

# Deep Learning

Prof.Dr. Bahadır AKTUĞ  
Machine Learning with Python

*\*Compiled from sources given in the references.*

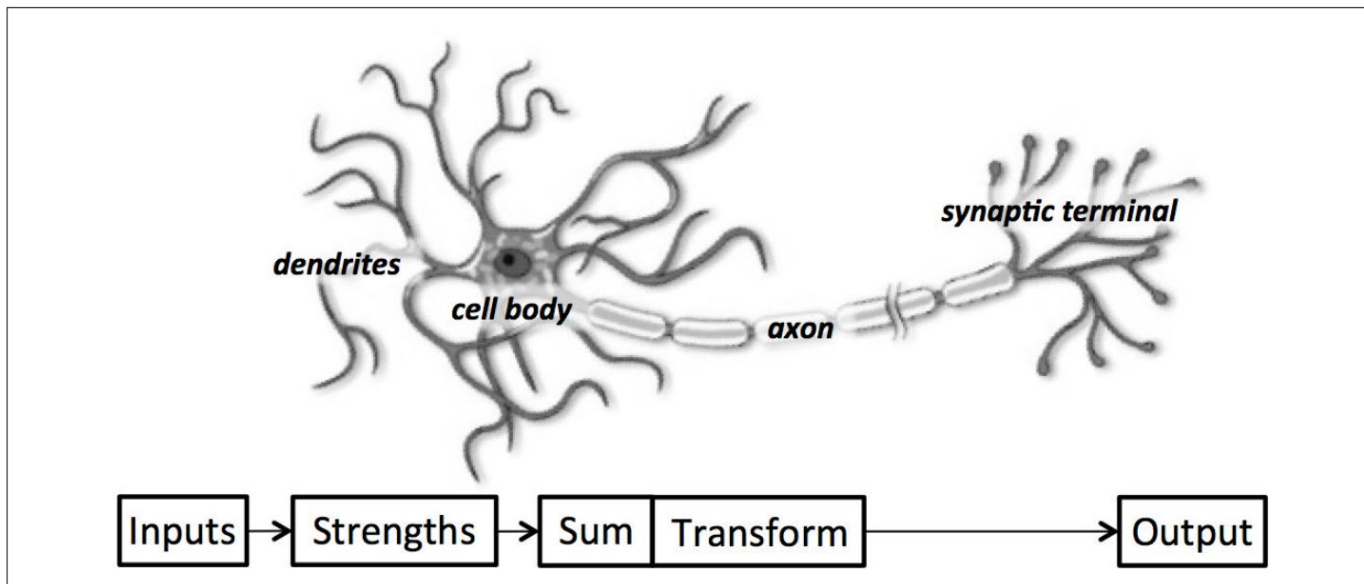
# Deep Learning

---

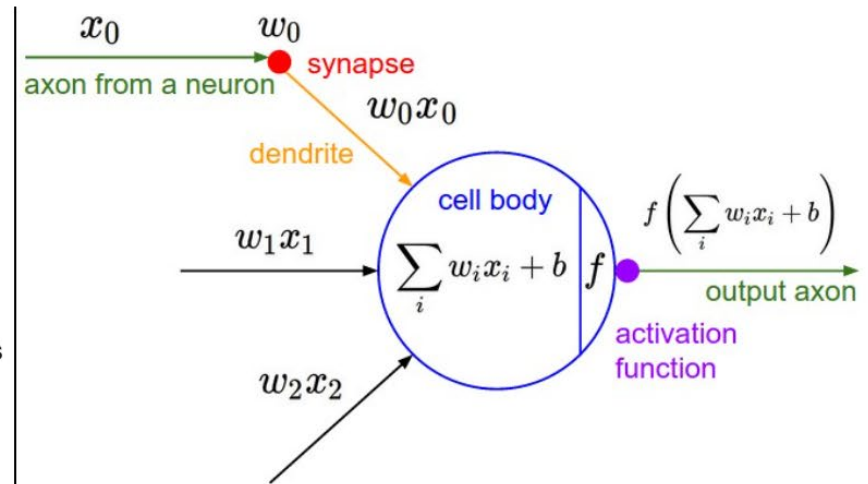
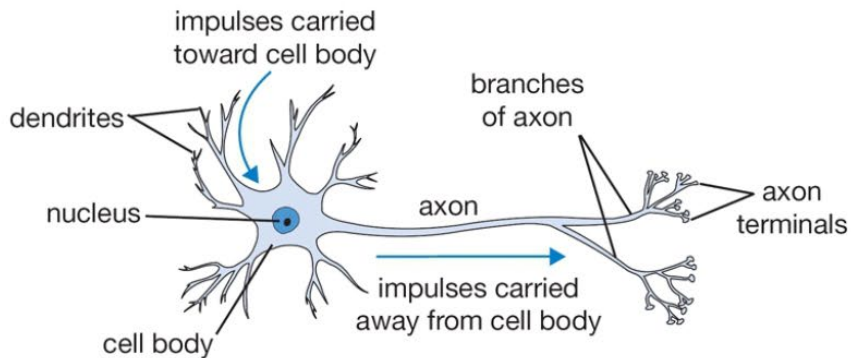
- ▶ Deep learning is a subset of a more general field of artificial intelligence «machine learning».
- ▶ Deep learning is devised in analogy to neurons. The neuron receives its inputs along antenna-like structures called dendrites.
- ▶ Each of these incoming connections is dynamically strengthened or weakened based on how often it is used (this is how we learn new concepts!),
- ▶ It is the strength of each connection that determines the contribution of the input to the neuron's output.

# Neurons

- ▶ After being weighted by the strength of their respective connections, the inputs are summed together in the cell body.
- ▶ This sum is then transformed into a new signal that's propagated along the cell's axon and sent off to other neurons.



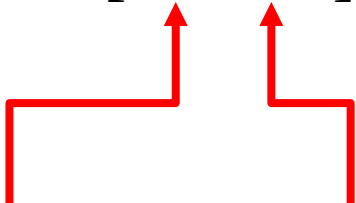
# Neurons Analogy



# Perceptron


- Let's start with an example: Let's say we wanted to determine how to predict exam performance based on the number of hours of sleep we get and the number of hours we study the previous day.

weights  $\longrightarrow \theta = [\theta_0 \ \theta_1 \ \theta_2]^T$

$$\mathbf{x} = [x_1 \ x_2]^T$$


hours of sleep      hours of study

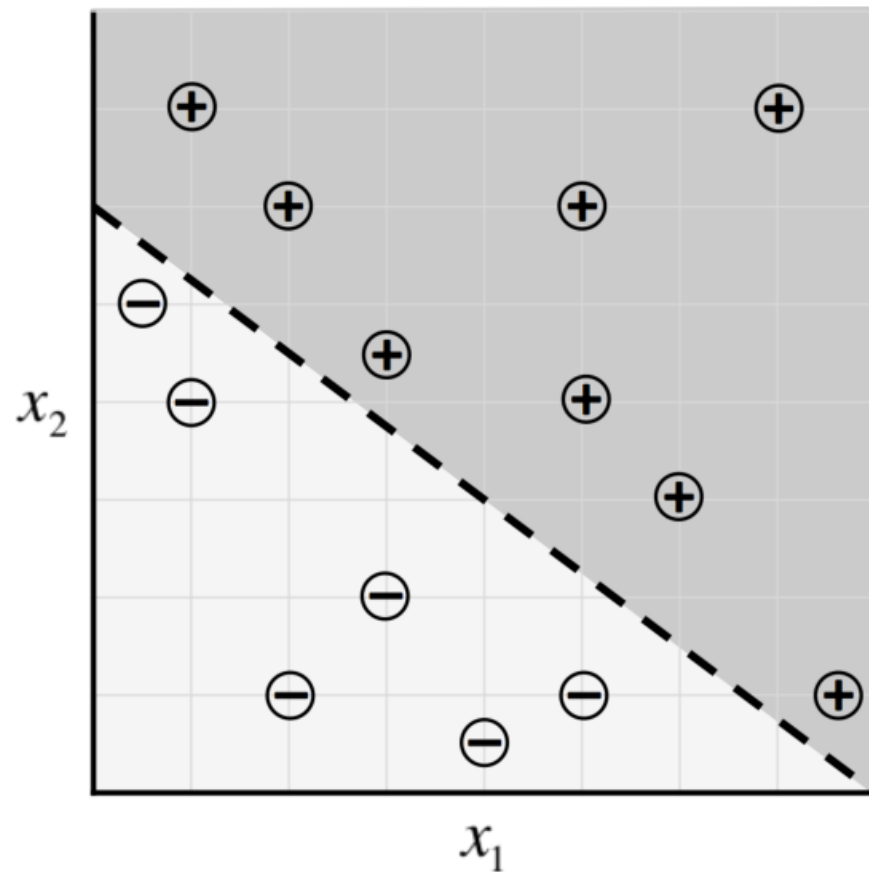
Result: Whether the subject performed above class average or not


$$h(\mathbf{x}, \theta) = \begin{cases} -1 & \text{if } \mathbf{x}^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 < 0 \\ 1 & \text{if } \mathbf{x}^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 \geq 0 \end{cases}$$

# Perceptron

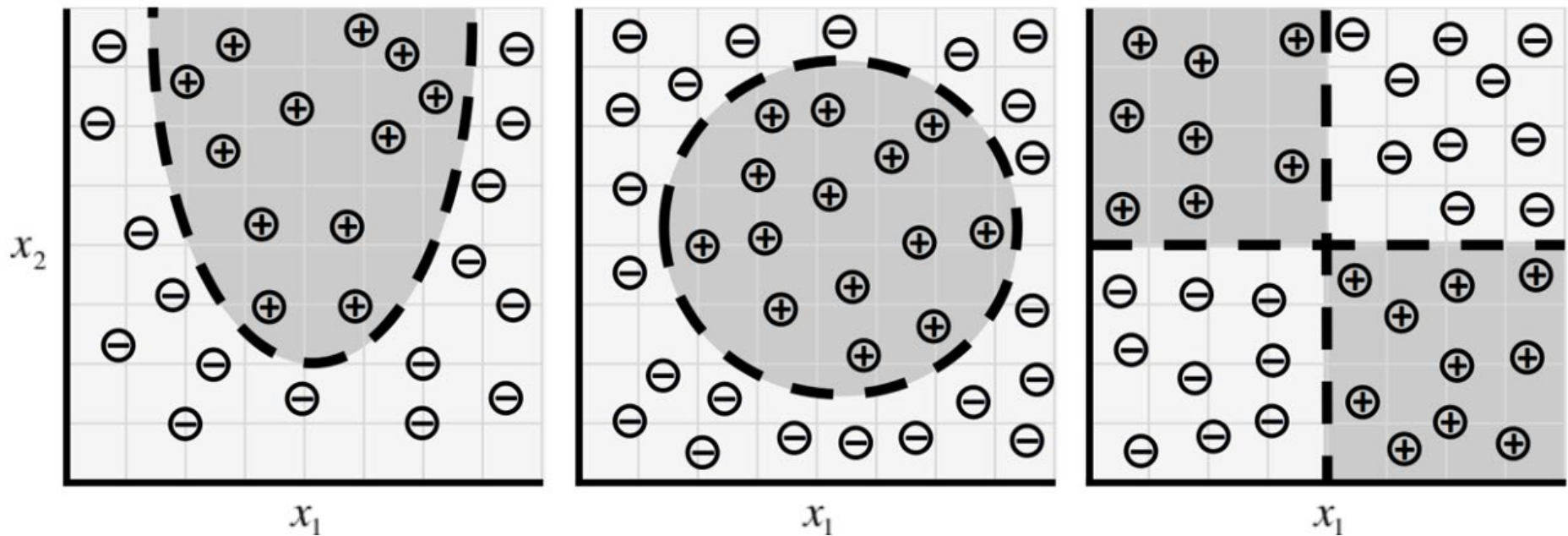
- ▶ Deep learning is a subset of a more general field of artificial intelligence «machine learning»

$$h(\mathbf{x}, \theta) = \begin{cases} -1 & \text{if } \mathbf{x}^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 < 0 \\ 1 & \text{if } \mathbf{x}^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 \geq 0 \end{cases}$$



# Non-Linear

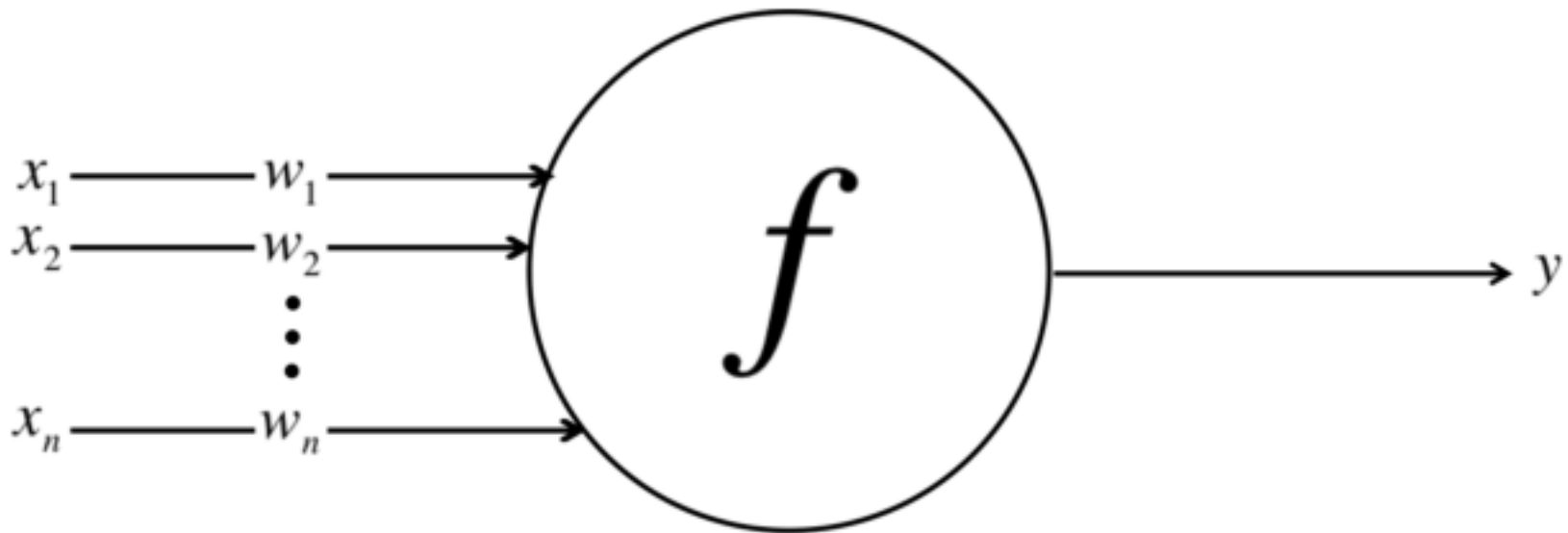
- ▶ The output could be non-linear which could not be handled by simple linear functions.



# The Neuron

---

- ▶ Neuron is more complex and generalized form a perceptron.
- ▶ Neuron could be non-linear and could have multiple outputs.

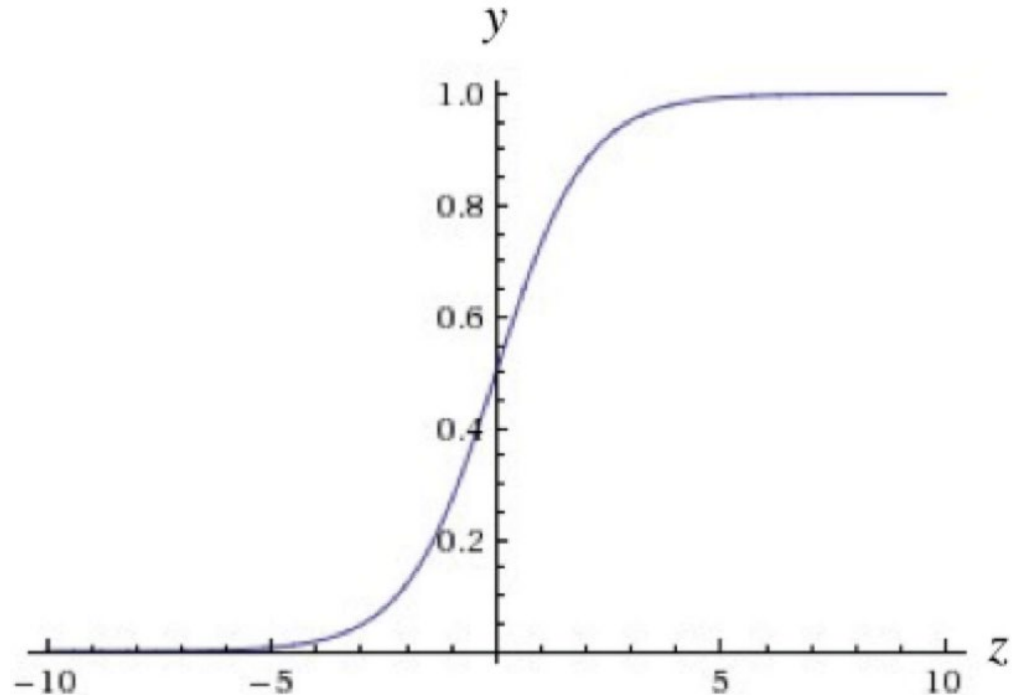




# Sigmoid Function

- ▶ The non-linearity can be dealt with non-linear «activation functions».
- ▶ One popular «activation function» is the «sigmoid».

$$f(z) = \frac{1}{1 + e^{-z}}$$



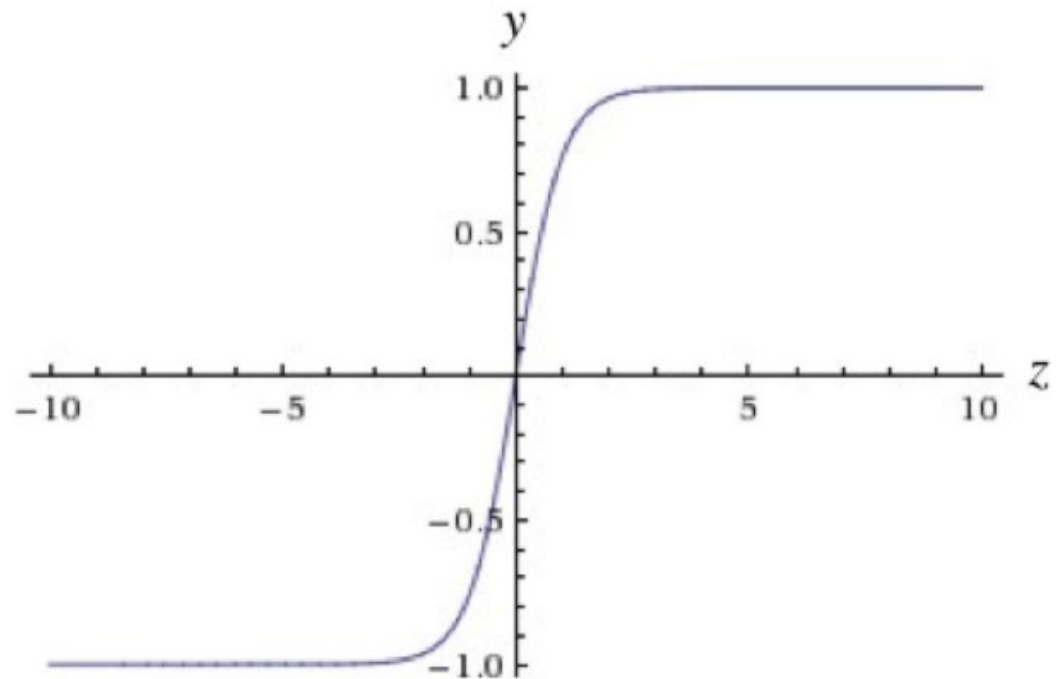
# Tanh

- ▶ Another popular one is the hyperbolic tangent.

$$\tanh(x) = 2 * \text{sigmoid}(2x) - 1$$

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

$$f(z) = \tanh(z)$$

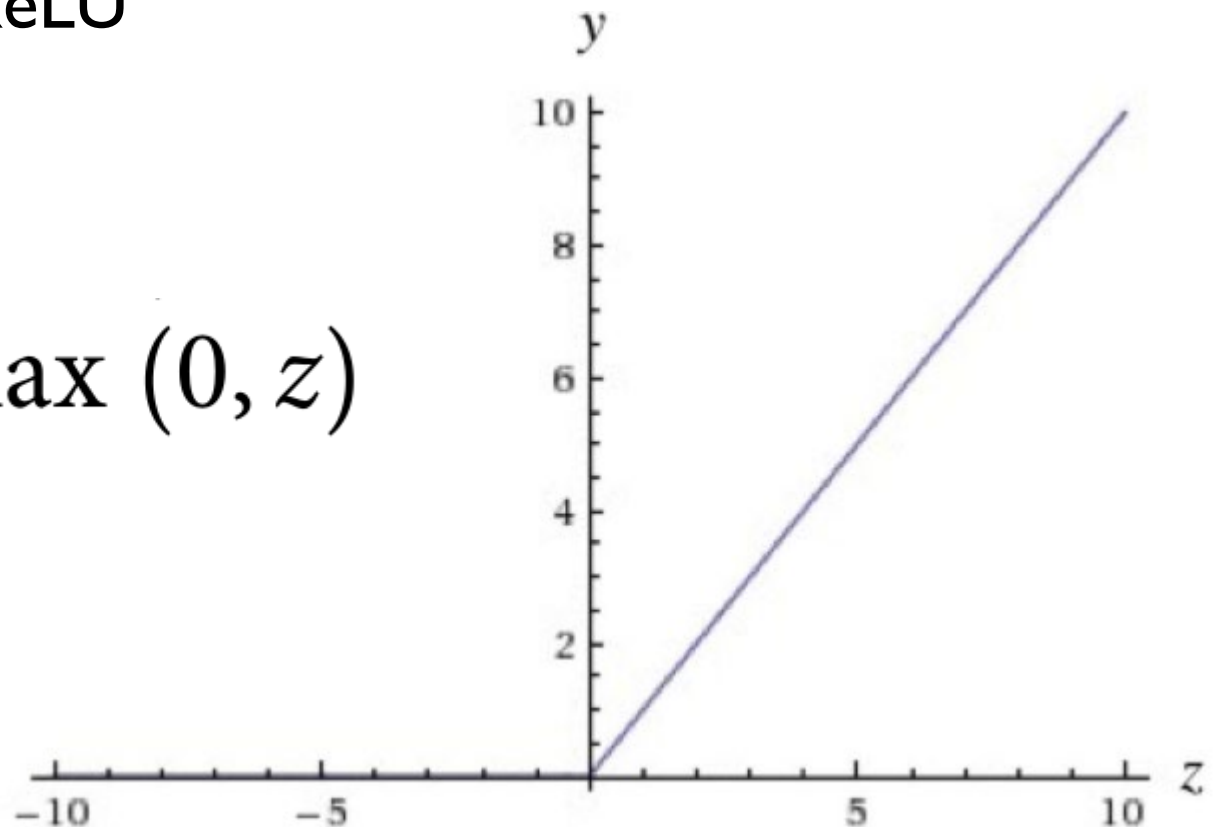


# Restricted linear unit (ReLU)

---

- Probably the most popular one is the ReLU

$$f(z) = \max(0, z)$$



# Softmax

---

- ▶ Sometimes we want our output vector to be a probability distribution over a set of mutually exclusive labels.
- ▶ For example, let's say we want to build a neural network to recognize handwritten digits from the MNIST dataset.
- ▶ Each label (0 through 9) is mutually exclusive, but it's unlikely that we will be able to recognize digits with 100% confidence. Using a probability distribution gives us a better idea of how confident we are in our predictions.

$$[p_0 \ p_1 \ p_2 \ p_3 \ \dots \ p_9] \quad \sum_{i=0}^9 p_i = 1$$

# Softmax

---

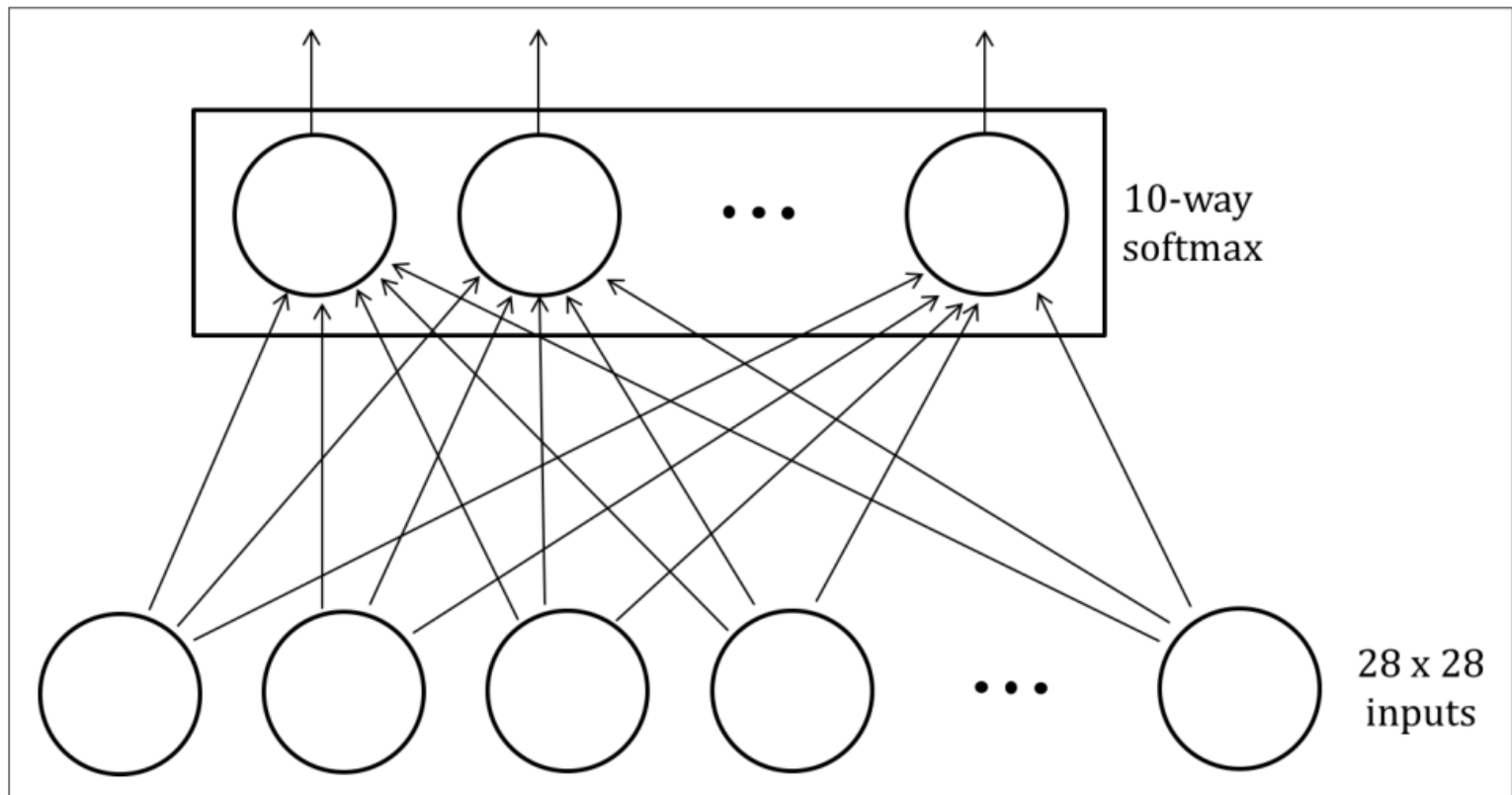
- ▶ Unlike in other kinds of layers, the output of a neuron in a softmax layer depends on the outputs of all the other neurons in its layer.
- ▶ This is because we require the sum of all the outputs to be equal to 1. This is achieved by normalizing the output such that:

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

- ▶ A strong prediction would have a single entry in the vector close to 1, while the remaining entries were close to 0. A weak prediction would have multiple possible labels that are more or less equally likely.

# Softmax

$$P(y = i | x) = \text{softmax}_i(Wx + b) = \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}}$$



# Choice of Activation/Loss functions

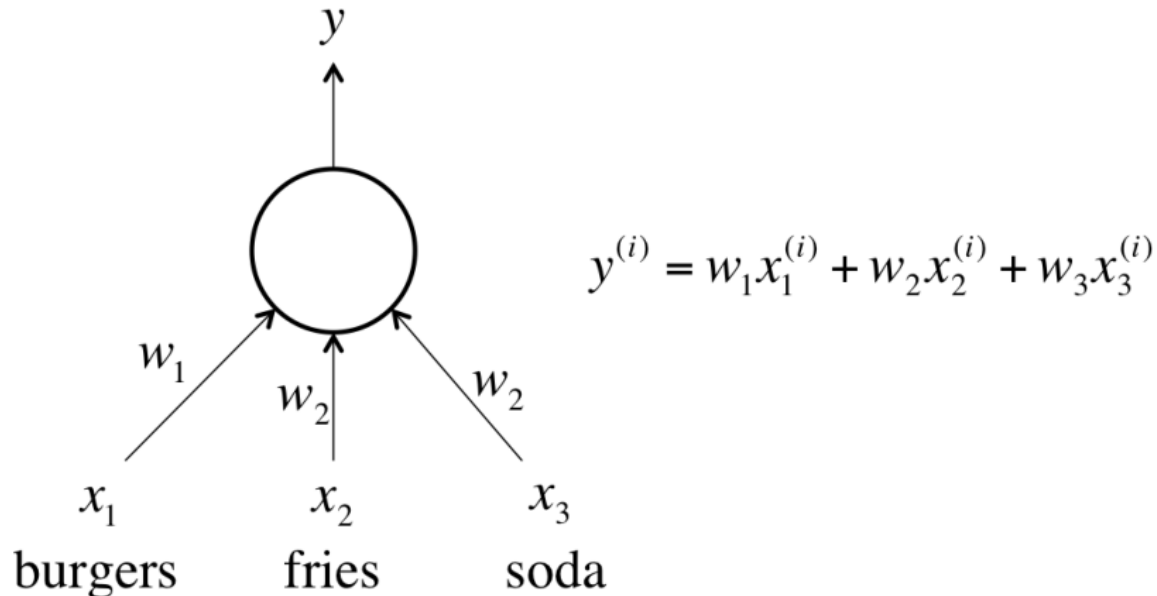
- ▶ There are many options for activation functions and loss functions.
- ▶ Here are some basic recommendations for choosing them with respect to the problem types:

Problem type	Last-layer activation	Loss function
Binary classification	<code>sigmoid</code>	<code>binary_crossentropy</code>
Multiclass, single-label classification	<code>softmax</code>	<code>categorical_crossentropy</code>
Multiclass, multilabel classification	<code>sigmoid</code>	<code>binary_crossentropy</code>
Regression to arbitrary values	None	<code>mse</code>

# Feed Forward Network

---

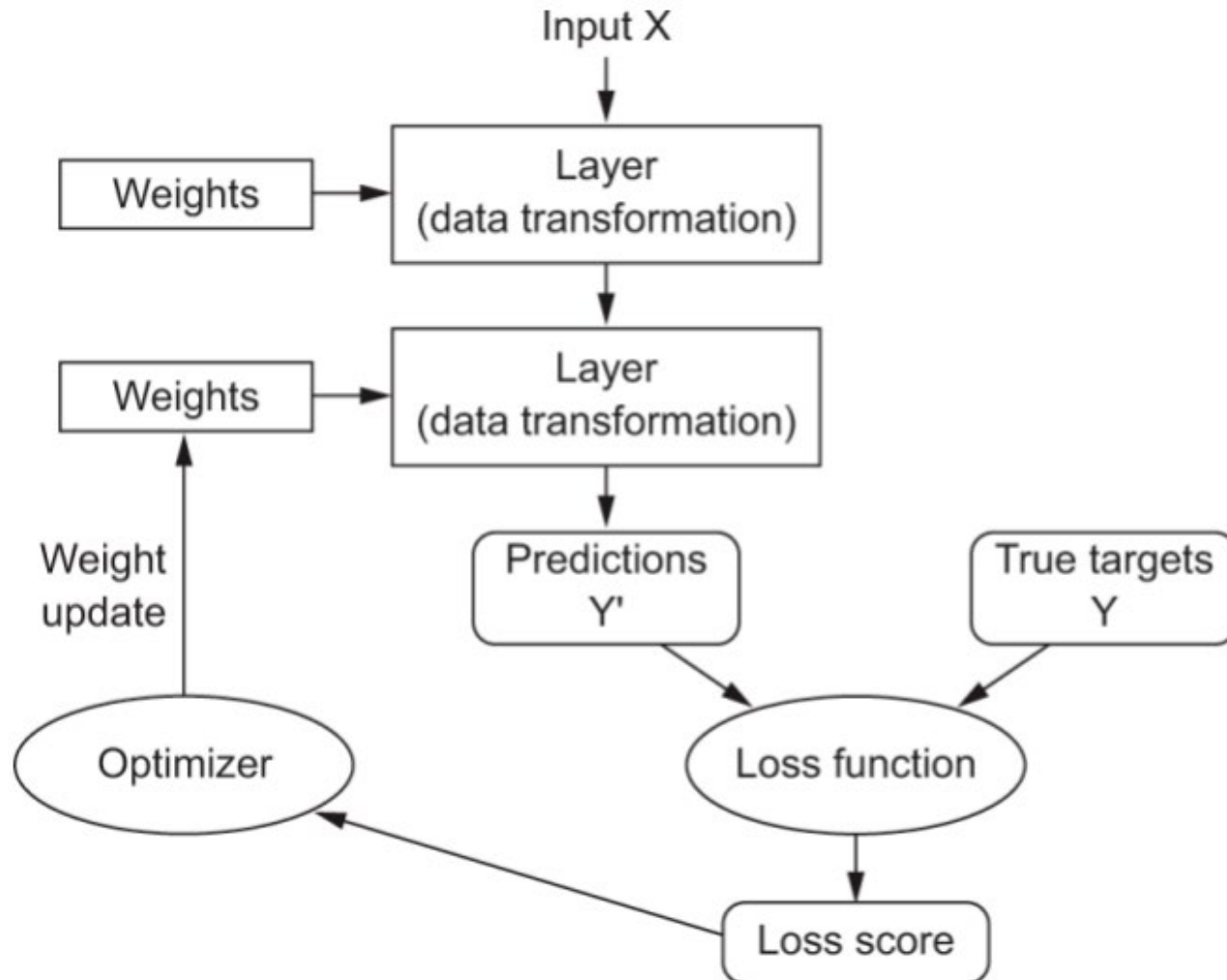
- ▶ The fundamental question in «Deep Learning» is how exactly do we figure out what the weights should be chosen?
- ▶ Find the optimal set of weights is called «training»





# Loss function

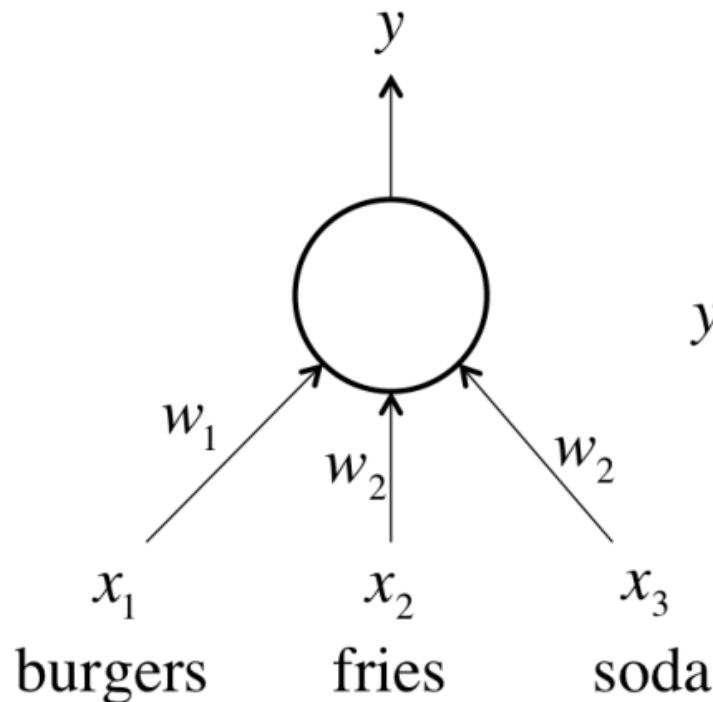
---



# Loss function

---

$$E = \frac{1}{2} \sum_i \left( t^{(i)} - y^{(i)} \right)^2$$



$$y^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + w_3 x_3^{(i)}$$

# Learning

---

- ▶ In «Deep Learning», learning is basically adjusting the network weights with respect to the loss function.
- ▶ Suppose we have the following loss function:

$$E = \frac{1}{2} \sum_i \left( t^{(i)} - y^{(i)} \right)^2$$

- ▶ In order to adjust the weights iteratively, the gradient of the loss function with respect to the weights is computed. This is called «gradient descent».

$$\Delta w_k = - \epsilon \frac{\partial E}{\partial w_k}$$

# Learning

$$E = \frac{1}{2} \sum_i (t^{(i)} - y^{(i)})^2$$

$$y^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + w_3 x_3^{(i)}$$

$$\begin{aligned} \Delta w_k &= -\epsilon \frac{\partial E}{\partial w_k} \\ &= -\epsilon \frac{\partial}{\partial w_k} \left( \frac{1}{2} \sum_i (t^{(i)} - y^{(i)})^2 \right) \end{aligned}$$

$$= \sum_i \epsilon (t^{(i)} - y^{(i)}) \frac{\partial y_i}{\partial w_k}$$

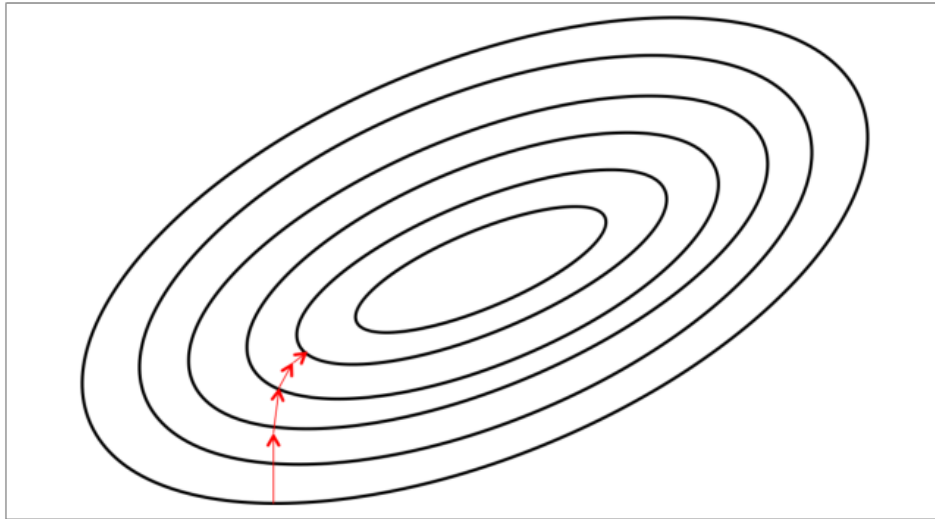
$$= \sum_i \epsilon x_k^{(i)} (t^{(i)} - y^{(i)})$$

$$\Delta w_k = -\epsilon \frac{\partial E}{\partial w_k}$$

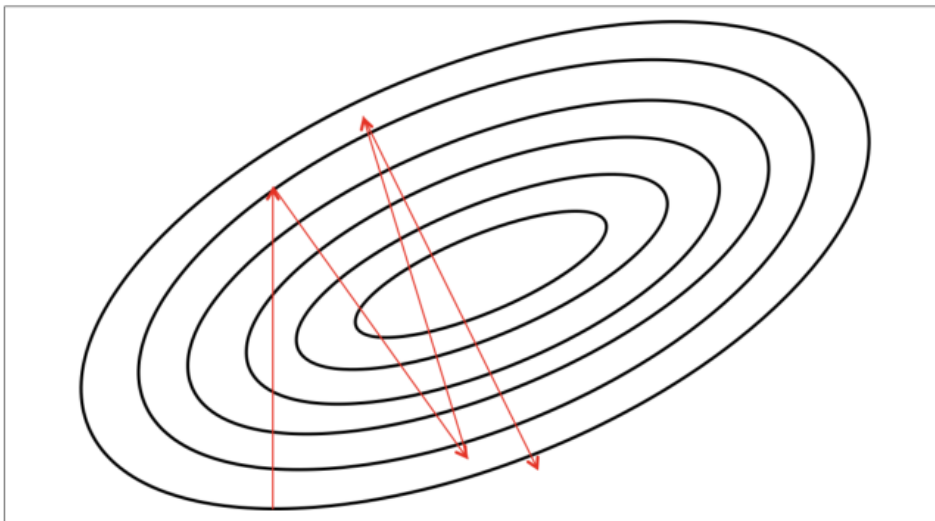
Learning Rate

# Learning

---



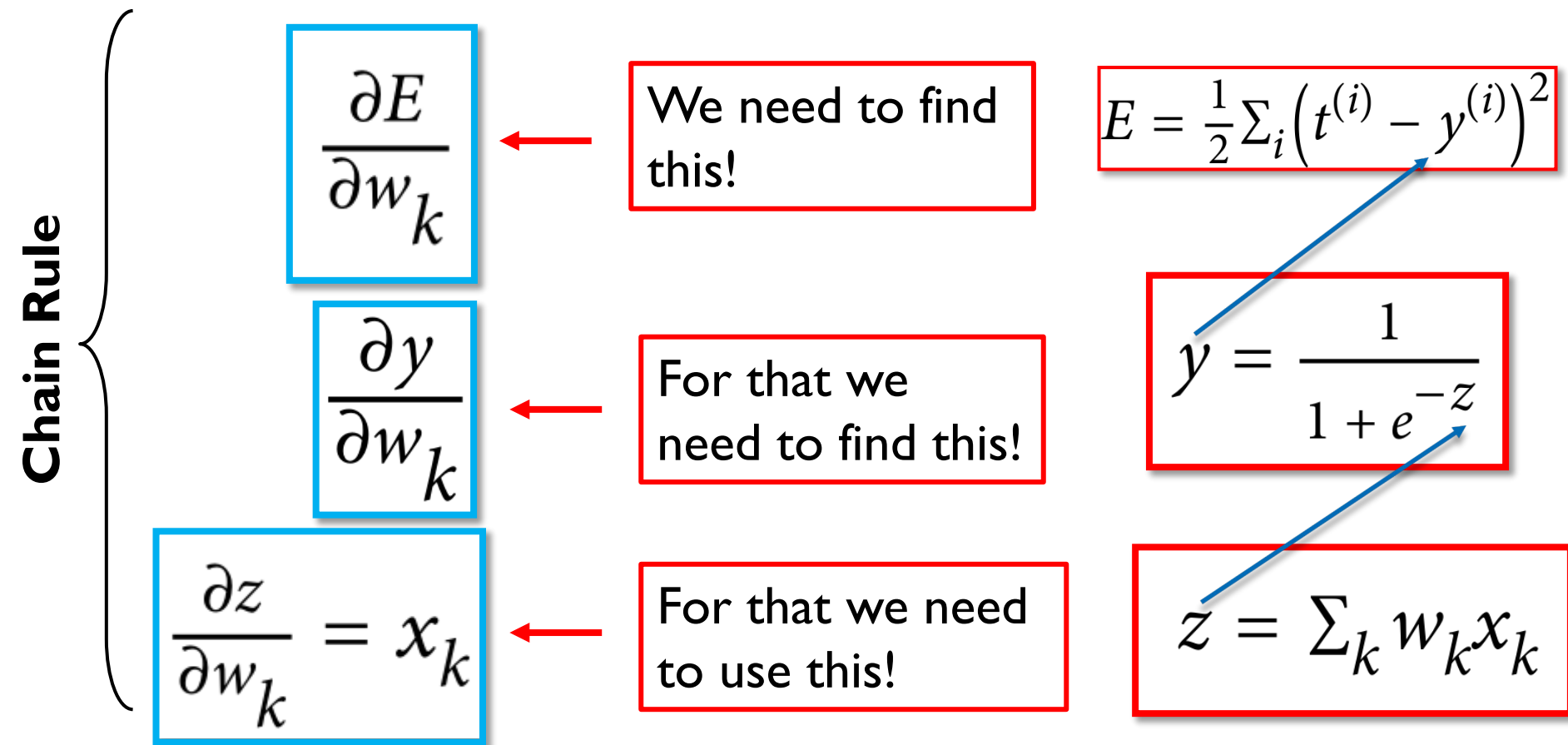
Normal Learning Rate



Too large Learning Rate

# Learning

- ▶ Let's compute the delta weights with a sigmoid neuron output:



# Learning

- First, the derivative of **y** with respect to **z**:

$$\begin{aligned}\frac{dy}{dz} &= \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1}{1 + e^{-z}} \frac{e^{-z}}{1 + e^{-z}} \\ &= \frac{1}{1 + e^{-z}} \left( 1 - \frac{1}{1 + e^{-z}} \right) \\ &= y(1 - y)\end{aligned}$$

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

$$\frac{\partial z}{\partial w_k} = x_k$$

$$\frac{\partial y}{\partial w_k} = \frac{dy}{dz} \frac{\partial z}{\partial w_k} = x_k y(1 - y)$$

# Learning

- ▶ Secondly, the derivative of  $E$  with respect to  $w_i$  can be found by using the chain rule:

$$E = \frac{1}{2} \sum_i \left( t^{(i)} - y^{(i)} \right)^2$$

$$\frac{\partial y}{\partial w_k} = \frac{dy}{dz} \frac{\partial z}{\partial w_k} = x_k y(1 - y)$$

$$\frac{\partial E}{\partial w_k} = \sum_i \frac{\partial E}{\partial y^{(i)}} \frac{\partial y^{(i)}}{\partial w_k} = - \sum_i x_k^{(i)} y^{(i)} (1 - y^{(i)}) (t^{(i)} - y^{(i)})$$



# Learning

- ▶ The final step is the update of the network weights:

$$\frac{\partial E}{\partial w_k} = \sum_i \frac{\partial E}{\partial y^{(i)}} \frac{\partial y^{(i)}}{\partial w_k} = - \sum_i x_k^{(i)} y^{(i)} (1 - y^{(i)}) (t^{(i)} - y^{(i)})$$

$$\Delta w_k = - \epsilon \frac{\partial E}{\partial w_k}$$

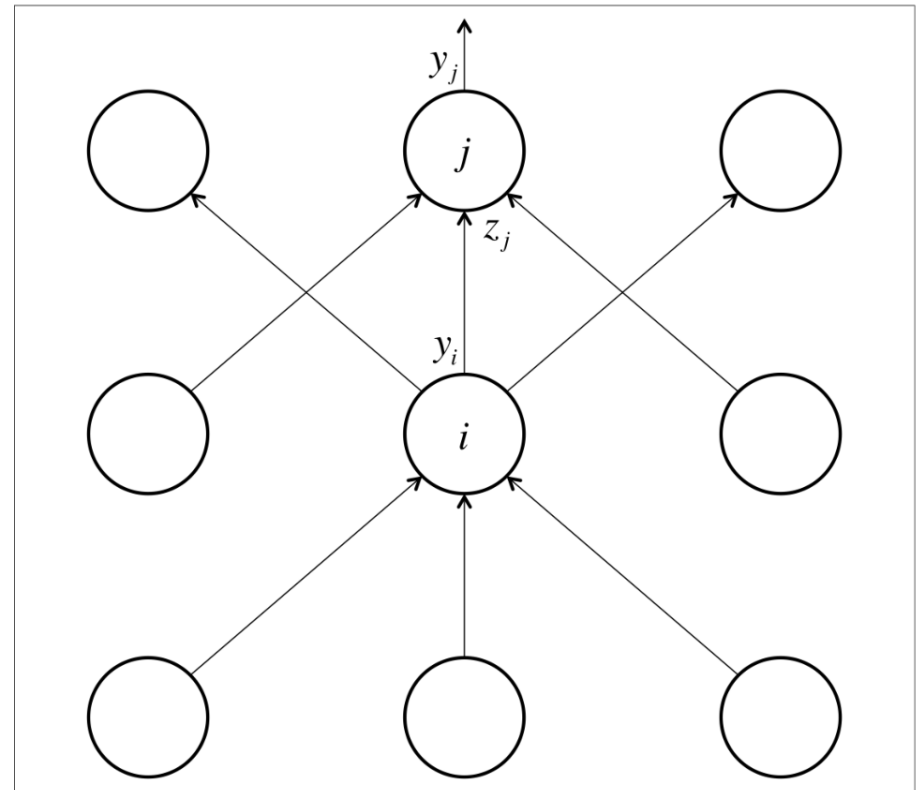


$$\Delta w_k = \sum_i \epsilon x_k^{(i)} y^{(i)} (1 - y^{(i)}) (t^{(i)} - y^{(i)})$$

# Backpropagation

- ▶ Backpropagation is the generalized algorithm for updating the weights iteratively.

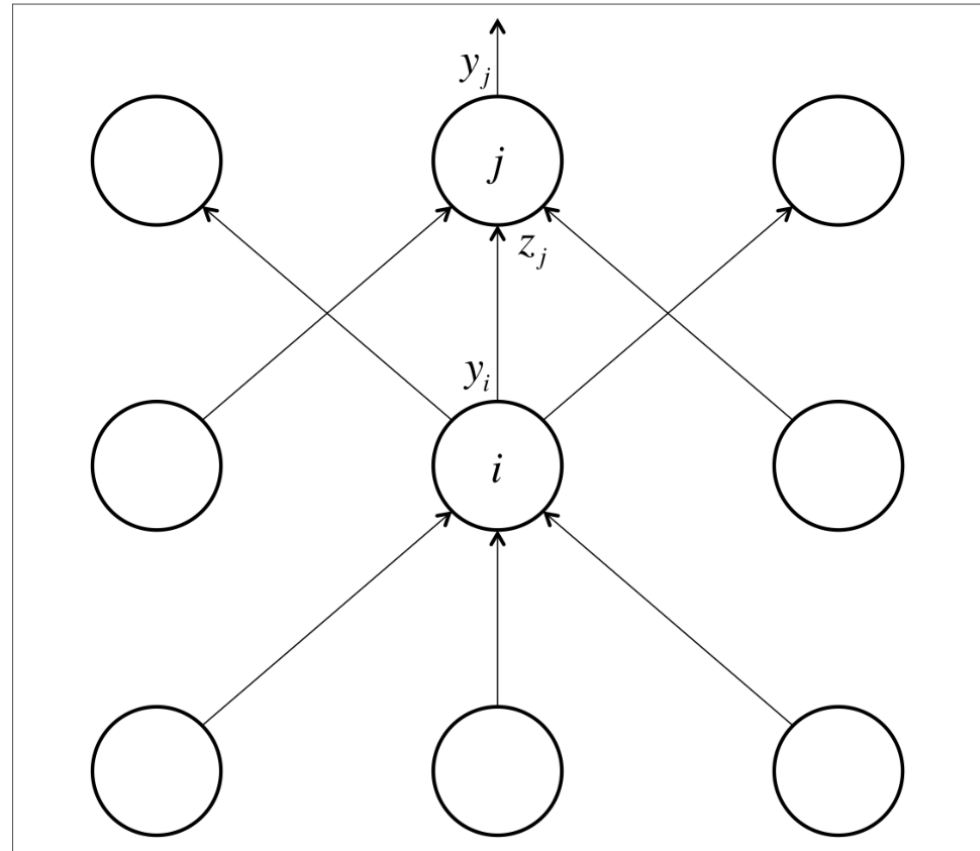
$$E = \frac{1}{2} \sum_{j \in \text{output}} (t_j - y_j)^2$$



# Backpropagation

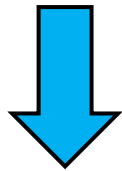
- But this time, we apply chain rule between the layers

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial z_j} \frac{dz_j}{dy_i} = \sum_j w_{ij} \frac{\partial E}{\partial z_j}$$

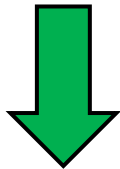


# Backpropagation

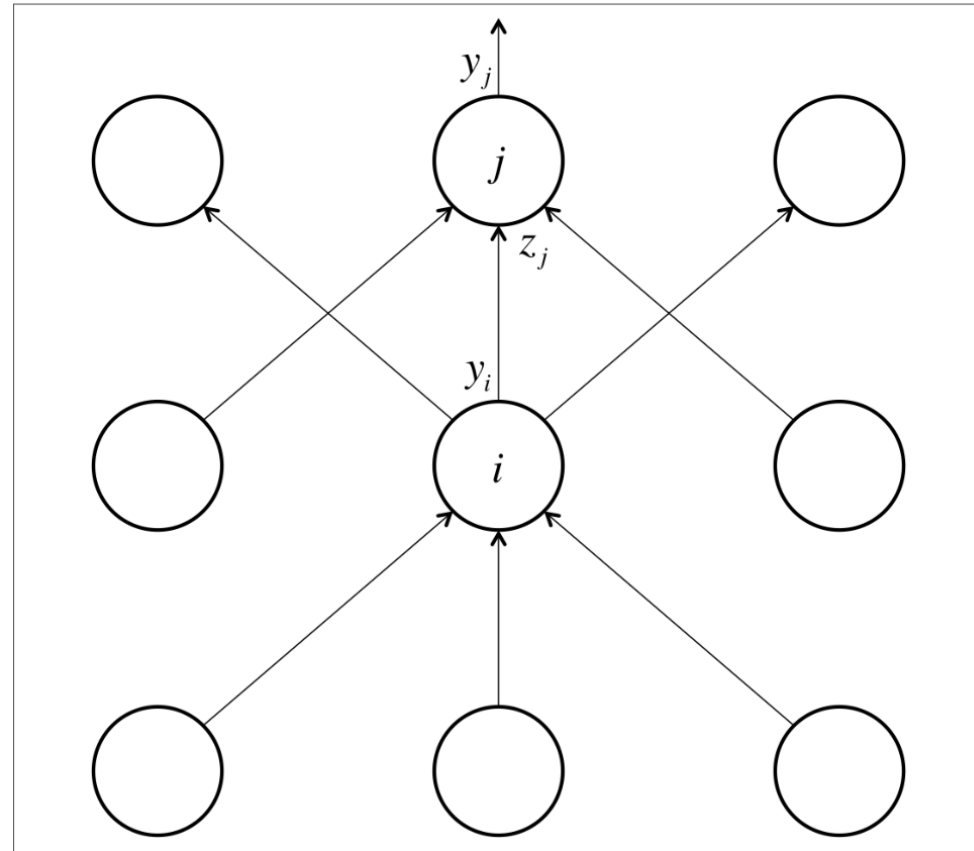
$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial z_j} \frac{dz_j}{dy_i} = \sum_j w_{ij} \frac{\partial E}{\partial z_j}$$



$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j} \frac{dy_j}{dz_j} = y_j(1 - y_j) \frac{\partial E}{\partial y_j}$$



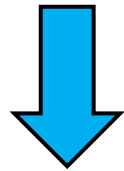
$$\frac{\partial E}{\partial y_i} = \sum_j w_{ij} y_j(1 - y_j) \frac{\partial E}{\partial y_j}$$



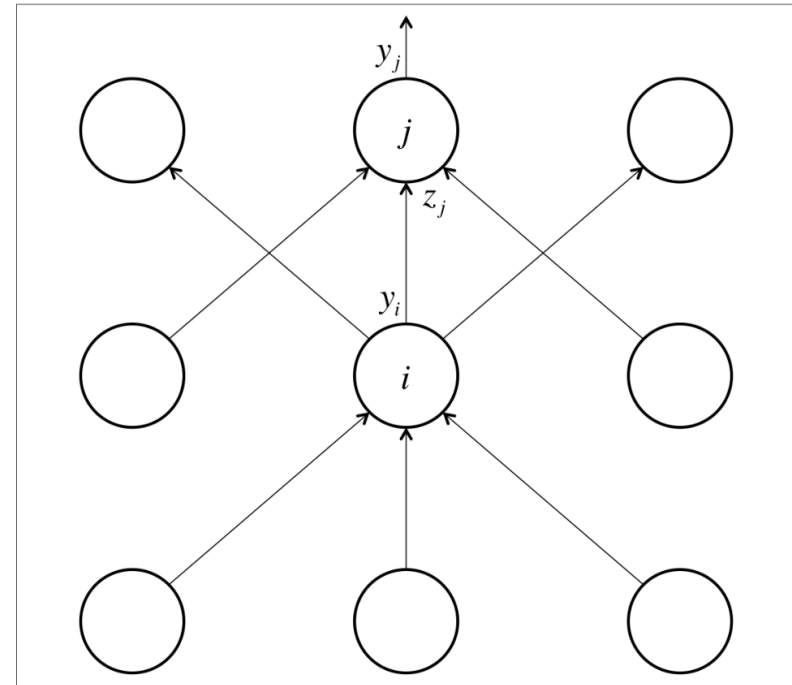
# Backpropagation

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = y_i y_j (1 - y_j) \frac{\partial E}{\partial y_j}$$

$$\Delta w_k = -\epsilon \frac{\partial E}{\partial w_k}$$



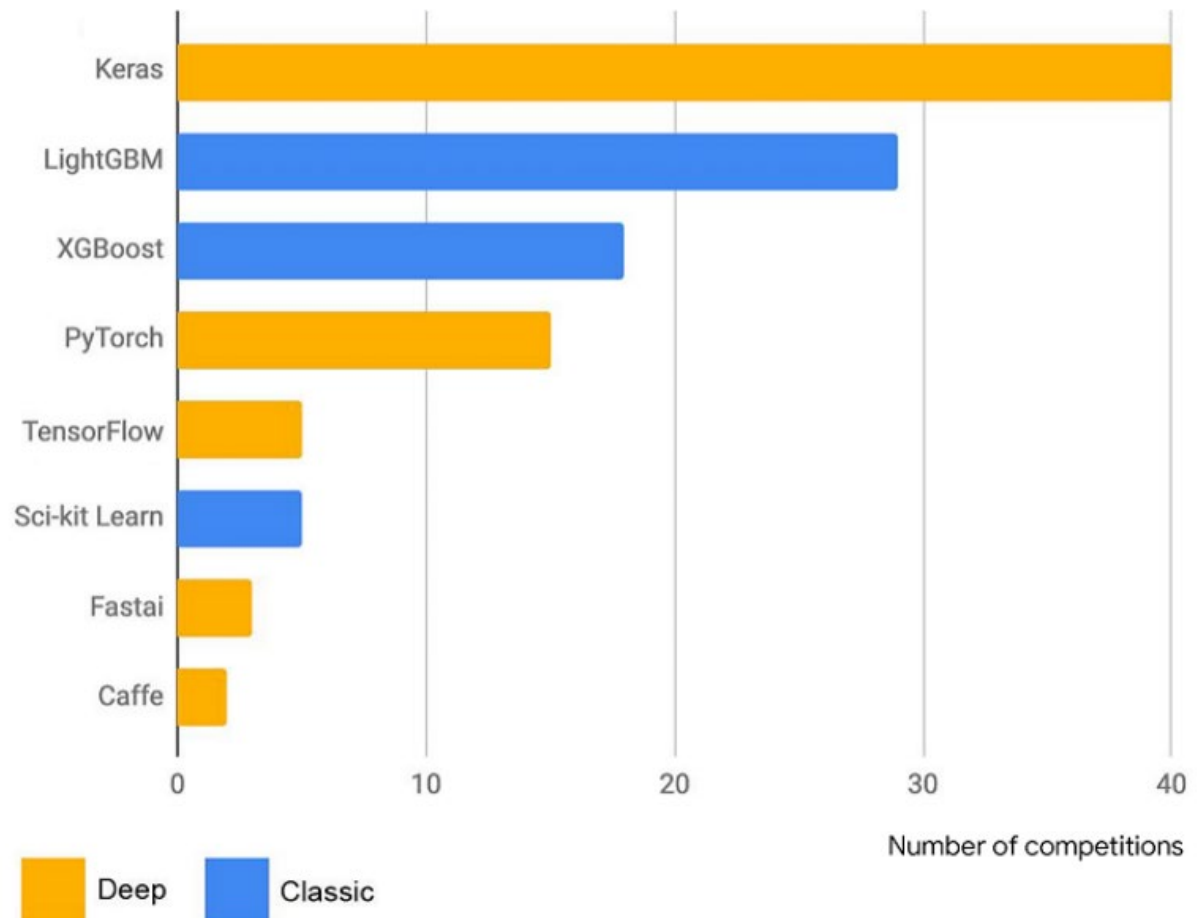
$$\Delta w_{ij} = -\sum_{k \in \text{dataset}} \epsilon y_i^{(k)} y_j^{(k)} (1 - y_j^{(k)}) \frac{\partial E^{(k)}}{\partial y_j^{(k)}}$$



# Deep Learning Frameworks

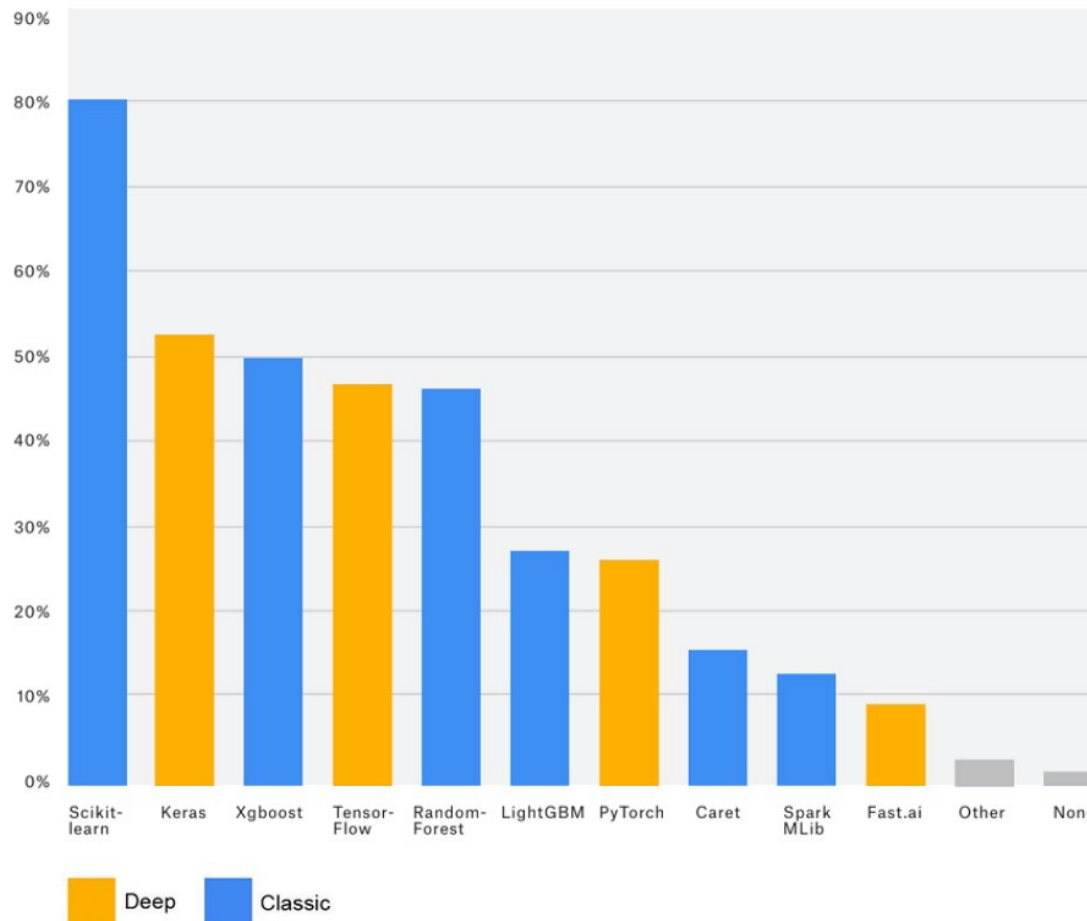
- ▶ Tensorflow
- ▶ Torch
- ▶ CNTK
- ▶ Theano

Primary ML tool used by top-5 teams in Kaggle competitions,  
2017-2018 (N=120)



# Deep Learning Frameworks

Percentage of machine learning & data science professionals using each ML software framework, 2019



# Data Categories in Deep Learning

---

- ▶ **Vector data:**
  - ▶ rank-2 tensors of shape (samples, features)
- ▶ **Timeseries data or sequence data**
  - ▶ rank-3 tensors of shape (samples, timesteps, features)
- ▶ **Images**
  - ▶ rank-4 tensors of shape
    - ▶ (samples, height, width, channels)
    - ▶ (samples, channels, height, width)
- ▶ **Video**
  - ▶ rank-5 tensors of shape
    - ▶ (samples, frames, height, width, channels)
    - ▶ (samples, frames, channels, height, width)



# Data Categories in Deep Learning

---

## ▶ Vector Data

- ▶ A dataset in which age, ZIP code, and income takes place can be characterized as a vector of 3 values per person, and thus an entire dataset of 100,000 people can be stored in a rank-2 tensor of shape (100,000,3)
- ▶ A dataset of text documents, where we represent each document by the counts of how many times each word appears in it (out of a dictionary of 20,000 common words). Each document can be encoded as a vector of 20,000 values (one count per word in the dictionary), and thus an entire dataset of 500 documents can be stored in a tensor of shape (500, 20000).

# Data Categories in Deep Learning

---

## ▶ Sequence Data

- ▶ A dataset of stock prices. Every minute, we store the current price of the stock, the highest price in the past minute, and the lowest price in the past minute. Thus every minute is encoded as a 3D vector, an entire day of trading is encoded as a matrix of shape  $(250, 390, 3)$  (there are 390 minutes in a trading day), and 250 days' worth of data can be  $(390, 3)$  stored in a rank-3 tensor of shape. Here, each sample would be one  $(250, 390, 3)$  day's worth of data.

# Data Categories in Deep Learning

---

## ► Image Data

- Images typically have three dimensions: height, width, and color depth. Although grayscale images (like our MNIST digits) have only a single color channel and could thus be stored in rank-2 tensors, by convention image tensors are always rank-3, with a one-dimensional color channel for grayscale images. A batch of 128 grayscale images of size  $256 \times 256$  could thus be stored in a tensor of shape, and a batch of 128 color images could be (128, 256, 256, 1) stored in a tensor of shape (128, 256, 256, 3)

# Data Categories in Deep Learning

---

## ▶ Video Data

- ▶ Video data is one of the few types of real-world data for which you'll need rank-5 tensors.
- ▶ A video can be understood as a sequence of frames, each frame being a color image. Because each frame can be stored in a rank-3 tensor, a sequence of frames (height, width, color\_depth) can be stored in a rank-4 tensor, and thus a batch (frames, height, width, color\_depth) of different videos can be stored in a rank-5 tensor of shape (samples, frames, height, width, color\_depth).

# MNIST Fashion Dataset

---

- ▶ This is a dataset of 60,000 28x28 grayscale images of 10 fashion categories, along with a test set of 10,000 images.

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

# MNIST Fashion Dataset



```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

layers = tf.keras.layers
mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

class_names = ['T-shirt/top', 'Trouser', 'Pullover',
                'Dress', 'Coat', 'Sandal', 'Shirt',
                'Sneaker', 'Bag', 'Ankle boot']
```

```
>>> y_train[:50]
array([9, 0, 0, 3, 0, 2, 7, 2, 5, 5, 0, 9, 5, 5, 7, 9, 1, 0, 6, 4, 3, 1,
       4, 8, 4, 3, 0, 2, 4, 4, 5, 3, 6, 6, 0, 8, 5, 2, 1, 6, 6, 7, 9, 5,
       9, 2, 7, 3, 0, 3], dtype=uint8)
```

# MNIST Fashion Dataset

```
>>> x_train[0]
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0, 13, 73,  0,  0,  1,  4,  0,  0,  0,  0,  1,
        1,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0, 36, 136, 127, 62, 54,  0,  0,  0,  1,  3,  4,  0,
        0,  3],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0, 102, 204, 176, 134, 144, 123, 23,  0,  0,  0,  0, 12,
        10,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0, 155, 236, 207, 178, 107, 156, 161, 109, 64, 23, 77, 130,
        72, 15],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,
        69, 207, 223, 218, 216, 216, 163, 127, 121, 122, 146, 141, 88,
        172, 66],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  0,
        200, 232, 232, 233, 229, 223, 223, 215, 213, 164, 127, 123, 196,
        229,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        183, 225, 216, 223, 228, 235, 227, 224, 222, 224, 221, 223, 245,
        173,  0]
```



# MNIST Fashion Dataset

```
# Build the neural network layer-by-layer
model = tf.keras.Sequential()
# Make the input layer one-dimensional
model.add(layers.Flatten())
# Layer has 64 nodes; Uses ReLU
model.add(layers.Dense(64, activation='relu'))
# Layer has 64 nodes; Uses ReLU
model.add(layers.Dense(64, activation='relu'))
# Layer has 64 nodes; Uses Softmax
model.add(layers.Dense(10, activation='softmax'))

# Choose an optimizer and loss function for training:
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train and evaluate the model's accuracy
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test, verbose=2)
```

# MNIST Fashion Dataset

```
Epoch 1/5
1875/1875 [=====] - 5s 2ms/step - loss: 0.5096 - accuracy: 0.8208
Epoch 2/5
1875/1875 [=====] - 4s 2ms/step - loss: 0.3786 - accuracy: 0.8631
Epoch 3/5
1875/1875 [=====] - 4s 2ms/step - loss: 0.3422 - accuracy: 0.8742
Epoch 4/5
1875/1875 [=====] - 4s 2ms/step - loss: 0.3199 - accuracy: 0.8818
Epoch 5/5
1875/1875 [=====] - 4s 2ms/step - loss: 0.3049 - accuracy: 0.8881
313/313 - 1s - loss: 0.3471 - accuracy: 0.8761
```



# Convolutional Networks

---

- ▶ A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other.
- ▶ The pre-processing required in a ConvNet is much lower as compared to other classification algorithms.
- ▶ While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

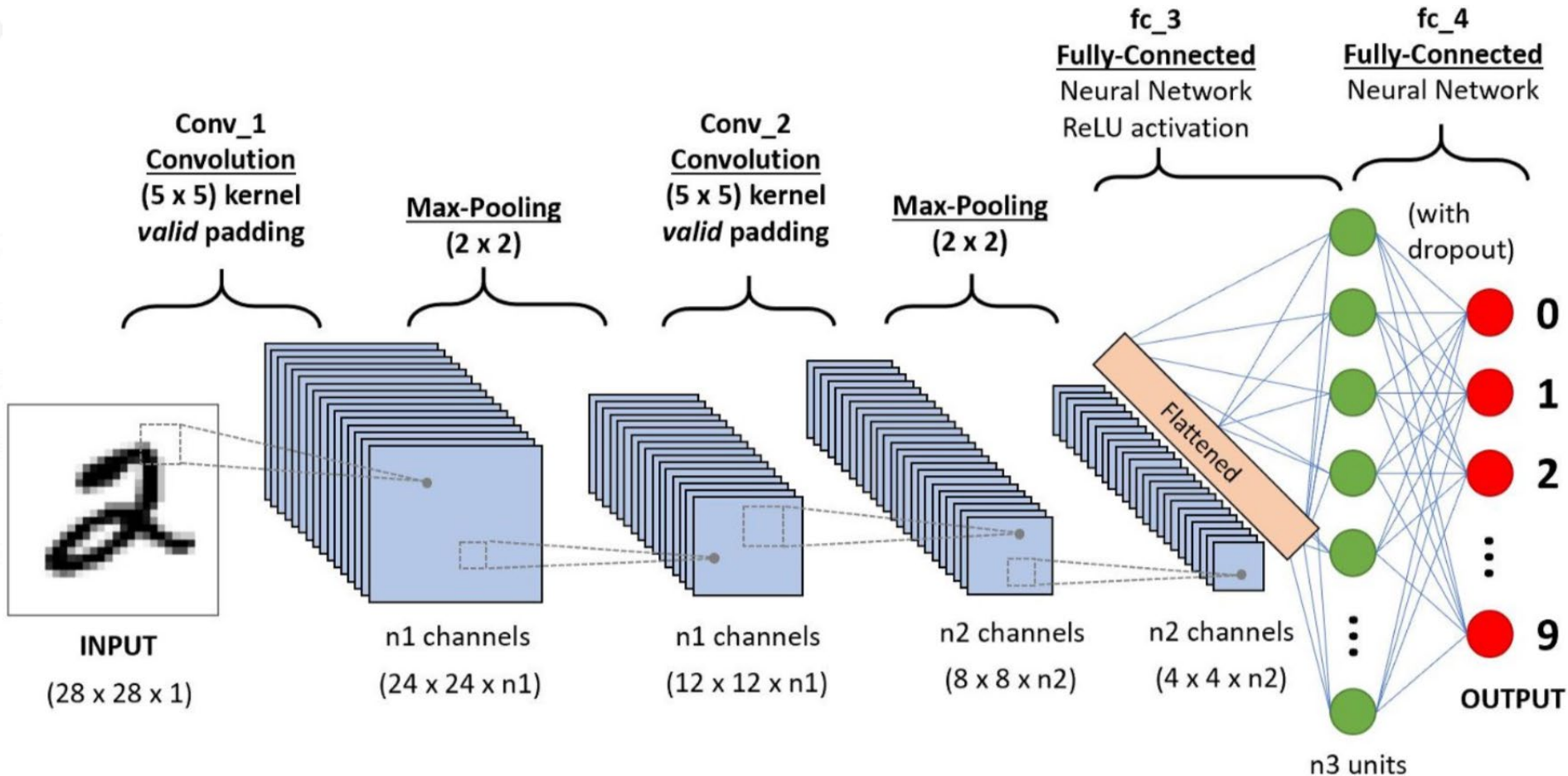
# Convolutional Networks

---

- ▶ The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex.
- ▶ Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field.
- ▶ A collection of such fields overlap to cover the entire visual area.



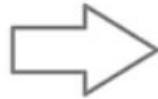
# Convolutional Networks



# Convolutional vs Feed-Forward

---

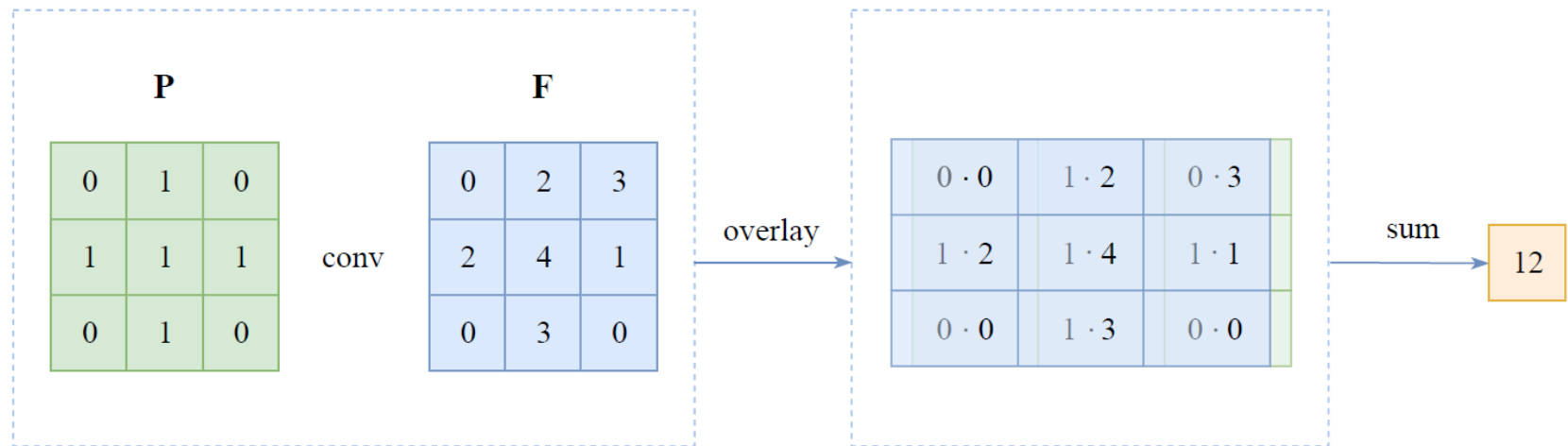
1	1	0
4	2	1
0	2	1



1
1
0
4
2
1
0
2
1

- ▶ Suppose we have an  $3 \times 3$  image.
- ▶ By flattening the image into a  $9 \times 1$  vector, we can train the network as Feed-forward network.
- ▶ However, except for basic monochrome images, feed-forward networks will present little accuracy since it does not account for the dependency on the neighboring pixels.

# Convolution Kernel



A convolution between two matrices.



# Convolution Kernel

1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

Image

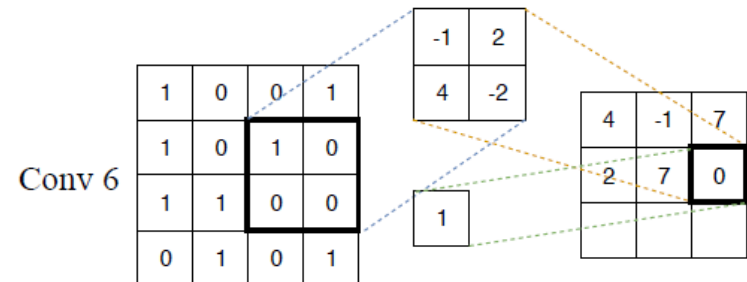
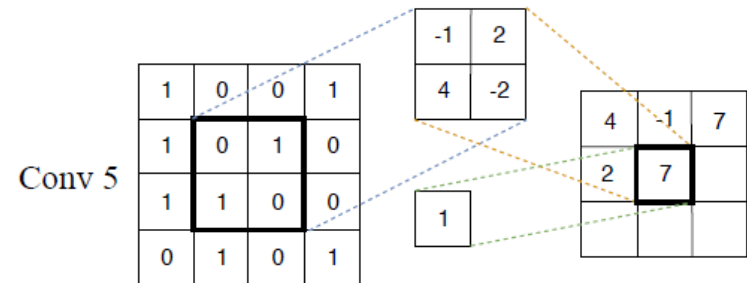
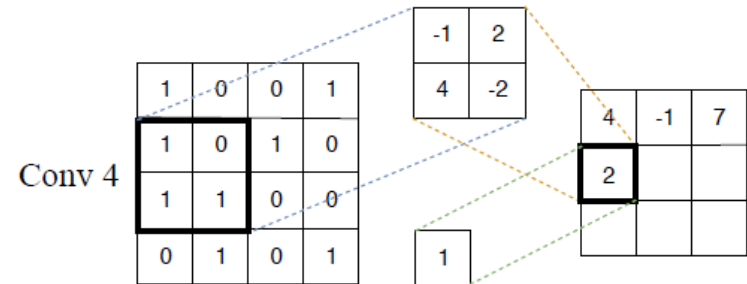
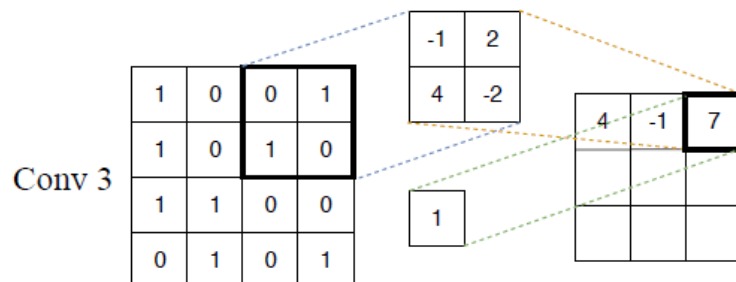
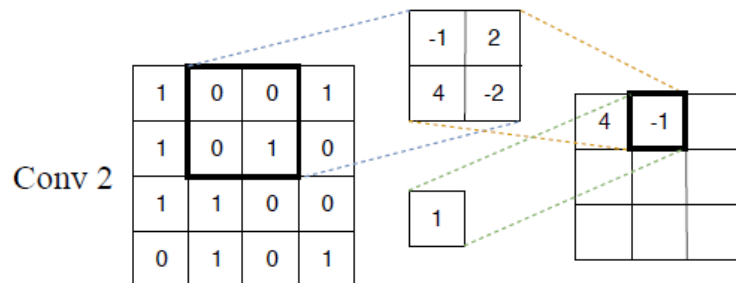
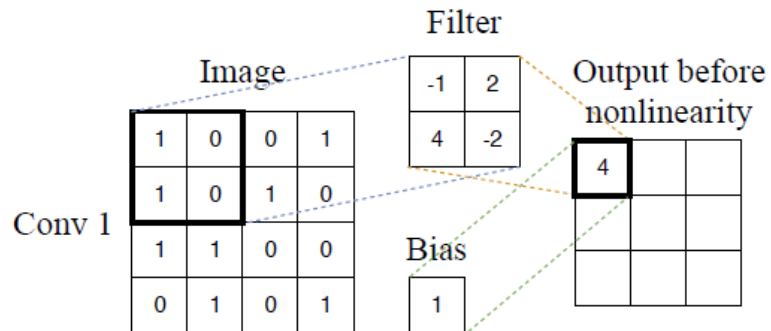
Kernel/Filter, K =

1	0	1
0	1	0
1	0	1

4		

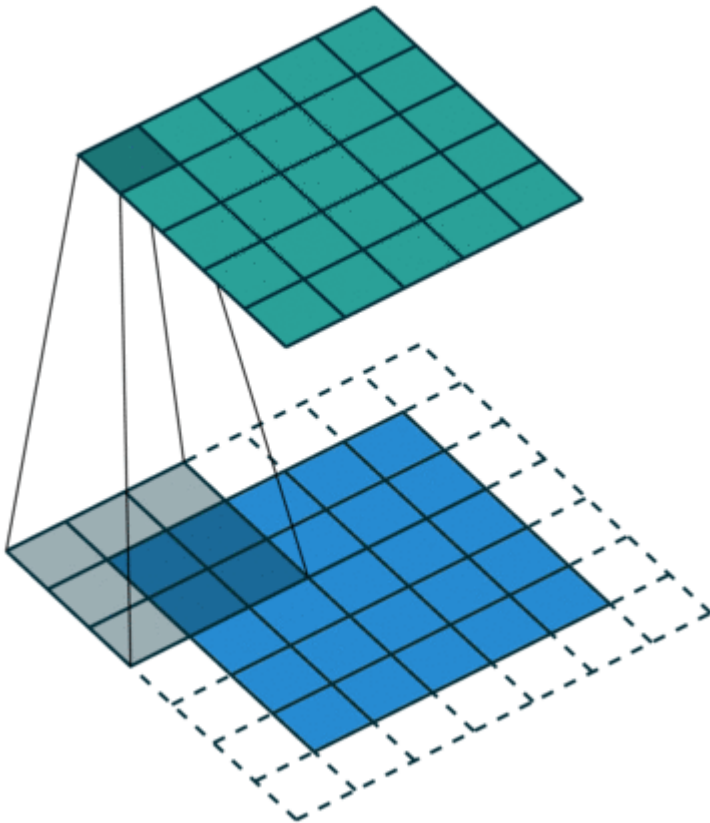
Convolved  
Feature

# Convolution Kernel and Bias

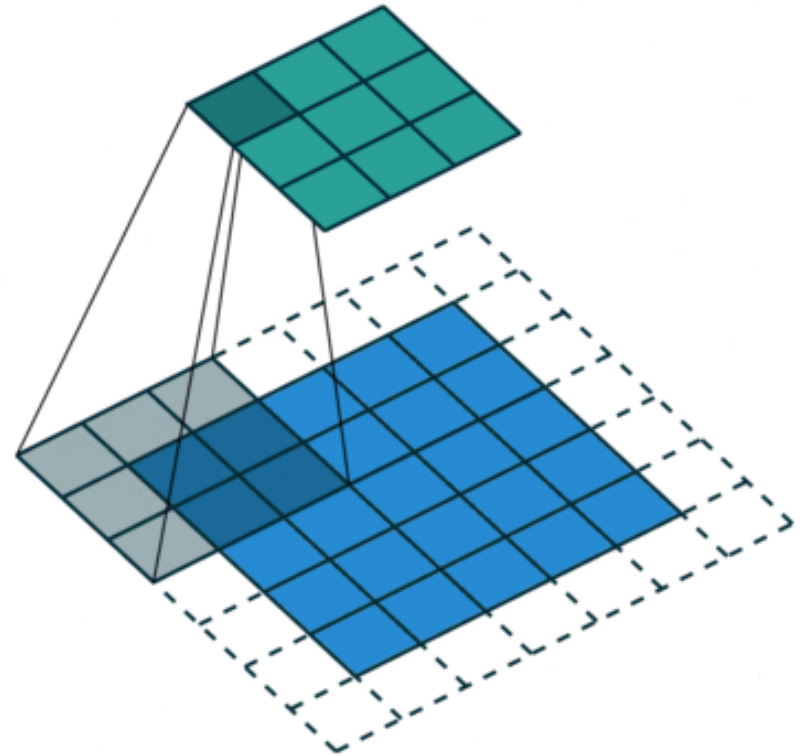


# Stride & Padding

- ▶ 5x5x1 image is padded with 0s to create a 6x6x1 image



- ▶ Convolution Operation with Stride Length = 2



# Pooling

---

- ▶ Pooling layer is responsible for reducing the spatial size of the Convolved Feature.
- ▶ Pooling is also useful to extract dominant features are rotational and positional invariant.
- ▶ There are two types of Pooling: Max Pooling and Average Pooling.
- ▶ **Max Pooling** returns the **maximum value** from the portion of the image covered by the Kernel usually performs better.
- ▶ On the other hand, **Average Pooling** returns the **average of all the values** from the portion of the image covered by the Kernel.
- ▶ Max Pooling also performs as a **Noise Suppressant**.

# Pooling

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

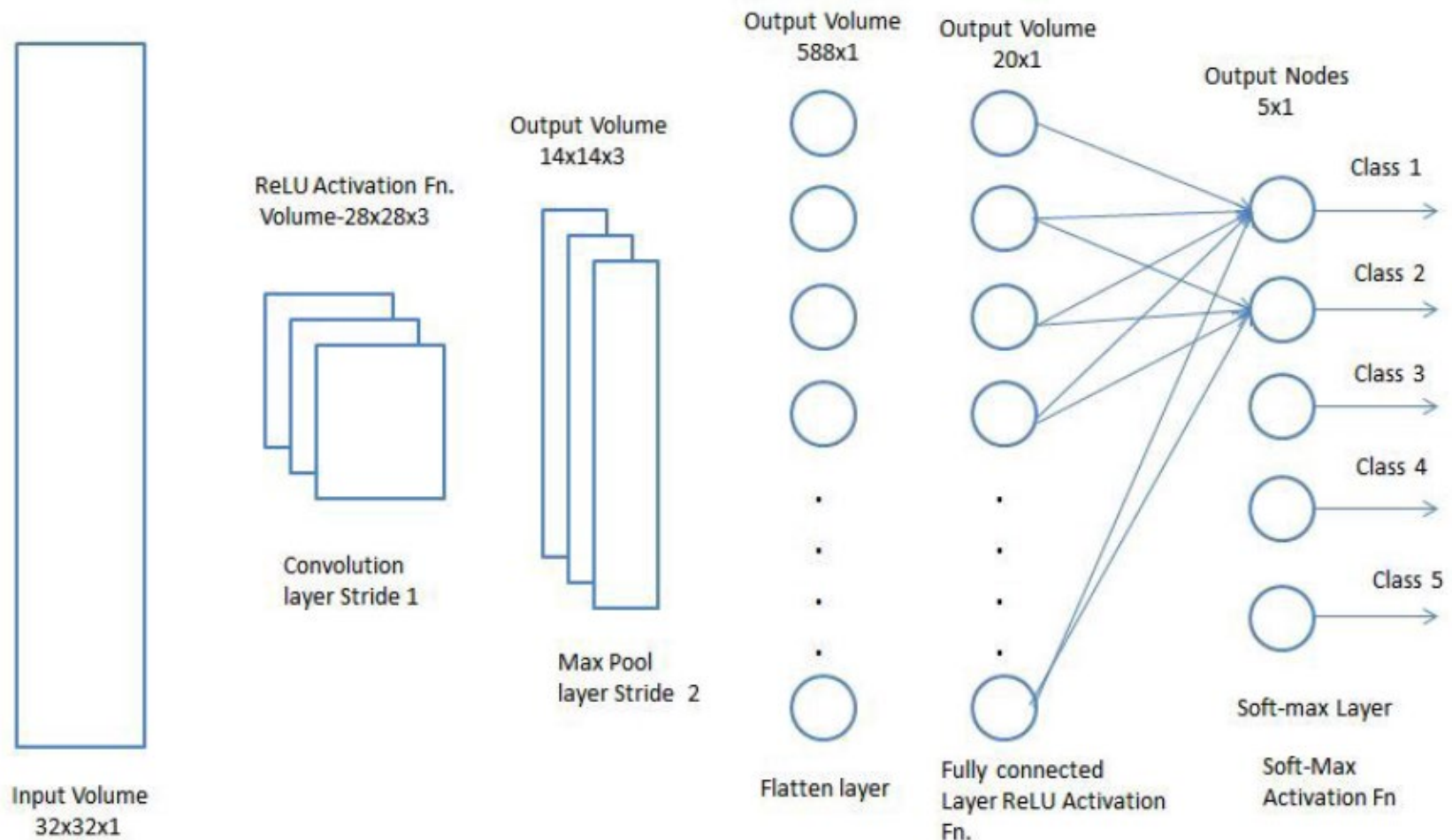
max pooling

20	30
112	37

average pooling

13	8
79	20

# Basic CNN Architecture



# Some well-known CNN Architectures

---

- ▶ There are various architectures of CNNs available which have been key in building algorithms which power and shall power AI as a whole in the foreseeable future.
- ▶ Some of them have been listed below:
  - ❖ LeNet
  - ❖ AlexNet
  - ❖ VGGNet
  - ❖ GoogLeNet
  - ❖ ResNet
  - ❖ ZFNet

# Output Size in Convolutional Layer

---

The size ( $O$ ) of the output image is given by

$$O = \frac{I - K + 2P}{S} + 1$$

$O$  = Size (width) of output image.

$I$  = Size (width) of input image.

$K$  = Size (width) of kernels used in the Conv Layer.

$N$  = Number of kernels.

$S$  = Stride of the convolution operation.

$P$  = Padding.

The number of channels in the output image is equal to the number of kernels  $N$ .

**Example:** In AlexNet, the input image is of size 227x227x3. The first convolutional layer has 96 kernels of size 11x11x3. The stride is 4 and padding is 0. Therefore the size of the output image right after the first bank of convolutional layers is

$$O = \frac{227 - 11 + 2 \times 0}{4} + 1 = 55$$

So, the output image is of size **55x55x96** ( one channel for each kernel ).



# Output Size in Pooling

The size ( $O$ ) of the output image is given by

$$O = \frac{I - P_s}{S} + 1$$

Note that this can be obtained using the formula for the convolution layer by making padding equal to zero and keeping  $P_s$  same as the kernel size. But unlike the convolution layer, the number of channels in the maxpool layer's output is unchanged.

**Example:** In AlexNet, the MaxPool layer after the bank of convolution filters has a pool size of 3 and stride of 2. We know from the previous section, the image at this stage is of size 55x55x96. The output image after the MaxPool layer is of size

$$O = \frac{55 - 3}{2} + 1 = 27$$

So, the output image is of size **27x27x96**.

$O$  = Size (width) of output image.

$I$  = Size (width) of input image.

$S$  = Stride of the convolution operation.

$P_s$  = Pool size.

# # of Parameters in Convolutional Layer

---

- ▶ In a CNN, each layer has two kinds of parameters: weights and biases.
- ▶ The total number of parameters is just the sum of all weights and biases.

$$W_c = K^2 \times C \times N$$

$$B_c = N$$

$$P_c = W_c + B_c$$

$W_c$  = Number of weights of the Conv Layer.

$B_c$  = Number of biases of the Conv Layer.

$P_c$  = Number of parameters of the Conv Layer.

$K$  = Size (width) of kernels used in the Conv Layer.

In a Conv Layer, the depth of every kernel is always equal to the number of channels in the input image. So every kernel has  $K^2 \times C$  parameters, and there are  $N$  such kernels. That's how we come up with the above formula.

**Example:** In AlexNet, at the first Conv Layer, the number of channels ( $C$ ) of the input image is 3, the kernel size ( $K$ ) is 11, the number of kernels ( $N$ ) is 96. So the number of parameters is given by

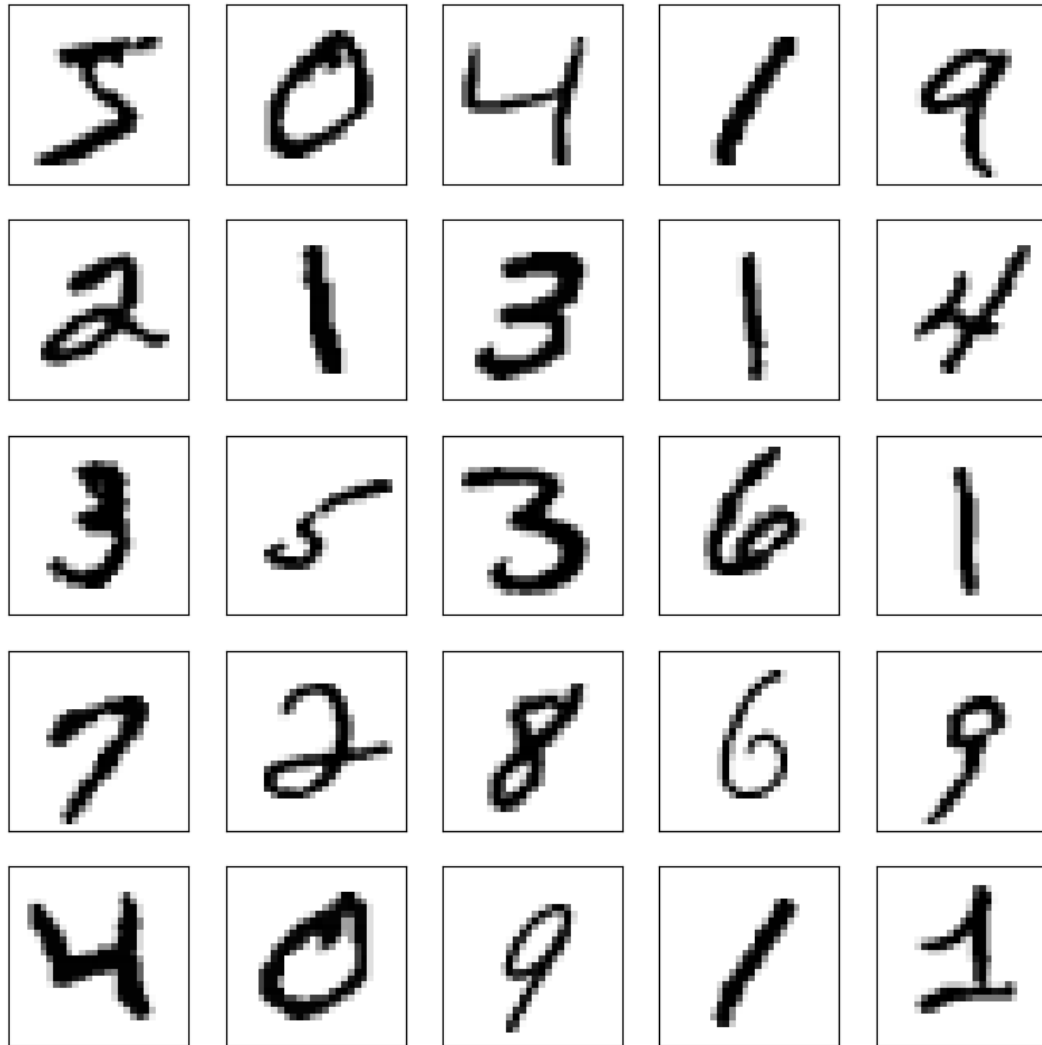
$$W_c = 11^2 \times 3 \times 96 = 34,848$$

$$B_c = 96$$

$$P_c = 34,848 + 96 = 34,944$$

# MNIST Dataset

---



# MNIST Dataset

---

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers

# Model / data parameters
num_classes = 10
input_shape = (28, 28, 1)

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")
>>> y_train[0]
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

10000 test samples

# MNIST Dataset

---

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

```
>>> y_train[0].shape
(10,)
>>> y_train[0]
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

```
>>> x_test.shape
(10000, 28, 28, 1)
>>> y_test.shape
(10000, 10)
```

# MNIST Dataset

---

```
model = keras.Sequential(  
    [  
        keras.Input(shape=input_shape),  
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),  
        layers.MaxPooling2D(pool_size=(2, 2)),  
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),  
        layers.MaxPooling2D(pool_size=(2, 2)),  
        layers.Flatten(),  
        layers.Dropout(0.5),  
        layers.Dense(num_classes, activation="softmax"),  
    ]  
)
```

# MNIST Dataset

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
conv2d (Conv2D)	(None, 26, 26, 32)	320
-----		
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
-----		
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
-----		
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
-----		
flatten (Flatten)	(None, 1600)	0
-----		
dropout (Dropout)	(None, 1600)	0
-----		
dense (Dense)	(None, 10)	16010

=====

Total params: 34,826

Trainable params: 34,826

Non-trainable params: 0

# MNIST Dataset

---

```
batch_size = 128
epochs = 15

model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
```

```
score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```



# MNIST Dataset

```
Epoch 1/15
422/422 [=====] - 14s 32ms/step - loss: 0.3683 - accuracy: 0.8867 - val_loss: 0.0856 - val_accuracy: 0.9778
Epoch 2/15
422/422 [=====] - 13s 32ms/step - loss: 0.1122 - accuracy: 0.9657 - val_loss: 0.0582 - val_accuracy: 0.9833
Epoch 3/15
422/422 [=====] - 13s 32ms/step - loss: 0.0846 - accuracy: 0.9736 - val_loss: 0.0484 - val_accuracy: 0.9867
Epoch 4/15
422/422 [=====] - 14s 32ms/step - loss: 0.0701 - accuracy: 0.9786 - val_loss: 0.0429 - val_accuracy: 0.9888
Epoch 5/15
422/422 [=====] - 14s 32ms/step - loss: 0.0607 - accuracy: 0.9814 - val_loss: 0.0385 - val_accuracy: 0.9890
Epoch 6/15
422/422 [=====] - 14s 32ms/step - loss: 0.0555 - accuracy: 0.9826 - val_loss: 0.0372 - val_accuracy: 0.9895
Epoch 7/15
422/422 [=====] - 14s 32ms/step - loss: 0.0513 - accuracy: 0.9841 - val_loss: 0.0374 - val_accuracy: 0.9880
Epoch 8/15
422/422 [=====] - 14s 32ms/step - loss: 0.0482 - accuracy: 0.9842 - val_loss: 0.0336 - val_accuracy: 0.9915
Epoch 9/15
422/422 [=====] - 14s 33ms/step - loss: 0.0452 - accuracy: 0.9856 - val_loss: 0.0321 - val_accuracy: 0.9913
Epoch 10/15
422/422 [=====] - 14s 33ms/step - loss: 0.0421 - accuracy: 0.9868 - val_loss: 0.0345 - val_accuracy: 0.9903
Epoch 11/15
422/422 [=====] - 14s 32ms/step - loss: 0.0403 - accuracy: 0.9872 - val_loss: 0.0312 - val_accuracy: 0.9910
Epoch 12/15
422/422 [=====] - 14s 32ms/step - loss: 0.0372 - accuracy: 0.9878 - val_loss: 0.0323 - val_accuracy: 0.9905
Epoch 13/15
422/422 [=====] - 14s 32ms/step - loss: 0.0357 - accuracy: 0.9889 - val_loss: 0.0308 - val_accuracy: 0.9913
Epoch 14/15
422/422 [=====] - 13s 32ms/step - loss: 0.0349 - accuracy: 0.9887 - val_loss: 0.0329 - val_accuracy: 0.9917
Epoch 15/15
422/422 [=====] - 14s 32ms/step - loss: 0.0329 - accuracy: 0.9894 - val_loss: 0.0301 - val_accuracy: 0.9913
```

```
Test loss: 0.02621997892856598
Test accuracy: 0.991100013256073
```

# IMDB Reviews Application

- ▶ We'll work with the IMDB dataset: a set of 50,000 highly polarized reviews from the Internet Movie Database. They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews

```
from tensorflow.keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
```

```
>>> train_data.shape
(25000,)
>>> train_data[0]
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2,
, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16,
6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17,
12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5,
25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 5952, 15, 25
6, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2
071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 10
4, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 5345, 19, 178, 32]
```

# IMDB Reviews Application

---

- ▶ We need to vectorize input:

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
```

```
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

```
>>> x_train[0]
array([0., 1., 1., ..., 0., 0., 0.])
```

# IMDB Reviews Application

---

```
model = keras.Sequential([
    layers.Dense(16, activation='relu'),
    layers.Dense(16, activation='relu'),
    layers.Dense(1, activation='sigmoid')])
```

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

history = model.fit(partial_x_train, partial_y_train,
                    epochs=20, batch_size=512, validation_data=(x_val, y_val))

history_dict = history.history
```

# IMDB Reviews Application

```
model = keras.Sequential([  
    layers.Dense(16, activation='relu'),  
    layers.Dense(16, activation='relu'),  
    layers.Dense(1, activation='sigmoid')])
```

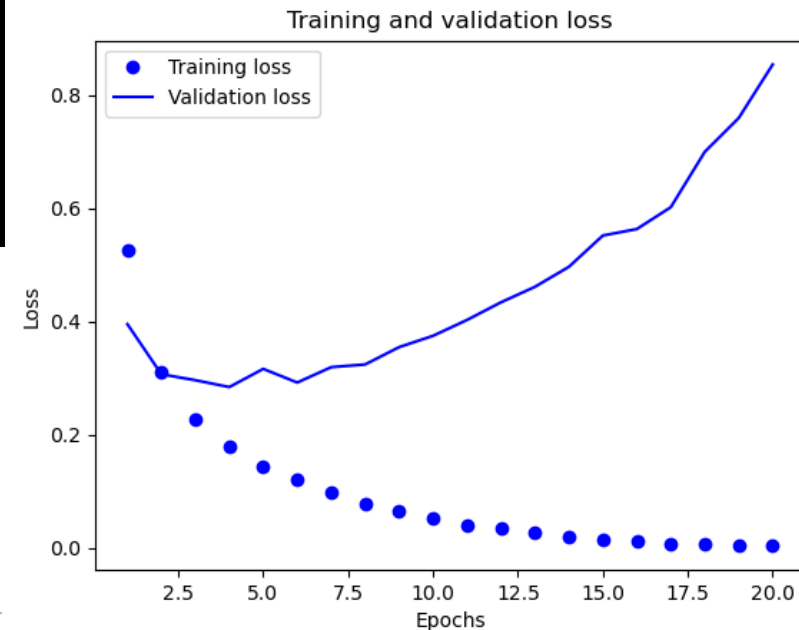
```
>>> model.summary()  
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 16)	160016
dense_1 (Dense)	(None, 16)	272
dense_2 (Dense)	(None, 1)	17

Total params: 160,305  
Trainable params: 160,305  
Non-trainable params: 0

# IMDB Reviews Application

```
import matplotlib.pyplot as plt
history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

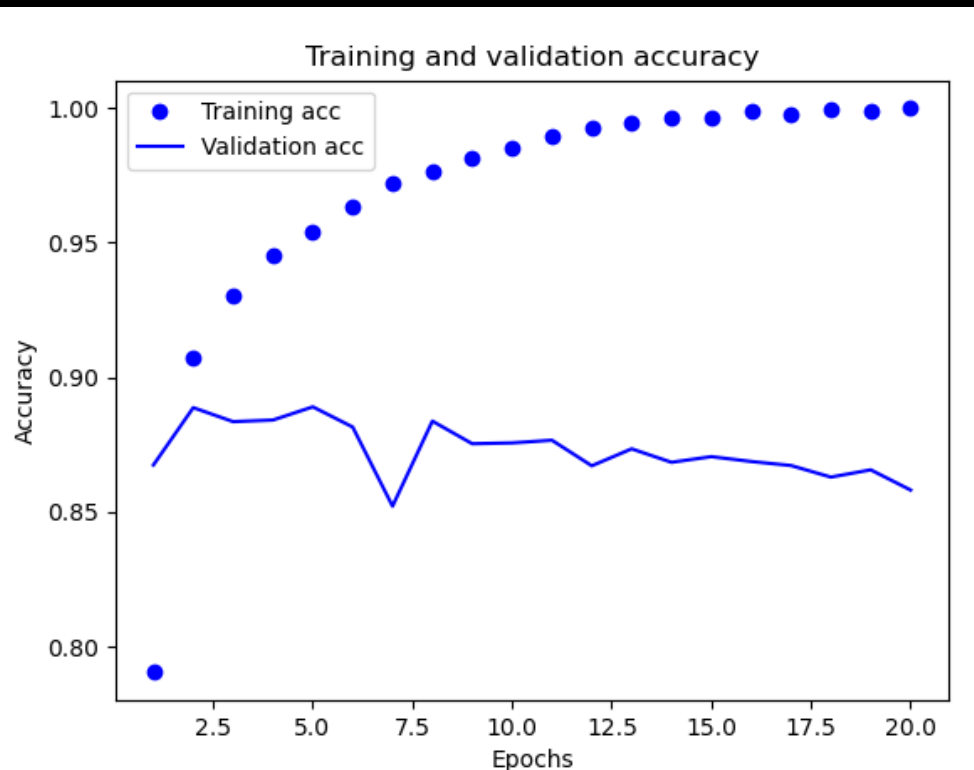


# IMDB Reviews Application

```
(tf_2) E:\Elements\Ders_Notlari\803400815021_MachineLearningWithPython\deepLearning>python -i imdbreviews.py
Epoch 1/20
30/30 [=====] - 2s 25ms/step - loss: 0.5256 - acc: 0.7795 - val_loss: 0.3954 - val_acc: 0.8693
Epoch 2/20
30/30 [=====] - 0s 11ms/step - loss: 0.3118 - acc: 0.9035 - val_loss: 0.3069 - val_acc: 0.8863
Epoch 3/20
30/30 [=====] - 0s 11ms/step - loss: 0.2272 - acc: 0.9277 - val_loss: 0.2964 - val_acc: 0.8824
Epoch 4/20
30/30 [=====] - 0s 11ms/step - loss: 0.1784 - acc: 0.9411 - val_loss: 0.2842 - val_acc: 0.8860
Epoch 5/20
30/30 [=====] - 0s 11ms/step - loss: 0.1433 - acc: 0.9545 - val_loss: 0.3164 - val_acc: 0.8760
Epoch 6/20
30/30 [=====] - 0s 11ms/step - loss: 0.1199 - acc: 0.9626 - val_loss: 0.2922 - val_acc: 0.8855
Epoch 7/20
30/30 [=====] - 0s 11ms/step - loss: 0.0971 - acc: 0.9713 - val_loss: 0.3194 - val_acc: 0.8814
Epoch 8/20
30/30 [=====] - 0s 11ms/step - loss: 0.0792 - acc: 0.9780 - val_loss: 0.3242 - val_acc: 0.8832
Epoch 9/20
30/30 [=====] - 0s 11ms/step - loss: 0.0652 - acc: 0.9823 - val_loss: 0.3548 - val_acc: 0.8766
Epoch 10/20
30/30 [=====] - 0s 11ms/step - loss: 0.0528 - acc: 0.9864 - val_loss: 0.3748 - val_acc: 0.8779
Epoch 11/20
30/30 [=====] - 0s 11ms/step - loss: 0.0396 - acc: 0.9917 - val_loss: 0.4028 - val_acc: 0.8763
Epoch 12/20
30/30 [=====] - 0s 11ms/step - loss: 0.0350 - acc: 0.9926 - val_loss: 0.4340 - val_acc: 0.8753
Epoch 13/20
30/30 [=====] - 0s 11ms/step - loss: 0.0260 - acc: 0.9949 - val_loss: 0.4613 - val_acc: 0.8756
Epoch 14/20
30/30 [=====] - 0s 11ms/step - loss: 0.0208 - acc: 0.9963 - val_loss: 0.4967 - val_acc: 0.8695
Epoch 15/20
30/30 [=====] - 0s 11ms/step - loss: 0.0143 - acc: 0.9989 - val_loss: 0.5520 - val_acc: 0.8630
Epoch 16/20
30/30 [=====] - 0s 11ms/step - loss: 0.0131 - acc: 0.9981 - val_loss: 0.5634 - val_acc: 0.8704
Epoch 17/20
30/30 [=====] - 0s 11ms/step - loss: 0.0071 - acc: 0.9997 - val_loss: 0.6020 - val_acc: 0.8662
Epoch 18/20
30/30 [=====] - 0s 11ms/step - loss: 0.0073 - acc: 0.9997 - val_loss: 0.6998 - val_acc: 0.8567
Epoch 19/20
30/30 [=====] - 0s 11ms/step - loss: 0.0044 - acc: 0.9998 - val_loss: 0.7596 - val_acc: 0.8608
Epoch 20/20
30/30 [=====] - 0s 10ms/step - loss: 0.0035 - acc: 0.9998 - val_loss: 0.8541 - val_acc: 0.8524
```

# IMDB Reviews Application

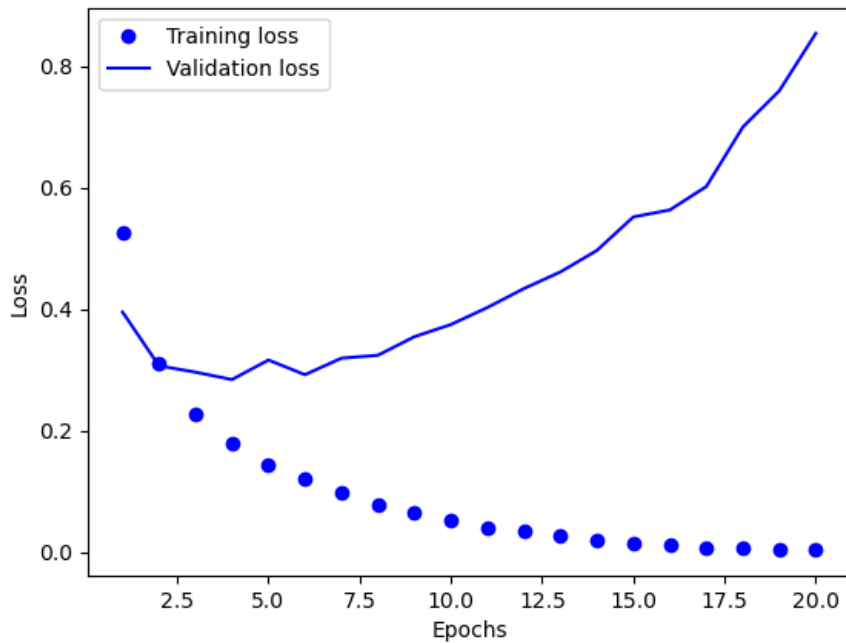
```
acc = history_dict["acc"]
val_acc = history_dict["val_acc"]
plt.plot(epochs, acc, "bo", label="Training acc")
plt.plot(epochs, val_acc, "b", label="Validation acc")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```



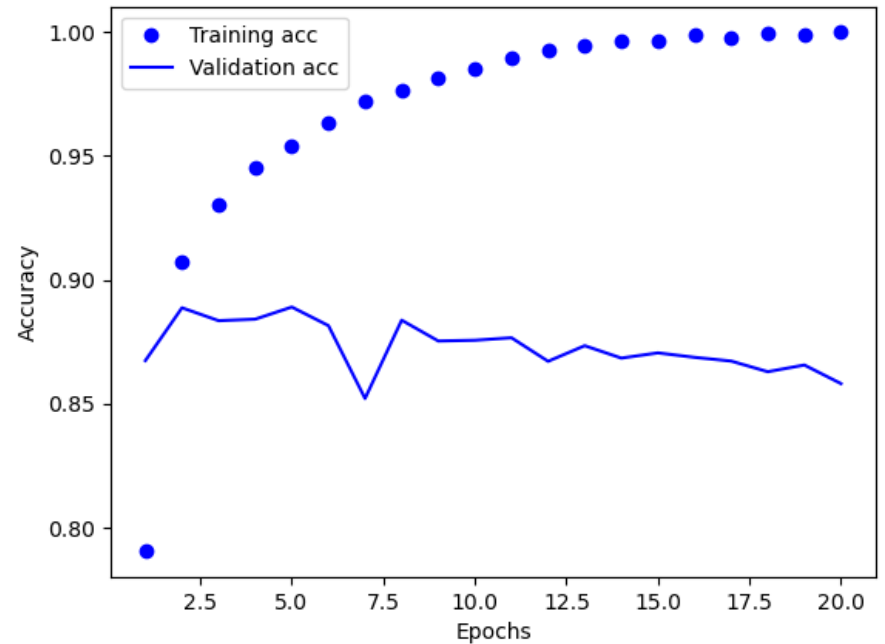


# IMDB Reviews Application

Training and validation loss



Training and validation accuracy



## ► References

- 1 Buduma, N., Locascio, N. (2017). *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*, O'Reilly Publications.
- 2 Chollet, F. (2020). *Deep Learning with Python*, Manning Publishing.
- 3 <https://scikit-learn.org/>
- 4 <https://towardsdatascience.com/>
- 5 McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* 2nd Edition.
- 6 Albon, C. (2018). *Machine Learning with Python Cookbook: Practical Solutions from Preprocessing to Deep Learning*
- 7 Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems* 1st Edition
- 8 Müller, A. C., Guido, S. (2016). *Introduction to Machine Learning with Python: A Guide for Data Scientists*
- 9 Burkov, A. (2019). *The Hundred-Page Machine Learning Book*.
- 10 Burkov, A. (2020). *Machine Learning Engineering*.
- 11 <https://www.kaggle.com/ankitjha/comparing-regression-models/notebook>
- 12 <https://towardsdatascience.com/>
- 13 <https://towardsdatascience.com/machine-learning-project-17-compare-classification-algorithms-87cb50e1cb60>
- 14 <https://machinelearningmastery.com/compare-machine-learning-algorithms-python-scikit-learn/>
- 15 <https://developers.google.com/edu/python/>
- 16 <http://learnpythonthehardway.org/book/>
- 17 <https://blog.kthais.com/using-shap-to-explain-machine-learning-models-3f8f9c3b1f5e>
- 18 <https://keras.io>