



# Neural networks

## Multi-Layer Perceptron



Michel.RIVEILL@univ-cotedazur.fr

# Main parameters for a neural network

## 1. Architecture of the network

- ▶ Input layer
- ▶ Hidden layer (number of layers / number of neurons / activation function)
- ▶ Output layer

## 2. Loss and metric function

- ▶ Metric: model evaluation criteria
- ▶ Cost/Loss: Function minimized during learning, must be derivable
- ▶ Ideally: Metric = Cost

## 3. Optimization algorithm

- ▶ Objective to converge as quickly as possible to the minimum of the cost function.
- ▶ Exercice sur le playground.



# ***Network architecture***

# Input / Output layers

---

- ▶ Input layer
  - ▶ Size: depends on the number of features
- ▶ Output layer
  - ▶ Regression
    - ▶ Activation: linear
    - ▶ Number of neurons depends on the number of variables to predict
      - 1 neuron by variable
  - ▶ Classification
    - ▶ Activation: softmax
    - ▶ Number of neurons depends on the number of class to predict
      - 2 classes: 1 neuron
      - $N > 2$  classes: 1 neuron by classes
        - Classes are one hot encoded

# Softmax

- ▶ Softmax takes an N-dimensional vector of real numbers and transforms it into a vector of real number with a sum equal to 1

- ▶  $p_i = \frac{e^i}{\sum_j e^j}$

- ▶ Its outputs could be interpreted as a probability distribution in classification

Category	Scoring function	unnormalized probabilities $UP = \exp^{ULP}$	normalized probabilities $P = \frac{UP}{\sum UP_j}$	normalized log loss $LL = -\ln(P)$
Dog	-3.44	0.0321	0.0006	7.4186
Cat	1.16	3.1899	0.0596	2.8201
Boat	-0.81	0.4449	0.0083	4.7915
Airplane	3.91	49.8990	0.9315	0.0709
			predict(X)	predict_proba(X)

# Loss and Metrics

- ▶ Loss is used for updating weights
- ▶ Metric is used for performance evaluation
  - ▶ It's better if you use only one criteria
  - ▶ Difficult to compare if there are multiple criteria
    - ▶ it is possible to compose the criteria into one:
    - ▶  $F(\text{Execution time, Memory space, Accuracy})$
- ▶ Regression
  - ▶ Loss: MAE, MSE, RMSE, etc.
  - ▶ Metrics: generally, the same
- ▶ Classification
  - ▶ Loss: cross entropy
    - ▶ Binary cross entropy:  $-\sum (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$
    - ▶ Categorical cross entropy:  $-\sum y_i (\log(\hat{y}_i))$
    - ▶ Interpret the result as a conditional probability:  $\text{Log}(P(X=c|y))$
  - ▶ Metrics: accuracy, recall, precision, F1, etc.

# Epoch vs Batch Size vs Iterations

## ▶ Epochs

- ▶ One Epoch is when an **ENTIRE** dataset is passed forward and backward through the neural network only ONCE.

## ▶ Batch Size

- ▶ Total number of training examples present in a single batch.

## ▶ Iterations

- ▶ Iterations is the number of batches needed to complete one epoch.

- ▶ 
$$Iterations = \frac{nb\_items}{batch\_size}$$

## ▶ There are three types of Gradient Descent:

### ▶ Batch Gradient Descent

- ▶ batch size = number of items
- ▶ 1 iteration by epoch

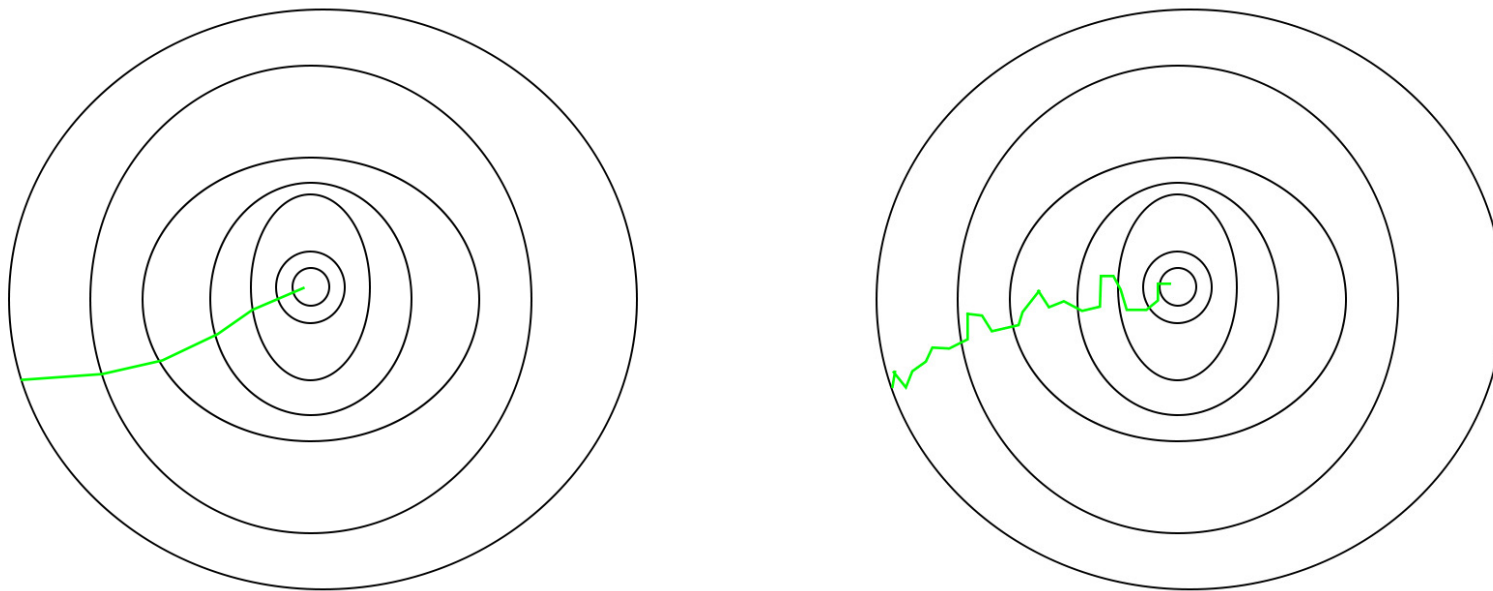
### ▶ Stochastic Gradient Descent

- ▶ Batch size = 1
- ▶ Nb\_items iterations by epoch

### ▶ Mini-batch Gradient Descent

- ▶  $1 < \text{batch size} \ll \text{number of items}$

# Impact of the batch size



in which figure:

- Batch size == 1 ?
- Batch size == nb of items ?



# Batch size selection

- ▶ SGD (batch size=1 or  $\ll N$ ) is generally noisier than a Gradient Descent per epoch
  - ▶ more iterations to reach the minima, due to the random nature of the descent
  - ▶ but it is generally faster because it requires fewer epochs.
- ▶ Generally use batch size around
  - ▶ 32, 64, 128
  - ▶ it depends also on the use case and the system memory,
  - ▶ i.e., we should ensure that a single mini-batch should be able to fit in the system memory.

# Gradient descent algorithm

```
for  $i$  in range(Number of training steps) :  
     $batches = \text{miniBatchGenerator}(X, Y, \text{batch\_size})$   
    for  $j$  in range(Number of batches) :  
         $\text{minibatch}X, \text{minibatch}Y = batches[j]$   
        Forward Propagation using  $\text{minibatch}X$  to calculate  $y'$   
        Calculate cost using  $\text{minibatch}Y$   
        Backward Propagation to calculate derivative  $dW$  and  $dB$   
        Parameter Updation using following rule:  
             $W = W - \alpha.dW$   
             $b = b - \alpha.db$ 
```

# How to choose learning rate

- ▶ This is probably one of the most important hyper-parameter
  - ▶ Set the learning rate too small and your model might take ages to converge
  - ▶ Make it too large and within initial few training examples, your loss might shoot up to sky
- ▶ Generally
  - ▶ Use large learning rate at the beginning
  - ▶ Use small learning rate when you reach the minima
- ▶ But, today, a lot of optimizer use adaptative learning rate
  - ▶ Adam use an adaptative learning rate



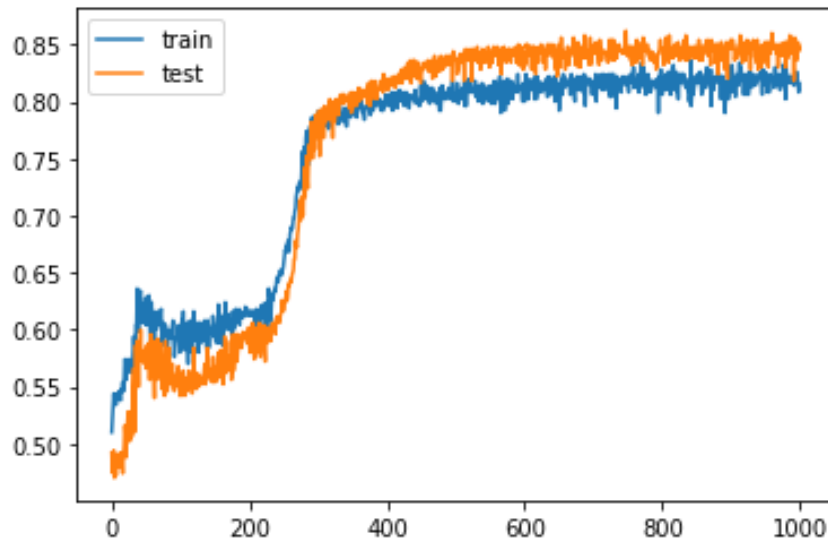
## ***The vanishing gradient problem***

The diagram illustrates the architecture of the proposed deep learning model. It starts with an 'Input' block, which feeds into a 'LocalAttentionForm' module (green). This is followed by a series of convolutional (blue) and pooling (red) layers. The architecture is divided into four main stages, each containing a 'LocalAttentionForm' module and a series of convolutional and pooling layers. The final output is produced by a 'Softmax Activation' module (yellow), which is connected to a 'Softmax1' module. The diagram uses color-coded boxes: blue for convolutional layers, red for pooling layers, green for LocalAttentionForm modules, and yellow for Softmax Activation modules. The input is labeled 'Input' and the output is labeled 'Softmax1'.

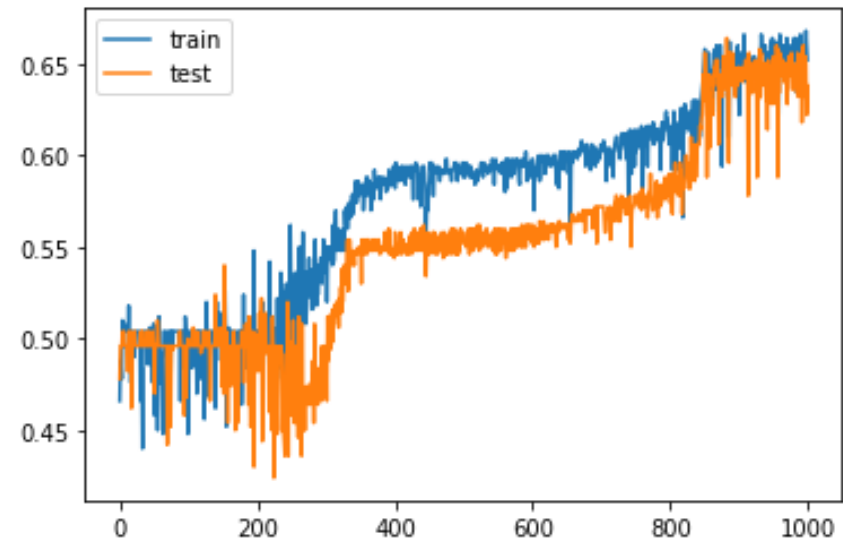
# The problem

- ▶ When the gradient becomes negligible, subtracting it from original matrix doesn't make any sense and hence the model stops learning.

**1 hidden layer**  
**tanh, random uniform (-0.5, +0.5)**

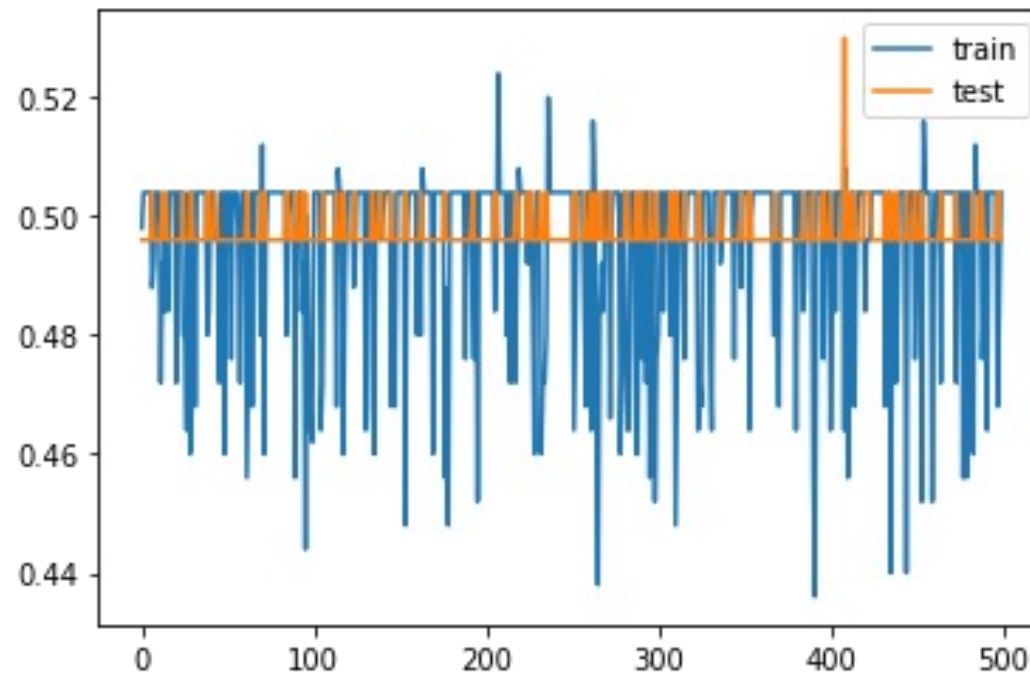


**10 hidden layers**  
**tanh, random uniform (-0.5, +0.5)**



# The problem

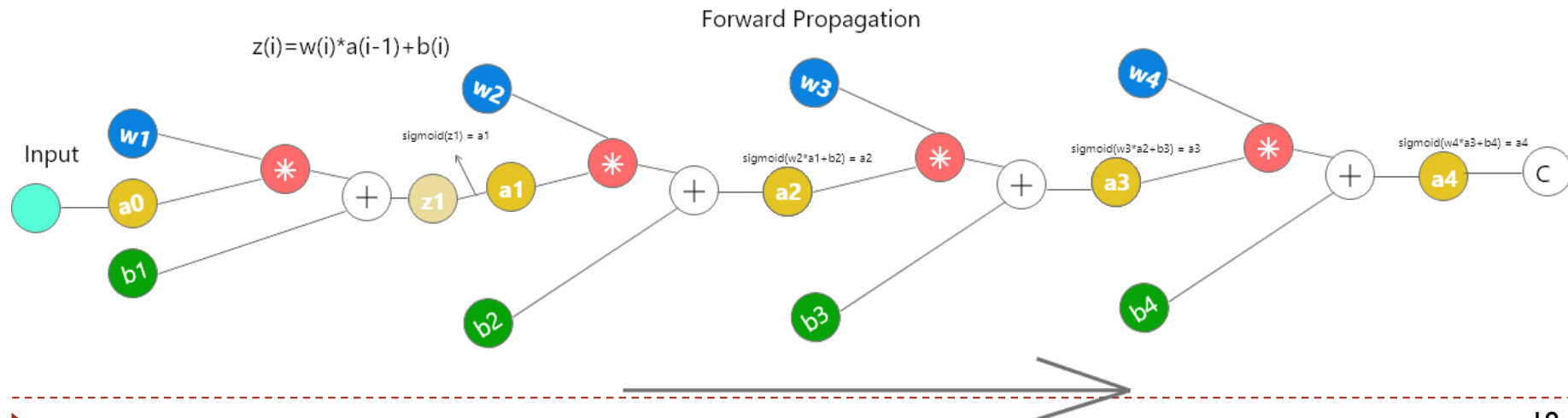
**But for 20 hidden layers  
tanh, random uniform (-0.5, +0.5)**



# Forward (4 hidden layers)

- ▶ Consider a neural network with 4 hidden layers with a single neuron in each matrix.

- ▶  $a1 = \sigma(z1) = \sigma(w1 * a0 + b1)$
- ▶  $a2 = \sigma(z2) = \sigma(w2 * a1 + b2) = \sigma(w2 * \sigma(z1) + b2)$   
 $= \sigma(w2 * \sigma(w1 * a0 + b1) + b2)$
- ▶  $a3 = \sigma(z3) = \sigma(w3 * a2 + b3) = \sigma(w3 * \sigma(z2) + b3)$   
 $= \sigma(w3 * \sigma(w2 * \sigma(w1 * a0 + b1) + b2) + b3)$
- ▶  $C = a4 = \sigma(z4) = \sigma(w4 * a3 + b4) = \sigma(w4 * \sigma(z3) + b4)$   
 $= \sigma(w4 * \sigma(w3 * \sigma(w2 * \sigma(w1 * a0 + b1) + b2) + b3) + b4)$

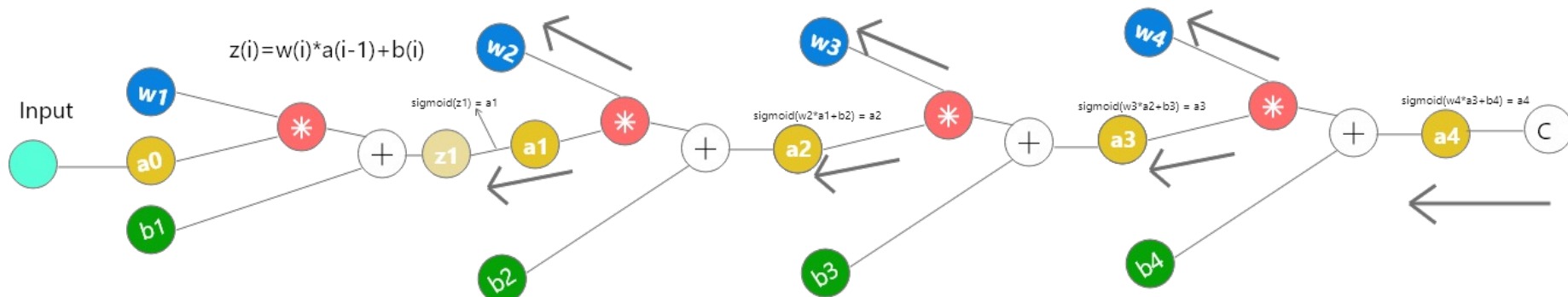




# Backpropagation

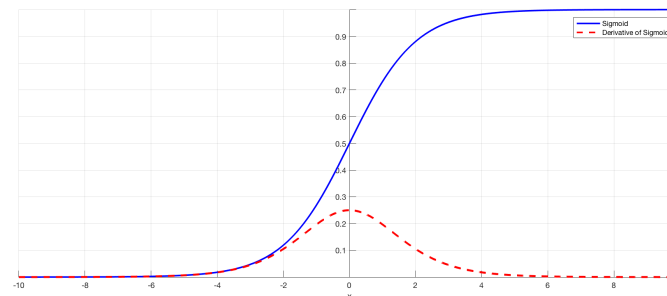
- ▶ During backpropagation, we calculate the derivative of the output with respect to the different weights
- ▶ Suppose that we want to modify the matrix of weight  $w_1$ 
  - ▶  $C = \sigma(w_4 * \sigma(w_3 * \sigma(w_2 * \sigma(w_1 * a_0 + b_1) + b_2) + b_2) + b_1)$

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



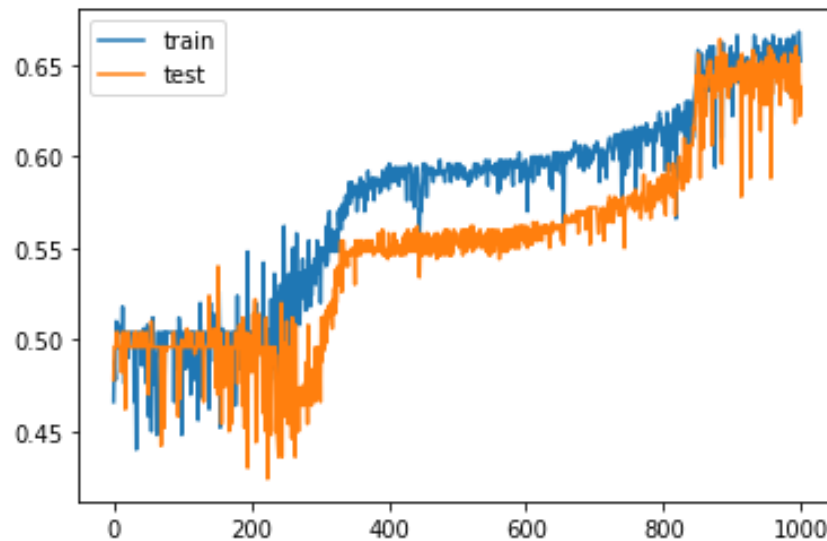
# Backpropagation

- ▶ Each sigmoid'(z1),sigmoid'(z2).. Etc. are less than 1/4
  - ▶ Because derivative of sigmoid function is less than 1/4
- ▶ Each weight matrices w1,w2,w3,w4 are generally initialized using gaussian method to have a mean of 0 and standard deviation of 1.
  - ▶  $\|w_i\| \leq 1$
- ▶ Each term of  $\frac{\partial C}{\partial w_1}$  are  $\leq 1$  or  $\leq 1/4$
- ▶ Multiplying such small terms for a huge number of times we get very small gradient which makes the model to almost stop learning.
  - ▶  $(1/4)^2 = 0,0625$
  - ▶  $(1/4)^4 = 0,00390625$
  - ▶  $(1/4)^8 = 0,00001526$

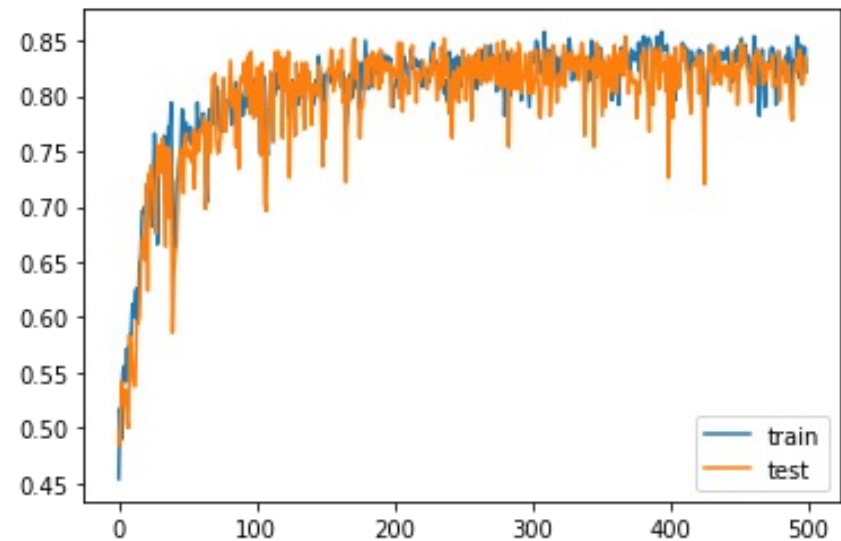


# How do you fix « the vanishing gradient problem »

- ▶ First step: replace 'sigmoid' or 'tanh' by 'relu'
  - ▶ With relu use he\_uniform as initializer



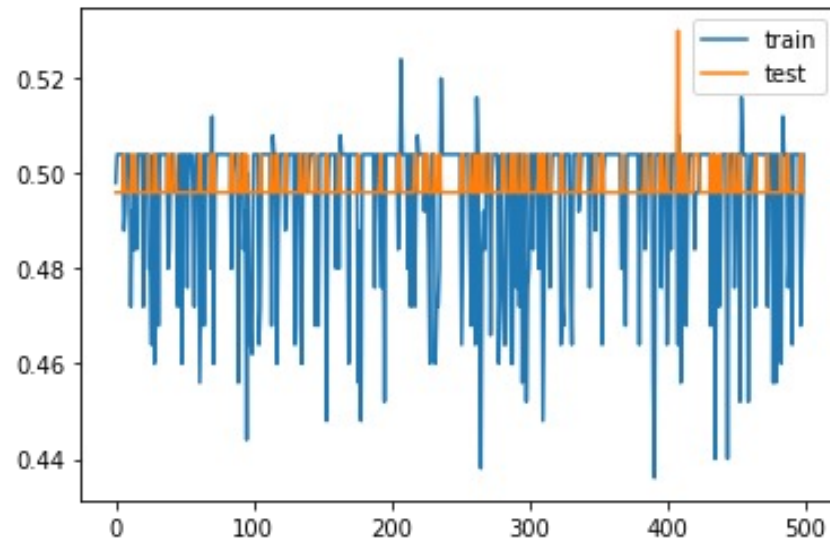
**10 hidden layers  
tanh, random uniform (-0.5, +0.5)**



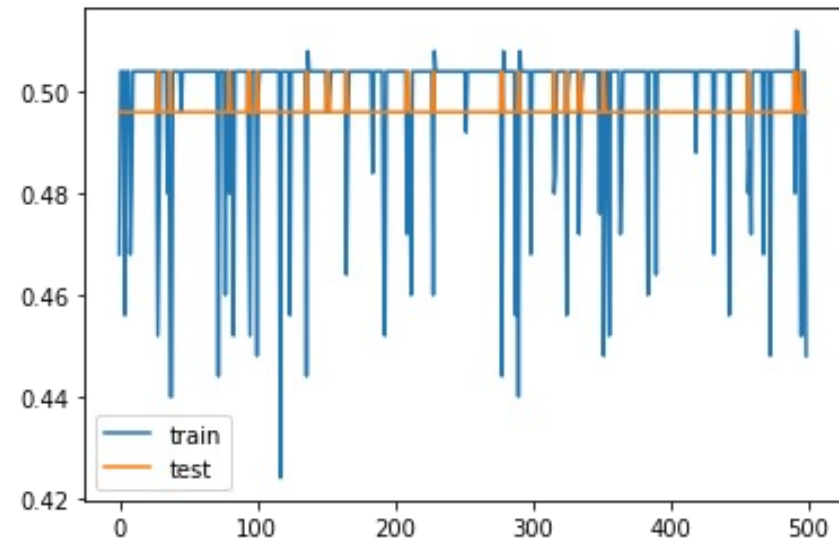
**10 hidden layers  
relu, he\_uniform**

# How do you fix « the vanishing gradient problem »

- ▶ First step: replace 'sigmoid' by 'relu'
  - ▶ With relu use he\_uniform as initializer



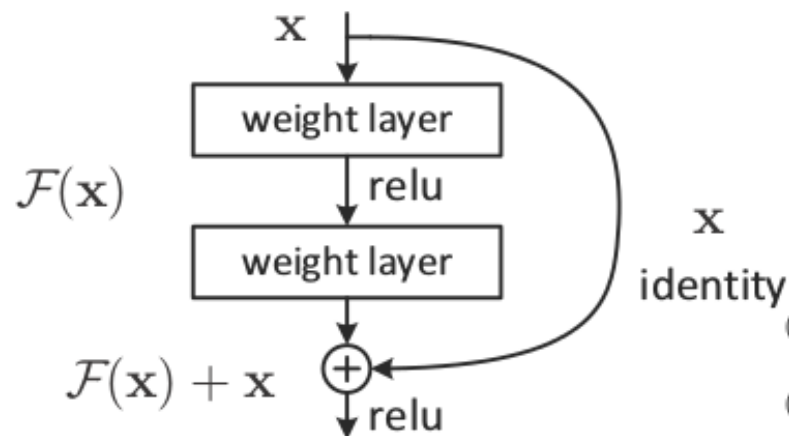
**But for 20 hidden layers  
tanh, random uniform (-0.5, +0.5)**



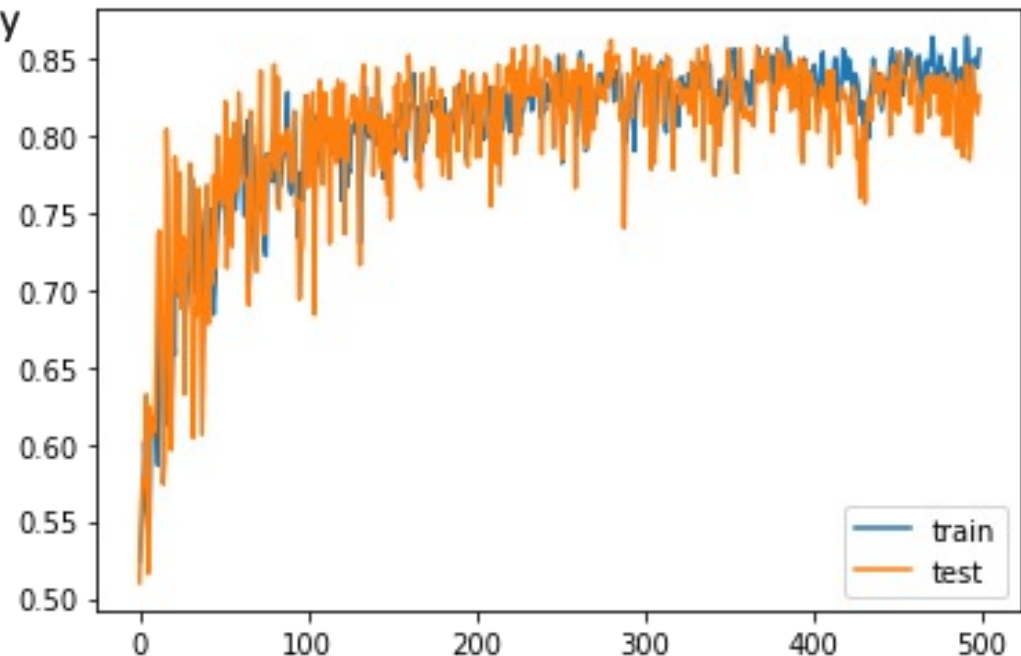
**20 hidden layers  
relu, he\_uniform**

# How do you fix « the vanishing gradient problem »

- Improve solution: use residual

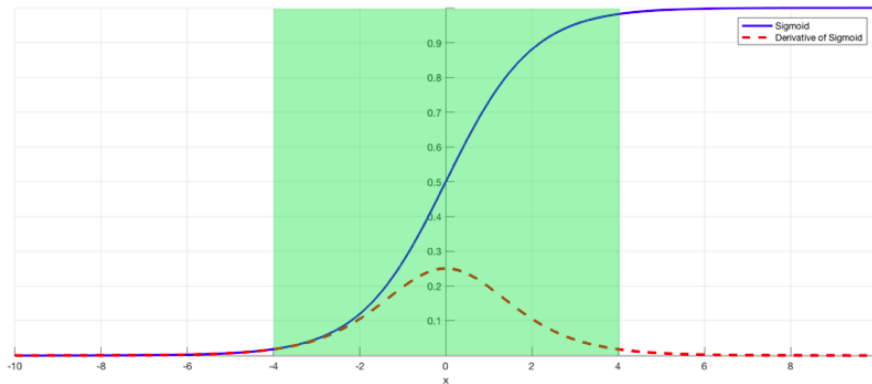


**Residual, 20 hidden layers  
relu, random\_uniform (-0,5, +0,5)**

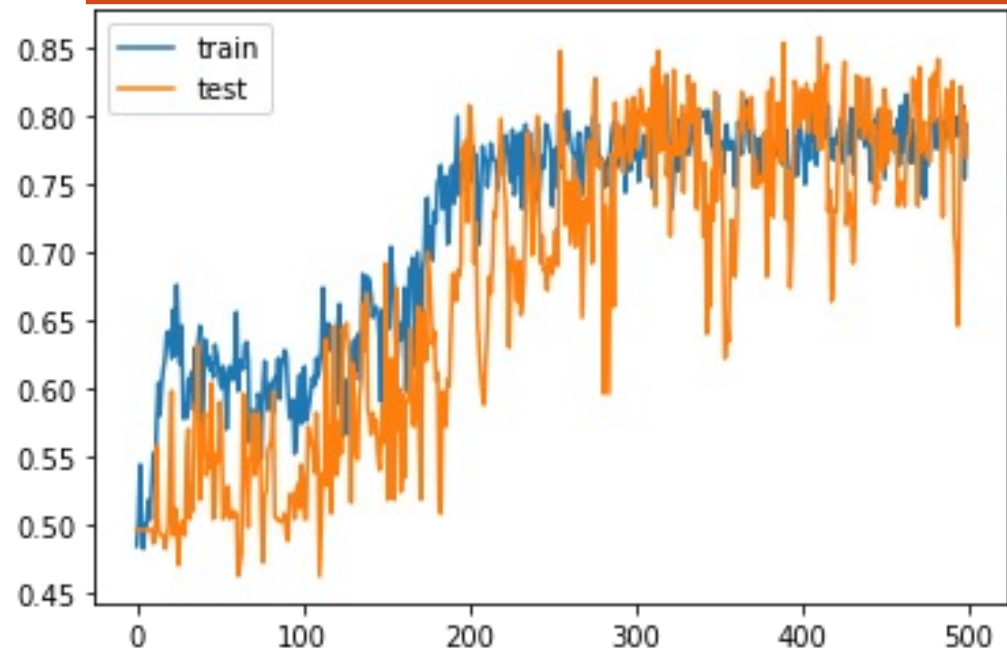


# How do you fix « the vanishing gradient problem »

- Improve solution: use batch normalisation

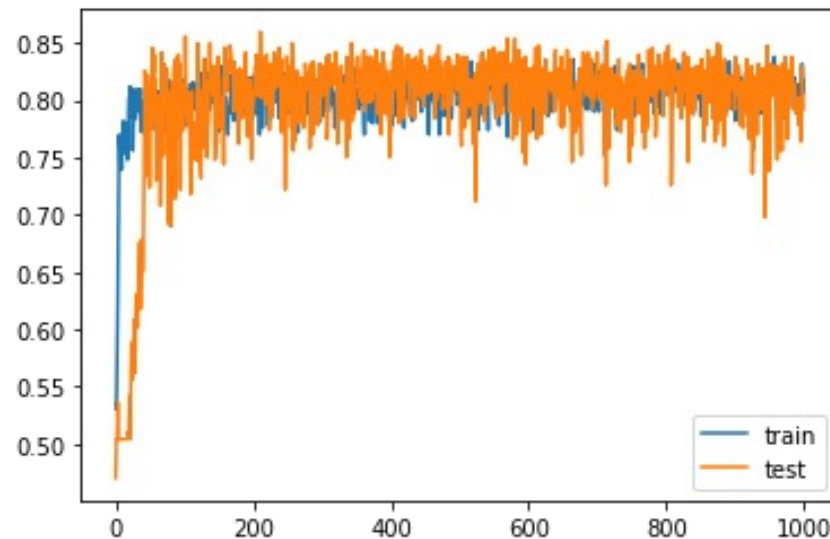


**Batch normalization, 20 hidden layers  
tanh, random\_uniform (-0,5, +0,5)**

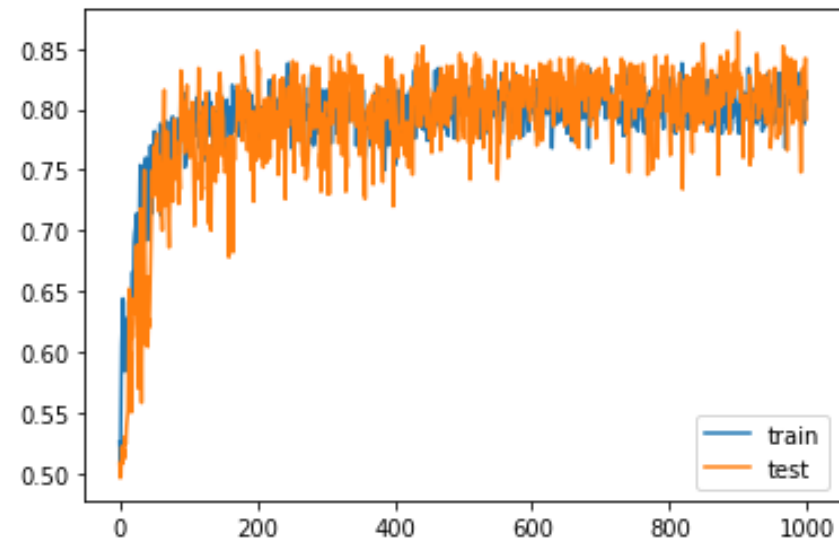


# How do you fix « the vanishing gradient problem »

- ▶ You can combine the solution : Residual + Batch normalization



**20 hidden layers**  
**Residual + Batch normalization**  
**tanh, random uniform (-0.5, +0.5)**



**20 hidden layers**  
**Residual + Batch normalization**  
**relu, he\_uniform**

# What have we learned

- ▶ For not deep [1, 3]
  - ▶ Tanh/sigmoid with random uniform (-0,5, +0,5)
- ▶ For medium deep ]3, 8]
  - ▶ Relu with he\_uniform
- ▶ For deep ]8, 25]
  - ▶ Residual + batch normalization

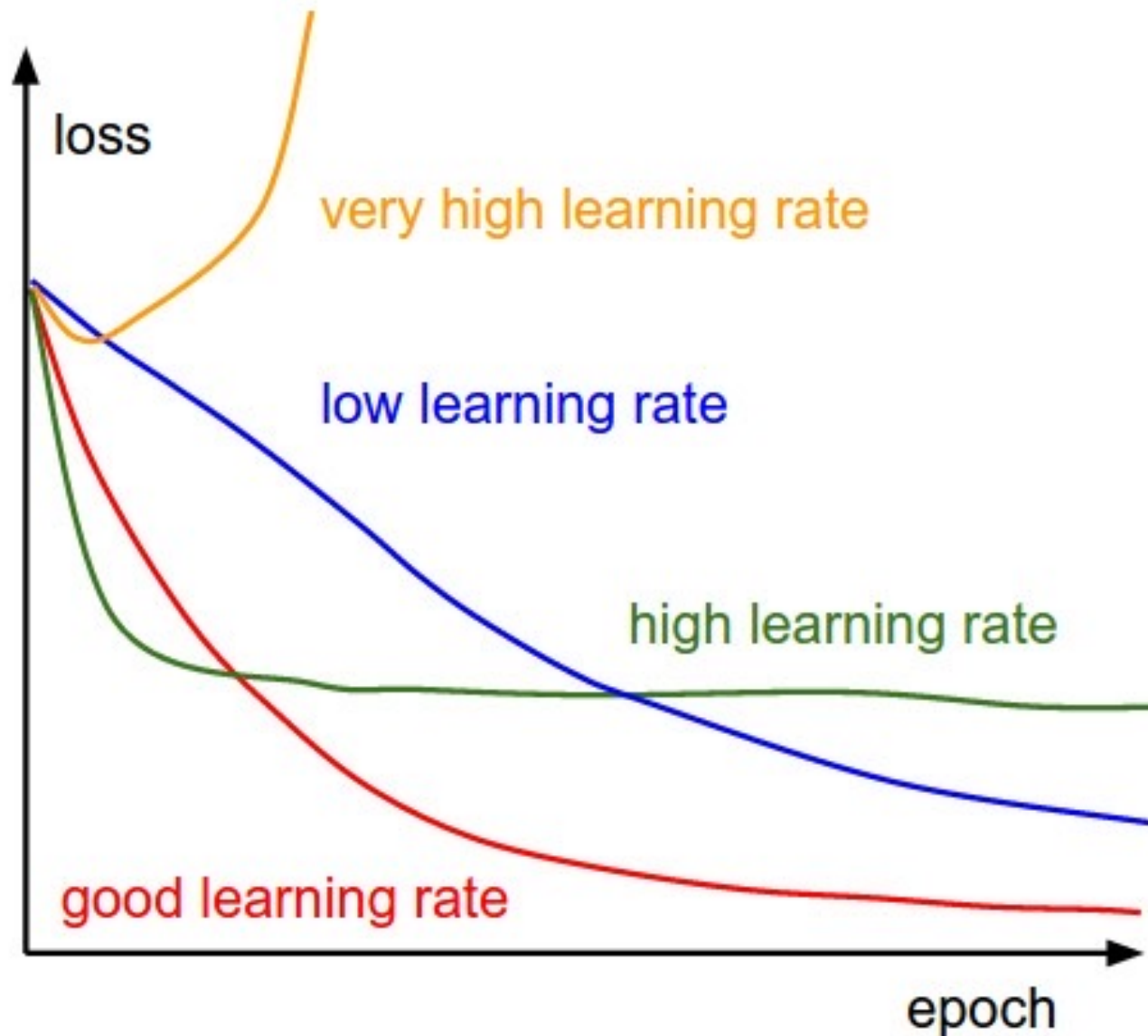




## ***How to avoid overfitting***

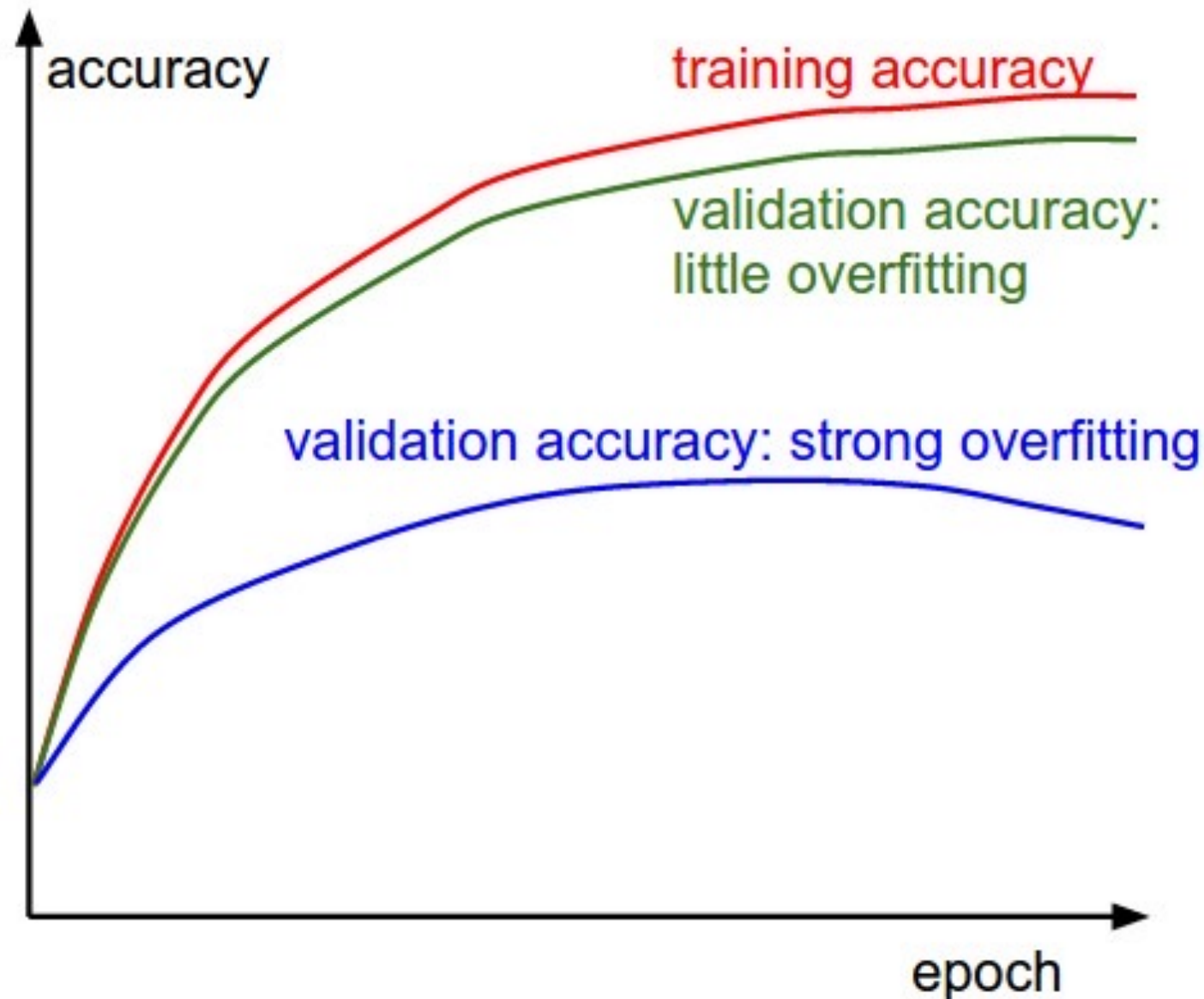
# Babysit your network!

---

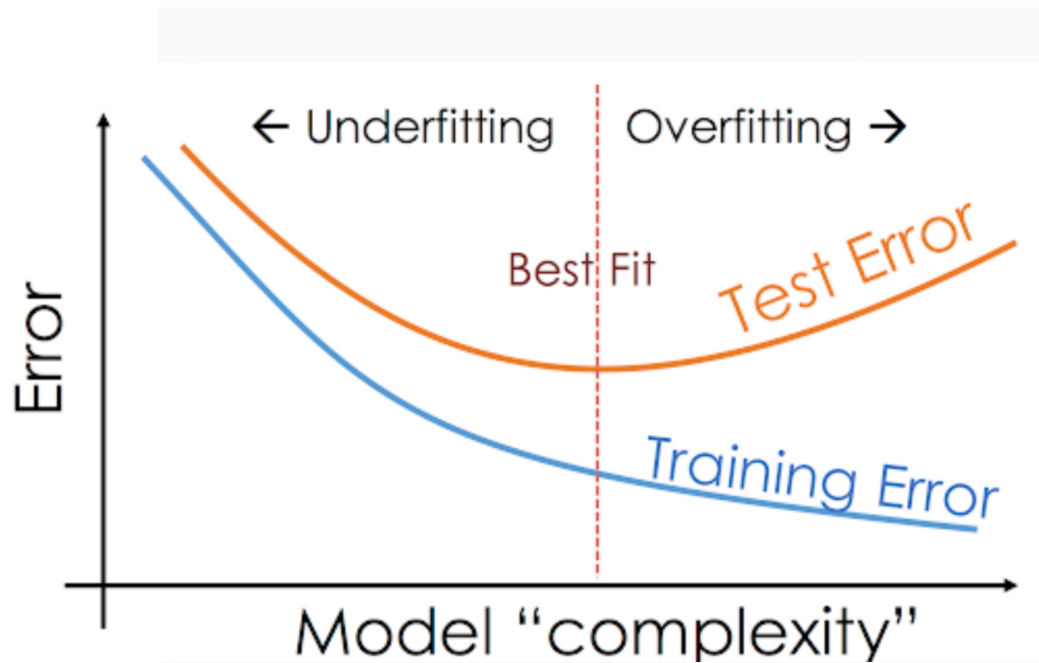


# Babysit your network!

---



# Babysit your network!



babysitting your deep network

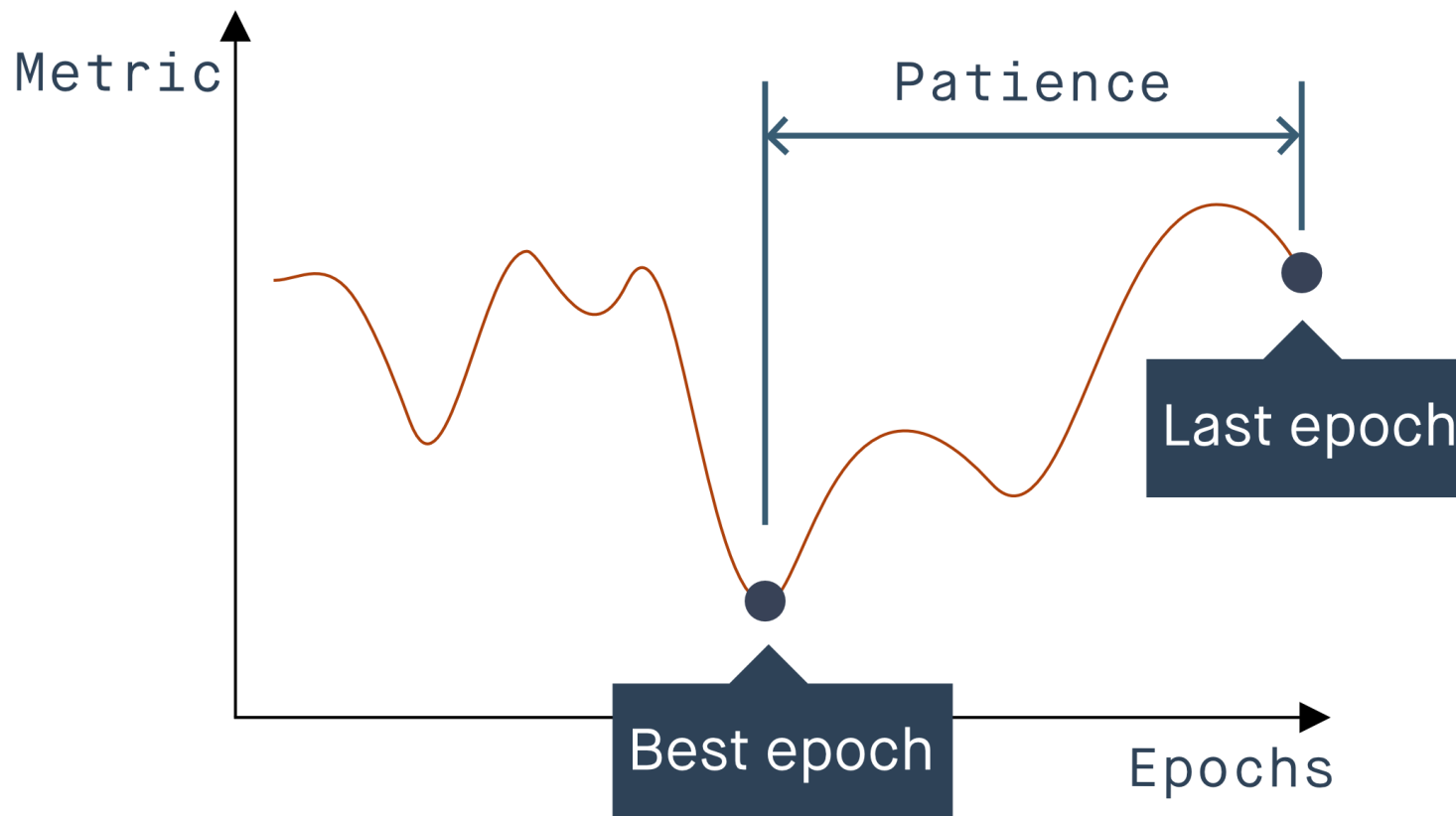
- 1 overfitting and underfitting
- 2 check accuracy before training [Saxe et al., 2011]
- 3 Y. Bengio : “check if the model is powerful enough to overfit, if not then change model structure or make model larger”

# How to stop at best fit

---

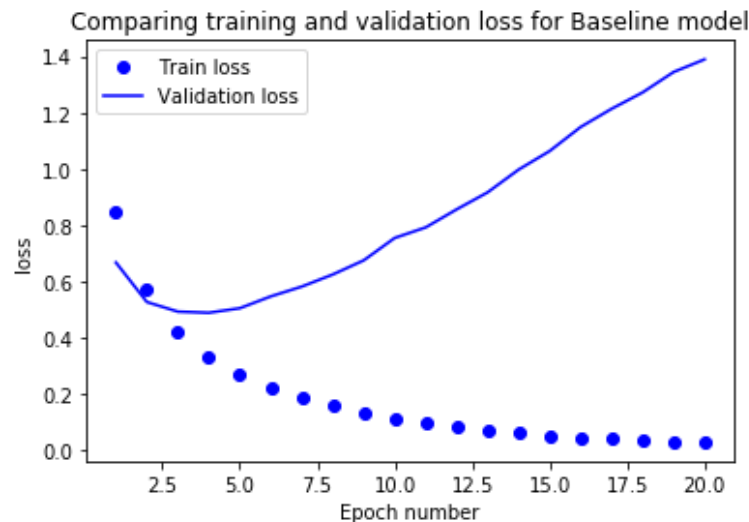
- ▶ Use earlystopping

- ▶ `es = tf.keras.callbacks.EarlyStopping(monitor='val_acc', patience=3)`
- ▶ `history = model.fit(X_train, y_train, validation_split=0.2, epochs=10000, callbacks=[es])`



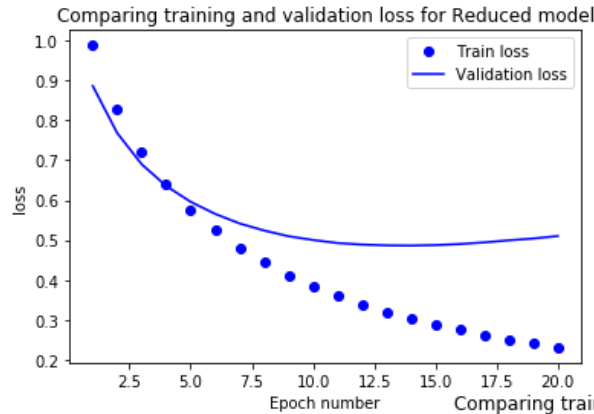
# Overfitting in neural networks

- ▶ Neural Networks are especially prone to overfitting
- ▶ Zero error is possible, but so is more extreme overfitting

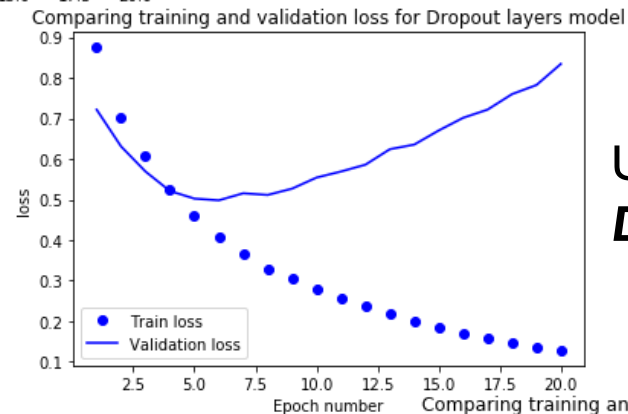


- ▶ **Handling overfitting**
  - ▶ **Reduce the network's capacity** by removing layers or reducing the number of elements in the hidden layers
  - ▶ Apply **regularization**, which comes down to adding a cost to the loss function for large weights
  - ▶ Use **Dropout layers** (add noise to data), which will randomly remove certain features by setting them to zero

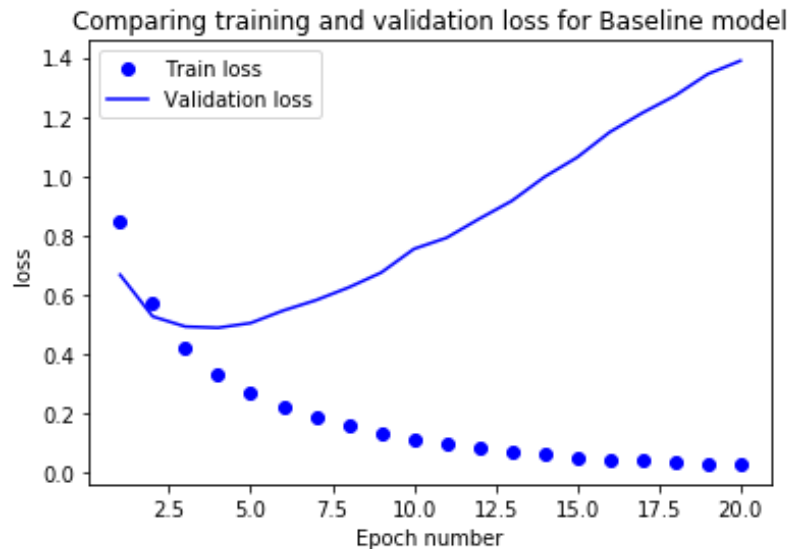
# Overfitting in neural networks



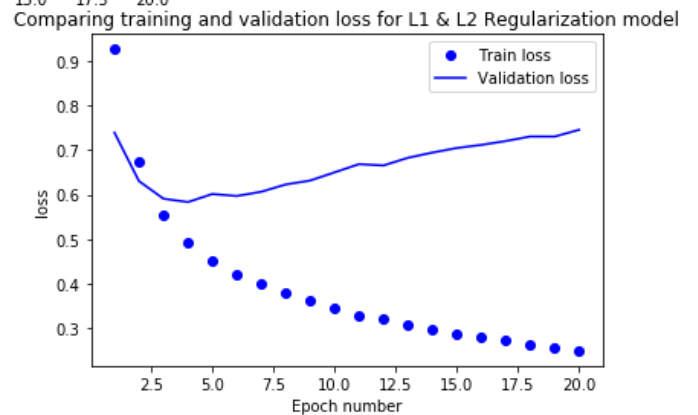
*Reduce  
network's capacity*



*Use  
Dropout layers*



*Apply  
regularization*



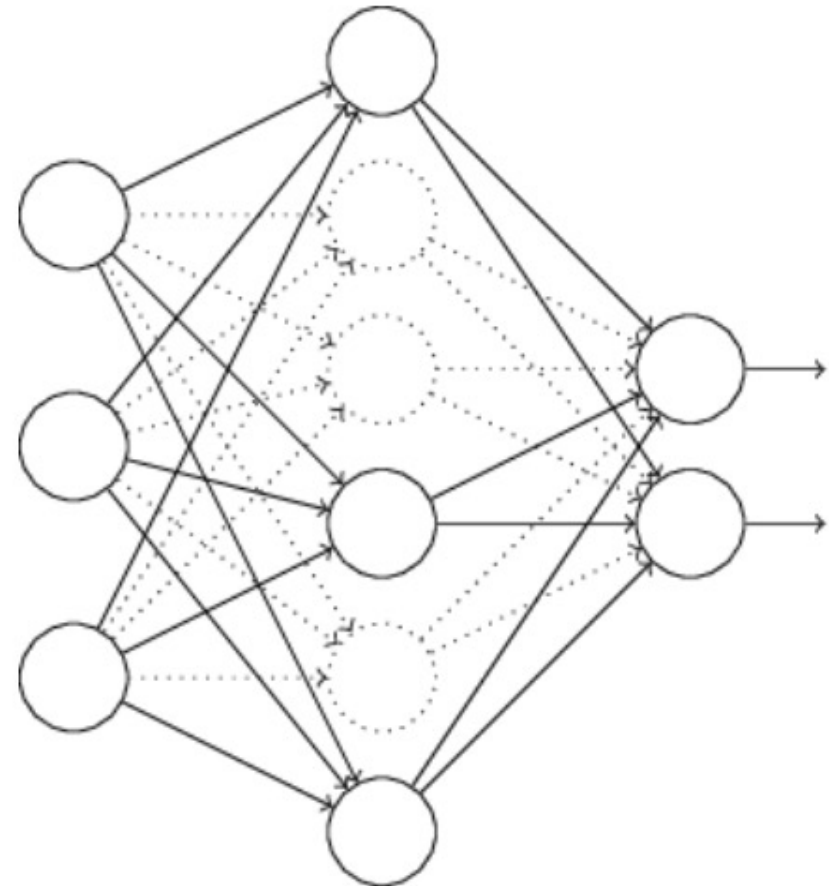
# Add dropout

- ▶ The networks are usually initialized with random weights.
  - ▶ At each training → the result is slightly different.
- ▶ Why random weights?
  - ▶ In order to have different weights
  - ▶ If the weights are all equal, the response of the filters will be equivalent.
    - ▶ The network does not train
- ▶ Why not train 5 different networks with random starts and vote on their outcome?
  - ▶ That's pretty much what dropout does
  - ▶ This works well!
  - ▶ It helps with generalization because the errors are averaged.



# Add dropout

- ▶ At each mini-batch:
  - Randomly select a subset of neurons.
  - Ignore them.
- ▶ On test: part of weights outgoing to compensate for training on part of neurons.
- ▶ Effect:
  - Neurons become less dependent on output of connected neurons.
  - Forces network to learn more robust features that are useful to more subsets of neurons.
  - Like averaging over many different trained networks with different random initializations.
  - Except cheaper to train.



# Add regularisation

---

- ▶ Regularization, in general, is a mechanism that penalizes model complexity
  - ▶ Add a term to the loss function that represents model complexity.
- ▶ L2 regularization
  - ▶ Penalize the large weights: indicate over fitting to the training data.
  - ▶  $J_{new}(X, y) = J(X, y) + \lambda \sum \sum w_{ij}^2$

# Add momentum

---

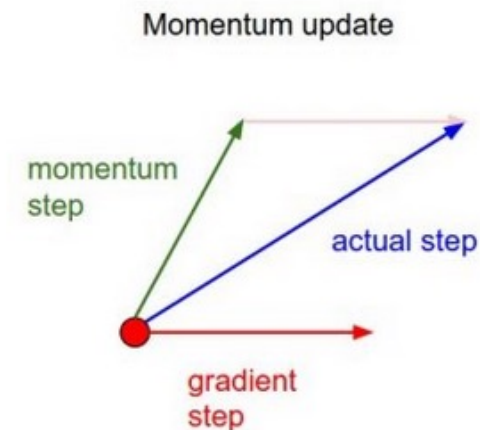
- ▶ Momentum regularization

- ▶ Adds a fraction of the past weight update to the current weight update
- ▶ Prevent the model from getting stuck in local minima, even if the current gradient is 0

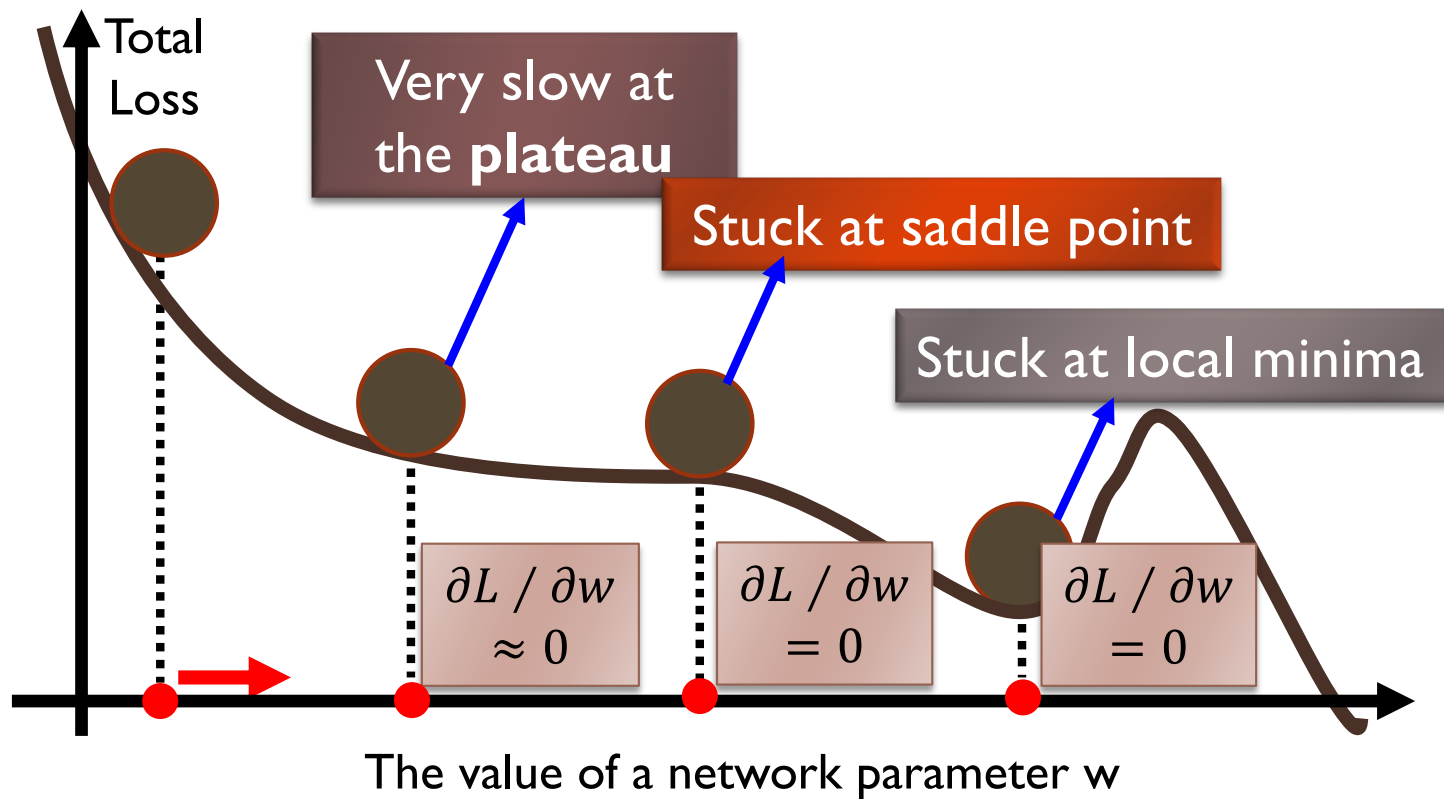
- ▶  $W_{t+1} = W_t - \beta \nabla W - \alpha \nabla J$

Momentum  
(past weight update)

Gradient cost function



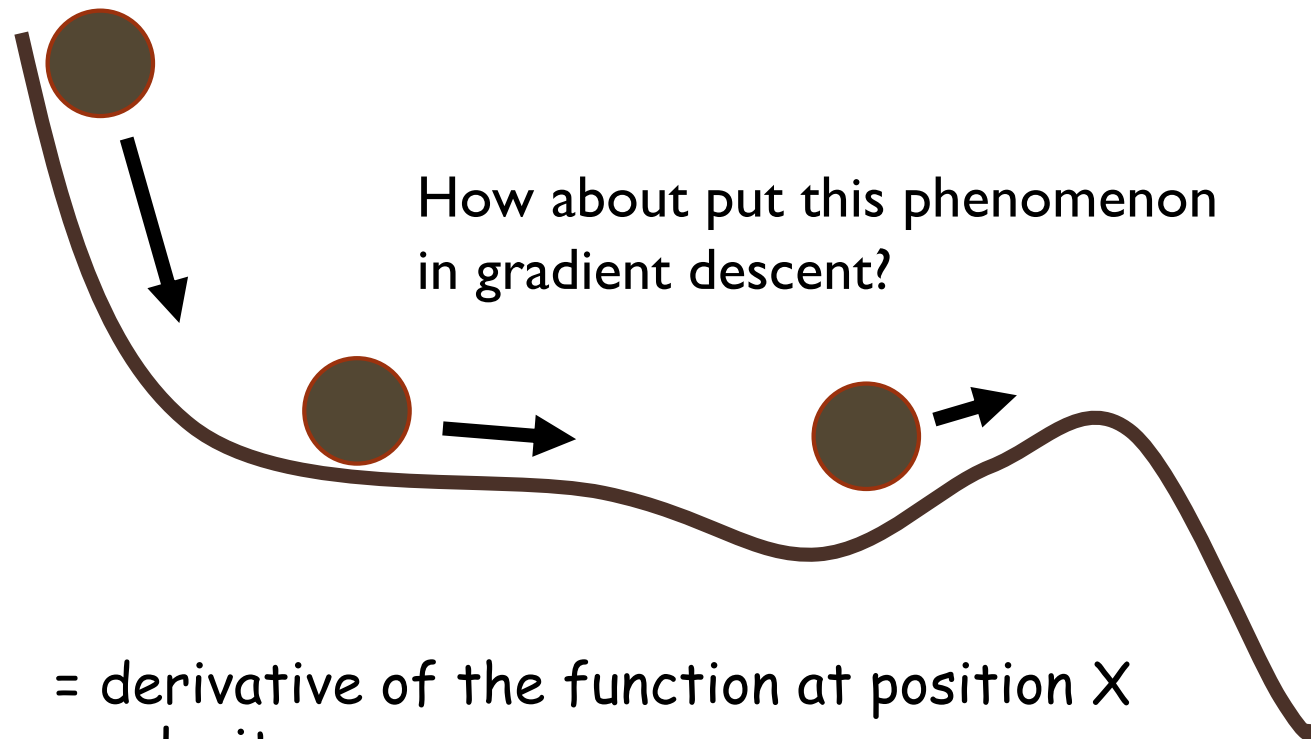
# Why momentum ?



# In physical world .....

---

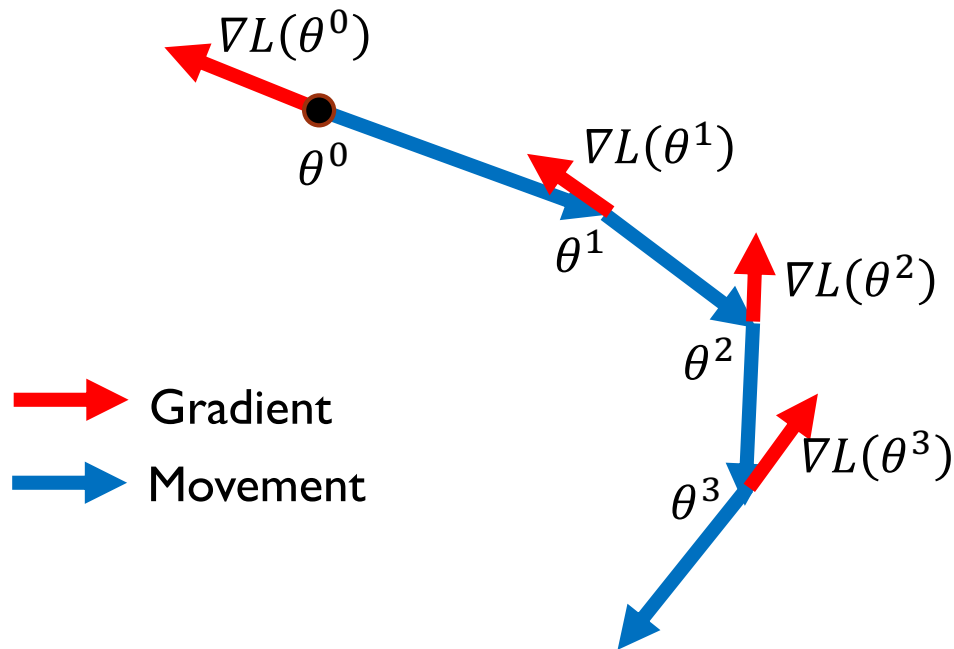
- ▶ **Momentum = “a little flick”**



- Gradient = derivative of the function at position  $X$
- = velocity
- If derived a second time = acceleration
- Momentum  $\approx$  adding information about the acceleration

# Vanilla Gradient Descent

---



Start at position  $\theta^0$

Compute gradient at  $\theta^0$

Move to  $\theta^1 = \theta^0 - \eta \nabla L(\theta^0)$

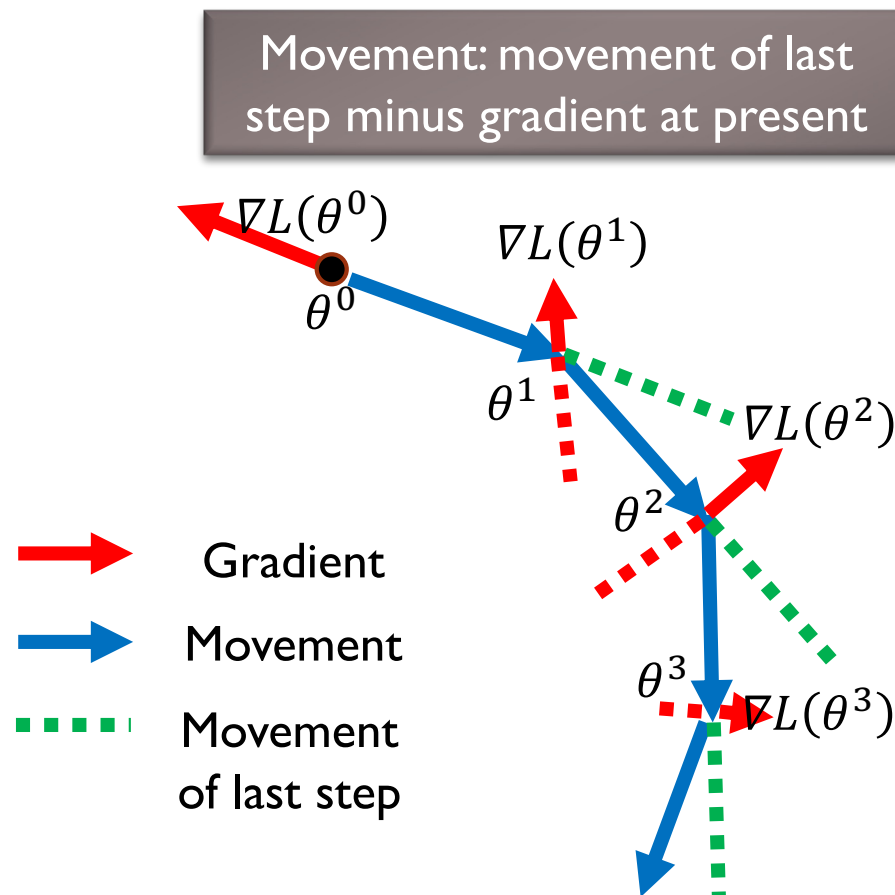
Compute gradient at  $\theta^1$

Move to  $\theta^2 = \theta^1 - \eta \nabla L(\theta^1)$

$\vdots$

Stop until  $\nabla L(\theta^t) \approx 0$

# Momentum = “a little flick”



Start at point  $\theta^0$

Movement  $v^0=0$

Compute gradient at  $\theta^0$

Movement  $v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$

Move to  $\theta^1 = \theta^0 + v^1$

Compute gradient at  $\theta^1$

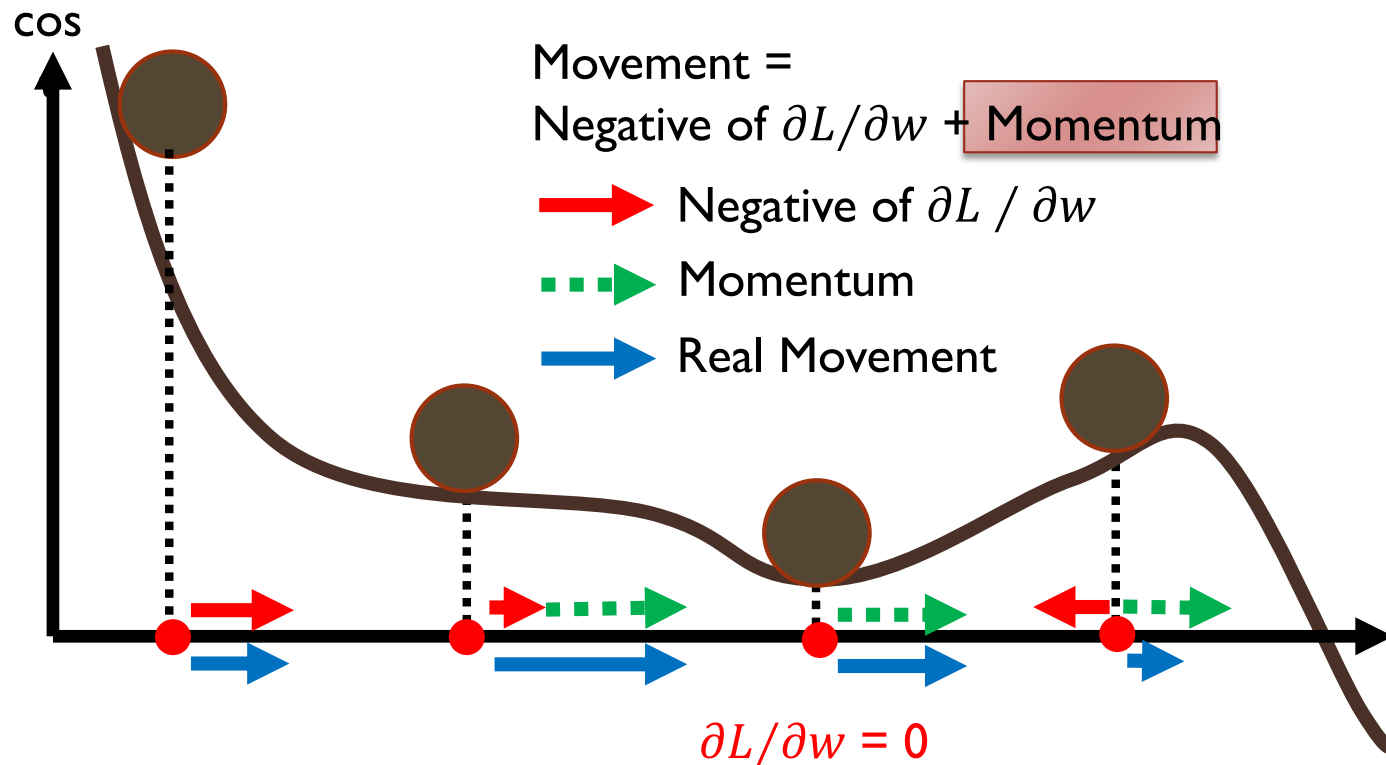
Movement  $v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$

Move to  $\theta^2 = \theta^1 + v^2$

Movement not just based on gradient, but previous movement.

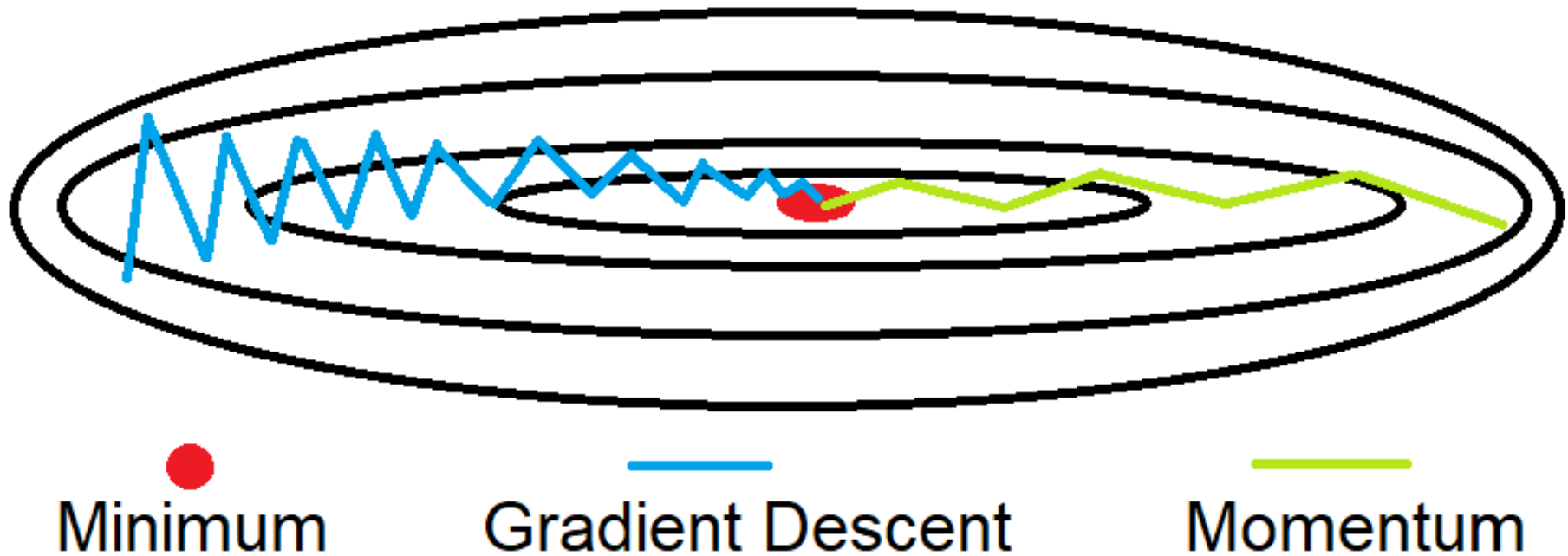
# Momentum = “a little flick”

Still not guarantee reaching global minima, but give some hope .....





# Add momentum



# How to add regularisation in Keras

## ▶ Dropout

- ▶ Add dropout layers
- ▶ Parameter = probability of training a given node
  - ▶ 1.0 means no dropout,
  - ▶ 0.0 means no outputs
- ▶ A good value for dropout in a hidden layer is between 0.2 and 0.5

## ▶ L1+L2 regularization

- ▶ Add regularizer on layer
  - ▶ `layer = layers.Dense(units=5, kernel_regularizer=regularizers.L1(0.01), activity_regularizer=regularizers.L2(0.01))`
  - ▶ `kernel_regularizer`: apply regularizer on weights
  - ▶ `activity_regularizer`: apply regularizer on output
- ▶ Available regularizer
  - ▶ `tf.keras.regularizers.L1()` or `tf.keras.regularizers.L2()`

## ▶ Momentum

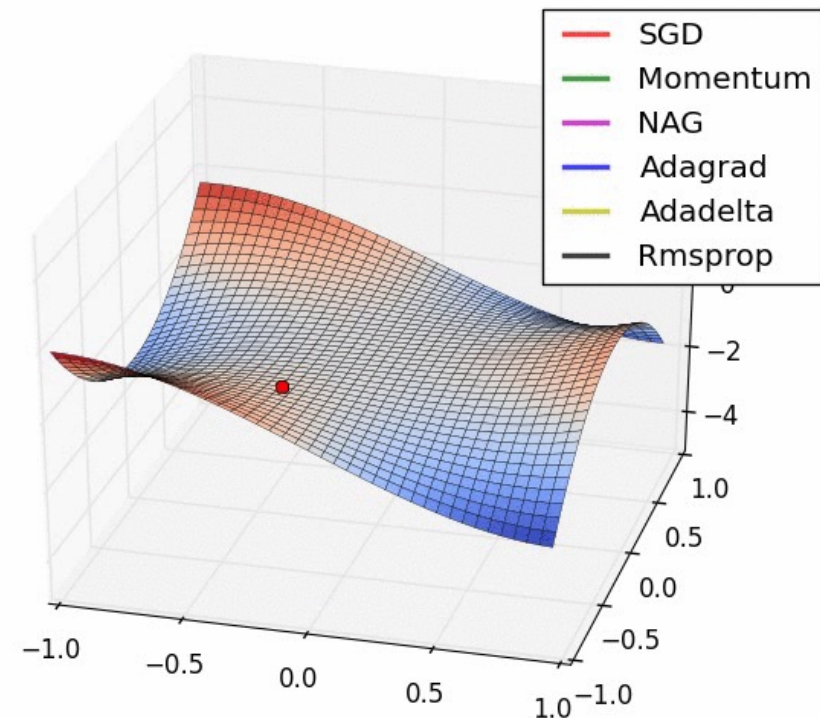
- ▶ Add momentum to the optimizer



# ***How to choose Optimizer***

# Choose optimizer algorithms

- ▶ It is crucial to train deep learning model trains in a shorter time without penalizing the accuracy
  - ▶ Several optimizers have been proposed
  - ▶ They differ in the way the weights are modified
- ▶ Some optimizer
  - ▶ SGD (Stochastic Gradient Descent)
  - ▶ RMSProp
  - ▶ SGD with Momentum
  - ▶ Nesterow Adaptive Gradient
  - ▶ Adam





## ***Hyper-parameters tuning***

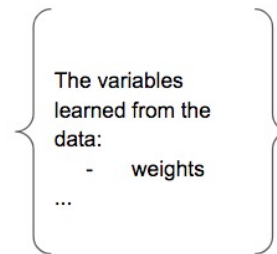
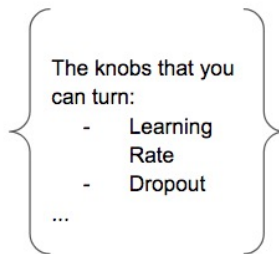
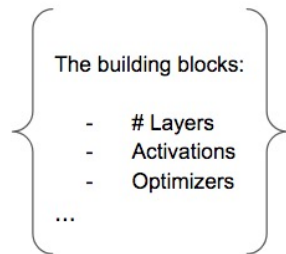
# Why ? When ?

---

- ▶ Neural architectures have a lot of hyper-parameters
  - ▶ Nb layers, nb neurons, dropout rate



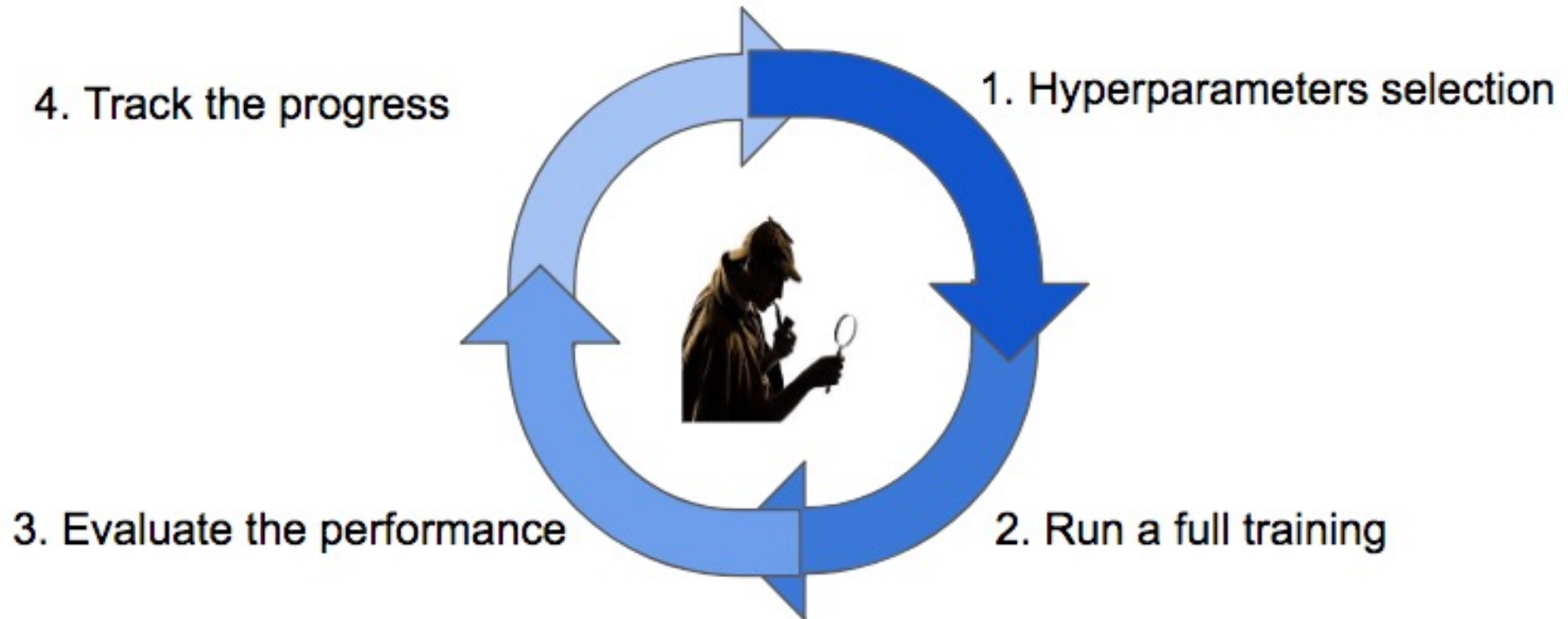
(Model Design + Hyperparameters) → Model Parameters



- ▶ Find the best configuration that gives the best score on the validation/test set
  - ▶ Must take into account available resources: 💻, 💰 and ⌚

# *It's an iterative process*

---

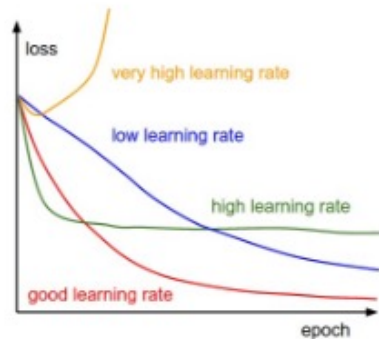


## **4 main strategies**

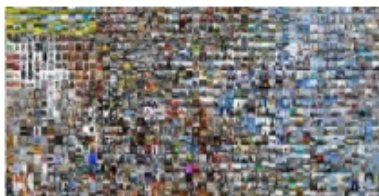
- Babysitting (aka Trial & Error)
- Grid Search
- Random Search
- Bayesian Optimization

# Babysitting

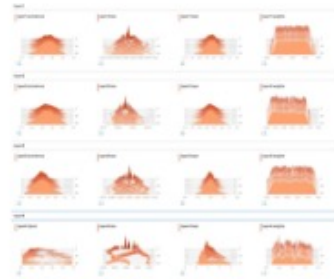
<https://www.coursera.org/lecture/deep-neural-network/hyperparameters-tuning-in-practice-pandas-vs-caviar-DHNcc>



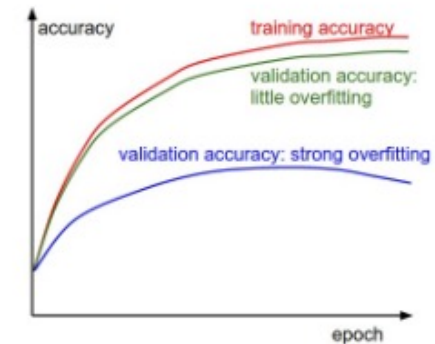
Loss



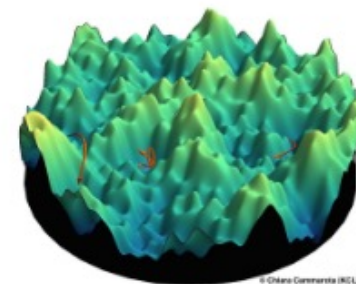
Data



Act/Grad/Filter



Metric

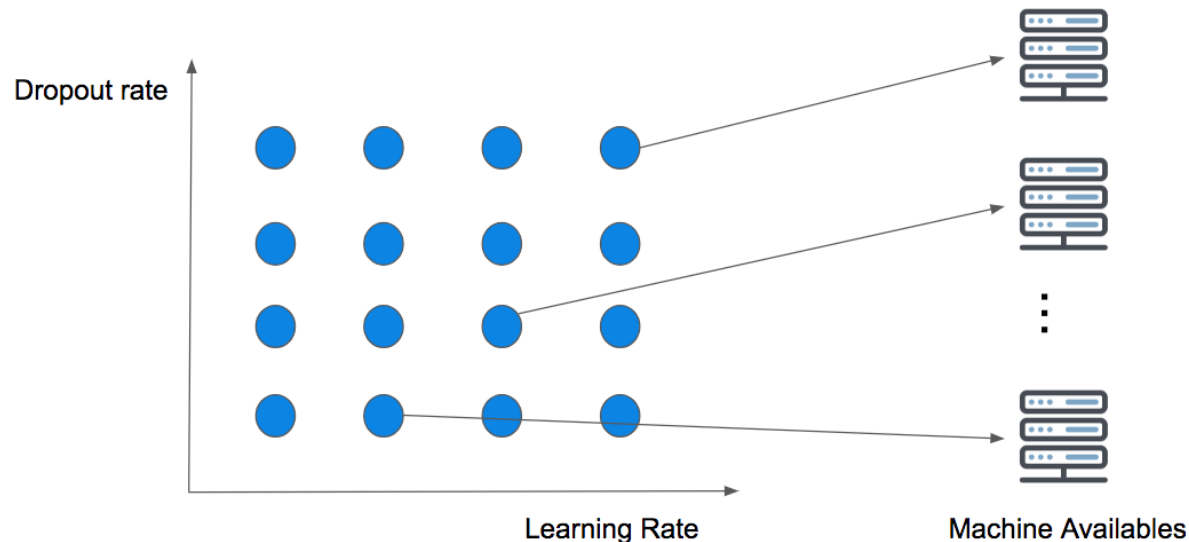


Space



# Grid search $\approx$ Just try everything!


---



- Define a grid on  $n$  dimensions
  - Each of these maps for an hyperparameter.
  - e.g.  $n = (\text{learning\_rate}, \text{dropout\_rate}, \text{batch\_size})$
- For each dimension, define the range of possible values:
  - e.g.  $\text{batch\_size} = [4, 8, 16, 32, 64, 128, 256]$
- Search for all the possible configurations and wait for the results to establish the best one:
  - e.g.  $C1 = (0.1, 0.3, 4) \rightarrow \text{acc} = 92\%$ ,  $C2 = (0.1, 0.35, 4) \rightarrow \text{acc} = 92.3\%$ , etc...

# Grid search $\approx$ Just try everything!

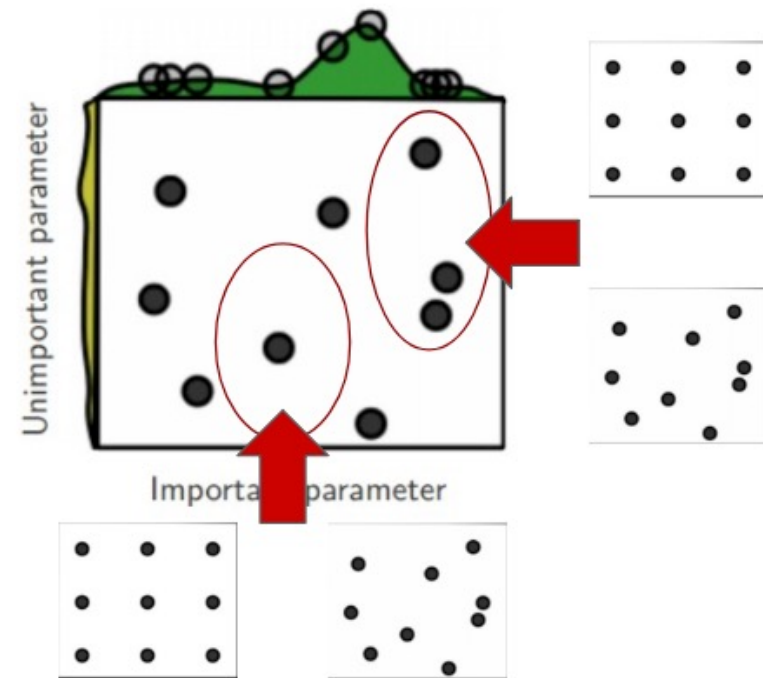
---

- ▶ Exhaustive: considers all parameter combinations
- ▶ The more computational resources  you have
  - ▶ the more you can parallelize the calculations and save time!
- ▶ Doesn't take into account the history of computations (we'll elaborate on that soon).
- ▶ The more dimensions you add, the more the search time explodes
  - ▶ Usually by an exponential factor,
  - ▶ Unfeasible strategy if too many hyper-parameters!
- ▶ Usable if dimensions are  $\leq 4$
- ▶ Guarantees to find the best configuration

# Random search $\approx$ Just try some candidates!

---

- ▶ Implements a random search on the parameters
- ▶ For each parameter, we give:
  - ▶ A list of discrete choices (which will be sampled uniformly)
  - ▶ A distribution over the possible values,
- ▶ Random Search has two advantages:
  - ▶ The number of combinations to be tested is chosen (trial number)
  - ▶ Adding parameters that do not influence performance does not decrease efficiency
    - ▶ Taking into account the previous calculations



# Bayesian Optimization

---

- ▶ Bayes theorem-based technique for efficiently and effectively minimizing/maximizing an objective function (loss, metrics)
  1. Builds a probabilistic model of the objective function, called the surrogate function,
  2. Then searches for the parameters of this surrogate function using an acquisition function
  3. This acquisition function is then used to search for candidates evaluated on the objective function
- ▶ These candidates are the hyper-parameters that are sought to optimize the chosen function (loss, metric)

---

## How to Implement Bayesian Optimization from Scratch in Python

- ▶ <https://machinelearningmastery.com/what-is-bayesian-optimization/>

# Search strategy comparison

---

Approach	ML	DL	Manual/ Auto	Cost	Space expl.	History	Parallel/ Distributed
Babysitting	👍	👎	🐼	💰	Low	✅	No
Grid	👍	👎	💻	💰 💰 💰 💰	High	🤖	Yes
Random	👍	👍	💻	💰 💰 💰	Medium	🤖	Yes
Bayes SMBO	👍	👍	💻	💰 💰	Medium - Driven	✅	*

\* The SMBO by definition is sequential, however, *it's possible, but not trivial*, to build parallel/distributed solution upon these optimizations.

# How to search hyper-parameter in Keras

---

- ▶ <https://blog.tensorflow.org/2020/01/hyperparameter-tuning-with-keras-tuner.html>
- ▶ Use Keras-tuner: `pip install -q -U keras-tuner`  
`import keras_tuner as kt`

```
def model_builder(hp):  
    model = keras.Sequential()  
    model.add(keras.layers.Flatten(input_shape=(28, 28)))  
    hp_units = hp.Int('units', min_value=32, max_value=512, step=32)  
    model.add(keras.layers.Dense(units=hp_units, activation='relu'))  
    model.add(keras.layers.Dense(10))  
  
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])  
  
    model.compile(optimizer='adam',  
                  loss='categorical_crossentropy',  
                  metrics=['accuracy'])  
  
    return model
```

# Keras tuner

---

```
tuner = kt.Hyperband(model_builder,
                    objective='val_accuracy',
                    max_epochs=10,
                    factor=3,
                    directory='my_dir',
                    project_name='intro_to_kt')

es = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)

tuner.search(X_train, y_train, epochs=50, validation_split=0.2, callbacks=[es])

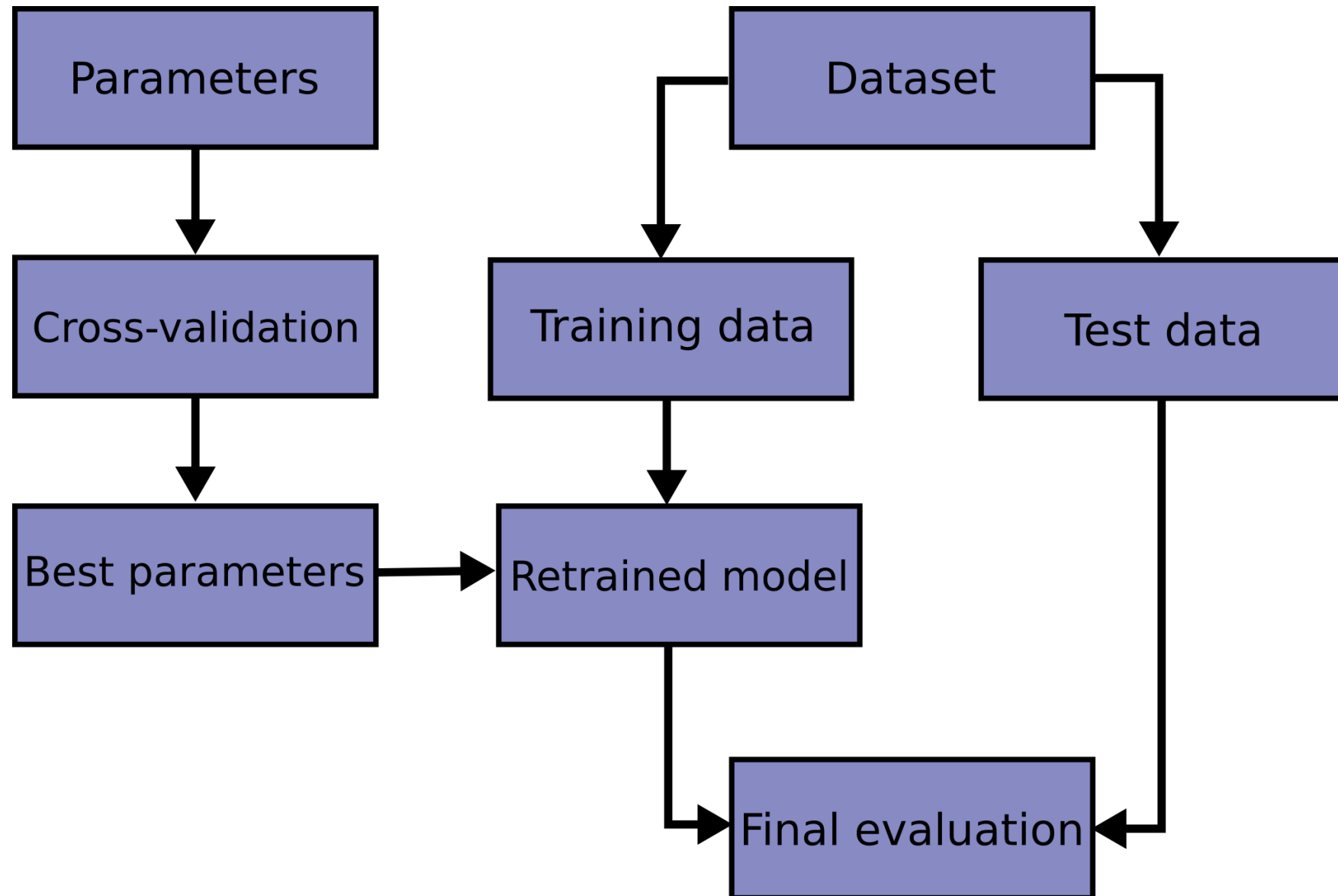
# Get the optimal hyperparameters
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the optimal hyperparameters
best = tuner.hypermodel.build(best_hps)
hist = best.fit(X_train, y_train, epochs=50, validation_split=0.2, callbacks=[es])

# Evaluate the model
eval = best.evaluate(X_test, y_test)
```

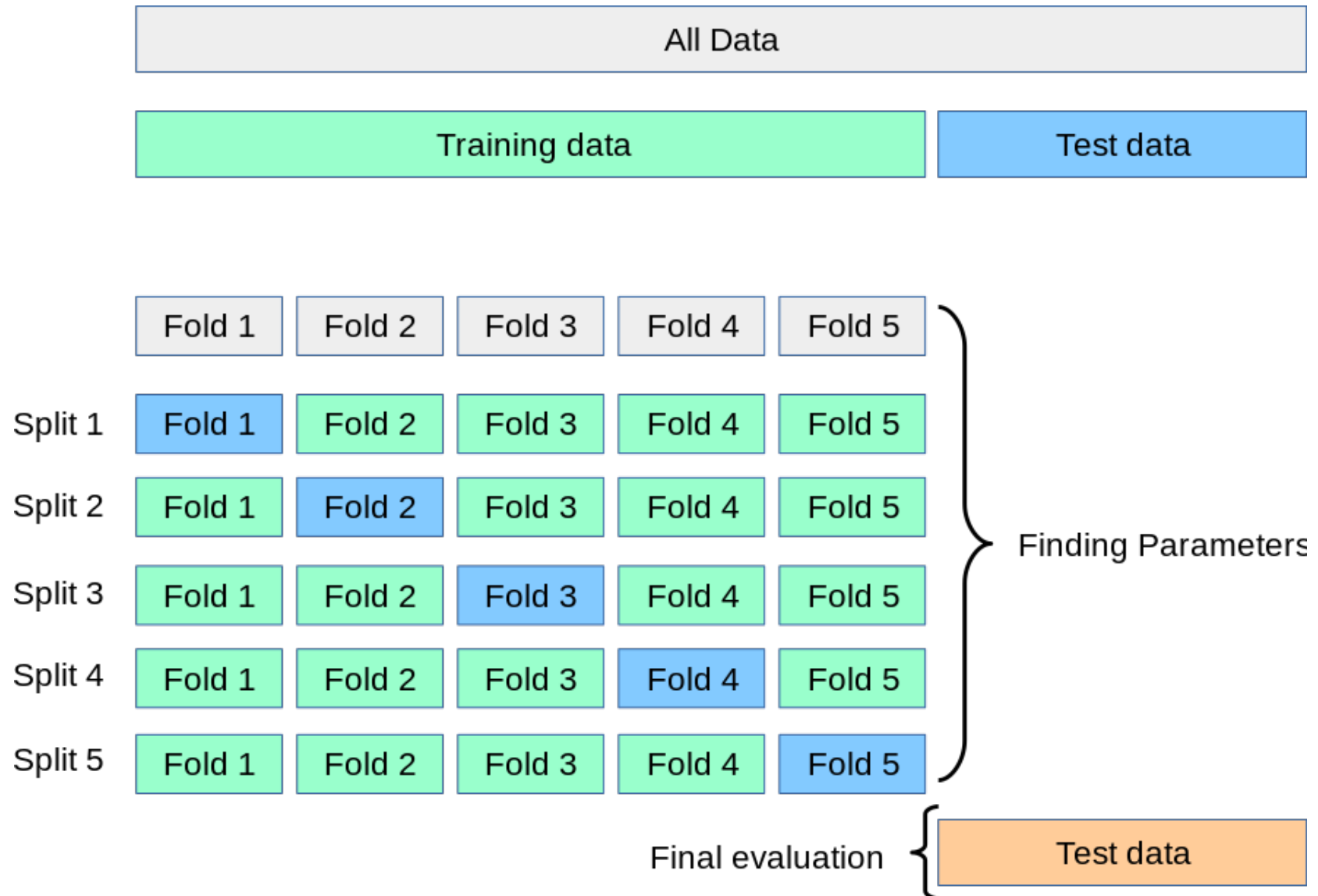


# Hyper-parameters search





# Cross validation



# Conclusion

- ▶ The search for the best hyper-parameters is necessary
- ▶ Deep network = very many parameters
- ▶ Exhaustive search = not realistic
- ▶ Use a 'support tool' e.g. keras-tuner with a Bayesian search.
- ▶ An old paper which presents the impact of various hyper-parameters
  - ▶ Lecun, Yann & Bottou, Leon & Orr, Genevieve & Müller, Klaus-Robert. (1998). Efficient BackProp.
  - ▶ <http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>