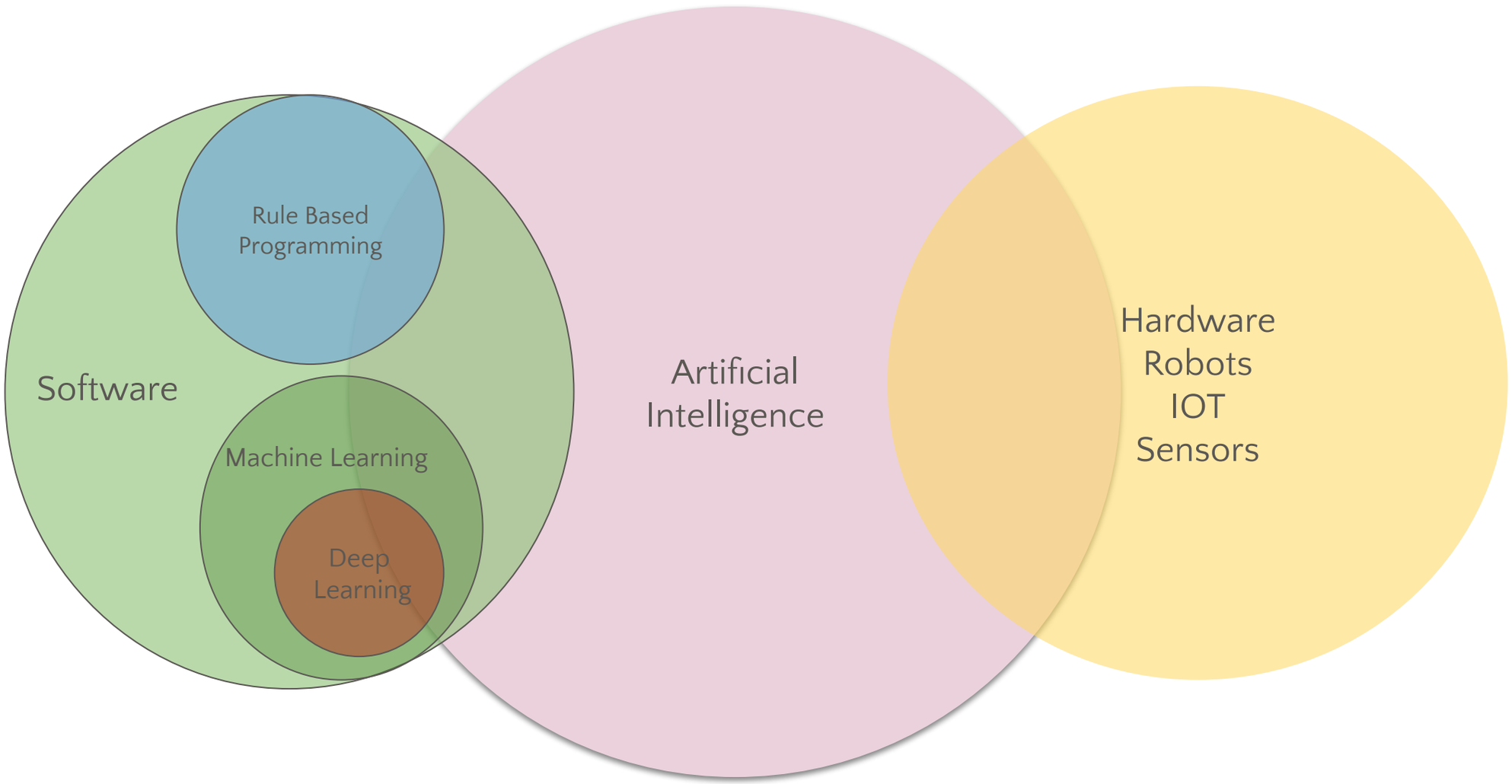
Certificate Course on Artificial Intelligence & Deep Learning by IIT Roorkee

Course Outline

1. Introduction to Linux (Self-Paced)
2. Python for Machine Learning (Self-Paced)
3. Statistics Foundations (Self-Paced)
4. Introduction to Machine Learning and Deep Learning
5. Data Preprocessing, Regression - Build end-to-end Machine Learning Project
6. Classification
7. Machine Learning Algorithms
8. → Introduction to Artificial Neural Networks with Keras
9. Training Deep Neural Networks
10. Custom Models and Training with TensorFlow
11. Loading and Preprocessing Data with TensorFlow
12. Deep Computer Vision using Convolutional Neural Network
13. Processing Sequences Using RNNs and CNNs
14. Natural Language Processing Concepts and RNNs
15. Generative Learning Using autoencoders and GANs
16. Reinforcement Learning

Overview of where we are

- AI is the End Objective
- Software (Machine Learning and deep learning) and Hardwares are tools to achieve AI



How much Math do we need for Deep Learning

How much Math do we need for Deep Learning

- You should have basic understanding of differentiation and linear algebra
- As opposed to the general belief, the maths is not that big an issue.
- Interviews are more around how would you solve a problem or understanding of how ML works
- Most of the algorithms are already there. You should know how to use them.

How much Machine Learning is needed for Deep Learning?

How much Machine Learning is needed for Deep Learning?

You should know

- How to represent data.
- The gradient descent
- How to use ML algorithms: Decision Trees, SVM etc.

Agenda for this Session

- Gradient Descent from Scratch (Hands On)
- Introduction to ANNs
- BackPropogation from Scratch
- Activation Functions
- Implement one using **Keras** ANN to tackle the **MNIST** fashion classification problem

Using TensorFlow in CloudxLab

- Let's get the latest code
- Go to Jupyter
- Open New Terminal
- Switch to the repository
  - `cd iitr-deep-learning-spl-tf2`
- Get the latest code
  - `git pull origin master`

```
cp backpropog_from_scratch.ipynb my_back.ipynb

git checkout -- backpropog_from_scratch.ipynb

git pull origin master
```

Using TensorFlow in CloudxLab

- We will use TensorFlow 2 in this course
- Here is the quick overview of how to launch TF2 in CloudxLab

Create new Jupyter notebook in TensorFlow 2

a. Login to your lab

b. Open Jupyter

c. On the right side of the screen, click **New**, and then click on **tensorflow2**

Create new Jupyter notebook in TensorFlow 2

Open existing Jupyter notebook in TensorFlow 2

a. Login to your lab

b. Open Jupyter

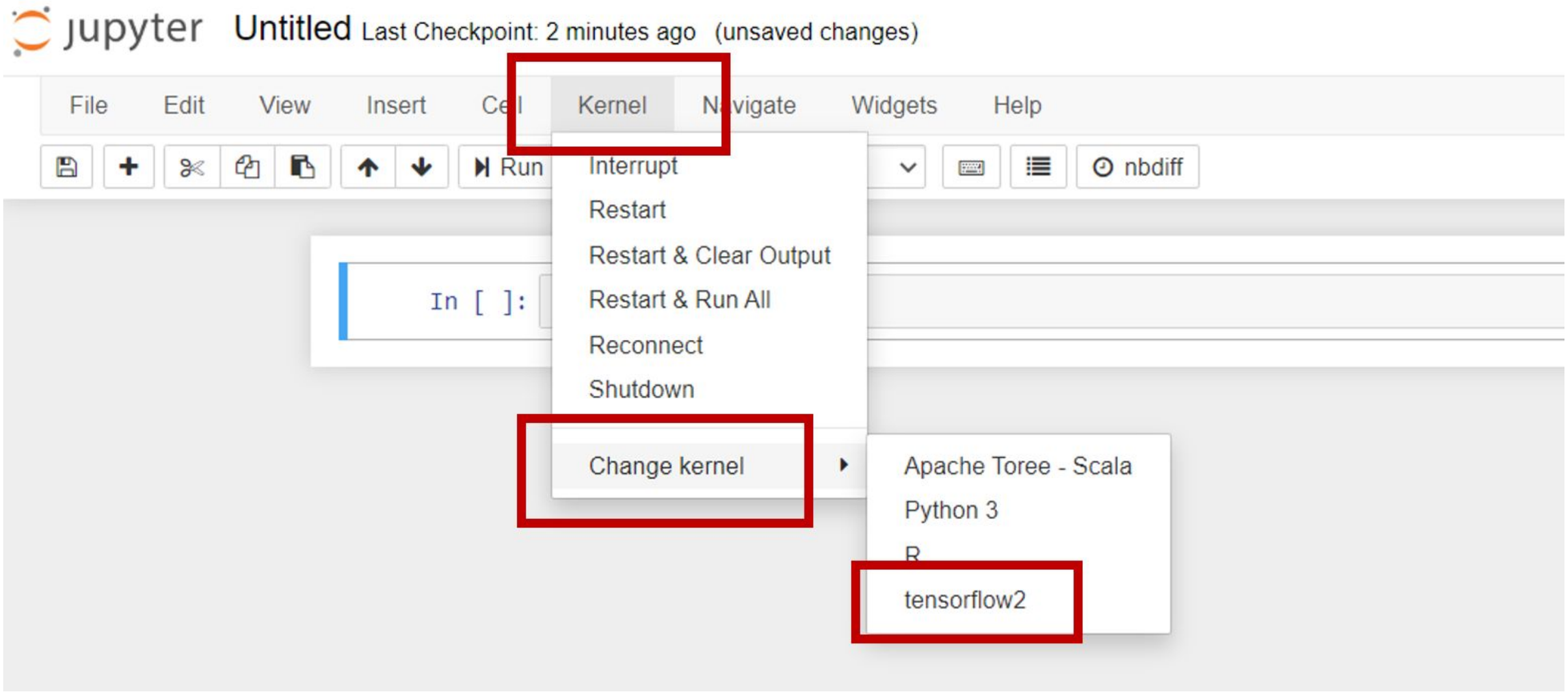
c. Open the existing Jupyter notebook

introduction_to_ann_with_keras.ipynb by clicking on it

d. In the Jupyter notebook click on **Kernel -> Change kernel ->**

tensorflow2 from the menu

Open existing Jupyter notebook in TensorFlow 2



Launch TensorFlow 2 Environment in Console

- Go to terminal
- Type
  - `/usr/local/anaconda/envs/tensorflow2/bin/python`
- Python shell will be launched with TensorFlow 2
- Verify it with below code in the Python shell

```
>>> import tensorflow as tf

>>> tf.__version__
```

Inspirations From Nature

Birds inspired us to fly

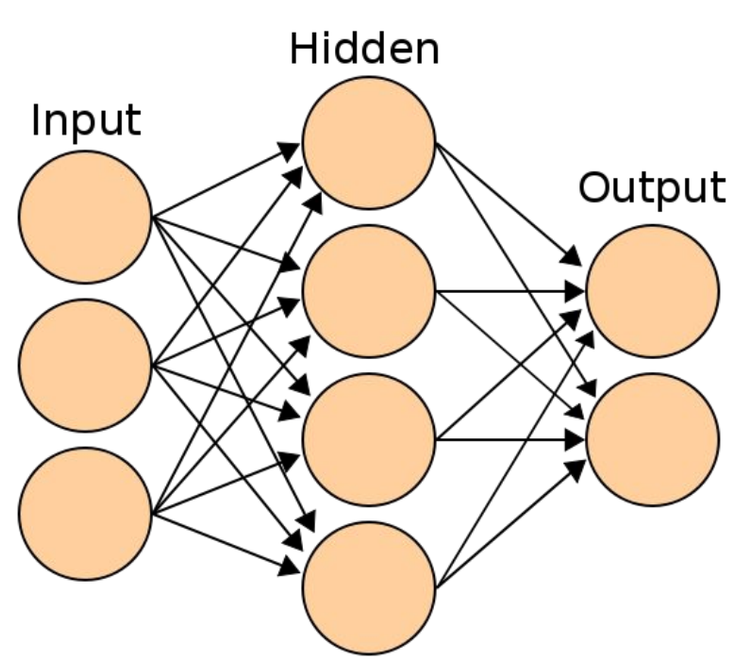
Dragonflys inspired

Helicopters

Inspiration from the Brain

- It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine
- This is the key idea that inspired **Artificial Neural Networks (ANNs)**

Your Brain



Artificial Neural Networks

Power of Artificial Neural Networks

- ANNs are at the very core of Deep Learning
- They are
  - Versatile
  - Powerful
  - Scalable
  - And ideal to tackle
    - Large and highly complex
    - Machine Learning tasks

Applications of Artificial Neural Networks

- Classifying billions of images
  - Google Images

Applications of Artificial Neural Networks

- Powering speech recognition services
  - Apple's Siri

Applications of Artificial Neural Networks

- Recommending the best videos to watch
  - To hundreds of millions of users every day
  - E.g., YouTube

Applications of Artificial Neural Networks

- Learning to beat the world champion
  - At the game of Go
  - By examining millions of past games and
  - Then playing against itself
  - E.g., DeepMind's AlphaGo

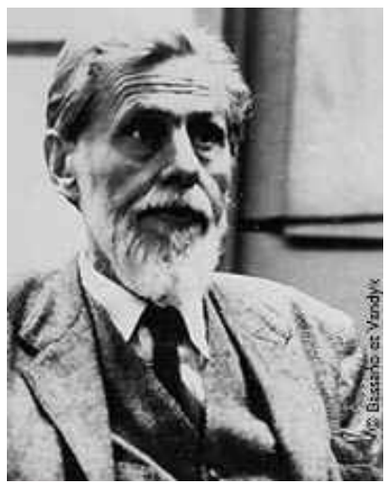
What we will learn?

In this topic we will cover –

- Introduction to **Artificial Neural Networks**
- Tour of the very first **ANN architectures**
- Multi-Layer Perceptrons (MLPs)
- Implement one using **Keras** to tackle the **MNIST** digit classification problem

History of Artificial Neural Networks

- Artificial Neural Networks were
  - First introduced in **1943** by
    - The neurophysiologist **Warren McCulloch** and
    - The mathematician **Walter Pitts**
  - In the paper "**A Logical Calculus of Ideas Immanent in Nervous Activity**"



Warren McCulloch

Walter Pitts

History of Artificial Neural Networks

- They presented a model of
  - How biological neurons might work together
  - In animal brains to perform
  - Complex computations
- This was the first artificial neural network architecture
- Since then many other architectures have been invented

History of Artificial Neural Networks

- The early successes of ANNs until **1960s** led to the
  - Belief that we would soon be conversing with intelligent machines
  - But due to various reasons
    - ANNs entered a long dark era

History of Artificial Neural Networks

- In the early **1980s** there was a revival of interest in ANNs
  - As new network architectures were invented
  - And better training techniques were developed

History of Artificial Neural Networks

- But by the **1990s**, powerful alternative Machine Learning techniques
  - Such as Support Vector Machines were favored by most researchers
  - As they seemed to offer better results and
  - Stronger theoretical foundations

Why ANN's are relevant today?

- Huge quantity of data available
  - To train neural networks
- ANNs frequently outperform
  - Other ML techniques on
  - Very large and complex problems
- Tremendous increase in computing power
  - Which makes it possible to train large neural networks
  - In a reasonable amount of time
  - Also, gaming industry
    - Produced powerful GPU cards

Biological neuron

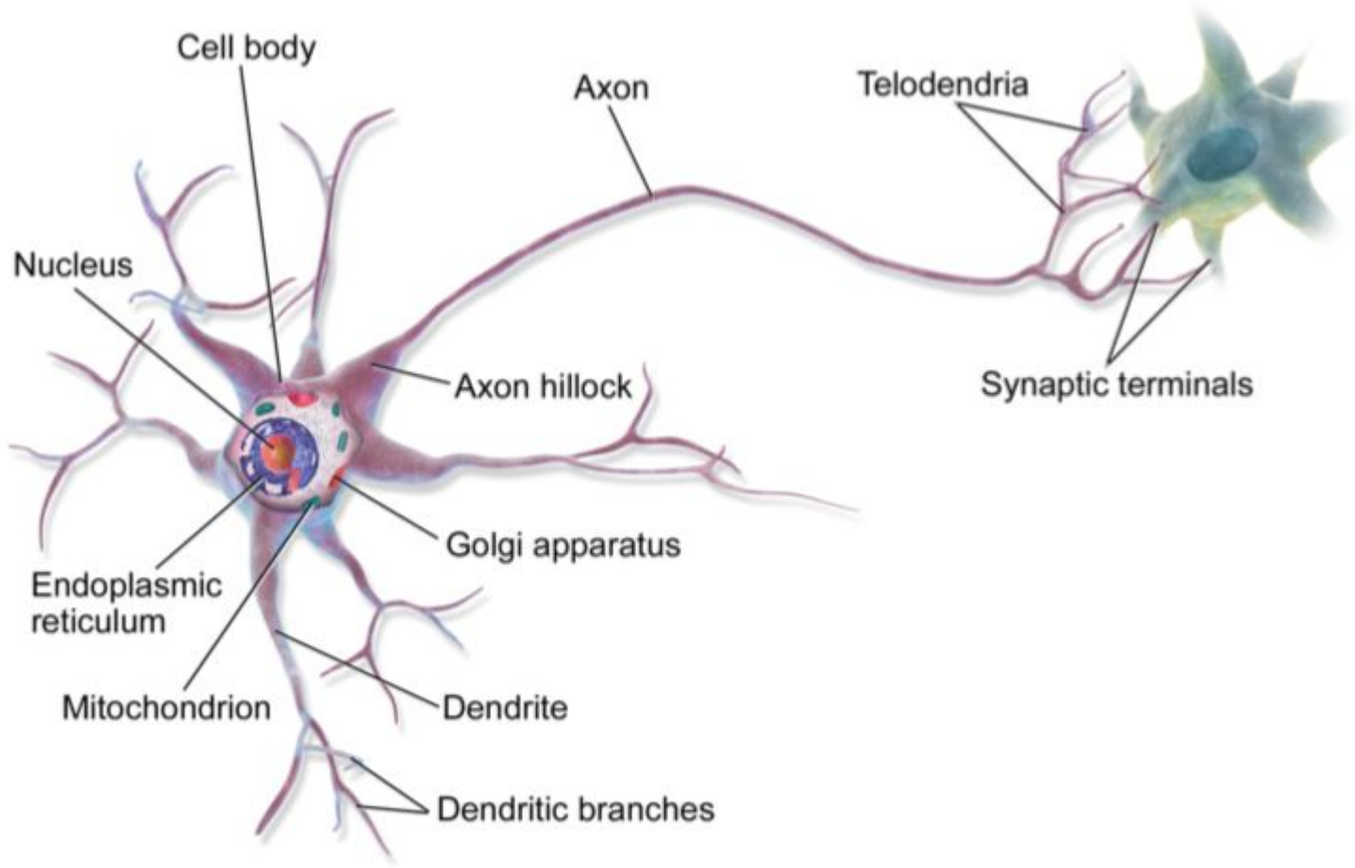
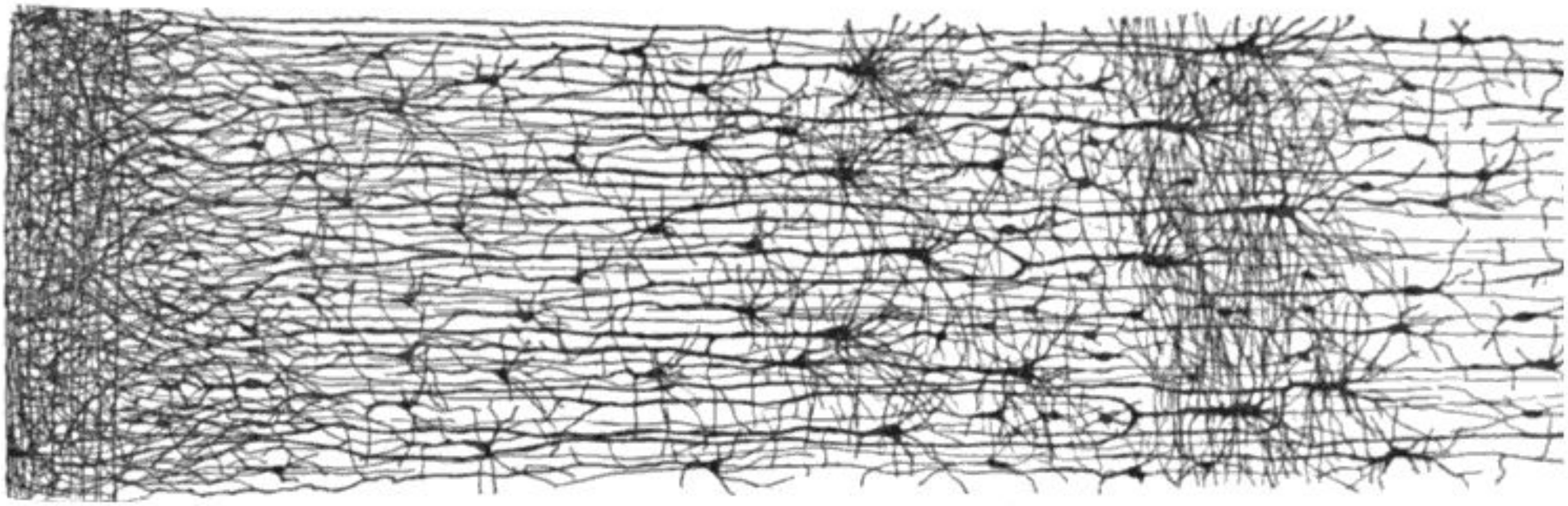


Image by Bruce Blaus (Creative Commons 3.0). Reproduced from https://en.wikipedia.org/wiki/Neuron.

Biological neuron

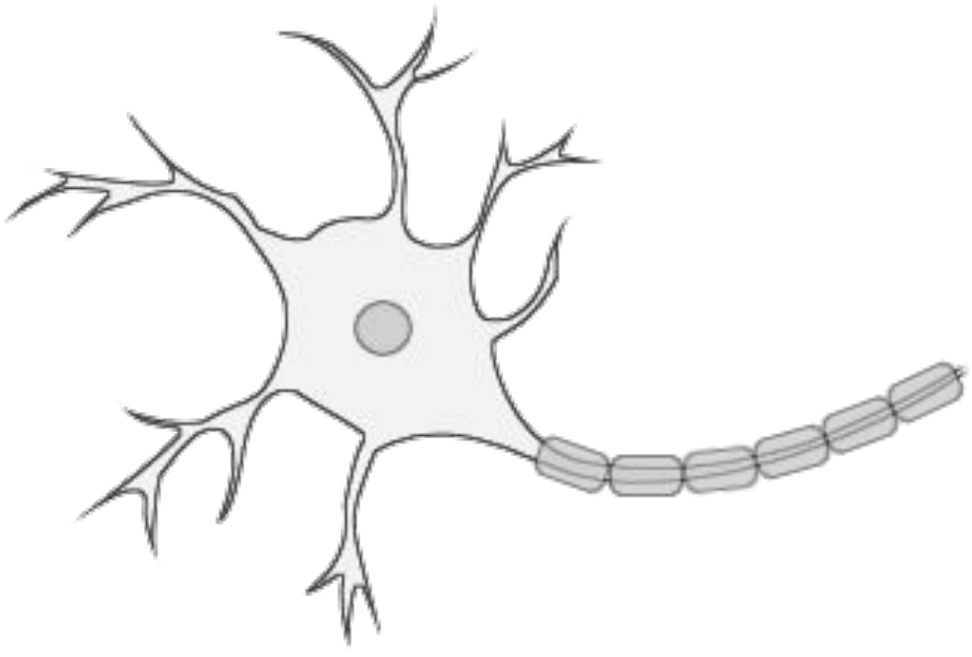


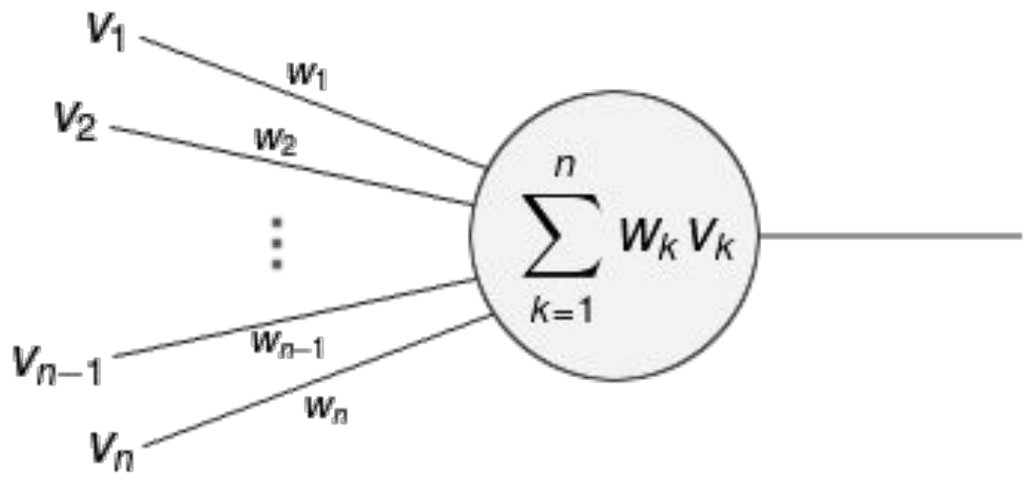
Multiple layers in a biological neural network (human cortex)

https://en.wikipedia.org/wiki/Cerebral_cortex

Logical Computations with Neurons

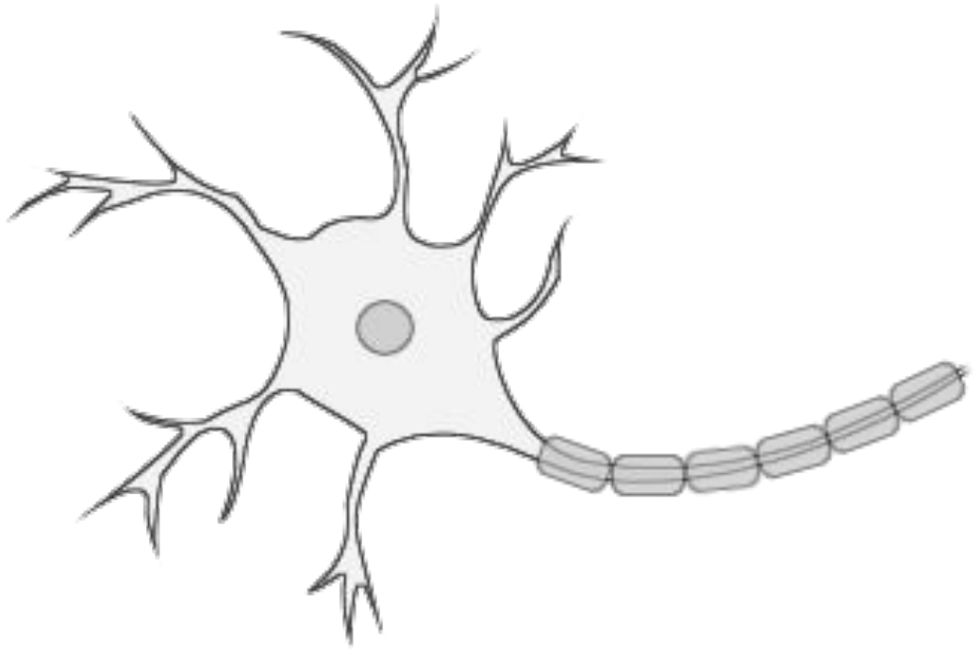
- **Warren McCulloch** and **Walter Pitts** proposed
  - A very simple model of the biological neuron
  - Known as an **artificial neuron**
- It has one or more binary (on/off) inputs and one binary output.

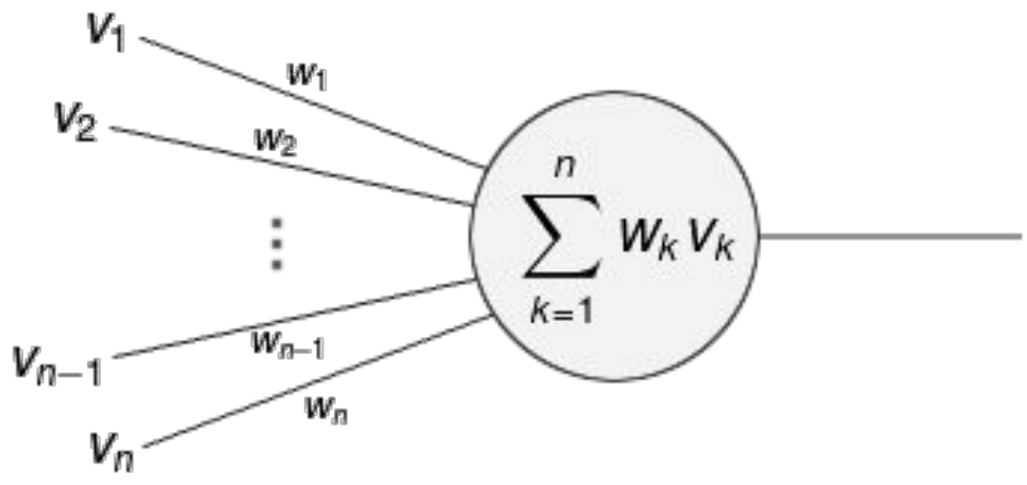




Logical Computations with Neurons

- The artificial neuron simply activates
  - Its output when more than
  - A certain number of its inputs are active

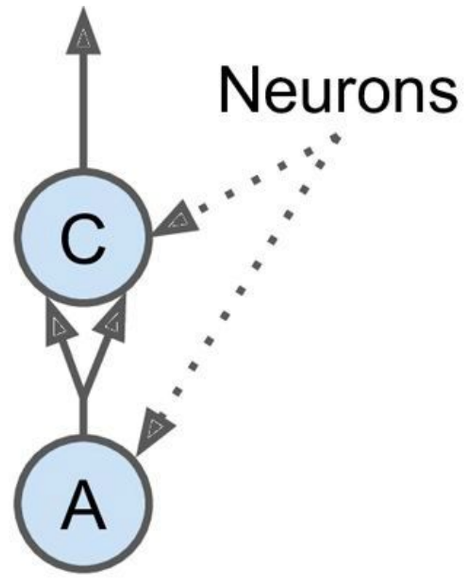




Logical Computations with Neurons

ANN – 1

- If neuron A is activated then
  - Neuron C gets activated as well
    - It receives two input signals from neuron A
  - But if neuron A is off, then neuron C is off as well

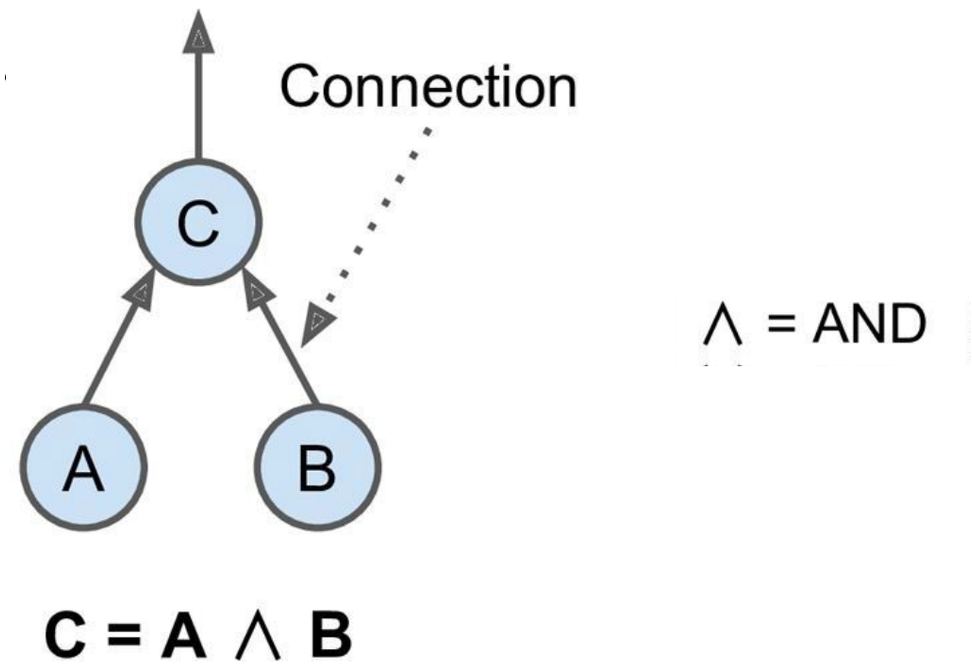


C = A

Logical Computations with Neurons

ANN – 2

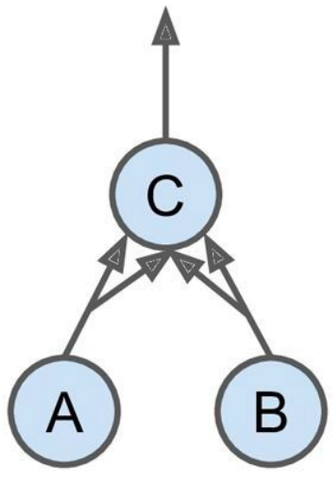
- It performs a logical AND
- Neuron C is activated only
  - When both neurons A and B are activated
- A single input signal is not enough
  - To activate neuron C



Logical Computations with Neurons

ANN – 3

- It performs a logical OR
- Neuron C gets activated if either
  - Neuron A or
  - Neuron B is activated
  - Or both



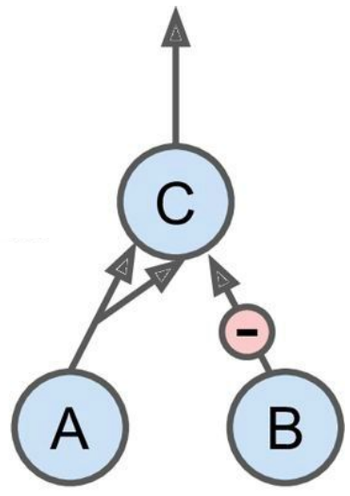
V = OR

C = A ∨ B

Logical Computations with Neurons

ANN – 4

- Neuron C is activated only
  - If neuron A is active and neuron B is off
- And Vice Versa



¬ = NOT

C = A ∧ ¬B

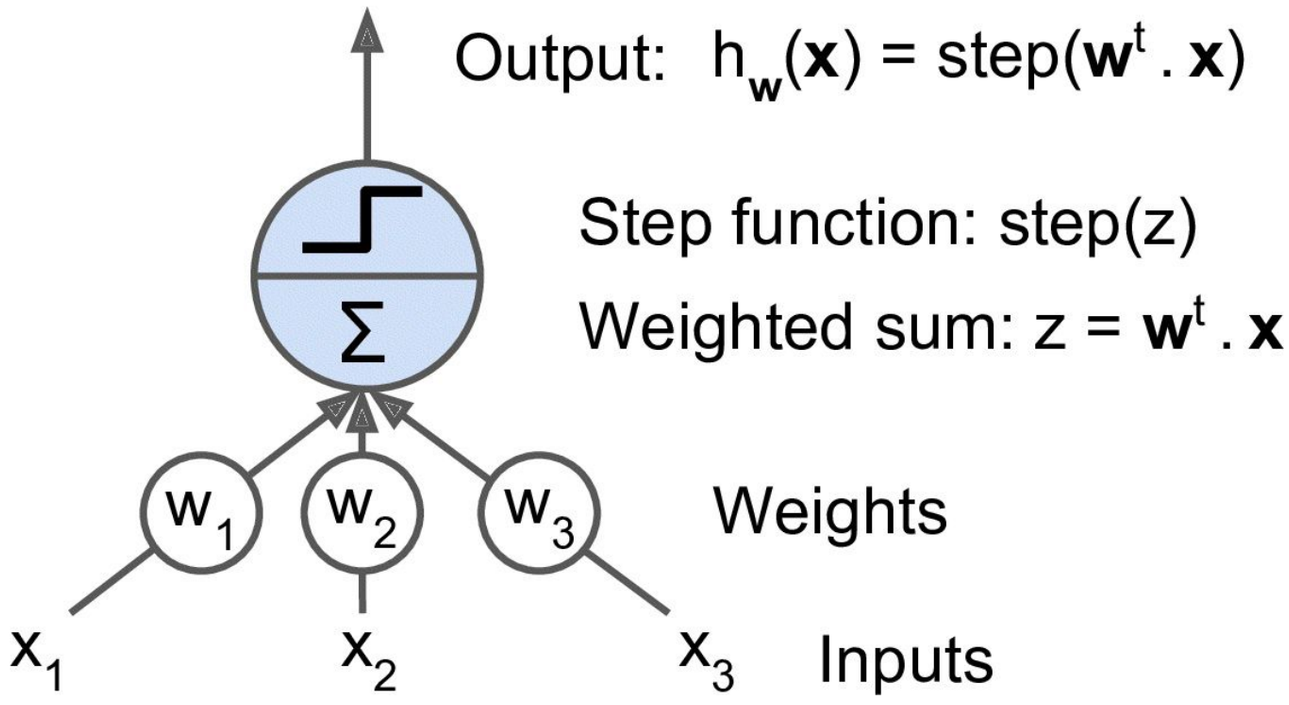
The Perceptron

- It is one of the simplest ANN architectures
- Invented in 1957 by Frank Rosenblatt
- Based on a slightly different artificial neuron called a **linear threshold unit (LTU)**

The Perceptron

Linear threshold unit

- The inputs and output are now numbers, instead of binary on/off values
- Each input connection is associated with a weight



The Perceptron

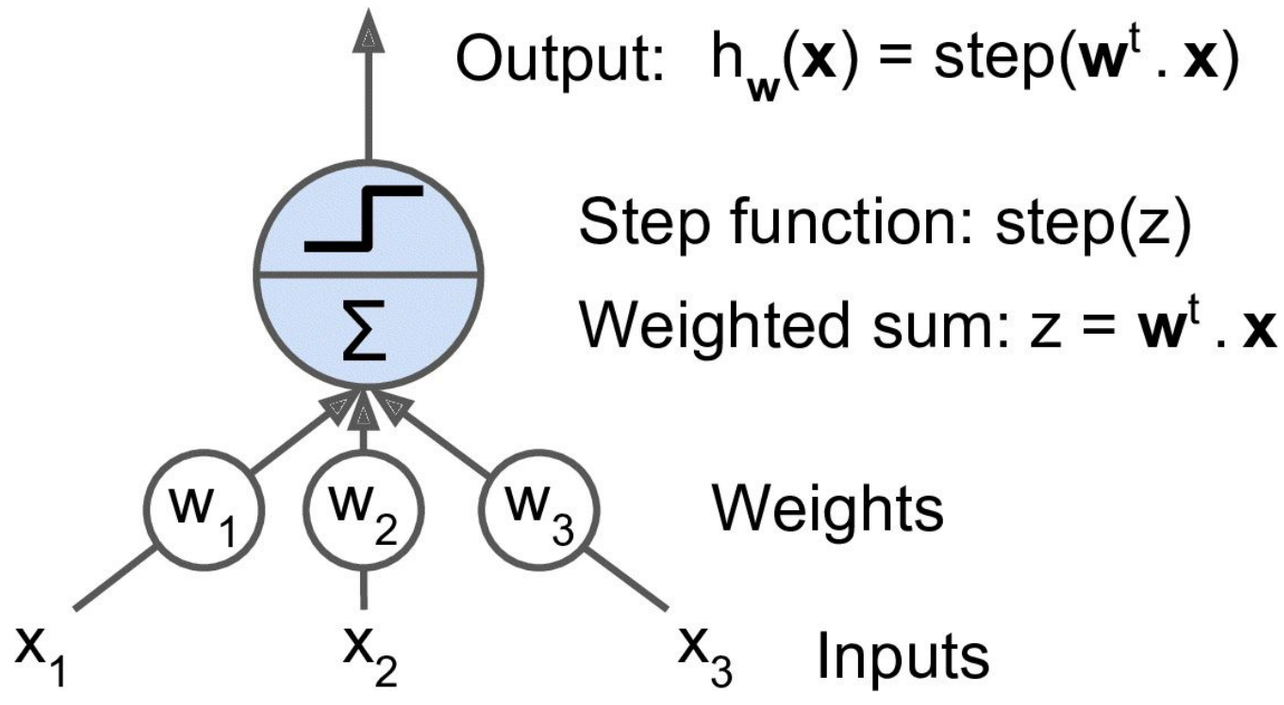
Linear threshold unit

- It computes a weighted sum of its inputs

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = w^T \cdot x$$

- Then applies a step function to that sum and outputs the result

$$h_w(x) = \text{step}(z) = \text{step}(w^T \cdot x)$$



The Perceptron

Linear threshold unit

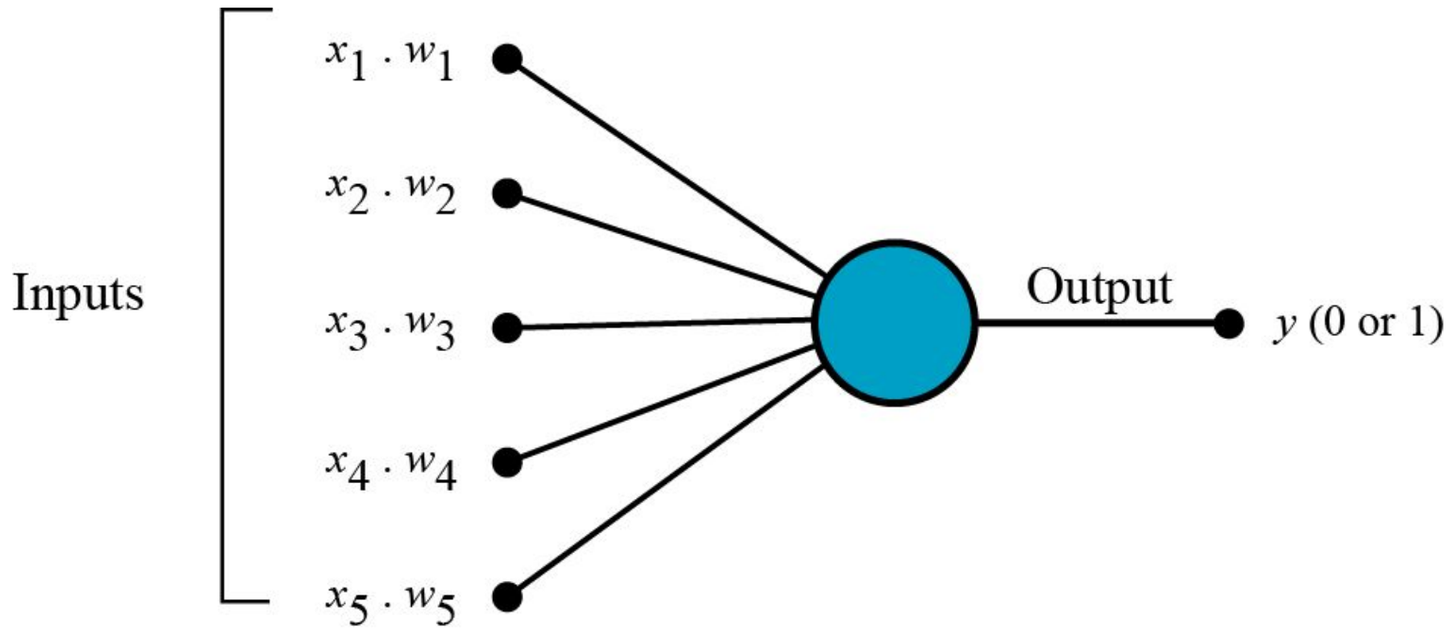
- The most common step function used in Perceptrons is the **Heaviside** step function
- Sometimes the **sign** function is used instead

$$\text{heaviside}\,(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}\,(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

The Perceptron

Linear threshold unit

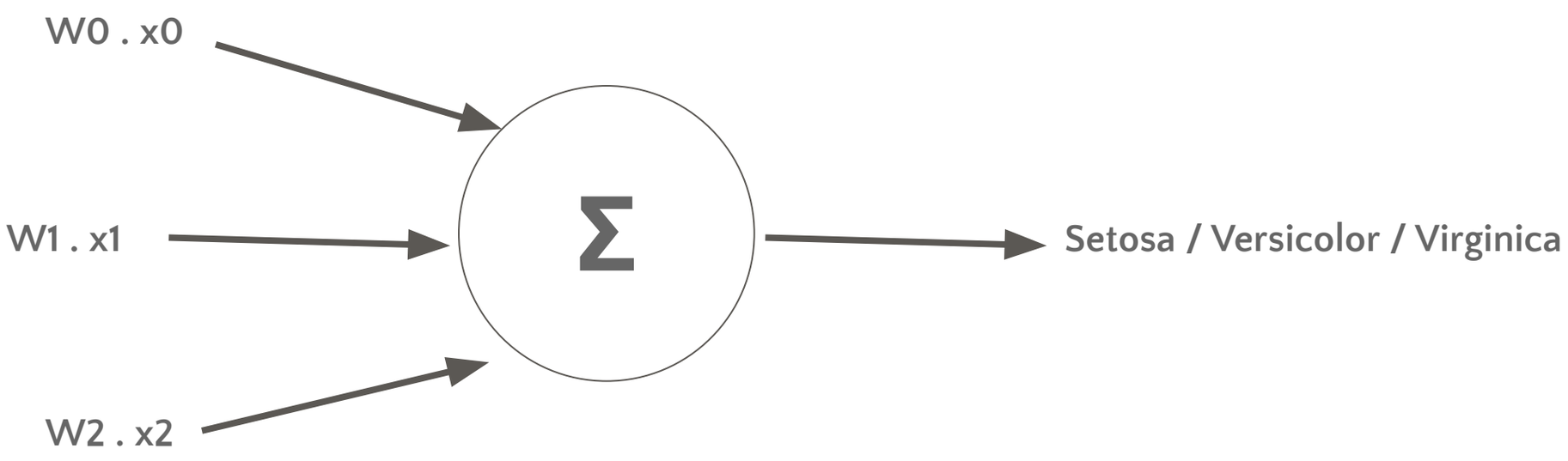
- A single LTU can be used for simple linear binary classification.
- It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class
- Just like a Logistic Regression classifier or a linear SVM



The Perceptron

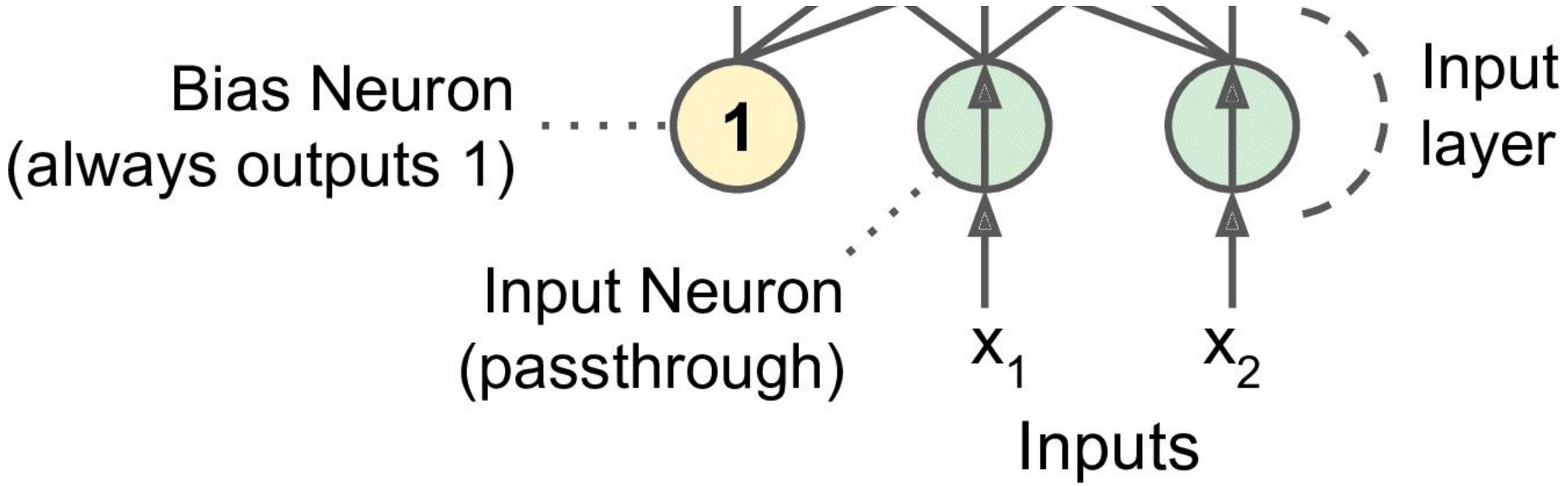
Linear threshold unit

- For example, you could use a single LTU to classify iris flowers based on the petal length and width
- And adding an extra bias feature x0 = 1
- Training an LTU means finding the right values for w0, w1, and w2



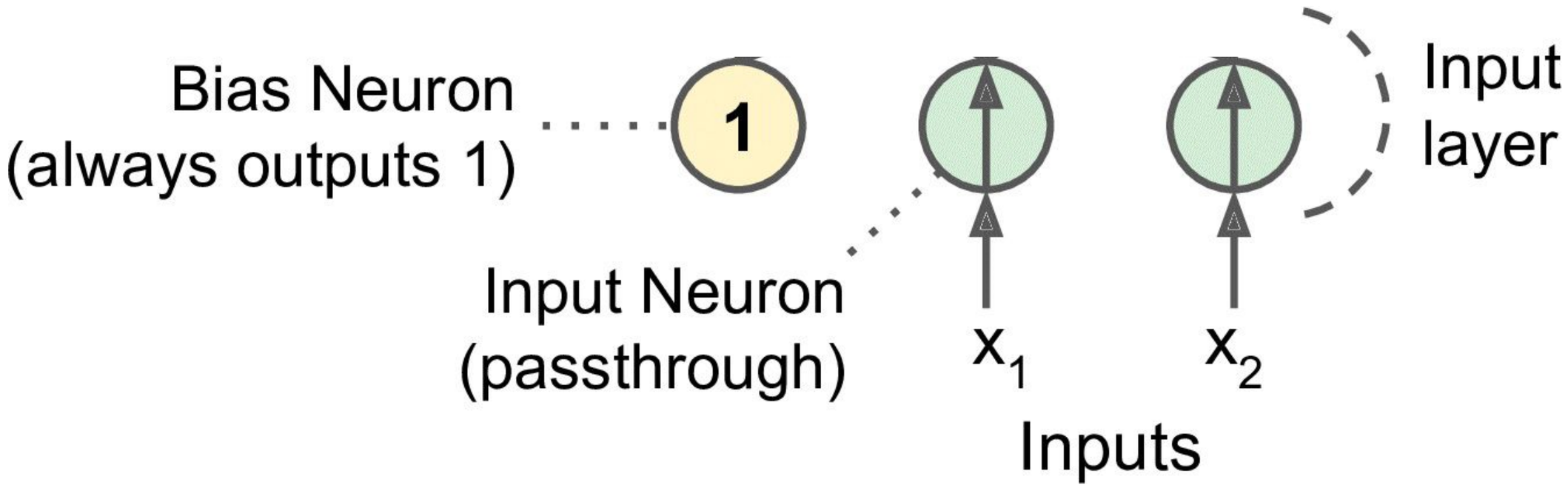
The Perceptron

- A Perceptron is simply composed of a single layer of LTUs, with each neuron connected to all the inputs.
- These connections are often represented using special passthrough neurons called **input neurons**

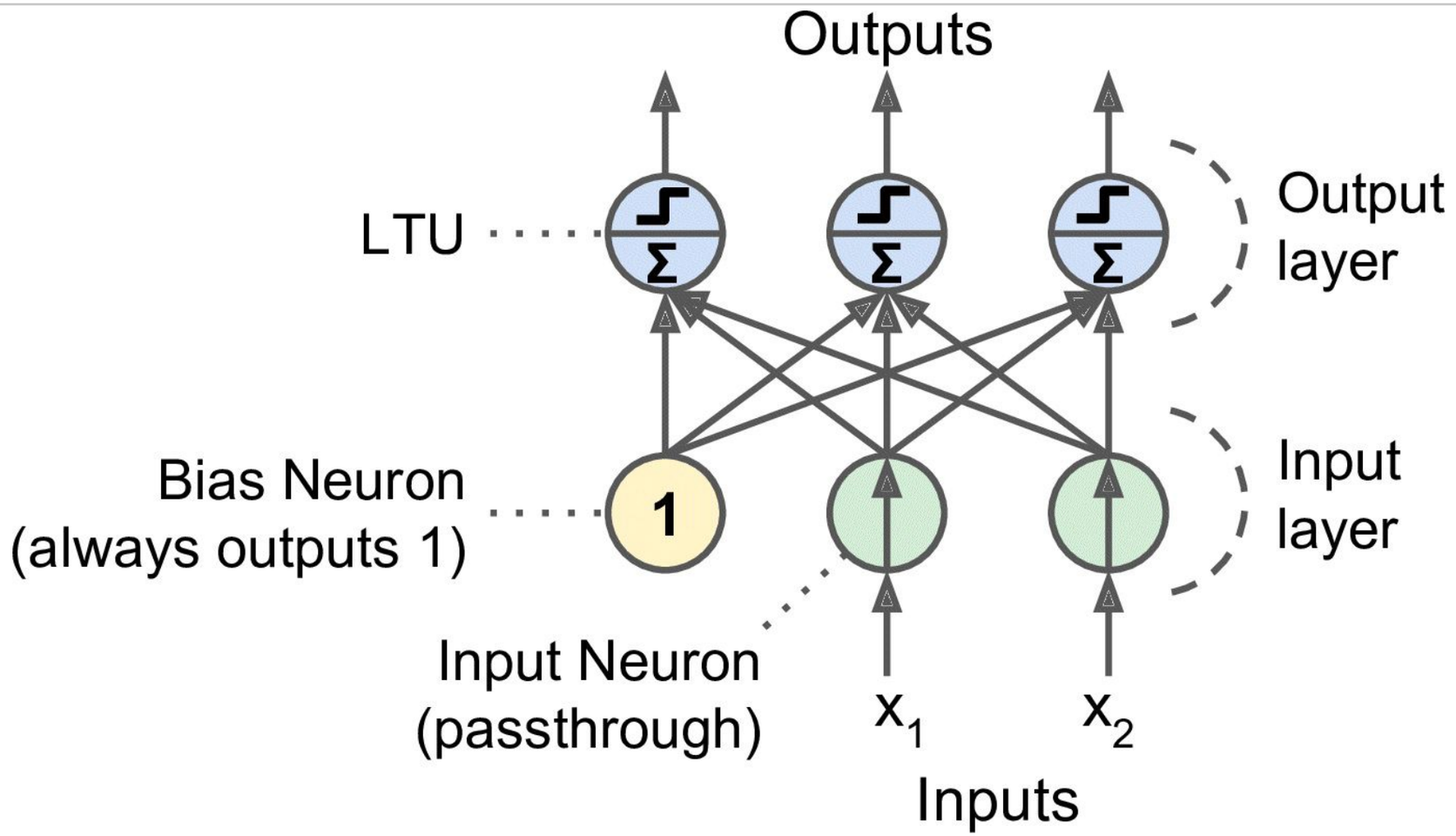


The Perceptron

- Input neurons just output whatever input they are fed
- An extra bias feature is generally added (x0 = 1)
- This bias feature is typically represented using a special type of neuron called a **bias neuron**, which just outputs **1** all the time



The Perceptron



- Above figure shows a Perceptron with two inputs and three output
- This Perceptron can classify instances simultaneously into three different binary classes, which makes it a **multioutput classifier**.

Training a Perceptron

Hebb's Rule

- In order to understand how we train a perceptron we need to understand Hebb's rule.
- In his book **The Organization of Behavior**, published in 1949, **Donald Hebb** suggested that

"When a biological neuron often triggers another neuron, the connection between these two neurons grows stronger. "

Training a Perceptron

The idea was later summarized by Siegrid Löwel in this catchy phrase:

"Cells that fire together, wire together."

Training a Perceptron

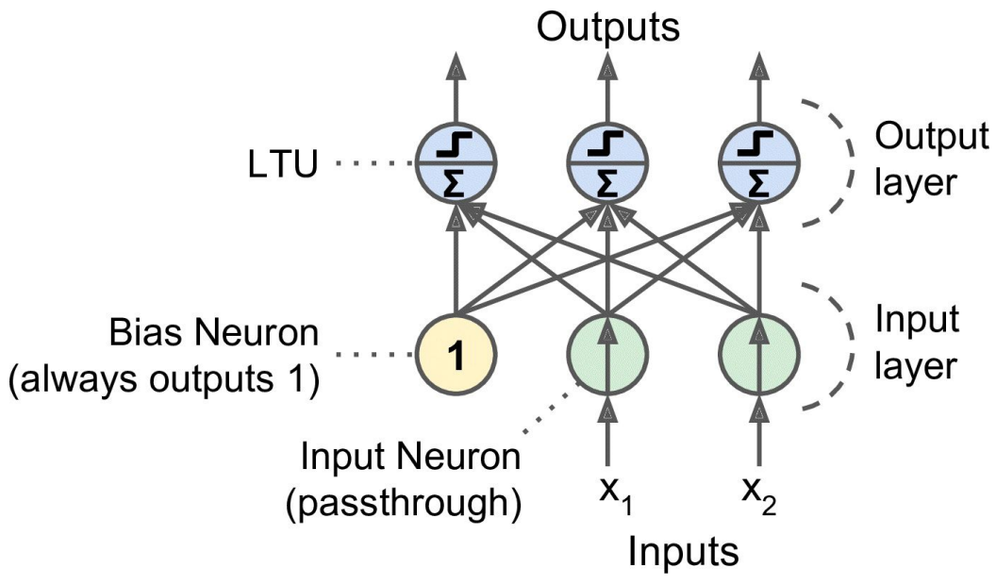
The main steps are:

- The Perceptron is fed one training instance at a time
- For each instance it makes its predictions
- For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction

Training a Perceptron

Perceptron learning rule (weight update)

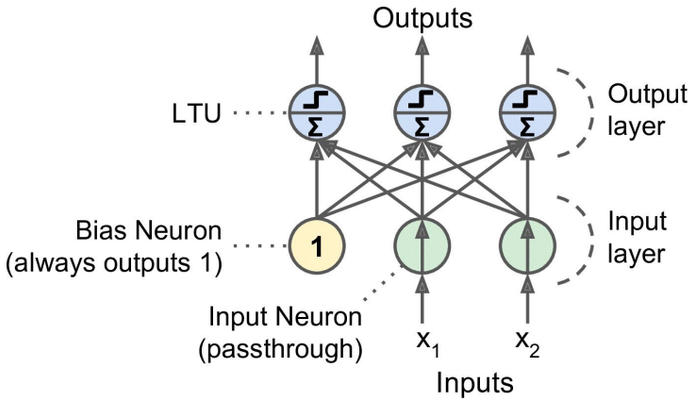
$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(\hat{y}_j - y_j)x_i$$



Training a Perceptron

Perceptron learning rule (weight update)

- $w_{i,j}$ is the connection weight between the ith input neuron and the jth output neuron.
- x_i is the ith input value of the current training instance.
- $\hat{y}_j$ is the output of the jth output neuron for the current training instance.

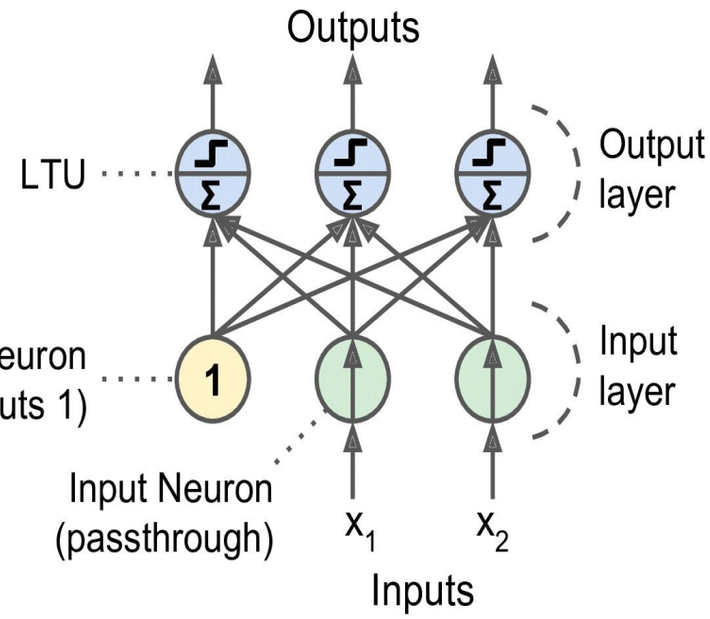


$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(\hat{y}_j - y_j)x_i$$

Training a Perceptron

Perceptron learning rule (weight update)

- y_j is the target output of the jth output neuron for the current training instance.
- η is the learning rate.



$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(\hat{y}_j - y_j)x_i$$

Training a Perceptron

Perceptron convergence theorem

- The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns.
- However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.
- This is called the **Perceptron convergence theorem**.

Training a Perceptron

Let's build our own perceptron using Scikit-Learn

We will test it on the iris dataset

```
>>> import numpy as np
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import Perceptron
>>> iris = load_iris()
```

Run it on Notebook

Training a Perceptron

```
>>> X = iris.data[:, (2, 3)] # petal length, petal width
>>> y = (iris.target == 0).astype(np.int) # Iris Setosa?
>>> per_clf = Perceptron(random_state=42)
>>> per_clf.fit(X, y)
>>> y_pred = per_clf.predict([[2, 0.5]])
```

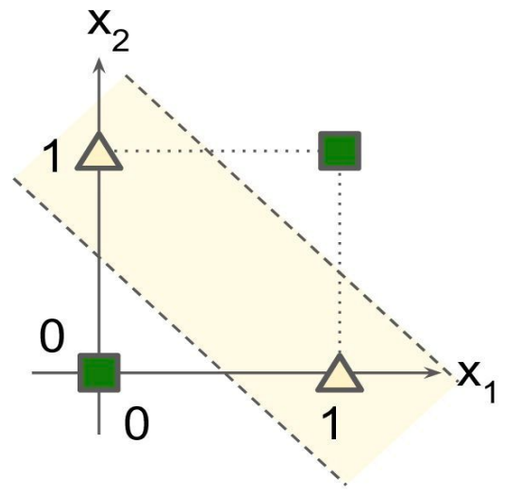
Run it on Notebook

Drawbacks of Perceptron model

Contrary to Logistic Regression classifiers, Perceptrons do not output a class probability, they just make predictions based on a hard threshold.

This is one of the good reasons to prefer Logistic Regression over Perceptrons.

They are incapable of solving some trivial problems like the Exclusive OR (XOR) classification problem.

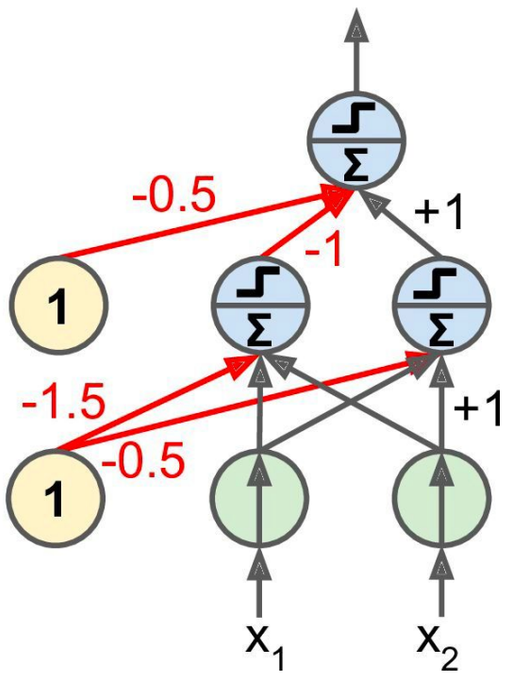


Inputs		Output
A	*B*	*X*
0	0	0
0	1	1
1	0	1
1	1	0

Multi-Layer Perceptron (MLP)

Solving the XOR problem

- It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons
- The resulting ANN is called a **Multi-Layer Perceptron (MLP)**
- In particular, an MLP can solve the XOR problem

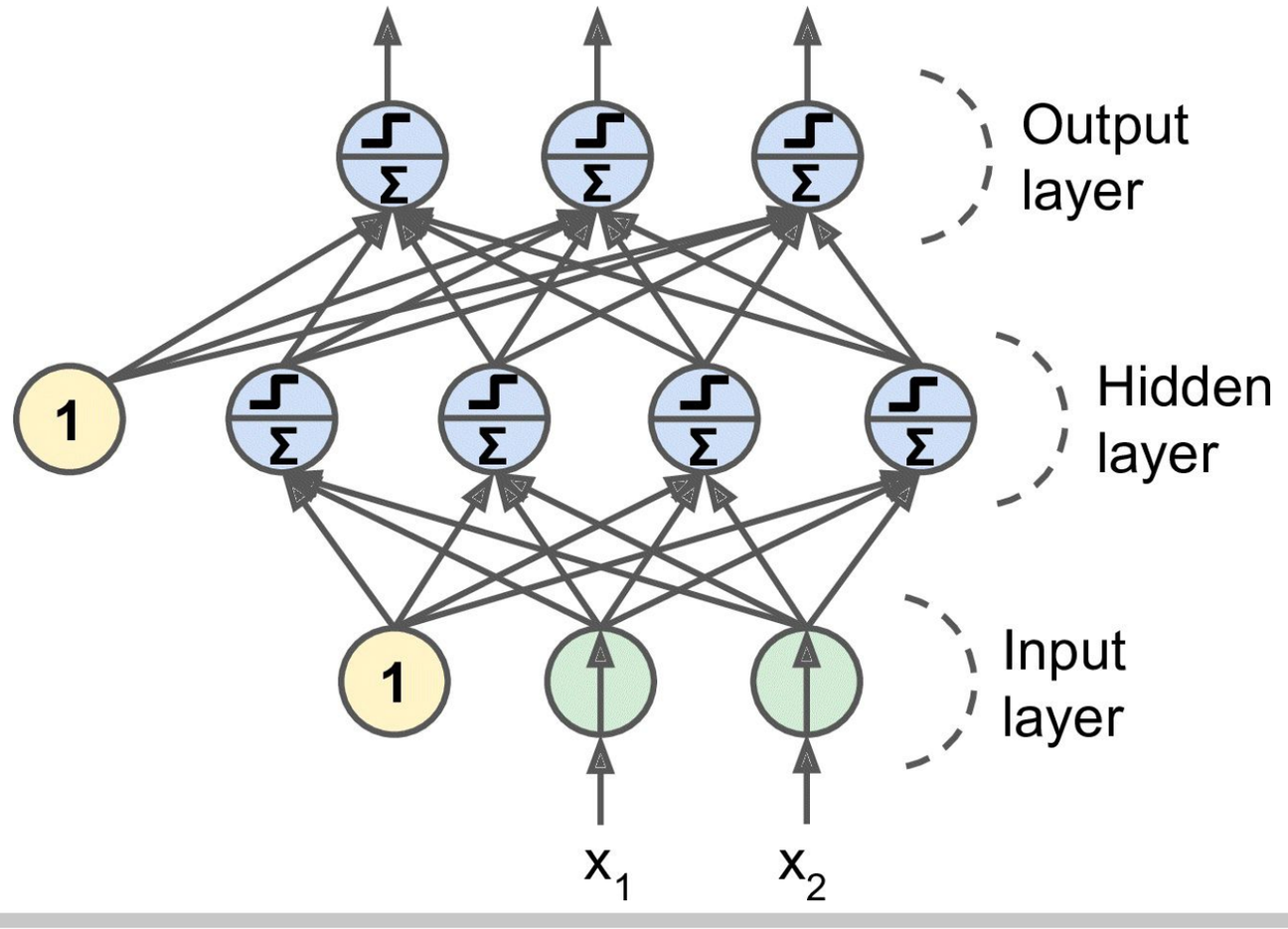


Inputs		Output
A	*B*	*X*
0	0	0
0	1	1
1	0	1
1	1	0

Multi-Layer Perceptron (MLP)

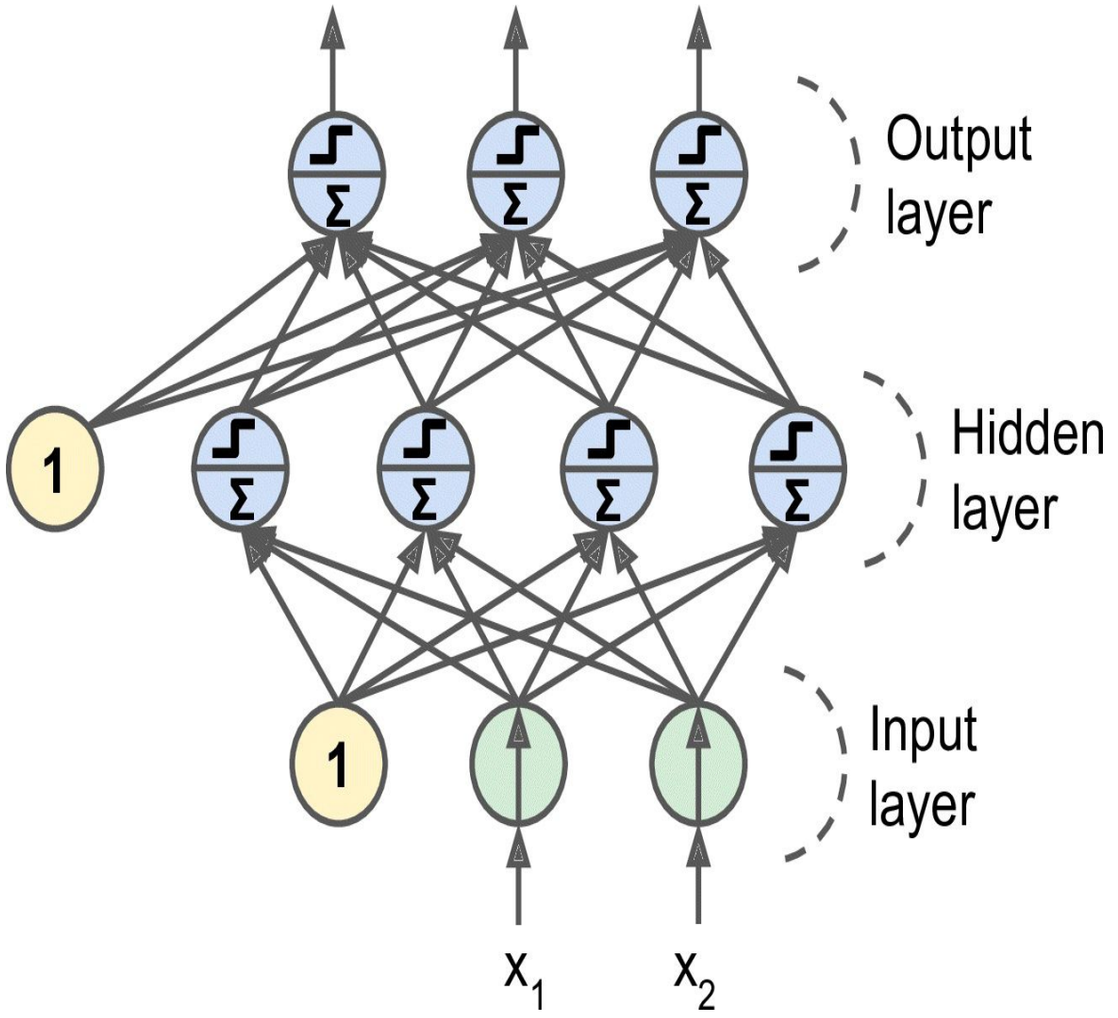
An MLP is composed of

- One **input layer**
- One or more layers of LTUs, called **hidden layers**
- And one final layer of LTUs called the **output layer**



Multi-Layer Perceptron (MLP)

Every layer except the output layer includes a bias neuron and is fully connected to the next layer.

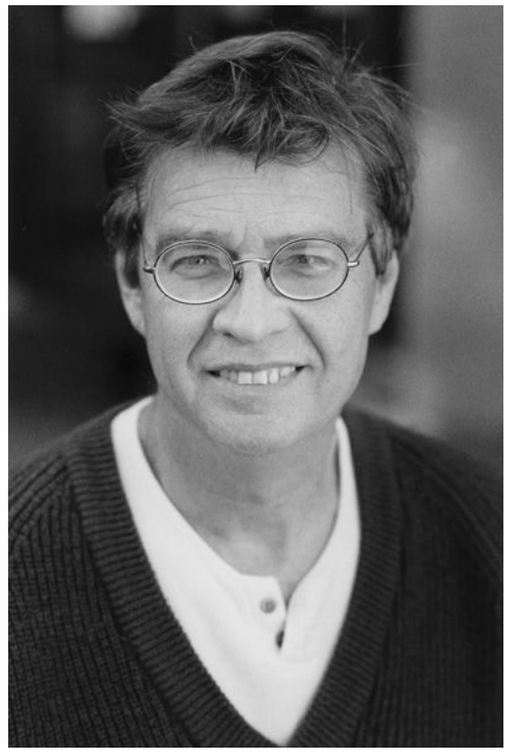


Deep Neural Network

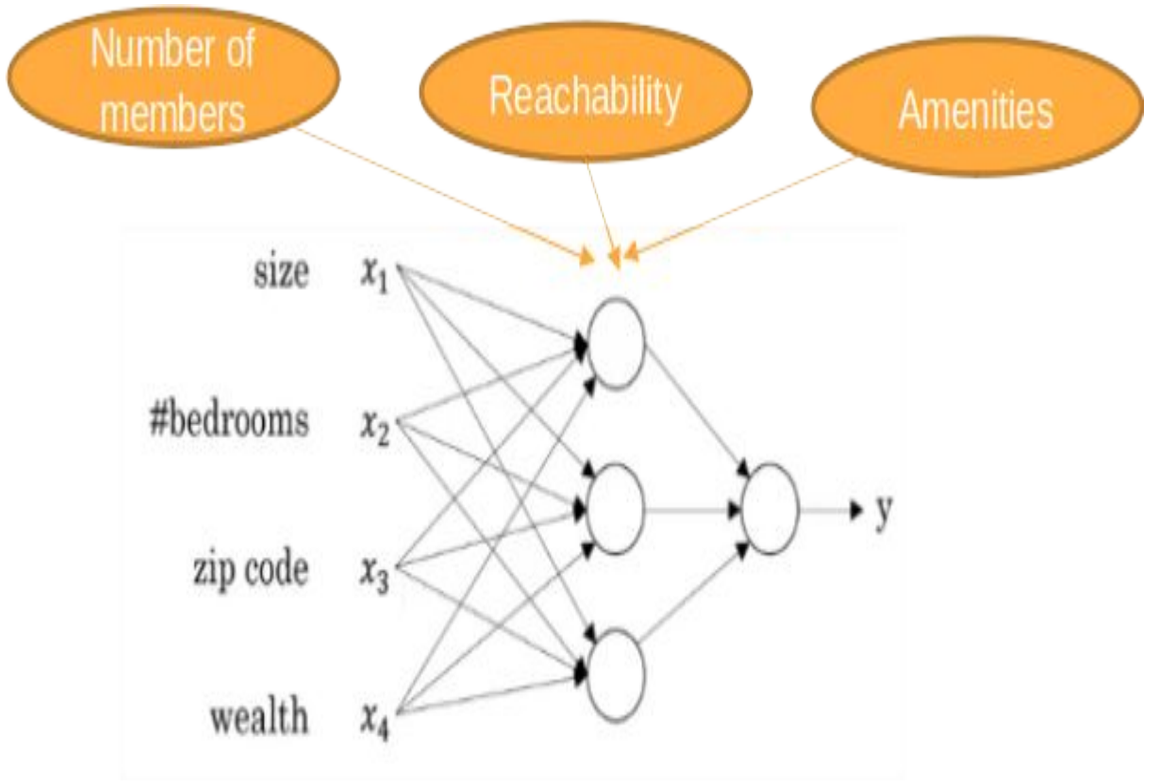
When an ANN has two or more hidden layers, it is called a deep neural network (DNN)

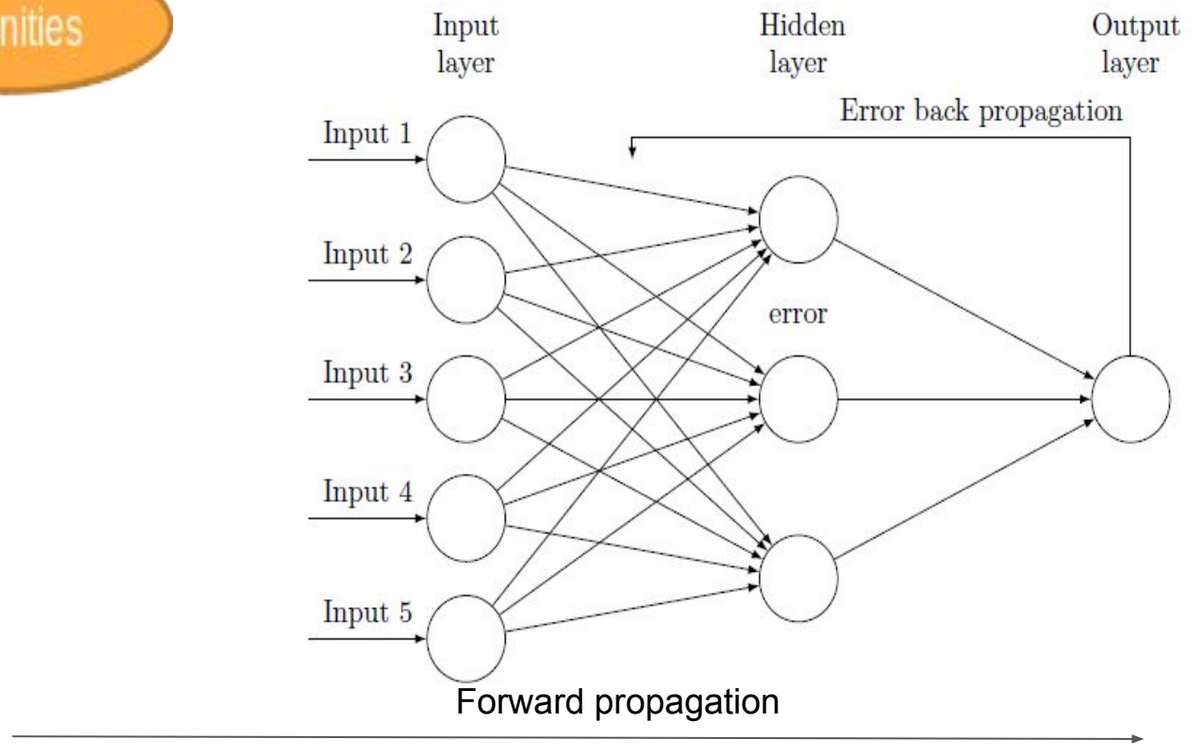
Multi-Layer Perceptron and Backpropagation

- For many years researchers struggled to find a way to train MLPs, without success
- In 1986, D. E. Rumelhart et al. published a groundbreaking article introducing the **backpropagation training algorithm**
- Today we would describe it as Gradient Descent using reverse-mode autodiff



Multi-Layer Perceptron: what is happening inside conceptually

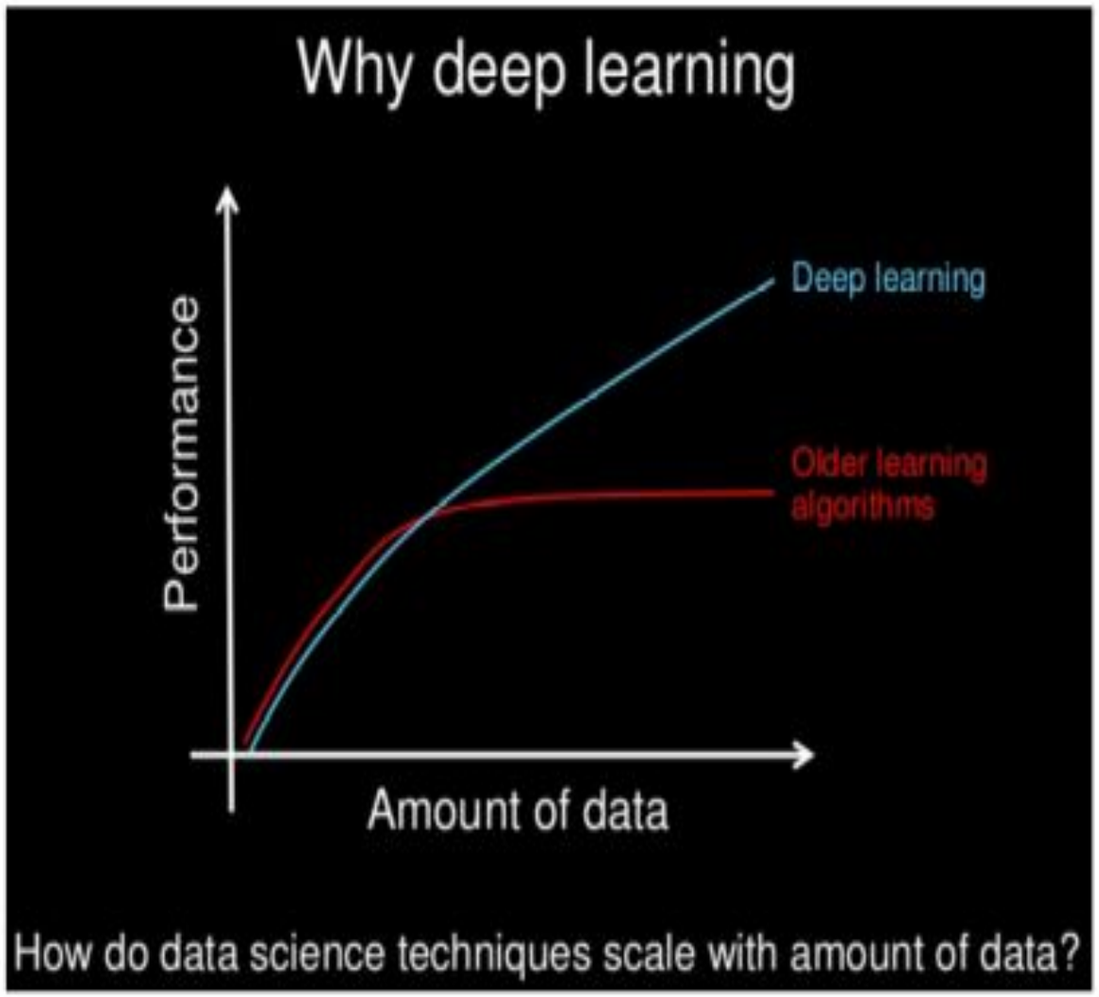


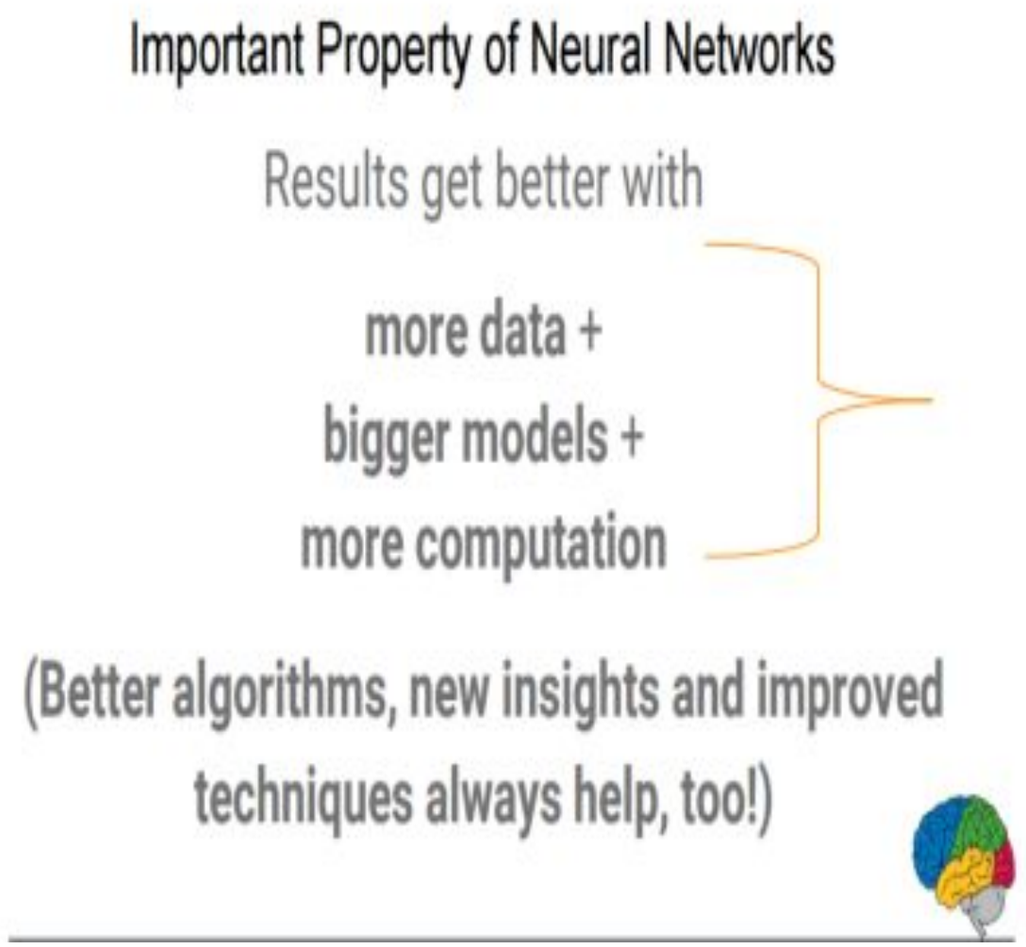


Output of every neuron in each consecutive layer is found

In forward propagation, we compute the value of the function for given w (randomly initialized and to be optimized by gradient descent), X and the bias term (randomly initialized and to be optimized by the gradient descent).

Multi-Layer Perceptron: what is happening inside conceputally



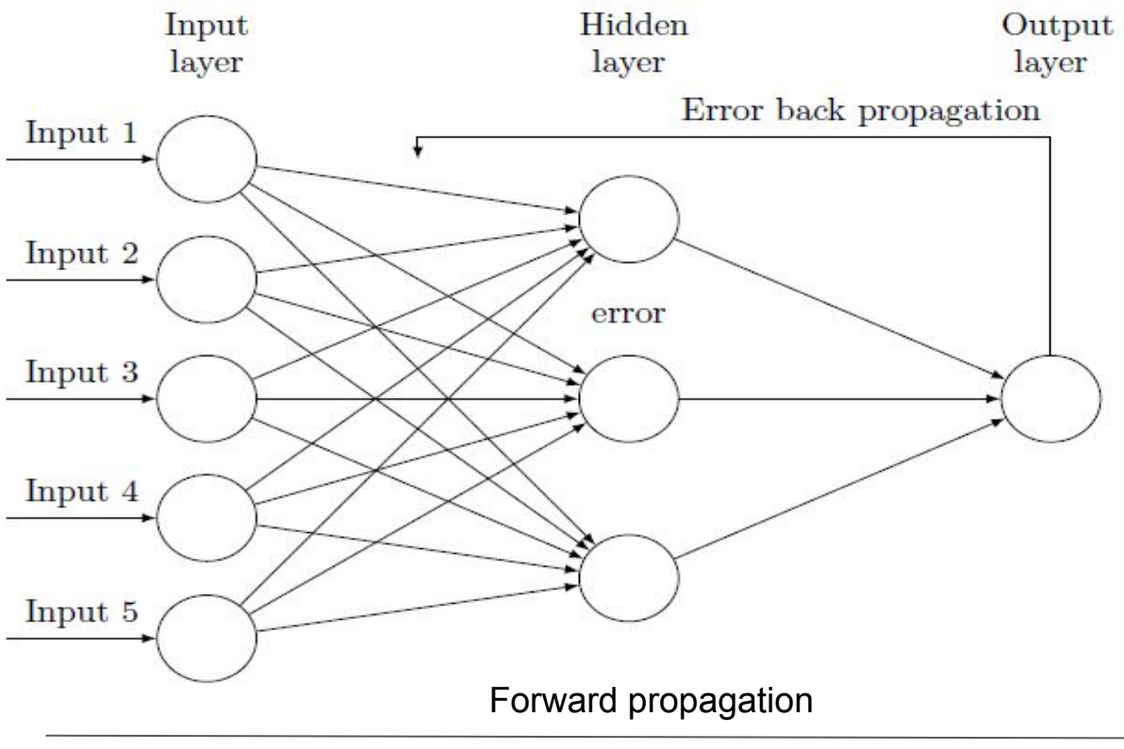


Multi-Layer Perceptron and Backpropagation

What is Backpropagation?

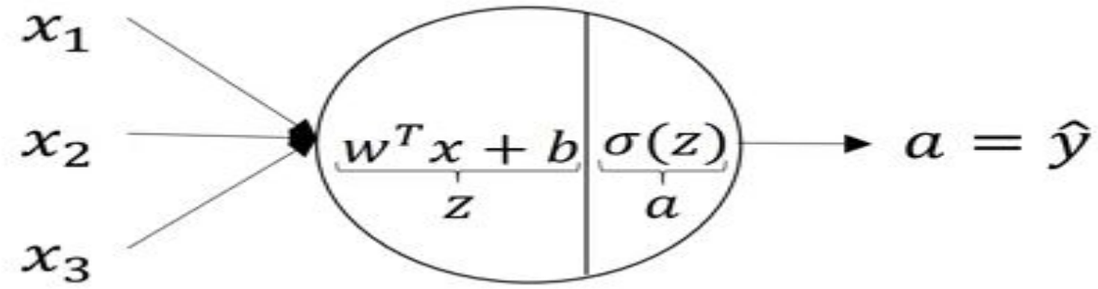
For each training instance

- The algorithm feeds it to the network
- And computes the output of every neuron in each consecutive layer
- This is the forward pass, just like when making predictions



Output of every neuron in each consecutive layer is found

Multi-Layer Perceptron: Forward Pass



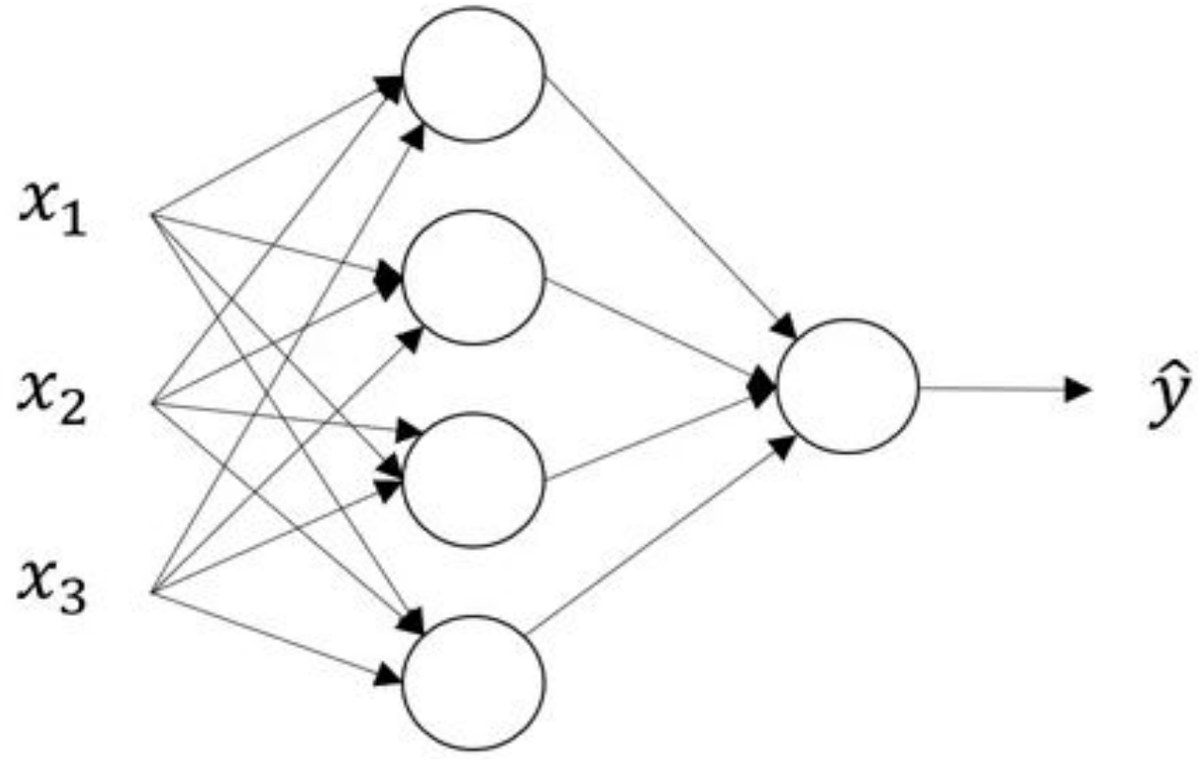
Processing of 4 neurons at hidden layer

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$



Forward propagation

Multi-Layer Perceptron: Forward Pass

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

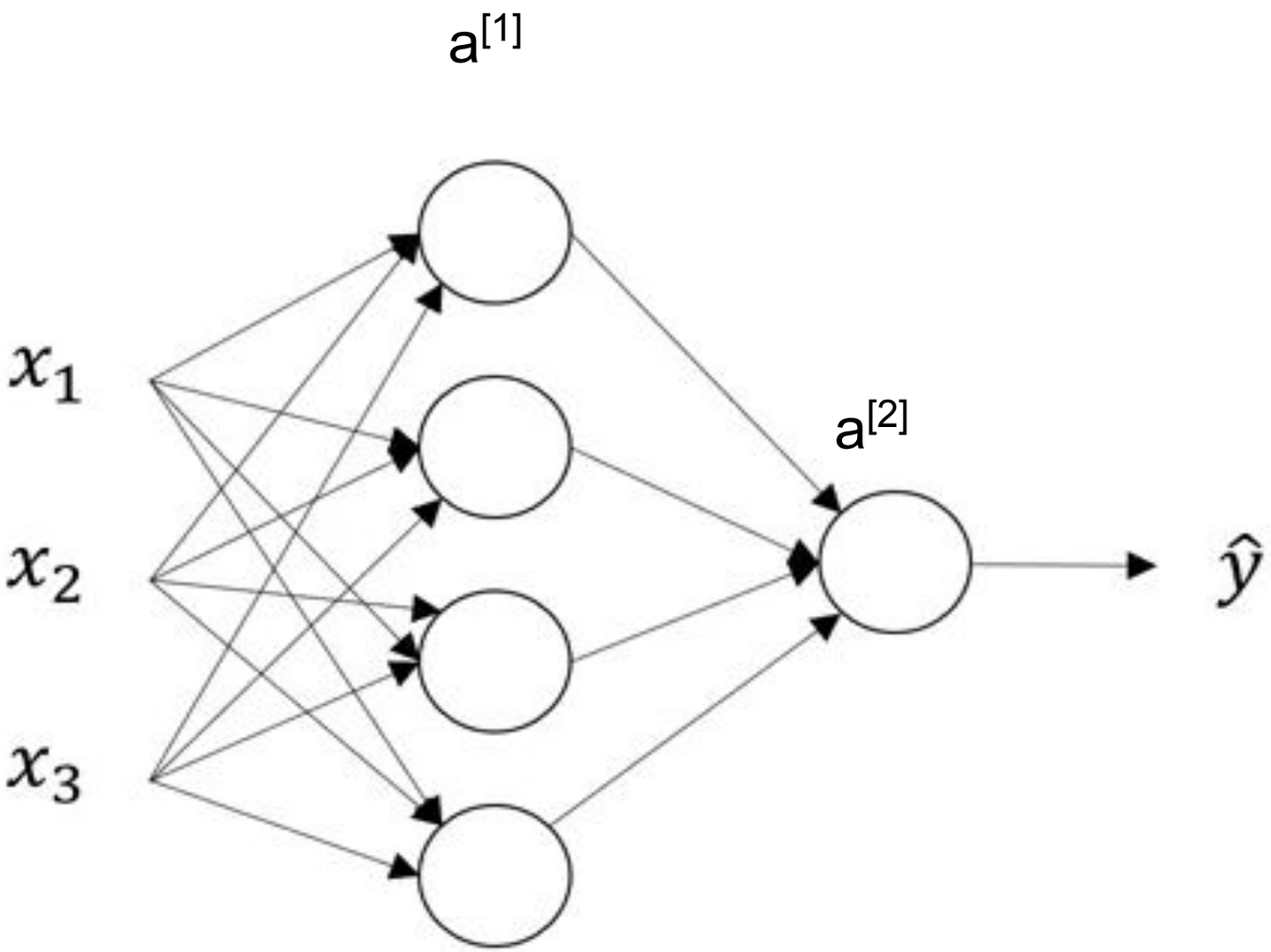
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

Processing of 4 neurons at hidden layer

- Computation at neurons of layer-1 (a[1]), superscript represents the layer and subscript represent the neuron position.
- $a_{[1]}^{T} = [a_1^{[1]}, a_2^{[1]}, a_3^{[1]}, a_4^{[1]}]$ // $a^{[1]}$ is a (4,1) vector
- $w_1^{[1]}$ is weight vector for neuron-1 at layer-1 with three weight elements for x_1, x_2, and x_3.
- $w[^{[1]T} = [w_1^{[1]}, w_2^{[1]}, w_3^{[1]}, w_4^{[1]}]$, //(4 (number of neurons in current layer), 3 (number of input units in previous layer)) matrix
- $b_1^{[1]}$ is the bias term for neuron-1 at layer-1.
- x is the feature vector for one training instance.

Forward propagation

Multi-Layer Perceptron: Forward Pass



$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = W^{[2]}x + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$

Forward propagation

Multi-Layer Perceptron: Forward Pass

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

$$W_1^{[1]} = \begin{pmatrix} W_{11}^{[1]} \\ W_{12}^{[1]} \\ W_{13}^{[1]} \end{pmatrix}$$

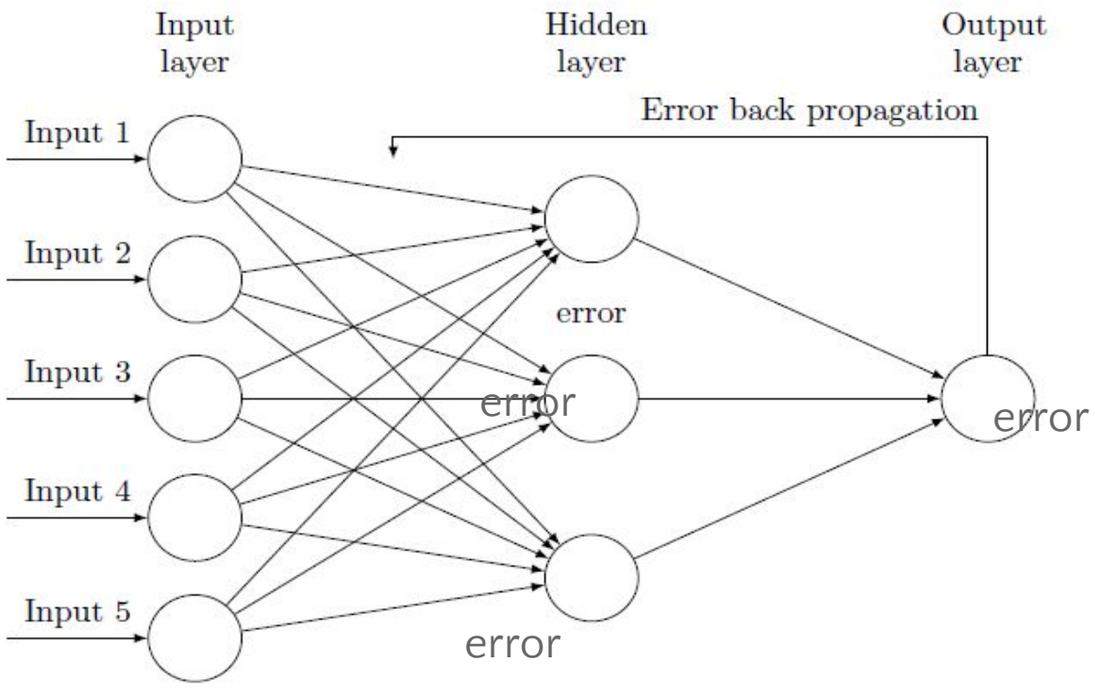
Each layer has a weight matrix.

Forward propagation

Multi-Layer Perceptron and Backpropagation

What is Backpropagation?

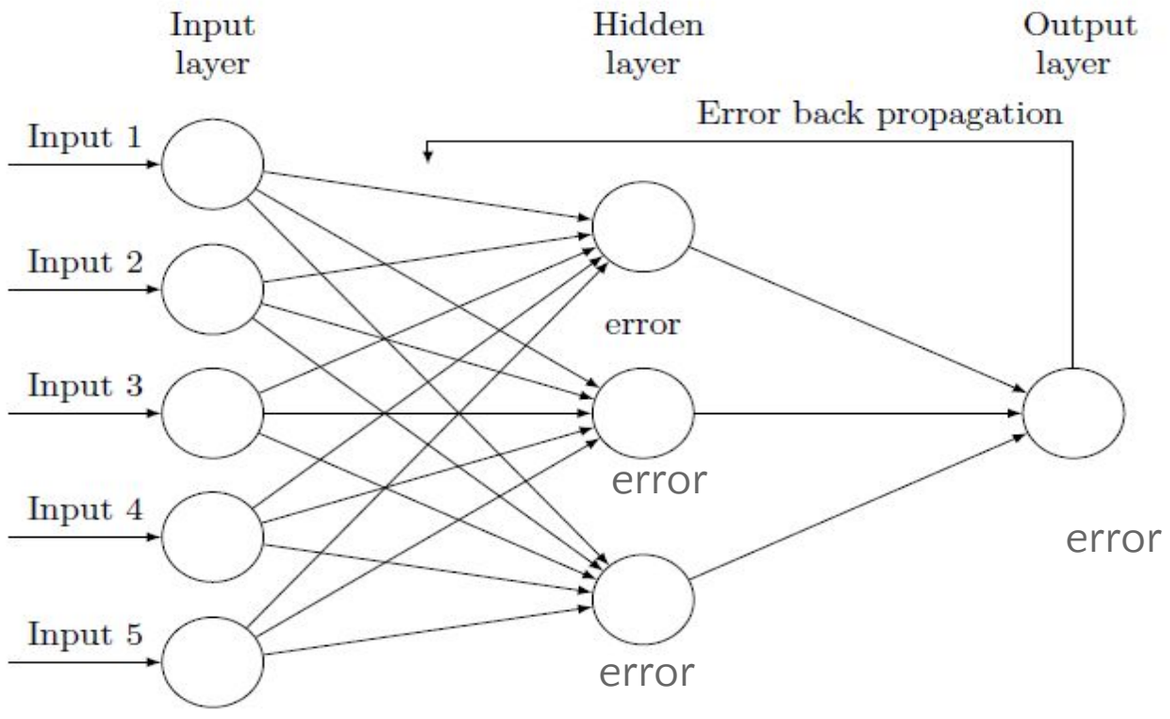
- Then it measures the network's output error i.e., the difference between the desired output and the actual output of the network
- It then computes how much each neuron in the last hidden layer contributed to each output neuron's error.



Multi-Layer Perceptron and Backpropagation

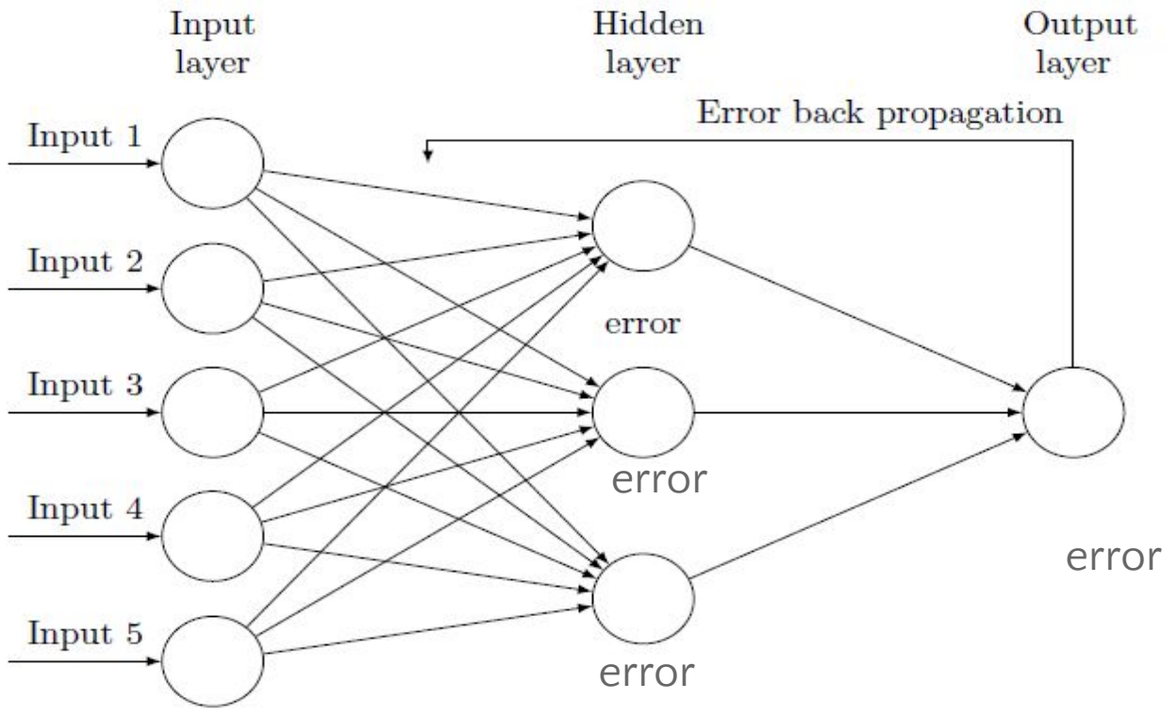
What is Backpropagation?

- It then proceeds to measure how much of these error contributions came from each neuron in the previous hidden layer — and so on until the algorithm reaches the input layer.



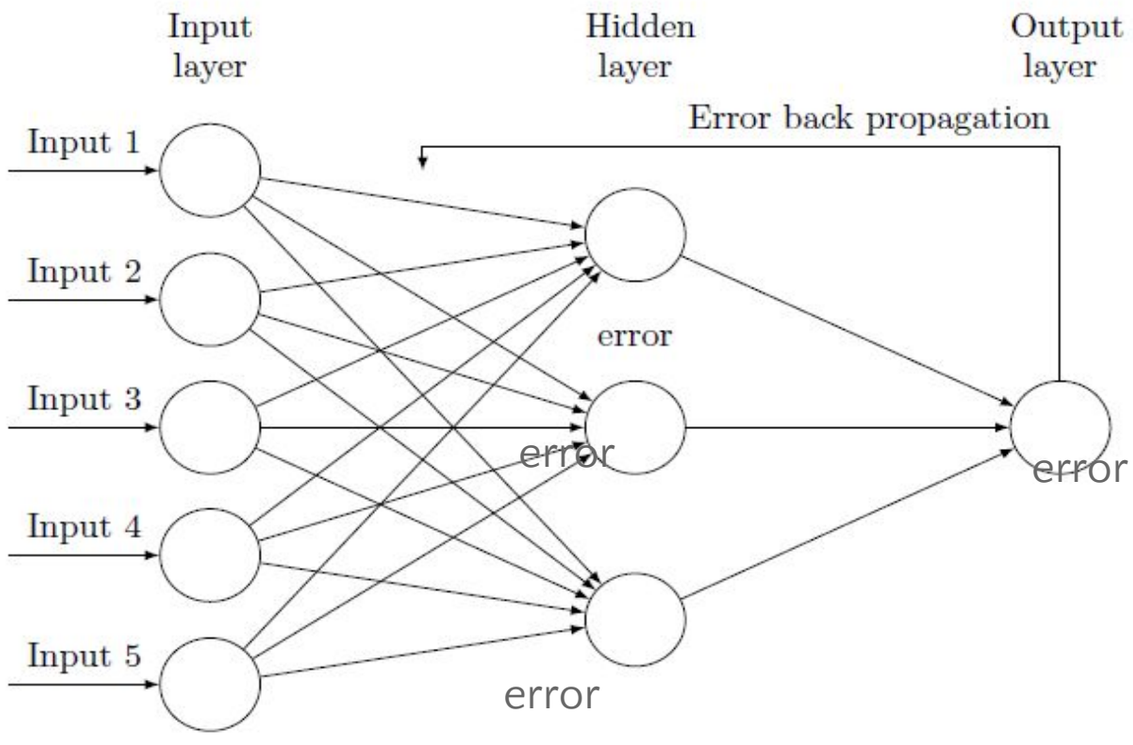
What is Backpropagation?

- The last step of the backpropagation algorithm is a **Gradient Descent step** on all the connection weights in the network, using the error gradients measured earlier.



What is Backpropagation?

- This reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward in the network. Hence the name of the algorithm is **Backpropagation**



Backpropagation – Partial Derivatives

$$y = x^2 + t^3 + z$$

$$dy / dx = ?$$

Backpropagation – Partial Derivatives

$$y = x^2 + t^3 + z$$

$$dy / dx = 2*x$$

Backpropagation – Partial Derivatives

$$y = x^2 + t^3 + z$$

$$dy / dt = ?$$

Backpropagation – Partial Derivatives

$$y = x^2 + t^3 + z$$

$$dy / dt = 3*t^2$$

Backpropagation – Partial Derivatives

$$y = x^2 + t^3 + z$$

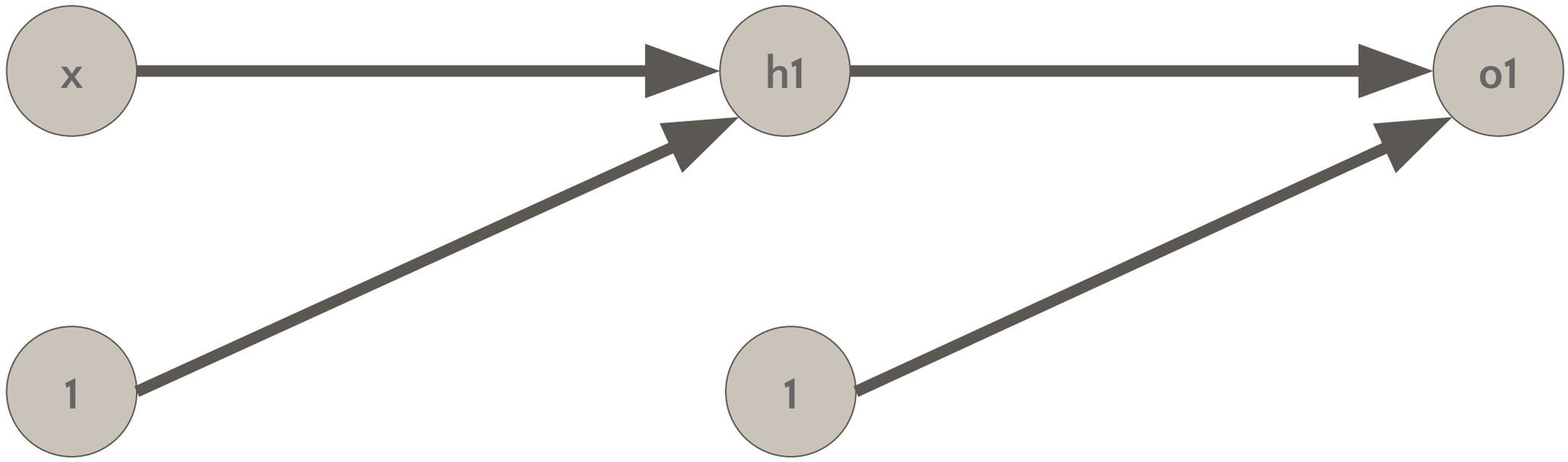
$$dy / dz = ?$$

Backpropagation – Partial Derivatives

$$y = x^2 + t^3 + z$$

$$dy / dz = 1$$

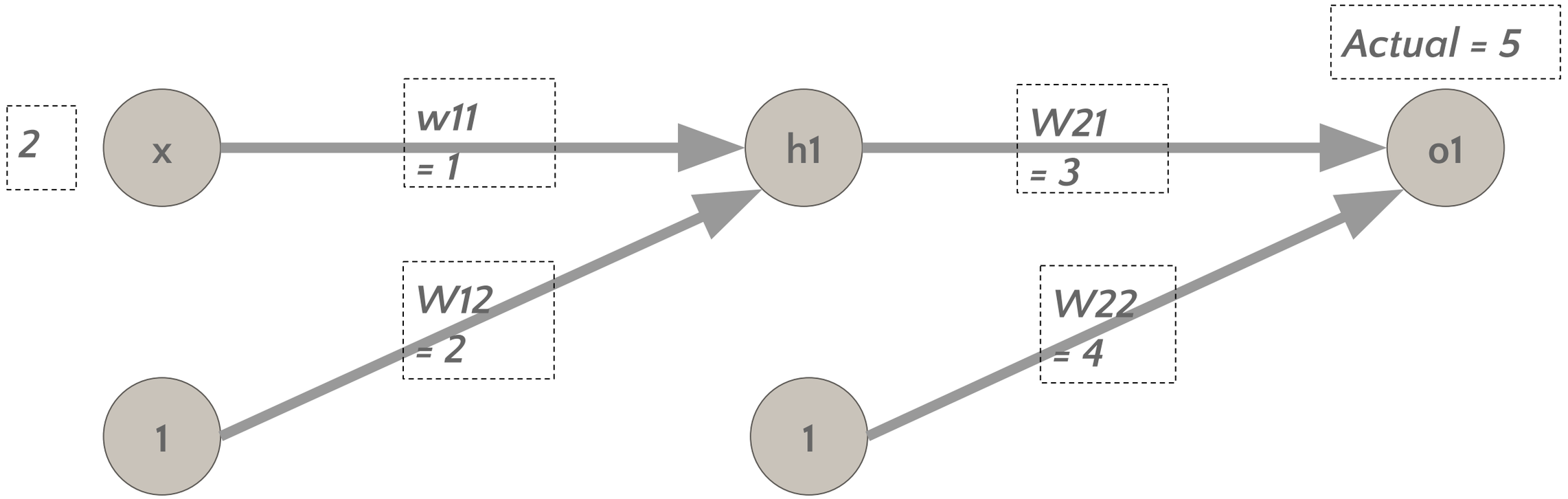
Multi-Layer Perceptron & Backpropagation



Example

- Regression
- One input feature and one output label, Error – Mean Square
- One hidden layer having one neuron, No activation function

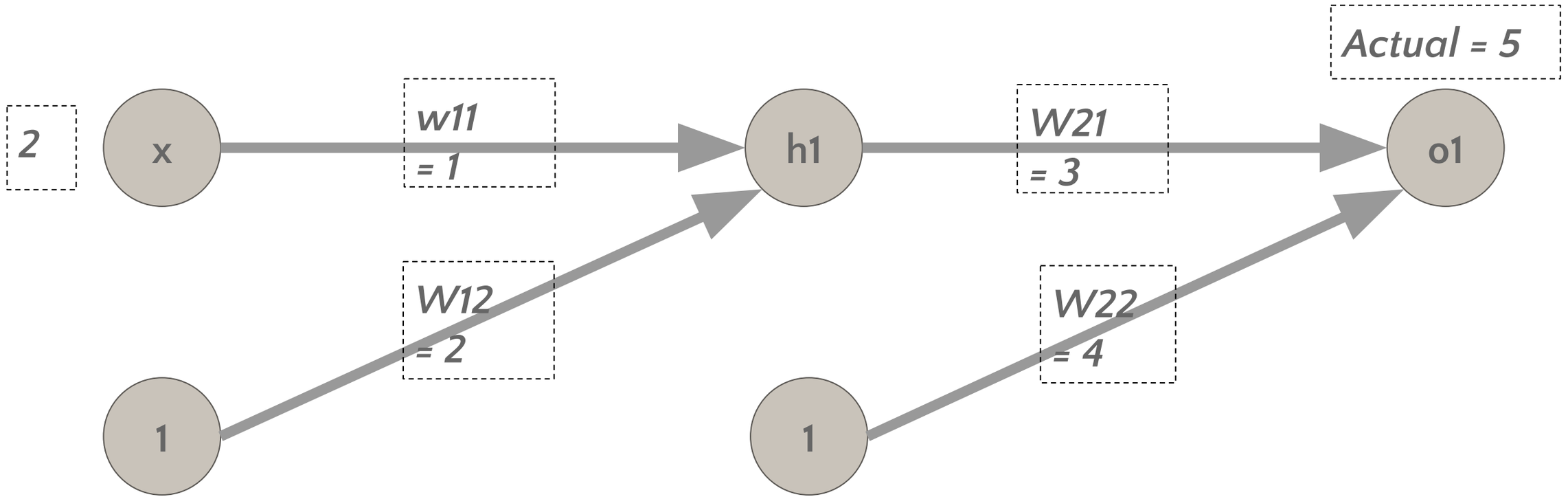
Multi-Layer Perceptron & Backpropagation



Example

- Say, x = 2 and y = 5
- And initial weights are say are 1, 2, 3, 4

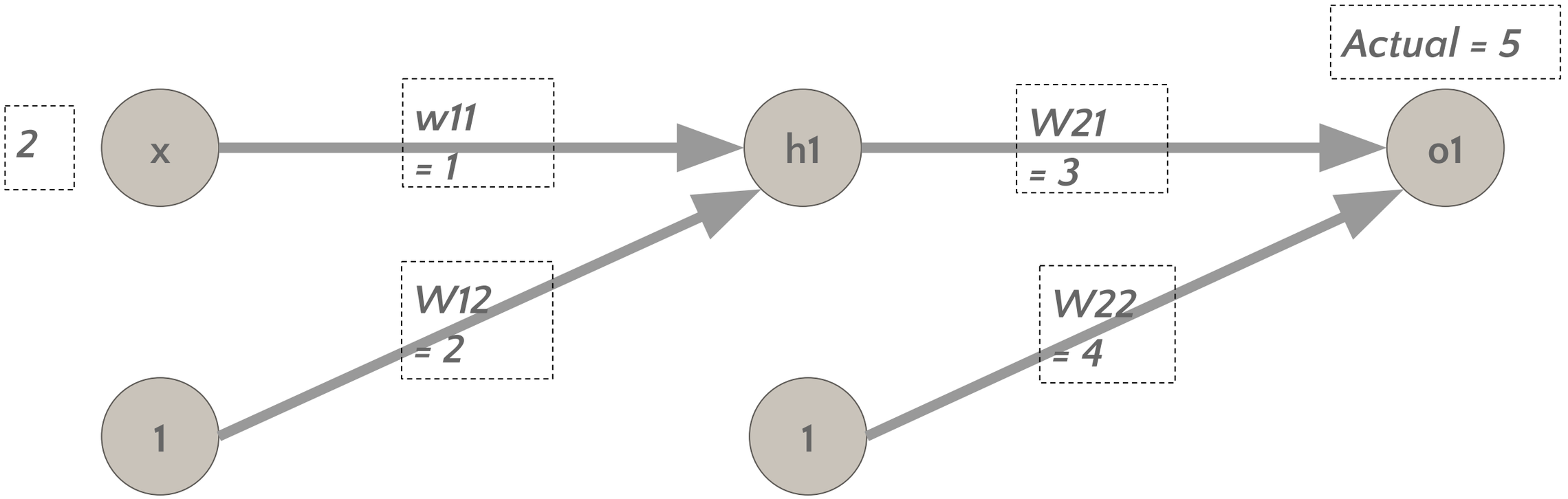
Multi-Layer Perceptron & Backpropagation



Example

- h1 =
- o1 =

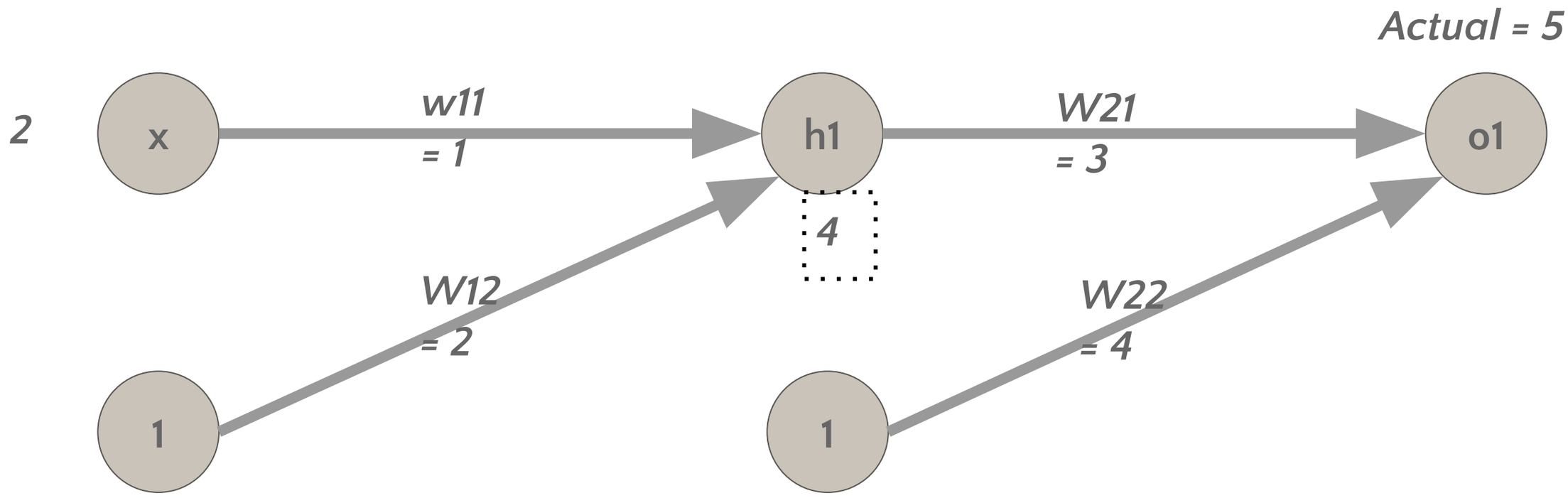
Multi-Layer Perceptron & Backpropagation



Example

- h1 = w11 * x + w12
- o1 = w21 * h1 + w22

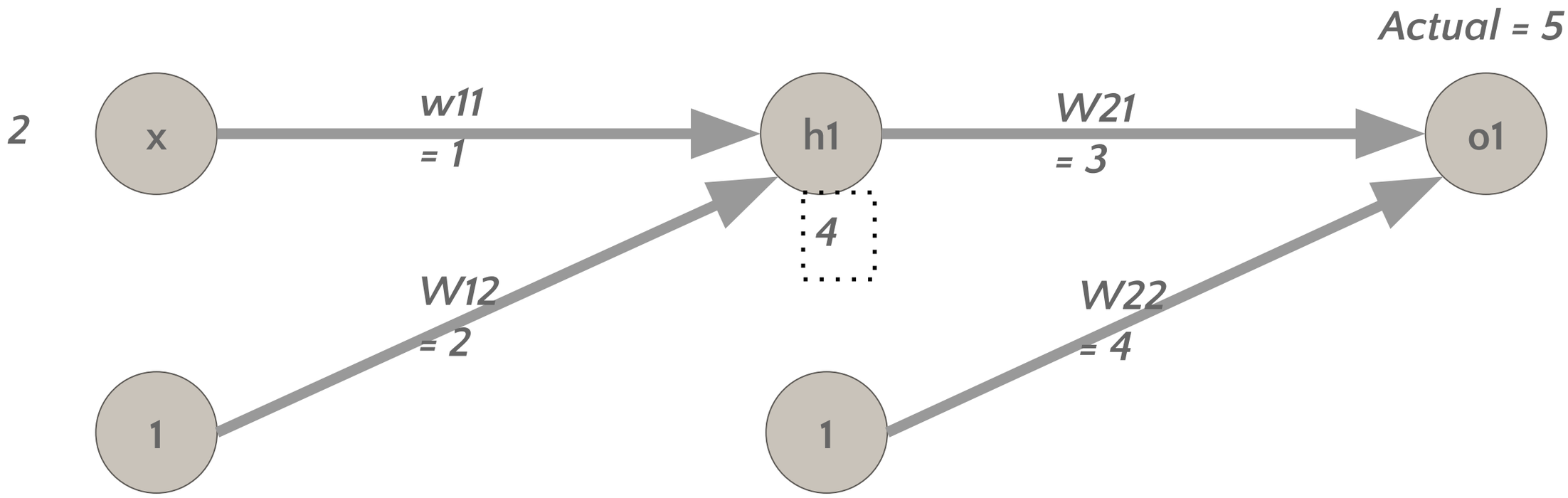
Multi-Layer Perceptron & Backpropagation



Example – compute first layer

- h1
  - = ?

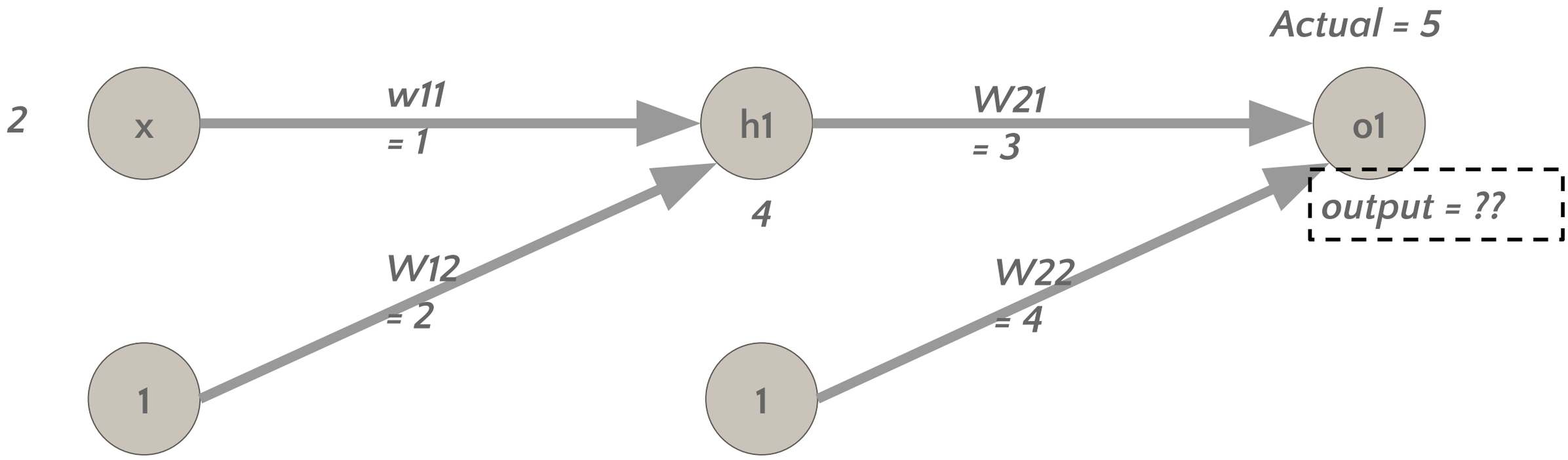
Multi-Layer Perceptron & Backpropagation



Example – compute first layer

- h1
  - = w11 * x + w12
  - = 2*1 + 1*2
  - = **4**

Multi-Layer Perceptron & Backpropagation

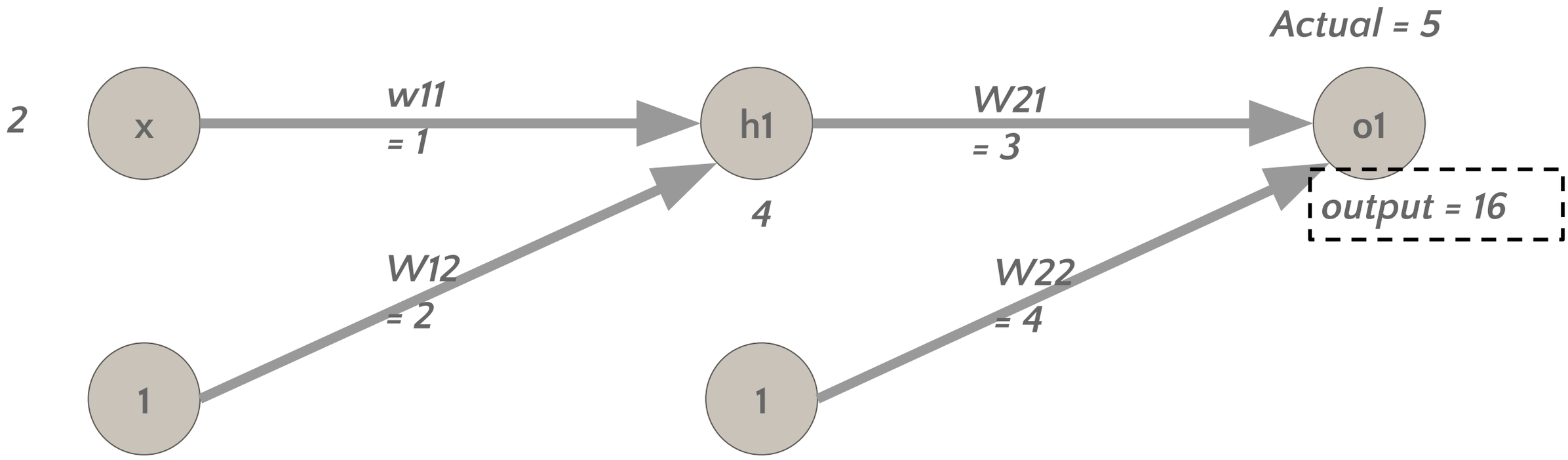


Example – compute output

o1

= ?

Multi-Layer Perceptron & Backpropagation



Example – compute output

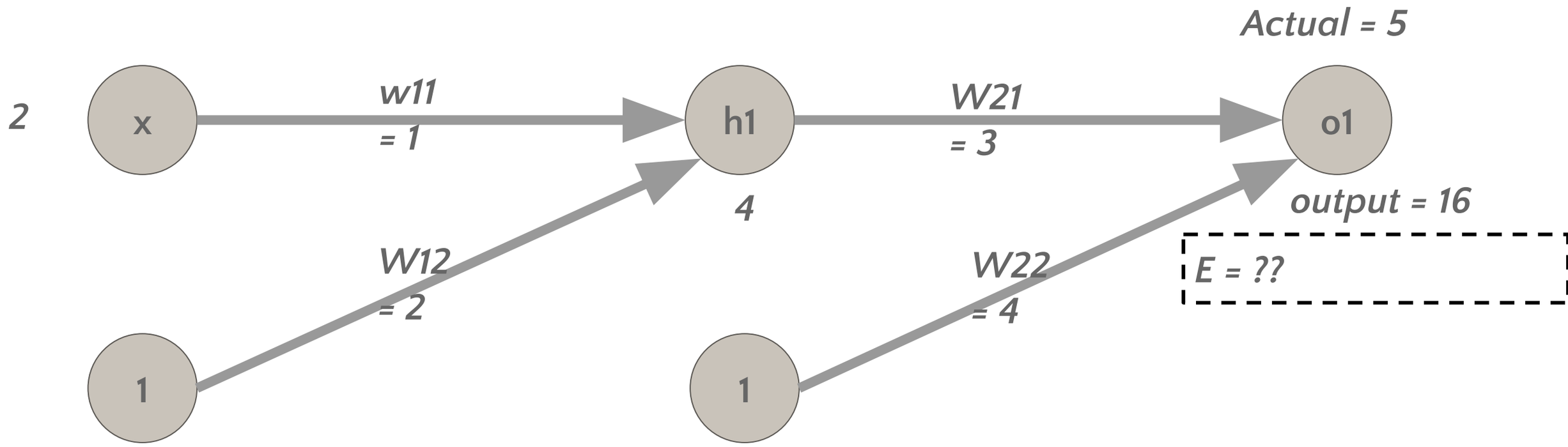
o1

= h1 * w21 + w22

= 3*4 + 4

= 16

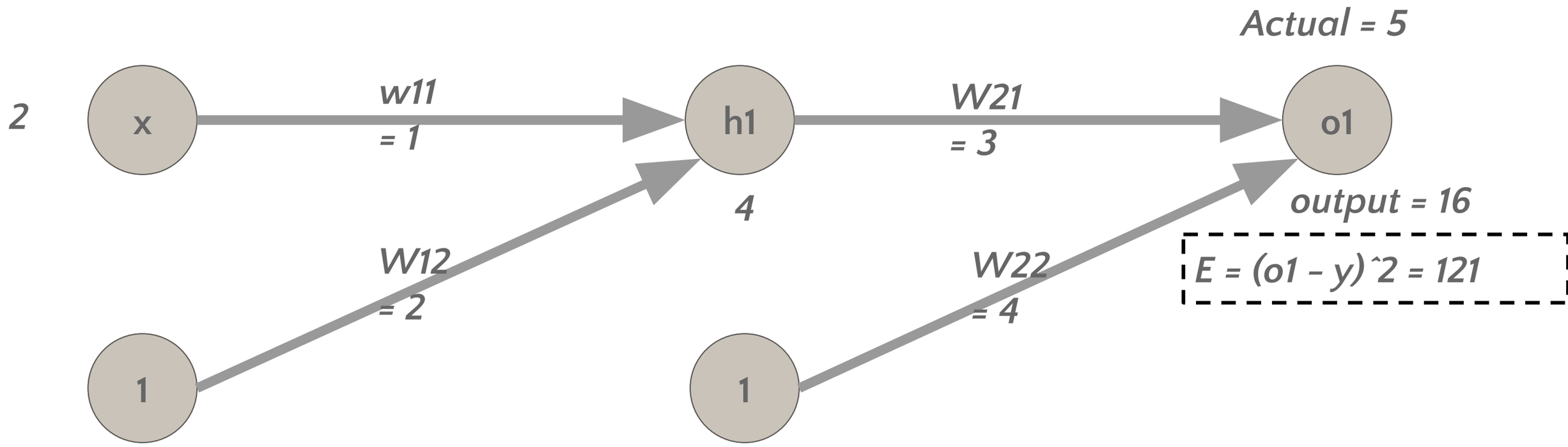
Multi-Layer Perceptron & Backpropagation



Example – Compute Error

E = ?

Multi-Layer Perceptron & Backpropagation



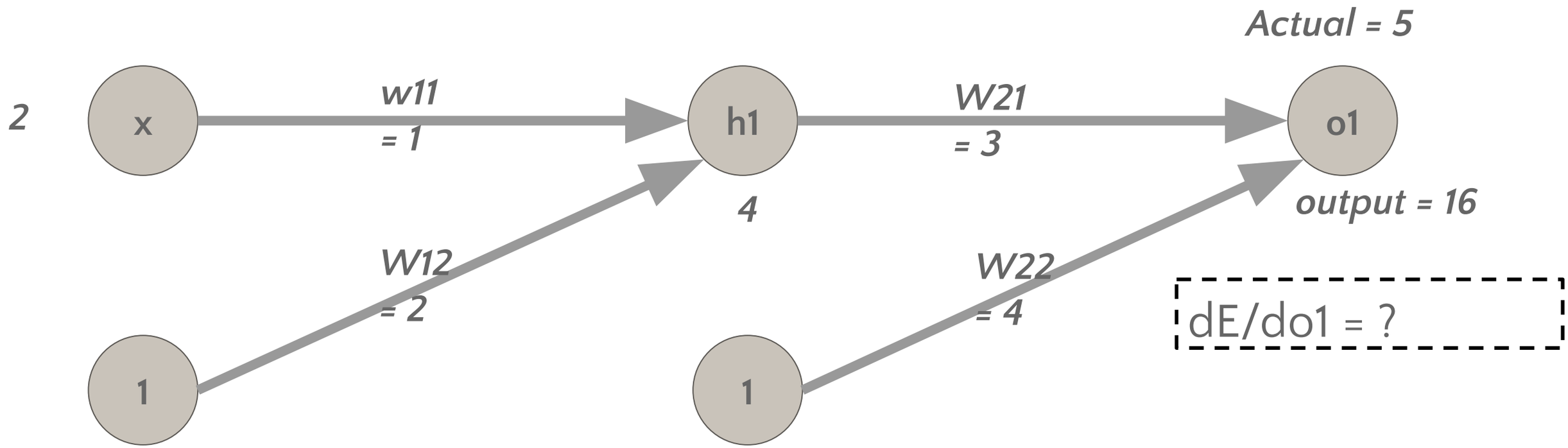
Example – Compute Error

E = (o1 – y)^2

= (16 – 5) ^ 2

= 121

Multi-Layer Perceptron & Backpropagation

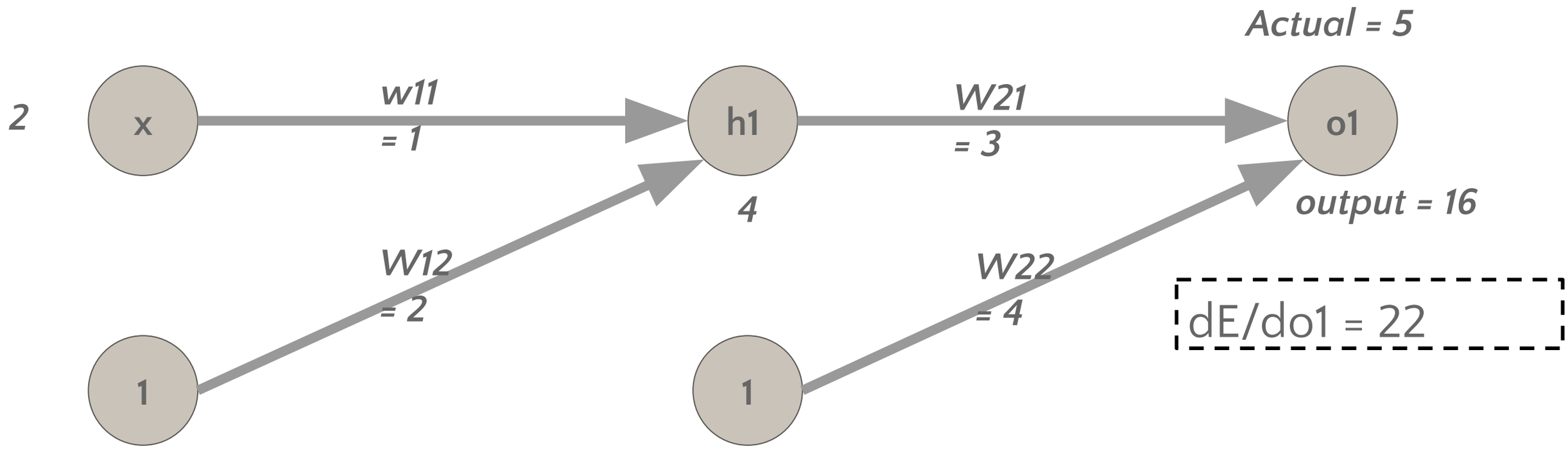


Example – Rate of change of error wrt output

dE/do1

= ?

Multi-Layer Perceptron & Backpropagation



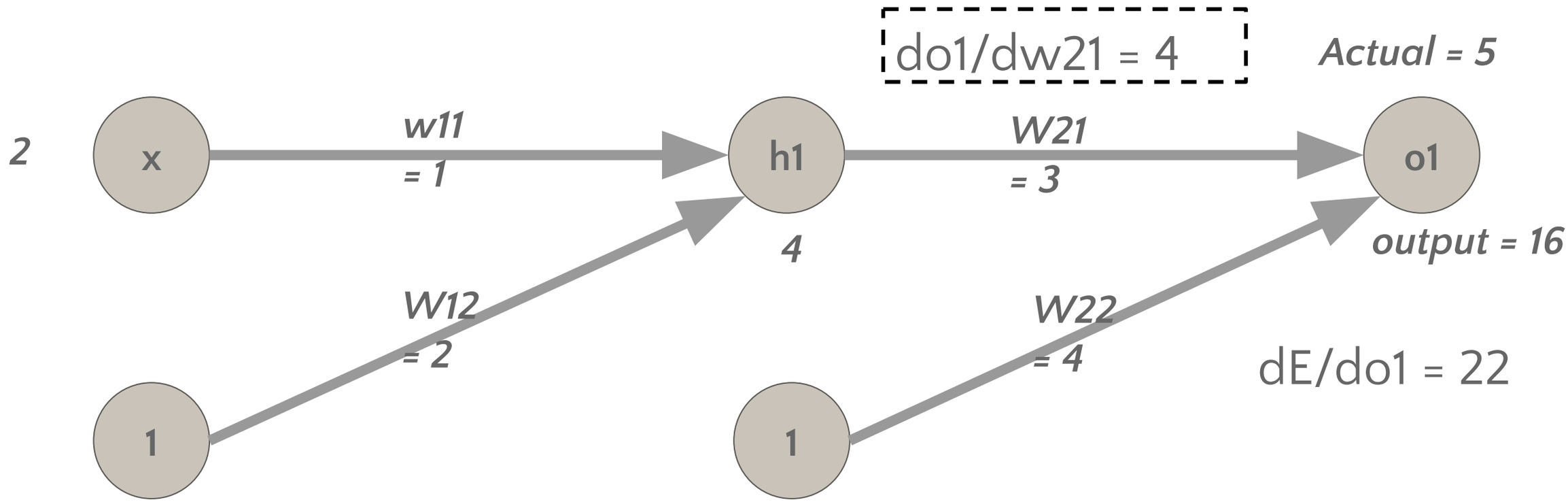
Example – Rate of change of error wrt output

dE/do1

= 2 (o1 – y)

= 2(16 – 5) = 22

Multi-Layer Perceptron & Backpropagation

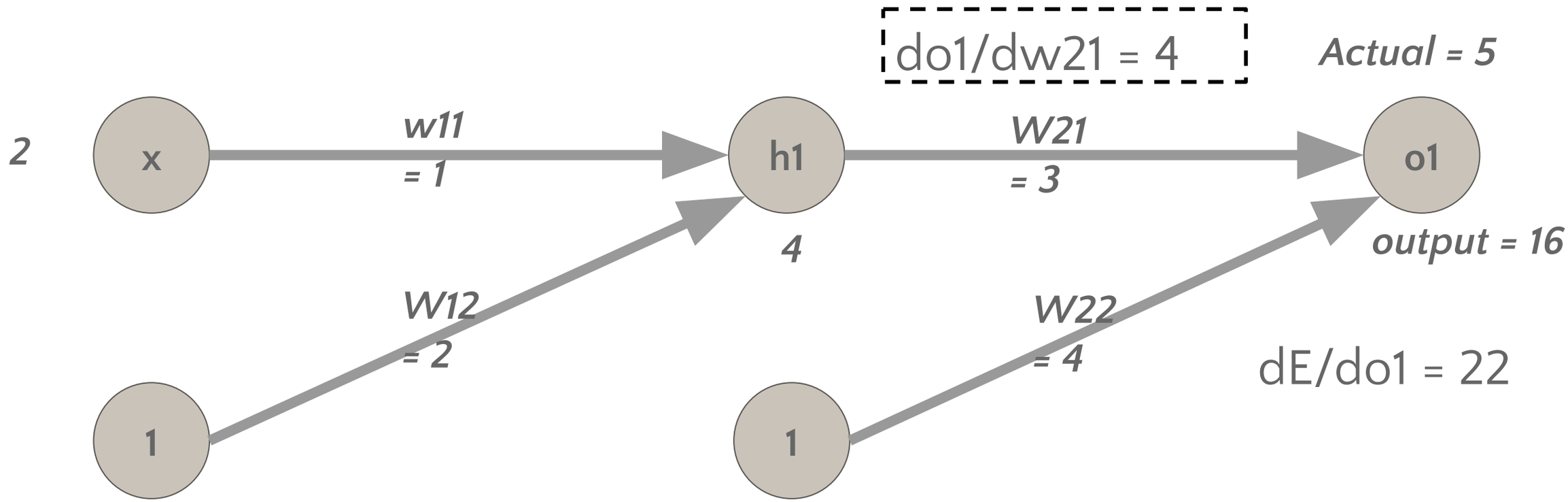


Example – Rate of change of output wrt w21

do1/dw21

= ?

Multi-Layer Perceptron & Backpropagation



Example – Rate of change of output wrt w21

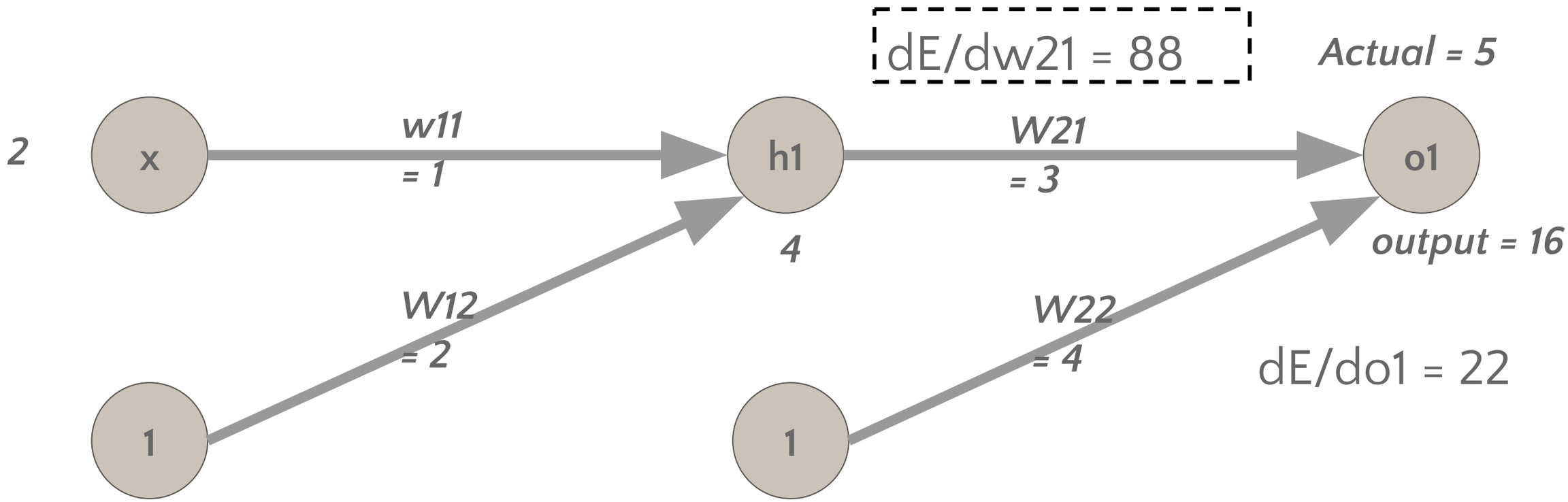
do1/dw21

= d (w21 * h1 + w22*1) / dw21

= h1

= 4

Multi-Layer Perceptron & Backpropagation



Example – Rate of change of error wrt w21

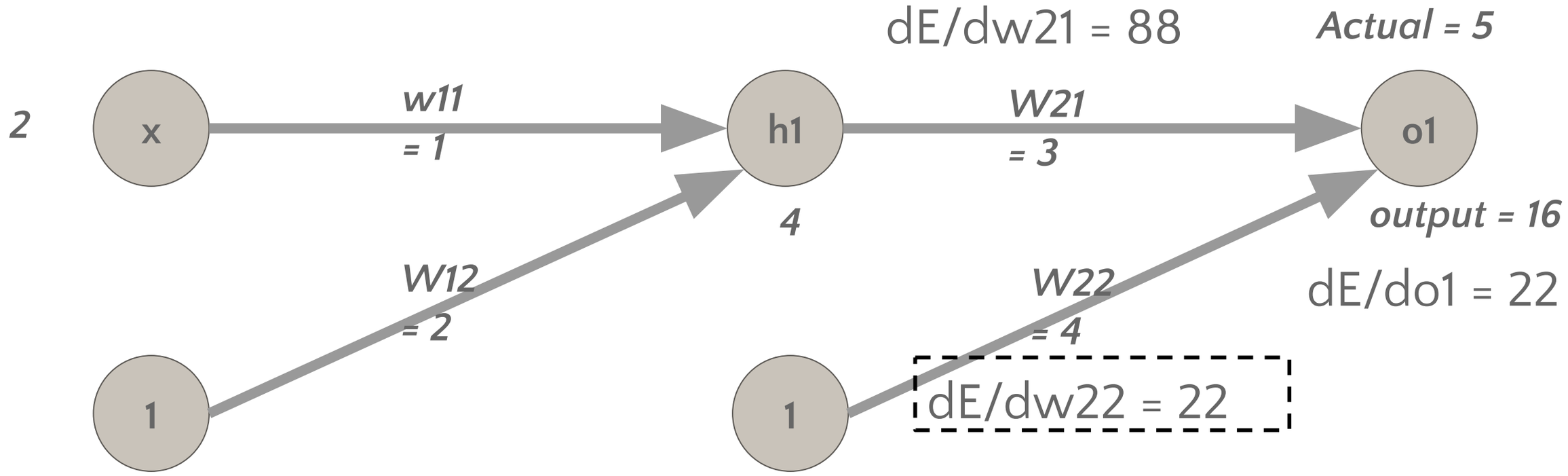
dE/dw21

= dE/do1 * do1/dw21

= 22 * 4

= 88

Multi-Layer Perceptron & Backpropagation



Example – Rate of change of error wrt w22

dE/dw22

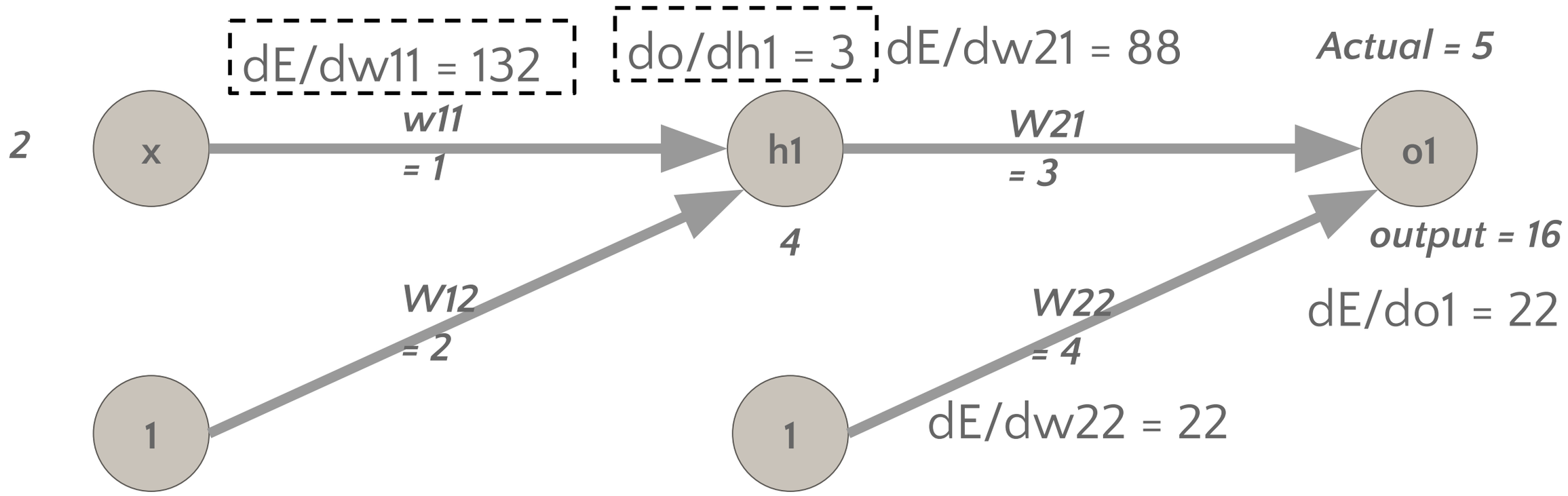
= dE/do1 * do1/dw22

= dE/do1 * d(h1 * w21 + w22) /dw22

= 22 * 1

= 22

Multi-Layer Perceptron & Backpropagation



Example – Rate of change of error wrt w11

dE/dw11

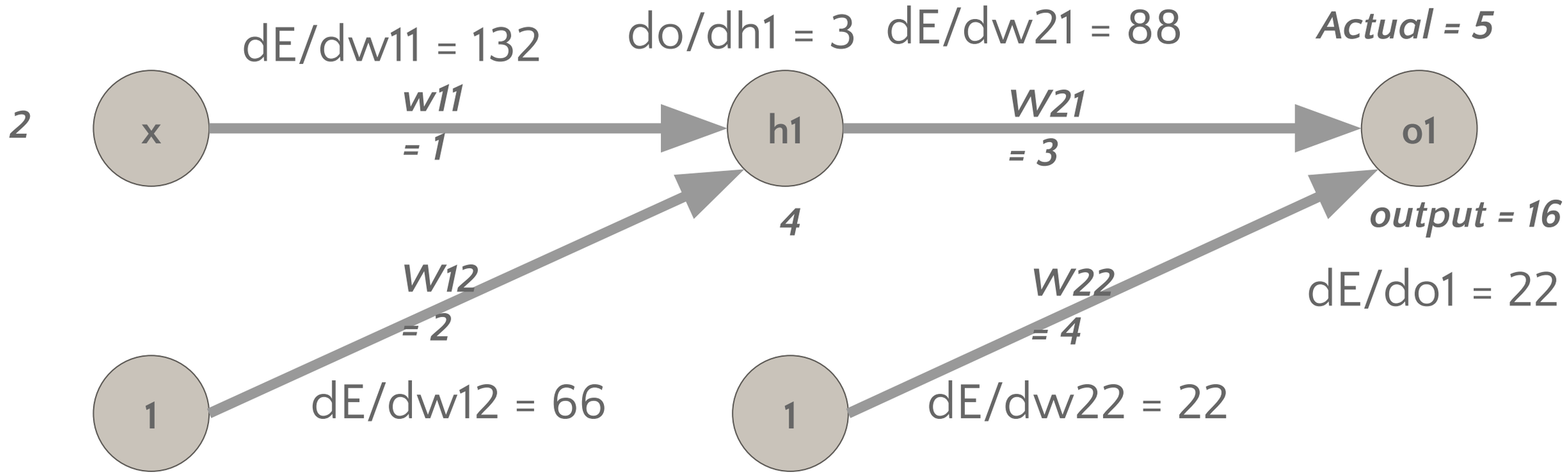
= dE/do1 * do1/dh1 * dh1/dw11

= 22 * d(w21*h1 + w22)/dh1 * d(w11 * x + w12) / dw11

= 22 * w21* x

= 22 * 3 * 2 = 132

Multi-Layer Perceptron & Backpropagation



Example – Rate of change of error wrt w12

dE/dw12

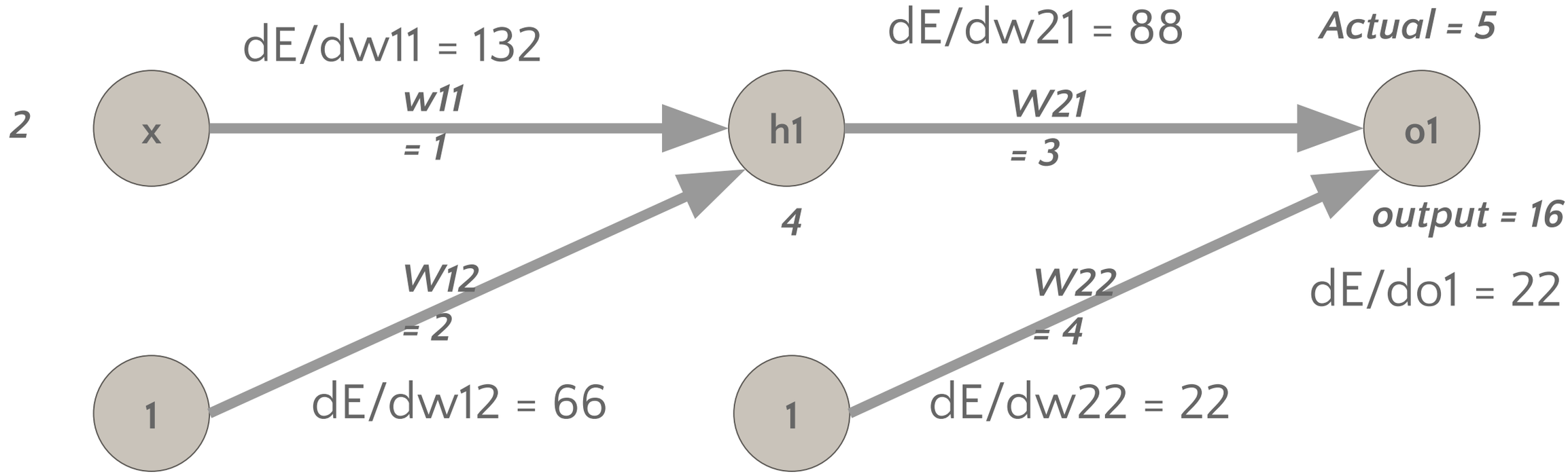
= dE/do1 * do1/dh1 * dh1/dw12

= 22 * 3 * d(w11 * x + w12) / dw12

= 22 * 3* 1

= 66

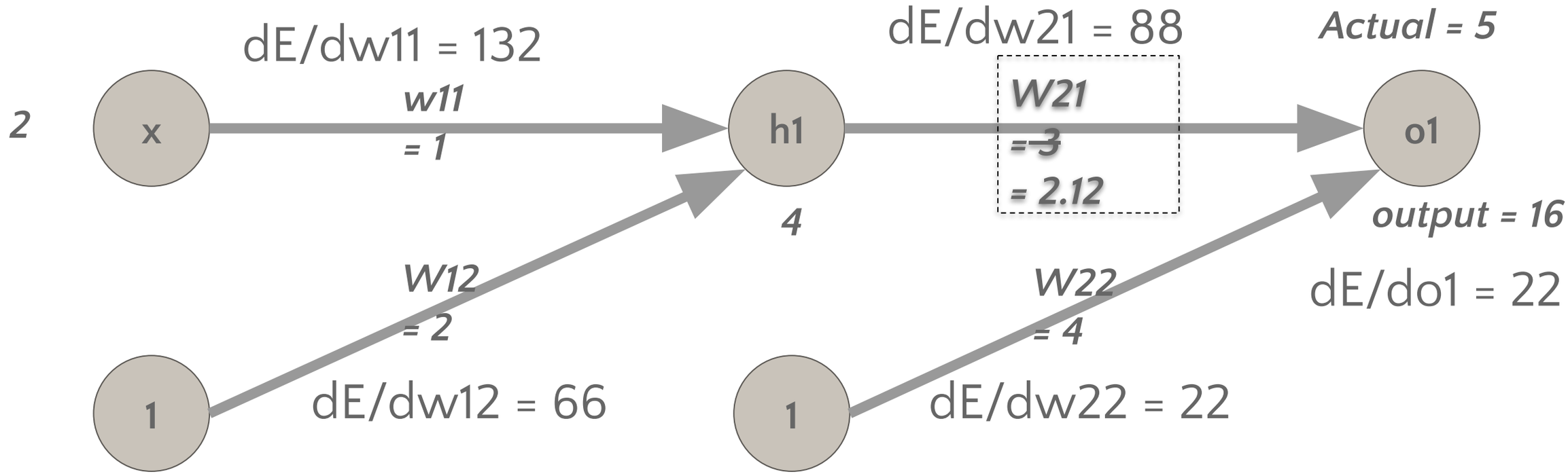
Multi-Layer Perceptron & Backpropagation



Now, we have the rates of changes of various weights. We can tweak the weights with gradient descent:

$$W_{new} = W_{old} - learning\_rate * dE/dW$$

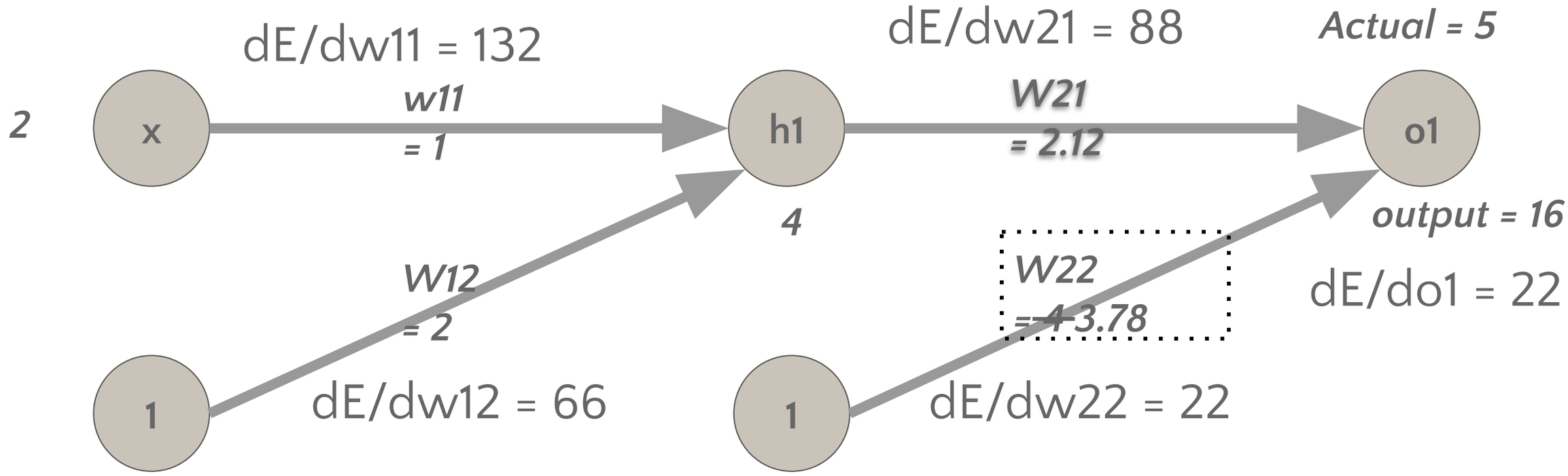
Multi-Layer Perceptron & Backpropagation



$$W21 = W21 - 0.01 * dE/dw21$$

$$= 3 - 0.01 * 88 = 2.12$$

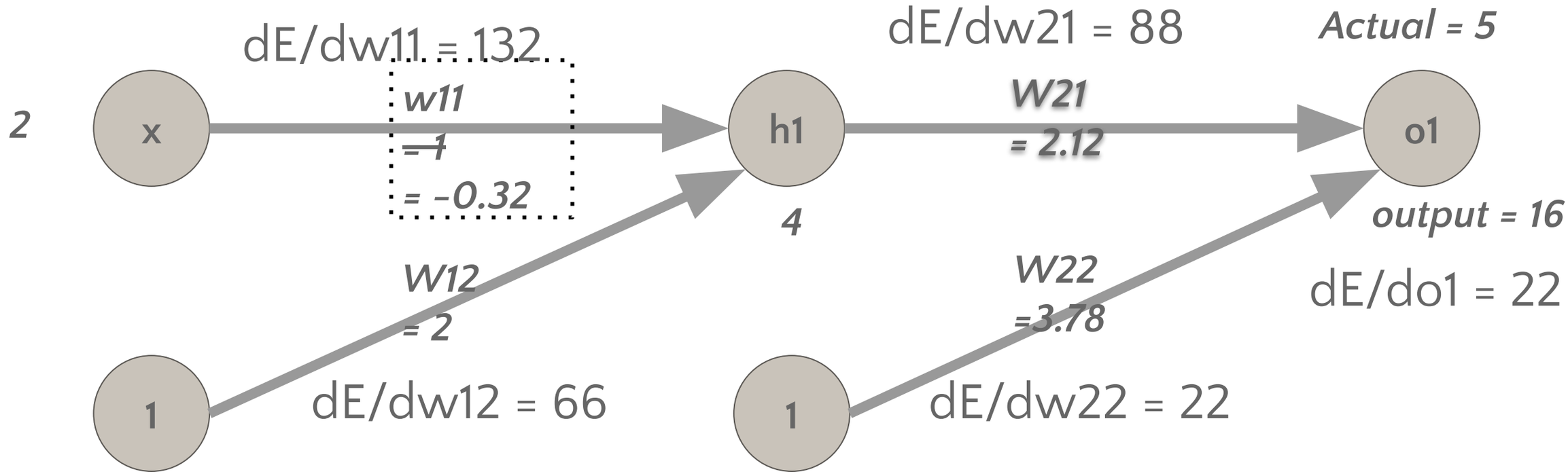
Multi-Layer Perceptron & Backpropagation



$$w22 = w22 - 0.01 * dE/ dw22$$

$$= 4 - 0.01 * 22 = 3. 78$$

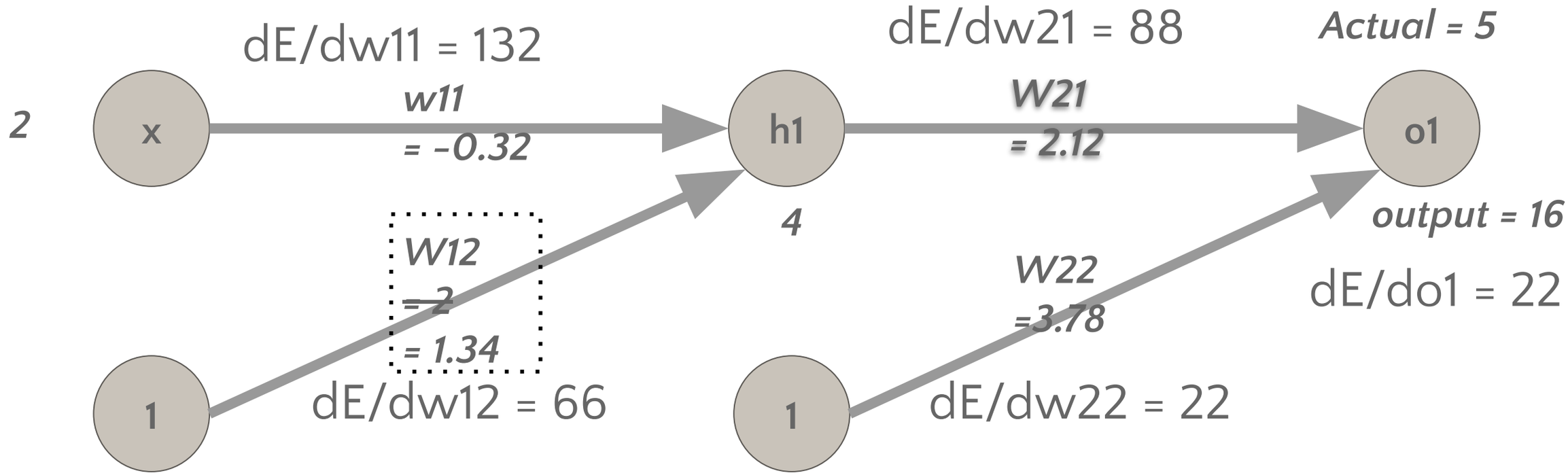
Multi-Layer Perceptron & Backpropagation



$$w11 = w11 - 0.01 * dE/dw11$$

$$= 1 - 0.01*132 = -.32$$

Multi-Layer Perceptron & Backpropagation

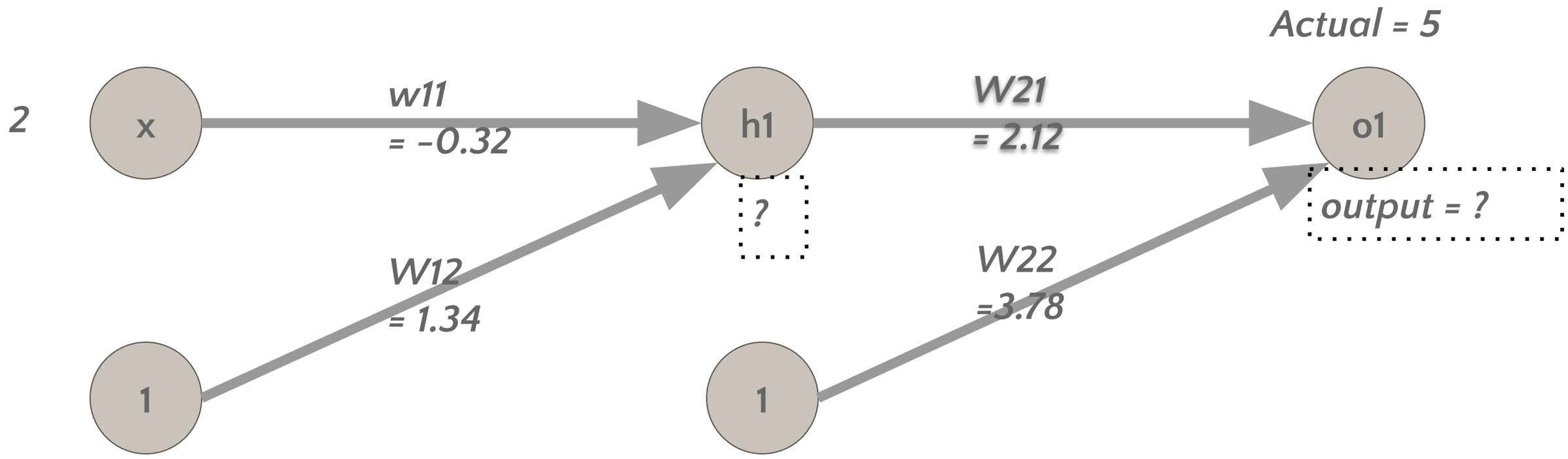


$$w12 = w12 - 0.01 * dE/dw12$$

$$= 1 - 0.01*66$$

$$= 1.34$$

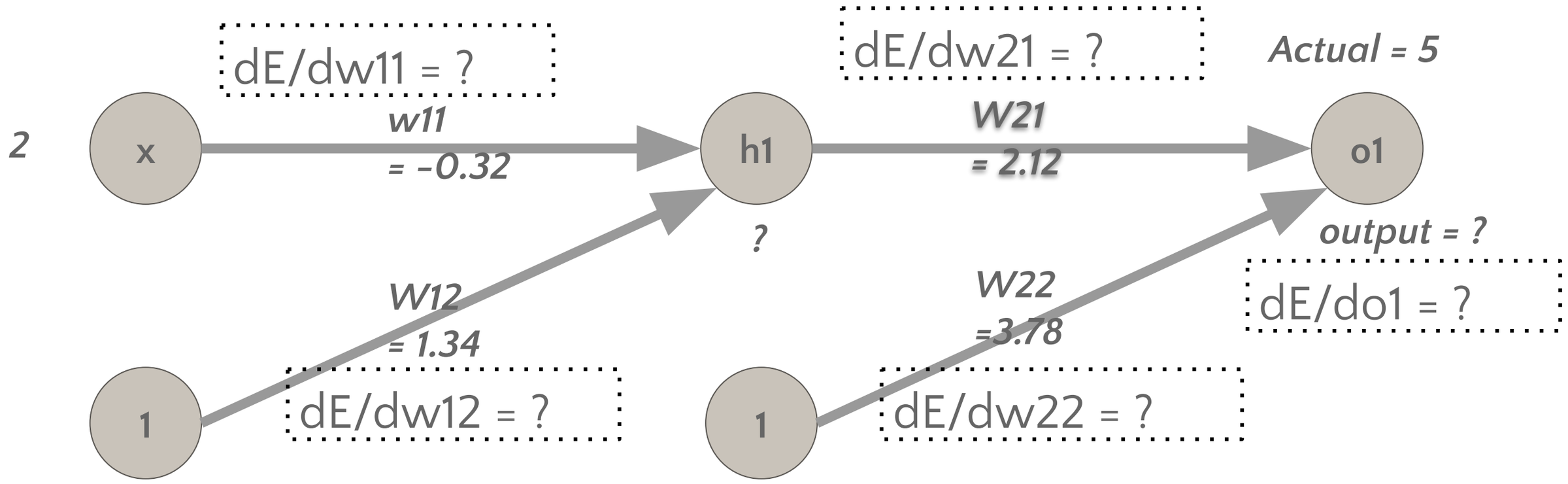
Multi-Layer Perceptron & Backpropagation



We will start the process again!

1. Forward Pass: Find the values of h1 and outputs

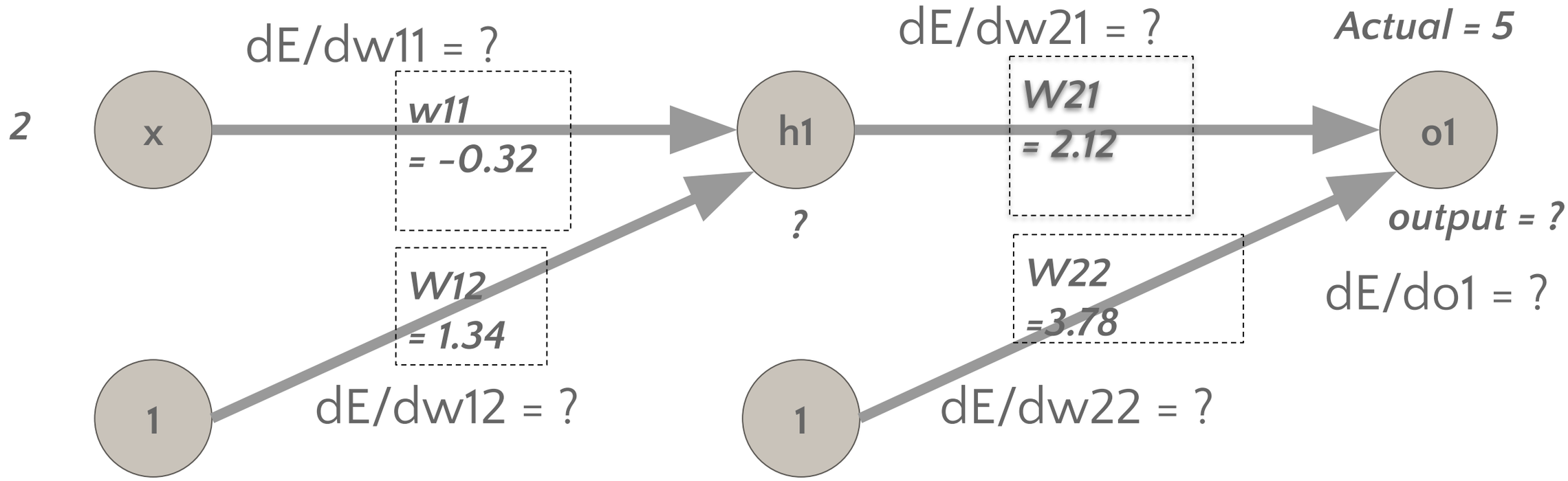
Multi-Layer Perceptron & Backpropagation



We will start the process again!

1. Forward Pass: Find the values of h1 and outputs
2. **Back propagation: Find the gradients**
3. Compute the weights

Multi-Layer Perceptron & Backpropagation



We will start the process again!

1. Forward Pass: Find the values of h1 and outputs
2. Back propagation: Find the gradients
3. **Compute the weights**

Multi-Layer Perceptron & Backpropagation

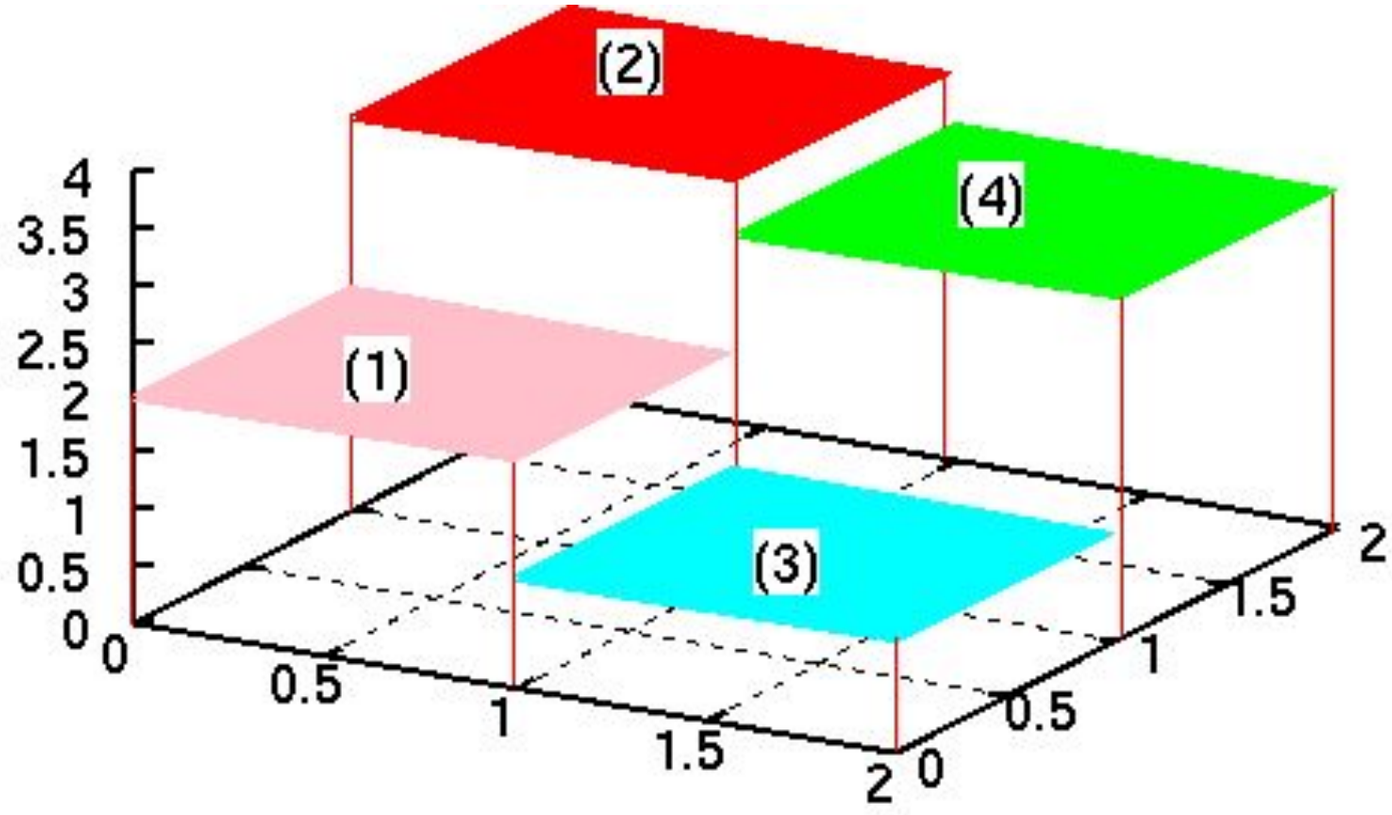
Let's summarize what we learnt about Backpropagation

1. **Forward pass** - for each training instance the backpropagation algorithm first makes a prediction
2. Measures the error
3. **Reverse pass** - then goes through each layer in reverse to measure the error contribution from each connection
4. **Gradient Descent step** - finally slightly tweaks the connection weights to reduce the error

Multi-Layer Perceptron & Backpropagation

How can we apply Gradient Descent to a output from step function ??

- Step function contains only **flat segments**, so there is no gradient to work with
- Gradient Descent cannot move on a flat surface



$$Y = f(x)$$

$$dy/dx = (f(x+delta) - f(x) / delta$$

$$dy/dx = f(x-delta) - f(x)/delta$$

$$= f(x+delta) - f(x-delta)/2*delta$$

How can we apply Gradient Descent to a output from step function ??

The solution

- In order for Gradient Descent to work properly, the key change to the MLP's architecture was made
- They replaced the step function with the logistic function,

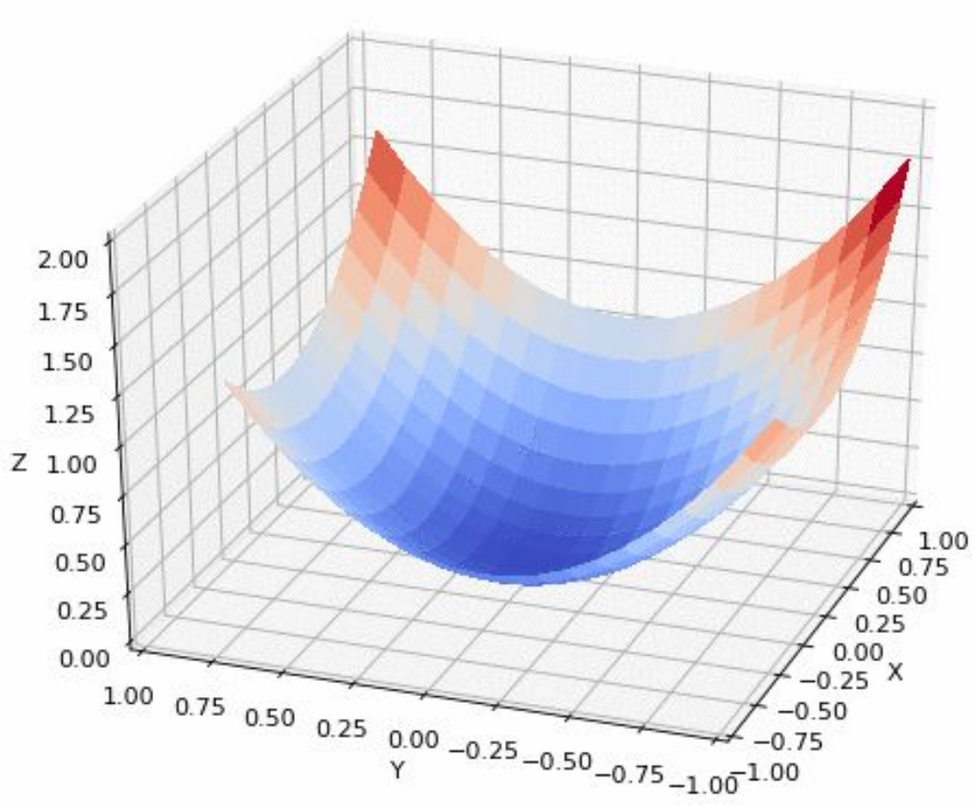
$$\sigma(z) = 1 / (1 + \exp(-z))$$

Run it on Notebook

How can we apply Gradient Descent to a output from step function ??

The solution

- The logistic function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step.



Other Activation Functions

The backpropagation algorithm may be used with other activation functions, instead of the logistic function.

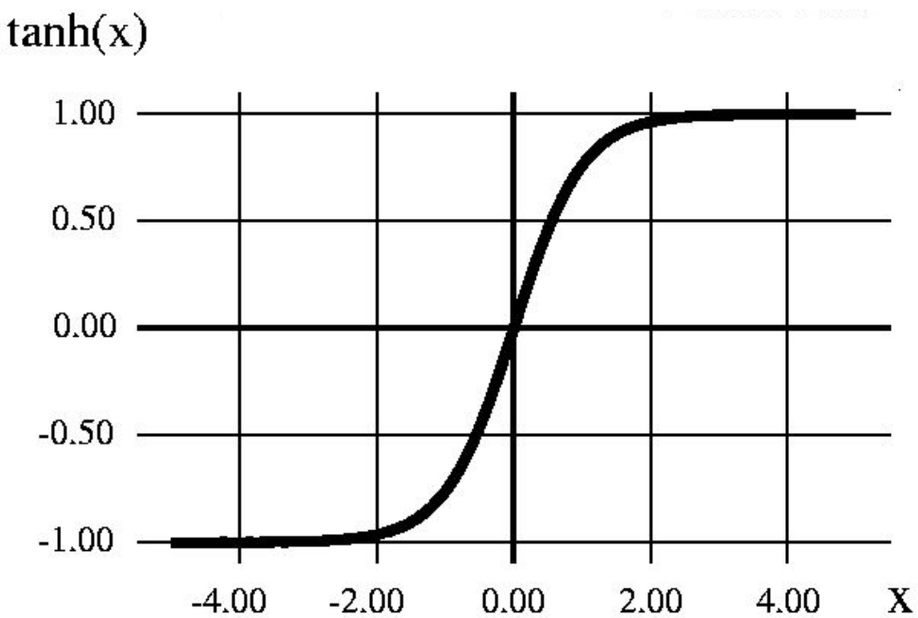
Two other popular activation functions are:

- *The hyperbolic tangent function tanh (z) = 2σ(2z) – 1*

  tanh (z) = (exp(2z)-1)/(exp(2x)+1)
- *The ReLU function*

Run it on Notebook

Other Activation Functions

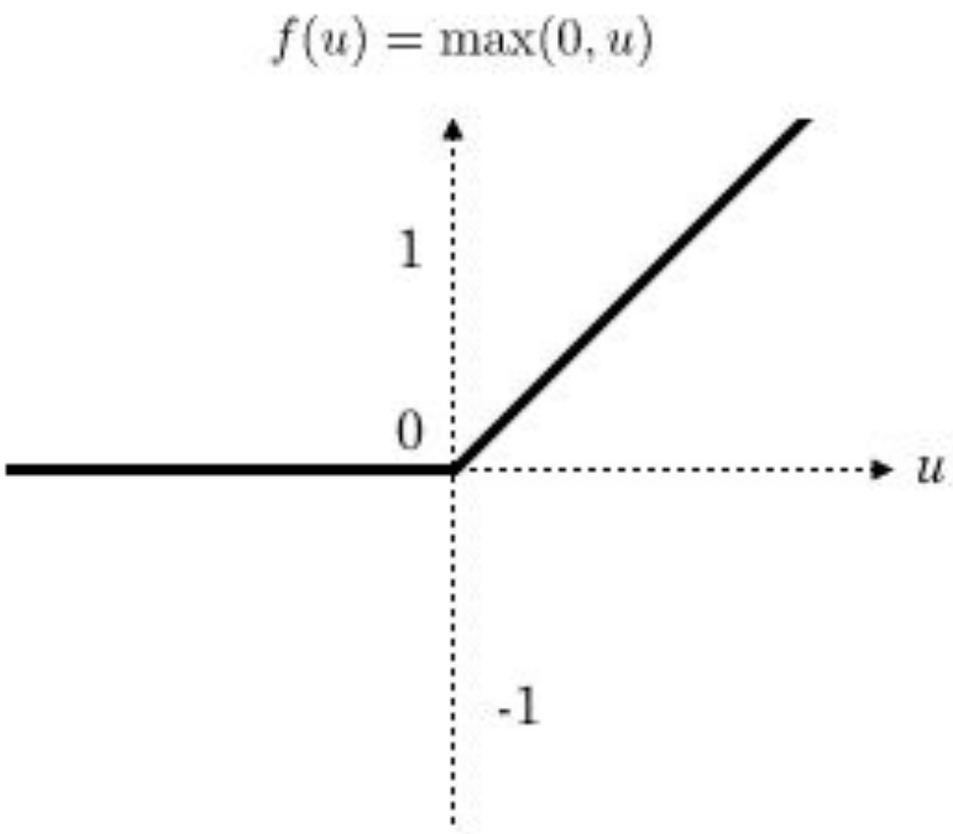


The hyperbolic tangent function tanh

(z) = 2σ(2z) – 1

- Just like the logistic function it is S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the logistic function),
- It tends to make each layer's output more or less normalized (i.e., centered around 0) at the beginning of training.
- This often helps speed up convergence.

Other Activation Functions



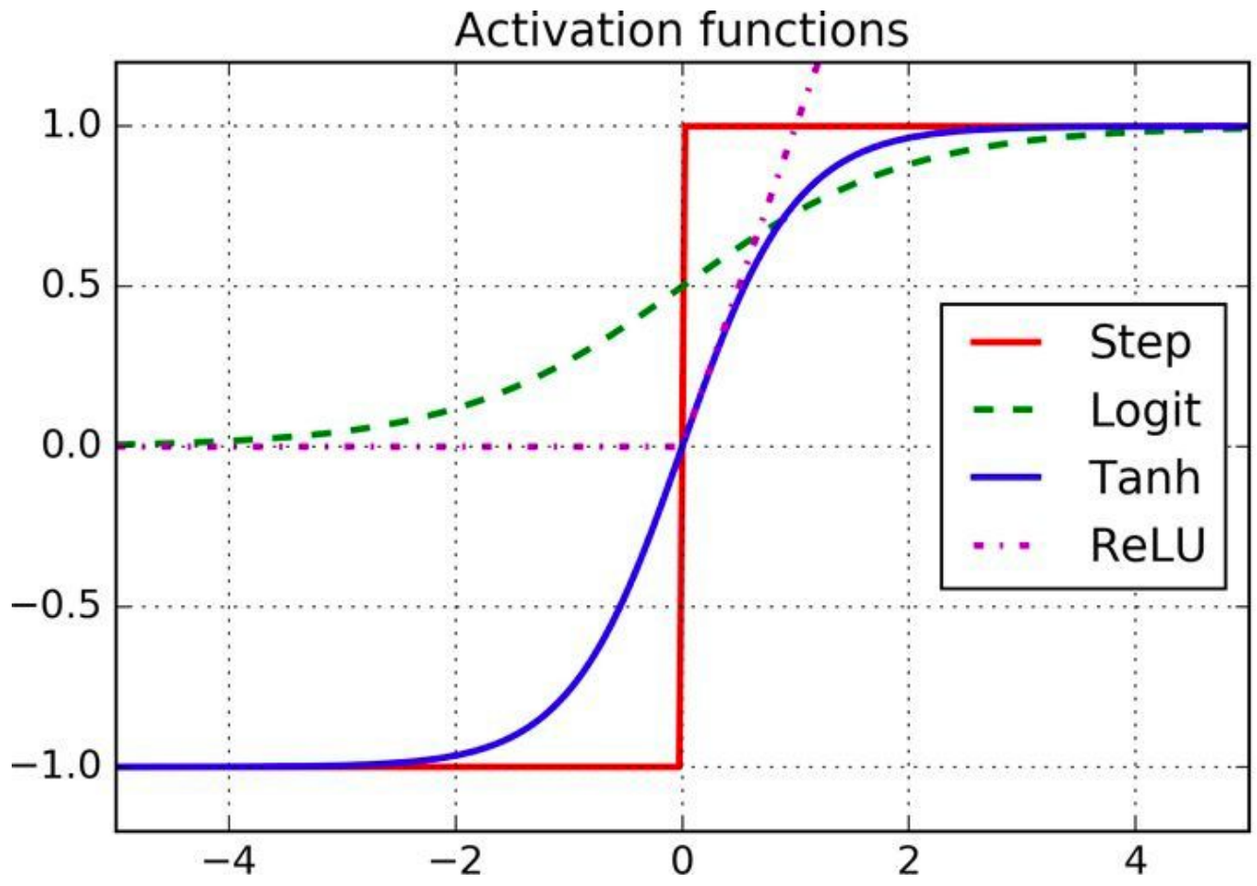
The ReLU function

ReLU (z) = max (0, z)

- It is continuous but unfortunately not differentiable at z = 0
- The slope changes abruptly, which can make Gradient Descent bounce around.
- In practice it works very well and has the advantage of being fast to compute.

Run it on Notebook

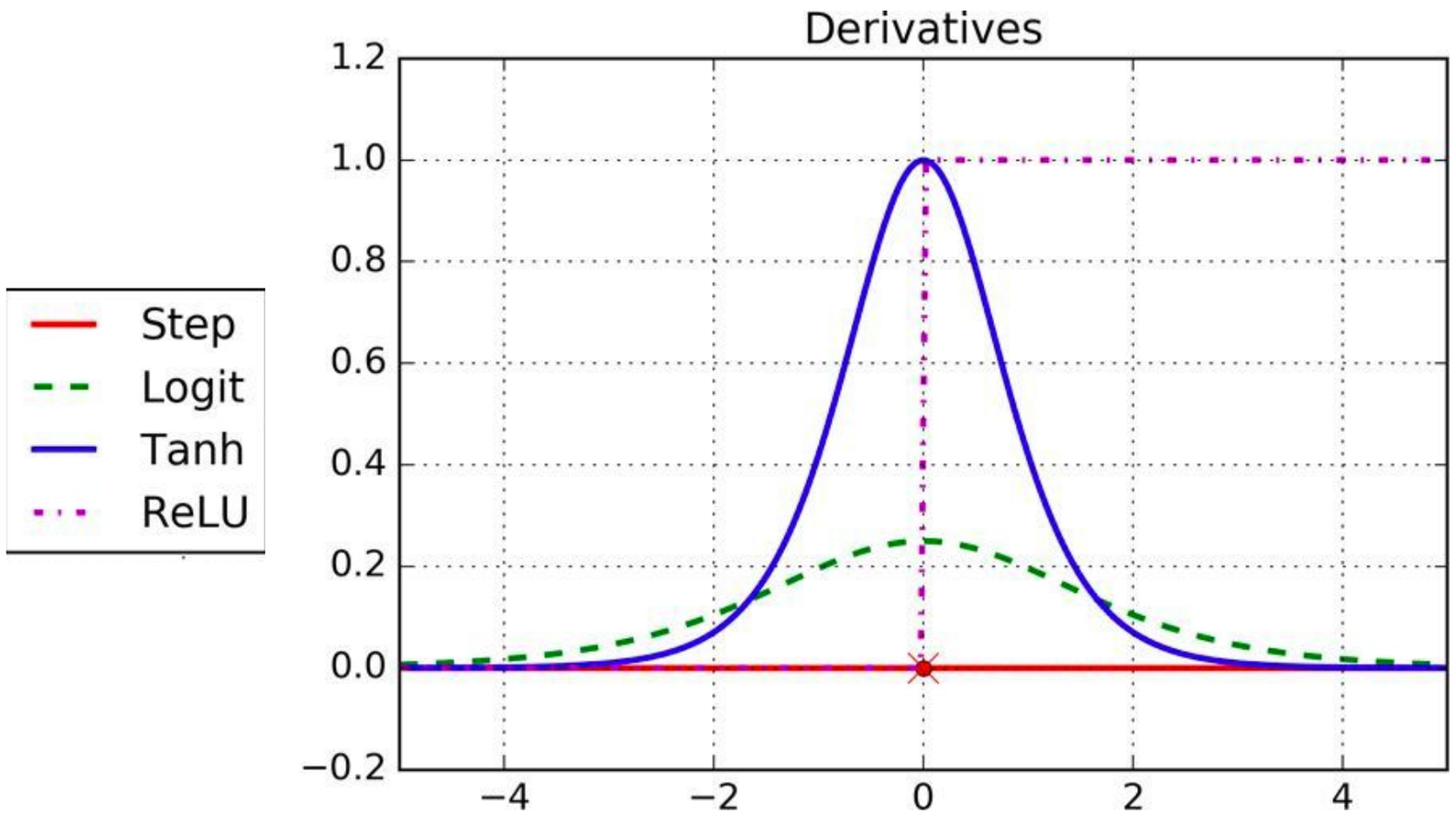
Comparison of Different Activation Functions



The above plot shows the variation of different activation functions.

Run it on Notebook

Comparison of Different Activation Functions

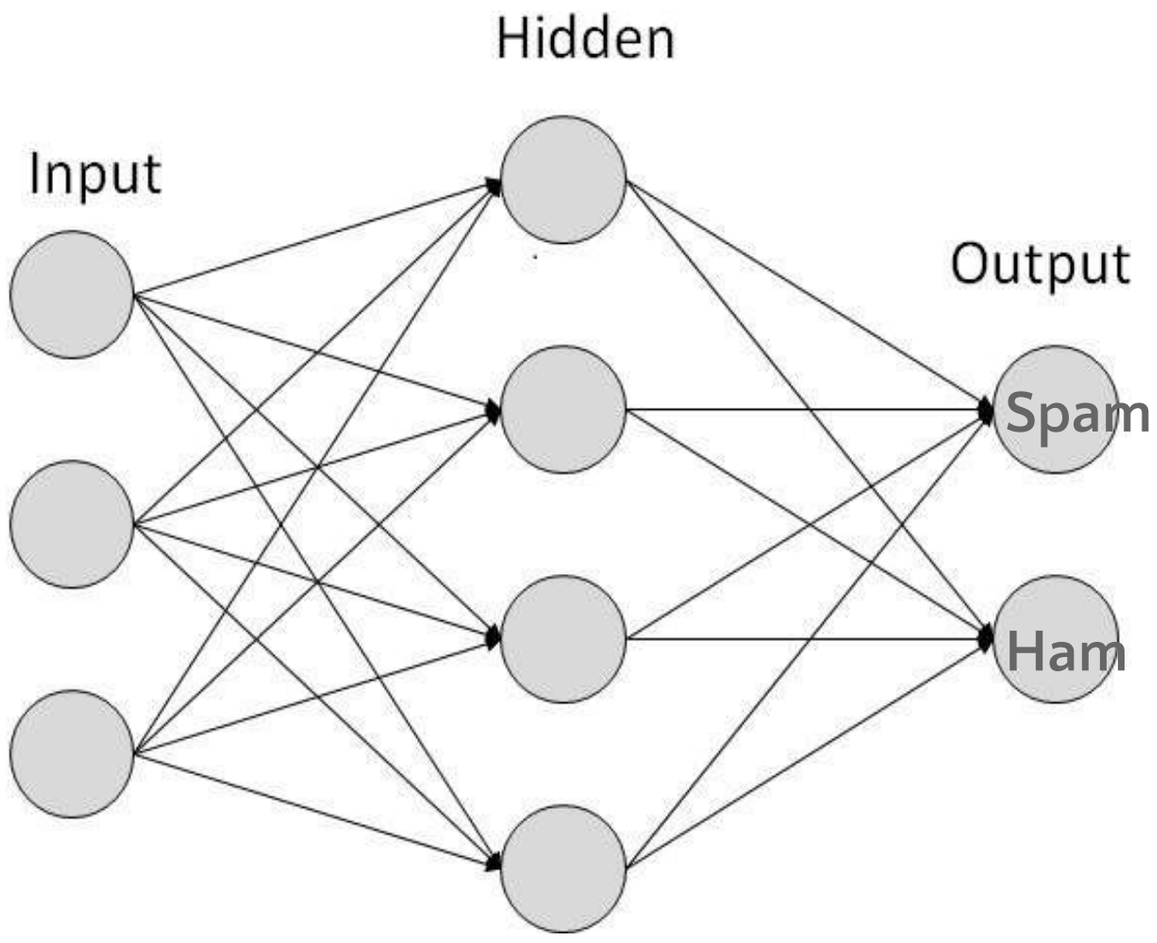


The above plot shows the derivatives of different activation functions.

Run it on Notebook

Classification with Multi-Layer Perceptron

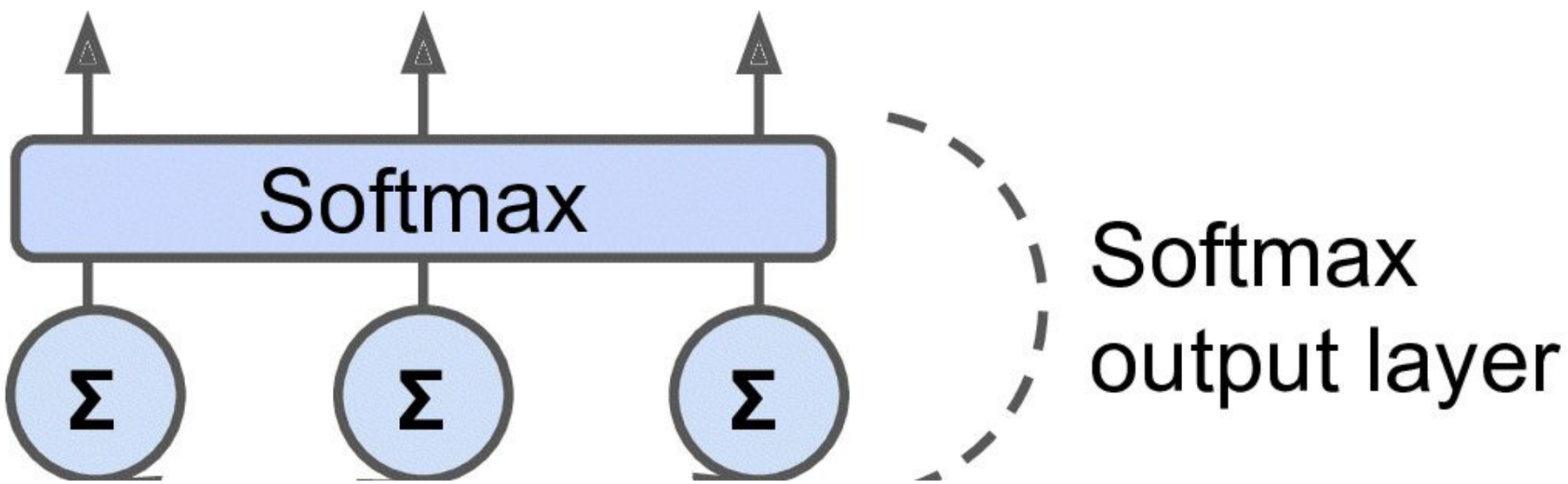
- An MLP is often used for classification, with each output corresponding to a different binary class
- Eg. spam/ham, urgent/not-urgent



Classification with Multi-Layer Perceptron

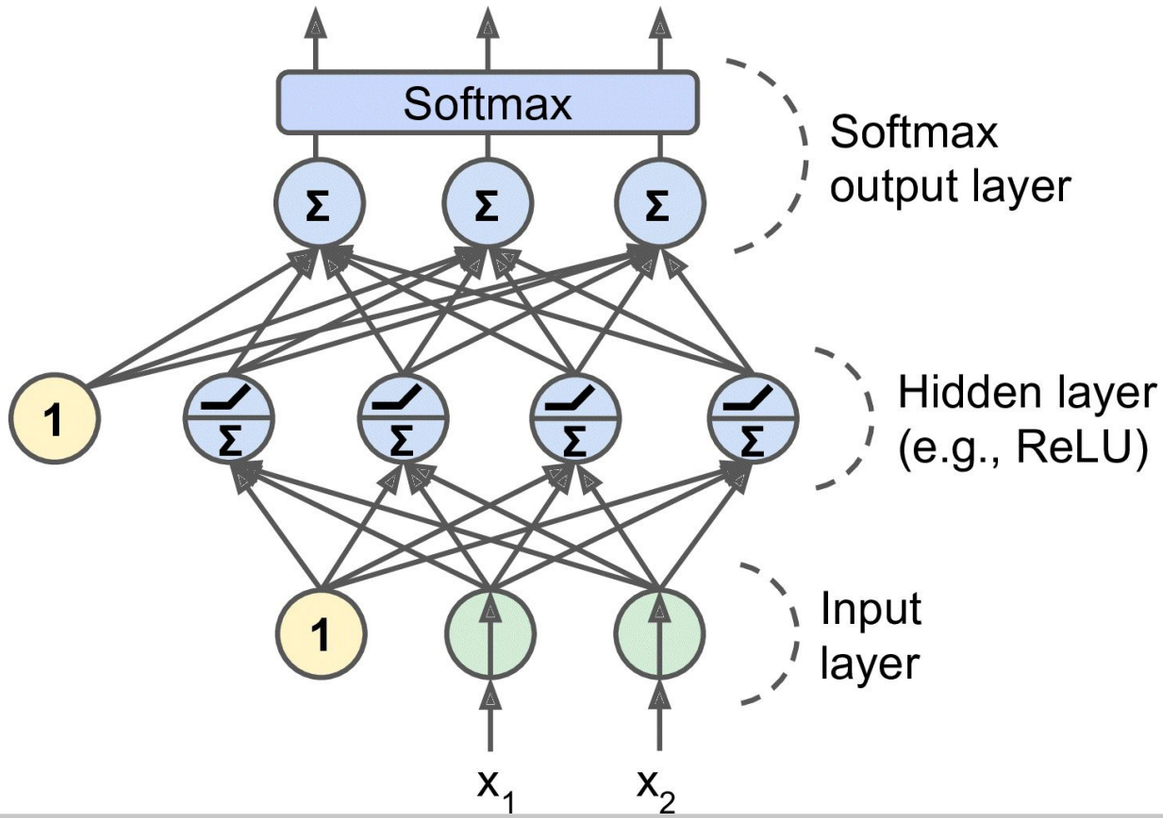
- When the classes are exclusive, the output layer is typically modified by replacing the individual activation functions by a shared softmax function
- E.g., classes 0 through 9 for digit image classification

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$



Classification with Multi-Layer Perceptron

- The output of each neuron corresponds to the estimated probability of the corresponding class
- The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a feedforward neural network (FNN)



Why Softmax

- Softmax is the activation function used at the last stage for classification neural networks
- It converts the outputs to probabilities
- The exponents ensure that even negative values are converted to positive probabilities
- It also amplifies the values due to making the highest numbers look even higher

Introduction to Keras

(We will be back to classification with MLP after quick intro to Keras)

What is Keras

Keras is a high-level Deep Learning API that allows you to

- Easily build
- Train
- Evaluate
- Execute all sorts of Neural Networks

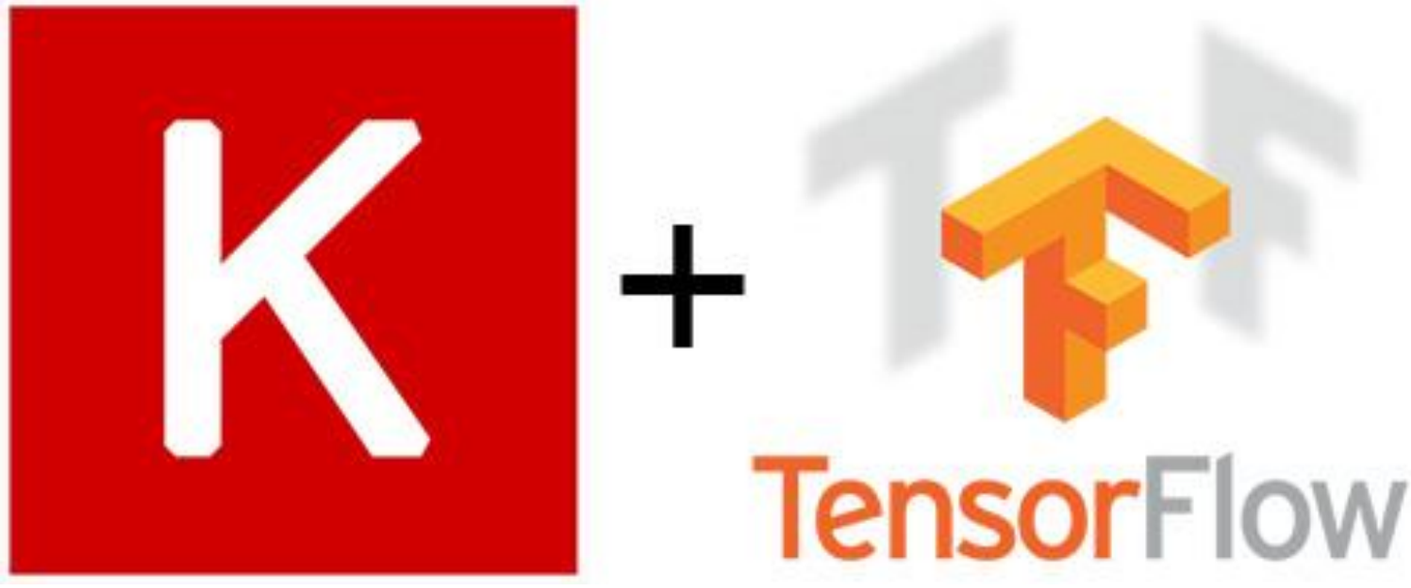
What is Keras

The reference implementation, also called Keras, was developed by **François Chollet** as part of a research project and was released as an open source project in March 2015.

What is Keras

- To perform the heavy computations required by neural networks, this reference implementation relies on a **computation backend**.
- At present, you can choose from 3 popular open source Deep Learning libraries:

  TensorFlow, Microsoft Cognitive Toolkit (CNTK), and Theano. **We will be using Keras with TensorFlow backend.**



Features of using Keras

Keras has the following features

- Consistent, simple and extensible API
- Minimal structure
- Supports multiple platforms and backends
- User friendly framework which runs on both CPU and GPU
- High scalability of computation

Benefits of using Keras

Keras' highly powerful and dynamic framework has the following advantages

- Larger community support.
- Easy to test.
- Keras neural networks are written in Python which makes things simpler.
- Keras supports both convolution and recurrent networks

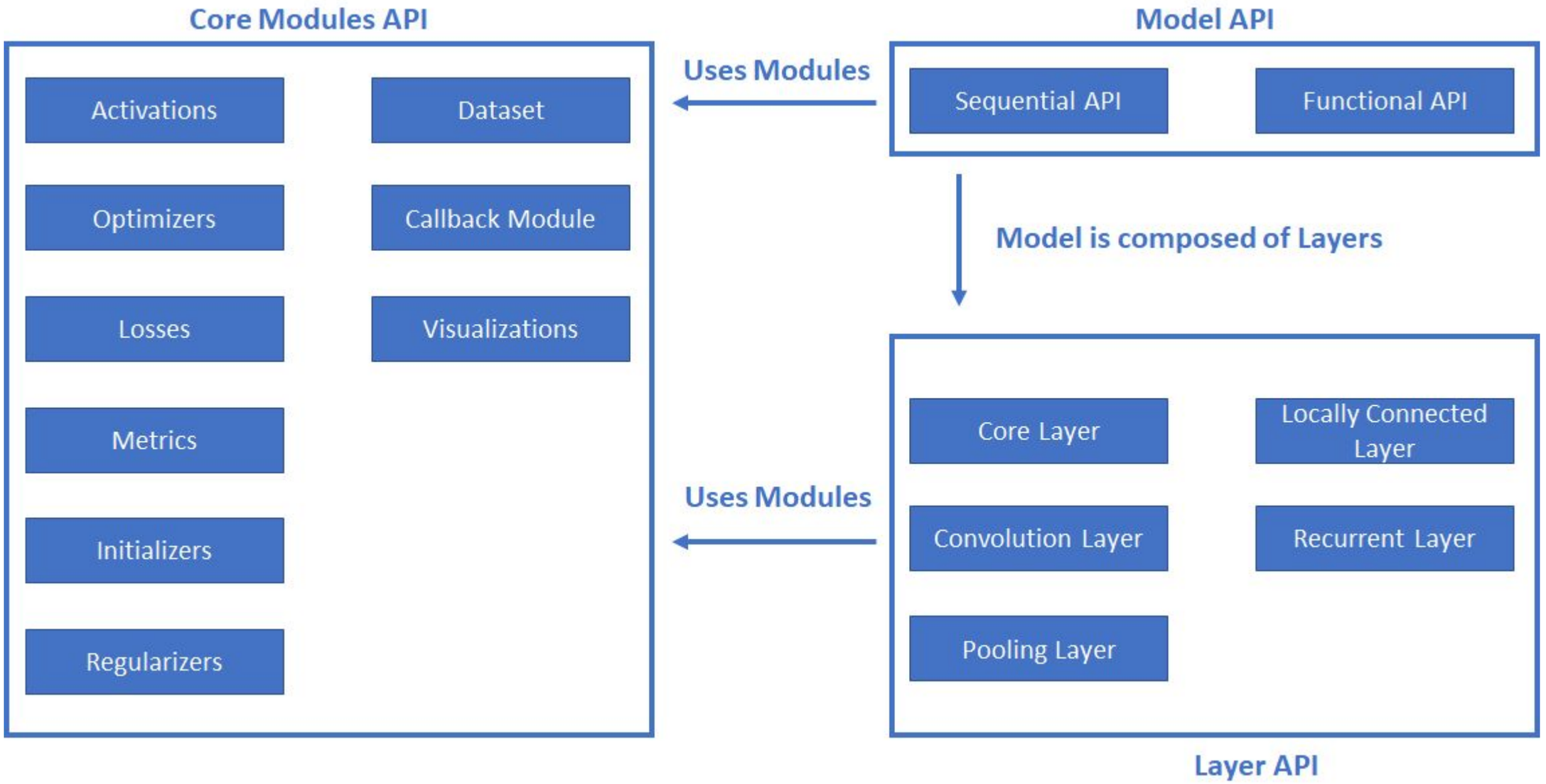
Architecture of Keras

Keras API can be divided into 3 main categories

- Model
- Layer
- Core Modules

Architecture of Keras

The following diagram depicts the relationship between model, layer and core modules



Keras Model

Keras has 2 types of models

- Sequential model
- Functional API

Keras Model

Sequential model is basically a linear composition of Keras Layers. It exposes Model class to create customized models as well. We can use sub-classing concept to create our own complex model. Eg:

```
>>> model = keras.models.Sequential()
>>> model.add(keras.layers.Flatten(input_shape=[28, 28]))
>>> model.add(keras.layers.Dense(300, activation="relu"))
```

Run it on Notebook

Keras Model

```
>>> model.add(keras.layers.Dense(100, activation="relu"))
>>> model.add(keras.layers.Dense(10,
activation="softmax"))
```

Functional API is basically used to create complex models.

Run it on Notebook

Keras Layer

Each **Keras layer** in the Keras model represent the corresponding layer (input layer, hidden layer and output layer) in the actual proposed neural network model. Keras provides a lot of **pre-build layers** so that any complex neural network can be easily created.

Keras Layer

Some of the important Keras layers are specified below,

- Core Layers
- Convolution Layers
- Pooling Layers
- Recurrent Layers

Keras Core Modules

Keras also provides a lot of built-in neural network related functions to properly create the Keras model and Keras layers. Some of the function are as follows

- Activations module
- Loss module
- Optimizer module
- Regularizers

Hands-On

Now Let's Come Back to Original Problem – Building an Image Classifier with Keras

Building an Image Classifier with Keras

We are going to create an Image Classifier on the **Fashion MNIST dataset** using the **Keras Sequential API** with TensorFlow backend.

Here is the checklist for creating the Image Classifier with Keras:

- Using Keras to load the dataset
- Creating the model using the Sequential API
- Compiling the model
- Training and evaluating the model
- Using the model to make predictions

Building an Image Classifier with Keras

We will use Fashion MNIST Dataset for the same

Load the Dataset

Using Keras to **load the dataset**

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> fashion_mnist = keras.datasets.fashion_mnist
>>> (X_train_full, y_train_full), (X_test, y_test) =
fashion_mnist.load_data()
```

Run it on Notebook

Load the Dataset

Using Keras to **load the dataset**

```
>>> X_valid, X_train = X_train_full[:5000] / 255.0,
X_train_full[5000:] / 255.0 # Scaling the data
>>> y_valid, y_train = y_train_full[:5000],
y_train_full[5000:]
```

Run it on Notebook

Load the Dataset

```
>>> class_names = ["T-shirt/top", "Trouser", "Pullover",
"Dress", "Coat",
"Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"] #
Adding class names
```

Run it on Notebook

Building an Image Classifier with Keras

We have completed loading the dataset, now will will create the model:

- Using Keras to load the dataset ✓
- Creating the model using the Sequential API
- Compiling the model
- Training and evaluating the model
- Using the model to make predictions

Create the Model

Creating the model using the Sequential API

```
>>> model = keras.models.Sequential()
>>> model.add(keras.layers.Flatten(input_shape=[28, 28]))
>>> model.add(keras.layers.Dense(300, activation="relu"))
>>> model.add(keras.layers.Dense(100, activation="relu"))
>>>model.add(keras.layers.Dense(10,
activation="softmax"))
```

Run it on Notebook

Create the Model

The codes explained!

```
>>> model = keras.models.Sequential()
```

This line creates the Sequential model.

```
>>> model.add(keras.layers.Flatten(input_shape=[28, 28]))
```

This layer convert each input image into a 1D array: if it receives input data X, it computes X.reshape(-1, 1).

Run it on Notebook

Create the Model

The codes explained!

```
>>> model.add(keras.layers.Dense(300, activation="relu"))
>>> model.add(keras.layers.Dense(100, activation="relu"))
```

These are the hidden layers with 300, and 100 neurons respectively. It will use the ReLU activation function.

Run it on Notebook

Create the Model

```
>>>model.add(keras.layers.Dense(10,
activation="softmax"))
```

This is the output layer with 10 neurons (one per class), using the softmax activation function (because the classes are exclusive).

Run it on Notebook

Create the Model

We can write the same code in one line as shown below:

```
>>> model = keras.models.Sequential([
        keras.layers.Flatten(input_shape=[28, 28]),
        keras.layers.Dense(300, activation="relu"),
        keras.layers.Dense(100, activation="relu"),
        keras.layers.Dense(10, activation="softmax")
    ])
```

Run it on Notebook

Building an Image Classifier with Keras

We have completed loading the dataset, now will will create the model:

- Using Keras to load the dataset ✓
- Creating the model using the Sequential API ✓
- Compiling the model
- Training and evaluating the model
- Using the model to make predictions

Compile the Model

Now we **compile the model** we created:

```
>>> model.compile(loss="sparse_categorical_crossentropy",
                  optimizer="sgd",
                  metrics=["accuracy"])
```

Run it on Notebook

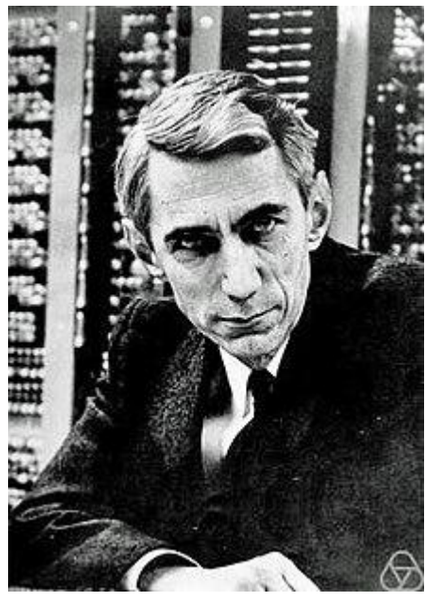
Compile the Model

We use the **"sparse_categorical_cross entropy"** loss because we have sparse labels, (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are exclusive. If instead we had one target probability per class for each instance, then we would need to use the **"categorical_crossentropy"** loss instead.

Run it on Notebook

Overview of Cross Entropy

- Cross-Entropy is a metric to find the error between predicted probabilities and actual value
- It comes from concept of information entropy proposed by Claude Shannon



Overview of Cross Entropy

- Cross-Entropy of probability distribution q, w.r.t probability distribution p
- p: Actual values
- q: Predicted values
- The log of values between 0 and 1 is -ve
- We always take the log of predicted values and not actual values

$$H(p, q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x)$$

p(x) = [0.1, 0.2, ..]
= 1

Cross Entropy, MSE and Accuracy

- Lets assume we are building a car and non-car classifier
- These are the 4 test images
- The actual values will be

(1, 0) (car, non-car)

(1, 0)

(0, 1)

(0, 1)

Cross Entropy, MSE and Accuracy

Instance	Actual		NN1 (Predicted)		NN1		
	Car	N-Car	Car	N-Car	Accuracy	MSE	Cross-Entropy
1	1	0	0.9	0.1	1	0.02	0.05
2	1	0	0.6	0.4	1	0.32	0.22
3	0	1	0.6	0.4	0	0.72	0.40
4	0	1	0.3	0.7	1	0.18	0.15

Building an Image Classifier with Keras

We have completed loading the dataset, now will will create the model:

- Using Keras to load the dataset ✓
- Creating the model using the Sequential API ✓
- Compiling the model ✓
- Training and evaluating the model
- Using the model to make predictions

Train and Evaluate the Model

Now we **train and evaluate** our model:

```
>>> history = model.fit(X_train, y_train, epochs=30,
                        validation_data=(X_valid, y_valid))
>>> model.evaluate(X_test, y_test)
```

Run it on Notebook

Building an Image Classifier with Keras

We have completed loading the dataset, now will will create the model:

- Using Keras to load the dataset ✓
- Creating the model using the Sequential API ✓
- Compiling the model ✓
- Training and evaluating the model ✓
- Using the model to make predictions

Make Predictions

And finally, we use our model to make **predictions**:

```
>>> X_new = X_test[:3]

>>> y_pred = model.predict_classes(X_new)

>>> np.array(class_names)[y_pred]


>>> y_new = y_test[:3]

>>> y_new
```

Run it on Notebook

Building an Image Classifier with Keras

We have completed loading the dataset, now will will create the model:

- Using Keras to load the dataset ✓
- Creating the model using the Sequential API ✓
- Compiling the model ✓
- Training and evaluating the model ✓
- Using the model to make predictions ✓

Now Let's Build a Regression MLP Using the Sequential API

Building a Regression MLP

Let's switch to the California housing problem and tackle it using a regression neural network:

- Using Keras to load the dataset
- Creating the model using the Sequential API
- Compiling the model
- Training and evaluating the model
- Using the model to make predictions

Building a Regression MLP

First, let's load the dataset, then divide it into training set, a validation set, and a test set, and we scale all the features.

For simplicity, we will use Scikit-Learn's fetch_california_housing() function to load the data. This dataset is simpler than the one we used in the End-to-End project since it contains only numerical features (there is no ocean_proximity feature), and there is no missing value.

Building a Regression MLP

```
>>> from sklearn.datasets import fetch_california_housing

>>> from sklearn.model_selection import train_test_split

>>> from sklearn.preprocessing import StandardScaler
```

```
>>> housing = fetch_california_housing()


>>>X_train_full, X_test, y_train_full, y_test =
train_test_split(housing.data, housing.target)
```

Run it on Notebook

Building a Regression MLP

```
>>> X_train, X_valid, y_train, y_valid =
train_test_split(X_train_full, y_train_full)
>>> scaler = StandardScaler()
>>> X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)

X_test = scaler.transform(X_test)
```

Run it on Notebook

Building a Regression MLP

Let's switch to the California housing problem and tackle it using a regression neural network:

- Using Keras to load the dataset ✓
- Creating the model using the Sequential API
- Compiling the model
- Training and evaluating the model
- Using the model to make predictions

Building a Regression MLP

Now let's create the model. Since the dataset is quite noisy, we just use a single hidden layer with fewer neurons than before, to avoid overfitting.

```
>>> model=keras.models.Sequential([keras.layers.Dense(30,
activation="relu",input_shape=X_train.shape[1:]),
keras.layers.Dense(1)])
```

Run it on Notebook

Building a Regression MLP

Let's switch to the California housing problem and tackle it using a regression neural network:

- Using Keras to load the dataset ✓
- Creating the model using the Sequential API ✓
- Compiling the model
- Training and evaluating the model
- Using the model to make predictions

Building a Regression MLP

Now we will compile the model we created:

```
>>>model.compile(loss="mean_squared_error",
optimizer="sgd")
```

Run it on Notebook

Building a Regression MLP

Let's switch to the California housing problem and tackle it using a regression neural network:

- Using Keras to load the dataset ✓
- Creating the model using the Sequential API ✓
- Compiling the model ✓
- Training and evaluating the model
- Using the model to make predictions

Building a Regression MLP

Now that we have compiled the model, let us train it, and evaluate it.

```
>>> history = model.fit(X_train, y_train, epochs=20,
validation_data=(X_valid, y_valid))
```

```
>>> mse_test = model.evaluate(X_test, y_test)
```

Run it on Notebook

Building a Regression MLP

Let's switch to the California housing problem and tackle it using a regression neural network:

- Using Keras to load the dataset ✓
- Creating the model using the Sequential API ✓
- Compiling the model ✓
- Training and evaluating the model ✓
- Using the model to make predictions

Building a Regression MLP

Let us now make some predictions with this model.

```
>>> X_new = X_test[:3] # pretend these are new instances
>>> y_pred = model.predict(X_new)
```

Run it on Notebook

Building a Regression MLP

Let's switch to the California housing problem and tackle it using a regression neural network:

- Using Keras to load the dataset ✓
- Creating the model using the Sequential API ✓
- Compiling the model ✓
- Training and evaluating the model ✓
- Using the model to make predictions ✓

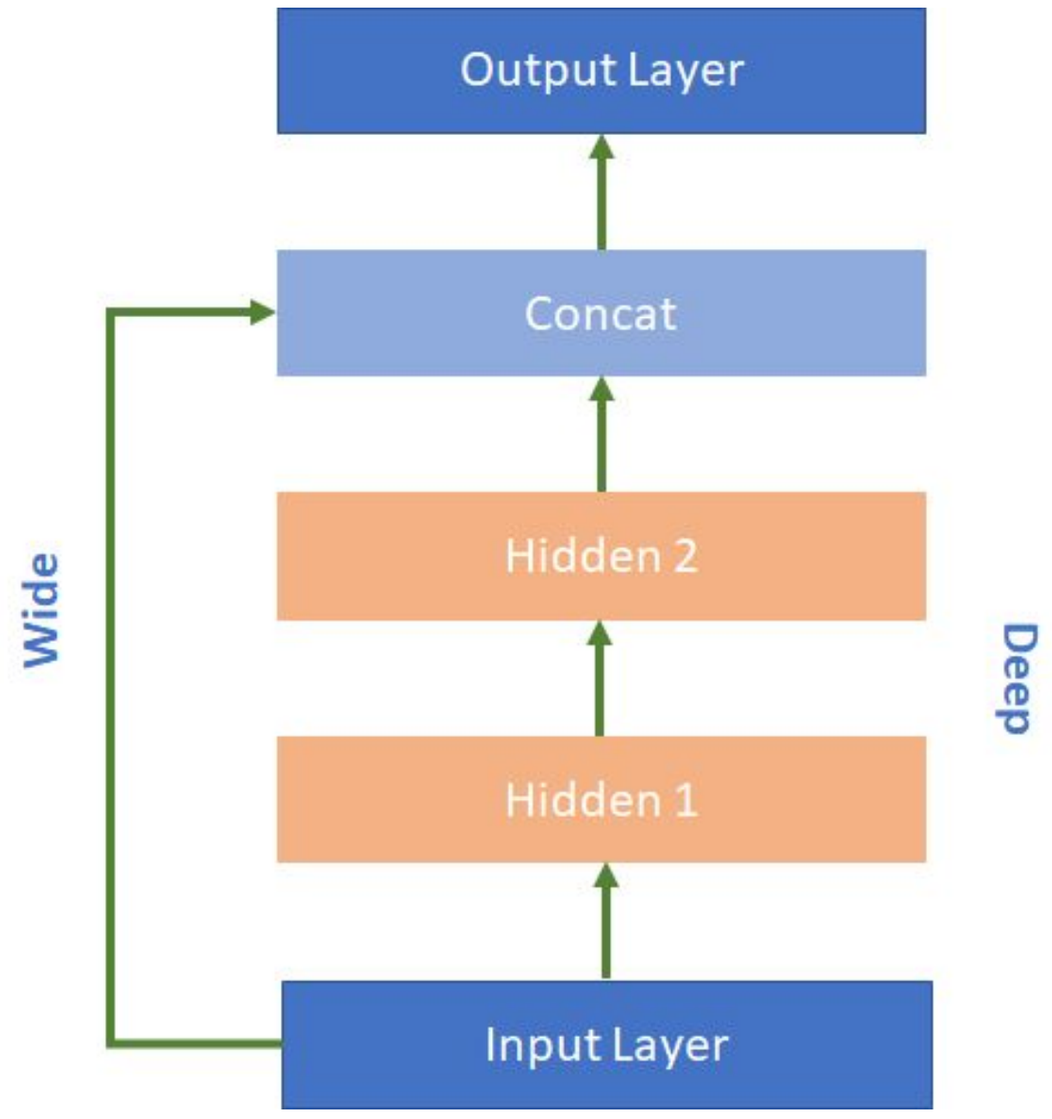
Building Complex Models Using the Functional API

Building a Complex Model

One example of a nonsequential neural network is a **Wide & Deep neural network**. This neural network architecture was introduced in a 2016 paper by Heng-Tze Cheng et al.1

Building a Complex Model

It connects all or part of the inputs directly to the output layer, as shown below:



Building a Complex Model

This architecture makes it possible for the neural network to

- learn both deep patterns (using the deep path)
- and simple rules (through the short path)

Building a Complex Model

This is an example of such a network to tackle the **California Housing** problem:

```
>>> input_ = keras.layers.Input(shape=X_train.shape[1:])
>>>hidden1=keras.layers.Dense(30,activation="relu")(input
_)
>>>hidden2=keras.layers.Dense(30,activation="relu")(hidde
n1)
```

Run it on Notebook

Building a Complex Model

```
>>> concat = keras.layers.Concatenate()([input_,
hidden2])
>>> output = keras.layers.Dense(1)(concat)
>>>model = keras.Model(inputs=[input_], outputs=[output])
```

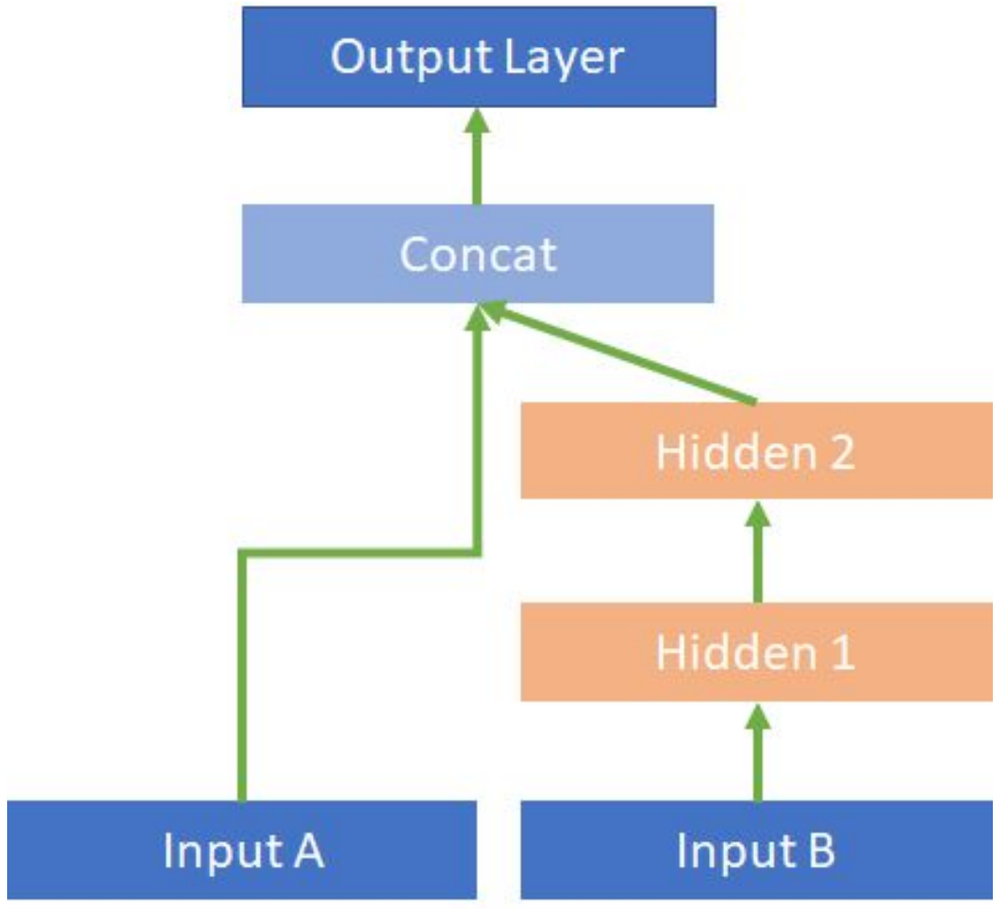
Run it on Notebook

Building a Complex Model

But what if you want to send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path?

Building a Complex Model

In this case, one solution is to use multiple inputs. For example, suppose we want to send five features through the wide path (features 0 to 4), and six features through the deep path.



Building a Complex Model

```
>>> input_A = keras.layers.Input(shape=[5],
name="wide_input")
>>> input_B = keras.layers.Input(shape=[6],
name="deep_input")
```

```
>>> hidden1 = keras.layers.Dense(30,
activation="relu")(input_B)
>>> hidden2 = keras.layers.Dense(30,
activation="relu")(hidden1)
```

Run it on Notebook

Building a Complex Model

```
>>> concat = keras.layers.concatenate([input_A, hidden2])
>>> output = keras.layers.Dense(1, name="output")(concat)
>>> model = keras.Model(inputs=[input_A, input_B],
outputs=[output])
```

Run it on Notebook

Building a Complex Model

Now we can compile the model as usual.

```
>>>model.compile(loss="mse",optimizer=keras.optimizers.SG
D(lr=1e-3))
```

Run it on Notebook

Building a Complex Model

But when we call the fit() method, instead of passing a single input matrix X_train, we must pass a pair of matrices (X_train_A, X_train_B): one per input.19 The same is true for X_valid, and also for X_test and X_new when you call evaluate() or predict().

Run it on Notebook

Building a Complex Model

```
>>> X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]

>>> X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]

>>> X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]

>>> X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

>>> history = model.fit((X_train_A, X_train_B), y_train,
epochs=20,validation_data=((X_valid_A,X_valid_B),
y_valid))
```

Run it on Notebook

Building a Complex Model

```
>>> mse_test = model.evaluate((X_test_A, X_test_B),
y_test)
>>> y_pred = model.predict((X_new_A, X_new_B))
```

Run it on Notebook

Building a Complex Model

Use cases for multiple outputs:

- Locate and classify the main object in a picture. multitask classification on pictures of faces, using one output to classify the person's facial expression (smiling, surprised, etc.) and another output to identify whether they are wearing glasses or not.
- As a regularization technique (i.e., a training constraint whose objective is to reduce overfitting and thus improve the model's ability to generalize).

Building a Complex Model

Adding extra outputs is quite easy: just connect them to the appropriate layers and add them to your model's list of outputs.

```
>>> output = keras.layers.Dense(1,
name="main_output")(concat)
>>> aux_output = keras.layers.Dense(1,
name="aux_output")(hidden2)
>>> model = keras.Model(inputs=[input_A, input_B],
outputs=[output, aux_output])
```

Run it on Notebook

Building a Complex Model

Each output will need its own loss function. Therefore, when we compile the model, we should pass a list of losses.

```
>>> model.compile(loss=["mse", "mse"], loss_weights=[0.9,
0.1], optimizer="sgd")
```

Run it on Notebook

Building a Complex Model

Now when we train the model, we need to provide labels for each output. In this example, the main output and the auxiliary output should try to predict the same thing, so they should use the same labels.

```
>>> history = model.fit( [X_train_A, X_train_B],
[y_train, y_train], epochs=20,
validation_data=([X_valid_A, X_valid_B], [y_valid,
y_valid]))
```

Run it on Notebook

Building a Complex Model

When we evaluate the model, Keras will return the total loss, as well as all the individual losses:

```
>>> total_loss, main_loss, aux_loss =
model.evaluate([X_test_A, X_test_B], [y_test, y_test])
```

Run it on Notebook

Building a Complex Model

Similarly, the predict() method will return predictions for each output:

```
>>> y_pred_main, y_pred_aux = model.predict([X_new_A,
X_new_B])
```

Run it on Notebook

Using the Subclassing API to Build Dynamic Models

Using the Subclassing

Both the Sequential API and the Functional API are declarative. This has many advantages:

- the model can easily be saved, cloned, and shared;
- its structure can be displayed and analyzed;
- the framework can infer shapes and check types;
- and more

But the flip side is just that: **it's static**.

Using the Subclassing

Simply subclass the Model class, create the layers you need in the constructor, and use them to perform the computations you want in the call() method.

Run it on Notebook

Using the Subclassing

For example, creating an instance of the following WideAndDeepModel class gives us an equivalent model to the one we just built with the Functional API. You can then compile it, evaluate it, and use it to make predictions.

Run it on Notebook

Using the Subclassing

Here we do not need to create the inputs:

- we just use the input argument to the call() method,
- and we separate the creation of the layers in the constructor from their usage in the call() method.

Using the Subclassing

The big difference is that you can do pretty much anything you want in the call() method: for loops, if statements, low-level TensorFlow operations—your imagination is the limit (see Chapter 12)! This makes it a great API for researchers experimenting with new ideas.

Saving and Restoring a Model

Saving and Restoring a Model

When using the Sequential API or the Functional API, saving a trained Keras model is as simple as it gets:

```
>>> model = keras.models.Sequential([...]) # or
keras.Model([...])
>>> model.compile([...])
>>> model.fit([...])
>>> model.save("my_keras_model.h5")
```

Run it on Notebook

Saving a Model

Keras will use the HDF5 format to save the:

- Model's architecture (including every layer's hyperparameters)
- The values of all the model parameters for every layer (e.g., connection weights and biases)
- The optimizer (including its hyperparameters and any state it may have).

Run it on Notebook

Restoring a Model

You will typically have a script that trains a model and saves it, and one or more scripts (or web services) that load the model and use it to make predictions. Loading the model is just as easy:

```
>>> model = keras.models.load_model("my_keras_model.h5")
```

Run it on Notebook

Saving the Checkpoints

The fit() method accepts a callbacks argument that lets you specify a list of objects that Keras will call at the start and end of training, at the start and end of each epoch, and even before and after processing each batch.

Run it on Notebook

Saving the Checkpoints

For example, the ModelCheckpoint callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch:

```
[...] # build and compile the model
>>> checkpoint_cb =
keras.callbacks.ModelCheckpoint("my_keras_model.h5")
>>> history = model.fit(X_train, y_train, epochs=10,
callbacks=[checkpoint_cb])
```

Run it on Notebook

Early Stopping

The following code is a simple way to implement early stopping:

```
>>> checkpoint_cb =
keras.callbacks.ModelCheckpoint("my_keras_model.h5",
save_best_only=True)
>>> history = model.fit(X_train, y_train, epochs=10,
validation_data=(X_valid, y_valid),
callbacks=[checkpoint_cb])
```

Run it on Notebook

Early Stopping

```
>>> model = keras.models.load_model("my_keras_model.h5")

# roll back to best model
```

Run it on Notebook

Early Stopping

Another way to implement early stopping is to simply use the EarlyStopping callback:

```
>>> early_stopping_cb =
keras.callbacks.EarlyStopping(patience=10,
restore_best_weights=True)
>>> history = model.fit(X_train, y_train, epochs=100,
validation_data=(X_valid, y_valid),
callbacks=[checkpoint_cb, early_stopping_cb])
```

Run it on Notebook

Tensorboard Visualization

Using Tensorboard for Visualization

TensorBoard is a great interactive visualization tool that you can use to

- View the learning curves during training
- Compare learning curves between multiple runs
- Visualize the computation graph
- Analyze training statistics

Run it on Notebook

Using Tensorboard for Visualization

TensorBoard is a great interactive visualization tool that you can use to

- View images generated by your model
- Visualize complex multidimensional data projected down to 3D and
- Automatically clustered for you
- and more!

Run it on Notebook

Using Tensorboard for Visualization

To use it, you must modify your program so that it outputs the data you want to visualize to special binary log files called **event files**. Each binary data record is called a **summary**.

In general, you want to point the TensorBoard server to a root log directory and configure your program so that it writes to a different subdirectory every time it runs.

Run it on Notebook

Using Tensorboard for Visualization

Let’s start by defining the root log directory we will use for our TensorBoard logs, plus a small function that will generate a subdirectory path based on the current date and time so that it’s different at every run.

Run it on Notebook

Using Tensorboard for Visualization

```
>>> import os
>>> root_logdir = os.path.join(os.curdir, "my_logs")
>>> def get_run_logdir():
>>>     import time
>>>     run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
>>>     return os.path.join(root_logdir, run_id)
>>> run_logdir = get_run_logdir()
```

Run it on Notebook

Using Tensorboard for Visualization

The good news is that Keras provides a nice TensorBoard() callback:

```
[...] # Build and compile your model
>>> tensorboard_cb =
keras.callbacks.TensorBoard(run_logdir)
>>> history = model.fit(X_train, y_train, epochs=30,
                        validation_data=(X_valid,
y_valid), callbacks=[tensorboard_cb])
```

Run it on Notebook

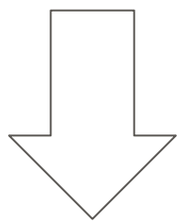
Using Tensorboard for Visualization

- Next we need to start the TensorBoard server
- Open terminal & type below command
- `/usr/local/anaconda/envs/tensorflow2/bin/tensorboard --logdir=./my_logs --port=4049 --bind_all`
- If there is an error that port is already in use
  - Then please use any other port in the range of 4040 to 4140.

Using Tensorboard for Visualization

- Once the server is started then go to below URL for accessing it
- Replace `your_web_console` with your CloudxLab web console address. For example, f.cloudxlab.com
- Replace `port` with the port number on which Tensorboard server is running. For example, 4049

```
http://your_web_console:port
```



```
http://f.cloudxlab.com:4049
```

Using Tensorboard for Visualization

Once you are done with Tensorboard, press “Ctrl + C” on your terminal to kill the server

Fine-Tuning Neural Networks Hyperparameters

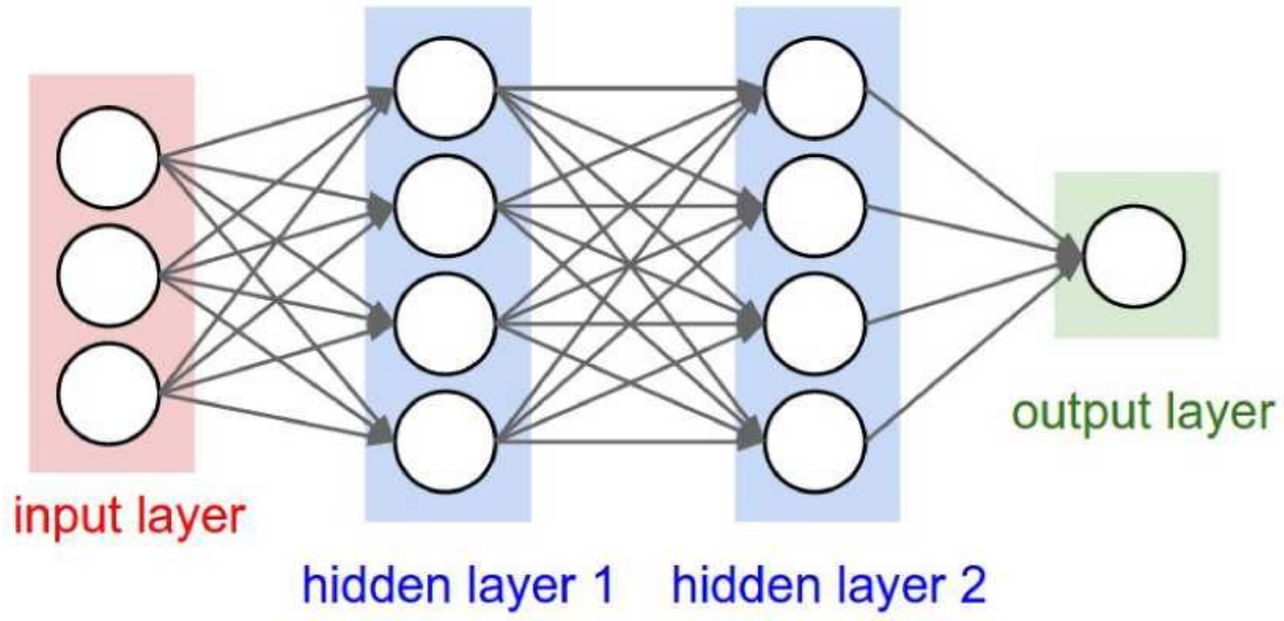
Fine-Tuning Neural Networks Hyperparameters

- The flexibility of neural networks is also one of their main drawbacks
- There are many hyperparameters to tweak.
- and...

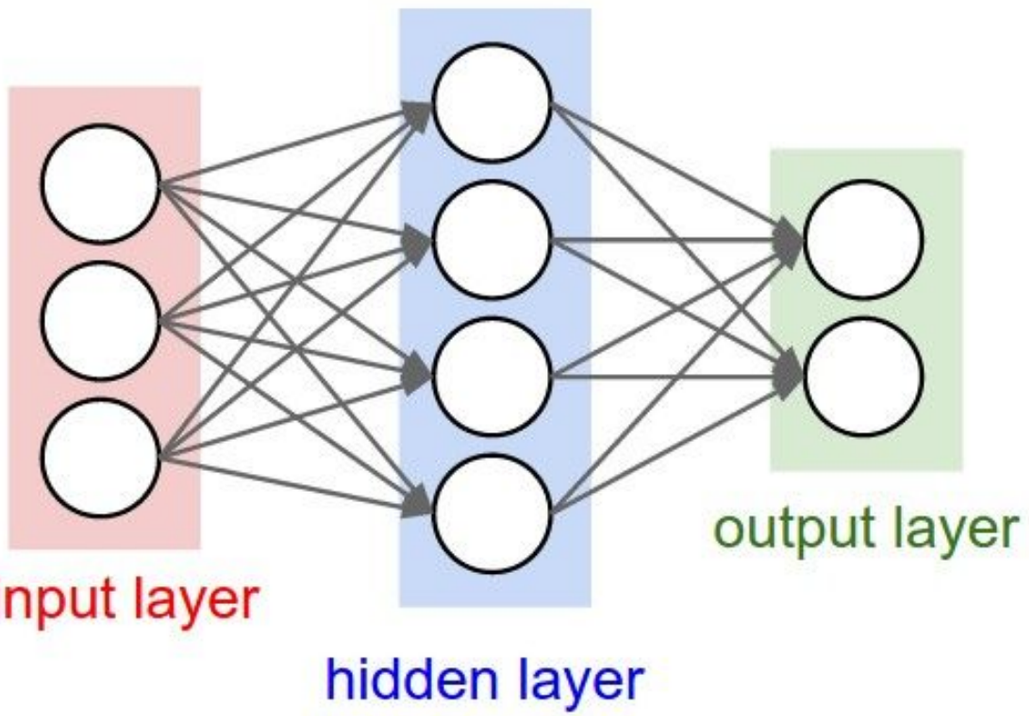
Fine-Tuning Neural Networks Hyperparameters

- You can use any imaginable network topology (how neurons are interconnected), but even in a simple MLP you can change
  - the number of layers,
  - the number of neurons per layer,
  - the type of activation function to use in each layer,
  - the weight initialization logic,
  - Learning rate
  - Epochs
  - # of batches
  - and much more.

Fine-Tuning Neural Networks Hyperparameters

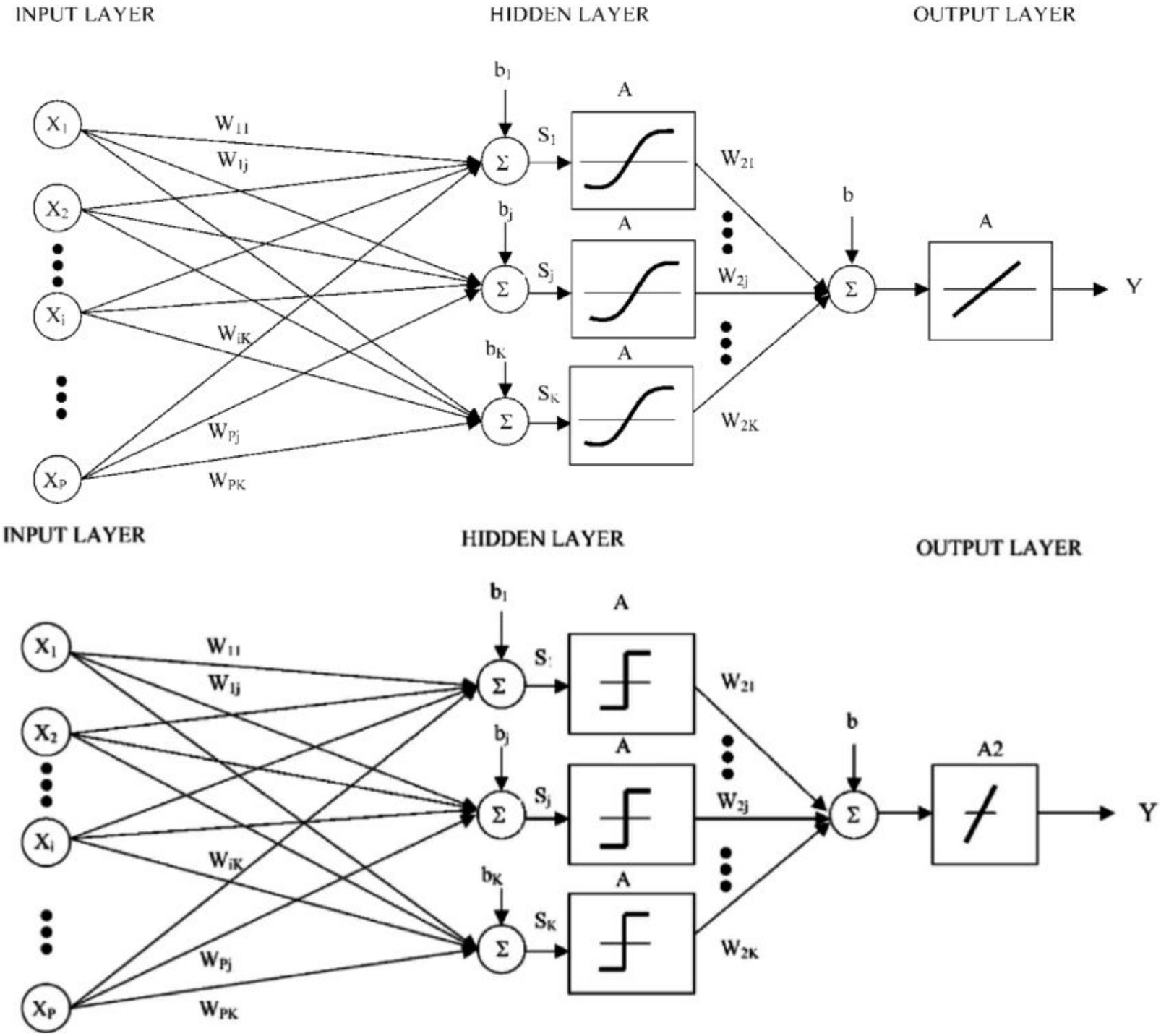


A neural network with 2 hidden layers having 4 neurons in each hidden layer



A neural network with 1 hidden layer having 4 neurons in the hidden layer

Fine-Tuning Neural Networks Hyperparameters



A neural network using Sigmoid function as the activation function in the hidden layer

A neural network using Step function as the activation function in the hidden layer

How do we know what combination of hyperparameters is the best for our task?

Fine-Tuning Neural Networks Hyperparameters

- One solution is to use Grid Search with cross-validation to find the right hyperparameters
  - But since there are **many hyperparameters** to tune,
  - And since training a neural network on **a large dataset takes a lot of time**,
  - We will only be able to explore a **tiny part** of the hyperparameter space in a reasonable amount of time
- Therefore it is much better to use **Randomized Search**

Fine-Tuning Neural Networks Hyperparameters

- It helps to have an idea of what values are reasonable for each hyperparameter
- So we can restrict the search space.

Let's start with the number of hidden layers

Fine-Tuning Neural Networks Hyperparameters

The Number of Hidden Layers – Deep or Shallow Neural Networks

- For many problems, we can just begin with a single hidden layer and you will get reasonable results

An MLP with just one hidden layer can model even the most complex functions provided it has enough neurons.

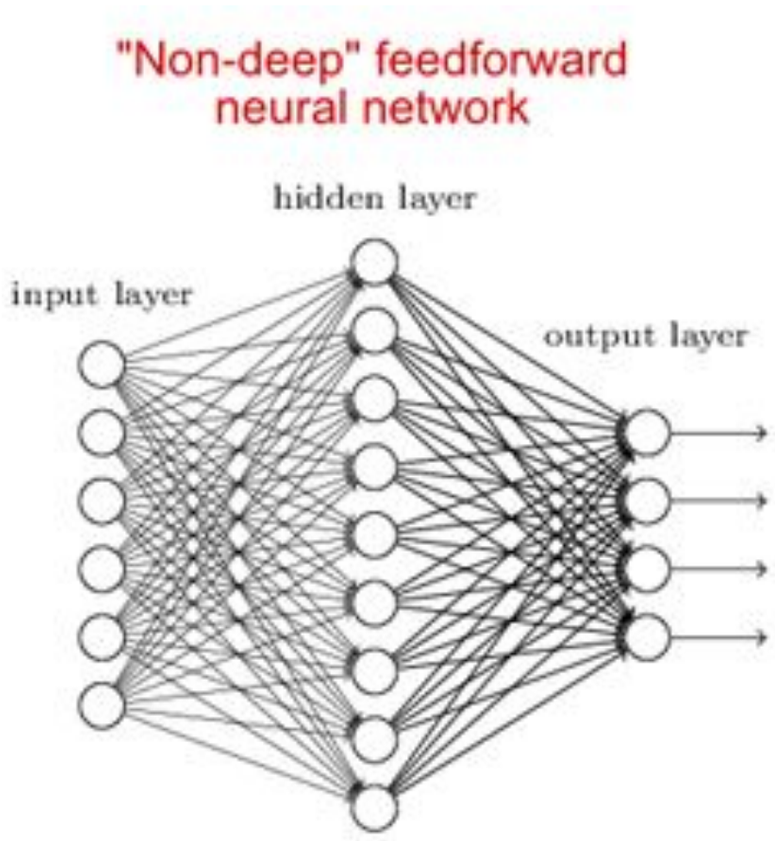
- For a long time, the facts convinced researchers that there was no need to investigate any deeper neural networks.

The Number of Hidden Layers – Deep or Shallow Neural Networks

- **But** they overlooked the fact that
  - Deep networks have a much higher **parameter efficiency** than shallow ones
  - This means they can model complex functions using exponentially fewer neurons than shallow nets
  - Making them much faster to train

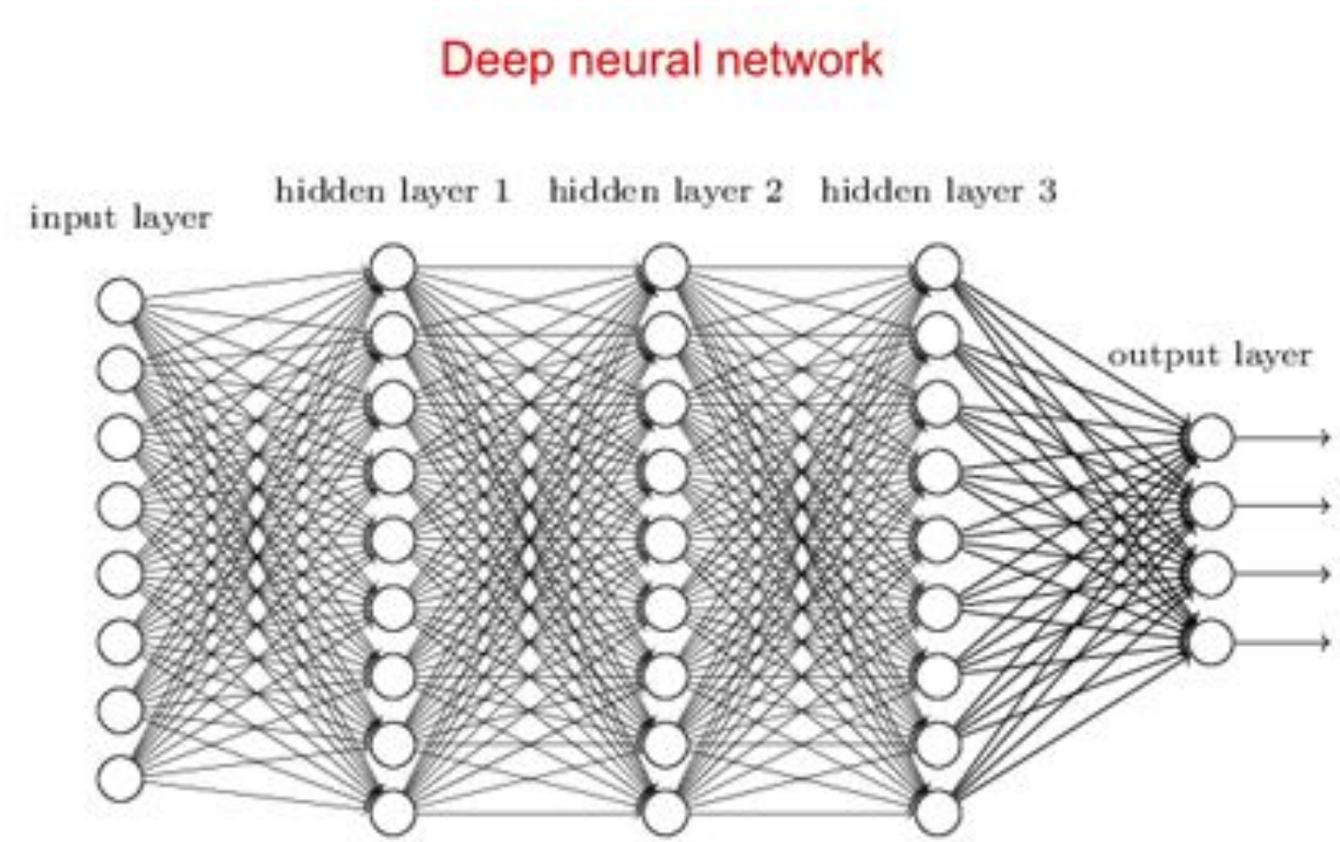
Fine-Tuning Neural Networks Hyperparameters

The Number of Hidden Layers – Deep or Shallow Neural Networks



- We can begin with a single hidden layer and get reasonable results
- Can model even the most complex functions provided it has enough neurons

The Number of Hidden Layers – Deep or Shallow Neural Networks



- Have a much higher parameter efficiency than shallow ones
- Can model complex functions using exponentially fewer neurons than shallow nets
- Much faster to train

The Number of Hidden Layers – Deep or Shallow Neural Networks

To understand why deep neural networks have a much higher parameter efficiency than shallow ones, consider the following analogy.

Suppose you are asked to draw a forest using some drawing software, but you are forbidden to use copy/paste

The Number of Hidden Layers – Deep or Shallow Neural Networks

Suppose you are asked to draw a forest using some drawing software, but you are forbidden to use copy/paste

You would have to draw each tree individually, branch per branch, leaf per leaf

The Number of Hidden Layers – Deep or Shallow Neural Networks

Suppose you are asked to draw a forest using some drawing software, but you are forbidden to use copy/paste

- If we could instead draw one leaf
- Copy/paste it to draw a branch,
- Then copy/paste that branch to create a tree,
- And finally copy/paste this tree to make a forest,

We would be finished in no time

The Number of Hidden Layers – Deep or Shallow Neural Networks

Real world data is often structured in such a hierarchical way and DNNs automatically take advantage of this fact

Let us understand it through a model that identifies faces

Fine-Tuning Neural Networks Hyperparameters

The Number of Hidden Layers – Deep or Shallow Neural Networks

In a model that recognized faces

- Lower hidden layers model low-level structures e.g., line segments of various shapes and orientations,
- Intermediate hidden layers combine these low-level structures to model intermediate-level structures (e.g., squares, circles),
- And the highest hidden layers and the output layer combine these intermediate structures to model high-level structures (e.g., faces)

The Number of Hidden Layers – Deep or Shallow Neural Networks

First Layer Representation | Second Layer Representation | Third Layer Representation

Fine-Tuning Neural Networks Hyperparameters

The Number of Hidden Layers – Deep or Shallow Neural Networks

- This hierarchical architecture help DNNs converge faster to a good solution
- It also improves their ability to generalize to new datasets

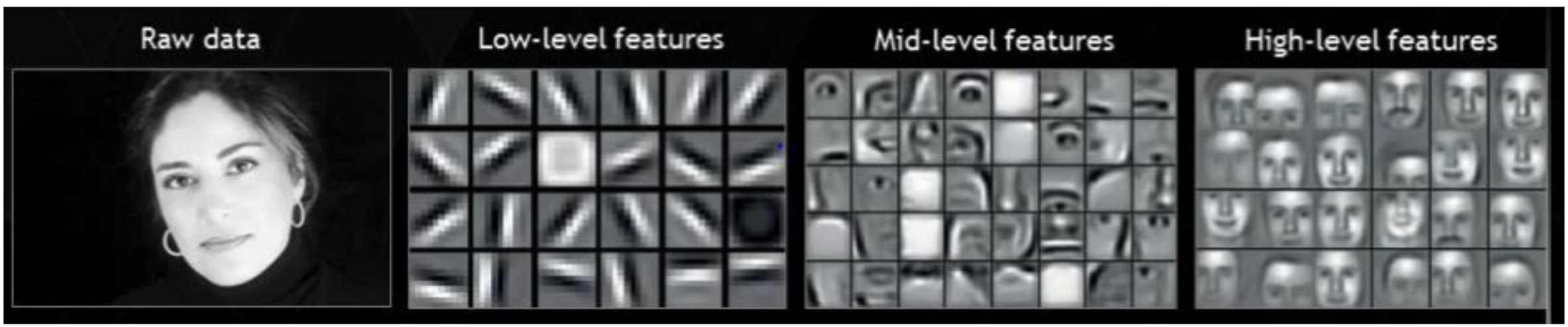
How does the hierarchical architecture help DNNs generalize to new datasets?

We will try to answer it using an example

Fine-Tuning Neural Networks Hyperparameters

The Number of Hidden Layers – Deep or Shallow Neural Networks

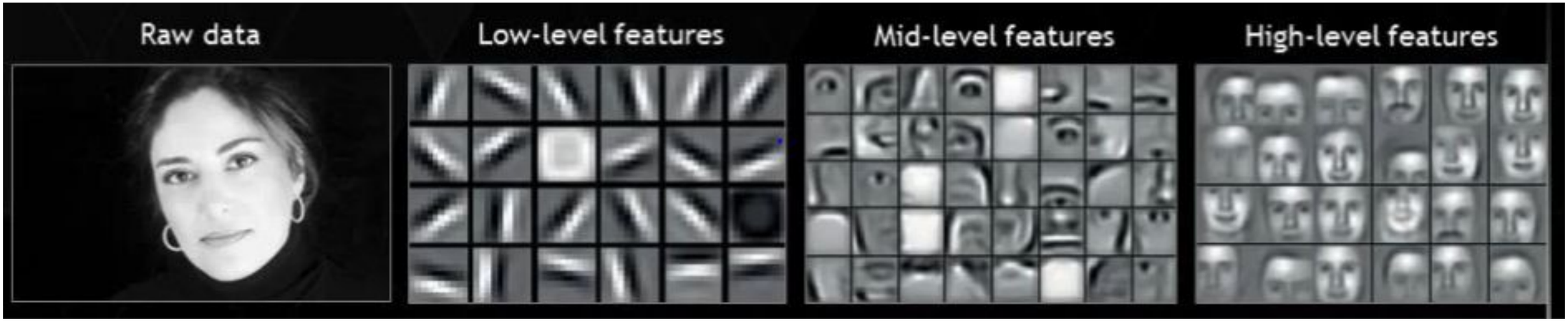
- Suppose we have already trained a model to recognize **faces** in pictures
- And we now want to train a new neural network to recognize hairstyles,



Fine-Tuning Neural Networks Hyperparameters

The Number of Hidden Layers – Deep or Shallow Neural Networks

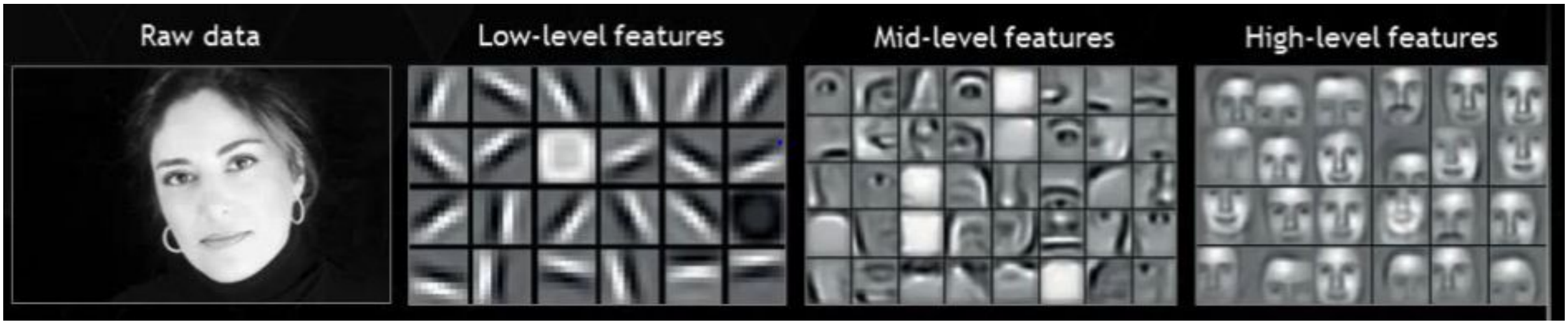
- We can kickstart training by reusing the lower layers of the first network.



Fine-Tuning Neural Networks Hyperparameters

The Number of Hidden Layers – Deep or Shallow Neural Networks

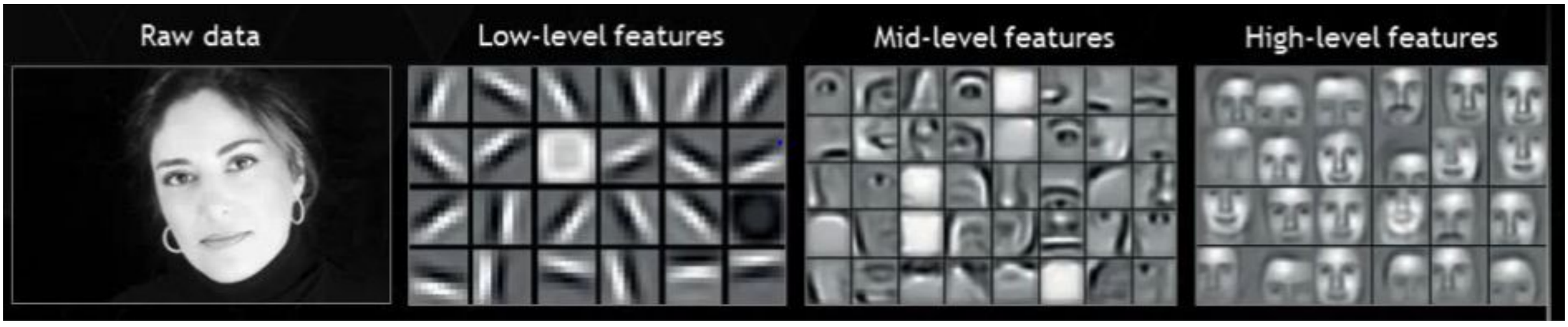
- Instead of randomly initializing the weights and biases of the first few layers of the new neural network, you can initialize them to the value of the weights and biases of the lower layers of the first network.



Fine-Tuning Neural Networks Hyperparameters

The Number of Hidden Layers – Deep or Shallow Neural Networks

- This way the network will not have to learn from scratch all the low-level structures that occur in most pictures; it will only have to learn the higher-level structures (e.g., hairstyles).



The Number of Hidden Layers – Deep or Shallow Neural Networks

Conclusion :

- For many problems we can start with just one or two hidden layers and it will work just fine
  - E.g., we can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons,
  - And above 98% accuracy using two hidden layers with the same total amount of neurons, in roughly the same amount of training time

The Number of Hidden Layers – Deep or Shallow Neural Networks

Conclusion :

- For more complex problems, we can gradually increase the number of hidden layers
- Until we start overfitting the training set.
- Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers or even hundreds and they need a huge amount of training data

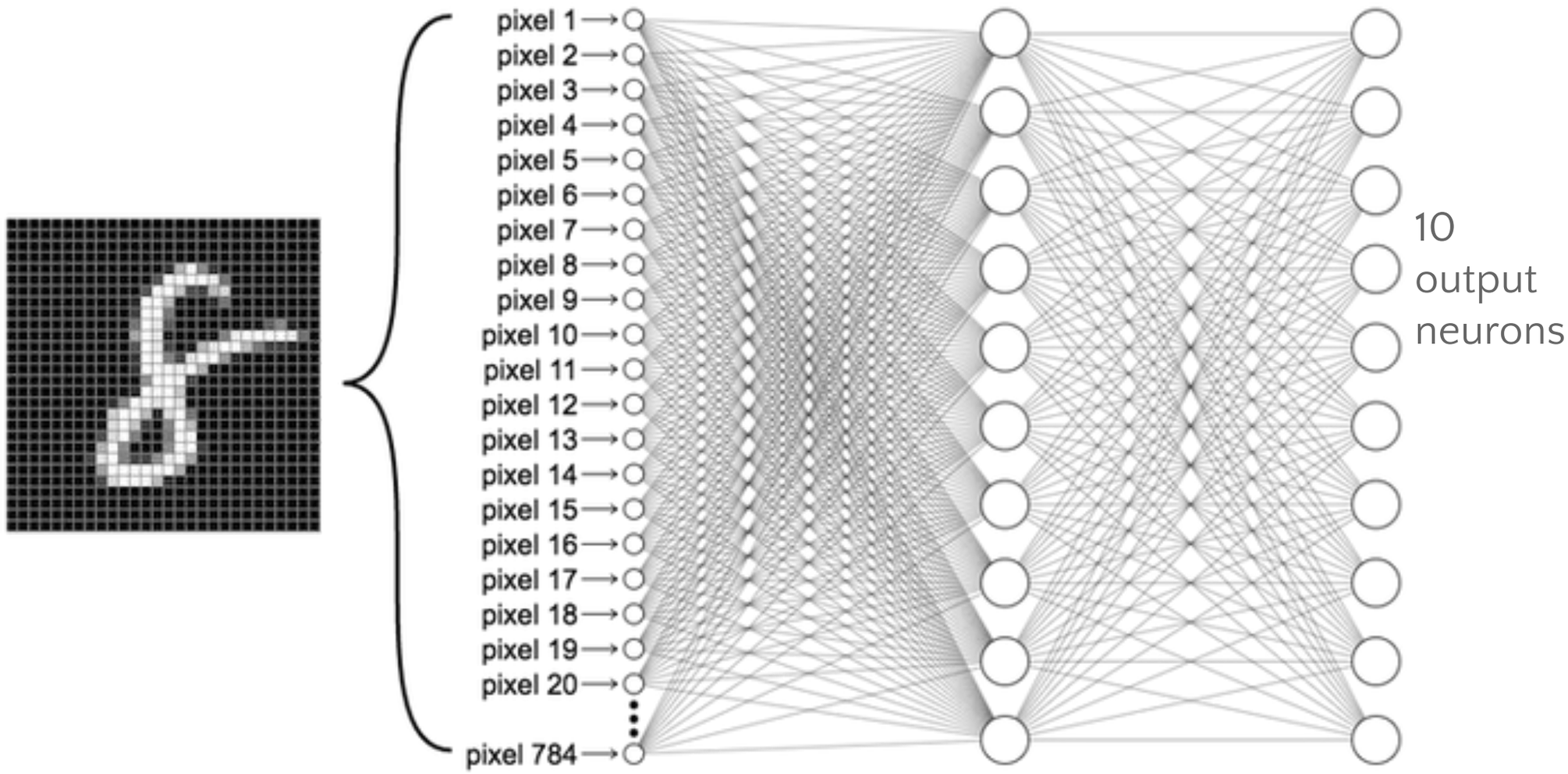
The Number of Neurons per Hidden Layer

Now let's see how do we decide the number of Neurons per hidden layer.

- The number of neurons in the input layer is determined by the type of input
- The number of neurons in the output layer is determined by the output our tasks require

The Number of Neurons per Hidden Layer

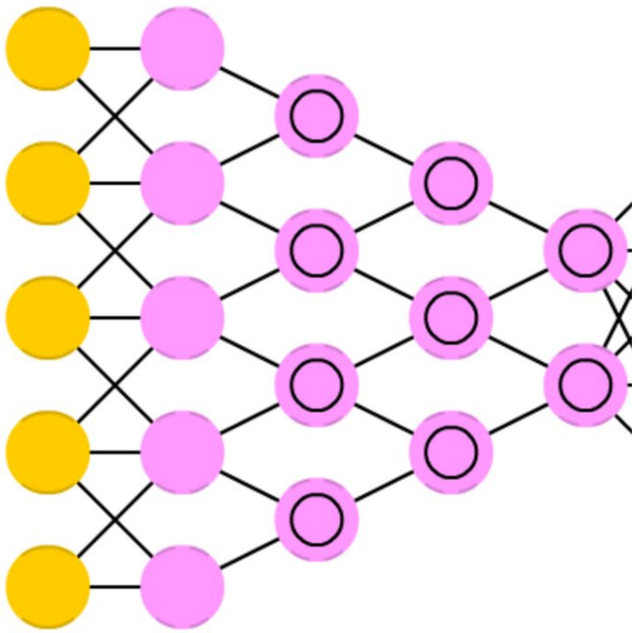
For example, the MNIST task requires 28 x 28 = 784 input neurons and 10 output neurons.



Fine-Tuning Neural Networks Hyperparameters

The Number of Neurons per Hidden Layer

- As for the hidden layers, a common practice is to size them to **form a funnel**, with fewer and fewer neurons at each layer
- The rationale being that many low-level features can coalesce into far fewer high-level features

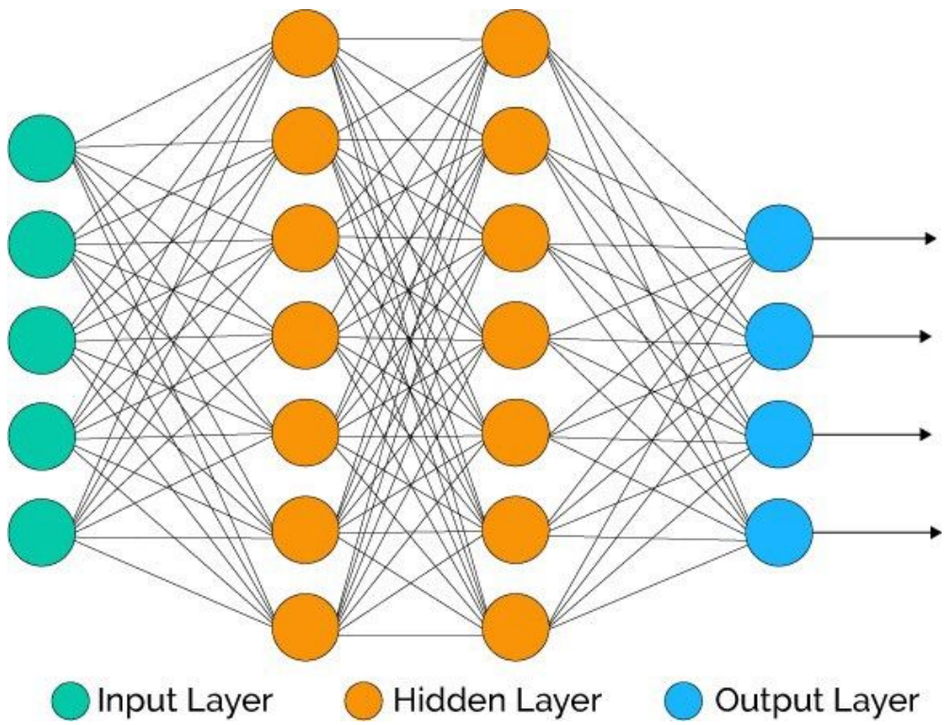


Fewer and fewer neurons at each layer

Fine-Tuning Neural Networks Hyperparameters

The Number of Neurons per Hidden Layer

- For example, a typical neural network for MNIST may have two hidden layers, the first with 300 neurons and the second with 100.
- However, this practice is not as common now, and you may simply use the same size for all hidden layers

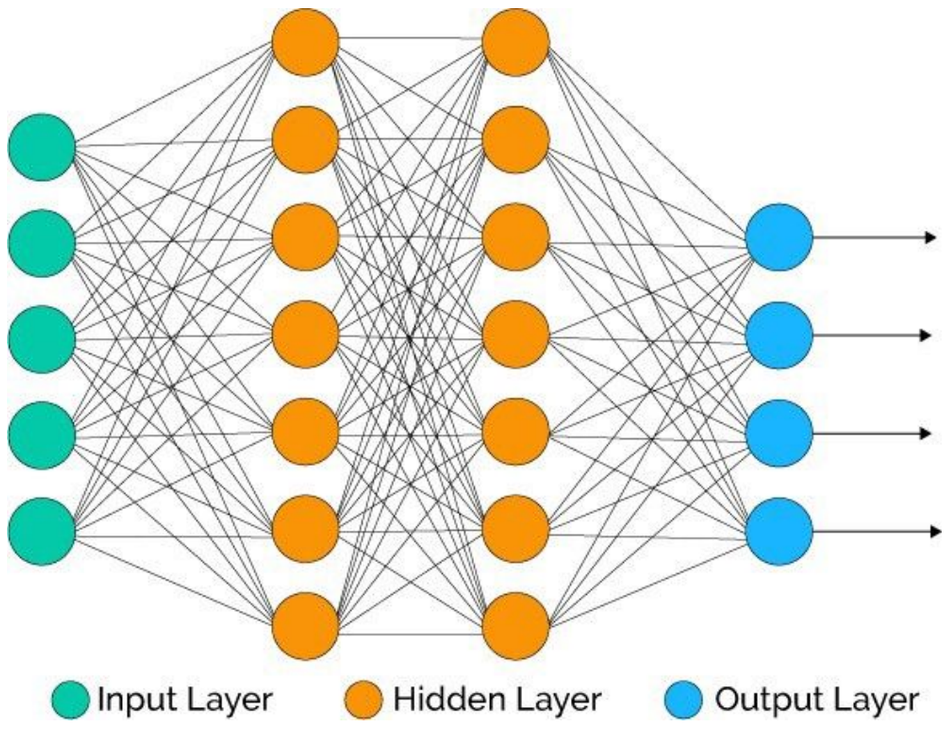


Neural network with same no.of neurons in the hidden layer. We just have to tune one hyperparameter

Fine-Tuning Neural Networks Hyperparameters

The Number of Neurons per Hidden Layer

- For example, all hidden layers with 150 neurons: that's just one hyperparameter to tune instead of one per layer.



Neural network with same no.of neurons in the hidden layer. We just have to tune one hyperparameter

The Number of Neurons per Hidden Layer

- Just like for the number of layers, we can try increasing the number of neurons gradually until the network starts overfitting.
- In general we will get better payoff by increasing the number of layers than the number of neurons per layer.
- **Unfortunately, as you can see, finding the perfect amount of neurons is still somewhat of a black art.**

The Number of Neurons per Hidden Layer

Conclusion :

- A simpler approach is to pick a model with more layers and neurons than you actually need, then use **early stopping** to prevent it from overfitting
- This has been dubbed the *"stretch pants"* approach, instead of wasting time looking for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size

Choosing the Activation Function

For the hidden layers

- In most cases you can use the **ReLU activation function** in the hidden layers or one of its variants
  - It is a bit faster to compute than other activation functions,
  - And Gradient Descent does not get stuck as much on plateaus,
  - It is due to the fact that it does not saturate for large input values
  - Unlike the logistic function or the hyperbolic tangent function, which saturate at 1

Choosing the Activation Function

For the output layer

- For the output layer, the softmax activation function is generally a good choice for classification tasks, when the classes are mutually exclusive
- For regression tasks, you can simply use no activation function at all

Fine-Tuning Neural Networks Hyperparameters

Here are some **Python libraries** you can use to optimize hyperparameters:

- Hyperopt
- Hyperas, kopt, or Talos
- Keras Tuner
- Scikit-Optimize (skopt)
- Spearmint
- Hyperband
- Sklearn-Deap

Fine-Tuning Neural Networks Hyperparameters

Many companies offer services for hyperparameter optimization:

- Google Cloud AI Platform's hyperparameter tuning service
- Arimo
- SigOpt
- CallDesk's Oscar

Questions?

https://discuss.cloudxlab.com

reachus@cloudxlab.com