# Deep Learning

Tristan Cazenave

Tristan.Cazenave@dauphine.psl.eu

# Why Deep Learning Now?

- Processing power: GPU
- Improvement of architectures: activations, residual connections, attention layers
- Open source frameworks: Tensorflow, Keras, Pytorch
- More and more available data

# Deep Learning

- Computer Vision
- Generative Networks
- Recurrent Neural Networks
- Natural Language Processing
- Graph Neural Networks
- Computer Games

# Computer Vision

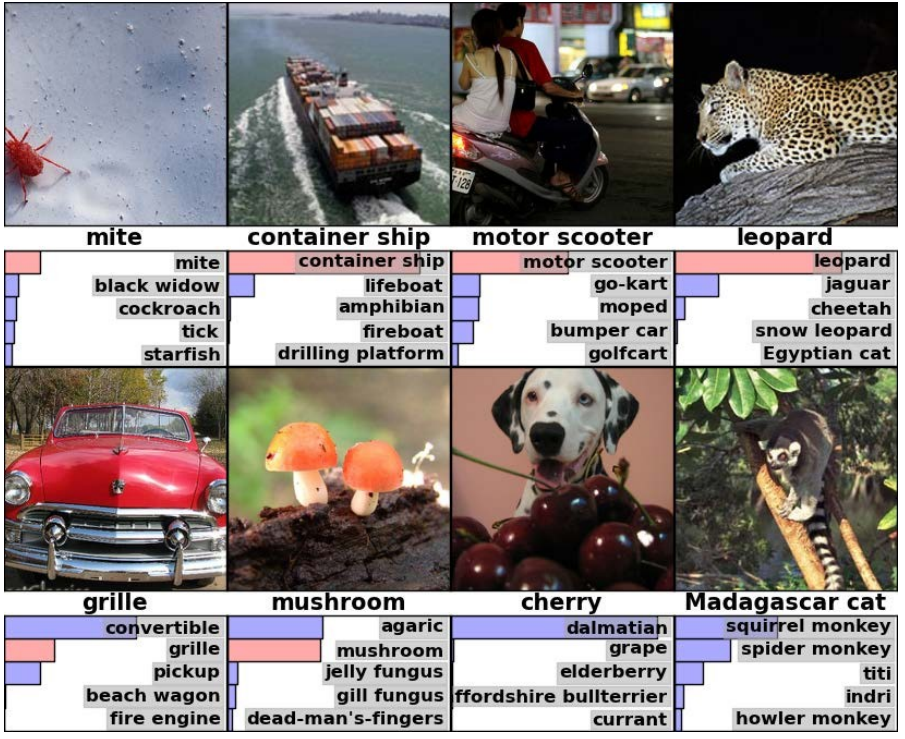
- Convolutional neural networks
- Small numbers of parameters
- Good generalization
- Bigger is better
- Vision transformers

# Computer Vision

- LeNet (1990)

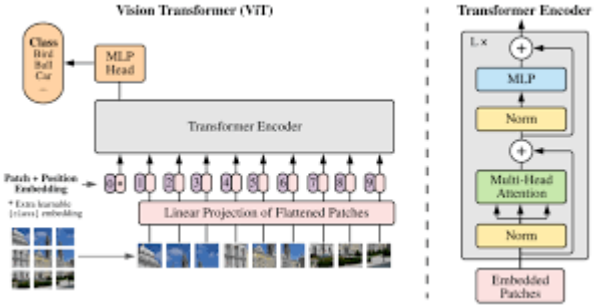
# Computer Vision

- AlexNet (2012)



# Computer Vision

- Vision Transformers (2021)




# Computer Vision

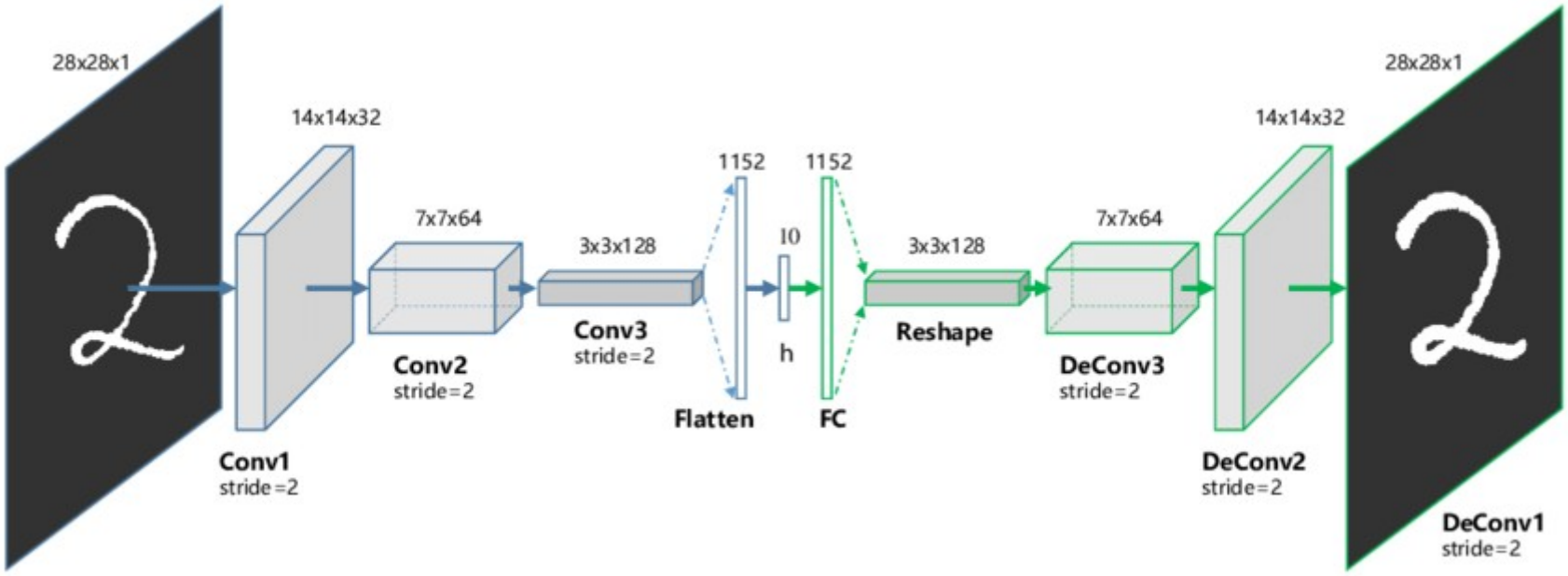
- Vision Transformers have been used in many Computer Vision tasks with excellent results:
  - Image Classification
  - Object Detection
  - Video Deepfake Detection
  - Image segmentation
  - Anomaly detection
  - Image Synthesis
  - Cluster analysis
  - Autonomous Driving

# Generative Networks

- Autoencoders
- Generative Adversarial Networks (GAN) : image generation
- GPT : text generation

# Generative Networks

- Autoencoders

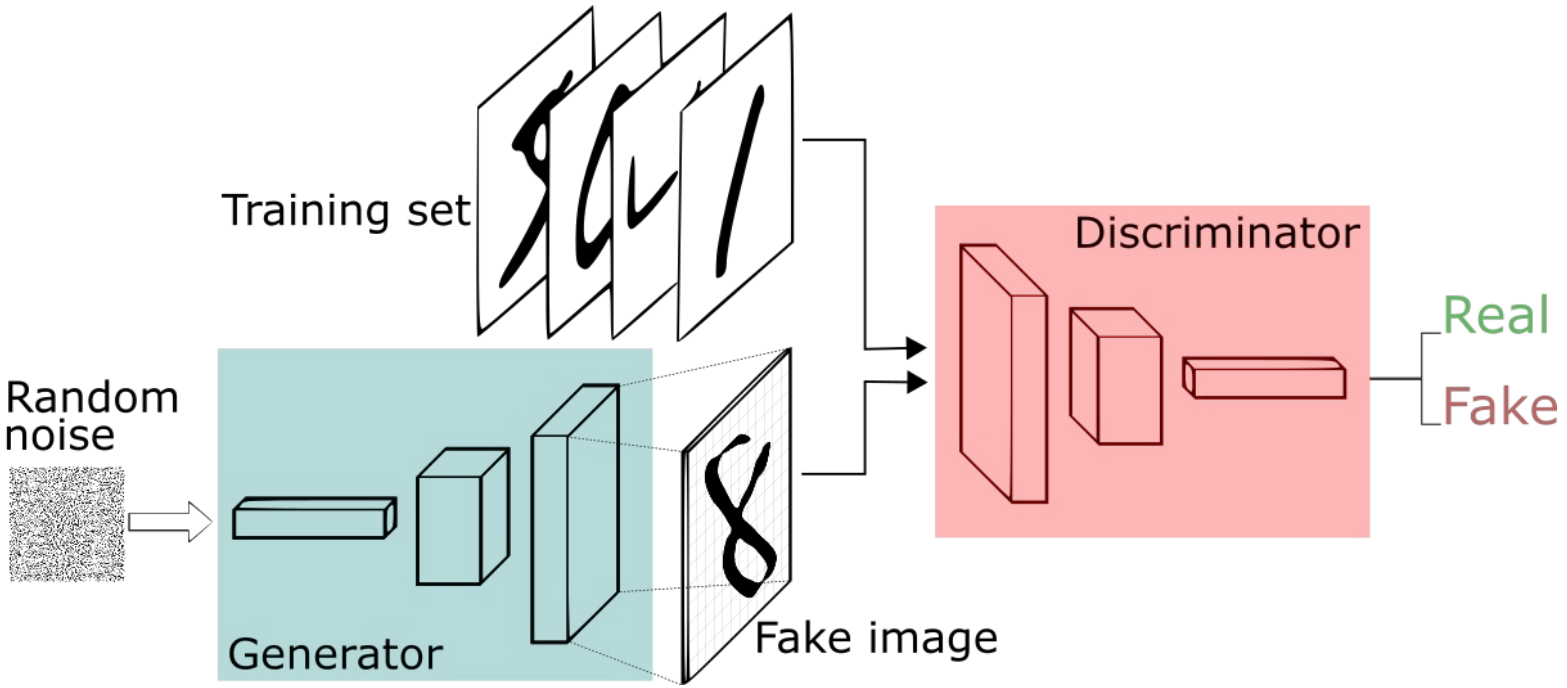


# Generative Networks

- Applications of Autoencoders :
  - Dimensionality Reduction
  - Feature Extraction
  - Image Denoising
  - Image Compression
  - Image Search
  - Anomaly Detection
  - Missing Value Imputation

# Generative Networks

- Generative Adversarial Networks (GAN) :

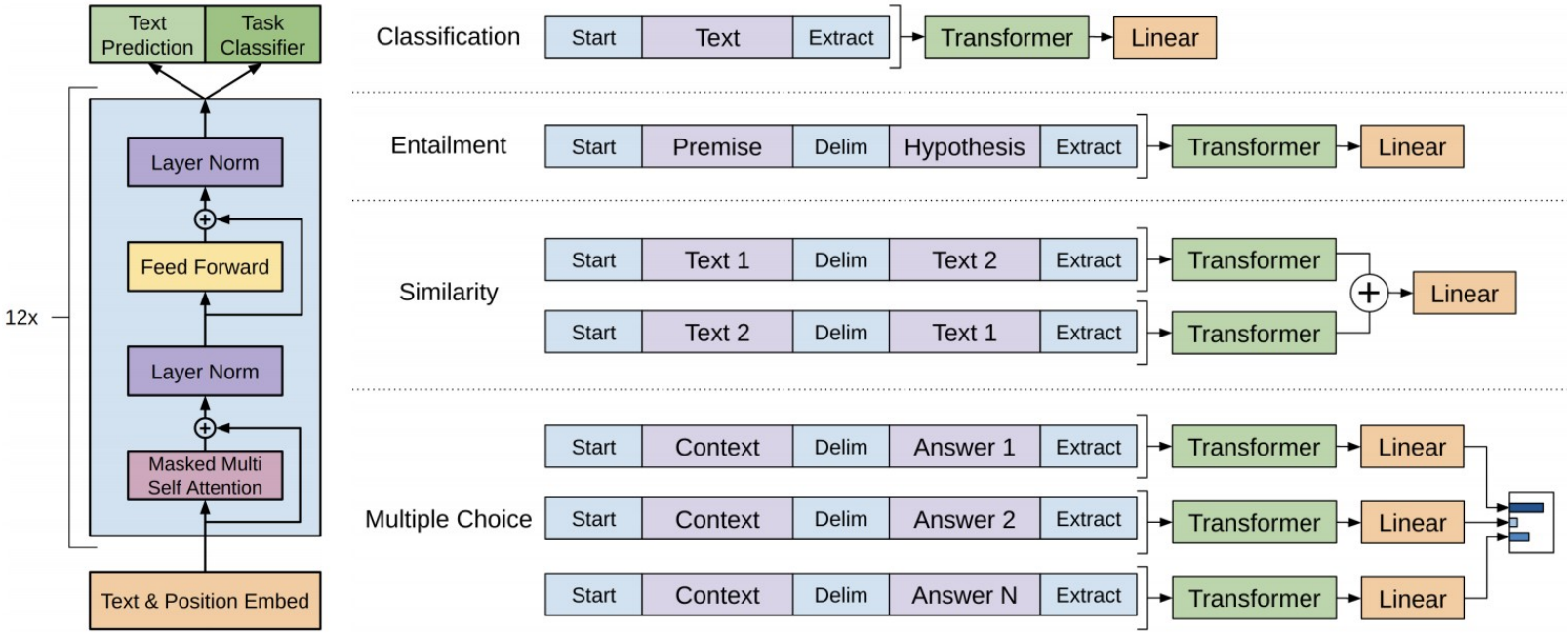


# Generative Networks

- Applications of GAN:
  - Text to Image Generation
  - Image to Image Translation (colorizing,...)
  - Increasing Image Resolution
  - Predicting Next Video Frame

# Generative Networks

- GPT (Generative Pre-trained Transformer):



# Generative Networks

- Applications of GPT:
  - GPT-3, specifically the Codex model, is the basis for GitHub Copilot, a code completion and generation software that can be used in various code editors and IDEs.
  - GPT-3 is used in certain Microsoft products to translate conventional language into formal computer code.
  - GPT-3 has been used by Andrew Mayne for AI Writer, which allows people to correspond with historical figures via email.
  - GPT-3 has been used by Jason Rohrer in a retro-themed chatbot project named "Project December", which is accessible online and allows users to converse with several Ais.
  - GPT-3 was used by The Guardian to write an article about AI being harmless to human beings. It was fed some ideas and produced eight different essays, which were ultimately merged into one article.
  - GPT-3 was used in AI Dungeon, which generates text-based adventure games. Later it was replaced by a competing model after OpenAI changed their policy regarding generated content.

# Generative Networks

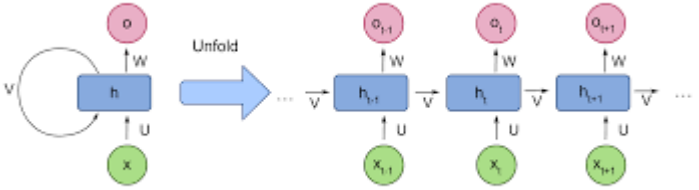
- Applications of GPT:
  - MoliAIre
  - MoliAIre-RIME
  - Brecht
  - https://www.lamsade.dauphine.fr/molierelebot

# Recurrent Neural Networks

- RNN
- LSTM
- GRU

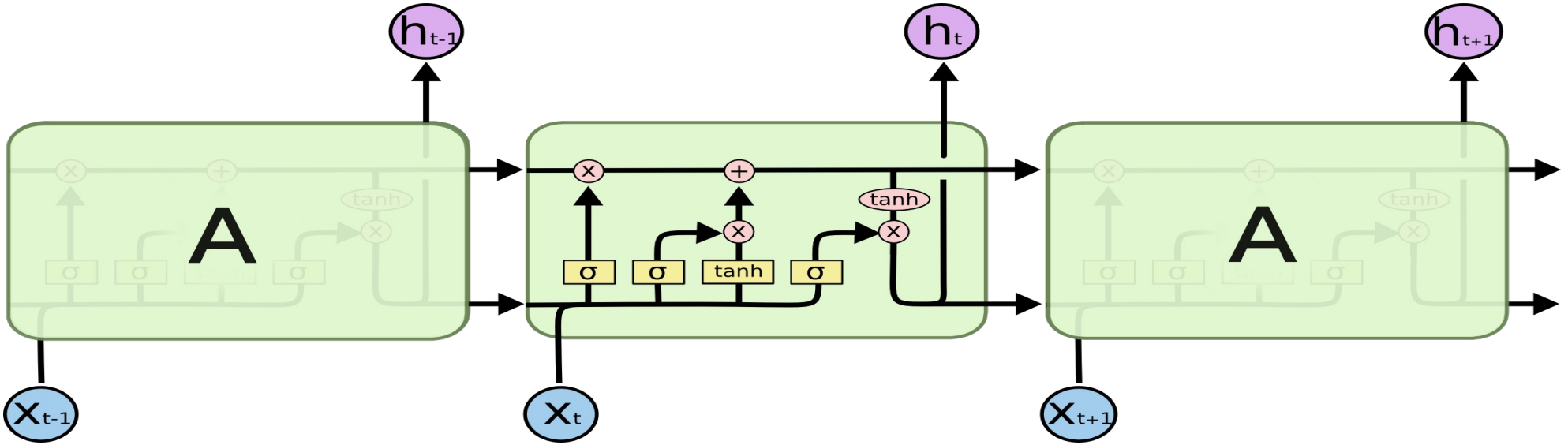
# Recurrent Neural Networks

- RNN :



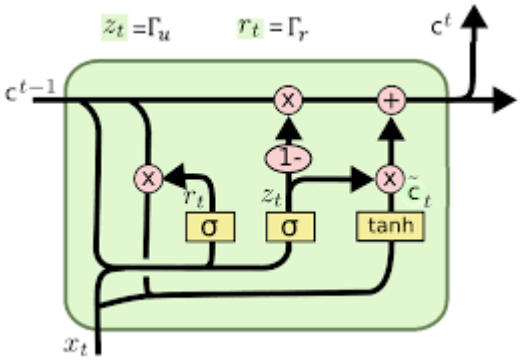
# Recurrent Neural Networks

- LSTM (Long Short-Term Memory) :

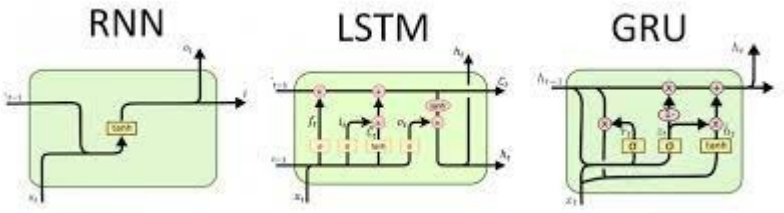


# Recurrent Neural Networks

- GRU (Gated Reccurent Unit):



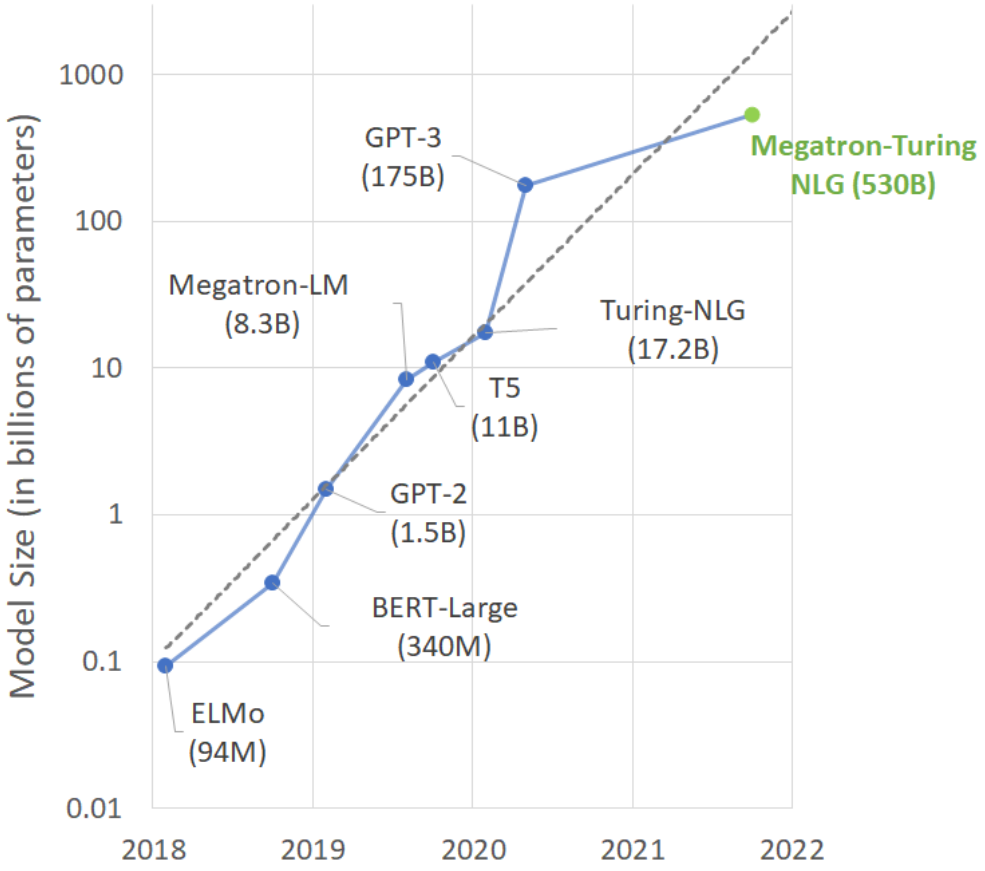
# Recurrent Neural Networks



# Natural Language Processing

- RNN
- Transformers :
  - Bert (RoBERTa, CamemBERT, FlauBERT)
  - GPT (GPT-2, GPT-3, GPT-fr, GPT-4)
  - Text to Text (T5)

# Natural Language Processing

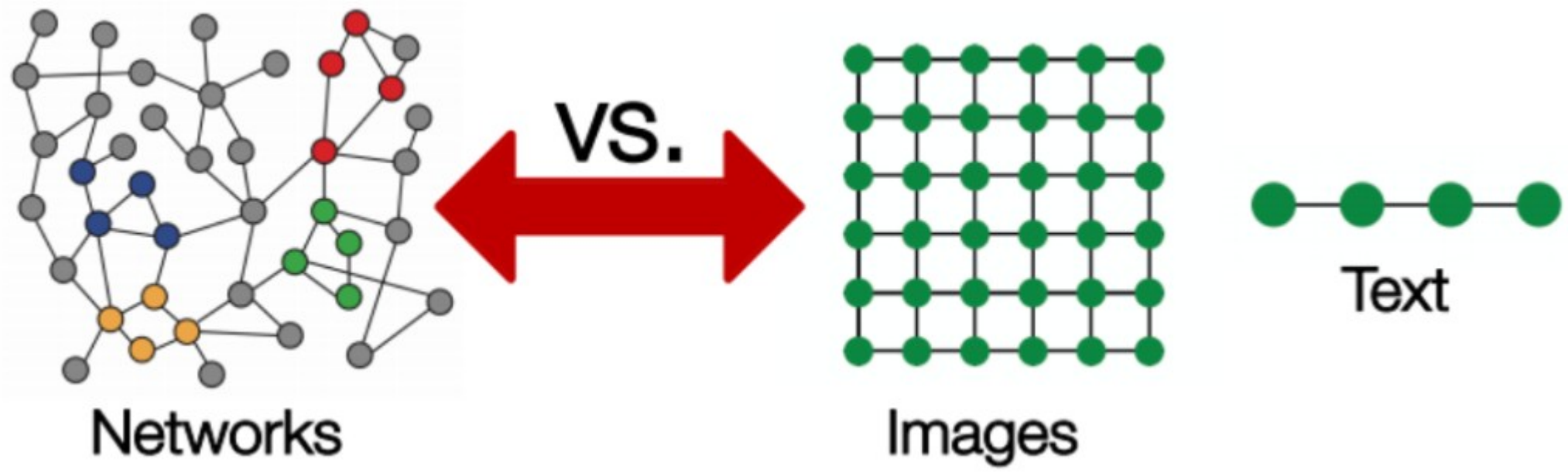


# Graph Neural Networks

- GNN
- GCN
- Spectral

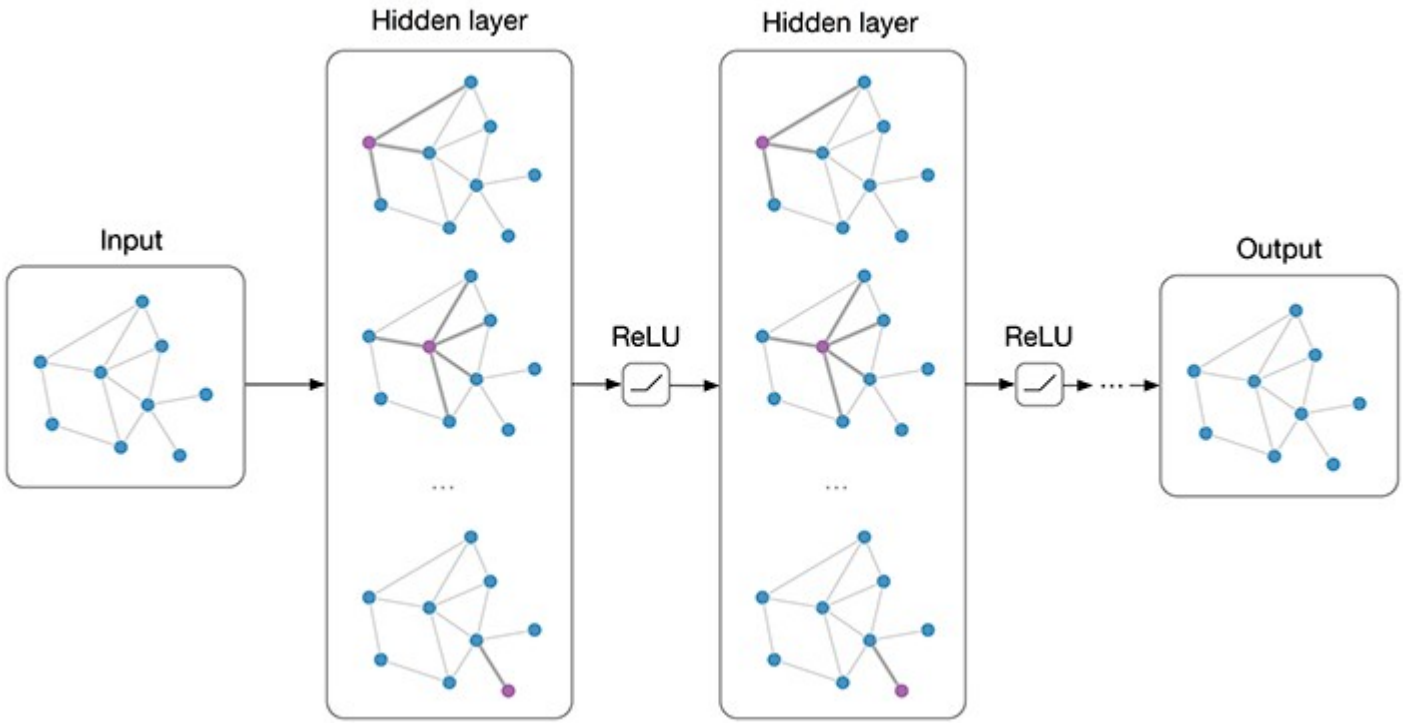
# Graph Neural Networks

- GNN :



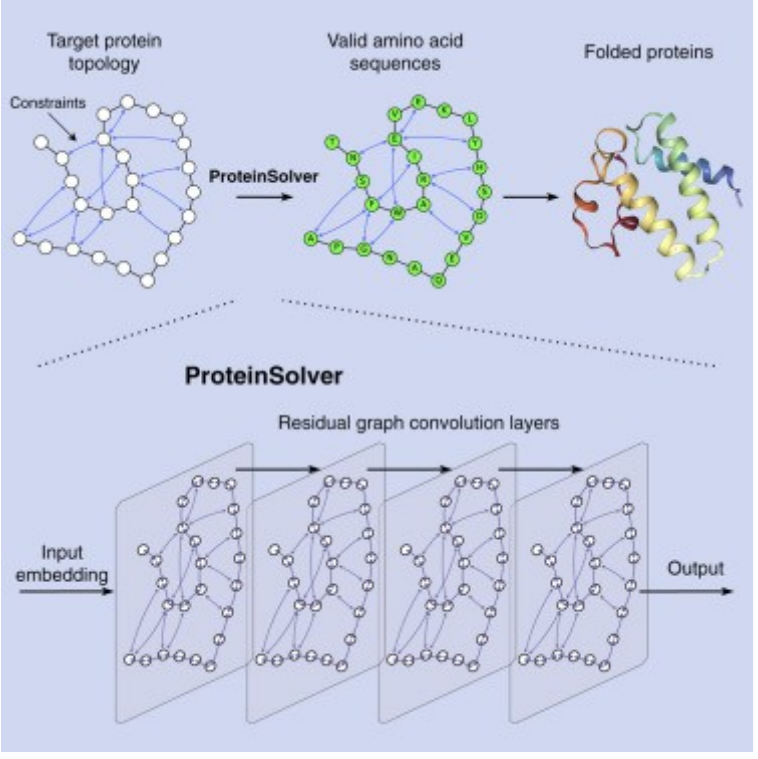
# Graph Neural Networks

- GCN (Graph Convolutional Networks) :



# Graph Neural Networks

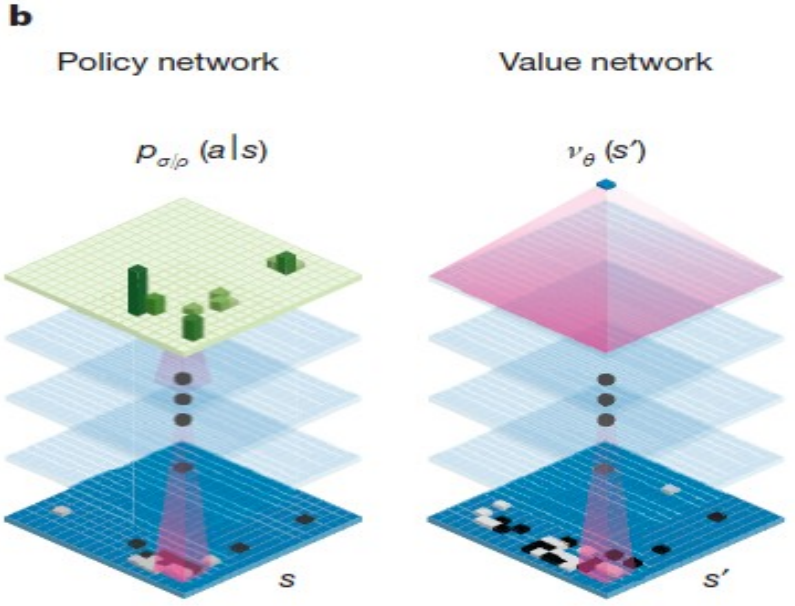
- Applications of GNN:



# Computer Games

- AlphaGo
- AlphaZero
- LeelaChess Zero
- AlphaStar
- Athénan

# AlphaGo



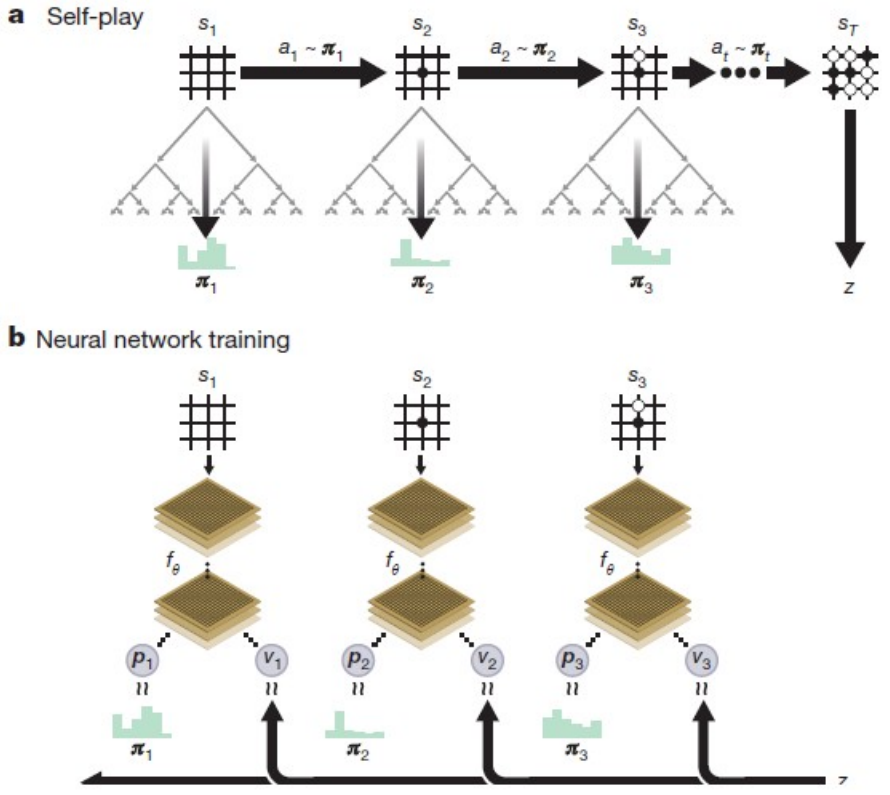
# AlphaGo

Lee Sedol is among the strongest and most famous 9p Go player:

AlphaGo Lee won 4-1 against Lee Sedol in march 2016.

# Computer Games

- Alpha Zero:



# Computer Games

- LeelaChess Zero:


SSDF

Rating list | Games | Hardware information | Changelog | Contact us

**2022-05-24**

**160235 games played by 424 computers**

The Swedish Ratinglist may be quoted in other magazines, but we insist that this will be done in a correct way! We expect, that not only the rating figures, but also the number of games and the margin of error will be quoted.

Please read the comment by the chairman, Lars Sandin. You may also download the list in DOS text format. Please note that this is a longer list, with almost all tested computers since SSDF began its work more than 20 years ago!

All games have been played on the tournament level, 40 moves/2 hours followed by 20 moves/each following hour. In matches between PC-programs, two separate PCs have been used, connected with an auto232-cable.

If you have any questions about the list you are welcome to contact us.

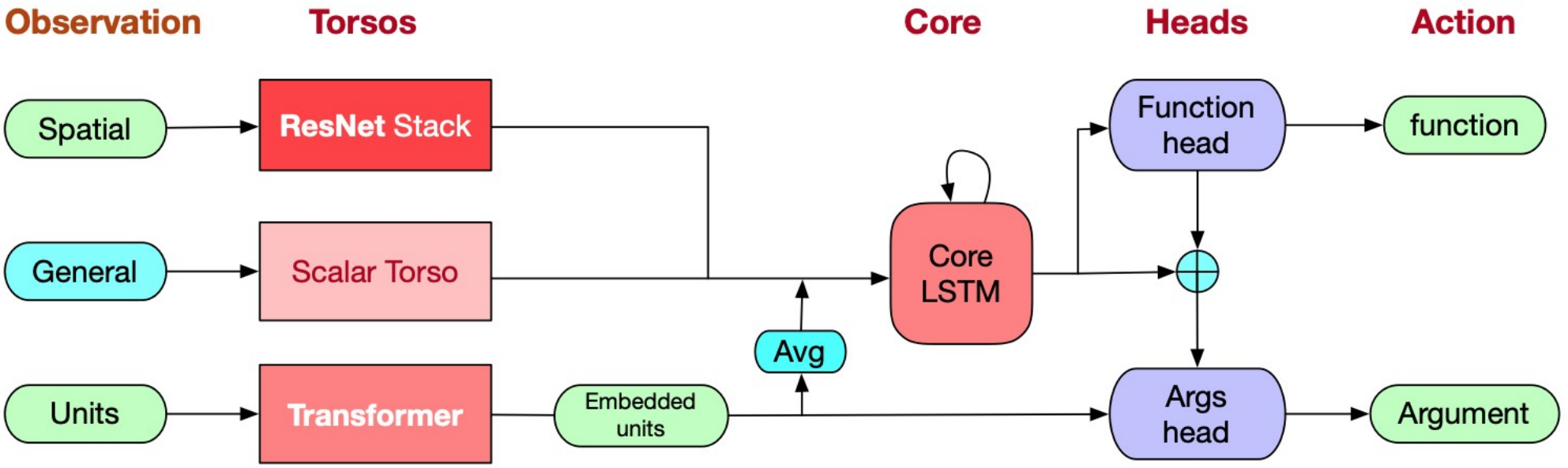
| | | Rating | + | - | Games | Won | Av.opp |
|---|---|---|---|---|---|---|---|
| 1 | Lc0 0.28.2 Cuda-611213 3060Ti | 3591 | 48 | -43 | 280 | 71% | 3438 |
| 2 | Stockfish 13 x64 1800X 3.6 GHz | 3577 | 37 | -34 | 440 | 70% | 3426 |
| 3 | Lc0 0.26.3 Cuda(67362) 3060Ti | 3572 | 29 | -27 | 680 | 68% | 3442 |
| 4 | Dragon Komodo 2.51 x64 1800X 3.6 GHz | 3566 | 42 | -39 | 320 | 65% | 3456 |
| 5 | Stockfish 14 x64 1800X 3.6 GHz | 3555 | 39 | -37 | 360 | 66% | 3444 |
| 6 | Stockfish 12 NNUE x64 1800X 3.6 GHz | 3555 | 30 | -29 | 560 | 62% | 3465 |
| 7 | Dragon by Komodo x64 1800X 3.6 GHz | 3544 | 34 | -32 | 460 | 63% | 3449 |
| 8 | Stockfish 11 x64 1800X 3.6 GHz | 3537 | 36 | -34 | 450 | 70% | 3393 |
| 9 | Stockfish 10 x64 1800X 3.6 GHz | 3514 | 25 | -24 | 880 | 68% | 3378 |
| 10 | Dragon Komodo 2 MCTS x64 1800X 3.6 GHz | 3479 | 45 | -44 | 240 | 55% | 3444 |
| 11 | Stockfish 9 x64 1800X 3.6 GHz | 3476 | 26 | -24 | 882 | 70% | 3330 |
| 12 | Komodo 13.1 x64 1800X 3.6 GHz | 3463 | 30 | -29 | 560 | 62% | 3379 |
| 13 | Komodo 14 x64 1800X 3.6 GHz | 3461 | 31 | -31 | 480 | 53% | 3442 |
| 14 | Komodo 13.02 x64 1800X 3.6 GHz | 3457 | 30 | -29 | 600 | 65% | 3346 |
| 15 | Komodo 12.3 x64 1800X 3.6 GHz | 3447 | 27 | -26 | 760 | 66% | 3329 |
| 16 | Arasan 23.01 x64 1800X 3.6 GHz | 3443 | 39 | -38 | 320 | 52% | 3429 |
| 17 | Stockfish 9 x64 Q6600 2.4 GHz | 3439 | 32 | -31 | 480 | 56% | 3394 |
| 18 | Booot 6.5 x64 1800X 3.6 GHz | 3426 | 38 | -38 | 320 | 49% | 3435 |
| 19 | Pedone 3.1 x64 1800X 3.6 GHz | 3423 | 46 | -47 | 220 | 48% | 3433 |
| 20 | Dragon Komodo MCTS x64 1800X 3.6 GHz | 3404 | 38 | -38 | 320 | 52% | 3391 |
| 21 | Wasp 5.5 x64 1800X 3.6 GHz | 3395 | 49 | -51 | 194 | 41% | 3456 |
| 22 | Wasp 5 x64 1800X 3.6 GHz | 3378 | 41 | -44 | 280 | 38% | 3470 |
| 23 | Booot 6.4 x64 1800X 3.6 GHz | 3366 | 26 | -26 | 360 | 51% | 3359 |

# Computer Games

- AlphaStar:

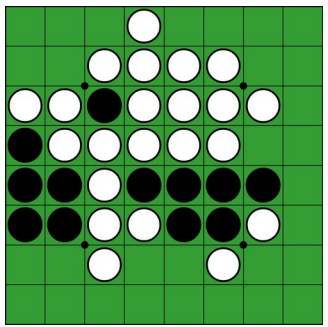
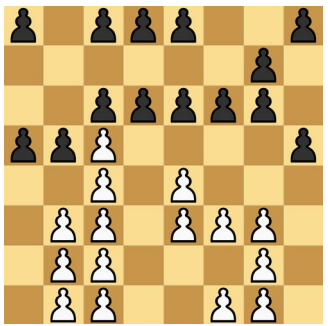
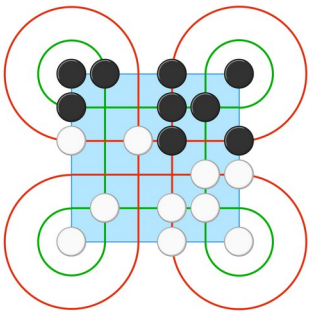
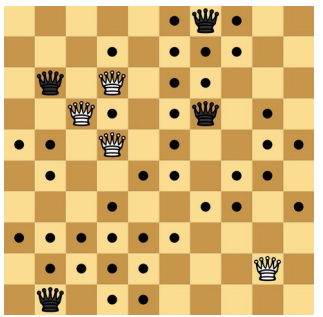
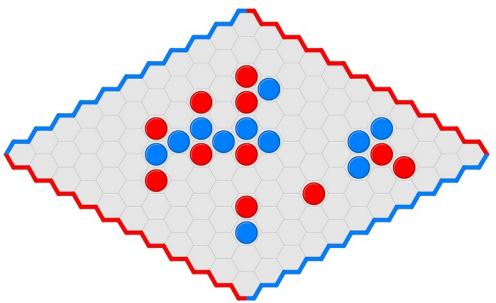
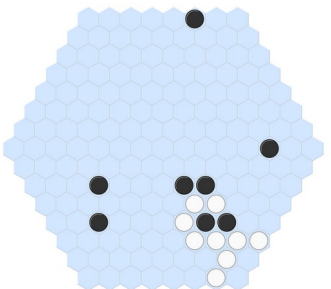
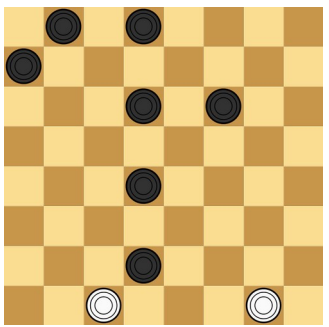
# Computer Games

- AlphaStar:



# Computer Games

Athénan: 11 gold medals at the 2021 Computer Olympiad!

# Historical Background

# A spectrum of machine learning tasks

Typical Statistics----------Artificial Intelligence

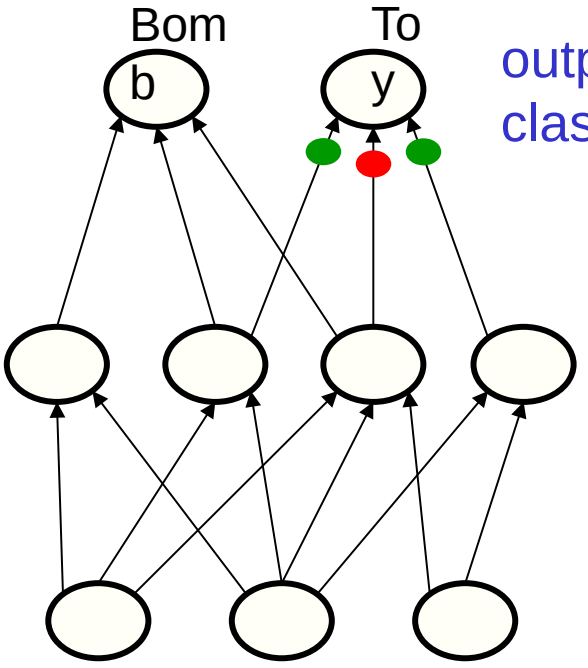
- Low-dimensional data (e.g. less than 100 dimensions)
- Lots of noise in the data
- There is not much structure in the data, and what structure there is, can be represented by a fairly simple model.
- The main problem is distinguishing true structure from noise.

- High-dimensional data (e.g. more than 100 dimensions)
- The noise is not sufficient to obscure the structure in the data if we process it right.
- There is a huge amount of structure in the data, but the structure is too complicated to be represented by a simple model.
- The main problem is figuring out a way to represent the complicated structure so that it can be learned.

# Historical background:

## First generation neural networks

- Perceptrons (~1960) used a layer of hand-coded features and tried to recognize objects by learning how to weight these features.
  - There was a neat learning algorithm for adjusting the weights.
  - But perceptrons are fundamentally limited in what they can learn to do.




output units e.g. class labels

non-adaptive hand-coded features

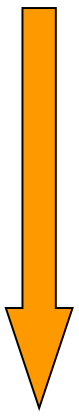
input units e.g. pixels

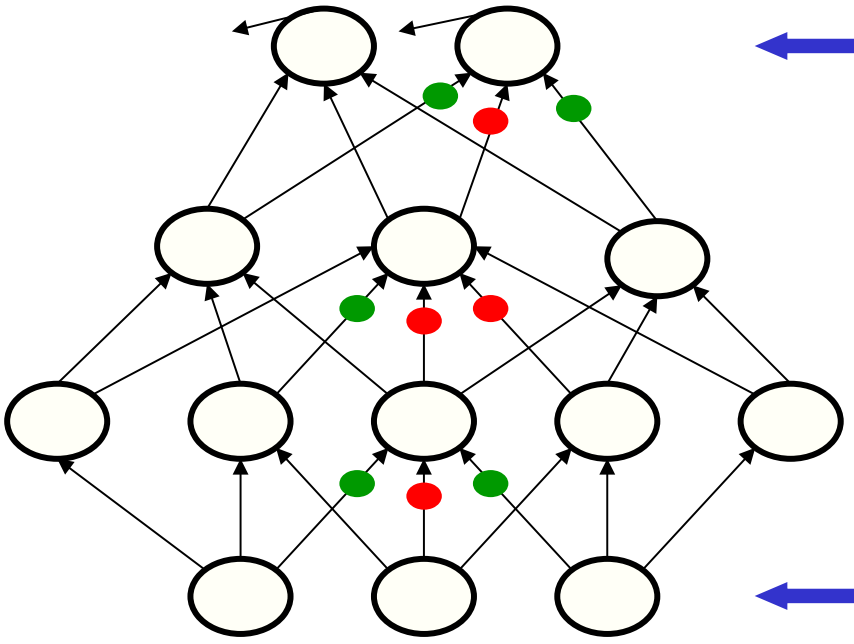
Sketch of a typical perceptron from the 1960's

# Second generation neural networks (~1985)

Compare outputs with correct answer to get error signal

Back-propagate error signal to get derivatives for learning





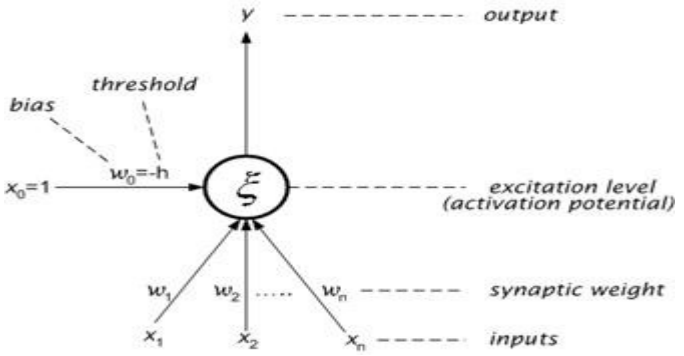
outputs

hidden layers

input vector

# Formalization of a neural network

- Each neuron computes its output y as a linear combinations of its inputs $x_i$ followed by an activation function :



# Formalization of a neural network

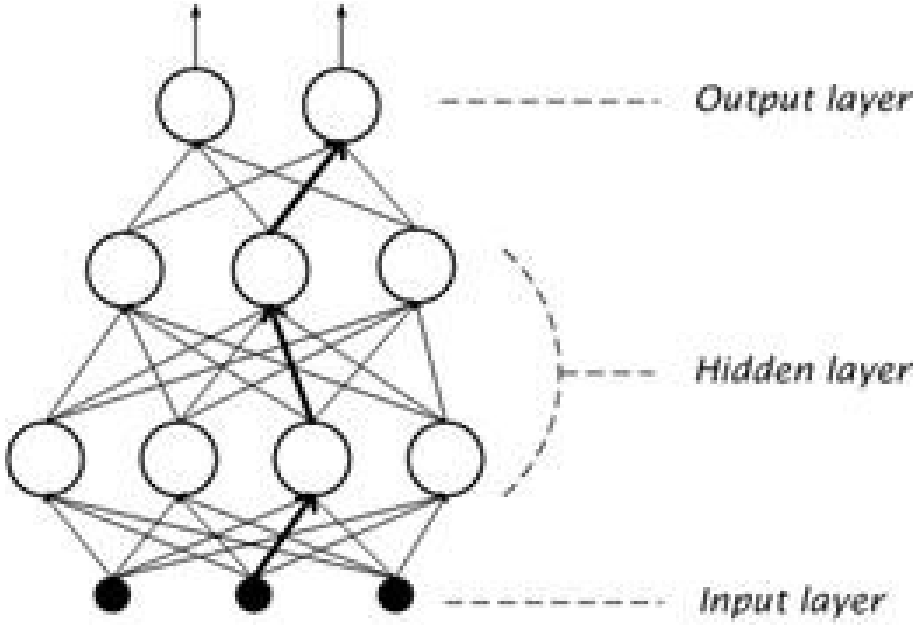
- Linear combination : $\xi = \sum_{i=1}^{n} w_i x_i$

- Activation function :

$$y = \sigma(\xi) = \begin{cases} 1 \; if \; \xi \geq 0 \\ 0 \; if \; \xi < 0 \end{cases} where \; \xi = \sum_{i=0}^{n} w_i x_i$$

# Formalization of a neural network

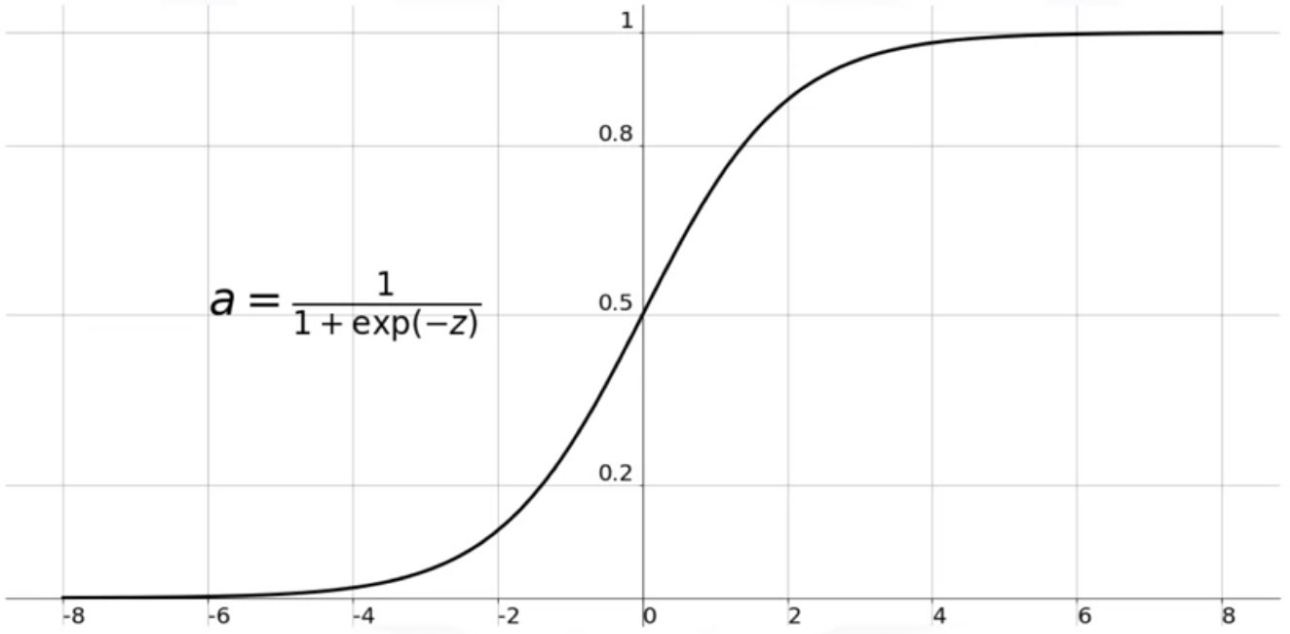
- Architecture :



# Activation Functions

## Sigmoid Function

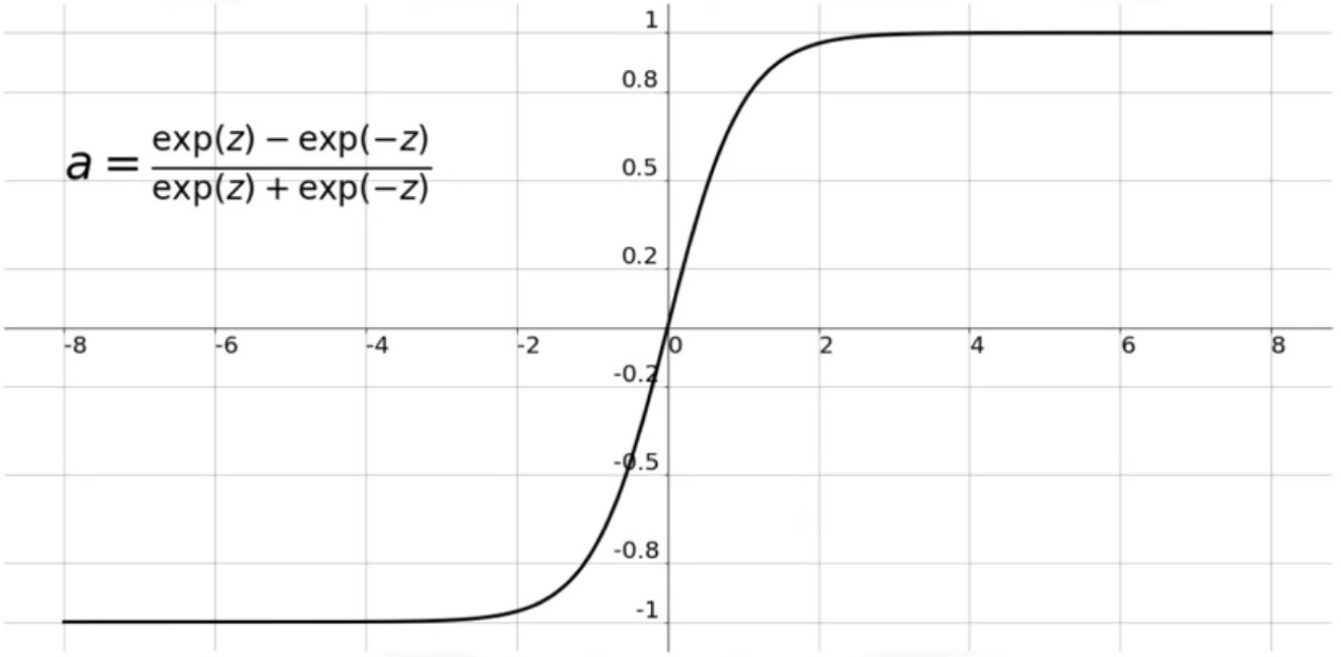
$$a = \frac{1}{1 + \exp(-z)}$$



# Activation Functions

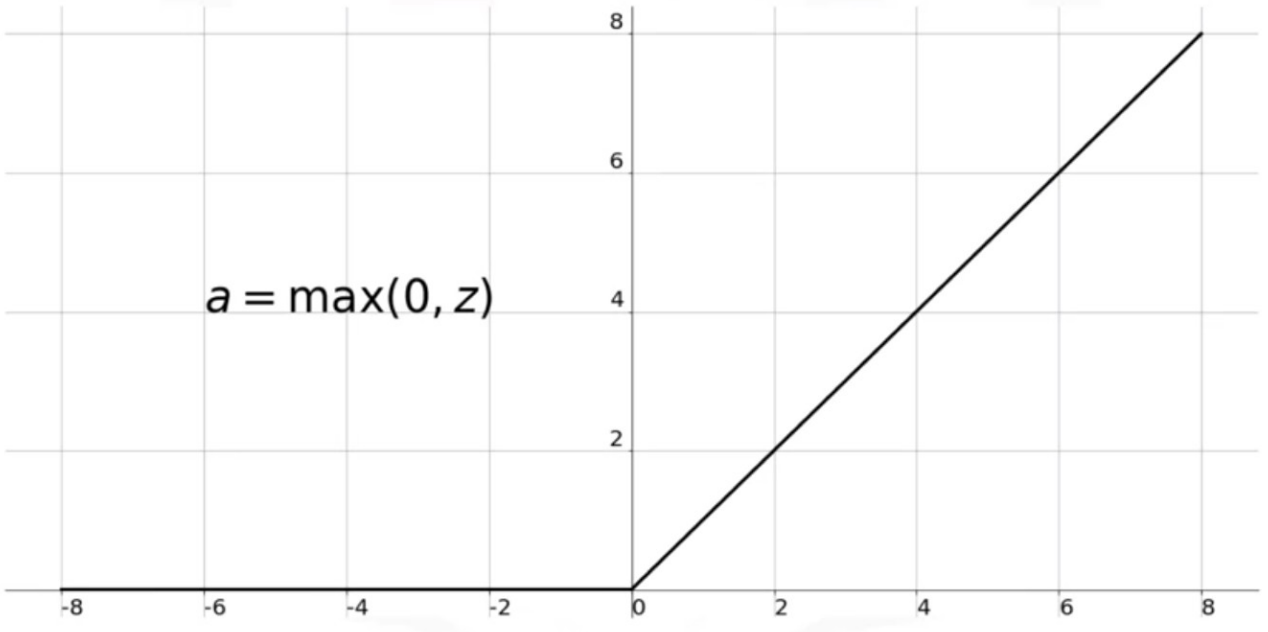
## Hyperbolic Tangent Function

$$a = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$



# Activation Functions

## ReLU Function



# Activation Functions

## Softmax Function

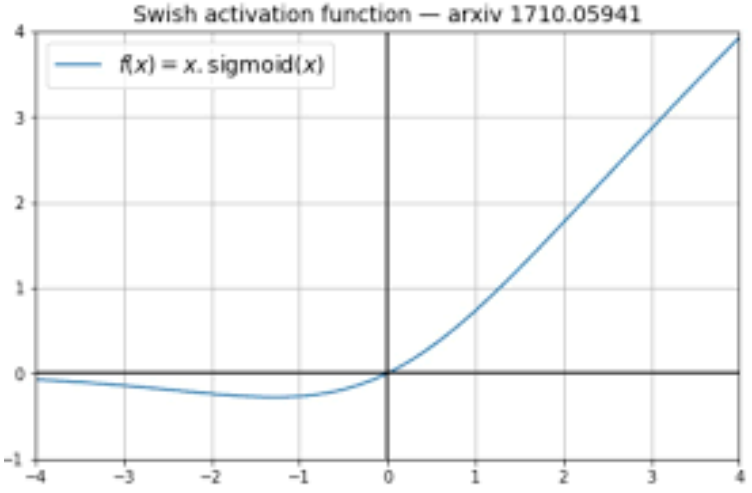
$$a_i = \frac{e^{z_i}}{\sum_{k=1}^{m} e^{z_k}}$$

$$z = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 1.6 \\ 0.55 \\ 0.98 \end{bmatrix}$$

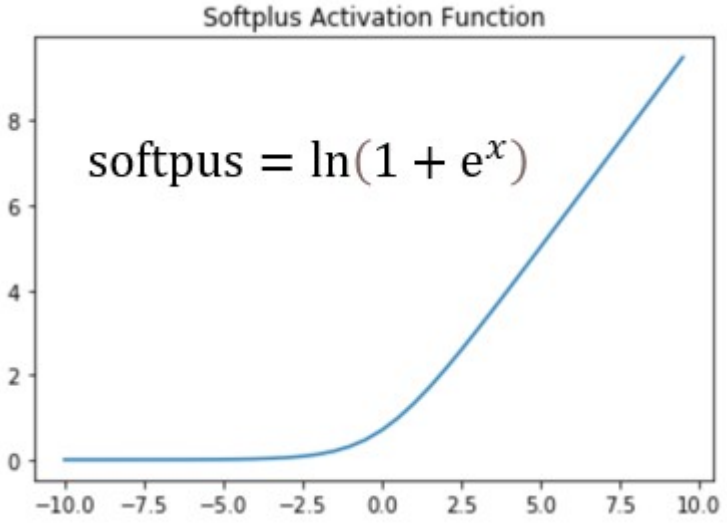
Softmax

$$a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0.51 \\ 0.18 \\ 0.31 \end{bmatrix}$$

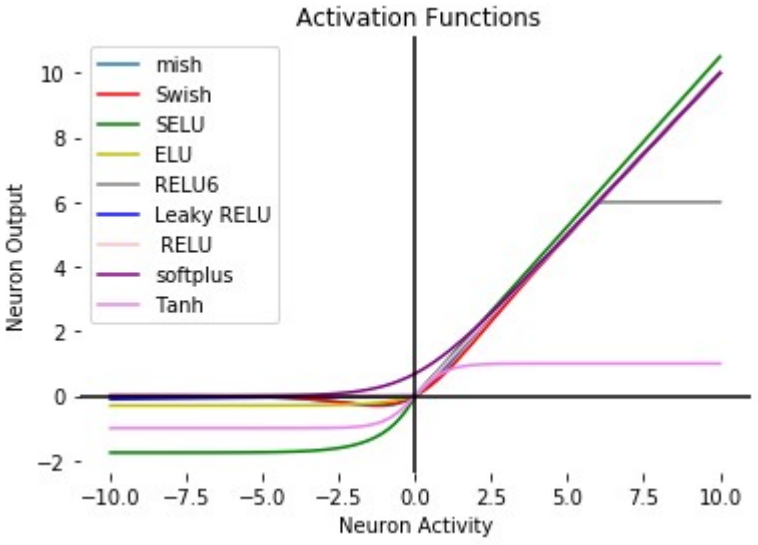
# Activation Functions



# Activation Functions



# Activation Functions



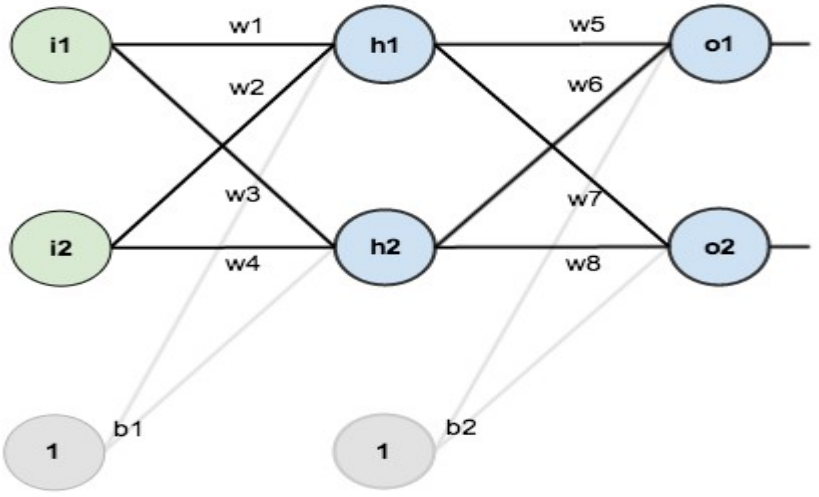
# Backpropagation

# Backpropagation

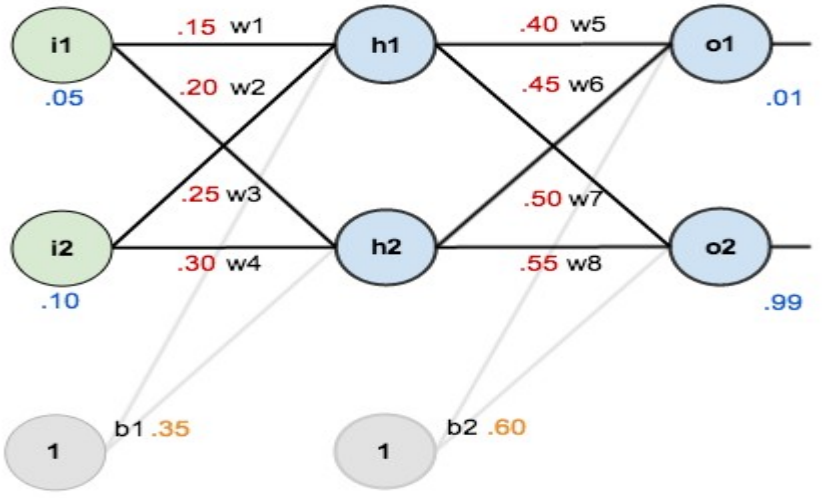
- We are going to explain backpropagation on a simple example.

- We take as example a network with two inputs, two outputs and two hidden neurons .

# Backpropagation



# Backpropagation



# Backpropagation

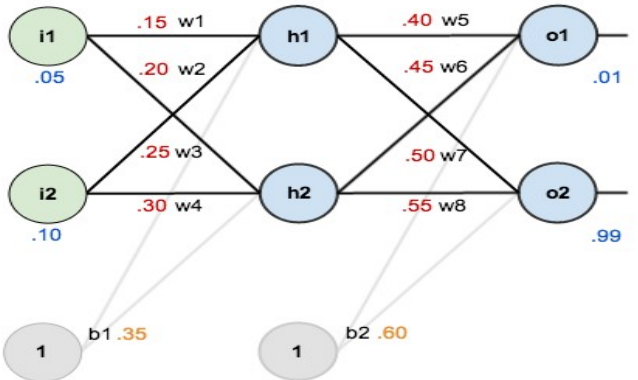
- The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.

- We are going to work with a single training set: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.

# The Forward Pass

- We figure out the total net input to each hidden layer neuron.
- Squash the total net input using an activation function (here we use the sigmoid function).
- Repeat the process with the output layer neurons.

# The Forward Pass



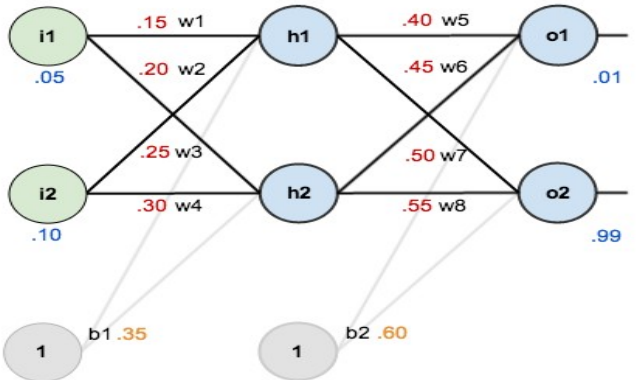

- Here's how we calculate the total net input for h1:

  $net_{h1} = w1 * i1 + w2 * i2 + b1 * 1$

  $net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$
- We then squash it using the sigmoid function to get the output of h1 :

  $out_{h1} = 1/(1+e^{-net_{h1}}) = 1/(1+e^{-0.3775}) = 0.593269992$
- Carrying out the same process for h2 we get : $out_{h2} = 0.596884378$

# The Forward Pass




- We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.
- Here's the output for o1: $net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$

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

  $out_{o1} = 1/(1+e^{-net_{o1}}) = 1/(1+e^{-1.105905967}) = 0.75136507$
- And carrying out the same process for o2 we get: $out_{o2} = 0.772928465$

# The Error

- We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

  $E_{total} = \Sigma\ 1/2\ (target - output)^2$

- For example, the target output for o1 is 0.01 but the neural network output 0.75136507, therefore its error is:

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

- Repeating this process for o2 we get: $E_{o2} = 0.023560026$
- The total error for the neural network is the sum of these errors:

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

# The Backwards Pass

- Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

# Output Layer

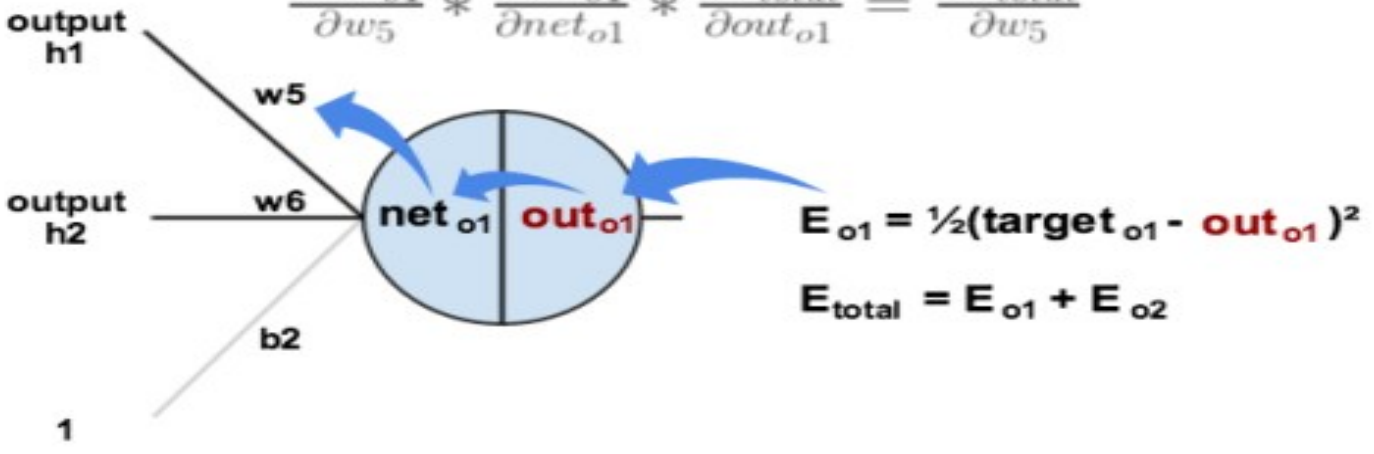
- Consider w5.
- We want to know how much a change in w5 affects the total error:

  $\delta E_{total} / \delta w5$
- $\delta E_{total} / \delta w5$ is read as “the partial derivative of $E_{total}$ with respect to w5“.
- You can also say “the gradient with respect to w5“.
- By applying the chain rule we know that:

  $\delta E_{total} / \delta w5 = \delta E_{total} / \delta out_{o1} * \delta out_{o1} / \delta net_{o1} * \delta net_{o1} / \delta w5$

# Output Layer

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



# Output Layer

- We need to figure out each piece in this equation.
- First, how much does the total error change with respect to the output?

$$E_{total} = 1/2\ (target_{o1} - out_{o1})^2 + 1/2\ (target_{o2} - out_{o2})^2$$

$$\delta E_{total} / \delta out_{o1} = -(target_{o1} - out_{o1})$$

$$\delta E_{total} / \delta out_{o1} = -(0.01 - 0.75136507) = 0.74136507$$

# Output Layer

- Next, how much does the output of o1 change with respect to its total net input?

  $out_{o1} = 1/(1+e^{(-net_{o1})})$

  $\delta out_{o1} / \delta net_{o1} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$

- Finally, how much does the total net input of o1 change with respect to w5?

  $net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$

  $\delta net_{o1} / \delta w5 = out_{h1} = 0.593269992$

# Output Layer

- Putting it all together:

$$\delta E_{total} / \delta w5 = \delta E_{total} / \delta out_{o1} * \delta out_{o1} / \delta net_{o1} * \delta net_{o1} / \delta w5$$

$$\delta E_{total} / \delta w5 = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

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

$$w5 = w5 - \eta * \delta E_{total} / \delta w5 = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

# Output Layer

- We can repeat this process to get the new weights w6, w7, and w8:

  w6 = 0.408666186

  w7 = 0.511301270

  w8 = 0.561370121

- We perform the actual updates in the neural network after we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).

# Hidden Layer

- Next, we'll continue the backwards pass by calculating new values for w1, w2, w3, and w4.

- Big picture, here's what we need to figure out:

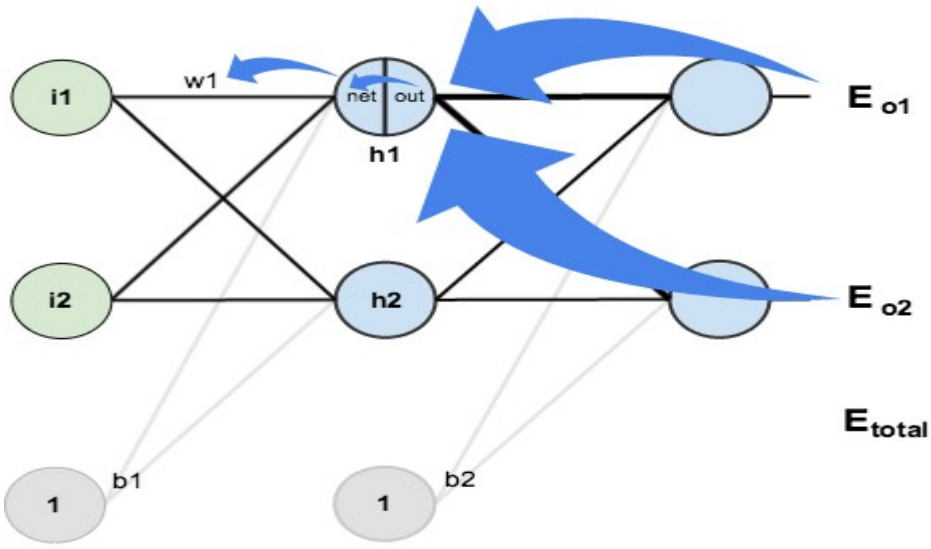
$$\delta E_{total} / \delta w1 = \delta E_{total} / \delta out_{h1} * \delta out_{h1} / \delta net_{h1} * \delta net_{h1} / \delta w1$$

# Hidden Layer

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

$$\downarrow$$

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




$E_{total} = E_{o1} + E_{o2}$

# Hidden Layer

- We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons.
- We know that $out_{h1}$ affects both $out_{o1}$ and $out_{o2}$ therefore the $\delta E_{total} / \delta out_{h1}$ needs to take into consideration its effect on the both output neurons:

$$\delta E_{total} / \delta out_{h1} = \delta E_{o1} / \delta out_{h1} + \delta E_{o2} / \delta out_{h1}$$

# Hidden Layer

- Starting with $\delta E_{o1} / \delta out_{h1}$:
  $\delta E_{o1} / \delta out_{h1} = \delta E_{o1} / \delta net_{o1} * \delta net_{o1} / \delta out_{h1}$

- We can calculate $\delta E_{o1} / \delta net_{o1}$ using values we calculated earlier:
  $\delta E_{o1} / \delta net_{o1} = \delta E_{o1} / \delta out_{o1} * \delta out_{o1} / \delta net_{o1} = 0.74136507 * 0.186815602 = 0.138498562$

- And $\delta net_{o1} / \delta out_{h1}$ is equal to w5:
  $net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$
  $\delta net_{o1} / \delta out_{h1} = w5 = 0.40$

- Plugging them in:
  $\delta E_{o1} / \delta out_{h1} = \delta E_{o1} / \delta net_{o1} * \delta net_{o1} / \delta out_{h1} = 0.138498562 * 0.40 = 0.055399425$

# Hidden Layer

- Following the same process for $\delta E_{o2} / \delta out_{h1}$, we get:

  $\delta E_{o2} / \delta out_{h1} = -0.019049119$

- Therefore:

  $\delta E_{total} / \delta out_{h1} = \delta E_{o1} / \delta out_{h1} + \delta E_{o2} / \delta out_{h1} = 0.055399425 + -0.019049119 = 0.036350306$

- Now that we have $\delta E_{total} / \delta out_{h1}$, we need to figure out $\delta out_{h1} / \delta net_{h1}$ and then $\delta net_{h1} / \delta w$ for each weight:

  $out_{h1} = 1/(1+e^{(-net_{h1})})$

  $\delta out_{h1} / \delta net_{h1} = out_{h1} (1 - out_{h1}) = 0.59326999(1 - 0.59326999 ) = 0.241300709$

# Hidden Layer

- We calculate the partial derivative of the total net input to h1 with respect to w1 the same as we did for the output neuron:

  $net_{h1} = w1 * i1 + w2 * i2 + b1 * 1$

  $\delta net_{h1} / \delta w1 = i1 = 0.05$

- Putting it all together:

  $\delta E_{total} / \delta w1 = \delta E_{total} / \delta out_{h1} * \delta out_{h1} / \delta net_{h1} * \delta net_{h1} / \delta w1$

  $\delta E_{total} / \delta w1 = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$

# Hidden Layer

- We can now update w1:

  $w1 = w1 - \eta * \delta E_{total} / \delta w1 = 0.15 - 0.5 * 0.000438568 = 0.149780716$

- Repeating this for w2, w3, and w4 :

  w2 = 0.19956143

  w3 = 0.24975114

  w4 = 0.29950229

# Backpropagation

- Finally, we've updated all of our weights!
- When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109.
- After this first round of backpropagation, the total error is now down to 0.291027924.
- It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.000035085.
- At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

# Backpropagation

- Network with two inputs, one output, one hidden layer of one neuron.
- Sigmoid activation function. No bias.
- Inputs are 0.1 and 0.5, desired output is 0.2.
- Write code for the forward pass.
- Compute the mean squared error.
- Backpropagate the error.
- Train the network.

# Backpropagation

```
import math

w1 = 0.4
w2 = 0.6
w3 = 0.1
i1 = 0.1
i2 = 0.5
out = 0.2
eta = 0.5

def sigmoid (x):
    return 1.0  / (1.0 + math.exp (-x))
```

# Backpropagation

# forward

neth = w1 * i1 + w2 * i2

outh = sigmoid (neth)

neto = w3 * outh

o = sigmoid (neto)

# Backpropagation

```
for i in range (1000):
    neth = w1 * i1 + w2 * i2
    outh = sigmoid (neth)
    neto = w3 * outh
    o = sigmoid (neto)
    err = 0.5 * (out - o) ** 2
    print (err, o)
    dw3 = (o - out) * o * (1.0 - o) * outh
    dw2 = (o - out) * o * (1.0 - o) * w3 * outh * (1.0 - outh) * i2
    dw1 = (o - out) * o * (1.0 - o) * w3 * outh * (1.0 - outh) * i1
    w3 = w3 - eta * dw3
    w2 = w2 - eta * dw2
    w1 = w1 - eta * dw1
```

# Backpropagation

Cross-entropy loss :

$C = -\Sigma_n \Sigma_i y_{in} \log o_{in}$

where $o_{in}$ is your network's output for class i, and $y_{in}$ = 1 if example n is of class i and 0 if it has some other class j.

$dC / do_{in} = -y_{in} / o_{in}$

# Backpropagation

Binary cross-entropy loss :

$$C = -\Sigma_n \Sigma_i (y_{in} \log o_{in} + (1 - y_{in}) \log (1 - o_{in}))$$

$$dC / do_{in} = -y_{in} / o_{in} + (1 - y_{in}) / (1 - o_{in})$$

# Backpropagation

Mean squared error loss :

$$C = \Sigma_n \Sigma_i (y_{in} - o_{in})^2$$

$$dC / do_{in} = -2 (y_{in} - o_{in}).$$

# Backpropagation

- For a binary classification binary cross entropy.
- For a multi-class classification softmax + cross entropy.
- For regression mean squared error.

# Backpropagation

- Train the network to have 1.0 as output with the binary cross-entropy loss.

# Backpropagation

```
# binary cross entropy
out = 1.0
for i in range (1000):
    neth = w1 * i1 + w2 * i2
    outh = sigmoid (neth)
    neto = w3 * outh
    o = sigmoid (neto)
    err = -out * math.log (o) - (1.0 - out) * math.log (1.0 - o)
    print (err, o)
    dw3 = (-out / o) * o * (1.0 - o) * outh
    dw2 = (-out / o) * o * (1.0 - o) * w3 * outh * (1.0 - outh) * i2
    dw1 = (-out / o) * o * (1.0 - o) * w3 * outh * (1.0 - outh) * i1
    w3 = w3 - eta * dw3
    w2 = w2 - eta * dw2
    w1 = w1 - eta * dw1
```

# Backpropagation

- Train a modified network with two outputs to have 1.0 and 0.0 as outputs with the categorical crossentropy loss.

# Backpropagation

```
# categorical cross entropy
out1 = 1.0
out2 = 0.0
w4 = 0.5
for i in range (1000):
    neth = w1 * i1 + w2 * i2
    outh = sigmoid (neth)
    neto1 = w3 * outh
    o1 = sigmoid (neto1)
    neto2 = w4 * outh
    o2 = sigmoid (neto2)
    err = -out1 * math.log (o1) - out2 * math.log (o2)
    print (err, o1, o2)
```

# Backpropagation

do2 = (-out2 / o2) * o2 * (1.0 - o2)

do1 = (-out1 / o1) * o1 * (1.0 - o1)

dw4 = do2 * outh

dw3 = do1 * outh

dw2 = (do1 * w3 + do2 * w4) * outh * (1.0 - outh) * i2

dw1 = (do1 * w3 + do2 * w4) * outh * (1.0 - outh) * i1

w4 = w4 - eta * dw4

w3 = w3 - eta * dw3

w2 = w2 - eta * dw2

w1 = w1 - eta * dw1

# Backpropagation

- What happens with the categorical crossentropy loss for this example ?
- Solutions ?

# Backpropagation

- Try two outputs 1 and 0 with the binary cross entropy loss.
- Try the softmax activation at the output.

# Backpropagation

```
# binary cross entropy
out1 = 1.0
out2 = 0.0
w4 = 0.5
for i in range (100000):
    neth = w1 * i1 + w2 * i2
    outh = sigmoid (neth)
    neto1 = w3 * outh
    o1 = sigmoid (neto1)
    neto2 = w4 * outh
    o2 = sigmoid (neto2)
    err = -out1 * math.log (o1) - (1.0 - out2) * math.log (1.0 - o2)
    print (err, o1, o2)
```

# Backpropagation

do2 = (1 - out2) / (1.0 - o2) * o2 * (1.0 - o2)

do1 = (-out1 / o1) * o1 * (1.0 - o1)

dw4 = do2 * outh

dw3 = do1 * outh

dw2 = (do1 * w3 + do2 * w4) * outh * (1.0 - outh) * i2

dw1 = (do1 * w3 + do2 * w4) * outh * (1.0 - outh) * i1

w4 = w4 - eta * dw4

w3 = w3 - eta * dw3

w2 = w2 - eta * dw2

w1 = w1 - eta * dw1

# Backpropagation

- Multi-class classification: $\text{label} = 2 \rightarrow \boldsymbol{y} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$
- Cross entropy: $\mathcal{L}(\boldsymbol{y}, \boldsymbol{s}) = -\sum_{i=1}^{C} y_i \cdot log(s_i)$
- Softmax: $s_i = \frac{e^{z_i}}{\sum_{l=1}^{n} e^{z_l}}, \qquad \forall i = 1, \dots, n$
- Derivative of the loss and the softmax: $\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} = \boldsymbol{s} - \boldsymbol{y}$

# Jacobian of the Softmax

$$J_{softmax} = \begin{pmatrix} \frac{\partial s_1}{\partial z_1} & \frac{\partial s_1}{\partial z_2} & \cdots & \frac{\partial s_1}{\partial z_n} \\ \frac{\partial s_2}{\partial z_1} & \frac{\partial s_2}{\partial z_2} & \cdots & \frac{\partial s_2}{\partial z_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial s_n}{\partial z_1} & \frac{\partial s_n}{\partial z_2} & \cdots & \frac{\partial s_n}{\partial z_n} \end{pmatrix}$$

$$s_i = \frac{e^{z_i}}{\sum_{l=1}^{n} e^{z_l}}, \qquad \forall i = 1, \ldots, n$$

# Jacobian of the Softmax

$$\frac{\partial}{\partial z_j} log(s_i) = \frac{1}{s_i} \cdot \frac{\partial s_i}{\partial z_j}$$

$$\frac{\partial s_i}{\partial z_j} = s_i \cdot \frac{\partial}{\partial z_j} log(s_i)$$

$$log\, s_i = log\left(\frac{e^{z_i}}{\sum_{l=1}^{n} e^{z_l}}\right) = z_i - log\left(\sum_{l=1}^{n} e^{z_l}\right)$$

$$\frac{\partial}{\partial z_j} log\, s_i = \frac{\partial z_i}{\partial z_j} - \frac{\partial}{\partial z_j} log\left(\sum_{l=1}^{n} e^{z_l}\right)$$

# Jacobian of the Softmax

$$\frac{\partial}{\partial z_j} log\, s_i = \frac{\partial z_i}{\partial z_j} - \frac{\partial}{\partial z_j} log\left(\sum_{l=1}^{n} e^{z_l}\right)$$

$$\frac{\partial z_i}{\partial z_j} = \begin{cases} 1, & if\ i = j \\ 0, & otherwise \end{cases}$$

$$\frac{\partial}{\partial z_j} log\, s_i = 1\{i = j\} - \frac{1}{\sum_{l=1}^{n} e^{z_l}} \cdot \left(\frac{\partial}{\partial z_j} \sum_{l=1}^{n} e^{z_l}\right)$$

# Jacobian of the Softmax

$$\frac{\partial}{\partial z_j}\sum_{l=1}^{n} e^{z_l} = \frac{\partial}{\partial z_j}\left[e^{z_1} + e^{z_2} + \cdots + e^{z_j} + \cdots + e^{z_n}\right] = \frac{\partial}{\partial z_j}\left[e^{z_j}\right] = e^{z_j}$$

$$\frac{\partial}{\partial z_j} log\, s_i = 1\{i = j\} - \frac{e^{z_j}}{\sum_{l=1}^{n} e^{z_l}} = 1\{i = j\} - s_j$$

$$\frac{\partial s_i}{\partial z_j} = s_i \cdot \frac{\partial}{\partial z_j} log(s_i) = s_i \cdot \left(1\{i = j\} - s_j\right)$$

# Jacobian of the Softmax

$$J_{softmax} = \begin{pmatrix} s_1 \cdot (1 - s_1) & -s_1 \cdot s_2 & -s_1 \cdot s_3 & -s_1 \cdot s_4 \\ -s_2 \cdot s_1 & s_2 \cdot (1 - s_2) & -s_2 \cdot s_3 & -s_2 \cdot s_4 \\ -s_3 \cdot s_1 & -s_3 \cdot s_2 & s_3 \cdot (1 - s_3) & -s_3 \cdot s_4 \\ -s_4 \cdot s_1 & -s_4 \cdot s_2 & -s_4 \cdot s_3 & s_4 \cdot (1 - s_4) \end{pmatrix}$$

# Backpropagation

$$\frac{\partial \mathcal{L}}{\partial z_j} = -\frac{\partial}{\partial z_j}\sum_{i=1}^{C} y_i \cdot log(s_i) = -\sum_{i=1}^{C} y_i \cdot \frac{\partial}{\partial z_j} log(s_i) = -\sum_{i=1}^{C} \frac{y_i}{s_i} \cdot \frac{\partial s_i}{\partial z_j}$$

$$\frac{\partial \mathcal{L}}{\partial z_j} = -\sum_{i=1}^{C} \frac{y_i}{s_i} \cdot s_i \cdot \left(1\{i = j\} - s_j\right) = -\sum_{i=1}^{C} y_i \cdot \left(1\{i = j\} - s_j\right)$$

$$\frac{\partial \mathcal{L}}{\partial z_j} = \left[-\sum_{i \neq j} y_i \cdot \left(1\{i = j\} - s_j\right)\right] - y_j \cdot \left(1\{j = j\} - s_j\right)$$

$$\frac{\partial \mathcal{L}}{\partial z_j} = \left[\sum_{i \neq j} y_i \cdot s_j\right] - y_j \cdot \left(1 - s_j\right) = \left[\sum_{i \neq j} y_i \cdot s_j\right] - y_j + y_j \cdot s_j$$

# Backpropagation

$$\frac{\partial \mathcal{L}}{\partial z_j} = \sum_{i=1}^{C} y_i \cdot s_j - y_j = s_j \cdot \sum_{i=1}^{C} y_i - y_j = s_j - y_j$$

# Keras

# Keras

- Keras is a high-level neural networks API, written in Python and integrated in TensorFlow. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

- Use Keras if you need a deep learning library that:

  Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).

  Supports both convolutional networks and recurrent networks, as well as combinations of the two.

  Runs seamlessly on CPU and GPU.

# Keras

- The core data structure of Keras is a model, a way to organize layers. The simplest type of model is the Sequential model, a linear stack of layers.

```
import tensorflow as tf
from tensorflow import keras

from tensorflow.keras import Sequential
model = Sequential()
```

# Keras

- In order to define a network you have to import the libraries for defining the layers and the libraries for the training algorithms :

  from tensorflow.keras.layers import Dense, Activation

  from tensorflow.keras.optimizers import SGD

# Keras

- Example

```
# as first layer in a sequential model:
model = Sequential()
model.add(Dense(32, input_shape=(16,)))
# now the model will take as input arrays of shape (*, 16)
# and output arrays of shape (*, 32)

# after the first layer, you don't need to specify
# the size of the input anymore:
model.add(Dense(32))
```

# Keras

Stacking layers is as easy as .add():

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(units=64, input_dim=100))
model.add(Activation(‘tanh'))
model.add(Dense(units=10))
model.add(Activation('softmax'))
```

# Keras

- Input and labels are represented as arrays :

```
import numpy as np

X = np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])
y = np.array([[0.],[1.],[1.],[0.]])
```

# Keras

```
sgd = SGD(lr=0.1)

model.compile(loss='binary_crossentropy', optimizer=sgd)

model.fit(X, y, verbose=1, batch_size=1, epochs=1000)

print(model.predict(X))
```

# Practical Work

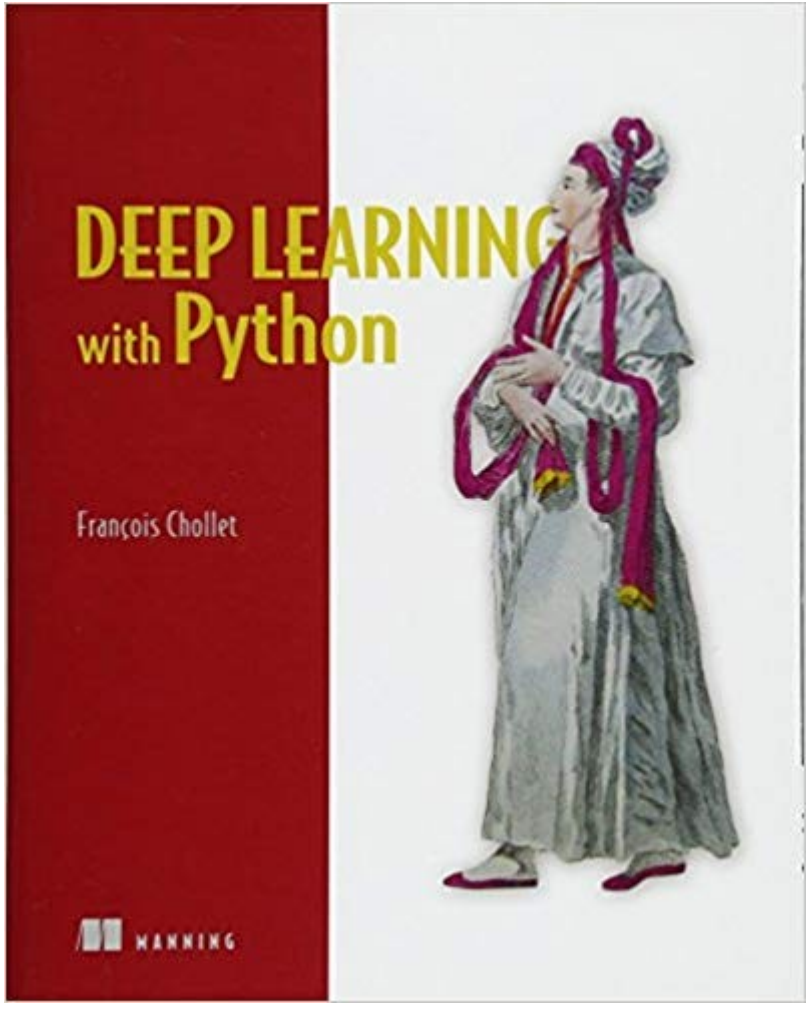
- Implement a two layers XOR network (one hidden layer).
- Make it learn the XOR function with two inputs and one output.

# XOR

```
import tensorflow as tf
from tensorflow import keras

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.optimizers import SGD

import numpy as np
X = np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])
y = np.array([[0.],[1.],[1.],[0.]])
```

# XOR

```
model = Sequential()
model.add(Dense(8, input_dim=2))
model.add(Activation('tanh'))
model.add(Dense(1))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.1)
model.compile(loss='mse', optimizer=sgd)
model.fit(X, y, verbose=1, batch_size=1, epochs=1000)
print(model.predict(X))
```


DEEP LEARNING
with Python
François Chollet
MANNING

# MNIST

The problem we're trying to solve here is to classify grayscale images of handwritten digits (28 × 28 pixels) into their 10 categories (0 through 9). We'll use the MNIST dataset, a classic in the machine-learning community, which has been around almost as long as the field itself and has been intensively studied. It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s. You can think of "solving" MNIST as the "hello world" of deep learning—it's what you do to verify that your algorithms are working as expected. As you become a machine-learning practitioner, you'll see MNIST come up over and over again in scientific papers, blog posts, and so on. You can see some MNIST samples in figure 2.1.

### Classes and labels

In machine learning, a *category* in a classification problem is called a *class*. Data points are called *samples*. The class associated with a specific sample is called a *label*.

Figure 2.1 MNIST sample digits

# MNIST

- Loading the MNIST data :

```
from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

# Exercise

Train a small dense network on the MNIST data :

- Prepare the data :
  - Inputs = vectors of real numbers (between 0.0 and 1.0) of size 28*28
  - Labels = vectors of real numbers of size 10 (nine 0 and a 1 at the index of the label)
- Define the network :
  - Fully connected network with 28*28 inputs and 10 outputs
- Define the loss and the optimizer
- Train the network
- Test the network
- Print an image in the test set and the predicted class

# MNIST

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

# MNIST

- Preparing the data :

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
from tensorflow.keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

# MNIST

- Defining the network :

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Activation


network = Sequential()
network.add(Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(Dense(10, activation='softmax'))
```

# MNIST

- Defining the optimizer and the loss :

```
network.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

- Training the network :

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

- Testing the network :

```
test_loss, test_acc = network.evaluate(test_images, test_labels)
print('test_acc:', test_acc)
```

# MNIST

- Predicting the class of an example :

```
import matplotlib.pyplot as plt
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
plt.imshow(test_images [0])
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
img = test_images [0].reshape ((1, 28*28))
print (network.predict(img))
```

# Binary Classification

# Binary Classification

Two-class classification, or binary classification, may be the most widely applied kind of machine-learning problem. In this example, you'll learn to classify movie reviews as positive or negative, based on the text content of the reviews.

### *The IMDB dataset*

You'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)
```

# Binary Classification

- The argument num_words = 10000 means you'll keep only the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows you to work with vector data of a manageable size.
- The variables train_data and test_data are lists of reviews; each review is a list of word indices (encoding a sequence of words).
- train_labels and test_labels are lists of 0s and 1s, where 0 stands for negative and 1 stands for positive :

  train_data[0]

  [1, 14, 22, 16, ... 178, 32]

  train_labels[0]

  1

# Exercise : Preparing the data

- You can’t feed lists of integers into a neural network. You have to turn your lists into tensors :
  - One-hot encode your lists to turn them into vectors of 0s and 1s.
  - This would mean, for instance, turning the sequence [3, 5] into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s.
  - Then you could use as the first layer in your network a dense layer, capable of handling floating-point vector data.

# IMDB

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

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i in range (len (sequences)):
        for j in range (len (sequences [i])):
            results [i] [sequences [i] [j]] = 1.
    return results

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

- You should also convert your labels from integer to numeric, which is straightforward:

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

# View Reviews

```
word_index = imdb.get_word_index()

reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])

decoded_review = ' '.join(
    [reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

# Exercise

Train a small dense network on the IMDB data :

- Define the network
- Define the loss and the optimizer
- Define a validation set
- Train the network using a validation set

# IMDB

- Defining the network :

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(16, activation='relu', input_shape=(10000,)))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

# IMDB

- Defining a validation set :

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

- Training with a validation set :

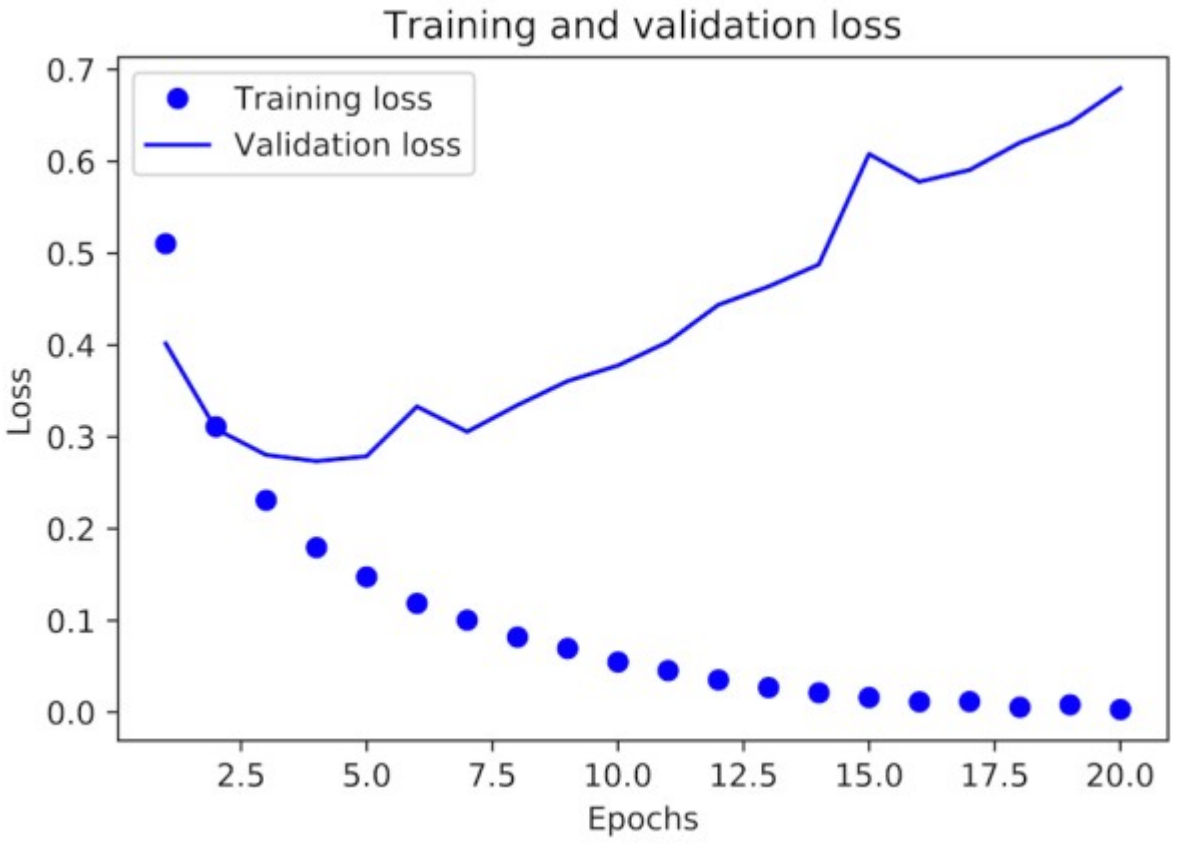
```
model.compile (optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit (partial_x_train, partial_y_train,epochs=20, batch_size=512, validation_data=(x_val, y_val))
```

# IMDB

- Visualize the training loss :

```
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

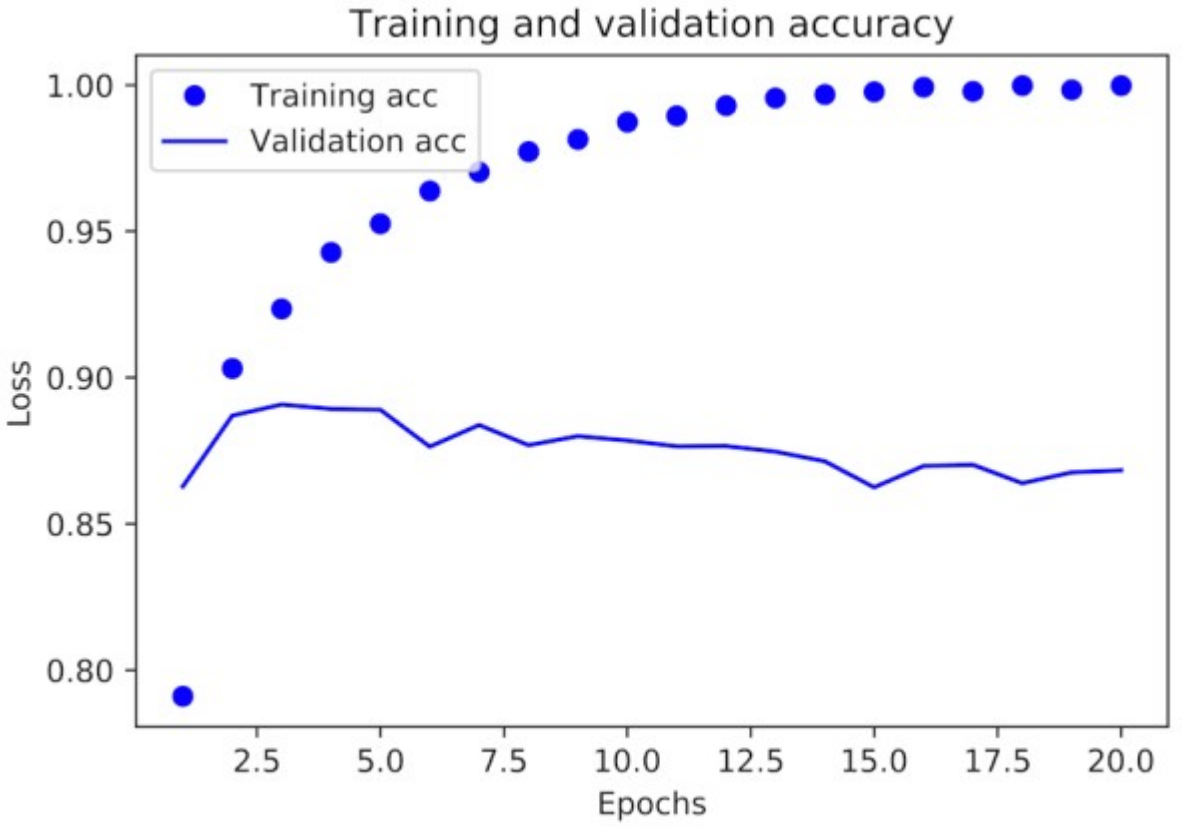

Training and validation loss
Training loss
Validation loss
Loss
Epochs
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

# IMDB

- Visualize the training accuracy:

```
plt.clf() #Clears the figure
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


Training and validation accuracy
Training acc
Validation acc
Loss
1.00
0.95
0.90
0.85
0.80
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs

# Weight Regularization

# Weight Regularization

- You may be familiar with the principle of Occam's razor : given two explanations for something, the explanation most likely to be correct is the simplest one—the one that makes fewer assumptions. This idea also applies to the models learned by neural networks: given some training data and a network architecture, multiple sets of weight values (multiple models) could explain the data. Simpler models are less likely to overfit than complex ones.

- A simple model in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters). Thus, a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to take only small values, which makes the distribution of weight values more regular. This is called weight regularization, and it's done by adding to the loss function of the network a cost associated with having large weights.

# Weight Regularization

- L2 regularization : The cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights). L2 regularization is also called weight decay in the context of neural networks.
- In Keras, weight regularization is added by passing weight regularizer instances to layers as keyword arguments :

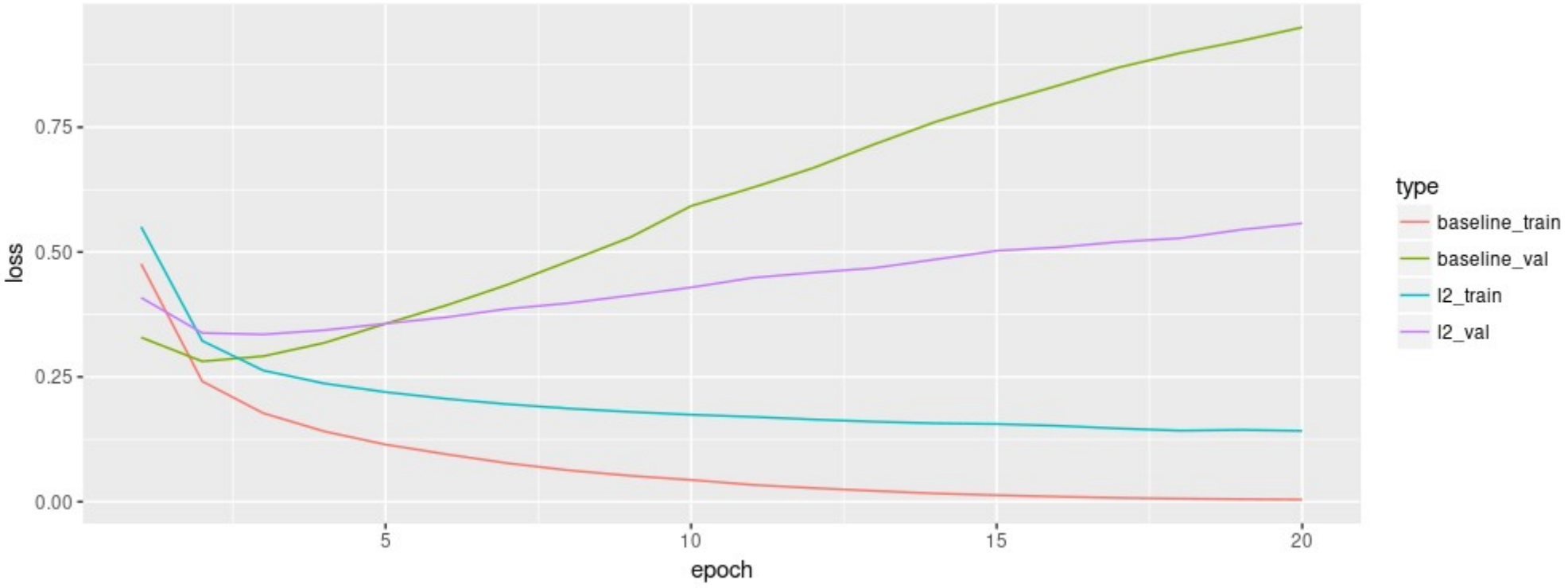
```
from tensorflow.keras import regularizers

model = Sequential()

model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001), activation='relu',
                                    input_shape=(10000,)))

model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001), activation='relu'))

model.add(layers.Dense(1, activation='sigmoid'))
```

- regularizer_l2(0.001) means every coefficient in the weight matrix of the layer will add 0.001 * weight_coefficient_value to the total loss of the network. Note that because this penalty is only added at training time, the loss for this network will be much higher at training time than at test time.

# Exercise

Compare for the IMDB network the effect of weight regularization on the evolution of training and test errors and accuracies.

# Regularization l2 IMDB



# Dropout

# Dropout

- Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Geoff Hinton and his students at the University of Toronto.
- Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training.
- Let’s say a given layer would normally return a vector [0.2, 0.5, 1.3, 0.8, 1.1] for a given input sample during training. After applying dropout, this vector will have a few zero entries distributed at random: for example, [0, 0.5, 1.3, 0, 1.1] .
- The dropout rate is the fraction of the features that are zeroed out; it’s usually set between 0.2 and 0.5.
- At test time, no units are dropped out; instead, the layer’s output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time.

# Dropout

- Consider a matrix containing the output of a layer, layer_output , of shape (batch_size, features) .
- At training time, we zero out at random a fraction of the values in the matrix:

  layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
- At test time, we scale down the output by the dropout rate. Here, we scale by 0.5 (because we previously dropped half the units):

  layer_output *= 0.5
- Note that this process can be implemented by doing both operations at training time and leaving the output unchanged at test time, which is often the way it’s implemented in practice :

  layer_output *= np.random.randint(0, high=2, size=layer_output.shape)

  layer_output /= 0.5

# Tips For Using Dropout

- Generally, use a small dropout value of 20%-50% of neurons with 20% providing a good starting point. A probability too low has minimal effect and a value too high results in under-learning by the network.

- Use a larger network. You are likely to get better performance when dropout is used on a larger network, giving the model more of an opportunity to learn independent representations.

# Dropout

- Dropout

  tensorflow.keras.layers.Dropout(rate, noise_shape=None, seed=None)

- Applies Dropout to the input.

  Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting.

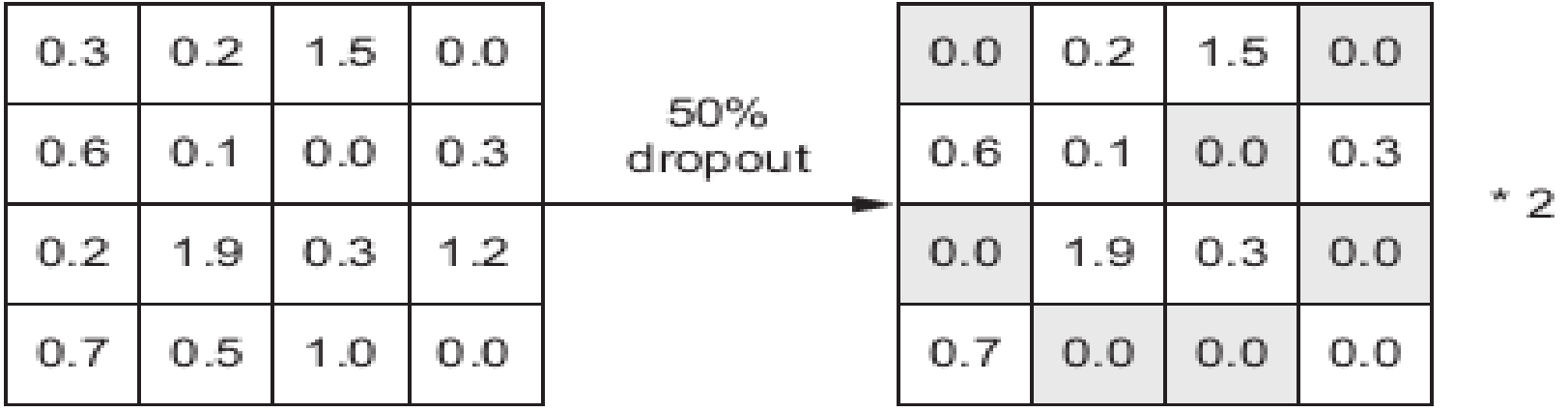
- Arguments

  rate: float between 0 and 1. Fraction of the input units to drop.

  noise_shape: 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input. For instance, if your inputs have shape (batch_size, timesteps, features) and you want the dropout mask to be the same for all timesteps, you can use noise_shape=(batch_size, 1, features).

  seed: A Python integer to use as random seed.

# Dropout


0.3 0.2 1.5 0.0
0.6 0.1 0.0 0.3
0.2 1.9 0.3 1.2
0.7 0.5 1.0 0.0
50%
dropout
0.0 0.2 1.5 0.0
0.6 0.1 0.0 0.3
0.0 1.9 0.3 0.0
0.7 0.0 0.0 0.0
* 2

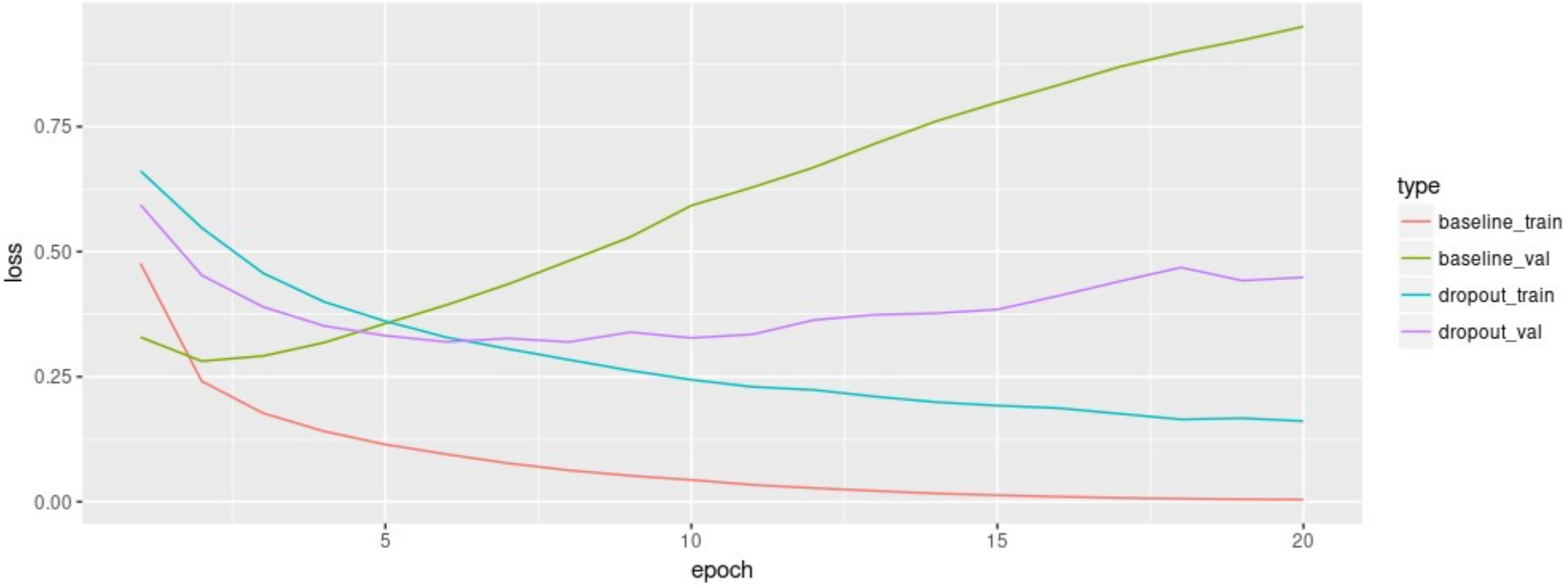
# Dropout

- Add two dropout layers in the IMDB network to see how well they do at reducing overfitting.

# Dropout

```
model = models.Sequential()
model.add(layers.Dense(16,activation='relu',
                                    input_shape=(10000,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

# Dropout


0.75
0.50
0.25
0.00
loss
5
10
15
20
epoch
type
baseline_train
baseline_val
dropout_train
dropout_val

# Avoid Overfitting

- To recap, these are the most common ways to prevent overfitting in neural networks:
  - Get more training data.
  - Reduce the capacity of the network.
  - Add weight regularization.
  - Add dropout.

# Multiclass Classification

# Multiclass Classification

- You'll work with the Reuters dataset, a set of short newswires and their topics, published by Reuters in 1986. It's a simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set.

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

# Exercise

Train a small dense network on the Reuters data :

- Encode the data
- Define the network
- Define the loss and the optimizer
- Define a validation set
- Train the network using a validation set

# Reuters

- Encoding the data :

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

from tensorflow.keras.utils import to_categorical
one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

# Reuters

- Defining the network :

```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

# Reuters

- Defining the optimizer and the loss :

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

- Defining a validation set :

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]
y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

- Training with a validation set :

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

# Reuters

- Visualize the training :

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

# Reuters

- Further experiments :
- Train on less epochs.
- Try using larger or smaller layers: 32 units, 128 units, and so on.
- The network used two hidden layers. Now try using a single hidden layer, or three hidden layers.

# Multiclass classification

Here’s what you should take away from this example:

- If you’re trying to classify data points among N classes, your network should end with a dense layer of size N.
- In a single-label, multiclass classification problem, your network should end with a softmax activation so that it will output a probability distribution over the N output classes.
- Categorical crossentropy is almost always the loss function you should use for such problems. It minimizes the distance between the probability distributions output by the network and the true distribution of the targets.
- Encoding the labels via categorical encoding (also known as one-hot encoding) and using categorical_crossentropy as a loss function
- If you need to classify data into a large number of categories, you should avoid creating information bottlenecks in your network due to intermediate layers that are too small.

# Regression

# Regression

- The Boston Housing Price dataset
- Predict the median price of homes in a given Boston suburb in the mid-1970s, given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on. The dataset you'll use has an interesting difference from the two previous examples. It has relatively few data points: only 506, split between 404 training samples and 102 test samples. And each feature in the input data (for example, the crime rate) has a different scale. For instance, some values are proportions, which take values between 0 and 1; others take values between 1 and 12, others between 0 and 100, and so on.

```
from tensorflow.keras.datasets import boston_housing
(train_data, train_targets), (test_data, test_targets) =
 boston_housing.load_data()
```

# Exercise

Train a small dense network on the Boston Housing data :

- Encode the data
- Define the network
- Define the loss and the optimizer
- Define a validation set
- Train the network using a validation set

# Regression

- Preparing the data

  It would be problematic to feed into a neural network values that all take wildly different ranges. The network might be able to automatically adapt to such heterogeneous data, but it would definitely make learning more difficult. A widespread best practice to deal with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), you subtract the mean of the feature and divide by the standard deviation, so that the feature is centered around 0 and has a unit standard deviation. This is easily done in Numpy.

  ```
  mean = train_data.mean(axis=0)
  train_data -= mean
  std = train_data.std(axis=0)
  train_data /= std
  test_data -= mean
  test_data /= std
  ```

# Regression

- Building your network

Because so few samples are available, you'll use a very small network with two hidden layers, each with 64 units. In general, the less training data you have, the worse overfitting will be, and using a small network is one way to mitigate overfitting.

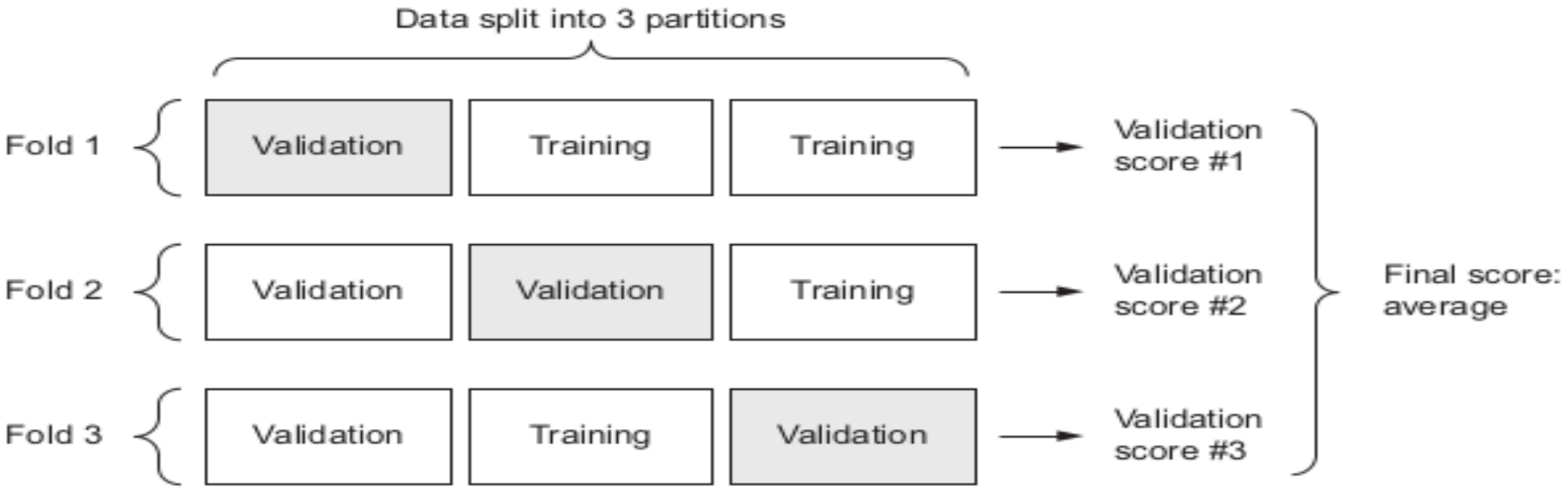
```
from tensorflow.keras import models
from tensorflow.keras import layers
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                           input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

# Regression

- In order to find the best hyperparameters (epochs, number of hidden layers, number of neurons,…) we will evaluate the hyperparameters with cross validation.
- The goal is to find hyperparameters that minimize the cross validation error.
- Once the hyperparameters have been chosen, the network with these hyperparameters is initialized and trained on the whole training set.

# Regression

- 3-fold cross-validation :



# Regression

```
num_epochs = 50
model = build_model()

num_val_samples = len(train_data) // 4

val_data = train_data[:num_val_samples]
val_targets = train_targets[:num_val_samples]
partial_train_data = train_data[num_val_samples:]
partial_train_targets = train_targets[num_val_samples:]

history = model.fit(partial_train_data, partial_train_targets,
                    validation_data=(val_data, val_targets),
                    epochs=num_epochs, batch_size=10, verbose=0)
```

# Regression

```
import numpy as np
k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print('processing fold #', i)
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
    partial_train_data = np.concatenate([train_data[:i * num_val_samples],train_data[(i + 1) * num_val_samples:]], axis=0)
    partial_train_targets = np.concatenate([train_targets[:i * num_val_samples],
                                            train_targets[(i + 1) * num_val_samples:]], axis=0)
    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

# Regression

```
num_epochs = 50
all_mae_histories = []
for i in range(k):
    print('processing fold #', i)
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
    partial_train_data = np.concatenate ([train_data[:i * num_val_samples],
                                          train_data[(i + 1) * num_val_samples:]], axis=0)
    partial_train_targets = np.concatenate([train_targets[:i * num_val_samples],
                                            train_targets[(i + 1) * num_val_samples:]], axis=0)
    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=1, verbose=0)
    mae_history = history.history['val_mae']
    all_mae_histories.append(mae_history)
```

# Regression

```
average_mae_history = [
np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]

import matplotlib.pyplot as plt
plt.plot(range(1, len(average_mae_history) + 1),
         average_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```

# Regression

- Once you’re finished tuning other parameters of the model (in addition to the number of epochs, you could also adjust the size of the hidden layers), you can train a final production model on all of the training data, with the best parameters, and then look at its performance on the test data.

```
model = build_model()

model.fit(train_data, train_targets,
          epochs=80, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

# Regression

Here’s what you should take away from this example:

- Regression is done using different loss functions than classification. Mean squared error ( MSE ) is a loss function commonly used for regression.
- Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally, the concept of accuracy doesn’t apply for regression. A common regression metric is mean absolute error ( MAE ).
- When features in the input data have values in different ranges, each feature should be scaled independently as a preprocessing step.
- When there is little data available, using K-fold validation is a great way to reliably evaluate a model.
- When little training data is available, it’s preferable to use a small network with few hidden layers (typically only one or two), in order to avoid severe overfitting.

# What we have learned

# What we have learned

- Choosing the right last-layer activation and loss function for your model :

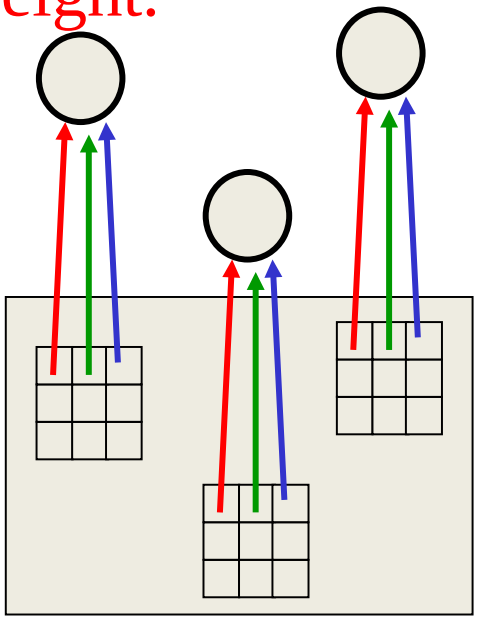
| Problem type | Last-layer activation | Loss function |
|---|---|---|
| Binary classification | `sigmoid` | `binary_crossentropy` |
| Multiclass, single-label classification | `softmax` | `categorical_crossentropy` |
| Multiclass, multilabel classification | `sigmoid` | `binary_crossentropy` |
| Regression to arbitrary values | None | `mse` |
| Regression to values between 0 and 1 | `sigmoid` | `mse` or `binary_crossentropy` |

# Convolutional Neural Networks

# The replicated feature approach
## (currently the dominant approach for neural networks)

- Use many different copies of the same feature detector with different positions.
  - Could also replicate across scale and orientation (tricky and expensive)
  - Replication greatly reduces the number of free parameters to be learned.
- Use several different feature types, each with its own map of replicated detectors.
  - Allows each patch of image to be represented in several ways.

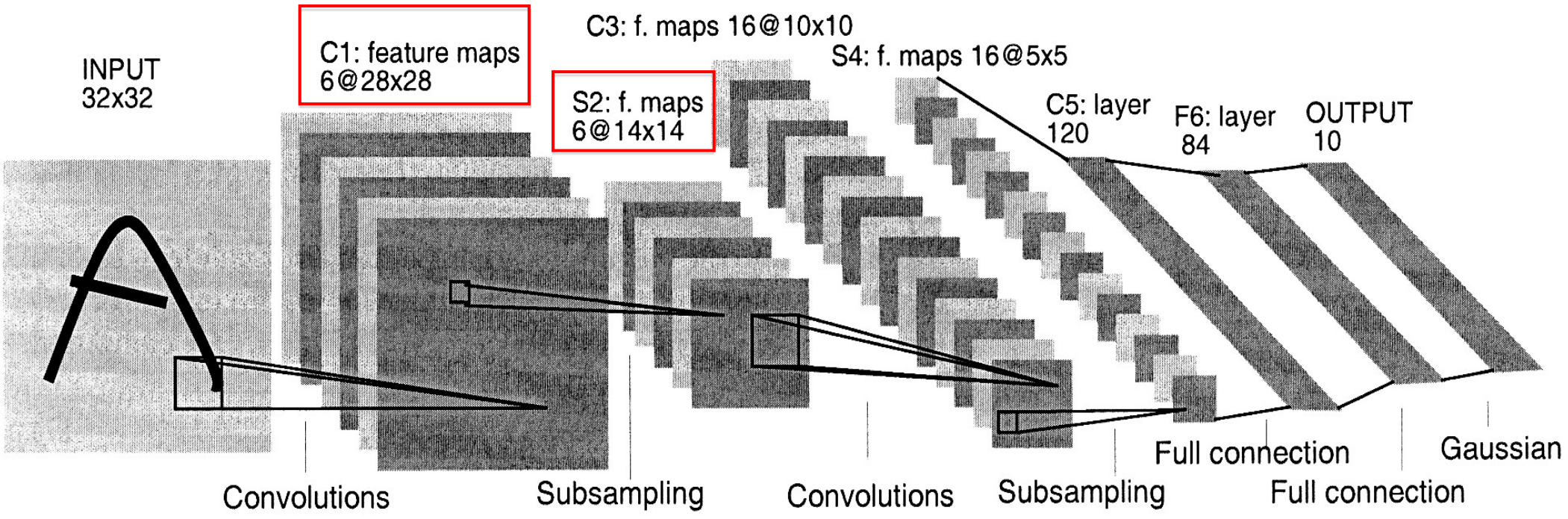
The red connections all have the same weight.



# Le Net

- Yann LeCun and his collaborators developed a really good recognizer for handwritten digits by using backpropagation in a feedforward net with:
  - Many hidden layers
  - Many maps of replicated units in each layer.
  - Pooling of the outputs of nearby replicated units.
  - A wide net that can cope with several characters at once even if they overlap.
  - A clever way of training a complete system, not just a recognizer.
- This net was used for reading ~10% of the checks in North America.
- Look the impressive demos of LENET at http://yann.lecun.com

# The architecture of LeNet5




# The 82 errors made by LeNet5

Notice that most of the errors are cases that people find quite easy.

The human error rate is probably 20 to 30 errors but nobody has had the patience to measure it.

# The ILSVRC-2012 competition on ImageNet

- The dataset has 1.2 million high-resolution training images.
- The classification task:
  - Get the “correct” class in your top 5 bets. There are 1000 classes.
- The localization task:
  - For each bet, put a box around the object. Your box must have at least 50% overlap with the correct box.

- Some of the best existing computer vision methods were tried on this dataset by leading computer vision groups from Oxford, INRIA, XRCE, …
  - Computer vision systems use complicated multi-stage systems.
  - The early stages are typically hand-tuned by optimizing a few parameters.

# Examples from the test set (with the network's guesses)

cheetah

cheetah
leopard
snow leopard
Egyptian cat

bullet train

bullet train
passenger car
subway train
electric locomotive

hand glass

scissors
hand glass
frying pan
stethoscope

# Error rates on the ILSVRC-2012 competition

| | classification | classification &localization |
|---|---|---|
| • University of Toronto (Alex Krizhevsky) | • 16.4% | 34.1% |
| • University of Tokyo | • 26.1% | 53.6% |
| • Oxford University Computer Vision Group | • 26.9% | 50.0% |
| • INRIA (French national research institute in CS) + XRCE (Xerox Research Center Europe) | • 27.0% | |
| • University of Amsterdam | • 29.5% | |

# A neural network for ImageNet

- Alex Krizhevsky (NIPS 2012) developed a very deep convolutional neural net of the type pioneered by Yann Le Cun. Its architecture was:
  - 7 hidden layers not counting some max pooling layers.
  - The early layers were convolutional.
  - The last two layers were globally connected.
- The activation functions were:
  - –Rectified linear units in every hidden layer. These train much faster and are more expressive than logistic units.
  - –Competitive normalization to suppress hidden activities when nearby units have stronger activities. This helps with variations in intensity.

# Tricks that significantly improve generalization

- Train on random 224x224 patches from the 256x256 images to get more data. Also use left-right reflections of the images.
  - At test time, combine the opinions from ten different patches: The four 224x224 corner patches plus the central 224x224 patch plus the reflections of those five patches.
- Use “dropout” to regularize the weights in the globally connected layers (which contain most of the parameters).
  - Dropout means that half of the hidden units in a layer are randomly removed for each training example.
  - This stops hidden units from relying too much on other hidden units.

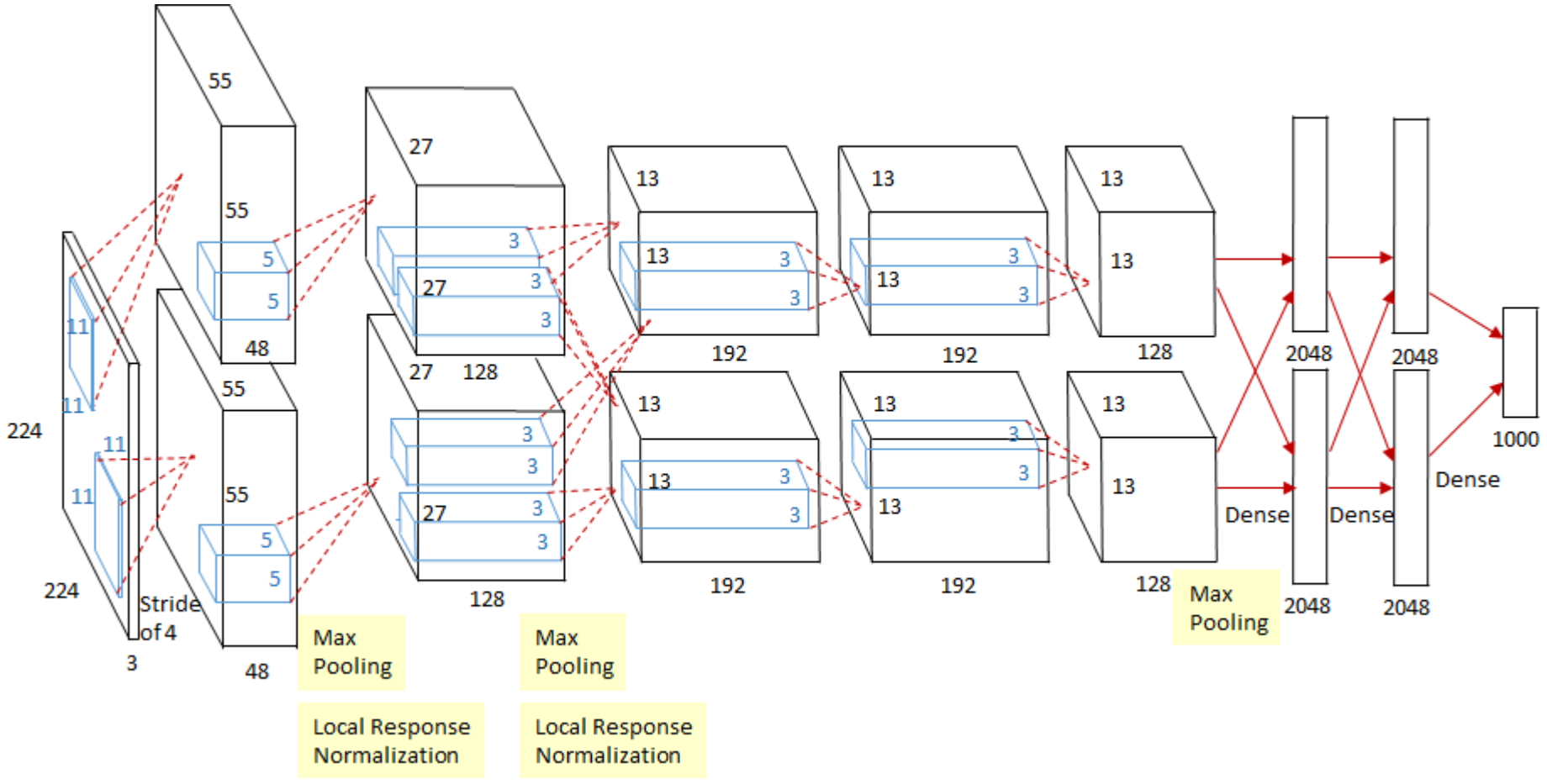
| mite | container ship | motor scooter | leopard |
|---|---|---|---|
| mite | container ship | motor scooter | leopard |
| black widow | lifeboat | go-kart | jaguar |
| cockroach | amphibian | moped | cheetah |
| tick | fireboat | bumper car | snow leopard |
| starfish | drilling platform | golfcart | Egyptian cat |

| grille | mushroom | cherry | Madagascar cat |
|---|---|---|---|
| convertible | agaric | dalmatian | squirrel monkey |
| grille | mushroom | grape | spider monkey |
| pickup | jelly fungus | elderberry | titi |
| beach wagon | gill fungus | ffordshire bullterrier | indri |
| fire engine | dead-man's-fingers | currant | howler monkey |

Some more examples of how well the deep net works for object recognition.

# The hardware required for Alex's net

- He uses a very efficient implementation of convolutional nets on two Nvidia GTX 580 Graphics Processor Units (over 1000 fast little cores)
  - GPUs are very good for matrix-matrix multiplies.
  - GPUs have very high bandwidth to memory.
  - This allows him to train the network in a week.
  - It also makes it quick to combine results from 10 patches at test time.
- We can spread a network over many cores if we can communicate the states fast enough.
- As cores get cheaper and datasets get bigger, big neural nets will improve faster than old-fashioned (*i.e.* pre Oct 2012) computer vision systems.


55
55
48
55
55
48
224
224
11
11
11
11
5
5
5
5
3
Stride of 4
27
27
27
128
128
3
3
3
3
3
3
Max Pooling
Local Response Normalization
Max Pooling
Local Response Normalization
13
13
13
13
13
13
13
13
13
13
13
13
192
192
128
192
192
128
Max Pooling
2048
2048
2048
2048
Dense
Dense
Dense
1000

# Convolutional Networks

- Conv2D

  keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

  2D convolution layer (e.g. spatial convolution over images).

  This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If use_bias is True, a bias vector is created and added to the outputs. Finally, if activation is not None, it is applied to the outputs as well.

  When using this layer as the first layer in a model, provide the keyword argument input_shape (tuple of integers, does not include the sample axis), e.g. input_shape=(128, 128, 3) for 128x128 RGB pictures in data_format="channels_last".

# Convolutional Networks

- Input shape

  4D tensor with shape: (samples, rows, cols, channels)

- Output shape

  4D tensor with shape: (samples, new_rows, new_cols, filters)
  rows and cols values might have changed due to padding.

# Convolutional Networks

- ReLU layer :
- ReLU is the abbreviation of Rectified Linear Units. This is a layer of neurons that applies the non-saturating activation function f (x) = max (0, x).
- Compared to other functions the usage of ReLU is preferable, because it results in the neural network training several times faster, without making a significant difference to generalisation accuracy.
- activation = ‘relu’ in the layer parameters.

# Convolutional Networks

- Softmax :
- Applies the Softmax function to an n-dimensional input Tensor, rescaling them so that the elements of the n-dimensional output Tensor lie in the range (0,1) and sum to 1.
- Softmax is defined as $\exp(x_i) / \Sigma \exp(x_j)$
- activation = ‘softmax‘ in the layer parameters.

# Convolutional Networks

- Reshape

  keras.layers.Reshape(target_shape)

  Reshapes an output to a certain shape.

- Arguments

  target_shape: target shape. Tuple of integers, does not include the samples dimension (batch size).

- Input shape

  Arbitrary, although all dimensions in the input shaped must be fixed. Use the keyword argument input_shape (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

- Output shape

  (batch_size,) + target_shape

# Convolutional Networks

- Example

```
# as first layer in a Sequential model
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
# now: model.output_shape == (None, 3, 4)
# note: `None` is the batch dimension

# as intermediate layer in a Sequential model
model.add(Reshape((6, 2)))
# now: model.output_shape == (None, 6, 2)

# also supports shape inference using `-1` as dimension
model.add(Reshape((-1, 2, 2)))
# now: model.output_shape == (None, 3, 2, 2)
```

# Convolutional Networks

- Flatten

  keras.layers.core.Flatten()

  Flattens the input. Does not affect the batch size.

- Example

```
model = Sequential()
model.add(Convolution2D(64, 3, 3,
        border_mode='same',
        input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

# MNIST

```
from keras.datasets import mnist

from keras.layers import Dense, Flatten
from keras.layers import Conv2D

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

# Practical Work

- Implement a convolutional network for MNIST.
- 3x3 filters
- 32 planes for the first layer
- 64 planes for the second layer
- Fully connected layer with 128 neurons
- 10 classes for the output layer
- ReLU
- Softmax

# MNIST

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

x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
input_shape = (28, 28, 1)

# normalisation of the inputs
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# convert class vectors to binary class matrices : 3 becomes [0,0,0,1,0,0,0,0,0 0]
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
```

# MNIST

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

# MNIST

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=’sgd’,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=512,
          epochs=10,
          verbose=1,
          validation_data=(x_test, y_test))

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

# Max Pooling

# Max Pooling

- Max pooling is a sample-based discretization process. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing it's dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned.

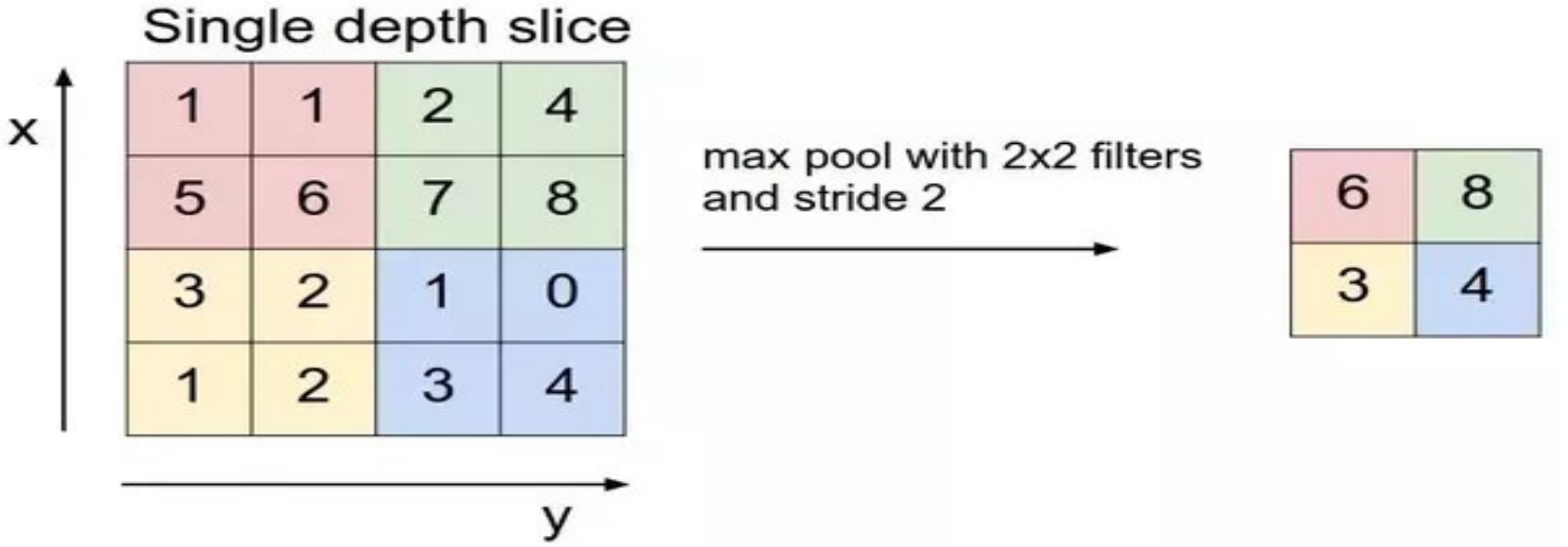
- This is done to in part to help over-fitting by providing an abstracted form of the representation. As well, it reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation.

- Max pooling is done by applying a max filter to non-overlapping sub-regions of the initial representation.

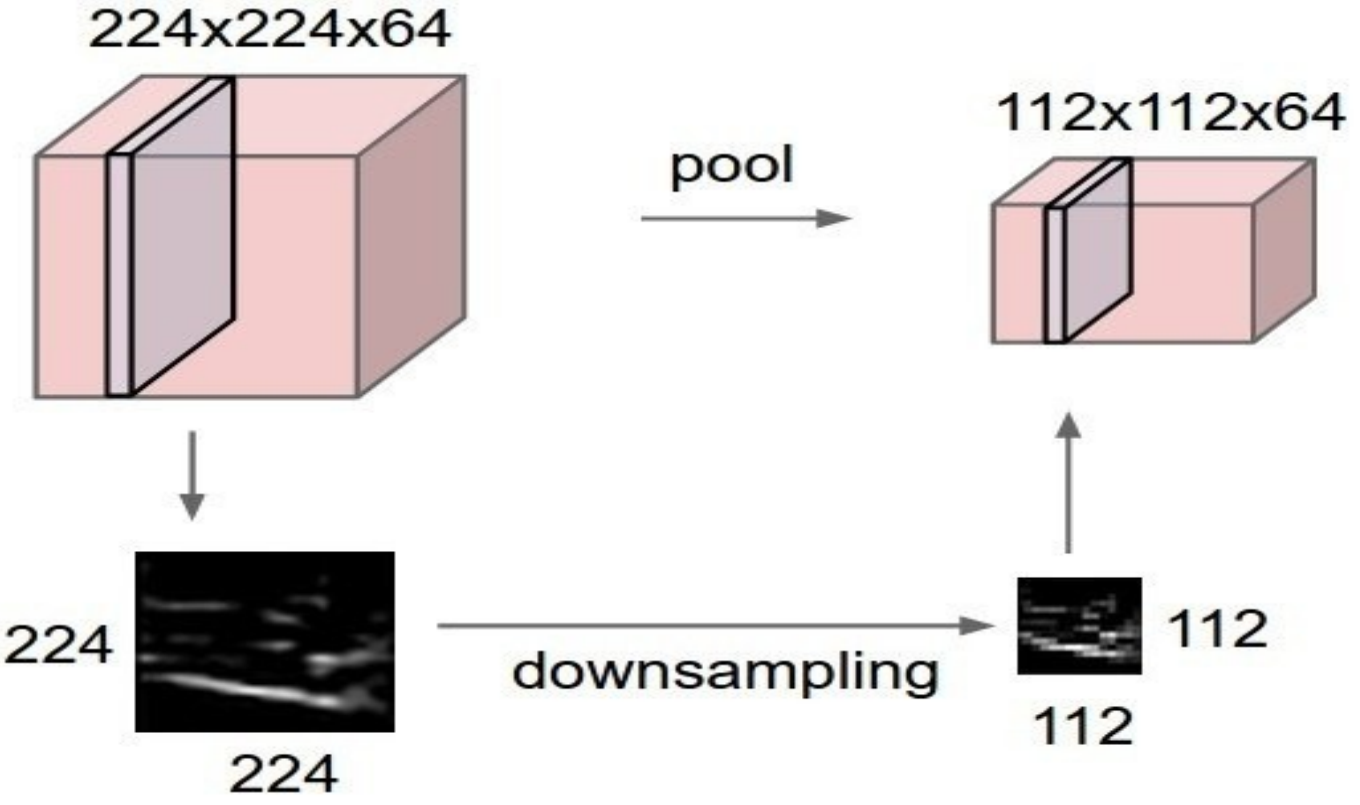
# Max Pooling

- For example a 4x4 matrix representing our initial input.
- We run a 2x2 filter over our input.
- Use a stride of 2 (meaning the (dx, dy) for stepping over our input will be (2, 2)) and won't overlap regions.
- For each of the regions represented by the filter, we will take the max of that region and create a new, output matrix where each element is the max of a region in the original input.

# Max Pooling


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

# Max Pooling



# Max Pooling

- MaxPooling2D

  MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None)

  Max pooling operation for spatial data.

- Arguments

  pool_size: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.

  strides: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to pool_size.

  padding: One of "valid" or "same" (case-insensitive).

  data_format: A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, height, width, channels) while channels_first corresponds to inputs with shape (batch, channels, height, width). It defaults to the image_data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".

# Max Pooling

- Input shape

  4D tensor with shape: (batch_size, rows, cols, channels)

- Output shape

  4D tensor with shape: (batch_size, pooled_rows, pooled_cols, channels)

# Dropout

- Dropout is a technique where randomly selected neurons are ignored during training.
- Their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.
- You can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.
- The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data.

# Practical Work

- Add Dropout and MaxPooling to the convolutional MNIST network.
- Train a convolutional network on the CIFAR10 image dataset.
- Add Dropout to the CIFAR10 network.

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D

(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
```

# MNIST

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=(28,28,1)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

# CIFAR10

```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0

train_labels = tf.keras.utils.to_categorical (train_labels)
test_labels = tf.keras.utils.to_categorical (test_labels)

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

# CIFAR10

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D

(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

train_labels = tf.keras.utils.to_categorical (train_labels)
test_labels = tf.keras.utils.to_categorical (test_labels)

train_images, test_images = train_images / 255.0, test_images / 255.0

model = models.Sequential()
model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.Adam(), metrics=['accuracy'])

model.fit(train_images, train_labels, batch_size=512, epochs=10, verbose=1, validation_data=(test_images, test_labels))
```

# The Functional API

# The Functional API

- The Functional API is a way to create models that is more flexible than Sequential:
- It can handle:
  - models with non-linear topology,
  - models with shared layers,
  - models with multiple inputs or outputs.

# The Functional API

- It's based on the idea that a deep learning model is usually a directed acyclic graph (DAG) of layers.
- The Functional API a set of tools for building graphs of layers.

# The Functional API

- Consider the following model:

  (input: 784-dimensional vectors)
  [Dense (64 units, relu activation)]
  [Dense (64 units, relu activation)]
  [Dense (10 units, softmax activation)]
  (output: probability distribution over 10 classes)

- It is a simple graph of 3 layers.

# The Functional API

- To build this model with the functional API, you would start by creating an input node:

```
from tensorflow import keras
inputs = keras.Input(shape=(784,))
```

- Here we just specify the shape of our data: 784-dimensional vectors.
- Note that the batch size is always omitted, we only specify the shape of each sample.
- For an input meant for images of shape (32, 32, 3), we would have used:

```
img_inputs = keras.Input(shape=(32, 32, 3))
```

- What gets returned, inputs, contains information about the shape and dtype of the input data that you expect to feed to your model:

```
inputs.shape = TensorShape([None, 784])
inputs.dtype = tf.float32
```

# The Functional API

- You create a new node in the graph of layers by calling a layer on this inputs object:

```
from tensorflow.keras import layers

dense = layers.Dense(64, activation='relu')
x = dense(inputs)
```

- The "layer call" action is like drawing an arrow from "inputs" to this layer we created.
- We're "passing" the inputs to the dense layer, and out we get x.

# The Functional API

- Let's add a few more layers to our graph of layers:

```
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)
```

- At this point, we can create a Model by specifying its inputs and outputs in the graph of layers:

```
model = keras.Model(inputs=inputs, outputs=outputs)
```

# The Functional API

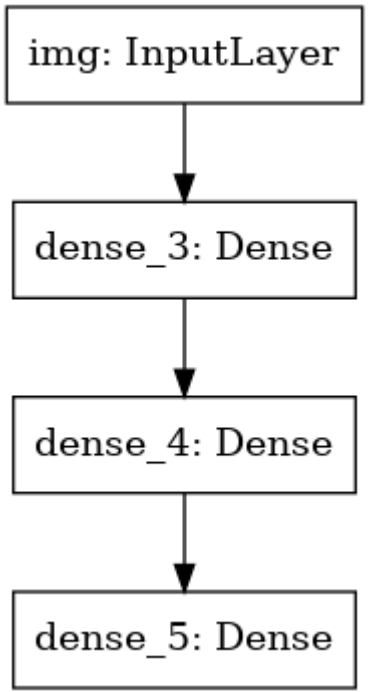
- To recap, here is our full model definition process:

```
inputs = keras.Input(shape=(784,), name='img')
x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs=inputs, outputs=outputs,
name='mnist_model')
```

# The Functional API

- We can also plot the model as a graph:

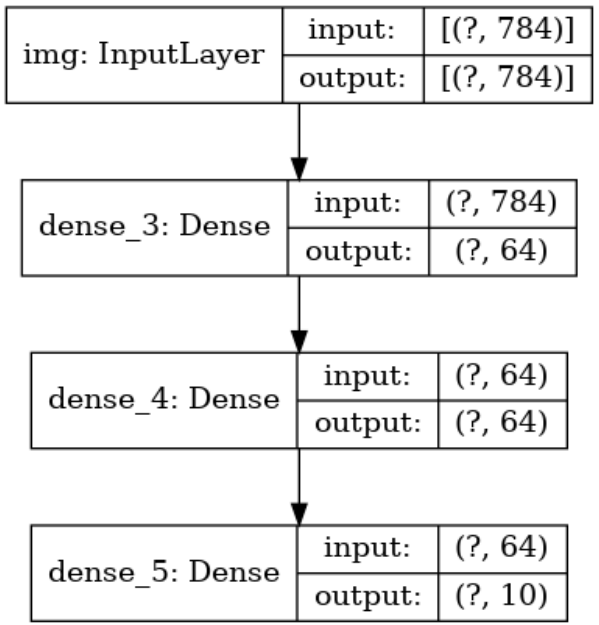
keras.utils.plot_model(model, 'my_first_model.png')



# The Functional API

- And optionally display the input and output shapes of each layer in the plotted graph:

  keras.utils.plot_model(model, 'my_first_model_with_shape_info.png', show_shapes=True)



# The Functional API

- Training, evaluation, and inference work exactly in the same way for models built using the Functional API as for Sequential models.
- Here is a quick demonstration.
- Here we load MNIST image data, reshape it into vectors, fit the model on the data (while monitoring performance on a validation split), and finally we evaluate our model on the test data:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255

model.compile(loss='sparse_categorical_crossentropy',
              optimizer=keras.optimizers.RMSprop(),
              metrics=['accuracy'])
history = model.fit(x_train, y_train,
                    batch_size=64,
                    epochs=5,
                    validation_split=0.2)
test_scores = model.evaluate(x_test, y_test, verbose=2)
print('Test loss:', test_scores[0])
print('Test accuracy:', test_scores[1])
```

# The Functional API

- Saving and serialization work exactly in the same way for models built using the Functional API as for Sequential models.

- The standard way to save a Functional model is to call model.save() to save the whole model into a single file.

- You can later recreate the same model from this file, even if you no longer have access to the code that created the model.

- This file includes: - The model's architecture - The model's weight values (which were learned during training) - The model's training config (what you passed to compile), if any - The optimizer and its state, if any (this enables you to restart training where you left off)

```
model.save('path_to_my_model.h5')
# Recreate the exact same model purely from the file:
model = keras.models.load_model('path_to_my_model.h5')
```

# Exercise

- Train a MNIST network defined with the functional API.

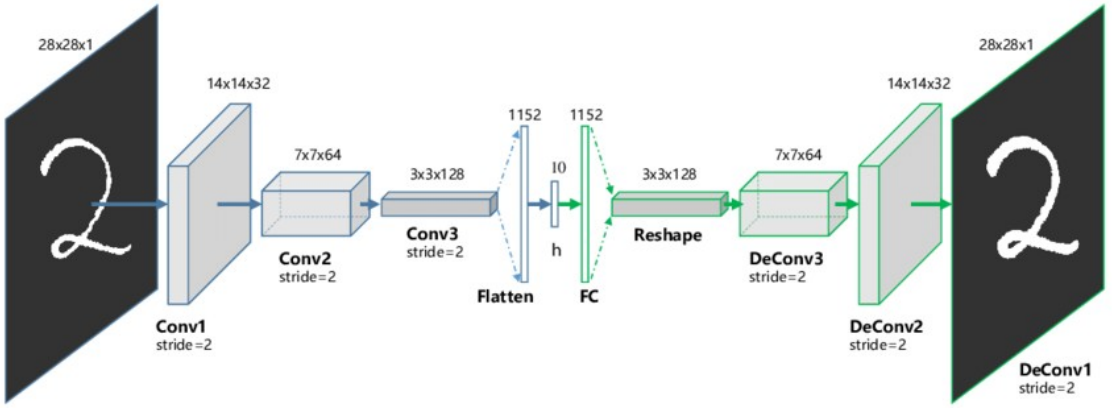
# MNIST

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
from tensorflow.keras.datasets import mnist
from tensorflow.keras import layers
from tensorflow.keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

inputs = keras.Input(shape=(784,), name='img')
x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs=inputs, outputs=outputs, name='mnist_model')
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

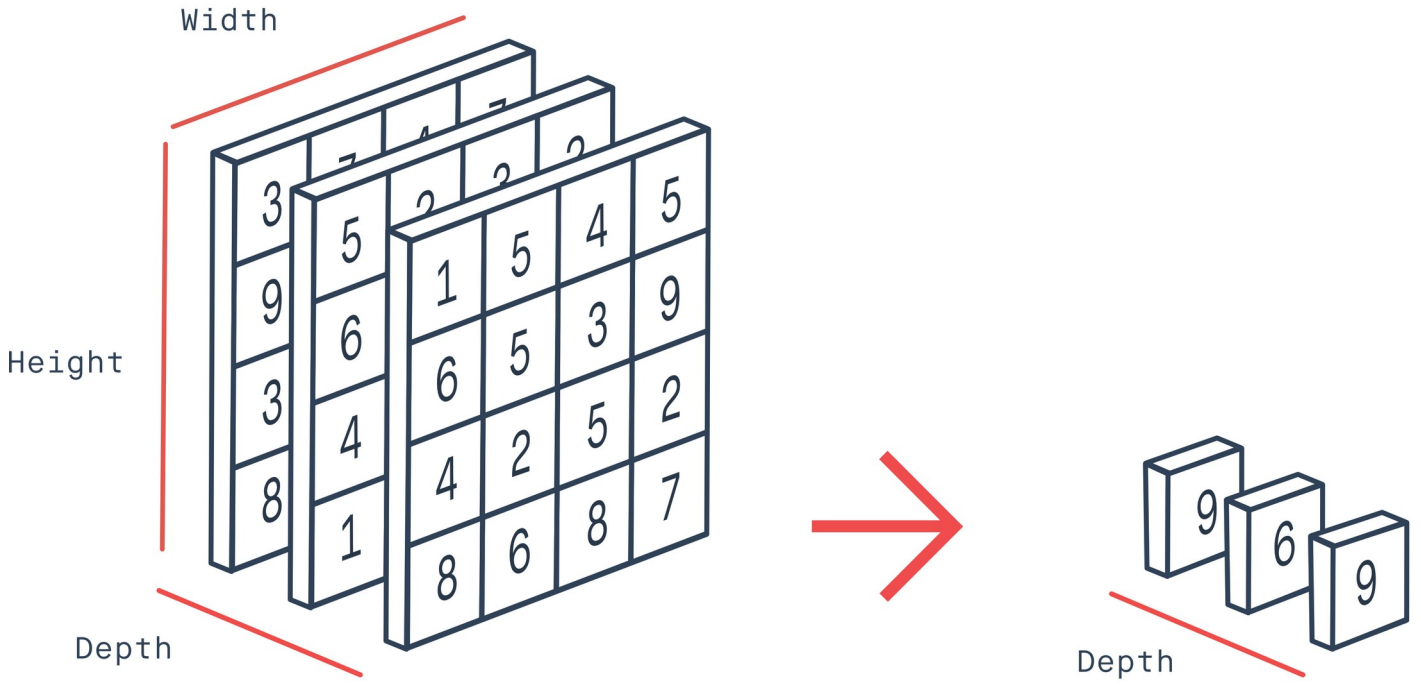
# Autoencoder

# Autoencoder

- We will train an Autoencoder on FashionMNIST.



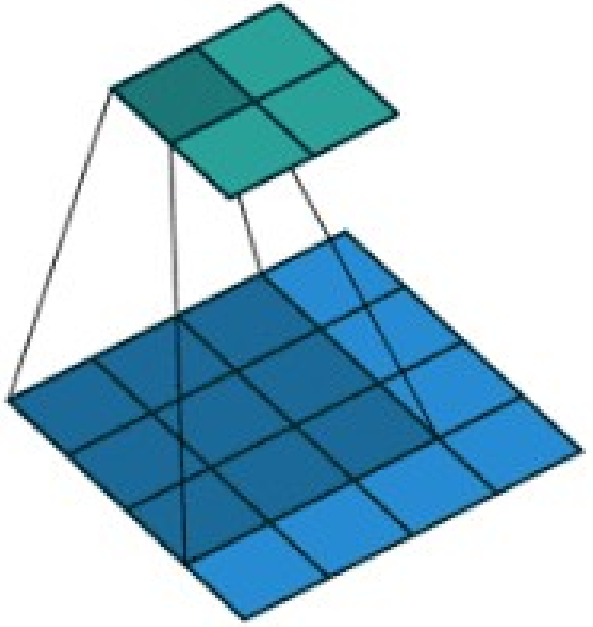
# GlobalMaxPooling2D



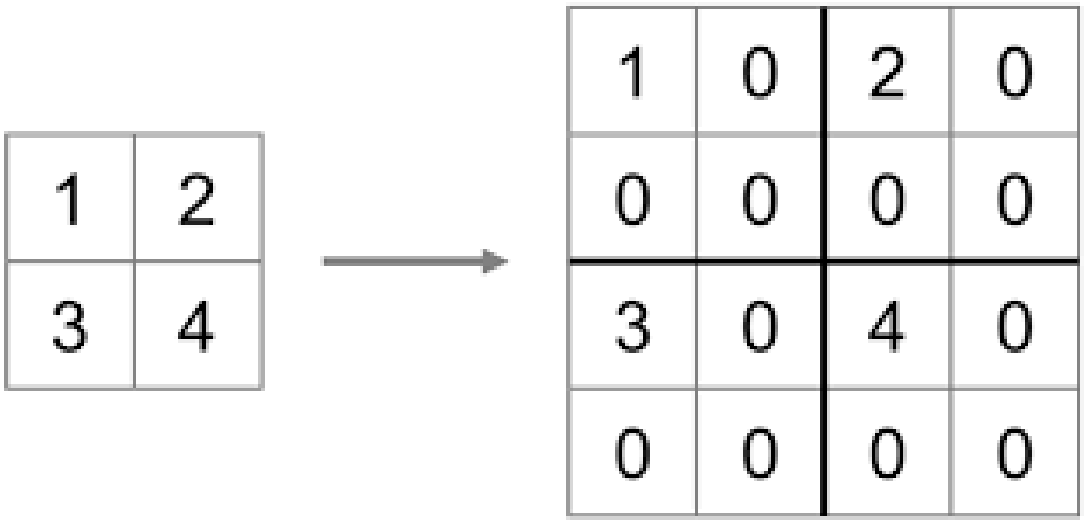

Height x Width x Depth

1 x 1 x Depth

# Conv2DTranspose



# UpSampling2D




- Nearest-Neighbor: Copies the value from the nearest pixel.
- Bilinear: Uses all nearby pixels to calculate the pixel's value, using linear interpolations.

# Autoencoder

- In the functional API, models are created by specifying their inputs and outputs in a graph of layers. That means that a single graph of layers can be used to generate multiple models.

- In the example below, we use the same stack of layers to instantiate two models: an encoder model that turns image inputs into 16-dimensional vectors, and an end-to-end autoencoder model for training.

```
encoder_input = keras.Input(shape=(28, 28, 1), name='img')
x = layers.Conv2D(16, 3, activation='relu')(encoder_input)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(3)(x)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.Conv2D(16, 3, activation='relu')(x)
encoder_output = layers.GlobalMaxPooling2D()(x)

encoder = keras.Model(encoder_input, encoder_output, name='encoder')
encoder.summary()
```

# Autoencoder

```
x = layers.Reshape((4, 4, 1))(encoder_output)
x = layers.Conv2DTranspose(16, 3, activation='relu')(x)
x = layers.Conv2DTranspose(32, 3, activation='relu')(x)
x = layers.UpSampling2D(3)(x)
x = layers.Conv2DTranspose(16, 3, activation='relu')(x)
decoder_output = layers.Conv2DTranspose(1, 3, activation='relu')(x)

autoencoder = keras.Model(encoder_input, decoder_output, name='autoencoder')
autoencoder.summary()
```

- Note that we make the decoding architecture strictly symmetrical to the encoding architecture, so that we get an output shape that is the same as the input shape (28, 28, 1).

- The reverse of a Conv2D layer is a Conv2DTranspose layer, and the reverse of a MaxPooling2D layer is an UpSampling2D layer.

# Autoencoder

- You can treat any model as if it were a layer, by calling it on an Input or on the output of another layer.
- Note that by calling a model you aren't just reusing the architecture of the model, you're also reusing its weights.
- Let's see this in action. Here's a different take on the autoencoder example that creates an encoder model, a decoder model, and chain them in two calls to obtain the autoencoder model:

```
encoder_input = keras.Input(shape=(28, 28, 1), name='original_img')
x = layers.Conv2D(16, 3, activation='relu')(encoder_input)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(3)(x)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.Conv2D(16, 3, activation='relu')(x)
encoder_output = layers.GlobalMaxPooling2D()(x)

encoder = keras.Model(encoder_input, encoder_output, name='encoder')
encoder.summary()
```

# Autoencoder

```
decoder_input = keras.Input(shape=(16,), name='encoded_img')
x = layers.Reshape((4, 4, 1))(decoder_input)
x = layers.Conv2DTranspose(16, 3, activation='relu')(x)
x = layers.Conv2DTranspose(32, 3, activation='relu')(x)
x = layers.UpSampling2D(3)(x)
x = layers.Conv2DTranspose(16, 3, activation='relu')(x)
decoder_output = layers.Conv2DTranspose(1, 3, activation='relu')(x)

decoder = keras.Model(decoder_input, decoder_output, name='decoder')
decoder.summary()

autoencoder_input = keras.Input(shape=(28, 28, 1), name='img')
encoded_img = encoder(autoencoder_input)
decoded_img = decoder(encoded_img)
autoencoder = keras.Model(autoencoder_input, decoded_img, name='autoencoder')
autoencoder.summary()
```

# Exercise

- Train a an autoencoder for fashion_mnist with the functional API.

# Autoencoder

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.datasets import fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

train_images = train_images.astype('float32') / 255
test_images = test_images.astype('float32') / 255
from tensorflow.keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

# Encoder

```
encoder_input = tf.keras.Input(shape=(28, 28, 1), name='original_img')
x = layers.Conv2D(16, 3, activation='relu')(encoder_input)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(3)(x)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.Conv2D(16, 3, activation='relu')(x)
encoder_output = layers.GlobalMaxPooling2D()(x)

encoder = tf.keras.Model(encoder_input, encoder_output, name='encoder')
encoder.summary()
```

# Decoder

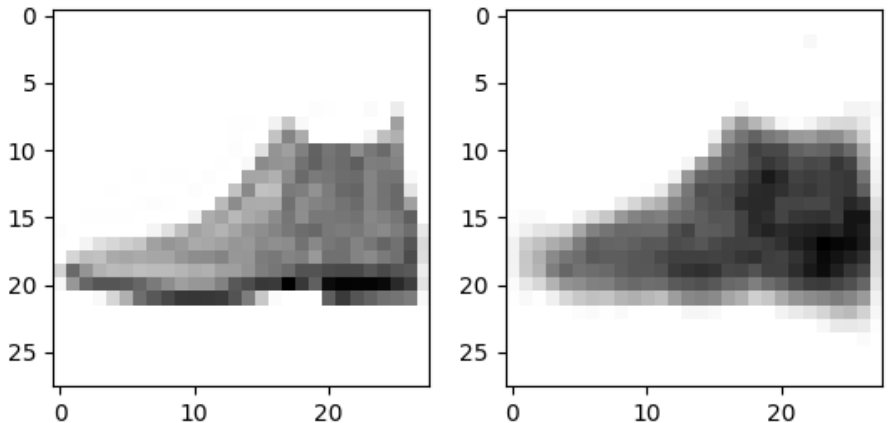
```
decoder_input = tf.keras.Input(shape=(16,), name='encoded_img')
x = layers.Reshape((4, 4, 1))(decoder_input)
x = layers.Conv2DTranspose(16, 3, activation='relu')(x)
x = layers.Conv2DTranspose(32, 3, activation='relu')(x)
x = layers.UpSampling2D(3)(x)
x = layers.Conv2DTranspose(16, 3, activation='relu')(x)
decoder_output = layers.Conv2DTranspose(1, 3, activation='relu')(x)

decoder = tf.keras.Model(decoder_input, decoder_output, name='decoder')
decoder.summary()
```

# Autoencoder

```
autoencoder_input = tf.keras.Input(shape=(28, 28, 1), name='img')

encoded_img = encoder(autoencoder_input)

decoded_img = decoder(encoded_img)

autoencoder = tf.keras.Model(autoencoder_input, decoded_img, name='autoencoder')

autoencoder.summary()


autoencoder.compile(optimizer='adam', loss='mse')


autoencoder.fit(train_images, train_images, epochs=5, batch_size=128, shuffle=True, validation_data=(test_images, test_images))
```

# Autoencoder

```
import matplotlib.pyplot as plt

print("and the actual image looks like")
img_out = autoencoder.predict(test_images[0].reshape(1,28,28,1))
f, axarr = plt.subplots(1,2)
axarr[0].imshow(test_images[0].reshape(28,28), cmap='Greys')
axarr[1].imshow(img_out.reshape(28,28), cmap='Greys')
plt.show()
```



# Exercise

- Try to add noise to the test images and see if the autoencoder can denoise them.
- Train an auto encoder to denoise images.

# Autoencoder

- Denoising the test set :

```
import numpy as np
n = test_images.shape [0]
n_rows = test_images.shape [1]
n_cols = test_images.shape [2]
mean = 0.1
stddev = 0.05
noise = np.random.normal(mean, stddev, (n, n_rows, n_cols))
test_images_noisy = test_images + noise
img_out = autoencoder.predict(test_images_noisy [0].reshape(1,28,28,1))
f, axarr = plt.subplots(1,2)
axarr[0].imshow(test_images[0].reshape(28,28), cmap='Greys')
axarr[1].imshow(img_out.reshape(28,28), cmap='Greys')
plt.show()
```

# Autoencoder

- Training to denoise :

```
import numpy as np
n = train_images.shape [0]
n_rows = train_images.shape [1]
n_cols = train_images.shape [2]
mean = 0.1
stddev = 0.05
noise = np.random.normal(mean, stddev, (n, n_rows, n_cols))
train_images_noisy = train_images + noise

autoencoder_input = tf.keras.Input(shape=(28, 28, 1), name='img')
encoded_img = encoder(autoencoder_input)
decoded_img = decoder(encoded_img)
autoencoder = tf.keras.Model(autoencoder_input, decoded_img, name='autoencoder')
autoencoder.summary()

autoencoder.compile(optimizer='adam', loss='mse')

autoencoder.fit(train_images_noisy, train_images, epochs=5, batch_size=128, shuffle=True, validation_data=(test_images, test_images))
```

# Autoencoder

- Generating images :

```
mean = 0.5
stddev = 0.3
noise1 = np.random.normal(mean, stddev, (16,))
noise2 = np.random.normal(mean, stddev, (16,))
img_out1 = decoder.predict(noise1.reshape(1,16))
img_out2 = decoder.predict(noise2.reshape(1,16))
f, axarr = plt.subplots(1,2)
axarr[0].imshow(img_out1.reshape(28,28), cmap='Greys')
axarr[1].imshow(img_out2.reshape(28,28), cmap='Greys')
plt.show()
```

# Variational Autoencoders

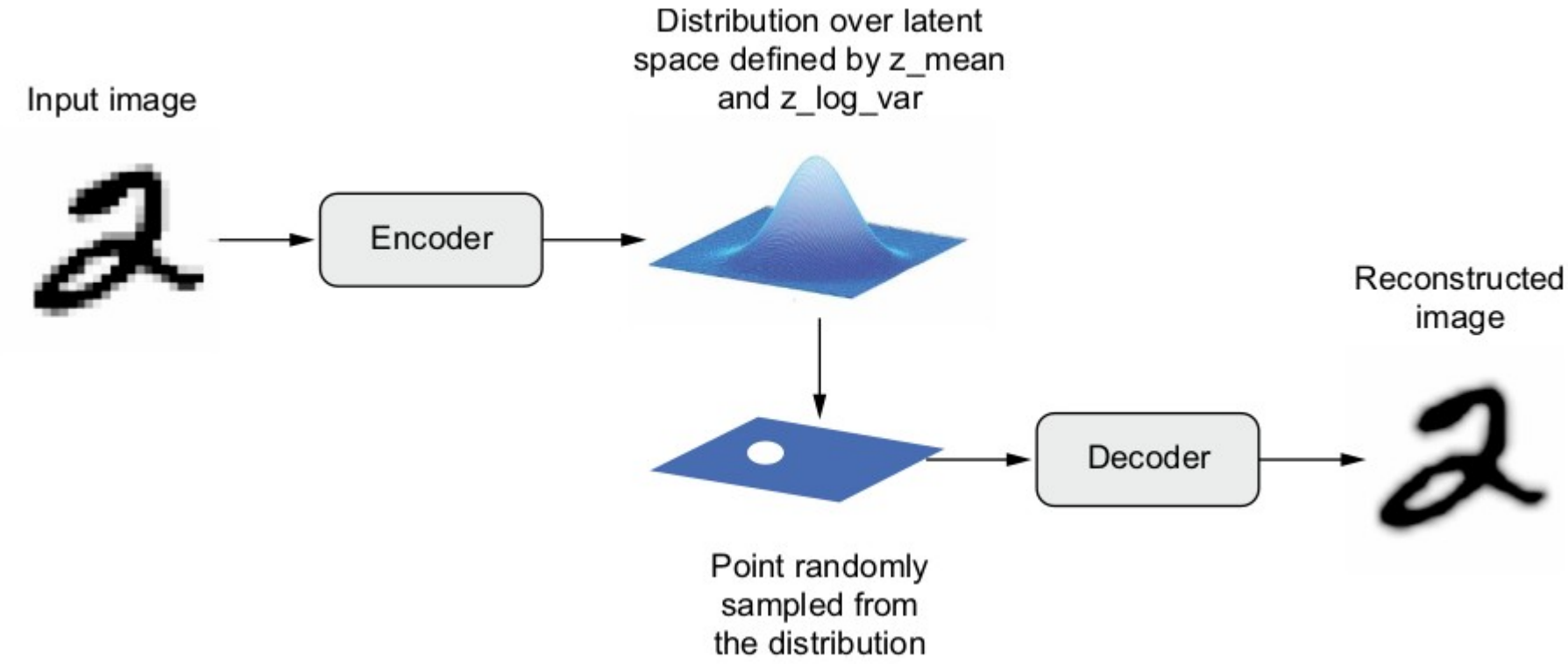
# Variational Autoencoders

- Classical autoencoders do not have nicely structured latent spaces.
- VAEs augment autoencoders with statistical constraints that force them to learn continuous, highly structured latent spaces.
- They have turned out to be a powerful tool for image generation.

# Variational Autoencoders

- A VAE, instead of compressing its input image into a fixed code in the latent space, turns the image into the parameters of a statistical distribution: a mean and a variance.
- The VAE uses the mean and variance parameters to randomly sample one element of the distribution, and decodes that element.
- More robustness.
- Forces the latent space to encode meaningful representations.

# Variational Autoencoders



# Variational Autoencoders

- An encoder encodes the input imput_img into two parameters in a latent space of representations, z_mean and z_log_variance.
- A point z is randomly sampled from the latent normal distribution :

  z = z_mean + exp(z_log_variance) * epsilon

  where epsilon is a random tensor of small values.
- A decoder decodes this point to the original input image.

# Variational Autoencoders

```
z_mean, z_log_variance = encoder(input_img)
z = z_mean + exp(z_log_variance) * epsilon
reconstructed_img = decoder(z)
model = Model(input_img, reconstructed_img)
```

# Variational Autoencoders

- Exercise :

  Train a variational autoencoder for fashion_mnist

# Variational Autoencoders

Encoder:

```
from tensorflow import keras
from tensorflow.keras import layers
latent_dim = 2
encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

# Variational Autoencoders

Latent space sampling layer:

```
import tensorflow as tf
class Sampler(layers.Layer):
    def call(self, z_mean, z_log_var):
        batch_size = tf.shape(z_mean)[0]
        z_size = tf.shape(z_mean)[1]
        epsilon = tf.random.normal(shape=(batch_size, z_size))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
```

# Variational Autoencoders

Decoder:

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

# Variational Autoencoders

VAE model with custom train_step():

```
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.sampler = Sampler()
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = keras.metrics.Mean(name="reconstruction_loss")
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")
```

# Variational Autoencoders

```
@property
def metrics(self):
    return [self.total_loss_tracker, self.reconstruction_loss_tracker, self.kl_loss_tracker]

def train_step(self, data):
    with tf.GradientTape() as tape:
        z_mean, z_log_var = self.encoder(data)
        z = self.sampler(z_mean, z_log_var)
        reconstruction = decoder(z)
        reconstruction_loss = tf.reduce_mean(
                                    tf.reduce_sum(keras.losses.binary_crossentropy(data, reconstruction),axis=(1, 2)))
        kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)) # Regularization: Kullback-Leibler divergence
        total_loss = reconstruction_loss + tf.reduce_mean(kl_loss)
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return {"total_loss": self.total_loss_tracker.result(),"reconstruction_loss": self.reconstruction_loss_tracker.result(),"kl_loss": self.kl_loss_tracker.result(),}
```

# Variational Autoencoders

Training the VAE:

```
import numpy as np
(x_train, _), (x_test, _) = keras.datasets.fashion_mnist.load_data()
X = np.concatenate([x_train, x_test], axis=0)
X = np.expand_dims(X, -1).astype("float32") / 255
vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam(), run_eagerly=True)
vae.fit(X, epochs=30, batch_size=128)
```

# Variational Autoencoders

Sampling a grid of images from the 2D latent space:

```
import matplotlib.pyplot as plt
n = 30
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
grid_x = np.linspace(-1, 1, n)
grid_y = np.linspace(-1, 1, n)[::-1]
for i, yi in enumerate(grid_y):
    for j, xi in enumerate(grid_x):
        z_sample = np.array([[xi, yi]])
        x_decoded = vae.decoder.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size : (i + 1) * digit_size, j * digit_size : (j + 1) * digit_size,] = digit
plt.figure(figsize=(15, 15))
start_range = digit_size // 2
end_range = n * digit_size + start_range
pixel_range = np.arange(start_range, end_range, digit_size)
sample_range_x = np.round(grid_x, 1)
sample_range_y = np.round(grid_y, 1)
plt.xticks(pixel_range, sample_range_x)
plt.yticks(pixel_range, sample_range_y)
plt.xlabel("z[0]")
plt.ylabel("z[1]")
plt.axis("off")
plt.imshow(figure, cmap="Greys_r")
```

# Ensemble

- As you can see, model can be nested: a model can contain submodels (since a model is just like a layer).
- A common use case for model nesting is ensembling. As an example, here's how to ensemble a set of models into a single model that averages their predictions:

```
def get_model():
  inputs = keras.Input(shape=(128,))
  outputs = layers.Dense(1, activation='sigmoid')(inputs)
  return keras.Model(inputs, outputs)

model1 = get_model()
model2 = get_model()
model3 = get_model()

inputs = keras.Input(shape=(128,))
y1 = model1(inputs)
y2 = model2(inputs)
y3 = model3(inputs)
outputs = layers.average([y1, y2, y3])
ensemble_model = keras.Model(inputs=inputs, outputs=outputs)
```

# Multiple Inputs and Outputs

- Models with multiple inputs and outputs
- The functional API makes it easy to manipulate multiple inputs and outputs. This cannot be handled with the Sequential API.
- Let's say you're building a system for ranking custom issue tickets by priority and routing them to the right department.

- Your model will have 3 inputs:

  Title of the ticket (text input)
  Text body of the ticket (text input)
  Any tags added by the user (categorical input)

- It will have two outputs:

  Priority score between 0 and 1 (scalar sigmoid output)
  The department that should handle the ticket (softmax output over the set of departments)

# Multiple Inputs and Outputs

- Let's built this model in a few lines with the Functional API.

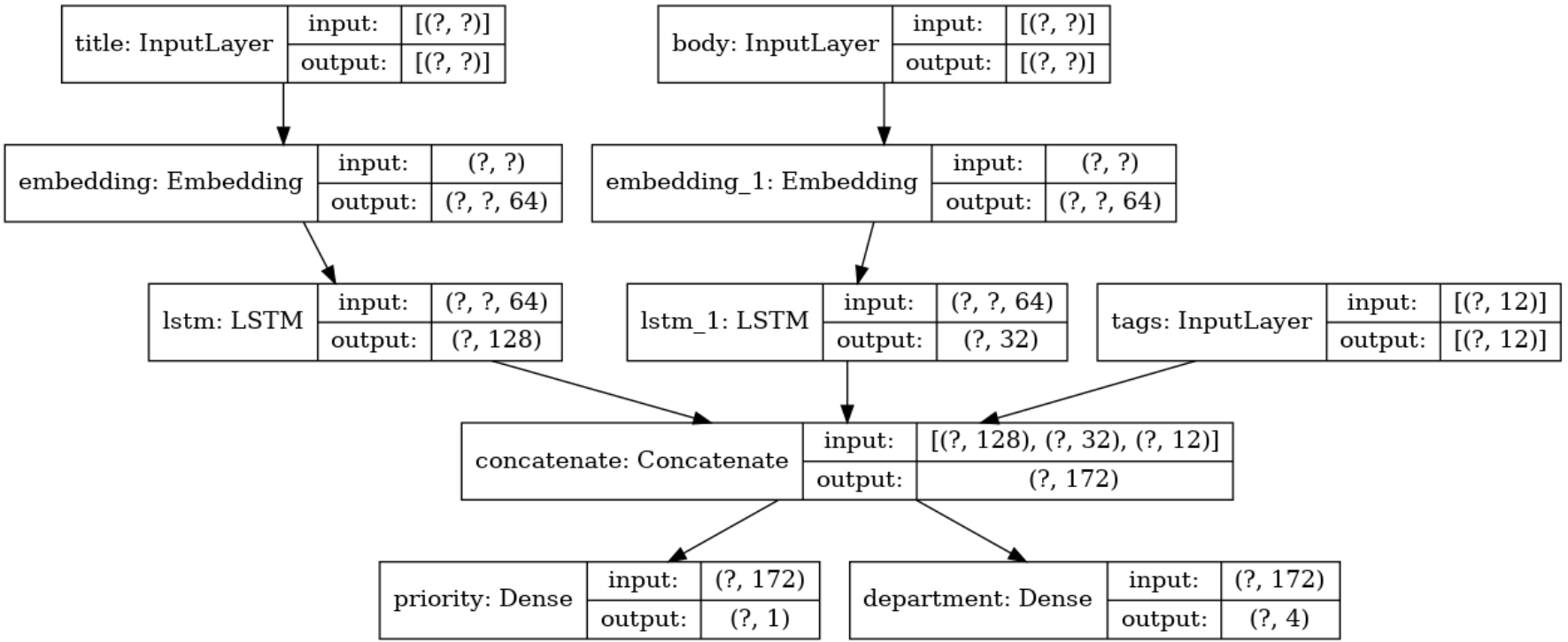
```
num_tags = 12  # Number of unique issue tags
num_words = 10000  # Size of vocabulary obtained when preprocessing text data
num_departments = 4  # Number of departments for predictions

title_input = keras.Input(shape=(None,), name='title')  # Variable-length sequence of ints
body_input = keras.Input(shape=(None,), name='body')  # Variable-length sequence of ints
tags_input = keras.Input(shape=(num_tags,), name='tags')  # Binary vectors of size `num_tags`

# Embed each word in the title into a 64-dimensional vector
title_features = layers.Embedding(num_words, 64)(title_input)
# Embed each word in the text into a 64-dimensional vector
body_features = layers.Embedding(num_words, 64)(body_input)
```

# Multiple Inputs and Outputs

```
# Reduce sequence of embedded words in the title into a single 128-dimensional vector
title_features = layers.LSTM(128)(title_features)
# Reduce sequence of embedded words in the body into a single 32-dimensional vector
body_features = layers.LSTM(32)(body_features)

# Merge all available features into a single large vector via concatenation
x = layers.concatenate([title_features, body_features, tags_input])

# Stick a logistic regression for priority prediction on top of the features
priority_pred = layers.Dense(1, activation='sigmoid', name='priority')(x)
# Stick a department classifier on top of the features
department_pred = layers.Dense(num_departments, activation='softmax', name='department')(x)

# Instantiate an end-to-end model predicting both priority and department
model = keras.Model(inputs=[title_input, body_input, tags_input],
                    outputs=[priority_pred, department_pred])
```

# Multiple Inputs and Outputs

keras.utils.plot_model(model, 'multi_input_and_output_model.png', show_shapes=True)



# Multiple Inputs and Outputs

- When compiling this model, we can assign different losses to each output.
- You can even assign different weights to each loss, to modulate their contribution to the total training loss.

```
model.compile(optimizer=keras.optimizers.RMSprop(1e-3),
              loss=['binary_crossentropy', 'categorical_crossentropy'],
              loss_weights=[1., 0.2])
```

# Multiple Inputs and Outputs

- Since we gave names to our output layers, we could also specify the loss like this:

```
model.compile(optimizer=keras.optimizers.RMSprop(1e-3),
              loss={'priority': 'binary_crossentropy',
                    'department': 'categorical_crossentropy'},
              loss_weights=[1., 0.2])
```

# Multiple Inputs and Outputs

- We can train the model by passing lists of Numpy arrays of inputs and targets:

```
import numpy as np

# Dummy input data
title_data = np.random.randint(num_words, size=(1280, 10))
body_data = np.random.randint(num_words, size=(1280, 100))
tags_data = np.random.randint(2, size=(1280, num_tags)).astype('float32')
# Dummy target data
priority_targets = np.random.random(size=(1280, 1))
dept_targets = np.random.randint(2, size=(1280, num_departments))

model.fit({'title': title_data, 'body': body_data, 'tags': tags_data},
          {'priority': priority_targets, 'department': dept_targets},
          epochs=2,
          batch_size=32)
```

# Multiple Inputs and Outputs

- When calling fit with a Dataset object, it should yield either a tuple of lists like :

  ([title_data, body_data, tags_data], [priority_targets, dept_targets])

- or a tuple of dictionaries like :

  ({'title': title_data, 'body': body_data, 'tags': tags_data}, {'priority': priority_targets, 'department': dept_targets}).

# Exercise

- Write a convolutional model that takes 31 19x19 planes as input and that outputs a vector of 361 with a softmax (the policy) and an output of 1 (the value).

- Train it on randomly generated data with different losses for the policy (categorical cross entropy) and the value (mse).

# Exercise

Generating random data :

```
N = 10000
planes = 31
moves = 361

input_data = np.random.randint(2, size=(N, 19, 19, planes))
input_data = input_data.astype ('float32')

policy = np.random.randint(moves, size=(N,))
policy = keras.utils.to_categorical (policy)

value = np.random.randint(2, size=(N,))
value = value.astype ('float32')
```

# Exercise

Building the model :

```
input = keras.Input(shape=(19, 19, planes), name='board')
x = layers.Conv2D(32, 3, activation='relu', padding='same')(input)
x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
policy_head = layers.Conv2D(1, 3, activation='relu', padding='same')(x)
policy_head = layers.Flatten()(policy_head)
policy_head = layers.Dense(moves, activation='softmax', name='policy')(policy_head)
value_head = layers.Flatten()(x)
value_head = layers.Dense(1, activation='sigmoid', name='value')(value_head)

model = keras.Model(inputs=input, outputs=[policy_head, value_head])

model.summary ()
```

# Exercise

Training and saving the model :

```
model.compile(optimizer=keras.optimizers.SGD(lr=0.001),
                    loss={'value': 'mse', 'policy': 'categorical_crossentropy'})

model.fit(input_data, {'policy': policy, 'value': value},
            epochs=20, batch_size=128, validation_split=0.1)

model.save ('test.h5')
```

# The Deep Learning Project

# The Deep Learning Project

- https://www.lamsade.dauphine.fr/~cazenave/DeepLearningProject.html
- The goal is to train a network for playing the game of Go.
- In order to be fair about training ressources the number of parameters for the networks you submit must be lower than 100 000.
- The maximum number of students per team is two.
- The data used for training comes from the Katago Go program self played games.
- There are 1 000 000 different games in total in the training set.
- The input data is composed of 31 19x19 planes (color to play, ladders, current state on two planes, two previous states on multiple planes).
- The output targets are the policy (a vector of size 361 with 1.0 for the move played, 0.0 for the other moves), and the value (a value between 0.0 and 1.0 given by the Monte-Carlo Tree Search representing the probability for White to win).

# The Deep Learning Project

- The project has been written and runs on Ubuntu 22.04.
- It uses Tensorflow 2.9 and Keras for the network.
- An example of a convolutional network with two heads is given in file golois.py and saved in file test.h5.
- The networks you design and train should also have the same policy and value heads and be saved in h5 format.
- An example network and training episode is given in file golois.py.
- If you want to compile the golois library you should install Pybind11 and call compile.sh.

# The Deep Learning Project

- Tournaments :
- Each two weeks or so I will organize a tournament between the networks you upload.
- Each network name is the names of the students who designed and trained the network.
- The model should be saved in keras h5 format.
- A round robin tournament will be organized and the results will be sent by email.
- Each network will be used by a PUCT engine that takes 2 seconds of CPU time at each move to play in the tournament.

# Project

```
planes = 31
moves = 361
N = 10000
epochs = 20
batch = 128
filters = 32

input_data = np.random.randint(2, size=(N, 19, 19, planes))
input_data = input_data.astype ('float32')

policy = np.random.randint(moves, size=(N,))
policy = keras.utils.to_categorical (policy)

value = np.random.randint(2, size=(N,))
value = value.astype ('float32')

end = np.random.randint(2, size=(N, 19, 19, 2))
end = end.astype ('float32')

groups = np.zeros((N, 19, 19, 1))
groups = groups.astype ('float32')
```

# Project

```
input = keras.Input(shape=(19, 19, planes), name='board')
x = layers.Conv2D(filters, 1, activation='relu', padding='same')(input)
for i in range (5):
    x = layers.Conv2D(filters, 3, activation='relu', padding='same')(x)
policy_head = layers.Conv2D(1, 1, activation='relu', padding='same', use_bias = False, kernel_regularizer=regularizers.l2(0.0001))(x)
policy_head = layers.Flatten()(policy_head)
policy_head = layers.Activation('softmax', name='policy')(policy_head)
value_head = layers.Conv2D(1, 1, activation='relu', padding='same', use_bias = False, kernel_regularizer=regularizers.l2(0.0001))(x)
value_head = layers.Flatten()(value_head)
value_head = layers.Dense(50, activation='relu', kernel_regularizer=regularizers.l2(0.0001))(value_head)
value_head = layers.Dense(1, activation='sigmoid', name='value', kernel_regularizer=regularizers.l2(0.0001))(value_head)

model = keras.Model(inputs=input, outputs=[policy_head, value_head])
```

# Project

```
model.compile(optimizer=keras.optimizers.SGD(learning_rate=0.0005, momentum=0.9),
              loss={'policy': 'categorical_crossentropy', 'value': 'binary_crossentropy'},
              loss_weights={'policy' : 1.0, 'value' : 1.0},
              metrics={'policy': 'categorical_accuracy', 'value': 'mse'})

for i in range (1, epochs + 1):
    print ('epoch ' + str (i))
    golois.getBatch (input_data, policy, value, end, groups, i * N)
    history = model.fit(input_data,
                        {'policy': policy, 'value': value},
                        epochs=1, batch_size=batch)
    if (i % 5 == 0):
        gc.collect ()
    if (i % 20 == 0):
        golois.getValidation (input_data, policy, value, end)
        val = model.evaluate (input_data,
                              [policy, value], verbose = 0, batch_size=batch)
        print ("val =", val)
        model.save ('test.h5')
```

# Project

- Train a network to play Go
- Submit trained networks by saturday evening
- Tournament of networks every sunday
- Upload a network by the end of the session

# Residual Networks

# Residual Networks

- In addition to models with multiple inputs and outputs, the Functional API makes it easy to manipulate non-linear connectivity topologies : models where layers are not connected sequentially.

- This also cannot be handled with the Sequential API (as the name indicates).

- A common use case for this is residual connections.

AlphaGo

David Silver

Aja Huang

# AlphaGo

Fan Hui is the european Go champion and a 2p professional Go player :


AlphaGo Fan won 5-0 against Fan Hui in November 2015.

Nature, January 2016.

# AlphaGo

Lee Sedol is among the strongest and most famous 9p Go player :

AlphaGo Lee won 4-1 against Lee Sedol in march 2016.

# AlphaGo

Ke Jie is the world champion of Go according to Elo ratings :

AlphaGo Master won 3-0 against Ke Jie in may 2017.

# MCTS and Deep RL

Monte Carlo Tree Search and Deep Reinforcement Learning to discover new fast matrix multiplication algorithms :

# Golois

- In 2016 I replicated the AlphaGo experiments with the policy and value networks.
- Golois policy network scores 58.54% on the test set (57.0% for AlphaGo).
- Policy alone (instant play) is ranked 4d on kgs instead of 3d for AlphaGo.
- The improvement is the use of a residual network for the policy.

# Residual Networks

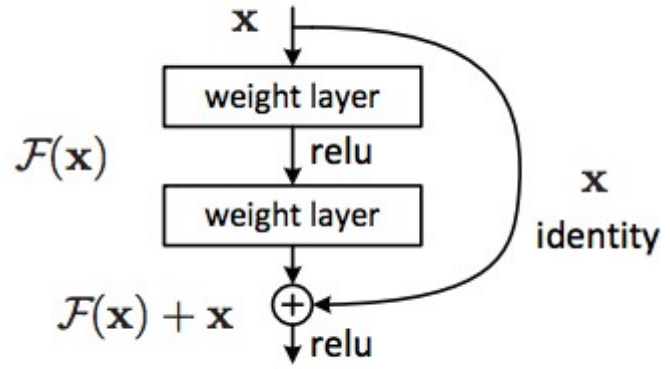
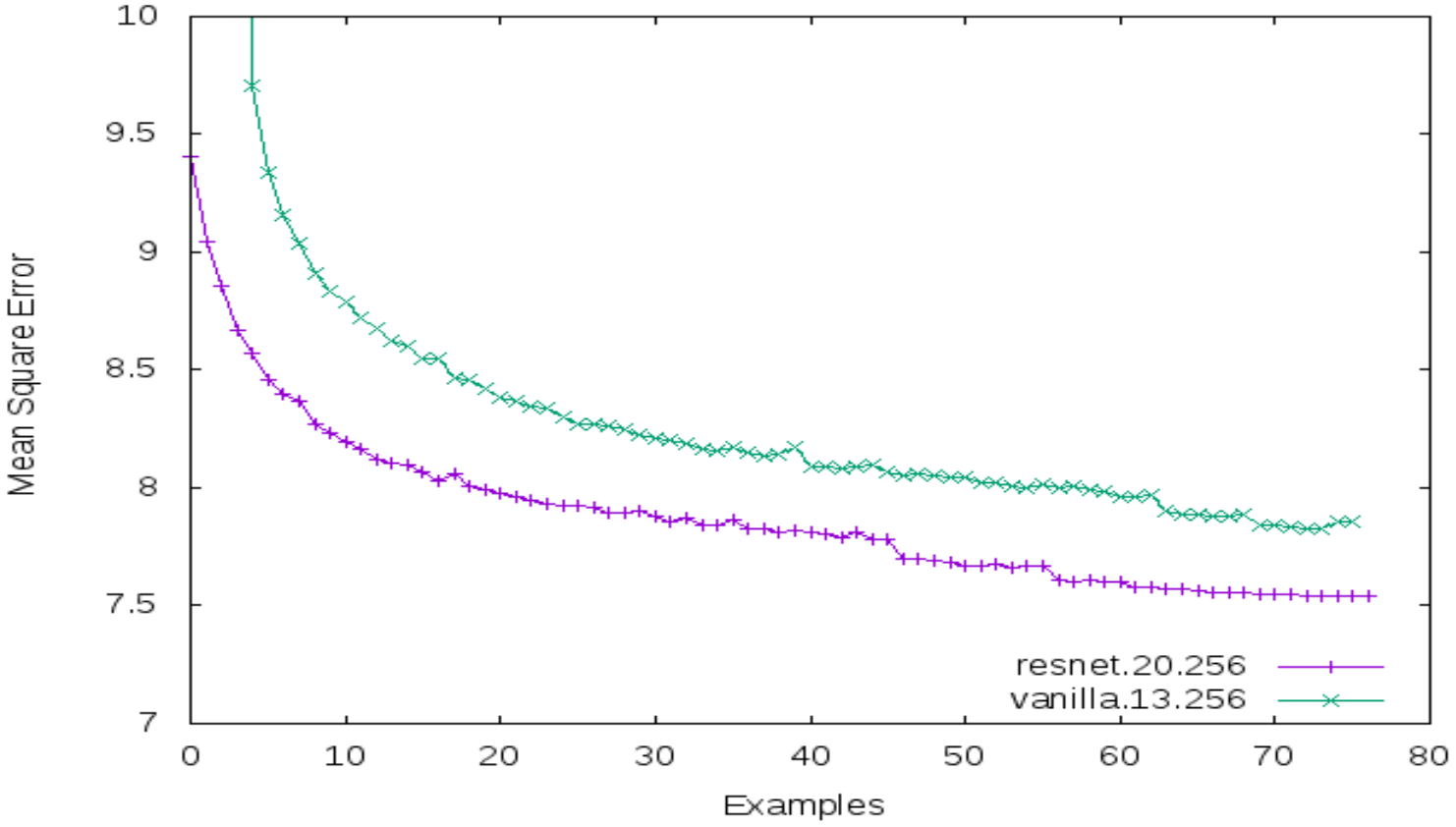



Figure 2. Residual learning: a building block.

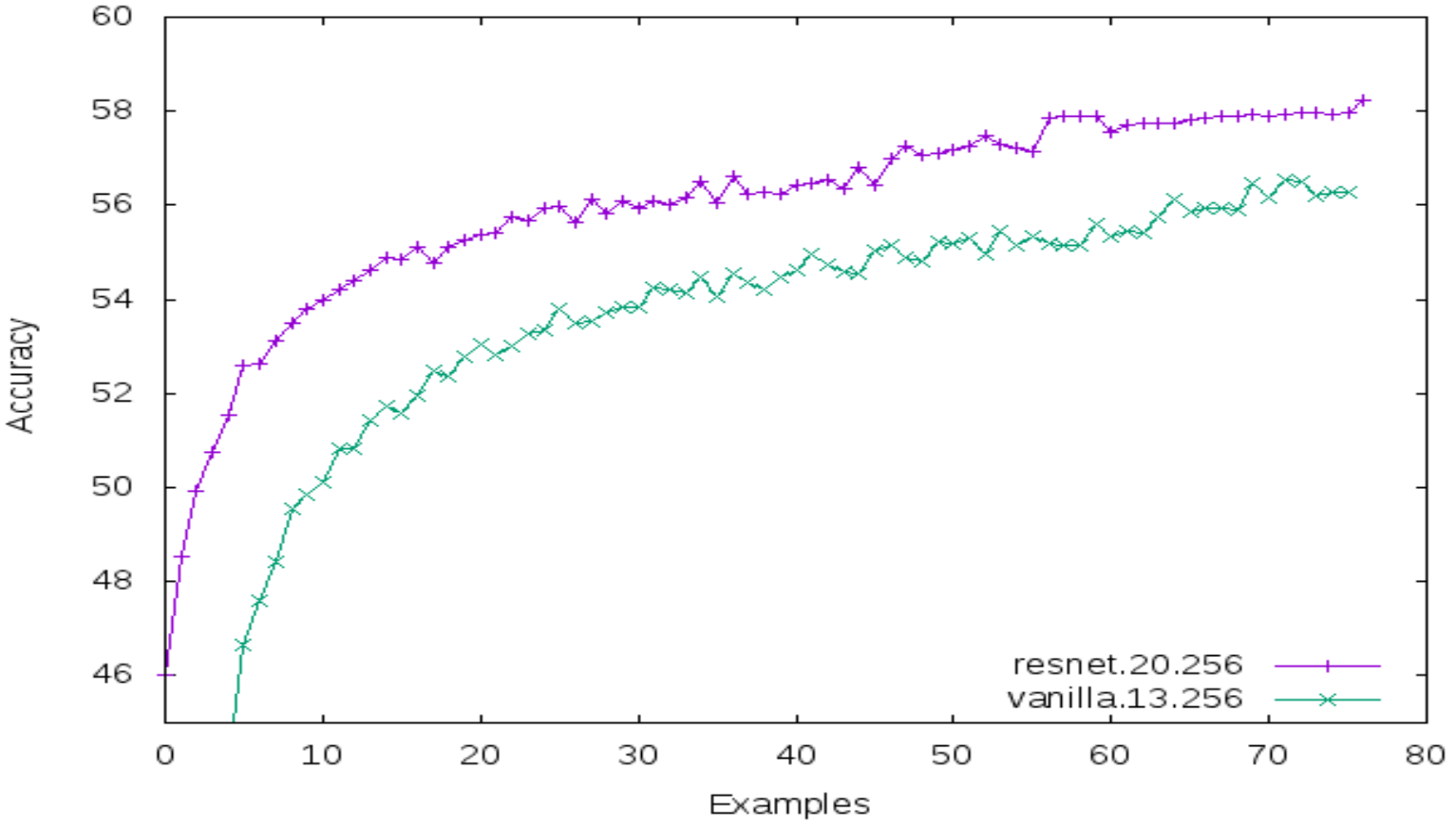
# Residual Networks

- “Deep Residual Learning for Image Recognition” [He et al. 2015].
- The error gradient information propagates noiselessly through a deep network.
- The input and the output of a block should have the same shape.
- Use padding=’same’ to have the same dimensions.
- To modify the shape use a 1x1 convolution with no activation.
- Very deep networks without having to worry about vanishing gradients.

# Evolution of the error



# Evolution of the accuracy



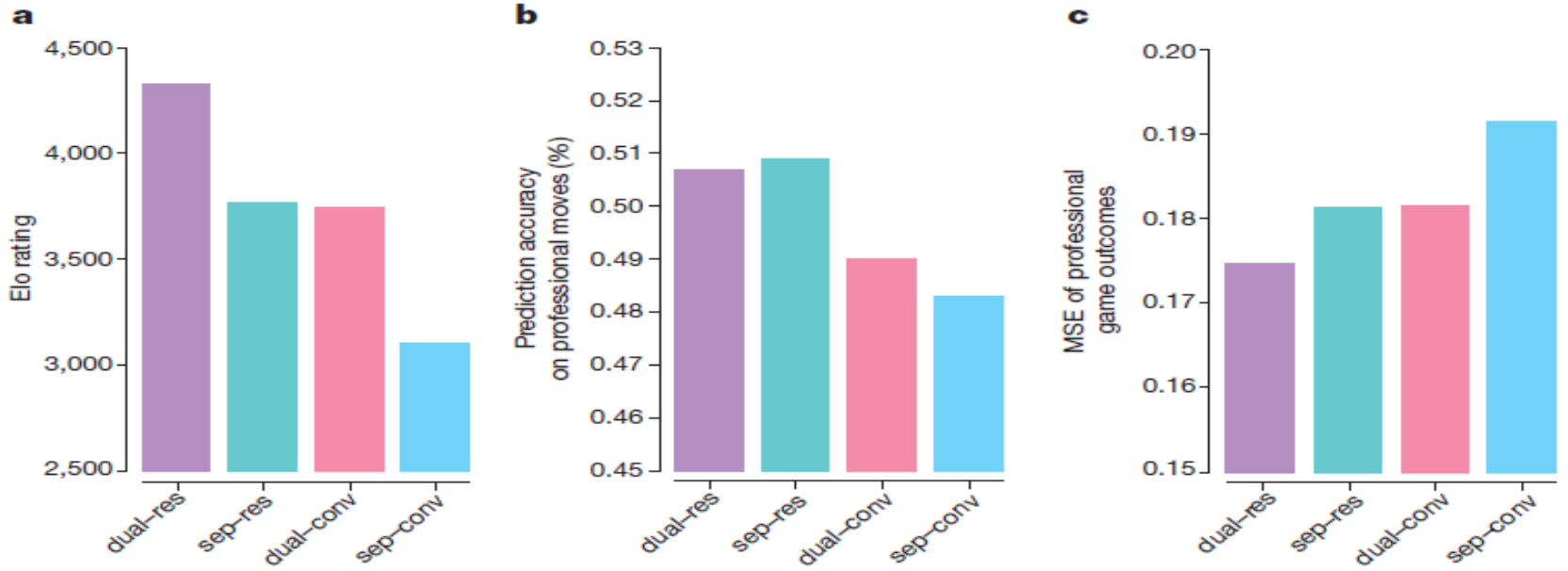
# AlphaGo

AlphaGo Zero learns to play Go from scratch playing against itself.

After 40 days of self play it surpasses AlphaGo Master.

Nature, 18 october 2017.

# AlphaGo Zero



# Residual Networks

Defining a residual model with layers :

```
import tensorflow as tf
from tensorflow.keras import layers
input = layers.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 1, activation='relu', padding='same')(input)
ident = x
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = layers.add([ident,x])
x = layers.Flatten()(x)
x = layers.Dense(10, activation='softmax')(x)
model = tf.keras.models.Model(inputs=input, outputs=x)
```

# Residual Networks

Train a standard convolutional network on the CIFAR10 image dataset.

Compare it to a deep residual network.

# Residual Networks

```
import tensorflow as tf
from tensorflow.keras import layers
input = layers.Input(shape=(32, 32, 3))
x = layers.Conv2D(64, 1, activation='relu', padding='same')(input)
for i in range (5):
    ident = x
    x = layers.Conv2D(64, 3, activation='relu', padding='same')(x)
    x = layers.Conv2D(64, 3, activation='relu', padding='same')(x)
    x = layers.add([ident,x])
flatten = layers.Flatten()(x)
dense = layers.Dense(10, activation="softmax")(flatten)
model = tf.keras.models.Model(inputs=input, outputs=dense)
```

# Residual Networks

Let's train it:

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

model.compile(optimizer=tf.keras.optimizers.RMSprop(1e-3),
              loss='categorical_crossentropy',
              metrics=['acc'])
model.fit(x_train, y_train,
          batch_size=64,
          epochs=1,
          validation_split=0.2)
```

# Reusing Layers

# Shared Layers

- Another good use for the functional API are models that use shared layers. Shared layers are layer instances that get reused multiple times in a same model: they learn features that correspond to multiple paths in the graph-of-layers.

- Shared layers are often used to encode inputs that come from similar spaces (say, two different pieces of text that feature similar vocabulary), since they enable sharing of information across these different inputs, and they make it possible to train such a model on less data.

- If a given word is seen in one of the inputs, that will benefit the processing of all inputs that go through the shared layer.

# Shared Layers

- To share a layer in the Functional API, just call the same layer instance multiple times.
- For instance, here's an Embedding layer shared across two different text inputs:

```
# Embedding for 1000 unique words mapped to 128-dimensional vectors
shared_embedding = layers.Embedding(1000, 128)

# Variable-length sequence of integers
text_input_a = keras.Input(shape=(None,), dtype='int32')

# Variable-length sequence of integers
text_input_b = keras.Input(shape=(None,), dtype='int32')

# We reuse the same layer to encode both inputs
encoded_input_a = shared_embedding(text_input_a)
encoded_input_b = shared_embedding(text_input_b)
```

# Transfer Learning

- Because the graph of layers you are manipulating in the Functional API is a static datastructure, it can be accessed and inspected.
- This means that we can access the activations of intermediate layers ("nodes" in the graph) and reuse them elsewhere.
- This is extremely useful for feature extraction.
- This is a VGG16 model with weights pre-trained on ImageNet:

```
from tensorflow.keras.applications import VGG16
vgg16 = VGG16()
```

# Transfer Learning

- And these are the intermediate activations of the model, obtained by querying the graph datastructure:

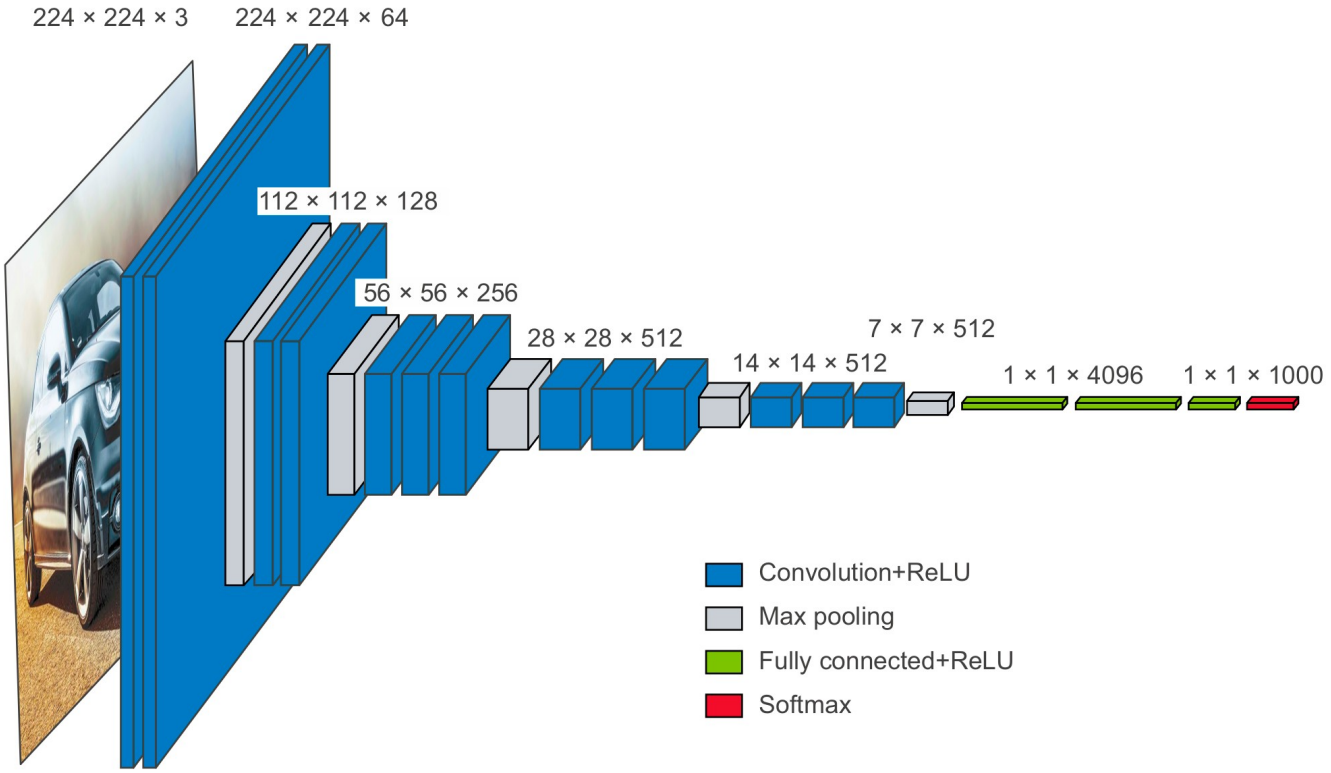
```
features_list = [layer.output for layer in vgg16.layers]
```

- We can use these features to create a new feature-extraction model, that returns the values of the intermediate layer activations -- and we can do all of this in 3 lines.

```
feat_extraction_model = keras.Model(inputs=vgg16.input, outputs=features_list)
img = np.random.random((1, 224, 224, 3)).astype('float32')
extracted_features = feat_extraction_model(img)
```

# Transfer Learning

VGG16 :



# Exercise

- Reuse the first layers of VGG16 to train a model with a different head on CIFAR10.

```
from tensorflow.keras.applications.vgg16 import VGG16

model = VGG16(include_top=False,
              input_shape=(32,32,3))
```

# Transfer Learning

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
model = tf.keras.applications.vgg16.VGG16(include_top=False,
                                          input_shape = (32,32,3))
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = tf.keras.applications.vgg16.preprocess_input(X_train)
X_test = tf.keras.applications.vgg16.preprocess_input(X_test)
y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)
```

# Transfer Learning

```
model.trainable = False
input_layer = model.input
x = model(input_layer)
x = tf.keras.layers.Flatten()(x)
x = tf.keras.layers.Dense(256, activation = 'relu')(x)
x = tf.keras.layers.Dense(10, activation = 'softmax')(x)

new_model = tf.keras.models.Model(inputs = input_layer, outputs = x)
new_model.summary()
```

# Transfer Learning

```
new_model.compile(loss ='categorical_crossentropy',
optimizer = 'adam', metrics = ['acc'])
```

```
history = new_model.fit(X_train, y_train,
validation_split = 0.1, epochs = 5)
```

```
new_model.evaluate(X_test, y_test)
```

# Lambda Layers

- Lambda layers enable to define layers without trainable weights.
- They are defined with Python code and can be used as a new layer inside a network.
- Lambda layers are best suited for simple operations or quick experimentation.
- For more advanced use cases, subclassing keras.layers.Layer is preferred.
- One reason for this is that when saving a Model, Lambda layers are saved by serializing the Python bytecode, whereas subclassed Layers are saved via overriding their get_config method and are thus more portable.

# Lambda Layers

```
# add a x -> x^2 layer
model.add(Lambda(lambda x: x ** 2))
```

# Lambda Layers

```
# add a layer that returns the concatenation
# of the positive part of the input and
# the opposite of the negative part
def antirectifier(x):
    x -= K.mean(x, axis=1, keepdims=True)
    x = K.l2_normalize(x, axis=1)
    pos = K.relu(x)
    neg = K.relu(-x)
    return K.concatenate([pos, neg], axis=1)

model.add(Lambda(antirectifier))
```

# Exercise

- Write a lambda layer that flips a board from left to right.

# Lambda Layers

```
def flip(x):
    x = tf.reverse (x, [-1])
    return x

model.add(Lambda(flip))
```

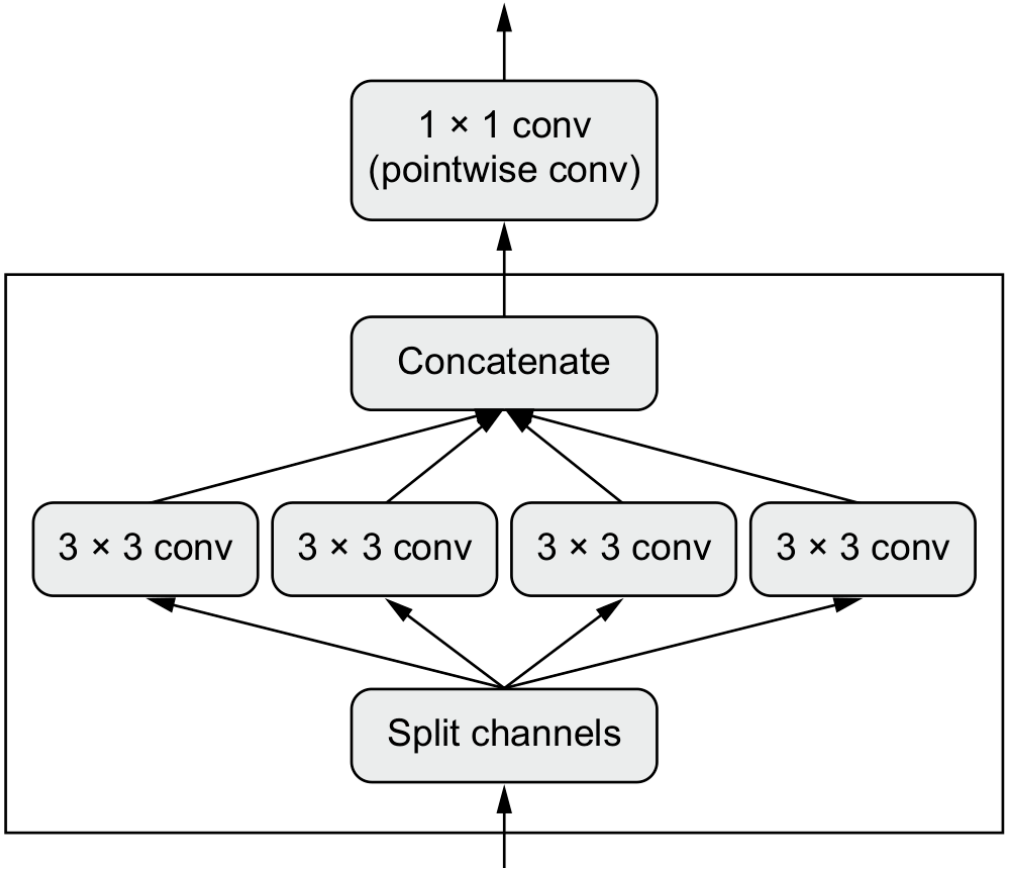
# Mobile Networks

# Mobile Networks

- Convolution:
  - All input planes are connected to all output planes.
- Depthwise convolution:
  - Each input plane is connected to one output plane.

# Mobile Networks

Depthwise convolution:



# Mobile Networks

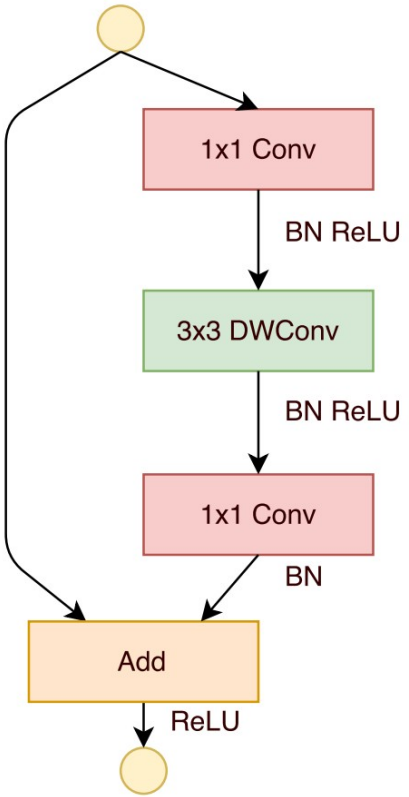
- Depthwise convolution:
  - Smaller model (fewer trainable weight parameters).
  - Leaner model (fewer floating-point operations), not yet efficient on GPU.
  - Better accuracy.
  - More memory if many planes.

# Mobile Networks

- Residual block:
  - Two convolutional layers per block.

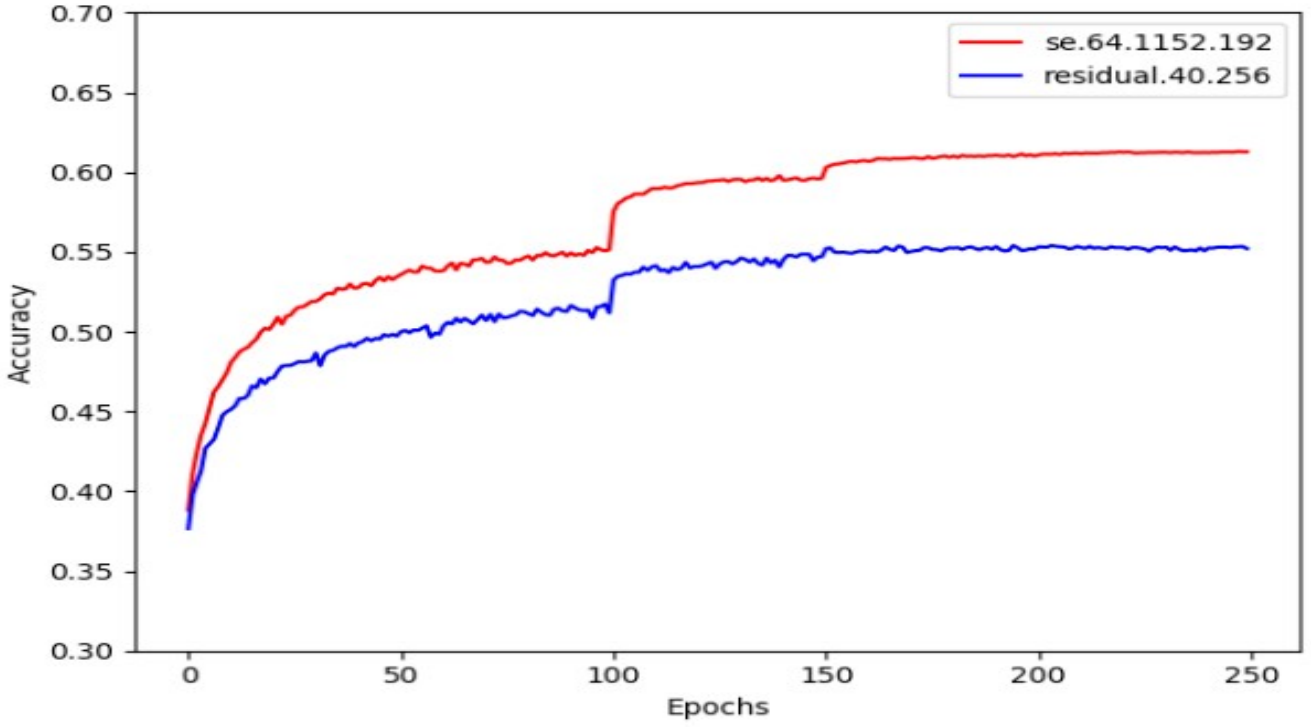
- Inverted residual block:
  - 1x1 convolutions tranform n planes of the trunk in 6*n planes of the block.
  - Depthwise convolution.
  - 1x1 convolutions tranform 6*n planes of the block in n planes of the trunk.

# Mobile Networks

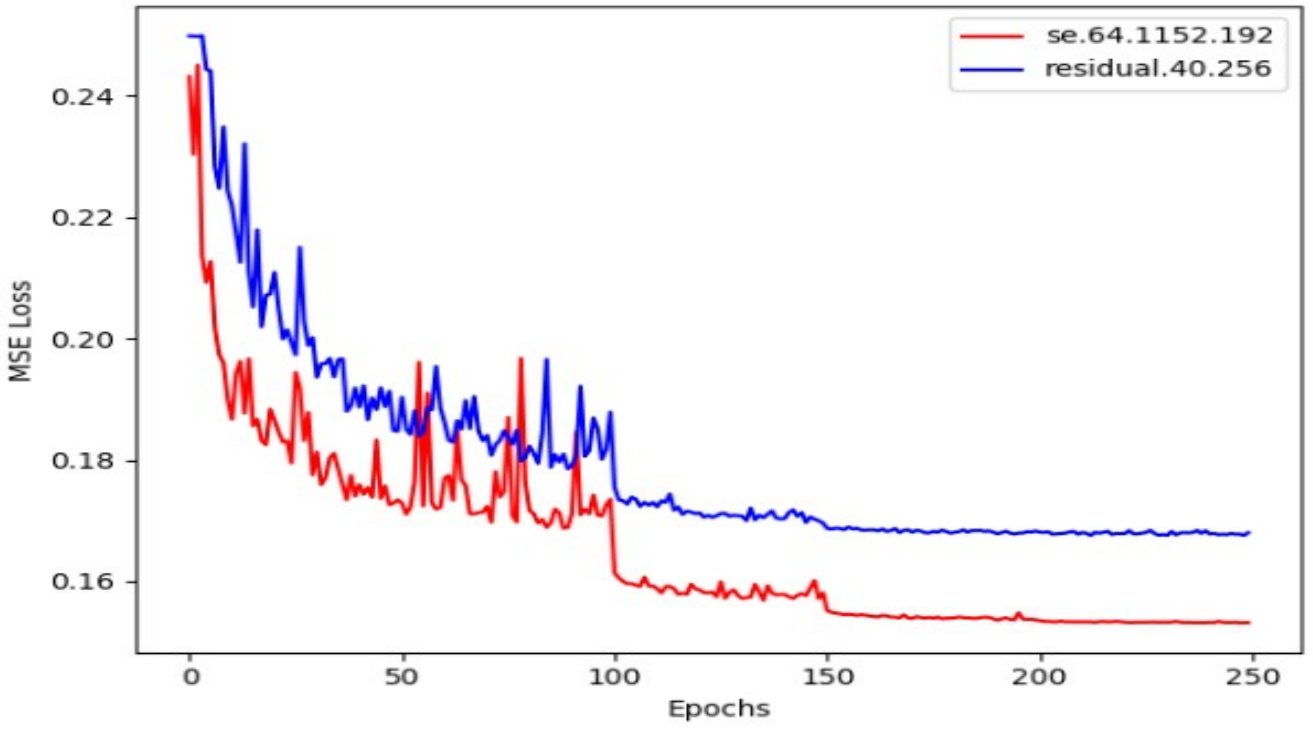



(a)

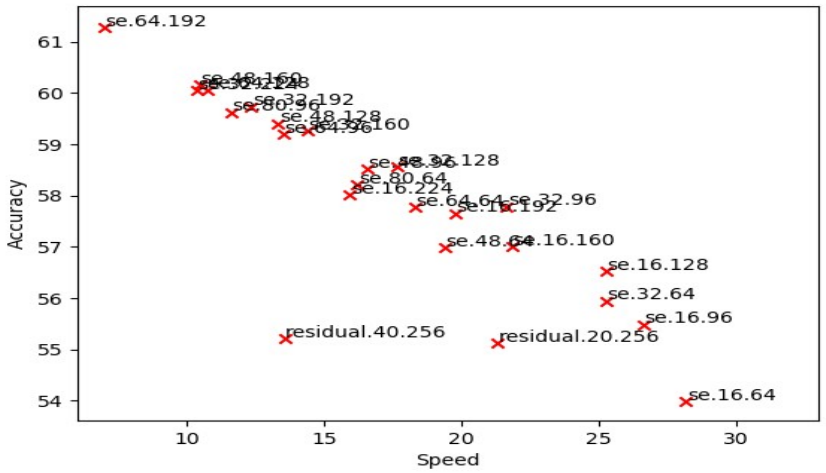
# Mobile Networks

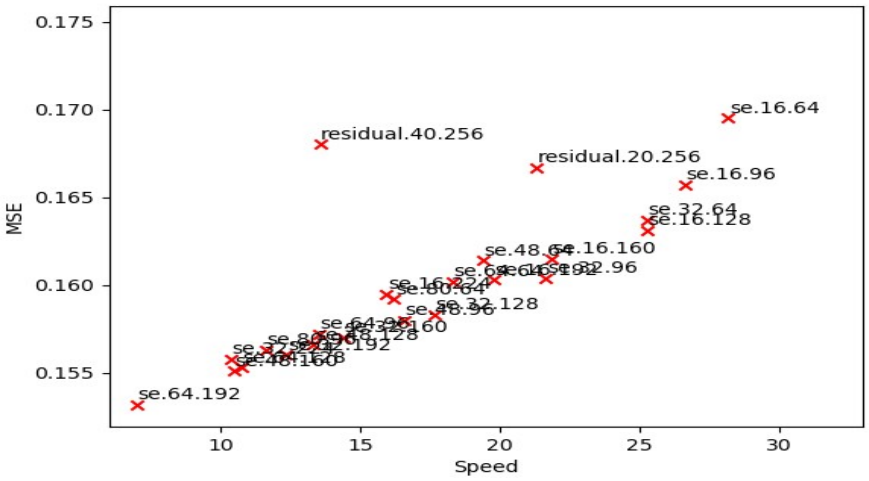


# Mobile Networks



# Mobile Networks



# Mobile Networks

- A MobileNet v2 is composed of inverted residual blocks:

- 1×1 Convolution with 1->6 filters followed by Batch Normalization and ReLU
- 3×3 DepthWise Convolution followed by Batch Normalization and ReLU
- 1×1 Convolution with 6->1 filters followed by Batch Normalization

- The tensor of the output of the block and the shortcut connection are then added.

- Exercise :
  Define an inverted residual block.
  Define a MobileNet v2 network with a tower of inverted residual blocks.
  Train the network on CIFAR 10.

# Mobile Networks

```
def bottleneck_block(x, expand=96, squeeze=16):
    m = layers.Conv2D(expand, (1,1), kernel_regularizer=regularizers.l2(0.0001), use_bias = False)(x)
    m = layers.BatchNormalization()(m)
    m = layers.Activation('relu')(m)
    m = layers.DepthwiseConv2D((3,3), padding='same', kernel_regularizer=regularizers.l2(0.0001), use_bias = False)(m)
    m = layers.BatchNormalization()(m)
    m = layers.Activation('relu')(m)
    m = layers.Conv2D(squeeze, (1,1), kernel_regularizer=regularizers.l2(0.0001), use_bias = False)(m)
    m = layers.BatchNormalization()(m)

    return layers.Add()([m, x])
```

# Mobile Networks

```
import tensorflow as tf
from tensorflow.keras import layers
input = layers.Input(shape=(32, 32, 3))
x = layers.Conv2D(16, 1, activation='relu', padding='same')(input)
for i in range (5):
    x = bottleneck_block(x)
flatten = layers.Flatten()(x)
dense = layers.Dense(10, activation="softmax")(flatten)
model = tf.keras.models.Model(inputs=input, outputs=dense)
```

# Mobile Networks

Let's train it:

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

model.compile(optimizer=tf.keras.optimizers.RMSprop(1e-3),
              loss='categorical_crossentropy',
              metrics=['acc'])
model.fit(x_train, y_train,
          batch_size=64,
          epochs=1,
          validation_split=0.2)
```

# Convnext

# Convnext

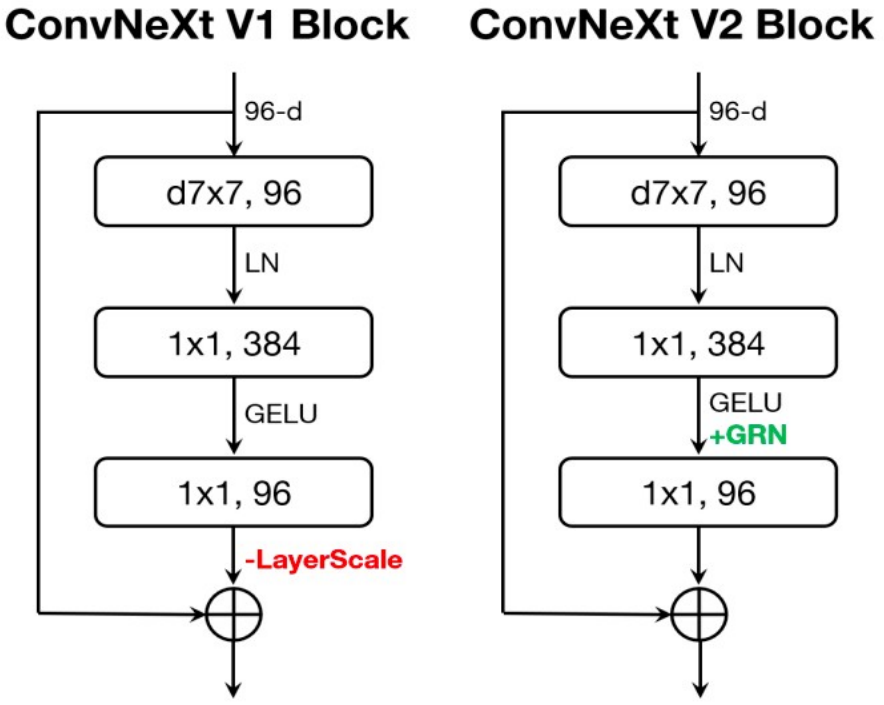



Figure 5. **ConvNeXt Block Designs**. In ConvNeXt V2, we add the GRN layer after the dimension-expansion MLP layer and drop LayerScale [41] as it becomes redundant.

# Convnext

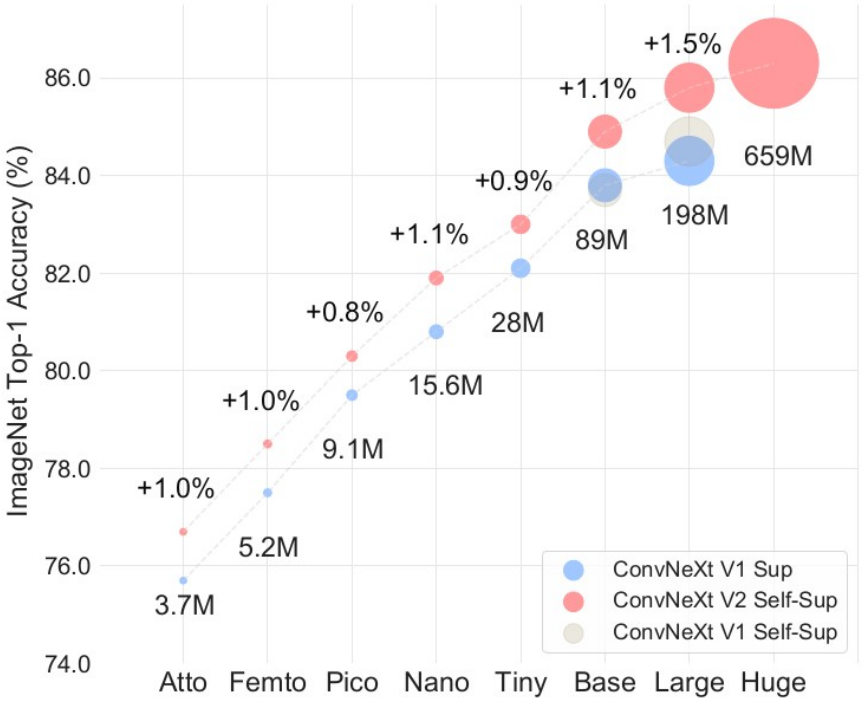



Figure 1. **ConvNeXt V2 model scaling**. The ConvNeXt V2 model, which has been pre-trained using our fully convolutional masked autoencoder framework, performs significantly better than the previous version across a wide range of model sizes.

# Convnext

| Backbone | Method | #param | PT epoch | FT acc. |
|---|---|---|---|---|
| ViT-B | BEiT | 88M | 800 | 83.2 |
| ViT-B | MAE | 88M | 1600 | 83.6 |
| Swin-B | SimMIM | 88M | 800 | 84.0 |
| ConvNeXt V2-B | FCMAE | 89M | 800 | 84.6 |
| ConvNeXt V2-B | FCMAE | 89M | 1600 | **84.9** |
| ViT-L | BEiT | 307M | 800 | 85.2 |
| ViT-L | MAE | 307M | 1600 | 85.9 |
| Swin-L | SimMIM | 197M | 800 | 85.4 |
| ConvNeXt V2-L | FCMAE | 198M | 800 | 85.6 |
| ConvNeXt V2-L | FCMAE | 198M | 1600 | **85.8** |
| ViT-H | MAE | 632M | 1600 | 86.9 |
| Swin V2-H | SimMIM | 658M | 800 | 85.7 |
| ConvNeXt V2-H | FCMAE | 659M | 800 | 85.8 |
| ConvNeXt V2-H | FCMAE | 659M | 1600 | **86.3** |

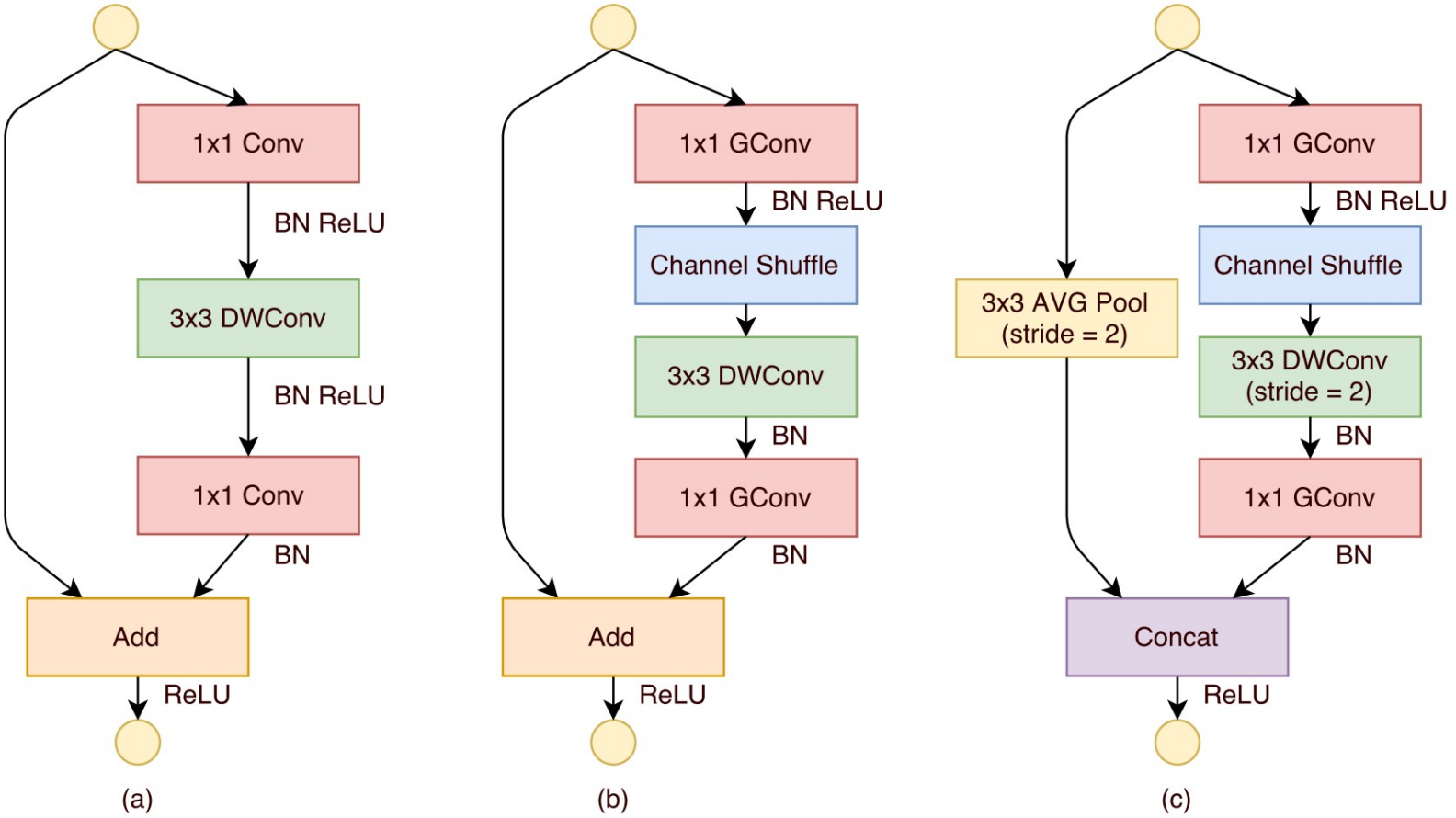
Table 4. **Comparisons with previous masked image modeling approaches**. The pre-training data is the IN-1K training set. All self-supervised methods are benchmarked by the end-to-end fine-tuning performance with an image size of 224. We underline the highest accuracy for each model size and bold our best results.

# Convnext v1

```
for i in range (blocks):
    x1 = tf.keras.layers.DepthwiseConv2D((7,7), padding='same', use_bias = False)(x)
    x1 = layers.LayerNormalization()(x1)
    x1 = layers.Conv2D(4 * filters, 1, padding='same', activation='gelu')(x1)
    x1 = layers.Conv2D(filters, 1, padding='same')(x1)
    x = layers.add([x1,x])
x = tf.keras.layers.BatchNormalization()(x)
```

# Shufflenet

# Mobile Networks / Shufflenet



# Shufflenet

- The main branch of the block consists of:

- 1×1 Group Convolution with 1/6 filters followed by Batch Normalization and ReLU
- Channel Shuffle
- 3×3 DepthWise Convolution followed by Batch Normalization
- 1×1 Group Convolution followed by Batch Normalization

- The tensors of the main branch and the shortcut connection are then concatenated and a ReLU activation is applied to the output.

# Shufflenet

- Channel Shuffle shuffles the channels of the tensor:
  1. reshape the input tensor
     - from width x height x channels
     - to width x height x groups x (channels/groups)
  2. permute the last two dimensions
  3. reshape the tensor to the original shape

# Shufflenet

- Import the necessary layers
- Write a function for the overall architecture
- Write a function for the Shufflenet block
- Write a function for the Group Convolution
- Write a function for the Channel Shuffle

# Shufflenet

```
from tensorflow.keras.layers import Input, Conv2D, DepthwiseConv2D, Dense, Concatenate, Add, ReLU, BatchNormalization,
                                    AvgPool2D, MaxPool2D, GlobalAveragePooling2D, Reshape, Permute, Lambda, Flatten, Activation

def getModel ():
    input = keras.Input(shape=(19, 19, planes), name='board')
    x = Conv2D(trunk, 1, padding='same', kernel_regularizer=regularizers.l2(0.0001))(input)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    for i in range (blocks):
        x = bottleneck_block (x, filters, trunk)
    policy_head = Conv2D(1, 1, activation='relu', padding='same', use_bias = False, kernel_regularizer=regularizers.l2(0.0001))(x)
    policy_head = Flatten()(policy_head)
    policy_head = Activation('softmax', name='policy')(policy_head)
    value_head = GlobalAveragePooling2D()(x)
    value_head = Dense(50, activation='relu', kernel_regularizer=regularizers.l2(0.0001))(value_head)
    value_head = Dense(1, activation='sigmoid', name='value', kernel_regularizer=regularizers.l2(0.0001))(value_head)

    model = keras.Model(inputs=input, outputs=[policy_head, value_head])
    return model
```

# Shufflenet

```
def bottleneck_block(tensor, expand=96, squeeze=16):
    x = gconv(tensor, channels=expand, groups=4)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = channel_shuffle(x, groups=4)
    x = DepthwiseConv2D(kernel_size=3, padding='same')(x)
    x = BatchNormalization()(x)

    x = gconv(x, channels=squeeze, groups=4)
    x = BatchNormalization()(x)

    x = Add()([tensor, x])
    output = ReLU()(x)
    return output
```

# Shufflenet

```
def gconv(tensor, channels, groups):
    input_ch = tensor.get_shape().as_list()[-1]
    group_ch = input_ch // groups
    output_ch = channels // groups
    groups_list = []

    for i in range(groups):
        group_tensor = tensor[:, :, :, i * group_ch: (i+1) * group_ch]
        group_tensor = Conv2D(output_ch, 1)(group_tensor)
        groups_list.append(group_tensor)

    output = Concatenate()(groups_list)
    return output
```

# Shufflenet

```
def channel_shuffle(x, groups):
    _, width, height, channels = x.get_shape().as_list()
    group_ch = channels // groups

    x = Reshape([width, height, group_ch, groups])(x)
    x = Permute([1, 2, 4, 3])(x)
    x = Reshape([width, height, channels])(x)
    return x
```

# Defining New Layers

# Defining New Layers

- The Layer class
- Layers encapsulate a state (weights) and some computation
- The main data structure you'll work with is the Layer. A layer encapsulates both a state (the layer's "weights") and a transformation from inputs to outputs (a "call", the layer's forward pass).

# Defining New Layers

```
from tensorflow.keras import layers

class Linear(layers.Layer):
  def __init__(self, units=32, input_dim=32):
    super(Linear, self).__init__()
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(initial_value=w_init(shape=(input_dim, units), dtype='float32'), trainable=True)
    b_init = tf.zeros_initializer()
    self.b = tf.Variable(initial_value=b_init(shape=(units,), dtype='float32'), trainable=True)

  def call(self, inputs):
    return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)

tf.Tensor(
[[-0.03533589 -0.02663077 -0.0507721  -0.02178559]
 [-0.03533589 -0.02663077 -0.0507721  -0.02178559]], shape=(2, 4), dtype=float32)
```

# Defining New Layers

Note you also have access to a quicker shortcut for adding weight to a layer: the add_weight method:

```
class Linear(layers.Layer):

  def __init__(self, units=32, input_dim=32):
    super(Linear, self).__init__()
    self.w = self.add_weight(shape=(input_dim, units), initializer='random_normal', trainable=True)
    self.b = self.add_weight(shape=(units,), initializer='zeros', trainable=True)

  def call(self, inputs):
    return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)

tf.Tensor(
[[-0.03276853 -0.02655794  0.10825785  0.00806852]
 [-0.03276853 -0.02655794  0.10825785  0.00806852]], shape=(2, 4), dtype=float32)
```

# Defining New Layers

Layers can have non-trainable weights

Besides trainable weights, you can add non-trainable weights to a layer as well. Such weights are meant not to be taken into account during backpropagation, when you are training the layer.

Here's how to add and use a non-trainable weight:

```
class ComputeSum(layers.Layer):

  def __init__(self, input_dim):
    super(ComputeSum, self).__init__()
    self.total = tf.Variable(initial_value=tf.zeros((input_dim,)), trainable=False)

  def call(self, inputs):
    self.total.assign_add(tf.reduce_sum(inputs, axis=0))
    return self.total

x = tf.ones((2, 2))
my_sum = ComputeSum(2)
y = my_sum(x)
print(y.numpy())
y = my_sum(x)
print(y.numpy())
```

```
[2. 2.]
[4. 4.]
```

# Defining New Layers

Best practice: deferring weight creation until the shape of the inputs is known

In the logistic regression example above, our Linear layer took an input_dim argument that was used to compute the shape of the weights w and b in __init__:

```
class Linear(layers.Layer):

  def __init__(self, units=32, input_dim=32):
      super(Linear, self).__init__()
      self.w = self.add_weight(shape=(input_dim, units),
                               initializer='random_normal',
                               trainable=True)
      self.b = self.add_weight(shape=(units,),
                               initializer='zeros',
                               trainable=True)
```

# Defining New Layers

In many cases, you may not know in advance the size of your inputs, and you would like to lazily create weights when that value becomes known, some time after instantiating the layer.

It is better to create layer weights in the build(inputs_shape) method of your layer. Like this:

```
class Linear(layers.Layer):

  def __init__(self, units=32):
    super(Linear, self).__init__()
    self.units = units

  def build(self, input_shape):
    self.w = self.add_weight(shape=(input_shape[-1], self.units), initializer='random_normal', trainable=True)
    self.b = self.add_weight(shape=(self.units,), initializer='random_normal', trainable=True)

  def call(self, inputs):
    return tf.matmul(inputs, self.w) + self.b
```

The __call__ method of your layer will automatically run build the first time it is called. You now have a layer that's lazy and easy to use:

```
linear_layer = Linear(32)  # At instantiation, we don't know on what inputs this is going to get called
y = linear_layer(x)  # The layer's weights are created dynamically the first time the layer is called
```

# Defining New Layers

- Layers are recursively composable
- If you assign a Layer instance as attribute of another Layer, the outer layer will start tracking the weights of the inner layer.
- We recommend creating such sublayers in the __init__ method (since the sublayers will typically have a build method, they will be built when the outer layer gets built).

# Defining New Layers

```
# Let's assume we are reusing the Linear class
# with a `build` method that we defined above.

class MLPBlock(layers.Layer):

  def __init__(self):
    super(MLPBlock, self).__init__()
    self.linear_1 = Linear(32)
    self.linear_2 = Linear(32)
    self.linear_3 = Linear(1)

  def call(self, inputs):
    x = self.linear_1(inputs)
    x = tf.nn.relu(x)
    x = self.linear_2(x)
    x = tf.nn.relu(x)
    return self.linear_3(x)

mlp = MLPBlock()
y = mlp(tf.ones(shape=(3, 64)))  # The first call to the `mlp` will create the weights
```

# Defining New Layers

The Model class

In general, you will use the Layer class to define inner computation blocks, and will use the Model class to define the outer model -- the object you will train.

For instance, in a ResNet50 model, you would have several ResNet blocks subclassing Layer, and a single Model encompassing the entire ResNet50 network.

The Model class has the same API as Layer, with the following differences:

- It exposes built-in training, evaluation, and prediction loops (model.fit(), model.evaluate(), model.predict()).
- It exposes the list of its inner layers, via the model.layers property.
- It exposes saving and serialization APIs.

# Defining New Layers

Effectively, the "Layer" class corresponds to what we refer to in the literature as a "layer" (as in "convolution layer" or "recurrent layer") or as a "block" (as in "ResNet block" or "Inception block").

Meanwhile, the "Model" class corresponds to what is referred to in the literature as a "model" (as in "deep learning model") or as a "network" (as in "deep neural network").

For instance, we could take our mini-resnet example above, and use it to build a Model that we could train with fit(), and that we could save with save_weights.

# Defining New Layers

```
class ResNet(tf.keras.Model):

    def __init__(self):
        super(ResNet, self).__init__()
        self.block_1 = ResNetBlock()
        self.block_2 = ResNetBlock()
        self.global_pool = layers.GlobalAveragePooling2D()
        self.classifier = Dense(num_classes)

    def call(self, inputs):
        x = self.block_1(inputs)
        x = self.block_2(x)
        x = self.global_pool(x)
        return self.classifier(x)


resnet = ResNet()
dataset = ...
resnet.fit(dataset, epochs=10)
resnet.save_weights(filepath)
```

# Exercise

- Write a class for a Resnet model and test it on CIFAR10.

# Defining New Layers

```
class ResNetBlock(tf.keras.layers.Layer):
  def __init__(self, kernel_size=3):
    super(ResNetBlock, self).__init__()
    self.kernel_size = (kernel_size, kernel_size)

  def build(self, input_shape):
    #print(input_shape)
    self.convol_1 = tf.keras.layers.Conv2D(filters = input_shape[-1], kernel_size = self.kernel_size,
                                           padding='same', activation='relu',
                                           input_shape=input_shape)
    self.convol_2 = tf.keras.layers.Conv2D(filters = input_shape[-1], kernel_size = self.kernel_size,
                                           padding='same', activation='relu',
                                           input_shape=input_shape)
  def call(self, inputs):
    x = self.convol_1(inputs)
    x = self.convol_2(x)
    x = tf.keras.layers.add([x, inputs])
    return x
```

# Defining New Layers

```
class ResNet(tf.keras.Model):
  def __init__(self, num_classes, f):
    super(ResNet, self).__init__()
    self.num_classes = num_classes
    self.f = f

  def build(self, input_shape):
    self.conv = tf.keras.layers.Conv2D(filters = self.f, kernel_size= (3,3), input_shape=input_shape)
    self.block_1 = ResNetBlock()
    self.block_2 = ResNetBlock()
    self.classifier = tf.keras.layers.Dense(self.num_classes, activation='softmax')

  def call(self, inputs):
    x = self.conv(inputs)
    x = self.block_1(x)
    x = self.block_2(x)
    x = tf.keras.layers.Flatten()(x)
    return self.classifier(x)
```

# Defining New Layers

```
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = X_train.astype('float32')/255.
X_test = X_test.astype('float32')/255.
y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)

model = ResNet(10, 64)

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc'])

history = model.fit(X_train, y_train, epochs = 10, validation_split = 0.1, batch_size = 256)
```

# Defining New Layers

Putting it all together: an end-to-end example

Here's what you've learned so far:

- A Layer encapsulate a state (created in __init__ or build) and some computation (in call).
- Layers can be recursively nested to create new, bigger computation blocks.
- Layers can create and track losses (typically regularization losses).

- The outer container, the thing you want to train, is a Model.
- A Model is just like a Layer, but with added training and serialization utilities.

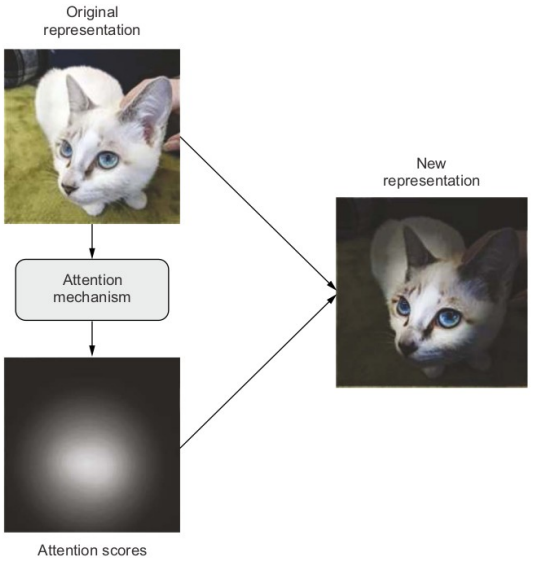
# Transformers

# Transformers

- Ashish Vaswani et al., “Attention is all you need” (2017)
  - https://arxiv.org/abs/1706.03762
- Neural attention for sequence models:
  - Self attention
  - Multi-head attention
  - Transformer encoder
  - Positional embedding
  - Application: Text classification
- Sequence to sequence learning:
  - Application: Machine translation

# Transformers

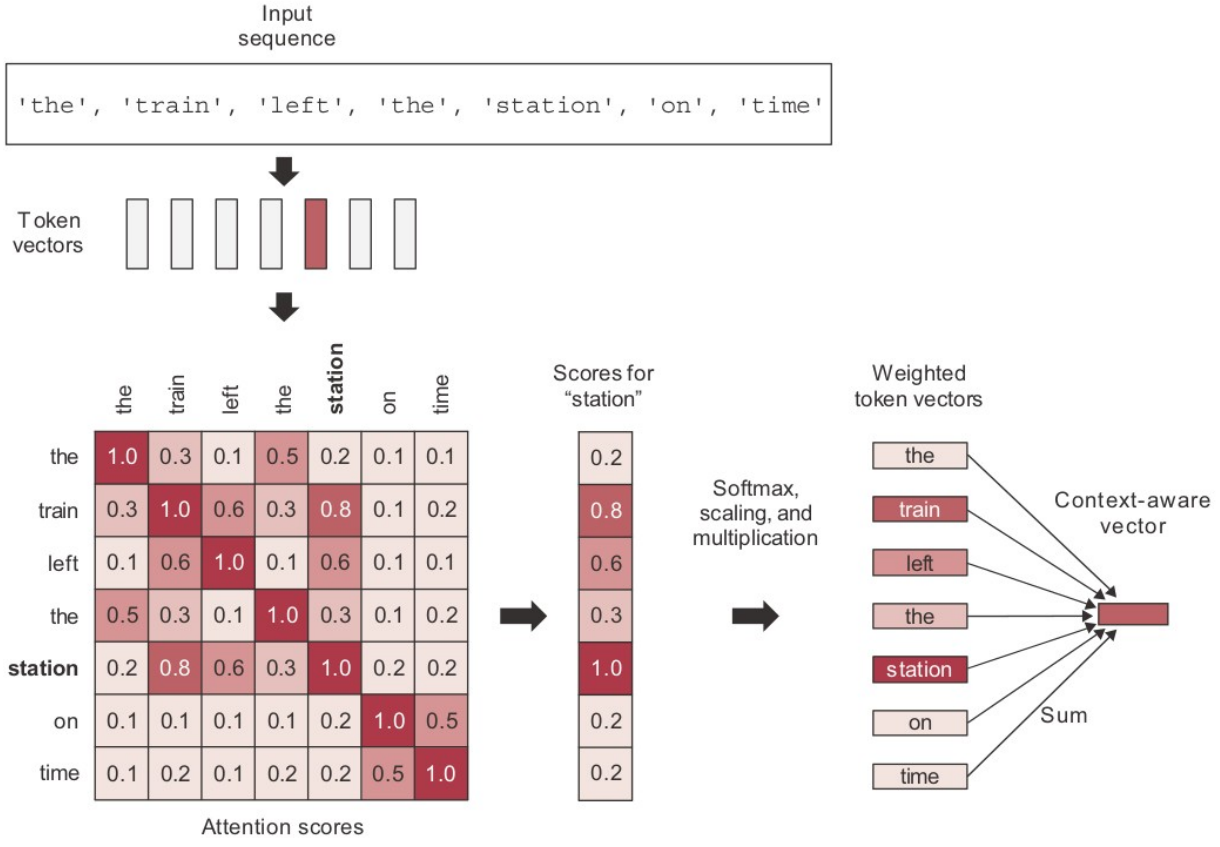
- Self attention:
  - Importance of features
  - Higher scores for important features



# Transformers

- The context is important to evaluate of features.
- Example: The meaning of pronouns like “he,” “it,” “in,” etc., is entirely sentence-specific.
- The purpose of self-attention is to modulate the representation of a token by using the representations of related tokens in the sequence.
- Consider an example sentence: “The train left the station on time.”
- Let’s see how self-attention deals with the ‘station’ word.

# Transformers

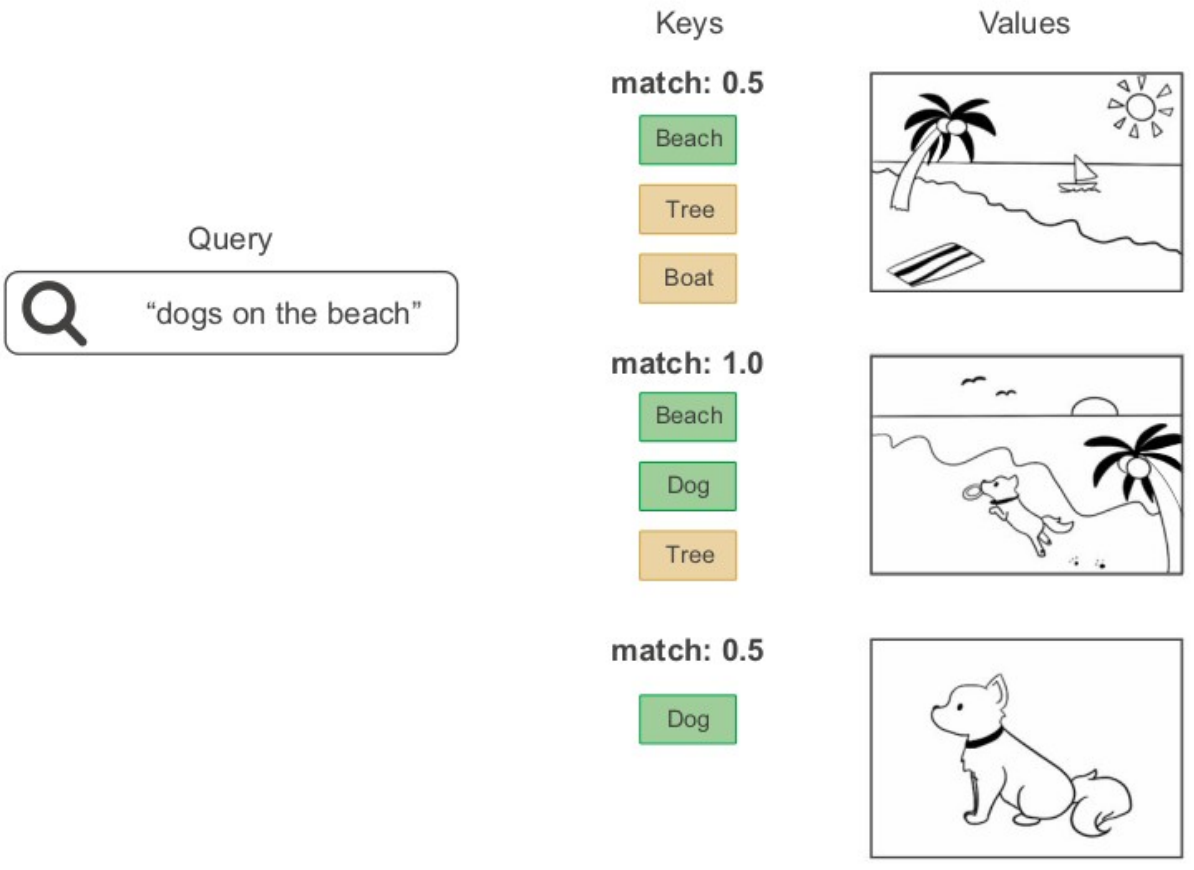


# Transformers

- Use the dot product between two word vectors as a measure of the strength of their relationship:

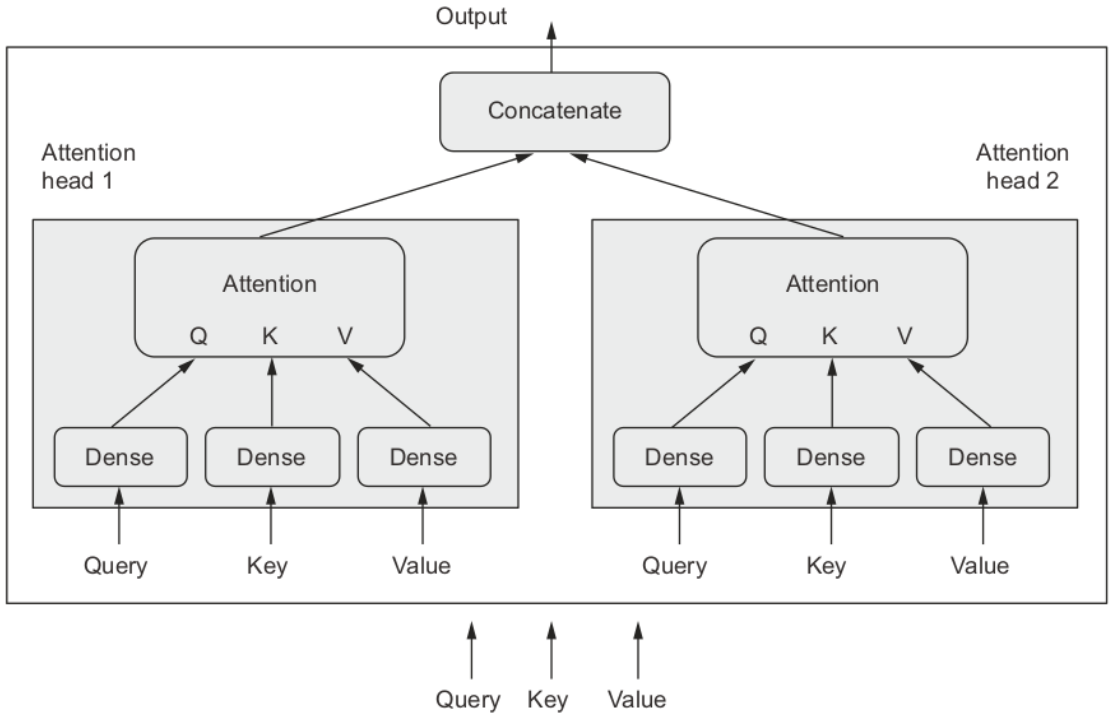
```
def self_attention(input_sequence):
    output = np.zeros(shape=input_sequence.shape)
    for i, pivot_vector in enumerate(input_sequence):
        scores = np.zeros(shape=(len(input_sequence),))
        for j, vector in enumerate(input_sequence):
            scores[j] = np.dot(pivot_vector, vector.T)
        scores /= np.sqrt(input_sequence.shape[1])
        scores = softmax(scores)
        new_pivot_representation = np.zeros(shape=pivot_vector.shape)
        for j, vector in enumerate(input_sequence):
            new_pivot_representation += vector * scores[j]
        output[i] = new_pivot_representation
    return output
```

# Transformers



# Transformers

- MultiHeadAttention:



# Transformers

- Mobile Networks:
  - Each channel is learned independently.
- Grouped convolutions:
  - Separation into independent groups.
- Multi-head attention:
  - Each attention head is independent.

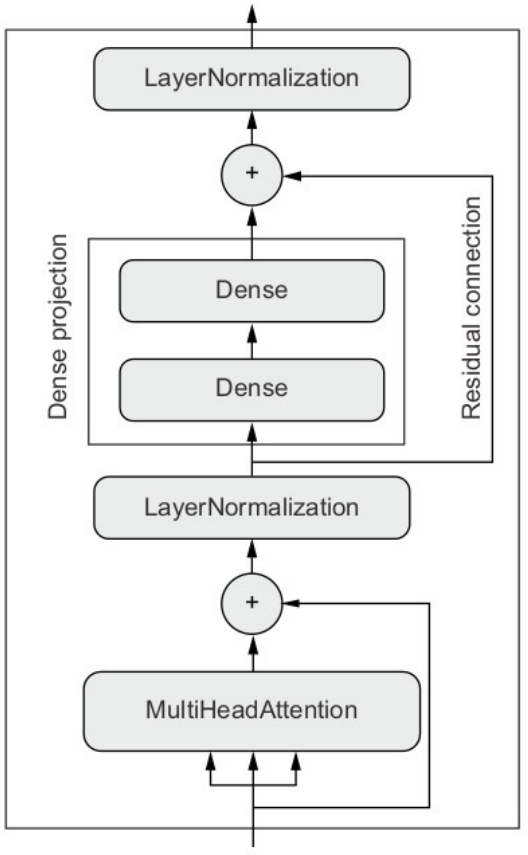
# Transformers

- With Keras:

```
num_heads = 4
embed_dim = 256
mha_layer = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
outputs = mha_layer(inputs, inputs, inputs)
```

# Transformers

- The Transformer Encoder:



# Transformers

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention = layers.MultiHeadAttention (num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential([layers.Dense(dense_dim, activation="relu"), layers.Dense(embed_dim),])
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
```

# Transformers

```
def call(self, inputs, mask=None):
    if mask is not None:
        mask = mask[:, tf.newaxis, :]
    attention_output = self.attention(inputs, inputs, attention_mask=mask)
    proj_input = self.layernorm_1(inputs + attention_output)
    proj_output = self.dense_proj(proj_input)
    return self.layernorm_2(proj_input + proj_output)

def get_config(self):
    config = super().get_config()
    config.update({"embed_dim": self.embed_dim,
                   "num_heads": self.num_heads,
                   "dense_dim": self.dense_dim,})
    return config
```

# Transformers

- Preparing the IMDB dataset:

```
!curl -O https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
!rm -r aclImdb/train/unsup

import os, pathlib, shutil, random

base_dir = pathlib.Path("aclImdb")
val_dir = base_dir / "val"
train_dir = base_dir / "train"
for category in ("neg", "pos"):
    os.makedirs(val_dir / category)
    files = os.listdir(train_dir / category)
    random.Random(1337).shuffle(files)
    num_val_samples = int(0.2 * len(files))
    val_files = files[-num_val_samples:]
    for fname in val_files:
        shutil.move(train_dir / category / fname, val_dir / category / fname)

from tensorflow import keras
batch_size = 32
train_ds = keras.utils.text_dataset_from_directory("aclImdb/train", batch_size=batch_size)
val_ds = keras.utils.text_dataset_from_directory("aclImdb/val", batch_size=batch_size)
test_ds = keras.utils.text_dataset_from_directory("aclImdb/test", batch_size=batch_size)
```

# Transformers

- Preparing the IMDB dataset:

```
text_vectorization = layers.TextVectorization(max_tokens=20000,output_mode="multi_hot",)
text_only_train_ds = train_ds.map(lambda x, y: x)
text_vectorization.adapt(text_only_train_ds)

from tensorflow.keras import layers

max_length = 600
max_tokens = 20000
text_vectorization = layers.TextVectorization(max_tokens=max_tokens, output_mode="int", output_sequence_length=max_length,)
text_vectorization.adapt(text_only_train_ds)

int_train_ds = train_ds.map(lambda x, y: (text_vectorization(x), y), num_parallel_calls=4)
int_val_ds = val_ds.map(lambda x, y: (text_vectorization(x), y), num_parallel_calls=4)
int_test_ds = test_ds.map(lambda x, y: (text_vectorization(x), y), num_parallel_calls=4)
```

# Transformers

- Using the Transformer encoder for text classification:

```
vocab_size = 20000
embed_dim = 256
num_heads = 2
dense_dim = 32
inputs = keras.Input(shape=(None,), dtype="int64")
x = layers.Embedding(vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)

model.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])
callbacks = [keras.callbacks.ModelCheckpoint("transformer_encoder.keras", save_best_only=True)]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20, callbacks=callbacks)
model = keras.models.load_model("transformer_encoder.keras", custom_objects={"TransformerEncoder": TransformerEncoder})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

# Positional Embedding

# Transformers

- Positional encoding:
  - Add a position vector to the embedding.
  - Positional Embedding = embedding of the position.

# Transformers

- Positional embedding:

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding(input_dim=input_dim, output_dim=output_dim)
        self.position_embeddings = layers.Embedding(input_dim=sequence_length, output_dim=output_dim)
        self.sequence_length = sequence_length
        self.input_dim = input_dim
        self.output_dim = output_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

    def compute_mask(self, inputs, mask=None):
        return tf.math.not_equal(inputs, 0)

    def get_config(self):
        config = super().get_config()
        config.update({"output_dim": self.output_dim,"sequence_length": self.sequence_length, "input_dim": self.input_dim,})
        return config
```

# Transformers

- Combining the Transformer encoder with positional embedding:

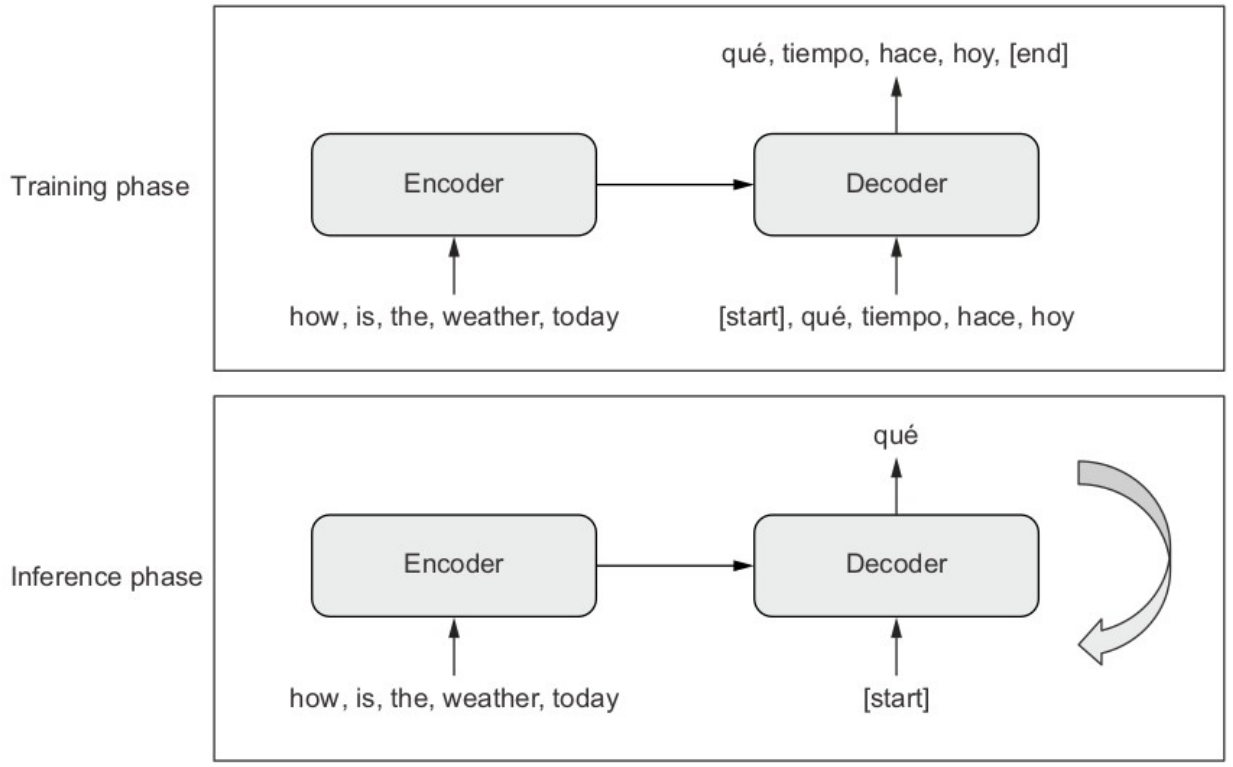
```
vocab_size = 20000
sequence_length = 600
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

callbacks = [keras.callbacks.ModelCheckpoint("full_transformer_encoder.keras", save_best_only=True)]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20, callbacks=callbacks)
model = keras.models.load_model("full_transformer_encoder.keras", custom_objects={"TransformerEncoder": TransformerEncoder,
                                                                                   "PositionalEmbedding": PositionalEmbedding})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

# Sequence to Sequence Learning

# Transformers

- Sequence to sequence learning:
  - Machine translation
  - Text summarization
  - Question answering
  - Chatbots
  - Text generation
  - Etc.

# Transformers



# Transformers

- English to Spanish dataset:

```
!wget http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip
!unzip -q spa-eng.zip
text_file = "spa-eng/spa.txt"
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
for line in lines:
    english, spanish = line.split("\t")
    spanish = "[start] " + spanish + " [end]"
    text_pairs.append((english, spanish))
import random
random.shuffle(text_pairs)
num_val_samples = int(0.15 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples:num_train_samples + num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples:]
```

# Transformers

- Vectorizing the English and Spanish text pairs:

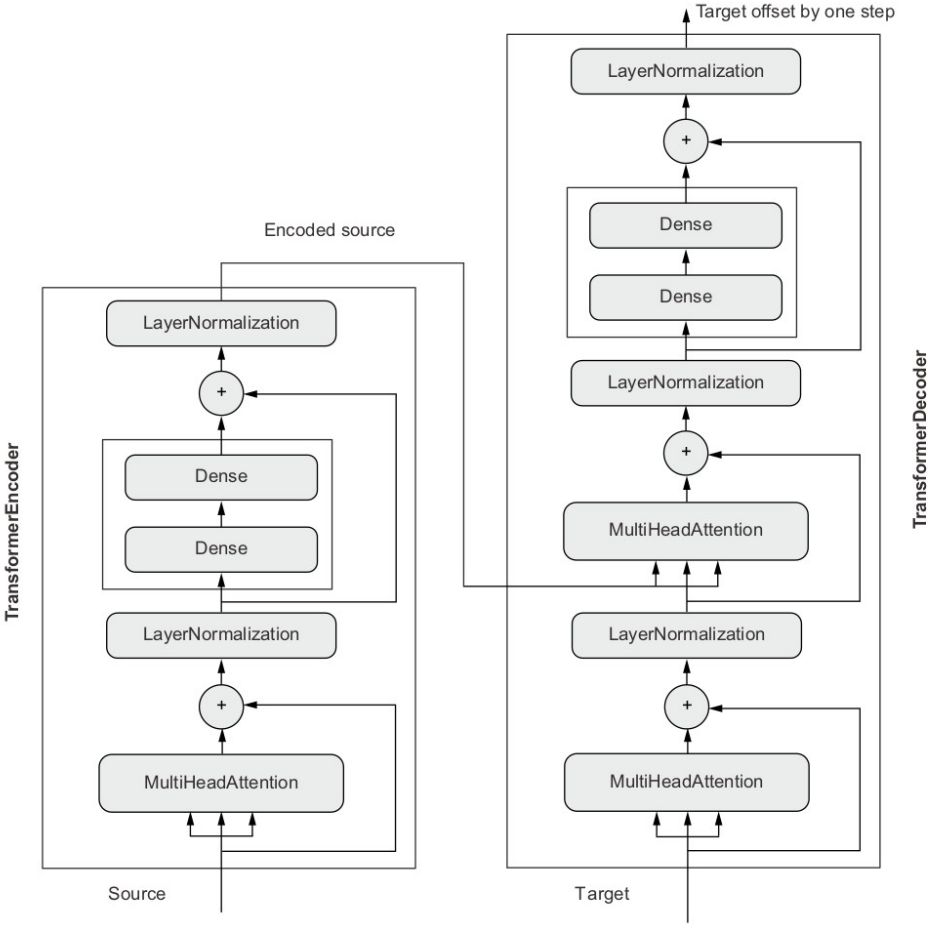
```
import tensorflow as tf
import string
import re
strip_chars = string.punctuation + "¿"
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(lowercase, f"[{re.escape(strip_chars)}]", "")

vocab_size = 15000
sequence_length = 20

source_vectorization = layers.TextVectorization(max_tokens=vocab_size, output_mode="int", output_sequence_length=sequence_length,)
target_vectorization = layers.TextVectorization(max_tokens=vocab_size, output_mode="int", output_sequence_length=sequence_length + 1,standardize=custom_standardization,)
train_english_texts = [pair[0] for pair in train_pairs]
train_spanish_texts = [pair[1] for pair in train_pairs]
source_vectorization.adapt(train_english_texts)
target_vectorization.adapt(train_spanish_texts)
```

# Transformers

- Preparing datasets for the translation task:

```
batch_size = 64
def format_dataset(eng, spa):
    eng = source_vectorization(eng)
    spa = target_vectorization(spa)
    return ({"english": eng, "spanish": spa[:, :-1],}, spa[:, 1:])

def make_dataset(pairs):
    eng_texts, spa_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    spa_texts = list(spa_texts)
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, spa_texts))
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(format_dataset, num_parallel_calls=4)
    return dataset.shuffle(2048).prefetch(16).cache()

train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)
```

# Transformers



# Transformers

- The Transformer Decoder:

```
class TransformerDecoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention_1 = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
        self.attention_2 = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential([layers.Dense(dense_dim, activation="relu"), layers.Dense(embed_dim),])
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.layernorm_3 = layers.LayerNormalization()
        self.supports_masking = True
    def get_config(self):
        config = super().get_config()
        config.update({"embed_dim": self.embed_dim, "num_heads": self.num_heads, "dense_dim": self.dense_dim,})
        return config
```

# Transformers

- Causal padding is absolutely critical to successfully training a sequence-to-sequence Transformer.
- The TransformerDecoder is order-agnostic: it looks at the entire target sequence at once.
- If it were allowed to use its entire input, it would simply learn to copy input step N+1 to location N in the output.
- The model would achieve perfect training accuracy.
- Mask the upper half of the pairwise attention matrix to prevent the model from paying any attention to information from the future
- get_causal_attention_mask(self, inputs) method

# Transformers

- The Transformer Decoder:

```
def get_causal_attention_mask(self, inputs):
    input_shape = tf.shape(inputs)
    batch_size, sequence_length = input_shape[0], input_shape[1]
    i = tf.range(sequence_length)[:, tf.newaxis]
    j = tf.range(sequence_length)
    mask = tf.cast(i >= j, dtype="int32")
    mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))
    mult = tf.concat([tf.expand_dims(batch_size, -1), tf.constant([1, 1], dtype=tf.int32)], axis=0)
    return tf.tile(mask, mult)
```

# Transformers

- The Transformer Decoder:

```
def call(self, inputs, encoder_outputs, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs)
    if mask is not None:
        padding_mask = tf.cast(mask[:, tf.newaxis, :], dtype="int32")
        padding_mask = tf.minimum(padding_mask, causal_mask)
    attention_output_1 = self.attention_1(query=inputs, value=inputs, key=inputs, attention_mask=causal_mask)
    attention_output_1 = self.layernorm_1(inputs + attention_output_1)
    attention_output_2 = self.attention_2(query=attention_output_1, value=encoder_outputs, key=encoder_outputs,
                                          attention_mask=padding_mask,)
    attention_output_2 = self.layernorm_2(attention_output_1 + attention_output_2)
    proj_output = self.dense_proj(attention_output_2)
    return self.layernorm_3(attention_output_2 + proj_output)
```

# Transformers

- End to end Transformer:

```
embed_dim = 256
dense_dim = 2048
num_heads = 8

encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="english")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)
encoder_outputs = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)

decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="spanish")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, dense_dim, num_heads)(x, encoder_outputs)
x = layers.Dropout(0.5)(x)
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
transformer = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)

transformer.compile(optimizer="rmsprop", loss="sparse_categorical_crossentropy", metrics=["accuracy"])
transformer.fit(train_ds, epochs=30, validation_data=val_ds)
```

# Transformers

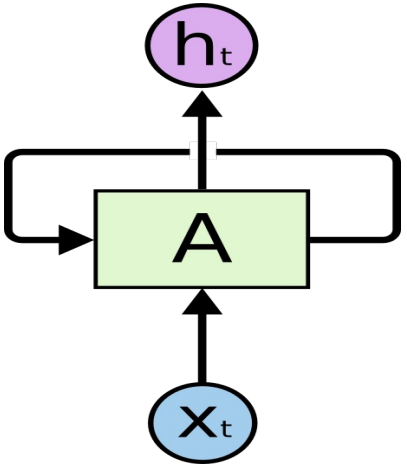
- Translating new sentences:

```
import numpy as np
spa_vocab = target_vectorization.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])[:, :-1]
        predictions = transformer([tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(20):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(decode_sequence(input_sentence))
```

# LSTM

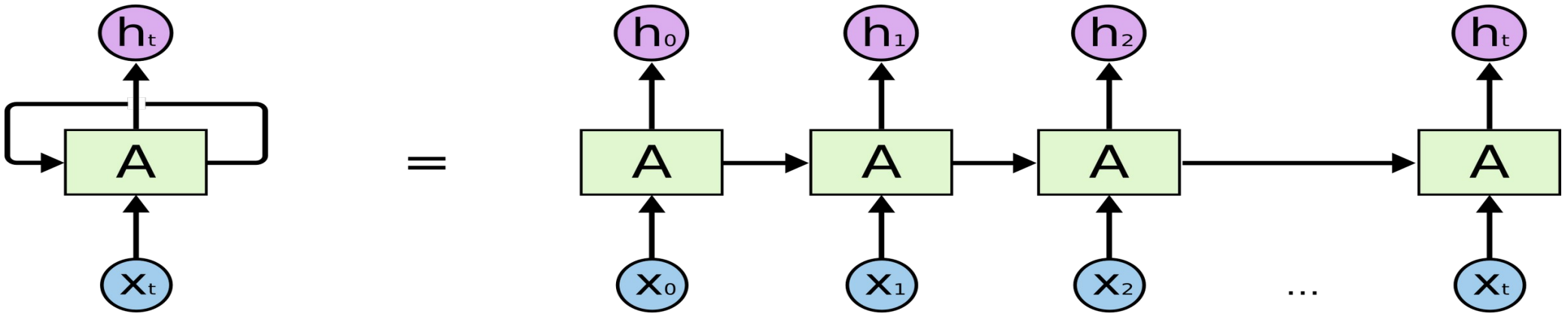
# LSTM

- Recurrent neural networks enable to memorize information about past events.
- They have loops that allow information to persist.



# LSTM

- A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor :



# LSTM

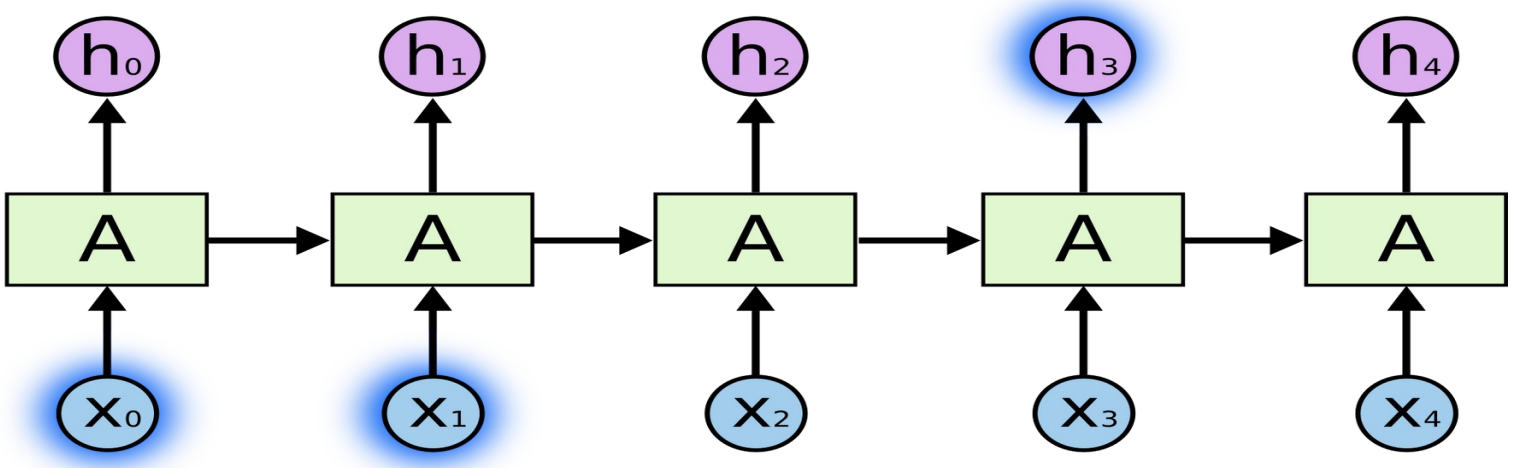
- RNNs have been successfully applied to a variety of problems:
- Speech recognition,
- Language modeling,
- Translation,
- Image captioning,
- ...

# LSTM

- LSTM is a very special kind of recurrent neural network.
- It works much better than the standard version.
- Many exciting results based on recurrent neural networks are achieved with them.
- There are more recent well performing layers such as GRU layers.

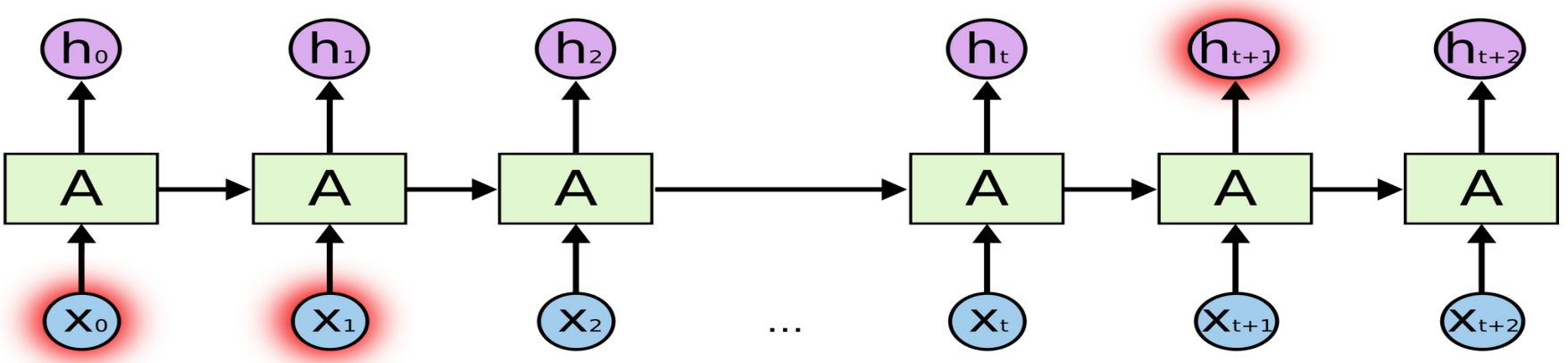
# LSTM

- Prediction of the next word based on the previous ones: “the clouds are in the ?”
- Relevant words are nearby, RNN work.



# LSTM

- As the gap grows, RNNs become unable to learn to connect the information:

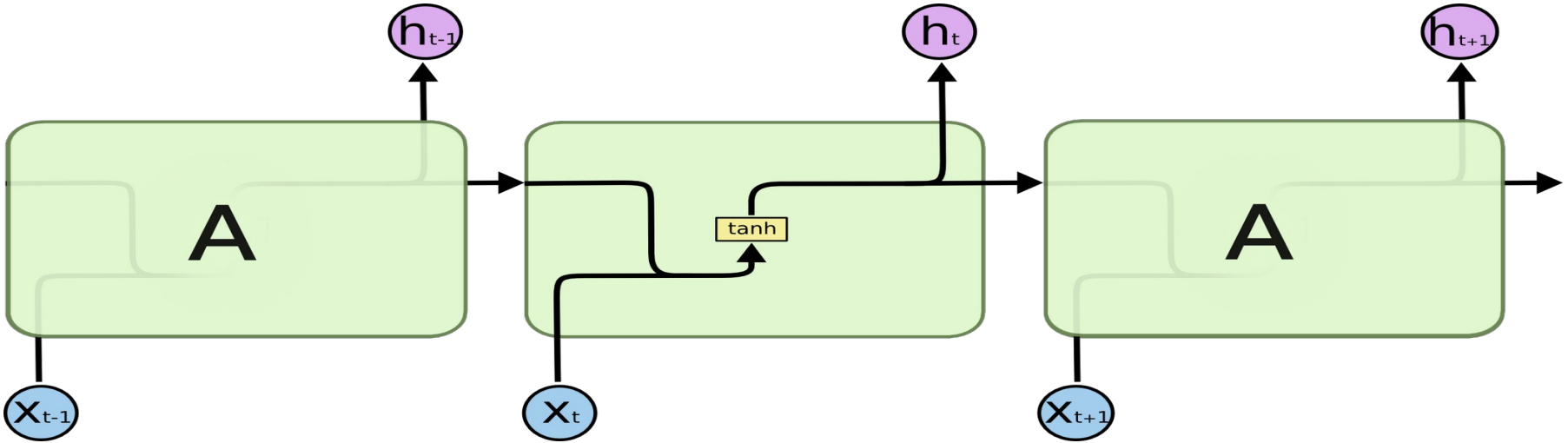


# LSTM

- Long Short Term Memory networks (LSTMs) are a special kind of RNN, capable of learning long-term dependencies.
- They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work.

# LSTM

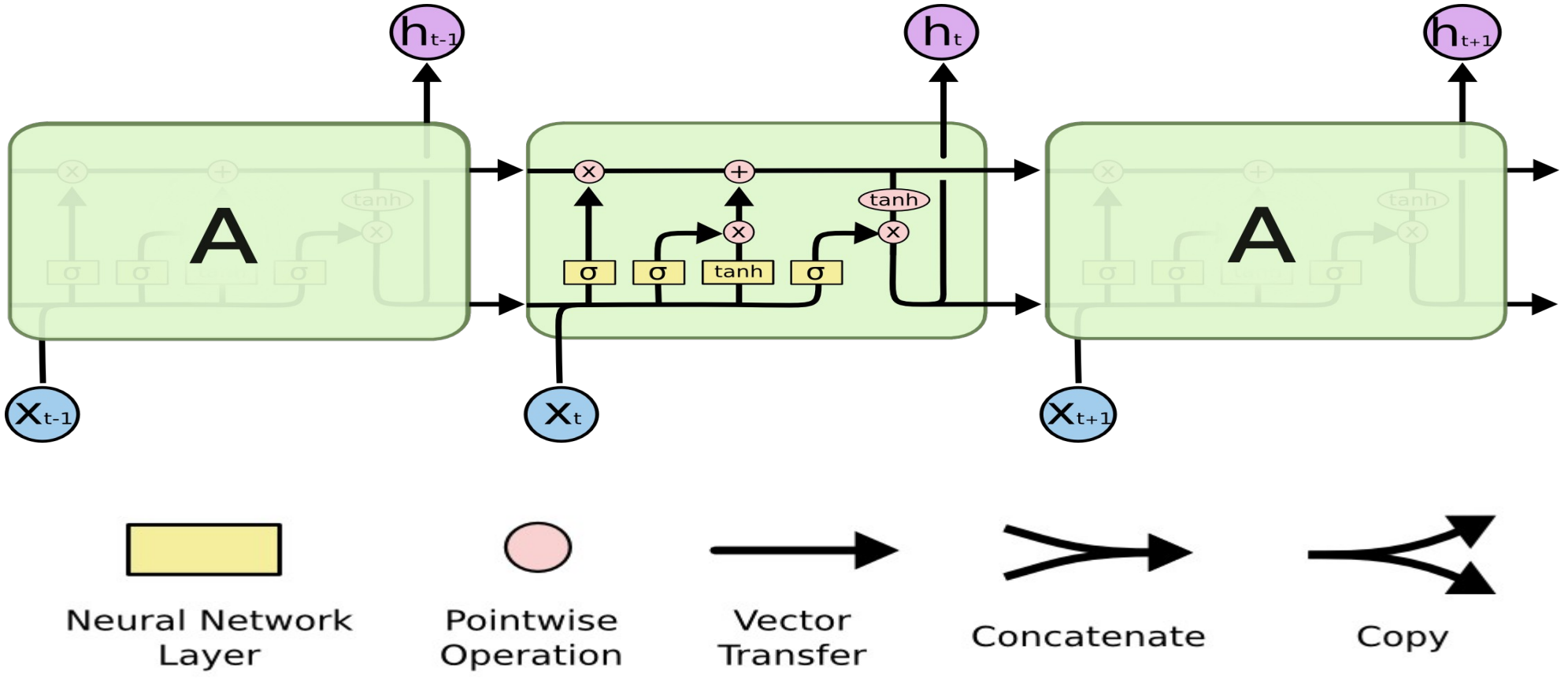
- In standard RNNs, the repeating module has a very simple structure, such as a single tanh layer.



# LSTM

- LSTMs also have this chain like structure, but the repeating module has a different structure.
- Instead of having a single neural network layer, there are four, interacting in a very special way.

# LSTM

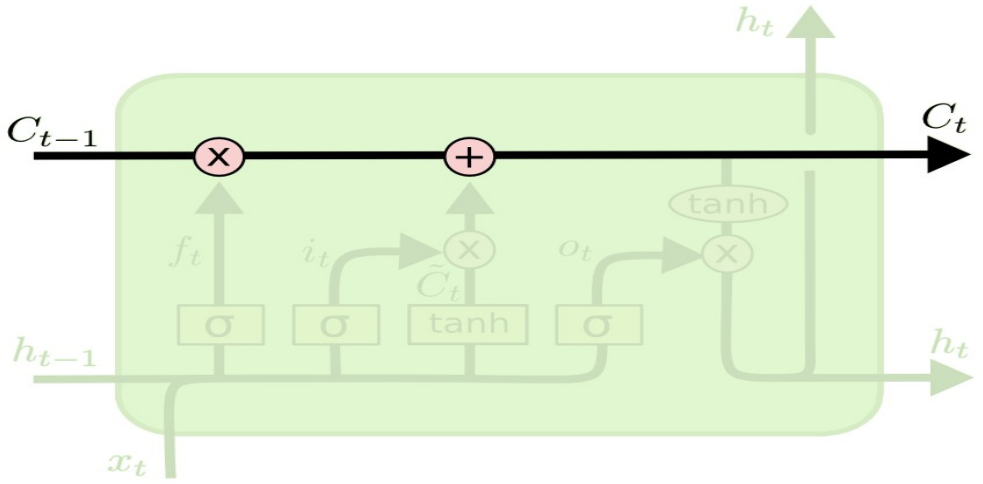

$h_{t-1}$
$h_t$
$h_{t+1}$
A
A
tanh
σ
σ
tanh
σ
tanh
$X_{t-1}$
$X_t$
$X_{t+1}$
Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

# LSTM

- The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.
- The cell state runs straight down the entire chain, with only some minor linear interactions.
- It’s very easy for information to just flow along it unchanged

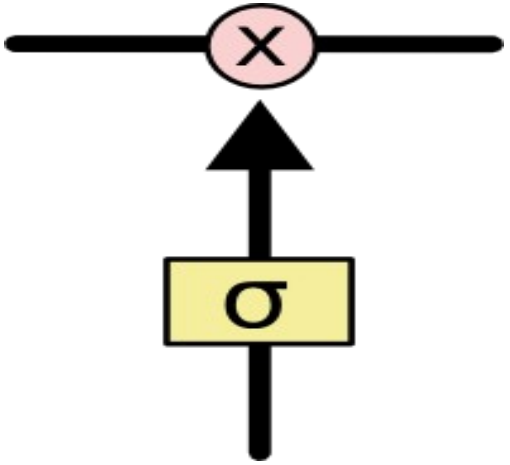
# LSTM

- Cell state :



# LSTM

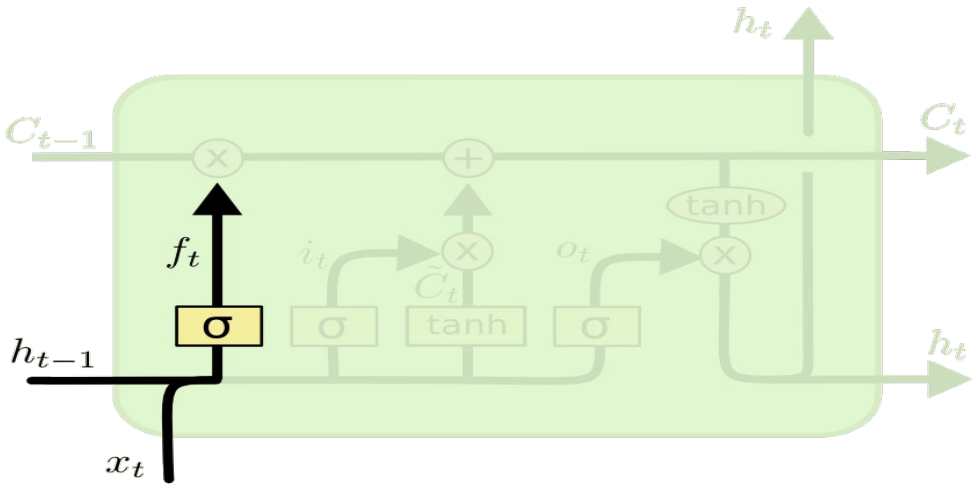
- LSTM remove or add information to the cell state using gates = sigmoid and multiplication:



# LSTM

- The forget gate layer looks at $h_{t-1}$ and $x_t$.
- It outputs a number between 0 and 1 for each number in the cell state $C_{t-1}$.
- 1 represents “completely keep this”
- 0 represents “completely get rid of this.”

# LSTM

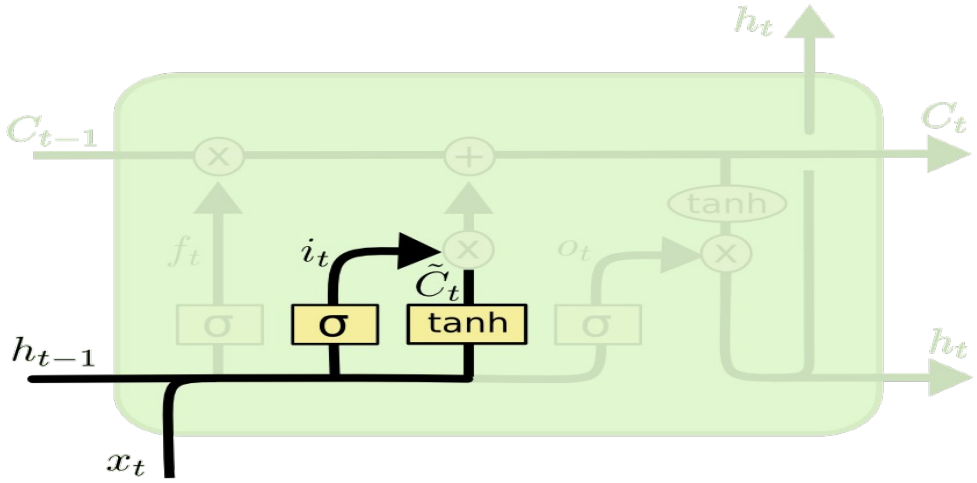



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

# LSTM

- The next step is to decide what new information to store in the cell state.
- A sigmoid layer called the “input gate layer” decides which values to update.
- A tanh layer creates a vector of new candidate values C~t that could be added to the state.

# LSTM



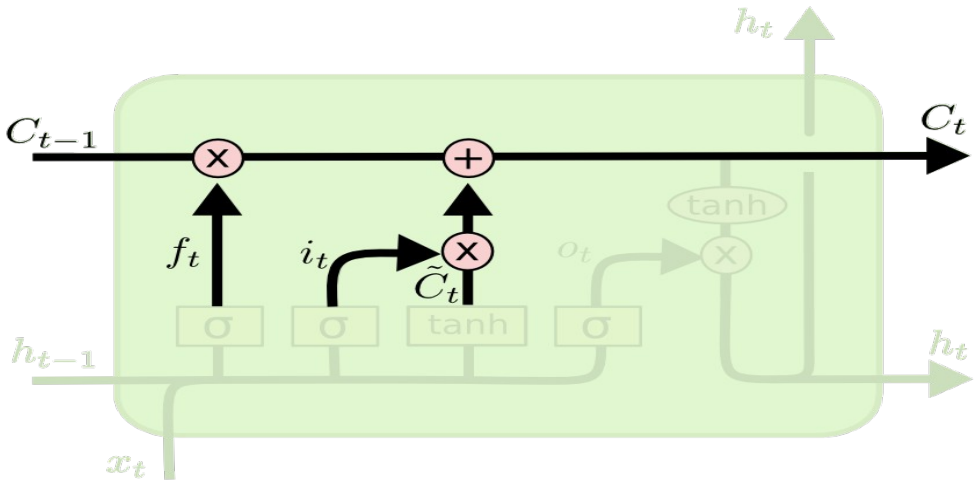

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

# LSTM

- We now update the old cell state, $C_{t-1}$, into the new cell state $C_t$.
- We multiply the old state by $f_t$, forgetting the things we decided to forget earlier.
- Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.

# LSTM

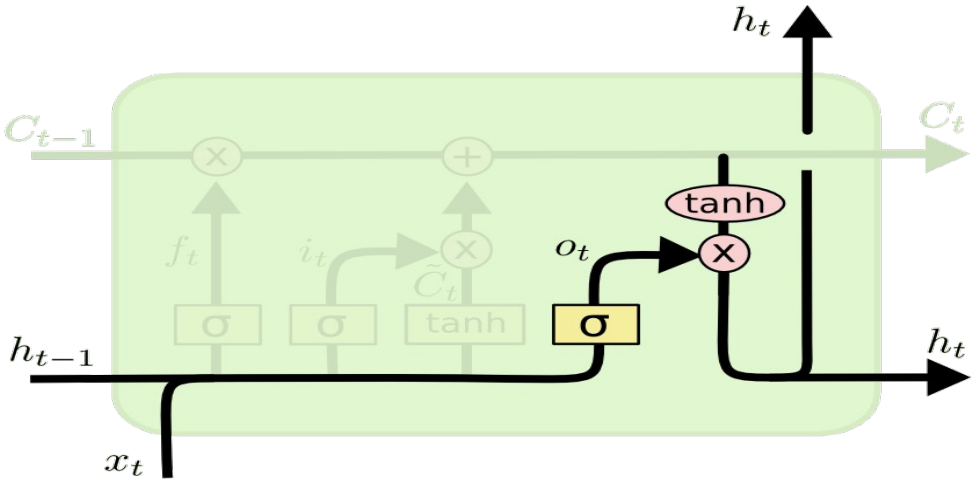



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

# LSTM

- Finally, we need to decide what we’re going to output.
- This output will be based on our cell state, but will be a filtered version.
- First, we run a sigmoid layer which decides what parts of the cell state we’re going to output.
- Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

# LSTM




$$o_t = \sigma\left(W_o\ [\,h_{t-1}, x_t]\ +\ b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

# Time Series

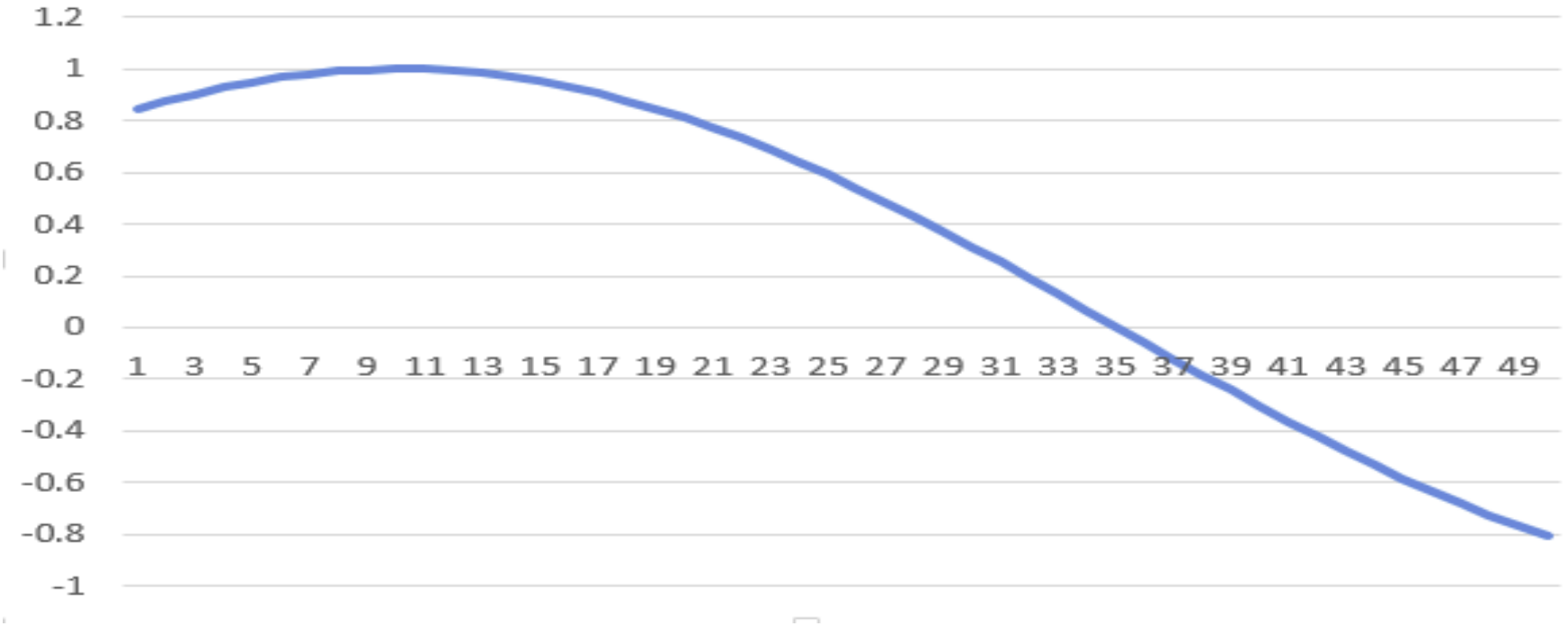
- We will start first with a simple sinus time series.
- www.lamsade.dauphine.fr/~cazenave/sinwave.csv
- We want the LSTM to learn the sin wave from a set window size of data.

# Time Series

- LSTM layers take a numpy array of 3 dimensions (N, W, F)
- N is the number of training sequences,
- W is the sequence length,
- F is the number of features of each sequence.

# Time Series

- Use sequences of length 50 :



# Time Series

Load the data from the csv file to a numpy array.
(www.lamsade.dauphine.fr/~cazenave/sinwave.csv)

```
f = open(filename, 'rb').read()
data = f.decode().split('\n')
```

Make an array of arrays of size 50.

Shuffle the data.

Separate training and test sets.

# Sinwave

```
import numpy as np

# lecture des donnees et découpage en sequence de seq_len elements
def load_data(filename, seq_len):
    f = open(filename, 'rb').read()
    data = np.array(f.decode().split('\n'), dtype = np.float32)

    sequence_length = seq_len + 1
    result = []
    for index in range(len(data) - sequence_length):
        result.append(data[index: index + sequence_length])

    result = np.array(result)
```

# Sinwave

```
    np.random.shuffle(result)
    row = round(0.9 * result.shape[0])
    train = result[:int(row), :] # 90 % des exemples pour l'apprentissage
    x_train = train[:, :-1] # On prend les séquences jusqu'à l'avant dernier élément
    y_train = train[:, -1] # On prend le dernier élément comme sortie à apprendre
    x_test = result[int(row):, :-1]
    y_test = result[int(row):, -1]

    # on transforme en un tenseur de dimension 3 avec une seule feature
    x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
    x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))

    return x_train, y_train, x_test, y_test

load_data ('sinwave.csv', 50)
```

# Keras LSTM

layers.LSTM (units, activation='tanh', recurrent_activation='hard_sigmoid', use_bias=True, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal', bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None, bias_constraint=None, dropout=0.0, recurrent_dropout=0.0)

units: Positive integer, dimensionality of the output space.

activation: Activation function to use (see activations). If you pass None, no activation is applied (ie. "linear" activation: a(x) = x).

return_sequences: Boolean. Whether to return the last output in the output sequence, or the full sequence.

# Keras LSTM

input_dim: dimensionality of the input (integer). This argument (or alternatively, the keyword argument input_shape) is required when using this layer as the first layer in a model.

input_length: Length of input sequences, to be specified when it is constant. This argument is required if you are going to connect Flatten then Dense layers upstream (without it, the shape of the dense outputs cannot be computed). Note that if the recurrent layer is not the first layer in your model, you would need to specify the input length at the level of the first layer (e.g. via the input_shape argument)

- Input shapes

  3D tensor with shape (batch_size, timesteps, input_dim), (Optional) 2D tensors with shape (batch_size, output_dim).

- Output shape

  if return_state: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape (batch_size, units).

  if return_sequences: 3D tensor with shape (batch_size, timesteps, units).

  else, 2D tensor with shape (batch_size, units).

# Time Series

- Build the network with 1 input layer (consisting of a sequence of size 50) which feeds into an LSTM layer with 50 neurons, that in turn feeds into another LSTM layer with 100 neurons which then feeds into a fully connected normal layer of 1 neuron with a linear activation function which will be used to give the prediction of the next time step.

# Time Series

Train the network.

Use the trained model to predict the sequence.

Plot the true data versus the prediction.

# Sinwave

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense,LSTM,Dropout

model = Sequential()
model.add(LSTM(50, input_shape=(50, 1), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(100, return_sequences=False))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(loss="mse", optimizer="rmsprop", metrics = ['mae'])
X_train, y_train, X_test, y_test = load_data('sinwave.csv', 50)
model.fit(X_train, y_train, batch_size=512, epochs=1, validation_split=0.05)
predict = model.predict (X_test)
```

# Time Series

Train the network on the sp500 data from 2000 to 2016 :

(www.lamsade.dauphine.fr/~cazenave/sp500.csv)

Normalize the window.

Take each n-sized window of training/testing data and normalize each one to reflect percentage changes from the start of that window (so the data at point i=0 will always be 0).

# Time Series

n = normalised list [window] of price changes

p = raw list [window] of adjusted daily return prices

Normalisation:

$$n_i = \left(\frac{p_i}{p_0}\right) - 1$$

De-Normalisation:

$$p_i = p_0(n_i + 1)$$

# Time Series

Train the network.

Use the trained model to predict the sequence.

Plot the true data versus the prediction.

# SP500

```
def normalise_windows(window_data):
  normalised_data = []
  for window in window_data:
      normalised_window = [((float(p) /
                  float(window[0])) - 1) for p in window]
      normalised_data.append(normalised_window)
  return normalised_data
```

# LSTM

- Generating texts.

  import keras

  import numpy as np

  path = keras.utils.get_file('nietzsche.txt', origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')

  text = open(path).read().lower()

  print('Corpus length:', len(text))

# LSTM

- Next, you'll extract partially overlapping sequences of length maxlen , one-hot encode them, and pack them in a 3D Numpy array x of shape (sequences, maxlen, unique_characters) .

- Simultaneously, you'll prepare an array y containing the corresponding targets: the one-hot-encoded characters that come after each extracted sequence.

# LSTM

```
maxlen = 60
step = 3
sentences = []
next_chars = []
for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i: i + maxlen])
    next_chars.append(text[i + maxlen])
print('Number of sequences:', len(sentences))
chars = sorted(list(set(text)))
print('Unique characters:', len(chars))
char_indices = dict((char, chars.index(char)) for char in chars)
print('Vectorization…')
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
    y[i, char_indices[next_chars[i]]] = 1
```

# LSTM

- Building the network
- This network is a single LSTM layer followed by a Dense classifier and softmax over all possible characters.
- But note that recurrent neural networks aren’t the only way to do sequence data generation;
- 1D convnets also have proven extremely successful at this task in recent times.

# LSTM

```
from keras import layers
model = keras.models.Sequential()
model.add(layers.LSTM(128, input_shape=(maxlen, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))

optimizer = keras.optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

# LSTM

Training the langage model and sampling from it

- Given a trained model and a seed text snippet, you can generate new text by doing the following repeatedly:
    1 Draw from the model a probability distribution for the next character, given the generated text available so far.
    2 Reweight the distribution to a certain temperature.
    3 Sample the next character at random according to the reweighted distribution.
    4 Add the new character at the end of the available text.

# LSTM

```
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

# LSTM

- Finally, a loop repeatedly trains and generates text.
- You begin generating text using a range of different temperatures after every epoch.
- This allows you to see how the generated text evolves as the model begins to converge, as well as the impact of temperature in the sampling strategy.

# LSTM

```
import random
import sys
for epoch in range(1, 60):
    model.fit(x, y, batch_size=128, epochs=1)
    print('epoch', epoch)
    start_index = random.randint(0, len(text) - maxlen - 1)
    generated_text = text[start_index: start_index + maxlen]
    for temperature in [0.2, 0.5, 1.0, 1.2]:
        sys.stdout.write(generated_text)
        print('\n\n------ temperature:', temperature)
        for i in range(400):
            sampled = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(generated_text):
                sampled[0, t, char_indices[char]] = 1.
            preds = model.predict(sampled, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_char = chars[next_index]
            generated_text += next_char
            generated_text = generated_text[1:]
            sys.stdout.write(next_char)
```