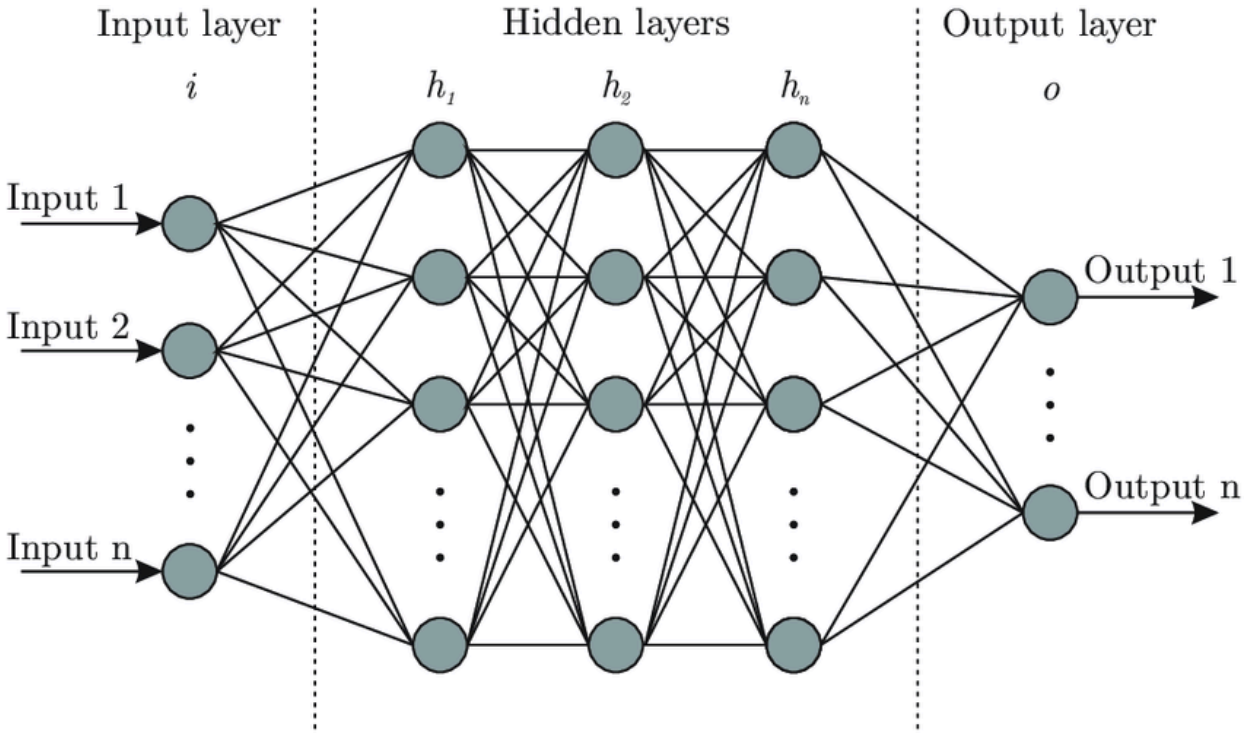
Deep Learning: Optimizer, Loss & Fitting

Plan

1. Reminders
2. What does `model.compile(...)` mean?
3. What is exactly `model.fit(...)`?
4. Regularization layers
5. Side note: Save and load models

0. Reminders

The neural network f_θ is just a composition of linear combinations



- A neural network is a stack of layers
- Each layer is composed of neurons
- A neuron is a { **linear combination** } + { **activation function** }

Keras Library

We use the Keras library to build neural networks.

- Keras functions on top of the TensorFlow package (developed by Google, which also developed the TensorFlow Playground).
- Import `tensorflow.keras` or `keras`
- Check the TensorFlow Keras documentation, or the Keras documentation.

A neural network is fully specified in three steps

```
###### Keras cheatsheet ######

# Step 1: Architecture
model = Sequential()
model.add(Input(shape=(128,)))          # ⚠️ Specify input size
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(5, activation='softmax')) # ⚠️ Depends on your task

# Step 2: Defining the optimization
model.compile(loss='categorical_crossentropy', optimizer='adam')

# Step 3: Training
model.fit(X, y, batch_size=32, epochs=100)
```

The last layer must correspond to the task at hand

```
### Regression with 1 output
model.add(layers.Dense(1, activation='linear'))
```

```
### Regression with 16 outputs
model.add(layers.Dense(16, activation='linear'))


### Classification with 2 classes
model.add(layers.Dense(1, activation='sigmoid'))


### Classification with 14 classes
model.add(layers.Dense(14, activation='softmax')
```

1. Compiling

```
model.compile(loss=...,
              optimizer=...,
              metrics=...)
```

This corresponds to the way the Neural Network optimizes its parameters θ.

```
# Regression
model.compile(loss='mse',
              optimizer='adam',
              metrics=['mae'])

# Classification with 2 classes
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

# Classification with more than 2 classes
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy', 'precision'])
```

Big picture

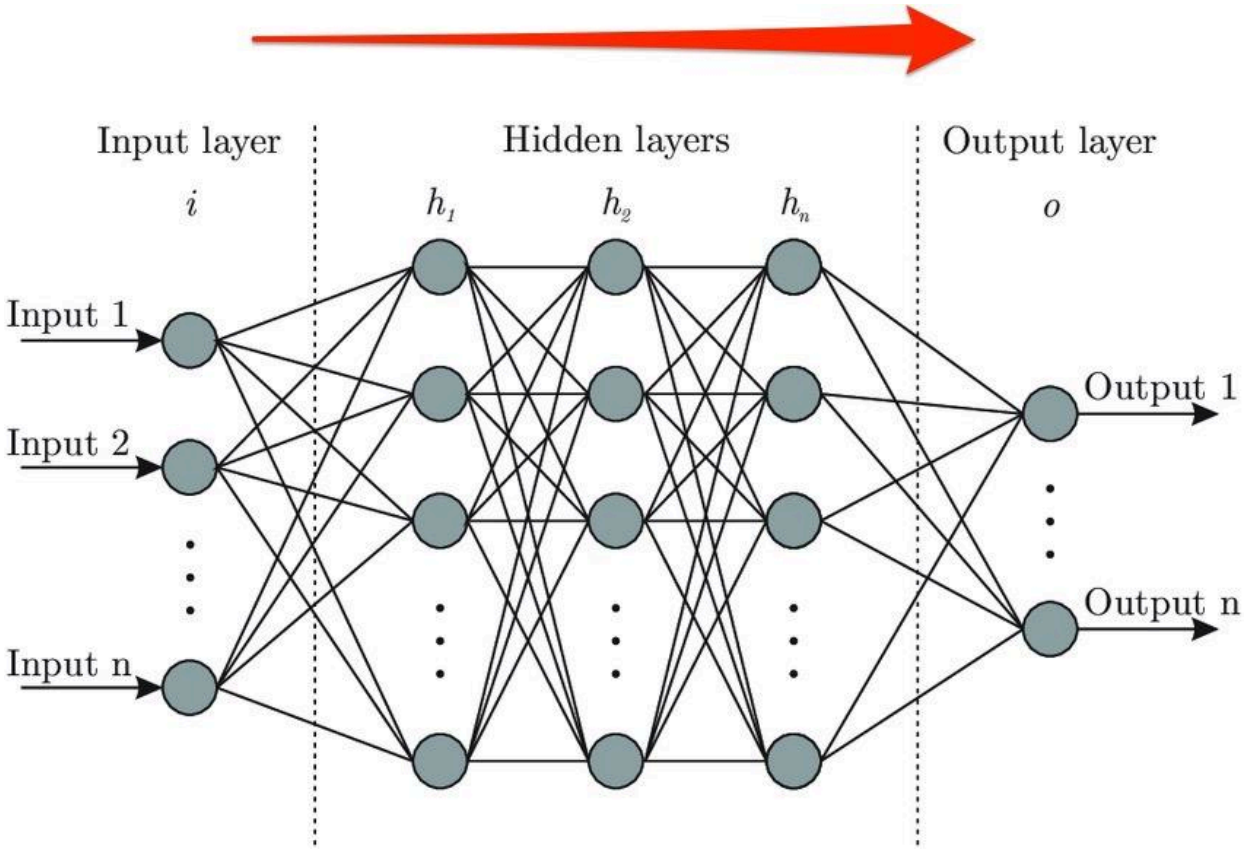
We have:

- The **model** $f\theta$ that predicts $\hat{}y=f\theta(X)$
- The **loss function** $Lx(\theta)$ which measures a "smooth" distance between $\hat{}y$ and $y \rightarrow$ used to optimize θ.
- The **metric(s)** that measures "human" distances between $\hat{}y$ and $y \rightarrow$ used for performance evaluation.

1.1 The `metrics`

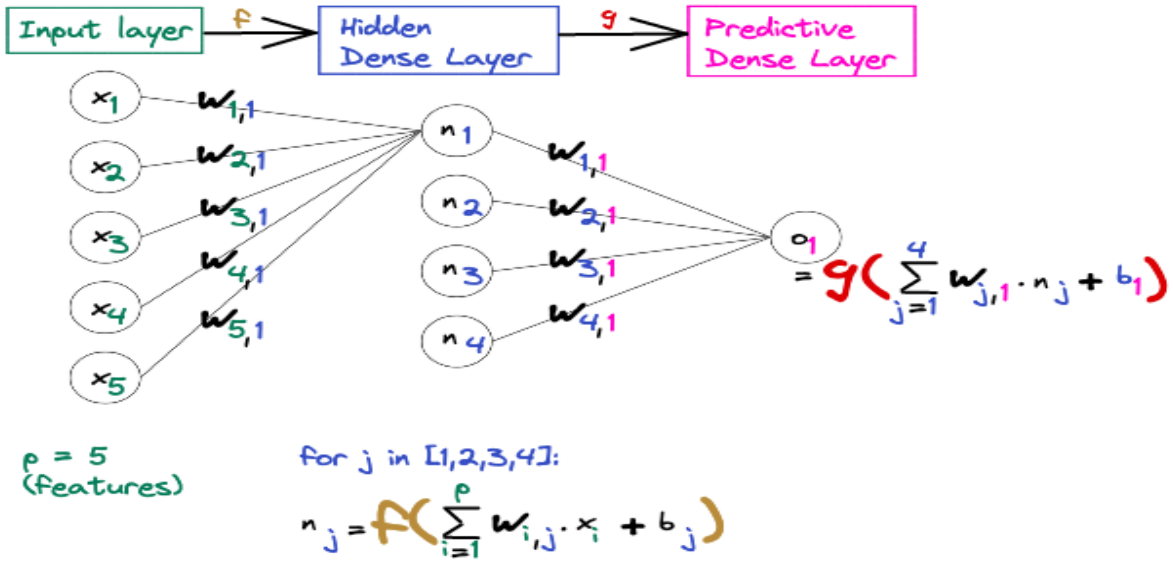
= human measures of *how good* the predictions are

→ computed by **forward propagation** at each epoch



🤔 What do we mean by **forward propagation** ?

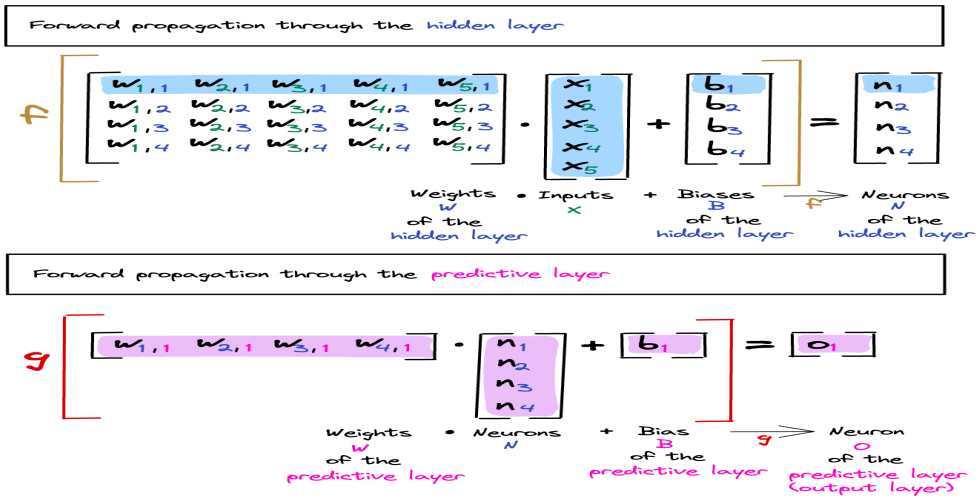
👇 Let's illustrate it through a quite simple Neural Network:



❓ How many parameters does this neural network have ❓

$((5+1)\times 4)+((4+1)\times 1)=24+5=29$

💡 **Forward Propagation = 1 matrix multiplication per layer**



Common metrics

- for ***classification tasks***:
  - Precision, Recall, Accuracy, F1-score, ROC-AUC, ...
- for ***regression tasks***:
  - MSE, MAE, RMSE, RMSLE, R-squared, ...

🧐 *Did you know?*

All metrics (and losses) are based on the mathematical notions of:

- **distances** (the smaller the better) : e.g. Euclidian, Manhattan, ...
- or **similarities** (the larger the better): e.g. Cosine, Jaccard ...

between two points A,B in a vector space

📚 Read more

In Keras (📚 Docs about the metrics):

```
# use strings for quick access
model.compile(metrics=['accuracy', 'precision'])

# use Keras metric objects for fine-tuning
auc_metric = keras.metrics.AUC(
    num_thresholds = 200,
    curve='ROC', # or curve='PR'
)
model.compile(metrics=[auc_metric])

# Custom metrics
def custom_mse(y_true, y_pred):
    squared_diff = tf.square(y_true - y_pred)
    return tf.reduce_mean(squared_diff)

model.compile(metrics=[custom_mse])
```

💻 Do not hesitate to play with `Tensor` objects, they are quite similar to Numpy `Array`

```
import tensorflow as tf

X = tf.ones((3,3))
X
```

```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]], dtype=float32)>
```

```
X.numpy()
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
```

1.2 The `loss function` $Lx(\theta)$

= the function you choose to optimize your algorithm!

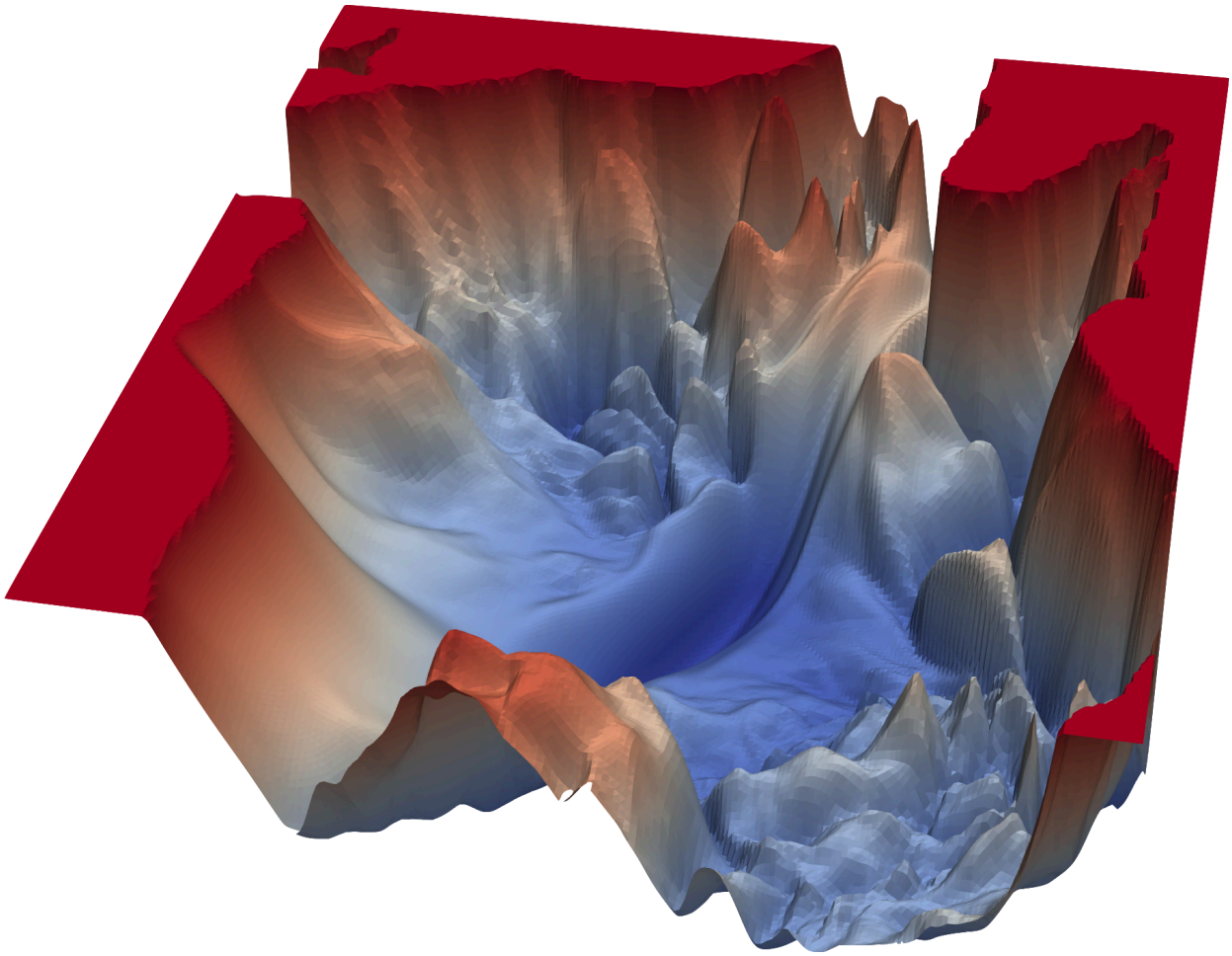
- `.fit()` aims at finding θ that **minimizes** the loss function
- Not necessarily equal to the metric
- Only one loss per model (whereas you can evaluate your model with different metrics!)
- Must be "smooth" (so that we can compute its gradient)

Loss function has to be "smooth"

It is not possible to use any function for the loss:

- Loss functions must be **continuous** and **(sub)differentiable** with respect to θ
- i.e. the distance between $f\theta(X)$ and y has to vary smoothly when θ varies
- So you can compute its gradient

Your "energy landscape" has to be smooth (but may be very complex) :



Examples:

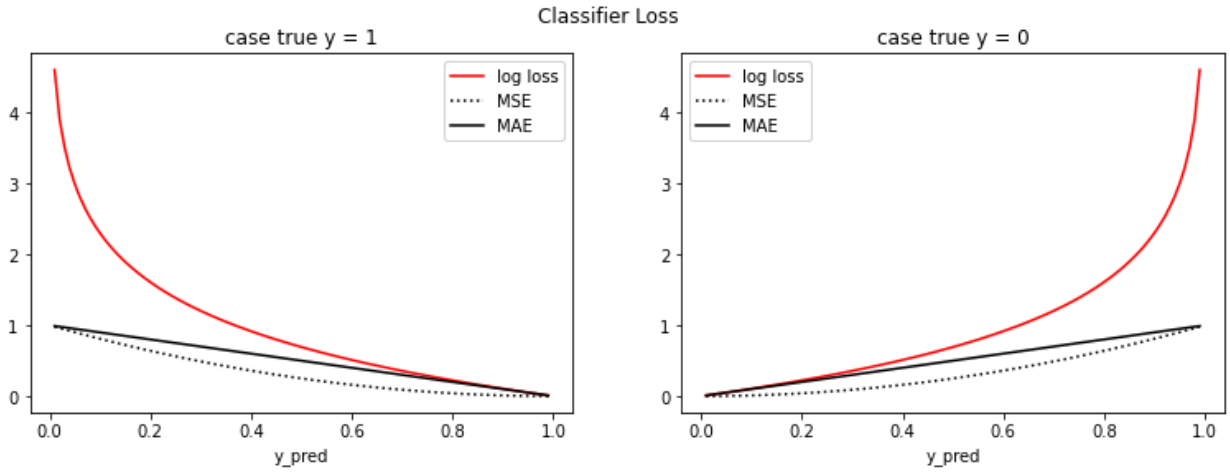
❌ **Accuracy** *jumps* from 0 to 1 as θ varies

✅ **Cross-entropies** output probabilities which vary smoothly

Binary Cross-Entropy (= Log Loss)

$$Log\ Loss = -\frac{1}{n}\sum_{i=0}^{n} y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i)$$

- $y = 1 \Rightarrow Log\ Loss = -log(\hat{y})$
- $y = 0 \Rightarrow Log\ Loss = -log(1-\hat{y})$



💡 The "Cross-Entropy" name comes from 📺 Shannon's Information Theory

In Keras?

```
# Use strings for quick access
model.compile(loss="binary_crossentropy")


# Use Keras metric objects for fine-tuning
loss = keras.losses.BinaryCrossentropy(...)
model.compile(loss=loss)

# Custom losses
def custom_mse(y_true, y_pred):
    squared_diff = tf.square(y_true - y_pred)
    return tf.reduce_mean(squared_diff)

model.compile(loss=custom_mse)
```

Pick a loss available in the documentation

Summary

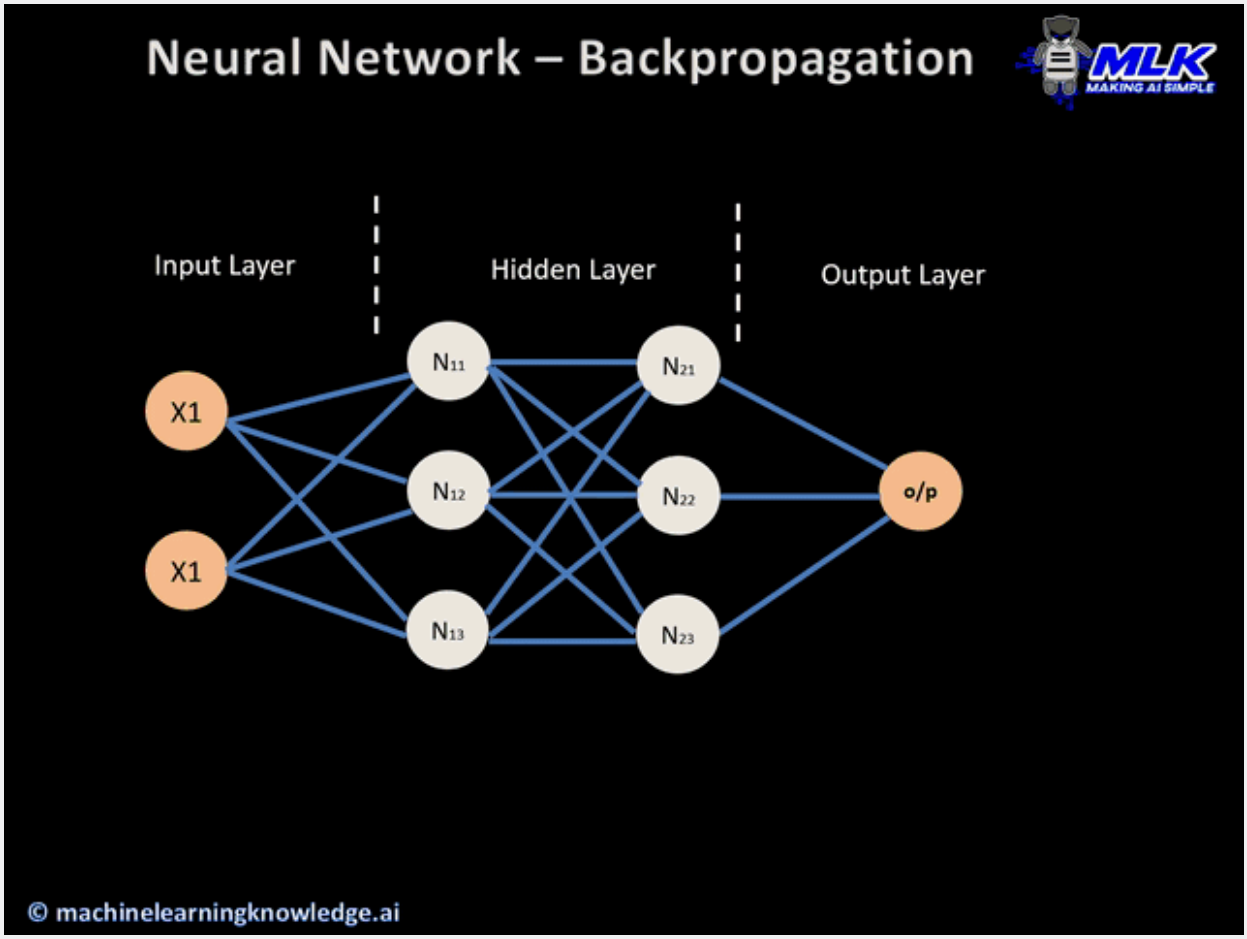
We have:

- The **model** f_θ that predicts $\hat{y}=f_\theta(X)$
- The **loss function** $L_X(\theta)$ which measures a "smooth" distance between $\hat{y}$ and y → used to optimize θ.
- The **metric(s)** that measures "human" distances between $\hat{y}$ and y → used for performance evaluation.

➡️ Next step: **how do we minimize the loss?**

👉 This is where the **optimizer** comes in!

2. The `optimizer`



2.1 Forward vs. Backward Propagation

The optimizer is "fed" with data **batch-by-batch** to update the weights of the model iteratively.

At each iteration k:

1. **Forward propagation**
    - Given *some* input X_{batch}, the model computes $f_{\theta_k}(X_{batch})$

    - The loss for this batch is computed $L(\theta_k)$
2. **Backward propagation**
    - The gradient of this loss ∇L is computed
    - The weights are updated using this gradient: $\theta_{(k+1)} \leftarrow Update(\theta_{(k)}, \nabla L)$

Why is it called `backpropagation`?

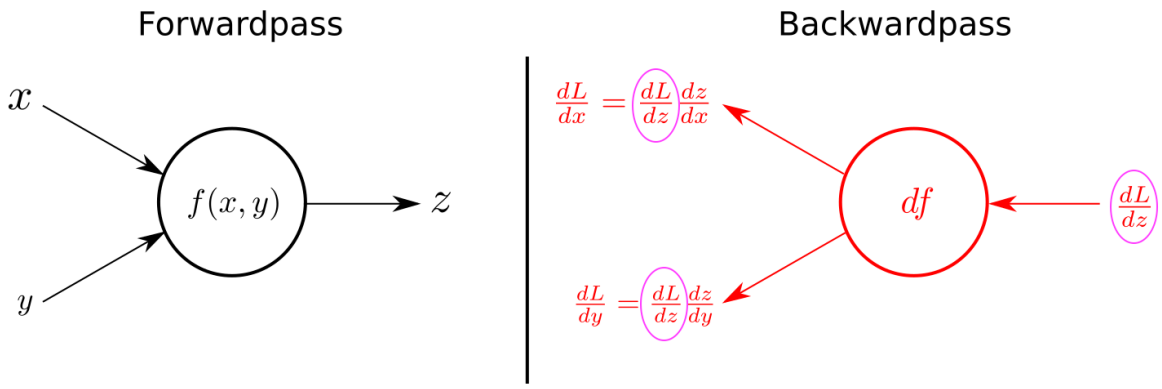
- Remember that we need to compute the gradient of the loss
- It consists of a very large number `p` of partial derivatives (one per weight)
- $\nabla L(\theta) = \begin{bmatrix} \partial L \partial \theta_1(\theta) \dots \partial L \partial \theta_p(\theta) \end{bmatrix}$.
-

If we were to compute them **numerically** one-by-one using the definition: $\partial L \partial \theta_i = \lim$

we would need to compute `p` forward passes per iteration! 🤯

💡 Instead, mathematicians have developed a clever **analytical** method to compute all these partial derivatives together:

- The network is made up of many simple **composite** functions (ex: addition, multiplication, sigmoids, ReLU ...)
- Individually, their derivatives are easy to compute (ex: derivative of ReLU is either 0 or 1)
- The derivative of the loss can be obtained from these individual contributions using the **Chain Rule**



☝️ If we iterate backward from the output layer, we can **re-use** many terms computed at previous steps.

📺 Khan Academy - Chain Rule (5 min)

Key benefits? The speed ⚡

One iteration (update) on a minibatch uses only:

➡️ 1 forward pass

- computes the outputs for each observation of the minibatch
- computes the loss for this minibatch
- and *stores intermediary computations in RAM*

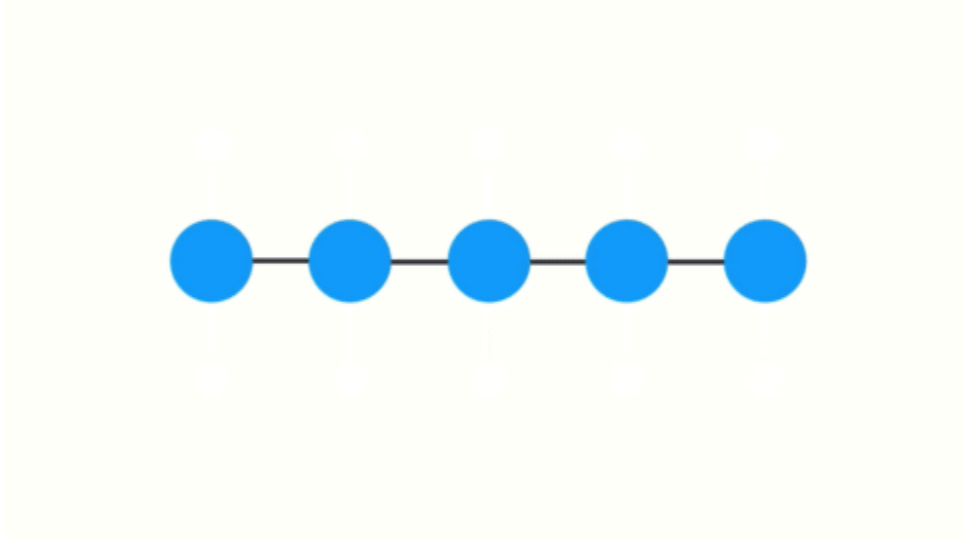
⬅️ 1 backward pass through the network (*re-use intermediary values for the chain rule*)

💡 The chain rule allows backpropagation to take (roughly) the same time as a forward pass!

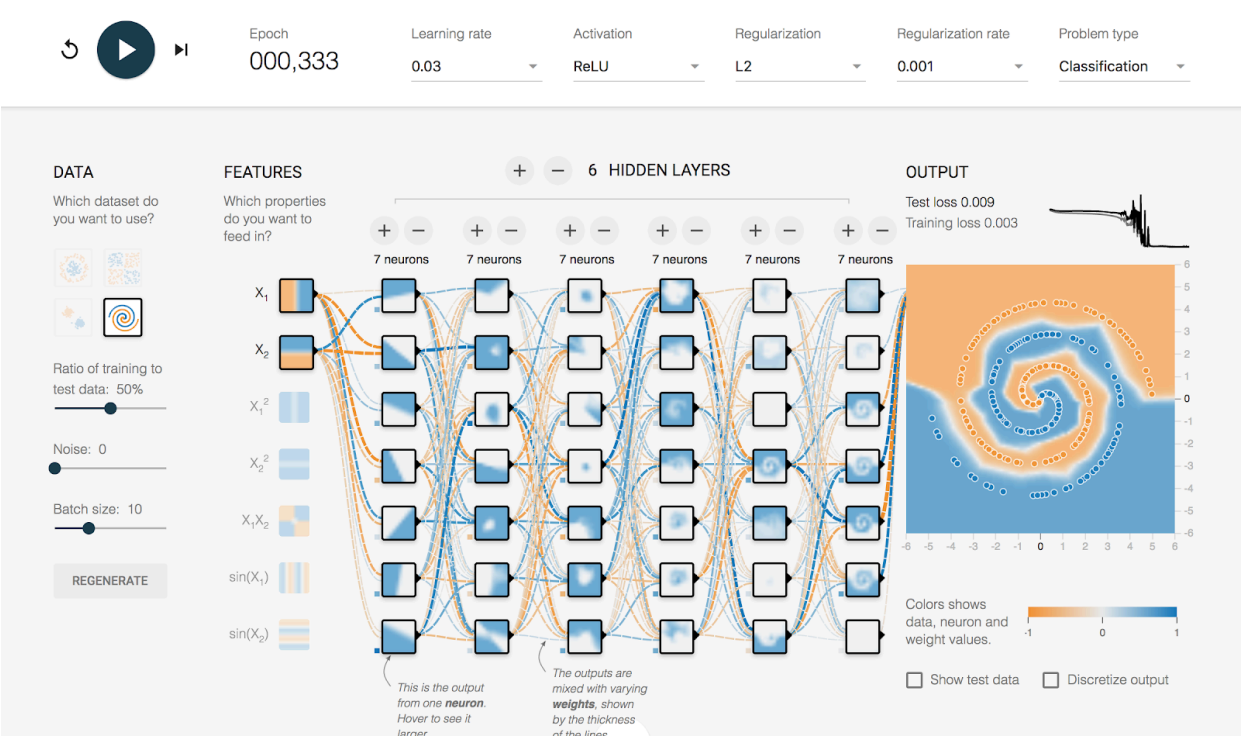
Popularized in 1987, this discovery is at the origin of a rush of people using neural networks.

❗ **Vanishing gradient** phenomenon ❗

> The weights of the first (deeper) layers are **harder** to move than the weights of the last layers (closer to the output)



👉 Playground example

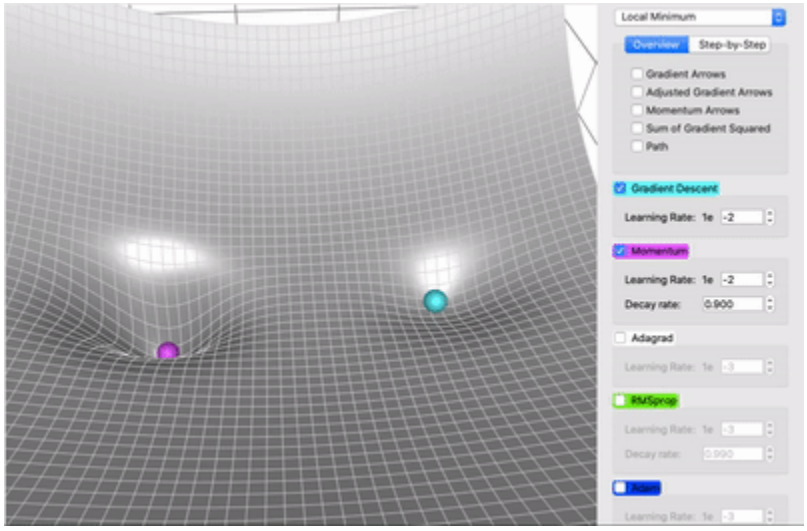


2.2 Which optimizer to choose?

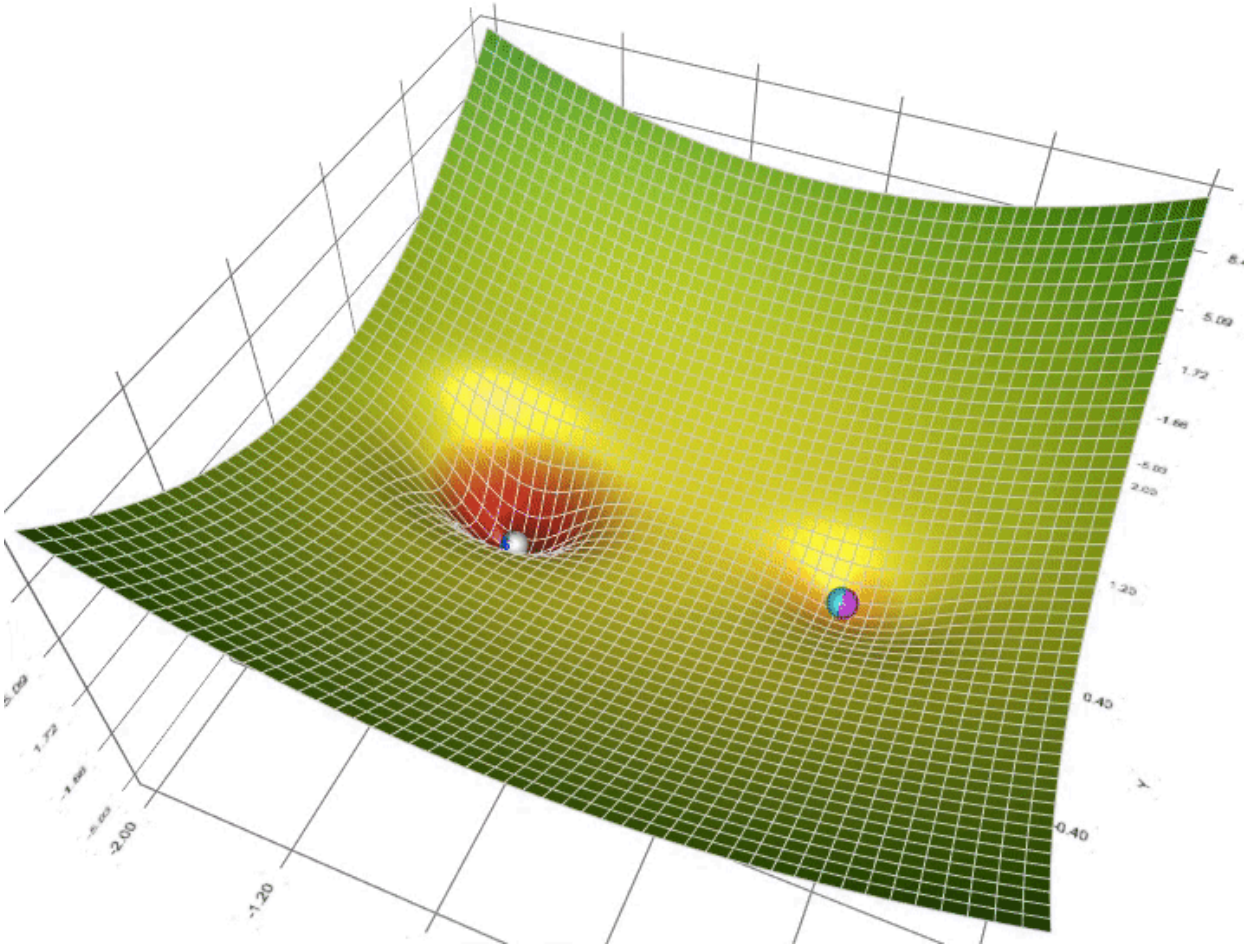
Simple gradient descent is not good enough in Deep Learning.

Loss functions are too complex and the optimizer gets stuck at **local minima**.

The documentation offers multiple choices: Adam, SGD, RMSProp, Adadelta, ...



- Gradient
- Momentum (adds inertia)



- Gradient
- Momentum (adds inertia)
- AdaGrad (adaptative learning rate per feature - priorize weakly updated params)
- RMSProp (adds decay - only recent gradient matters)
- Adam (all combined)

📚Credits

Adam = best choice of optimizer to start with

3. `optimizer` hyper-parameters

When you initialize an optimizer using a string, you start with the default hyperparameters.

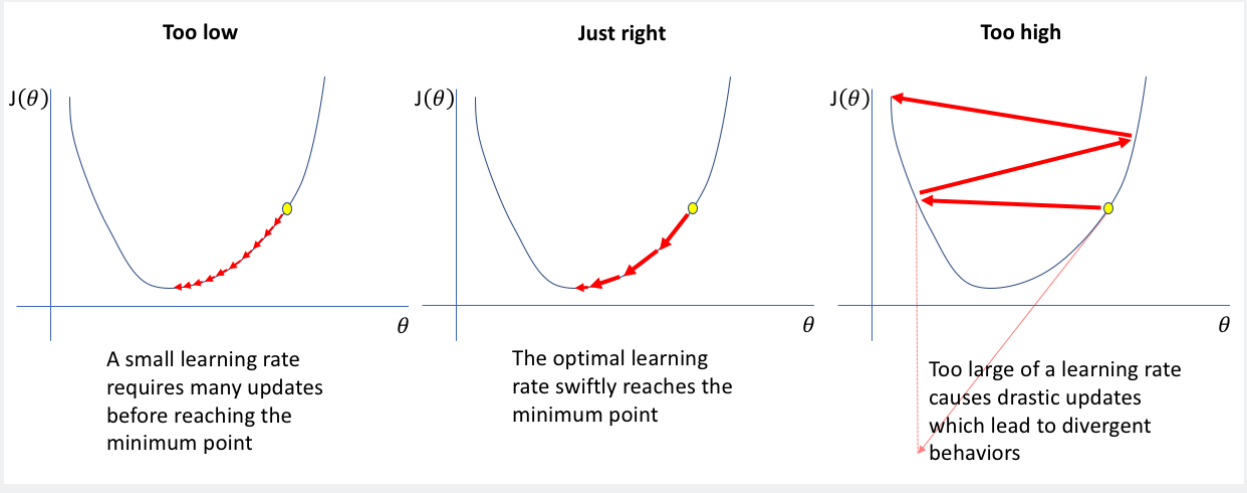
```
model.compile(loss=..., optimizer='adam')
```

You can select your own hyperparameters thanks to the following syntax:

```
opt = tensorflow.keras.optimizers.Adam(
    learning_rate=0.01, beta_1=0.9, beta_2=0.99
```

)
model.compile(loss=..., optimizer=opt)

3.1 `learning_rate`



- Start with the default implementation.
- Think about this rate as the **amount of change on the weights you want at each update**.
- Smaller rates will require more epochs

💡 For a deeper understanding of the learning rate, let's check this animation.

🏋️ : Schedulers (cf challenges) offer some clever ways to have a changing learning rate during training.

3.2 `batch_size`

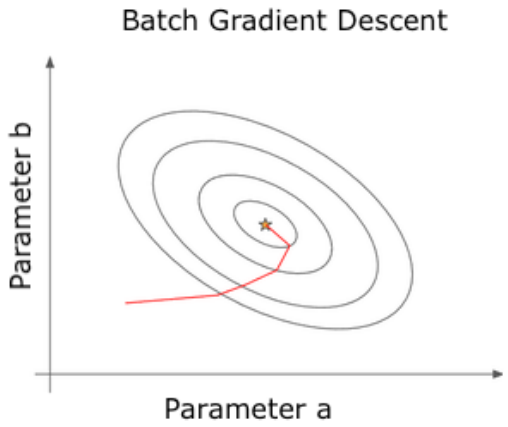
Let's say that you have 100 samples X_1, X_2, X_{100} and `batch_size=10`

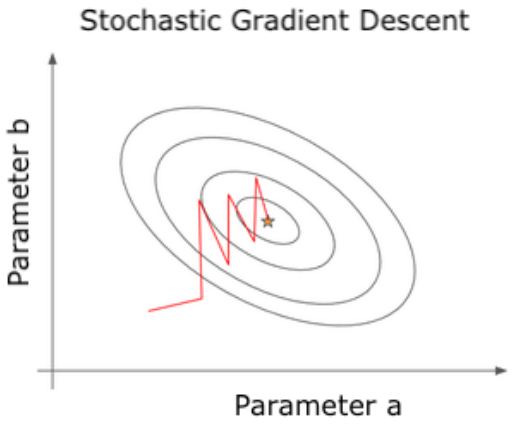
One epoch corresponds to:

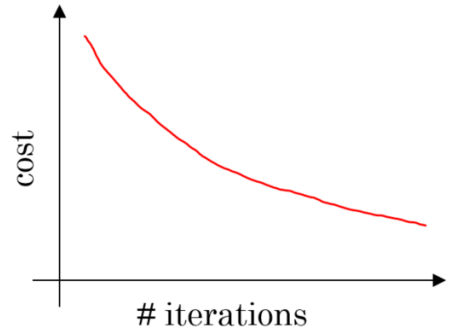
- \theta^{(0)} \rightarrow \theta^{(1)} on the samples (X_{1}, X_{2}, \dots, X_{10}),
- \theta^{(1)} \rightarrow \theta^{(2)} on the samples (X_{11}, X_{12}, \dots, X_{20}),
- ...
- \theta^{(9)} \rightarrow \theta^{(10)} on the samples (X_{91}, X_{92}, \dots, X_{100}),

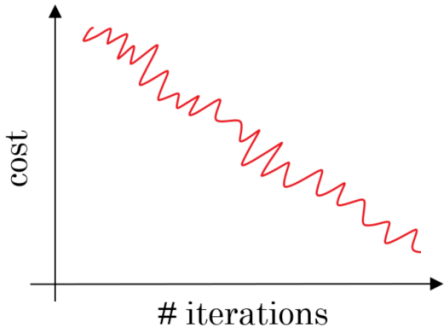
👉 Refer to the unit *`05-Machine-Learning/04-Under-The-Hood`* on Kitt

- The smaller the batch, the more stochastic the process is and the faster it may converge
- The larger the batch, the better it generalizes, but the more computationally intensive it becomes









Which batch size to choose?

Yann Le Cun has an answer for you:

In reality, it depends on the size of your inputs

- 16 or 32 is most for real-word data (eg: 128 \times 128 \times images)
- You can use more for very small data (eg: Tabular datasets, tiny images, etc...)

❗ **Remark** ❗ Why a power of 2? \rightarrow for computational reasons!

3.3 epochs number?

❗ The larger the batch size, the more epochs you will need ❗

How many epochs?

It does not matter, as many as possible as long as the neural network is able to generalize to unseen data.

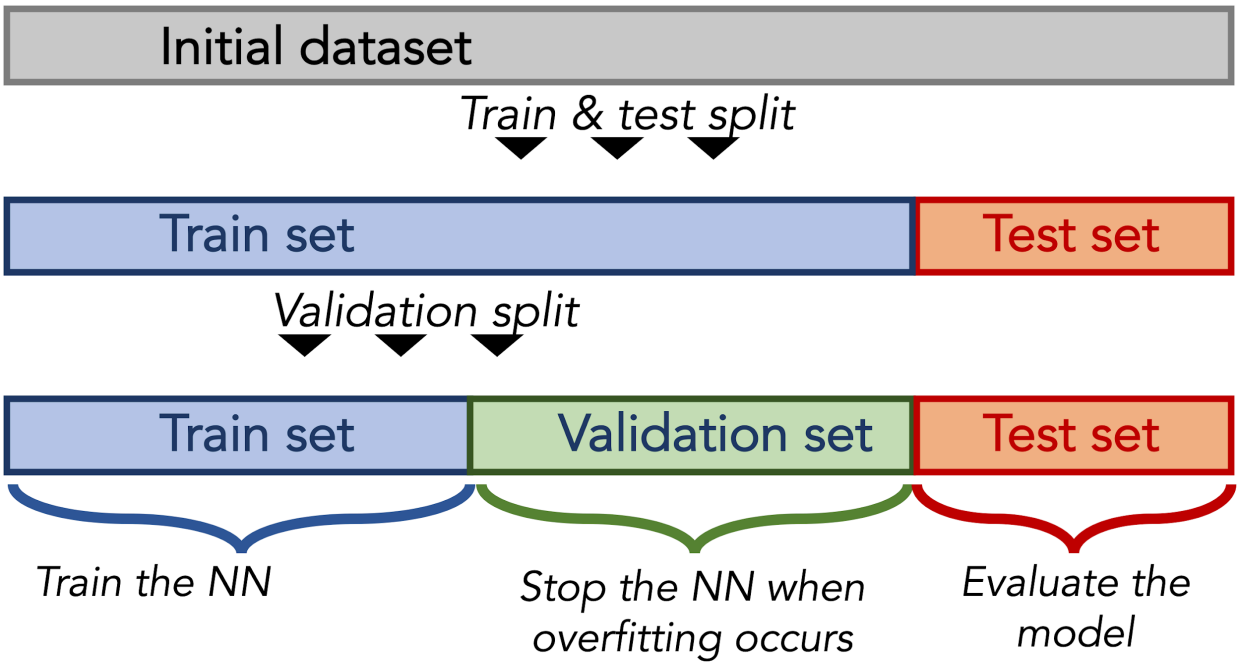
🤔 **When does my model start overfitting?**

💡 *Hint*: Do *NOT* use your real test set.

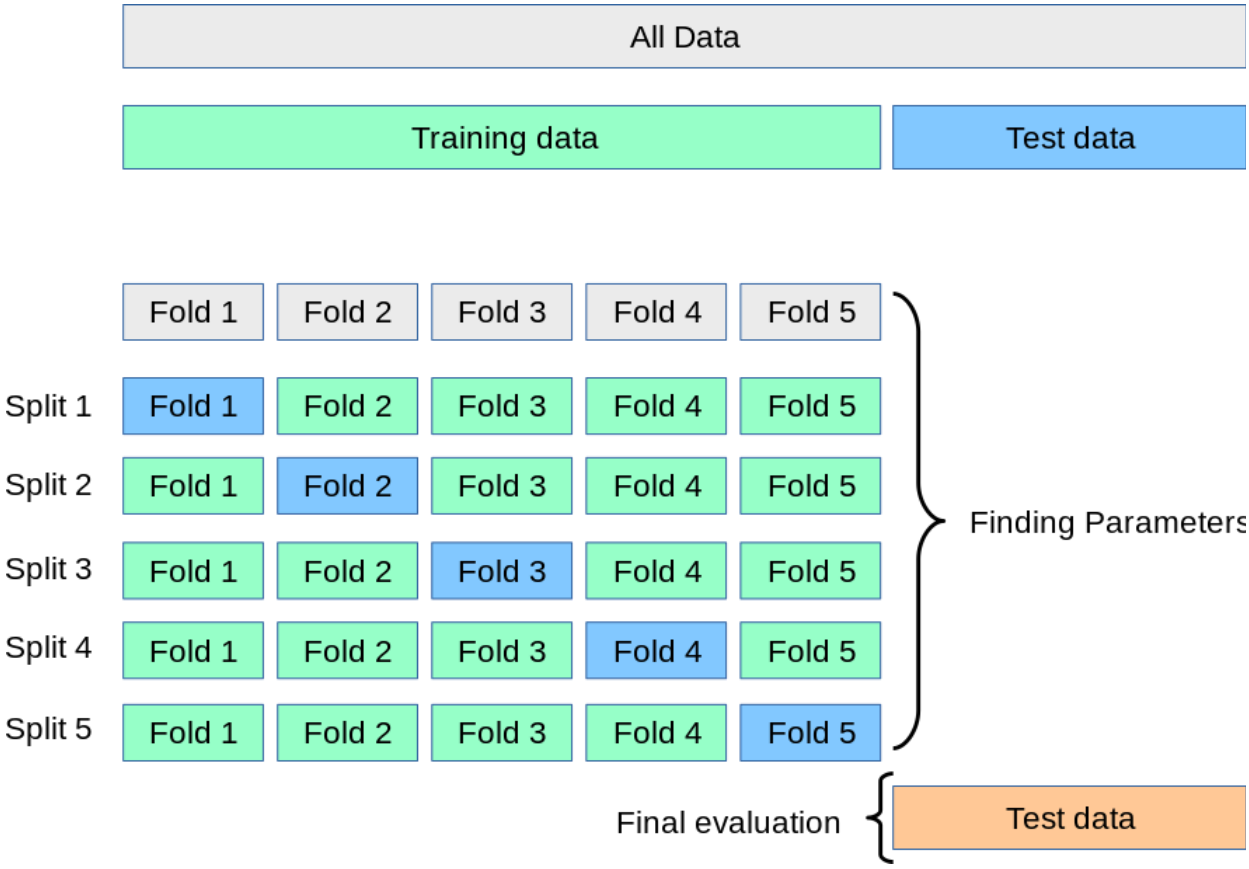
Train/Val/Test split

❗ If you use the test set to stop your algorithm (to prevent overfitting), you use the test set to *optimize* your algorithm. ❗

To prevent this type of data leakage, instead of estimating the loss on the test set, we do it on part of the training set. We call this the **validation set**.



❗ **K-fold cross-val** is better than a single holdout ❗



❗ **Cross-val in deep learning can be very time consuming** ❗

From now on, we will only do the split once.

In real life, don't forget to perform a real cross-validation.

In Keras, we have two options:

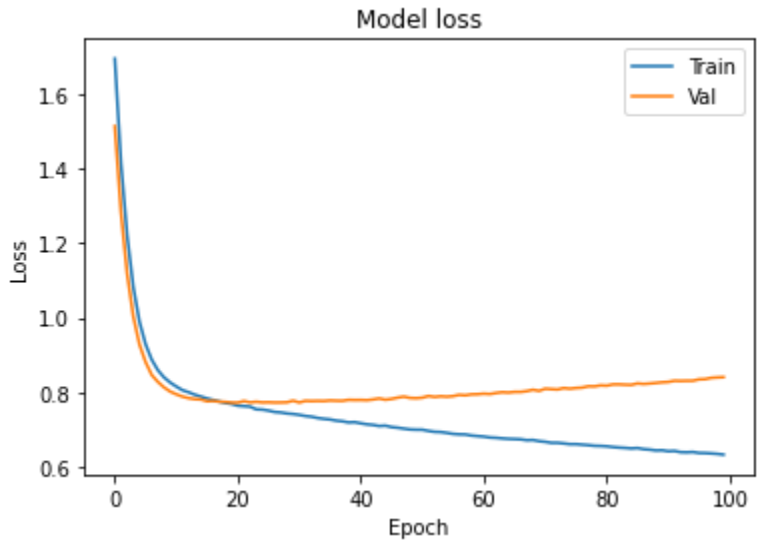
```
# Give validation set explicitly
history = model.fit(X_train, y_train,
          validation_data=(X_val, y_val),
          batch_size=16,
          epochs=100)
```

Or, use directly the `validation_split` keyword:

```
history = model.fit(
    X_train, y_train,
    validation_split=0.3, # ⚠️ LAST 30% of train indexes used for validation
    batch_size=16,
    epochs=100,)
    # shuffle=True)      # Training data is shuffled at each epoch by default 👍
```

Note: The validation split happens before the shuffling: for every epoch the same data is used for validation. So no data leakage 👍

Your `history` may then look like this:



3.4 `Early stopping`

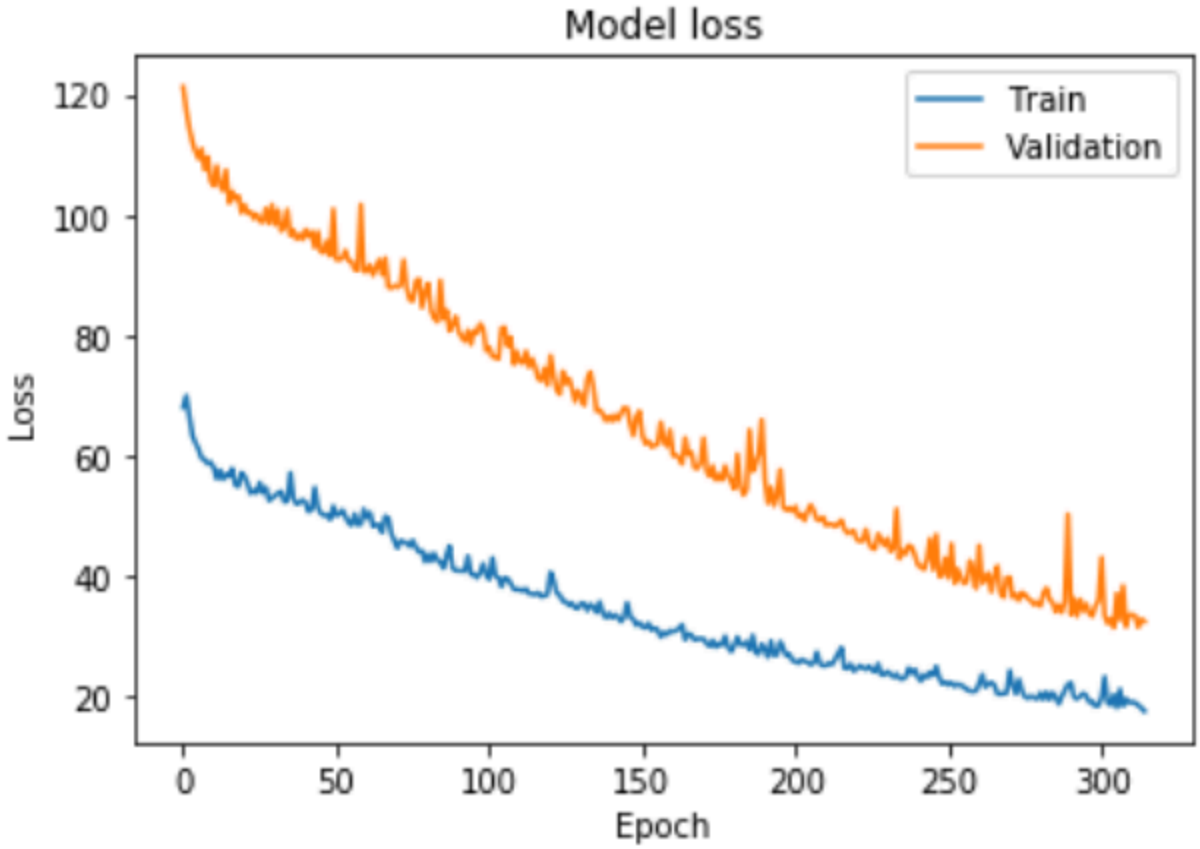
The previous code evaluates the model on the validation data at the end of each epoch. However, it will not stop training the model even after it starts overfitting.

👉 To properly stop the algorithm from training, we use the **Early Stopping Criterion**.

- It stops the algorithm if the validation loss at `epoch` k+1 is worse than at `epoch` k.

```
from tensorflow.keras.callbacks import EarlyStopping

es = EarlyStopping()

model.fit(X_train, y_train,
          batch_size=16,
          epochs=1000,
          validation_split=0.3,
          callbacks=[es])

# "callback" means that the early stopping criterion
# will be called at the end of each epoch
```

In reality, Neural Networks are **stochastic** algorithms and the loss can look like that:



👉 If you stop the run as soon as the validation loss gets worse, you will stop it too soon, even before 10 epochs.

We need to allow a given number of iterations without improvement, with the `patience` keyword.

```
es = EarlyStopping(patience=20)

model.fit(X_train, y_train,
          batch_size=16,
          epochs=1000,
          validation_split=0.3,
          callbacks=[es])
```

👇 Same patience but restores the weights that correspond to the best validation loss:

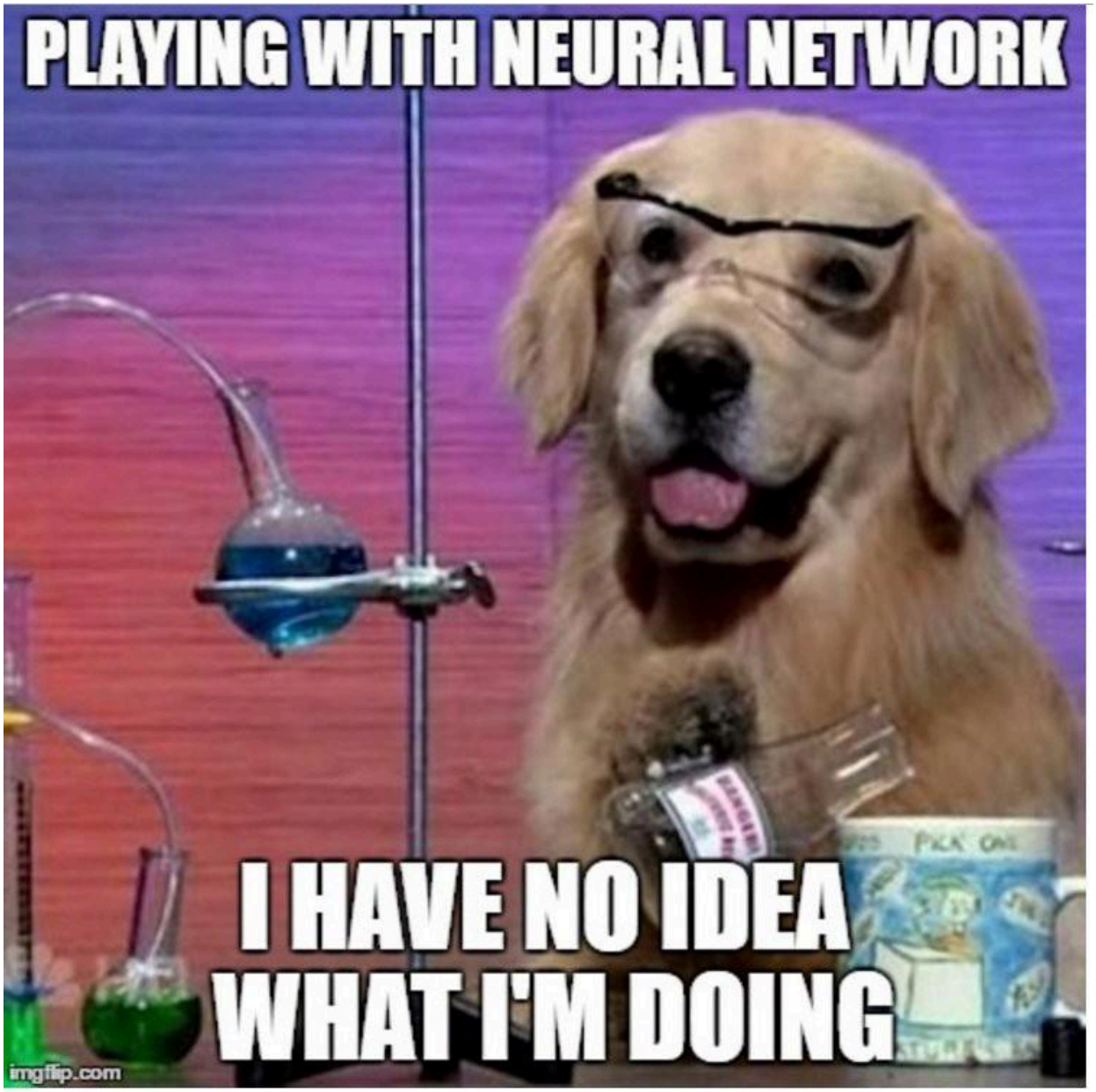
```
es = EarlyStopping(patience=20, restore_best_weights=True)

model.fit(X_train, y_train,
          batch_size=16,
          epochs=1000,
          validation_split=0.3,
```

```
        callbacks=[es])
```

4. Regularization

All the previous steps are somewhat *mandatory*: they are fundamental to building a Neural Network.

However, in practice, you might still feel lost as to **how** to design the neural network architecture. This is absolutely normal when you begin.



The art of *Deep Learning* is not to build the best architecture at first sight...

... but to try an initial architecture...
... and **know what to change to improve it according to the results**!

This is why practice is so important.
For instance, a common problem is that a Neural Network is **overfitting**! How do we solve it?

✅ **Early Stopping** criterion *stops* the model from learning before it overfits.

✅ **Regularization** layers really try to *prevent* your Neural Net from overfitting (📚 documentation).

4.1 Regularizers (L1, L2)

The regularization layers act the same way as L1 and L2 regularization in "vanilla" Machine Learning.

The Neural Network will optimize the loss you declared *plus* this regularization!
L2\ Loss = Loss\ + \color{red}{\alpha} \sum_{i} \theta_i^2L1\ Loss = Loss\ + \color{red}{\alpha} \sum_{i} | \theta_i|

But... to which \theta do you apply this? All of them?
👉 ***Layer per layer*** 👈

You can actually apply it to:

- All the **weights** w_i of a given layer (also called the `kernel_regularizer`)
- All the **biases** b_i of a given layer (`bias_regularizer`)
- The entire **output** of a given layer (`activity_regularizer`)

❗ This regularization is "active" only during the training

```
from keras import regularizers, Sequential, Input, layers

reg_l1 = regularizers.L1(0.01)
reg_l2 = regularizers.L2(0.01)
reg_l1_l2 = regularizers.l1_l2(l1=0.005, l2=0.0005)

model = Sequential()
model.add(Input(shape=(3,)))
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dense(50, activation='relu', kernel_regularizer=reg_l1))
model.add(layers.Dense(20, activation='relu', bias_regularizer=reg_l2))
model.add(layers.Dense(10, activation='relu', activity_regularizer=reg_l1_l2))
model.add(layers.Dense(1, activation='sigmoid'))
```

❓ What is the number of additional parameters induced by the regularization ❓

```
# Answer: zero additional parameters
model.summary()
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 100)	400
dense_6 (Dense)	(None, 50)	5,050
dense_7 (Dense)	(None, 20)	1,020
dense_8 (Dense)	(None, 10)	210
dense_9 (Dense)	(None, 1)	11

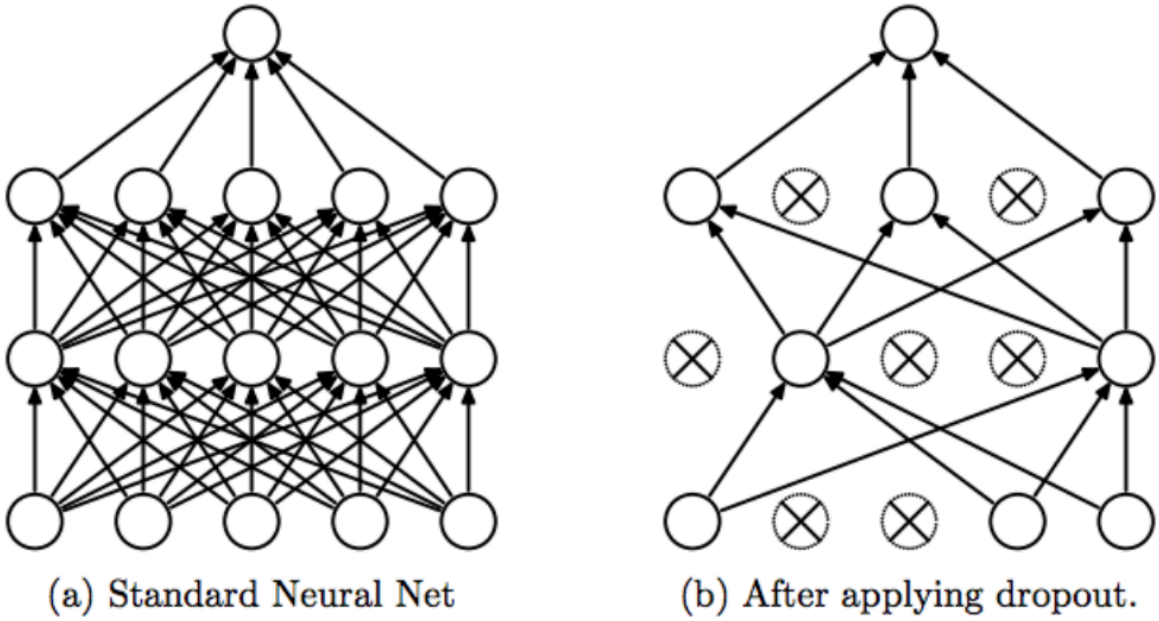
```
Total params: 6,691 (26.14 KB)

Trainable params: 6,691 (26.14 KB)

Non-trainable params: 0 (0.00 B)
```

```
# Indeed: for instance layer 2 is regularized
# yet has usual number of parameters
(100 + 1) * 50
```

5050

4.2 Dropout layer

During the training, at each iteration, the dropout randomly "kills" (=0) the activity from some neurons so their weights are not updated:



(a) Standard Neural Net (b) After applying dropout.

Consequences:

- Prevents any neuron from updating its weights only according to a particular input
- Prevents neurons from over-specializing / being too specific to this input and unable to generalize

```
model = Sequential()
model.add(Input(shape=(56,)))

model.add(layers.Dense(20, activation='relu'))
model.add(layers.Dropout(rate=0.2))  # The rate is the percentage of neurons that are "killed"

model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dropout(rate=0.2))

model.add(layers.Dense(3, activation='softmax'))


# —— What is the number of parameters of the Dropout layer?
```

```
# Zero additional parameters
```

model.summary()

```
Model: "sequential_2"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ dense_10 (Dense)                     │ (None, 20)                  │           1,140 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout (Dropout)                    │ (None, 20)                  │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_11 (Dense)                     │ (None, 10)                  │             210 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout_1 (Dropout)                  │ (None, 10)                  │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_12 (Dense)                     │ (None, 3)                   │              33 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
Total params: 1,383 (5.40 KB)
Trainable params: 1,383 (5.40 KB)
Non-trainable params: 0 (0.00 B)
```

Pro Tips 💡

Strongly recommended

- Start your model from the **last layer**
- Implement the **easiest architecture** first
- Stick with the same **batch size** (eg. 32 for large data, more if it fits on RAM). Change it only once you are confident about its impact.

- Don't think about the number of **epochs**: it has to hit your `early_stopping` criterion

Apply wisely

- Try to make your model **overfit** before regularizing (it is good to see that your model does learn, even too much!)
- If you can't overfit, try fine-tuning your **learning rate** (or change the model's architecture)
- If the train loss was on a *steep decreasing trajectory* when hitting early stopping, chances are that **regularization** will improve performance
- Only then, you can regularize
- Try to regularize the last layers before the first ones

5. Bonus: Preprocessing Pipelines in Tensorflow

```
# Imagine a dataset with one feature scaled differently

X,y = make_regression(n_samples=500, n_features=5, n_targets=3)
X[:,4]= 100 * X[:,4] + 50
pd.DataFrame(X).head()
```

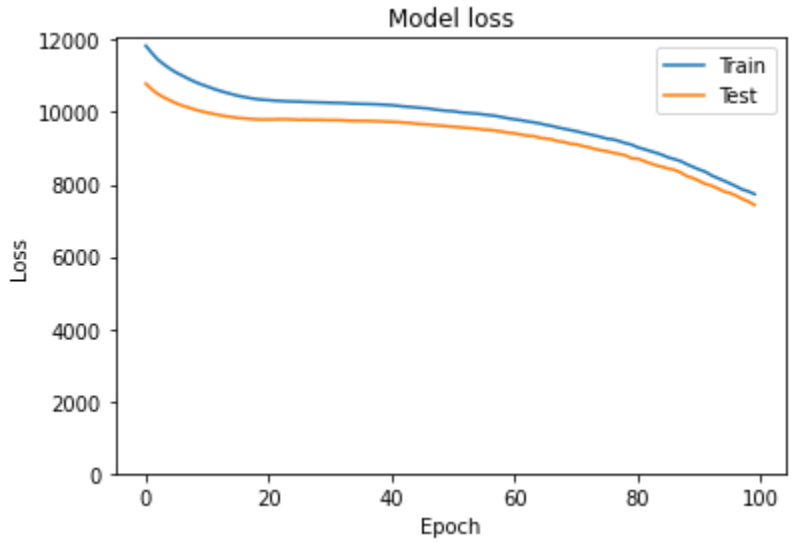
	0	1	2	3	4
0	-0.060995	-0.008898	-0.606240	-1.064232	-11.377092
1	0.906992	0.282420	0.261933	-0.892490	73.431349
2	-1.449867	-0.789354	-0.820485	1.003394	228.846071
3	-1.777572	-0.616447	-0.421114	0.536362	-85.798070
4	-0.665721	-1.967192	0.598795	-0.261152	-59.285840

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)
```

Bad option: No scaling

```
model = Sequential()
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(8, activation='relu'))
model.add(layers.Dense(3, activation='linear'))
```

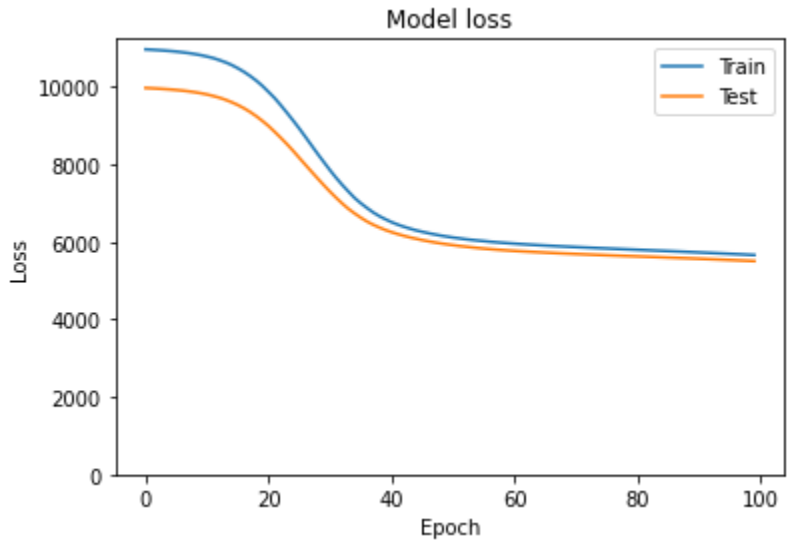
```
model.compile(loss='mse', optimizer='adam', metrics='mae')

history = model.fit(
    X_train, y_train, validation_data=(X_test, y_test),
    epochs=100, batch_size=32, verbose=0)

plot_history(history)
```



Option 1: Scale outside of of the neural network architecture

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

model = Sequential()
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(8, activation='relu'))
model.add(layers.Dense(3, activation='linear'))

model.compile(loss='mse', optimizer='adam', metrics='mae')

history = model.fit(
    X_train_scaled, y_train,
    validation_data=(X_test_scaled, y_test),
    epochs=100, batch_size=32, verbose=0)
```

```
plot_history(history)
```



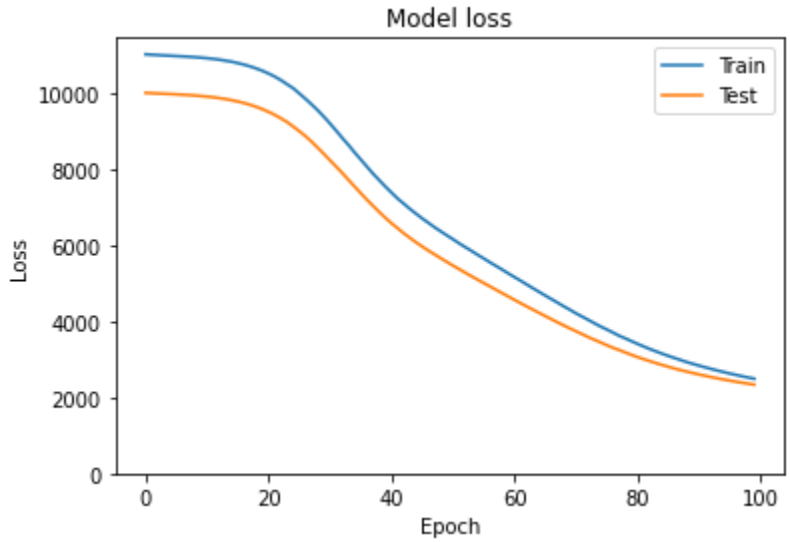
💡 **Option 2**: Integrate scaling **within the model's architecture** using `Normalization` layers

```
from tensorflow.keras.layers import Normalization

normalizer = Normalization() # Instantiate a "normalizer" layer
normalizer.adapt(X_train) # "Fit" it on the train set

# Once fitted, you can use it as a function

print(normalizer(X_train).numpy().std())
print(normalizer(X_test).numpy().std())
```

```
1.0
0.9552177
```

```
model = Sequential()

# Use the adapted normalizer as first sequential step
model.add(normalizer)
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(8, activation='relu'))
model.add(layers.Dense(3, activation='linear'))

model.compile(loss='mse', optimizer='adam', metrics='mae')

history = model.fit(
    X_train, y_train,
```

```
    validation_data=(X_test, y_test),
    epochs=100, batch_size=32, verbose=0)

plot_history(history)
```



👍 Use preprocessing layers whenever they exist. It greatly helps with data-engineering:

- Normalization
- CategoryEncoding (OneHotEncoder)
- TextVectorization (NLP...)
- Resizing (Images...)

Full list of preprocessing layers 📚 here

📚 Pipeline example from A to Z

One last thing: Save and load models

❗ ***Warning*** ❗ This section will be extremely important for the Data Science Projects.

There are many reasons to save your trained model. You can for exmaple send it to someone that can then load and use it without having to train it.

```
from tensorflow.keras import models

# Let's say that you have a `model`
# You can save it:
models.save_model(model, 'my_model.keras')

# and you can load it somewhere else:
```

```
loaded_model = models.load_model('my_model.keras')
```

⚠️ Using `tensorflow<2.16` and/or `keras<3`? Drop the `.keras` in the filename

⚠️ Models saved in those versions are not compatible with the `.keras` format of Keras 3

Bibliography

📚 In what sense is backprop a fast algorithm

📺 3Blue1Brown - Neural Networks - 1h

📺 Stanford - Graphs, and backpropagation - 1h20

Your turn 🚀