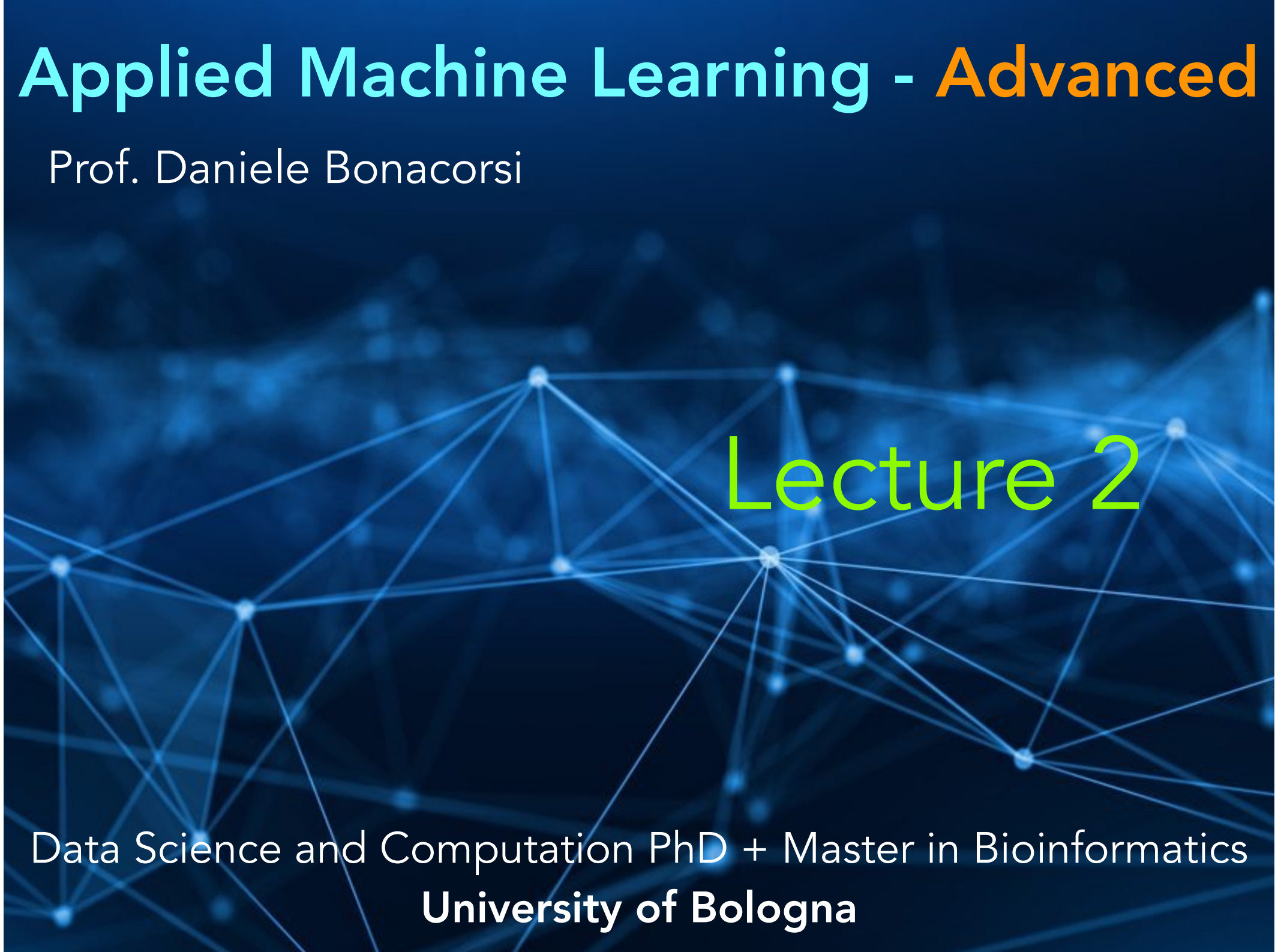
Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi



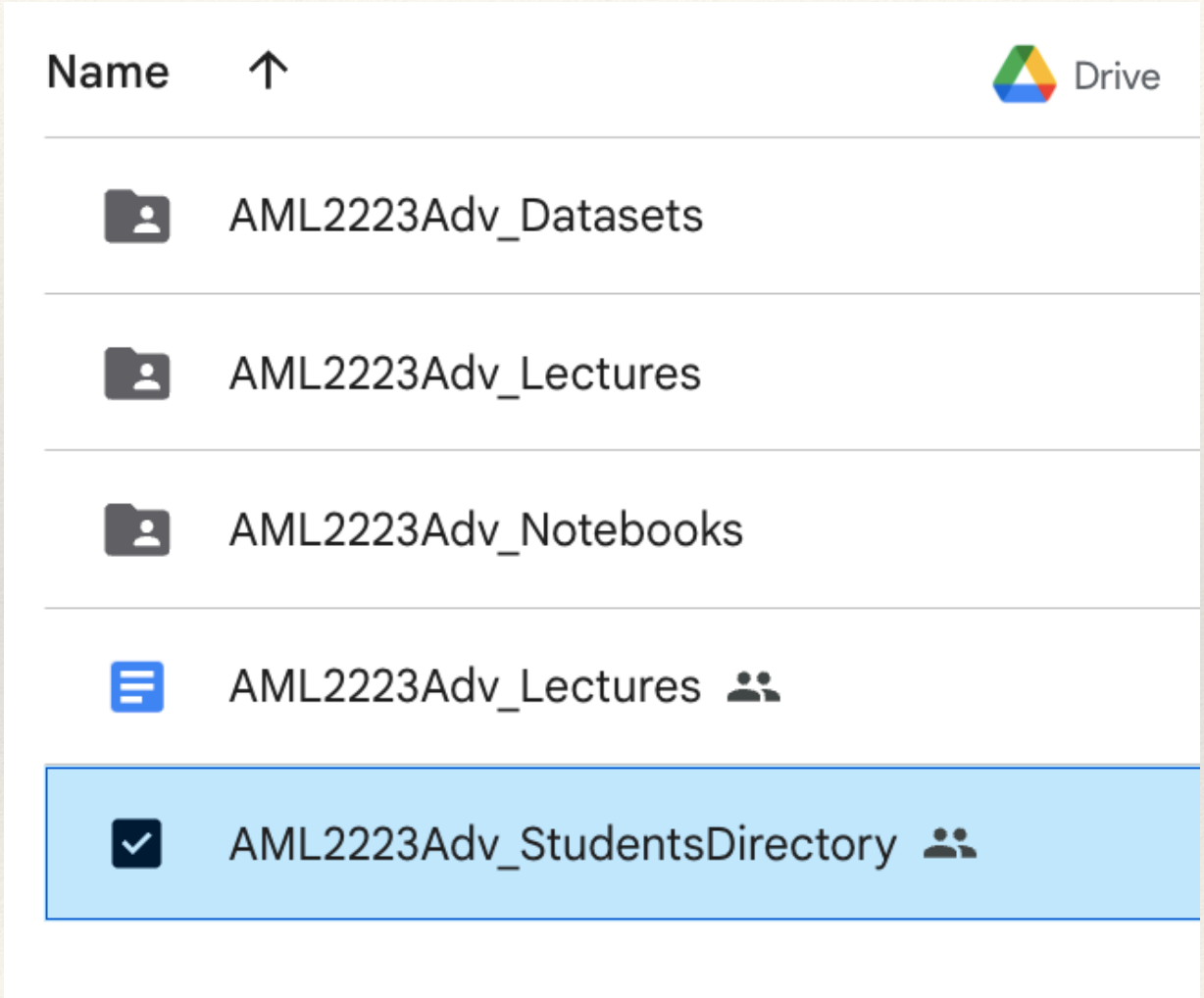
Lecture 2

Data Science and Computation PhD + Master in Bioinformatics

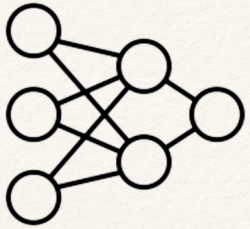
University of Bologna

Communications

Refresh and check the Adv course material



If you have NOT done it yet, please fill in your details here!

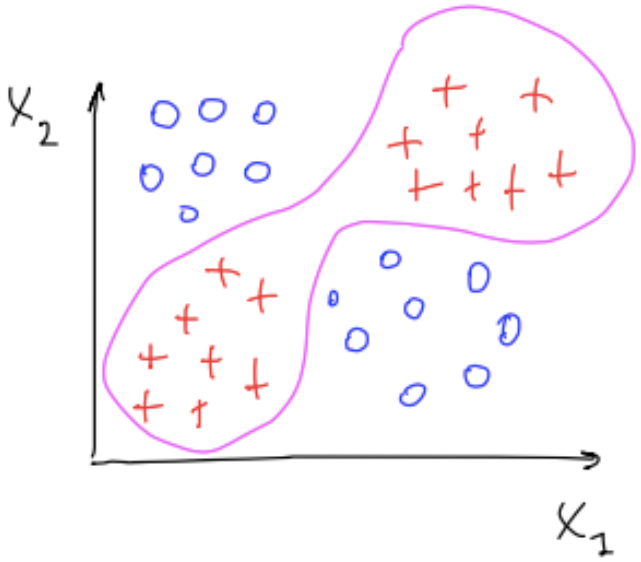


Build a NN for XNOR

(aka: complexity goes in the hidden layers)

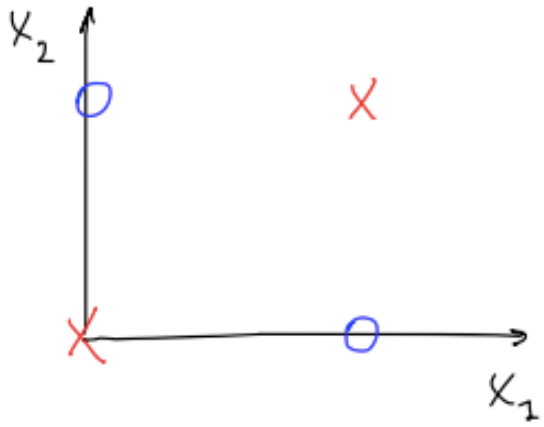
Let's see a detailed example of how a NN can compute a complex non-linear function of the input.

Non - linear classification



example:

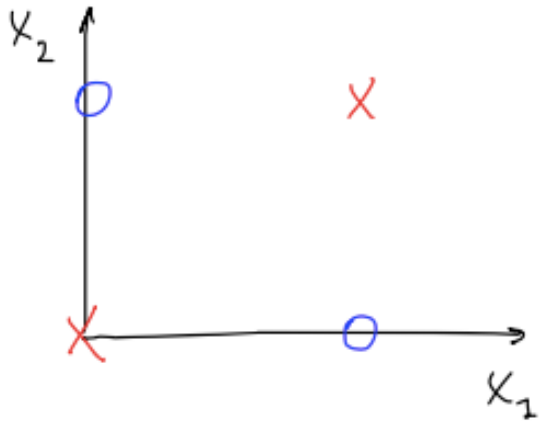
You want to separate with a non linear decision boundary the positive and the negative classes.



A simplification:

$$X_1, X_2 \in \{0, 1\}$$

only binary values

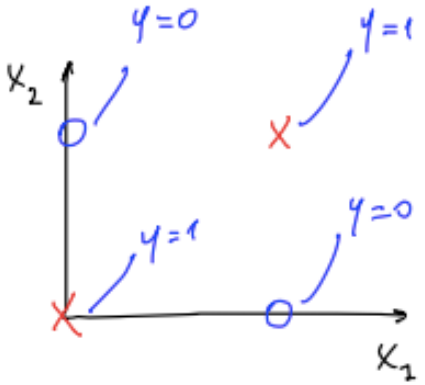


A simplification:

$$X_1, X_2 \in \{0, 1\}$$

only binary values

X_1	X_2	X_1 XOR X_2	X_1 XNOR X_2
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1



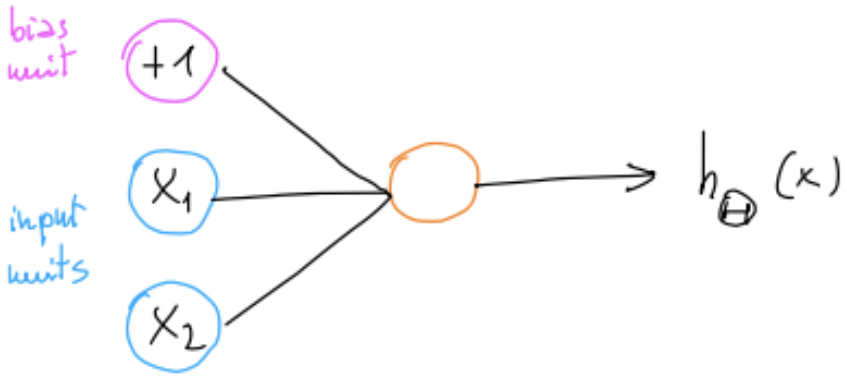
Can we build an ANN that fit this training set? (XNOR example

→ let's start with a simpler one ... AND.

Example : logical AND

$x_1, x_2 \in \{0, 1\} \longrightarrow y = x_1 \text{ AND } x_2$

input — target label

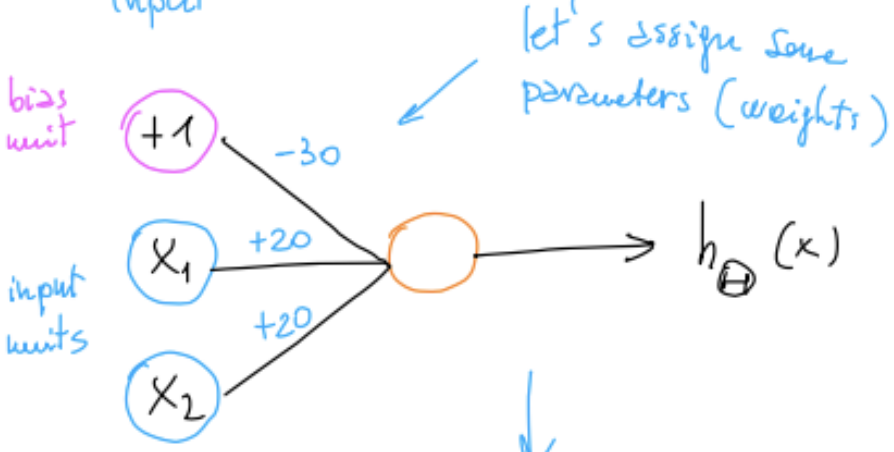


Example : logical AND

$x_1, x_2 \in \{0, 1\}$ ⟶ $y = x_1$ AND x_2

x_1, x_2: input

y: target label



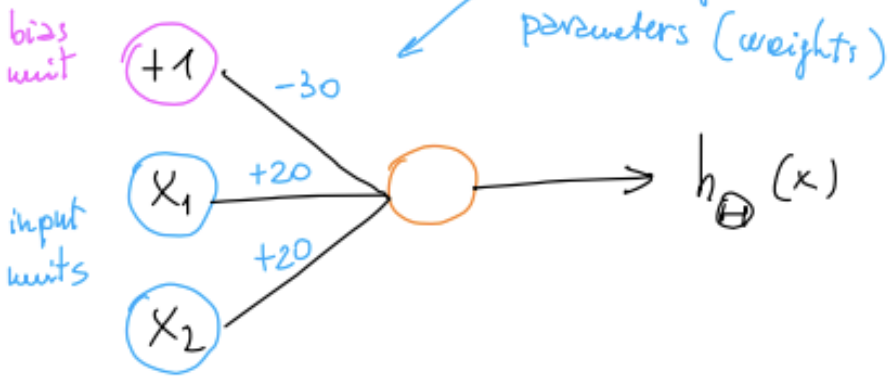
$$\Rightarrow h_\Theta(x) = g(\underbrace{-30}_{\Theta_{10}^{(1)}} + \underbrace{20}_{\Theta_{11}^{(1)}} x_1 + \underbrace{20}_{\Theta_{12}^{(1)}} x_2)$$

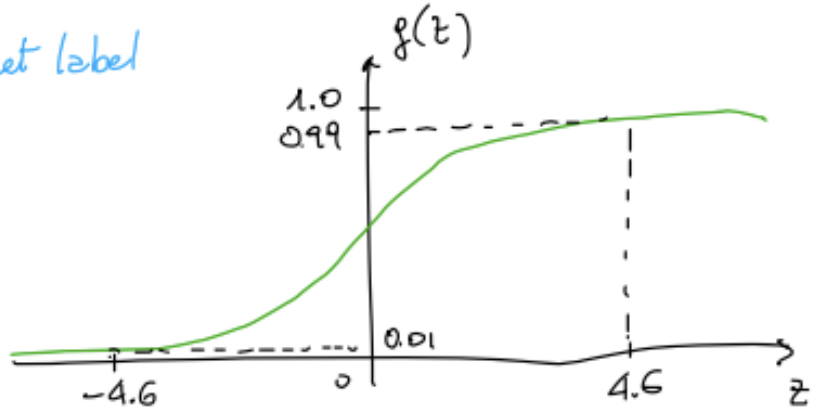
Example: logical AND

$x_1, x_2 \in \{0, 1\} \longrightarrow y = x_1 \text{ AND } x_2$

input — target label

let's assign some parameters (weights)





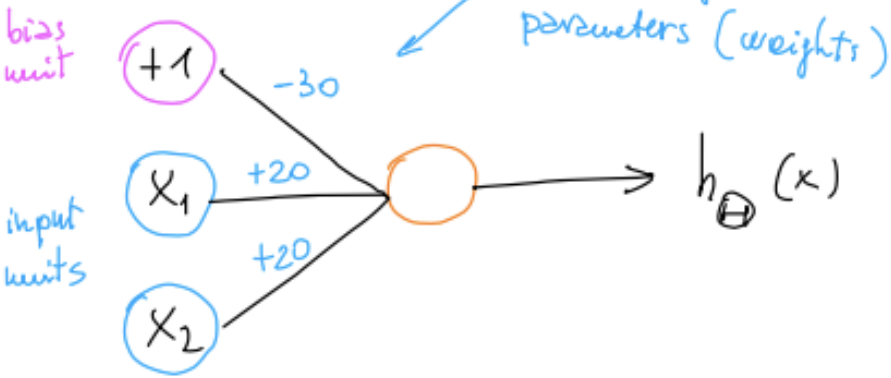
$$\Rightarrow h_\Theta(x) = g(\underbrace{-30}_{\Theta_{10}^{(1)}} + \underbrace{20}_{\Theta_{11}^{(1)}} x_1 + \underbrace{20}_{\Theta_{12}^{(1)}} x_2)$$

x_1	x_2	x_1 AND x_2
0	0	
0	1	
1	0	
1	1	

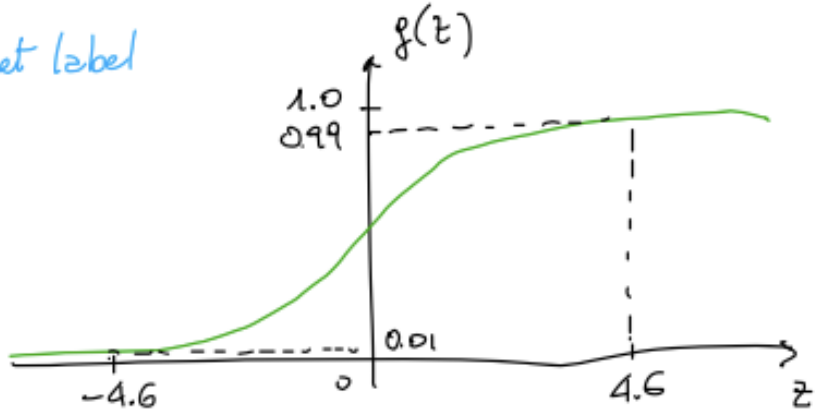
Example: logical AND

$x_1, x_2 \in \{0, 1\} \longrightarrow y = x_1 \text{ AND } x_2$

input — target label



let's assign some parameters (weights)



$$\Rightarrow h_\Theta(x) = g(\underbrace{-30}_{\Theta_{10}^{(1)}} + \underbrace{20}_{\Theta_{11}^{(1)}} x_1 + \underbrace{20}_{\Theta_{12}^{(1)}} x_2)$$

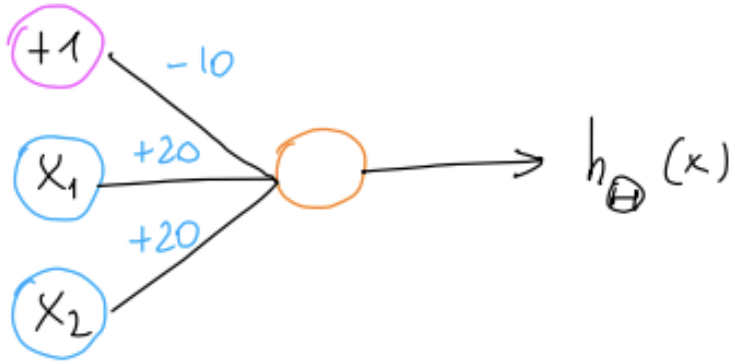
x_1	x_2	x_1 AND x_2
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

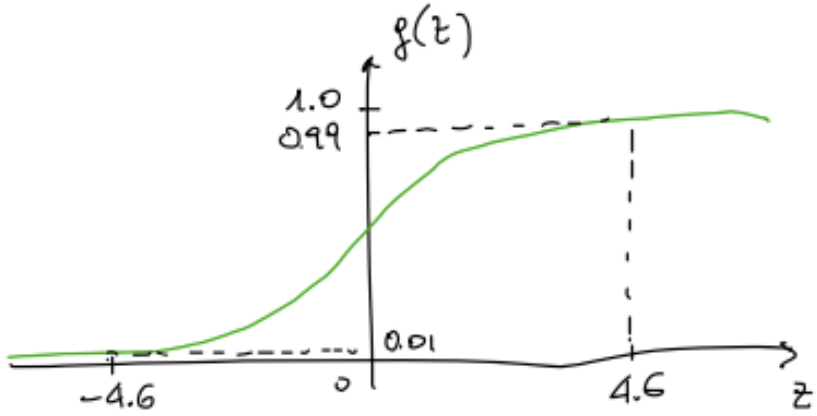
with a truth table you see the logical function our ANN computes

$h_\Theta(x) \approx x_1 \text{ AND } x_2$

Example : logical OR

$x_1, x_2 \in \{0, 1\} \longrightarrow y = x_1 \text{ OR } x_2$





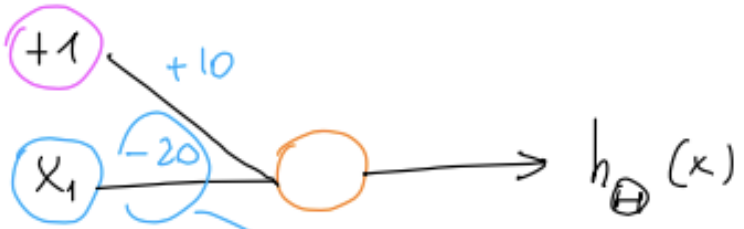
$\Rightarrow h_\Theta(x) = g(-10 + 20x_1 + 20x_2)$

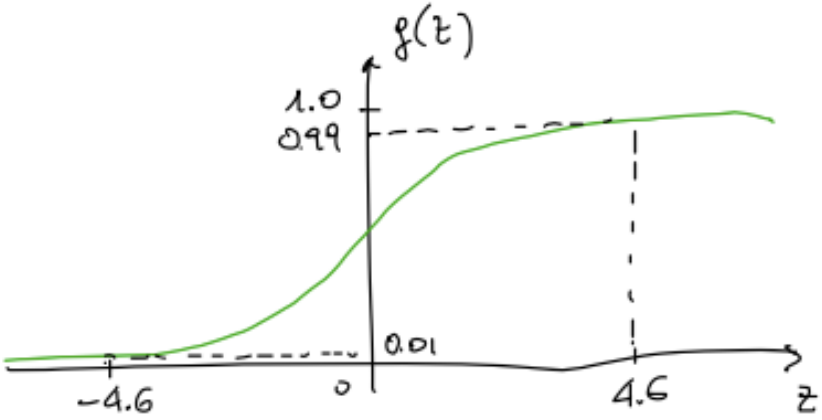
x_1	x_2	x_1 OR x_2
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	$g(10) \approx 1$
1	1	$g(30) \approx 1$

$h_\Theta(x) \approx x_1 \text{ OR } x_2$

Example : negation

$x_1 \in \{0, 1\} \longrightarrow y = \text{NOT } x_1$





only 1 input feature

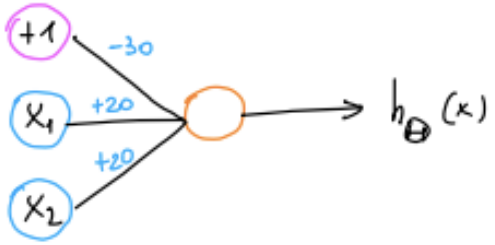
$\Rightarrow h_\Theta(x) = g(+10 - 20\, x_1)$

x_1	NOT x_1
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

put a large negative weight in front of the variable you want to negate

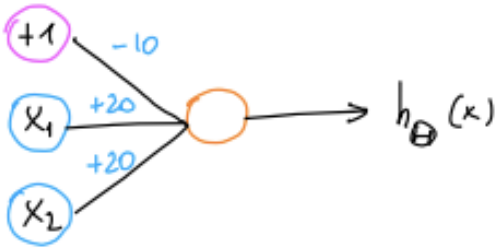
$h_\Theta(x) \approx \text{NOT } x_1$

AND



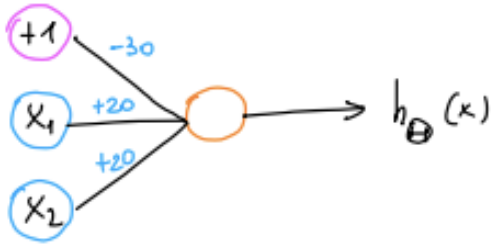
Look carefully ...

OR

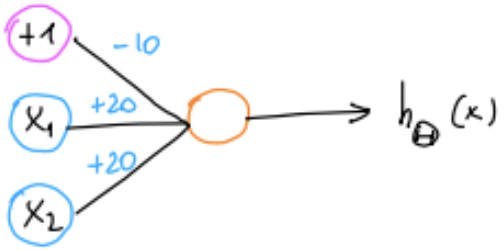


NOT

x_1 AND x_2

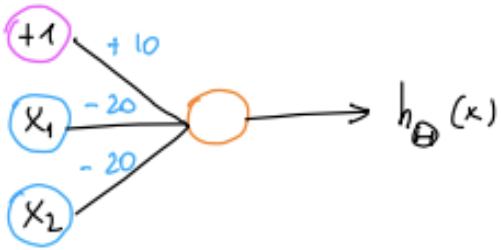


x_1 OR x_2

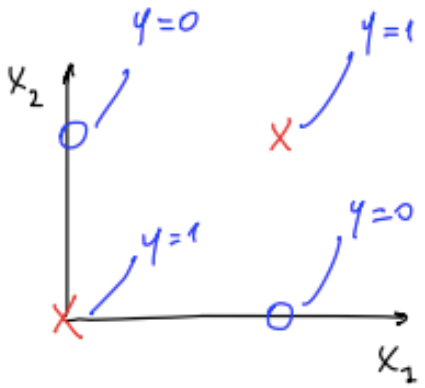


NOT x_1

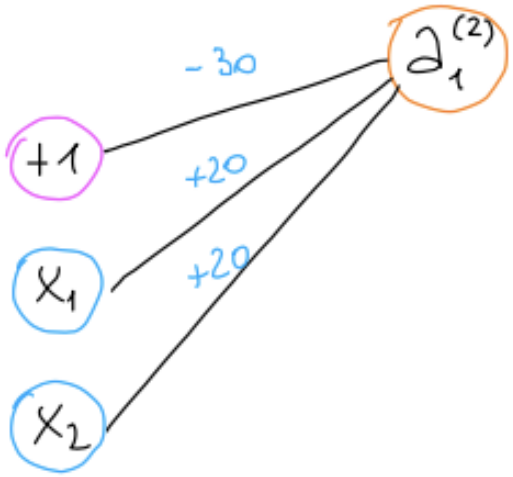
(NOT x_1) AND (NOT x_2)



XNOR

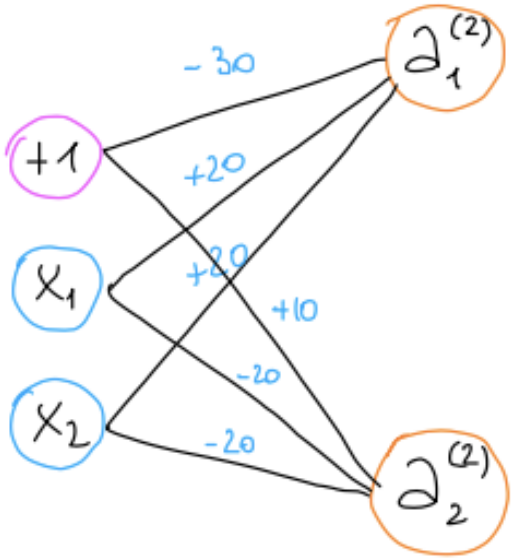


we need a non-linear decision boundary in order to separate positive and negative examples



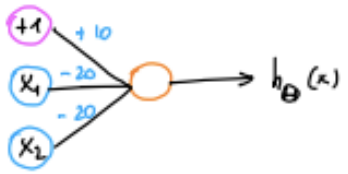
I create the first unit and copy the weights over → the weights of

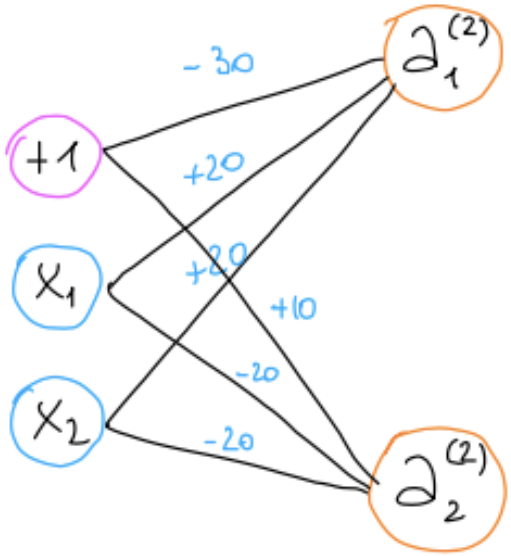
x_1 AND x_2



I create the second hidden unit and copy again the weights → the weights of

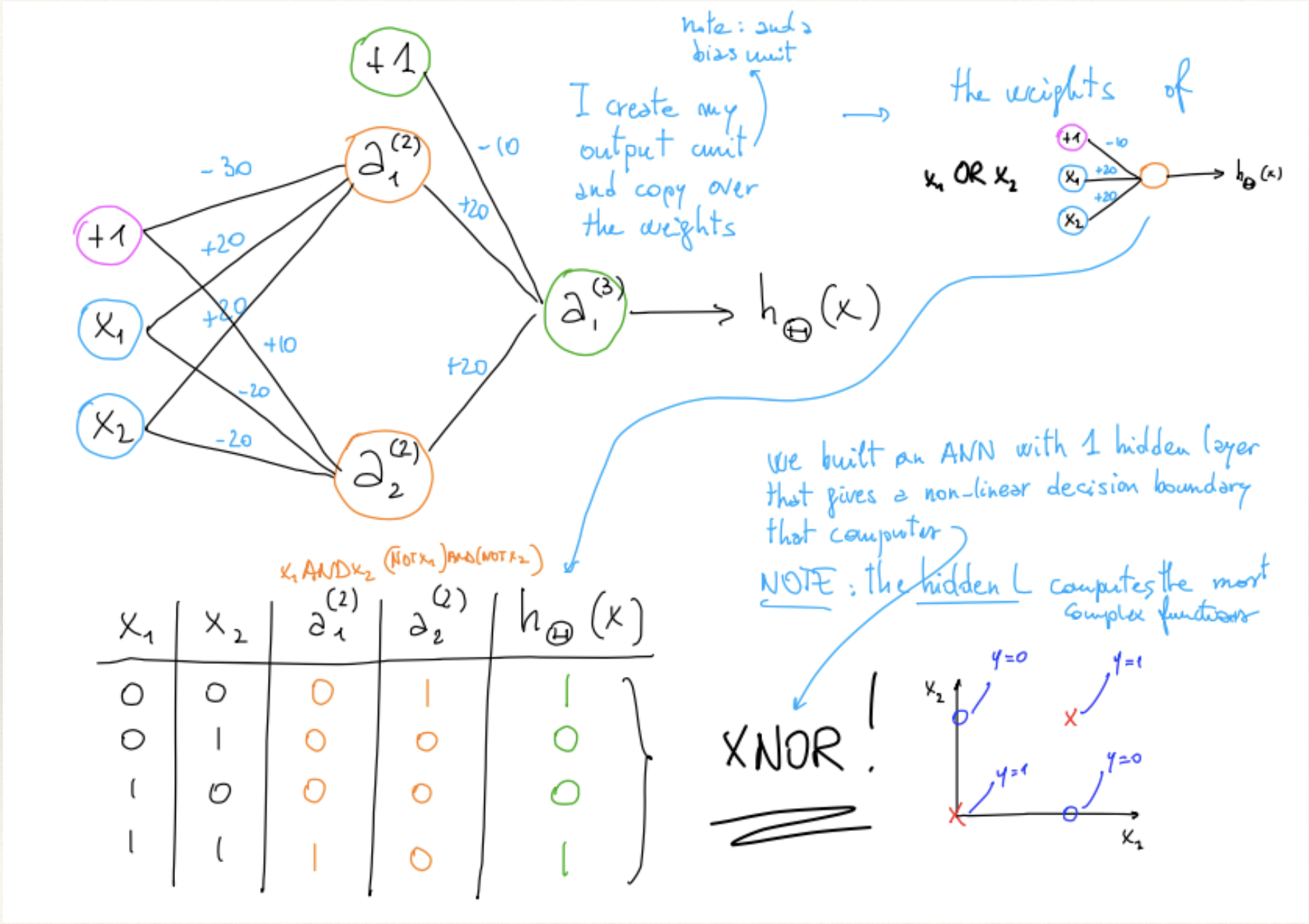
(NOT x_1) AND (NOT x_2)





x_1 AND x_2 (NOT x_1) AND (NOT x_2)

x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\Theta(x)$
0	0	0	1	
0	1	0	0	
1	0	0	0	
1	1	1	0	


+1
x_1
x_2
$a_1^{(2)}$
$a_2^{(2)}$
$a_1^{(3)}$
$h_\Theta(x)$
-30
+20
+20
+10
-20
-20
-10
+20
+20
note: add a bias unit
I create my output unit and copy over the weights
the weights of
x_1 OR x_2
x_1 AND x_2 (NOT x_1) AND (NOT x_2)
x_1 | x_2 | $a_1^{(2)}$ | $a_2^{(2)}$ | $h_\Theta(x)$
0 | 0 | 0 | 1 | 1
0 | 1 | 0 | 0 | 0
1 | 0 | 0 | 0 | 0
1 | 1 | 1 | 0 | 1
we built an ANN with 1 hidden layer that gives a non-linear decision boundary that computes
NOTE: the hidden L computes the most complex functions
XNOR !
y=0
y=1
y=1
y=0
x_2
x_1

Summary

We discussed how single units/neurons in a NN can be used together to build logical functions like the XNOR function

- XNOR not so complicated! but still worse than AND or OR..

We have constructed some of the fundamental operations in computers by using small NNs rather than using actual logical gates. The way simpler components combines to provide solution to more complex questions is evident!

Most important take-away message:

The most complicated part is dealt with by the hidden layer(s)

- **and I can build as many as I want..** (provided I have computing power)
- **.. so with ANNs I can deal with tasks as complicated as necessary!**

Before proceeding..

Let's familiarise with ***our (next!) favourite ML benchmark dataset****.. and profit of this step to recap* ***sklearn****, too..*

Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi

Hands-on: MNIST1

Data Science and Computation PhD + Master in Bioinformatics
University of Bologna

Hands-on: MNIST1

Classification with sklearn

[credits to: A. Geron, "Hands-On Machine Learning With Scikit-Learn and Tensorflow"]

Coding material

AML2223Adv_HandsOn_MNIST1.ipynb

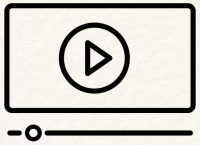
→ go to the "**MNIST1**" notebook

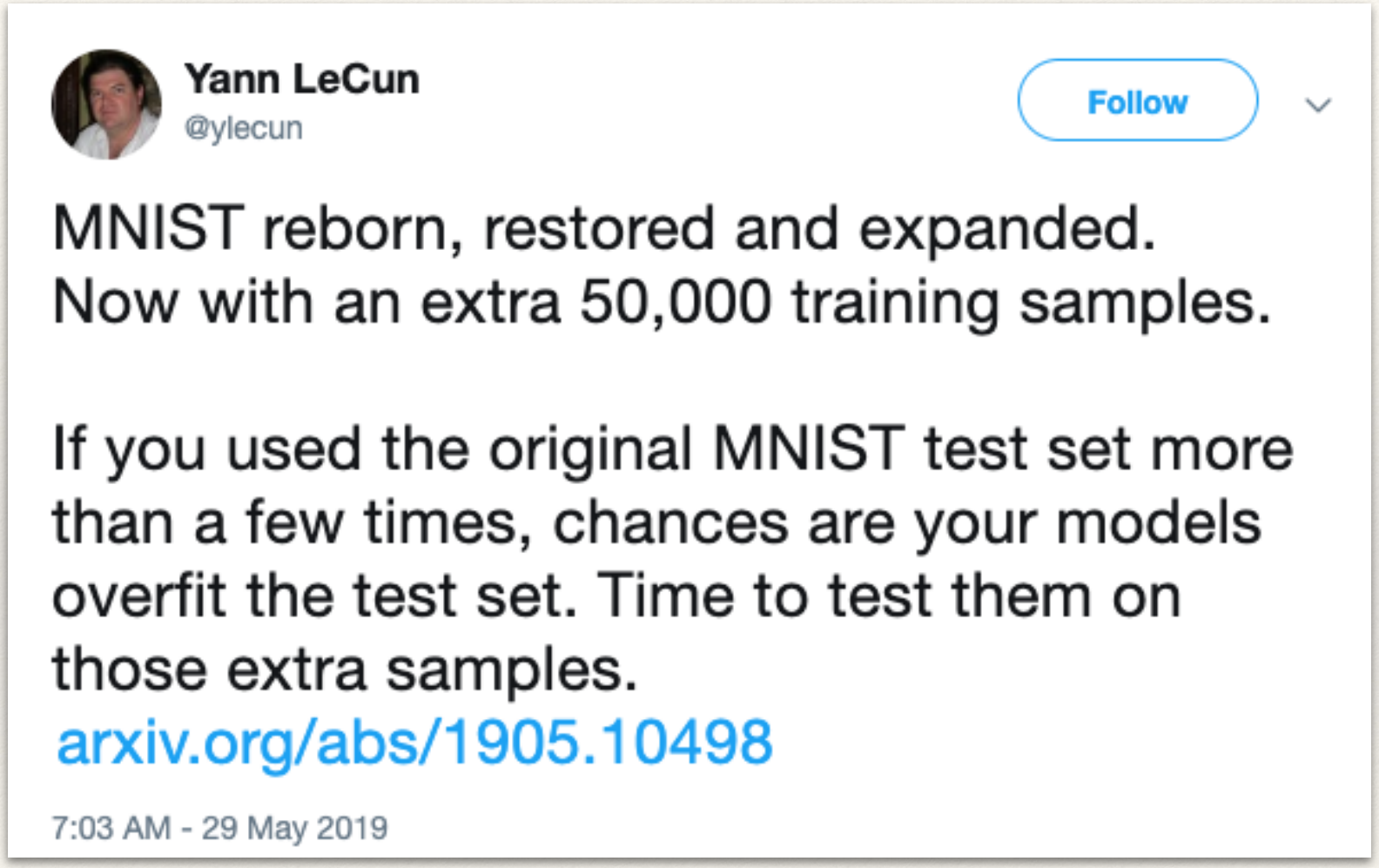
LOOK FOR THIS ARROW IN THE SLIDES
(this means there is code for you to play with the concepts!)

MNIST

The **MNIST** dataset is a set of 70k images of handwritten digits

- Each image is **labeled** with the digit it represents (i.e. like "this is a 3")
- 784 **features**: 28x28 pixels each, each features represent one pixel's intensity, from 0 (white) to 255 (black).
- one of the most famous "hello world" in ML → **multi-class classification**

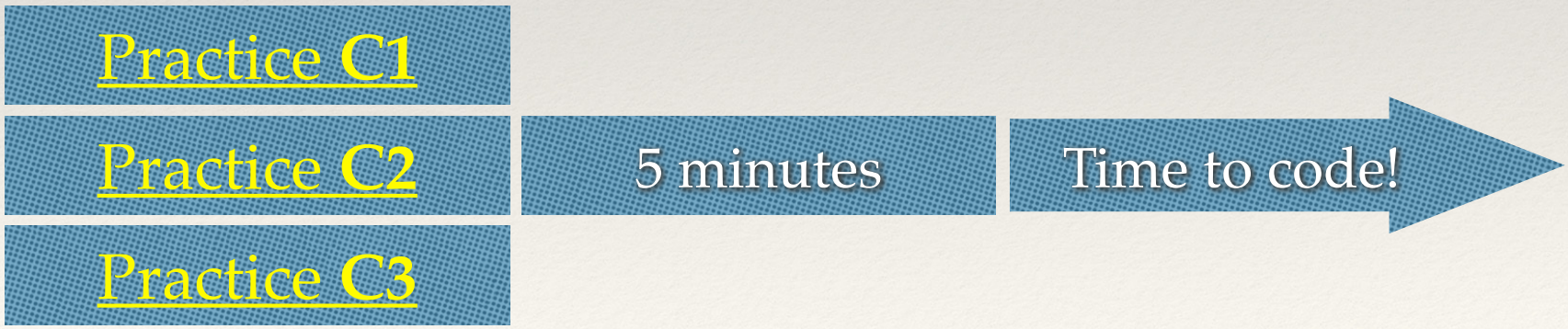



Yann LeCun
@ylecun
Follow
MNIST reborn, restored and expanded.
Now with an extra 50,000 training samples.
If you used the original MNIST test set more than a few times, chances are your models overfit the test set. Time to test them on those extra samples.
arxiv.org/abs/1905.10498
7:03 AM - 29 May 2019

Let’s start..

Hands-on in progress

Set up, import the data, inspect (briefly) the data, perform the train-test split.



Get the data

Get it from sklearn:

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details', 'categories', 'url'])
```

```
X, y = mnist["data"], mnist["target"]
X.shape
```

```
(70000, 784)
```

A **data** key containing an array with one row per instance and one column per feature

A **target** key containing an array with the labels

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

The dataset is **split into training + test**..

- 60k training, 10k test

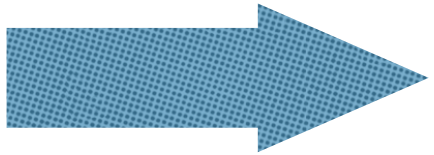
.. and it is already **shuffled**, so all CV folds will be similar

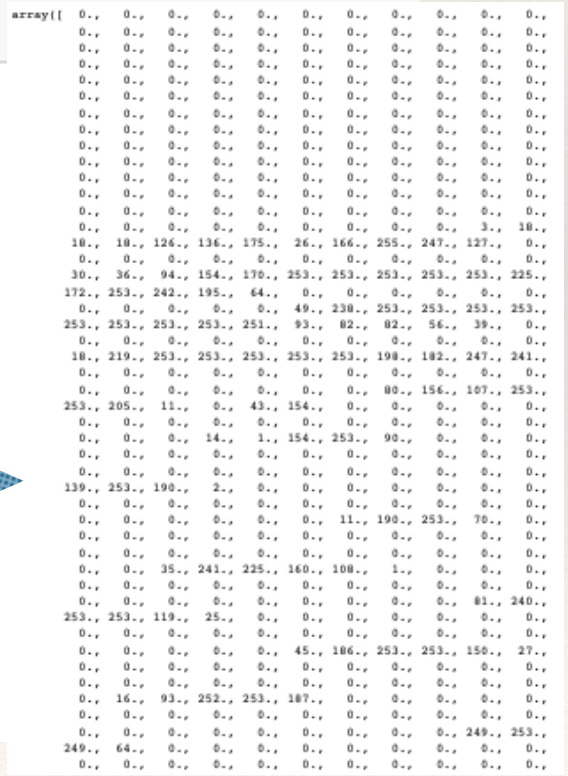
- you don't want one fold to be missing some digits

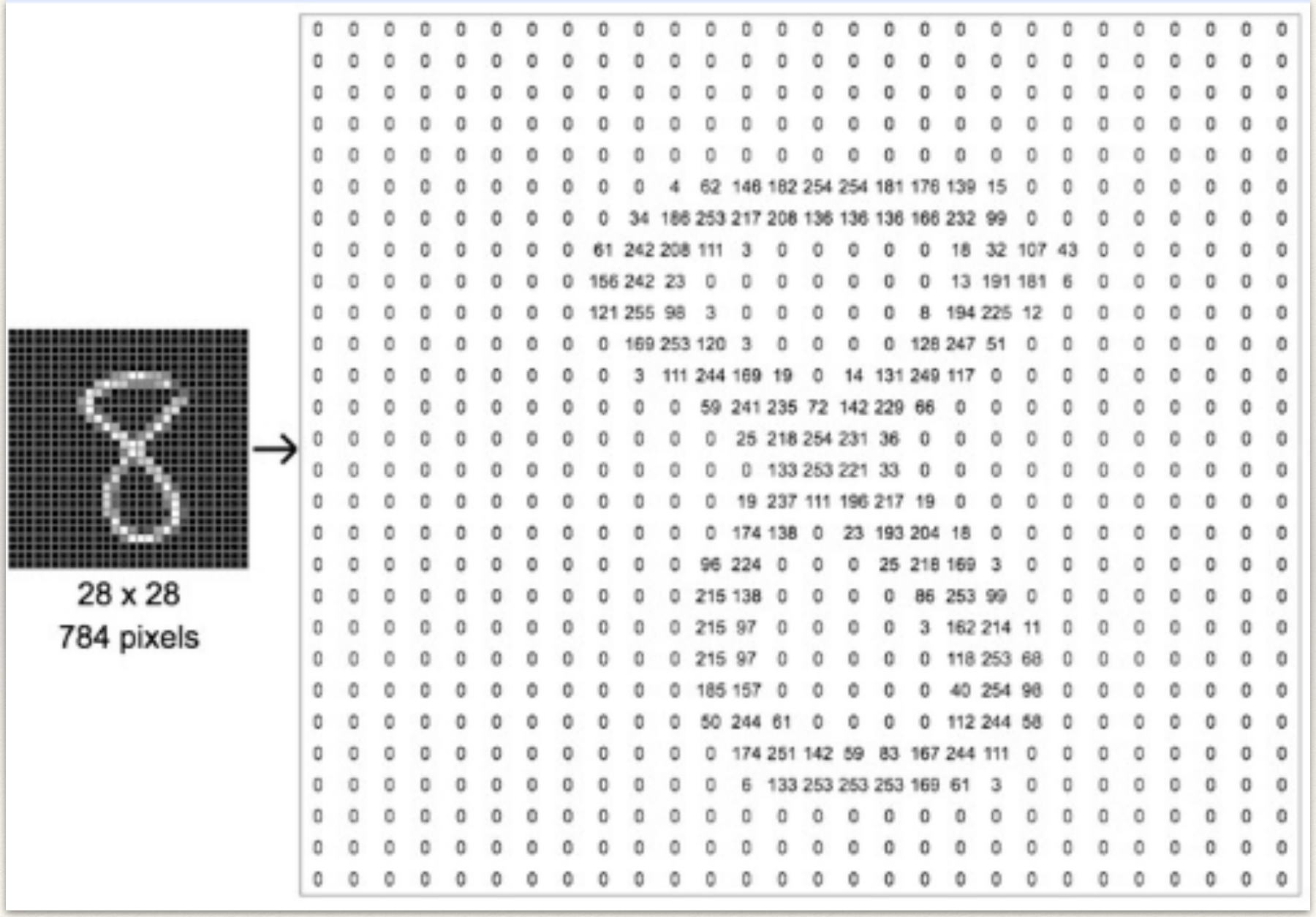
Inspect the data

```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap = mpl.cm.binary, interpolation="nearest")
plt.axis("off")

save_fig("some_digit_plot")
plt.show()
```

```
Saving figure some_digit_plot
```






28 x 28
784 pixels

Ready for next?

Hands-on in progress

Implement a 5-detector.

Train a **binary** classifier

Simplify and build a model that works e.g. as a "**5-detector**"

- capable of distinguishing between just two classes, "5" and "not-5"

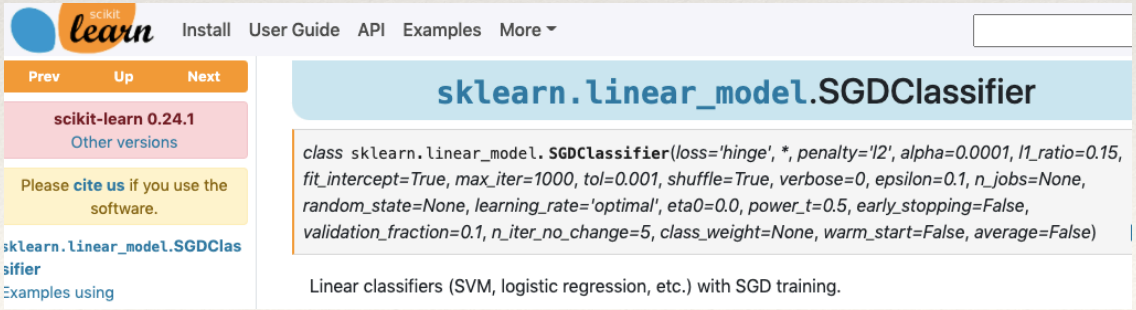
Create the label vectors (train and test sets) for this task:

```
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
```

Then, pick a classifier. An interesting choice is the **SGD classifier**

- capable of handling very large datasets efficiently (it deals with training instances independently, one at a time - which also makes SGD well suited for online learning)

Train and predict is easy..



Train a **binary** classifier

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

I know that X[0] is a 5, X[1] is a 0, X[2] is a 4:

```
print "y[0] =", y[0]
print "y[1] =", y[1]
print "y[2] =", y[2]
```

```
y[0] = 5
y[1] = 0
y[2] = 4
```

Let's check if the classifier we built above works for these 3 examples:

```
[31] sgd_clf.predict([X[0]]) # X[0] is a 5

     array([ True])
```

```
[32] sgd_clf.predict([X[1]]) # X[1] is a 0, so NOT a 5

     array([False])
```

```
[33] sgd_clf.predict([X[2]]) # X[2] is a 4, so NOT a 5

     array([False])
```

OK, it seems to work.. which is the performance of this model?

Ready for next?

Hands-on in progress

Compute the accuracy

- hint: use cross_val_score() function in sklearn to evaluate your SGDClassifier model using k-fold cross-validation, with k=3

Measuring performance (accuracy) using CV

Use cross_val_score() function in sklearn to evaluate your SGDClassifier model using k-fold cross-validation, with k=3

- i.e. make k trainings: split the training set into k folds, train and make predictions and evaluate them on each fold using a model trained on the remaining folds

```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
array([0.96355, 0.93795, 0.95615])
```

What!? 93-96% accuracy at first attempt!? Mmh..

- think at a very dumb classifier that just classifies every single image as if it belonged to the "not-5" class: it will have 90% accuracy! (if enough data, only about 10% of the images are 5s, so if you always guess that an image is a "not-5", you will be right roughly 90% of the time, by construction!

Accuracy is <u>not</u> the preferred performance measure for classifiers

- even worse if you are dealing with **skewed datasets** (i.e. when some classes are much more frequent than others).

Ready for next?

Hands-on in progress

Extract the **confusion matrix**.

Practice C6 | 2 minutes | Time to code!

Confusion matrix

Remember from AMLBas?

To evaluate the performance of a classifier, build the **confusion matrix**

- count misclassifications: e.g. how many times the classifier **confused** images of 5s with 3s? look in the 5th row and 3rd column of the **confusion** matrix

Use cross_val_predict() and confusion_matrix()

- cross_val_predict() is similar to cross_val_score(): performs K-fold CV but returns not the evaluation score, but the predictions made on each fold
- then, give the target classes (y_train_5) and the predicted classes (y_train_pred) to confusion_matrix()

```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

To clarify, perfection will look like this:

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_5, y_pred)
```

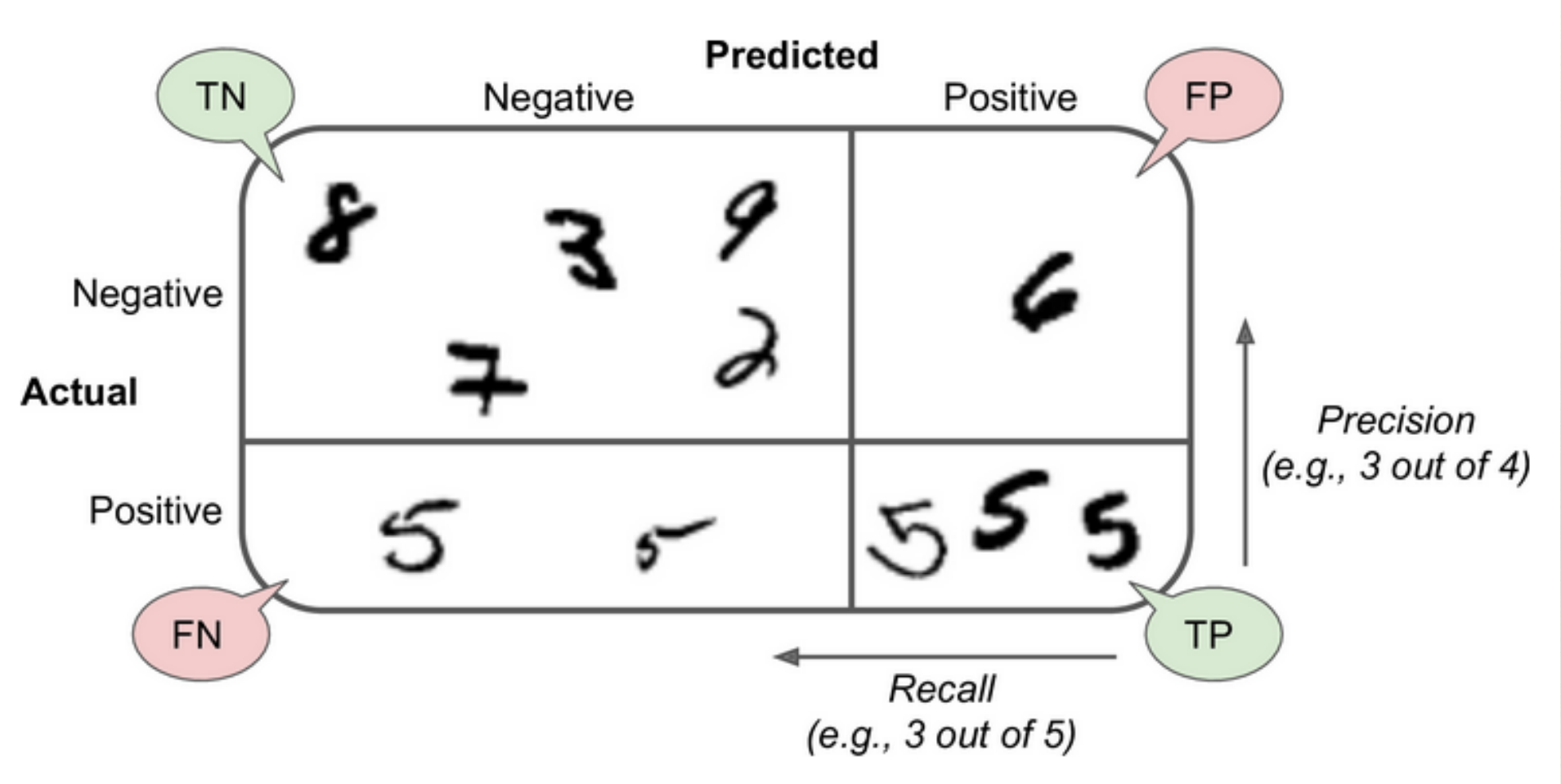
```
array([[62736,   951],
       [ 1516,  4797]])
```

```
y_train_perfect_predictions = y_train_5  # pretend we reached perfection
confusion_matrix(y_train_5, y_train_perfect_predictions)
```

```
array([[54579,     0],
       [    0,  5421]])
```

Confusion matrix

Remember from AMLBas?



Precision, Recall, F1 score

Remember from AMLBas?

Abandon accuracy, and compute **precision** and **recall**:

```
from sklearn.metrics import precision_score, recall_score

prec = precision_score(y_train_5, y_train_pred)
reca = recall_score(y_train_5, y_train_pred)
print("precision", prec)
print("recall", reca)
```

```
precision 0.7290850836596654
recall 0.7555801512636044
```

My 5-detector does not look as shiny as it did when I looked at its accuracy only…

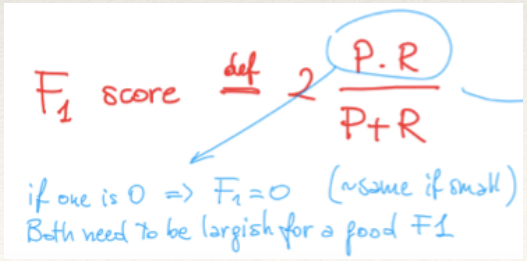
- when it claims an image represents a 5, it is correct only 72.9% of the time
- and it detects only 75.6% of the 5s

Convenient to combine them into a single metric: the **F1 score**

- **harmonic mean** of precision and recall: wrt regular mean, the harmonic mean does not treat all values equally, but gives much more weight to low values. As a result, the classifier will only get a **high F1 score if both recall and precision are high**
- additionally, good to have just one performance metric (if I need to compare 2 classifiers)

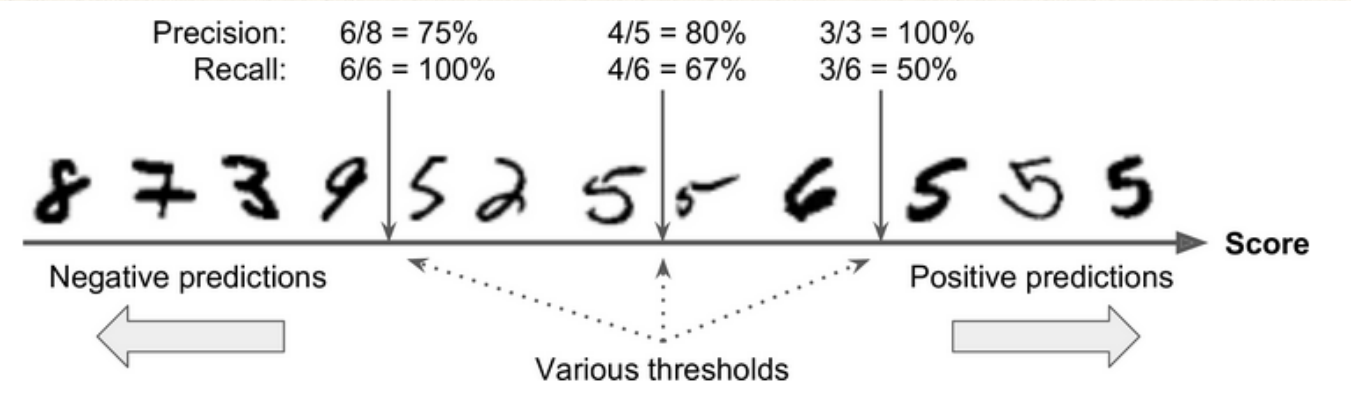
```
from sklearn.metrics import f1_score

f1_score(y_train_5, y_train_pred)
```

```
0.7420962043663375
```



Precision/Recall trade-off

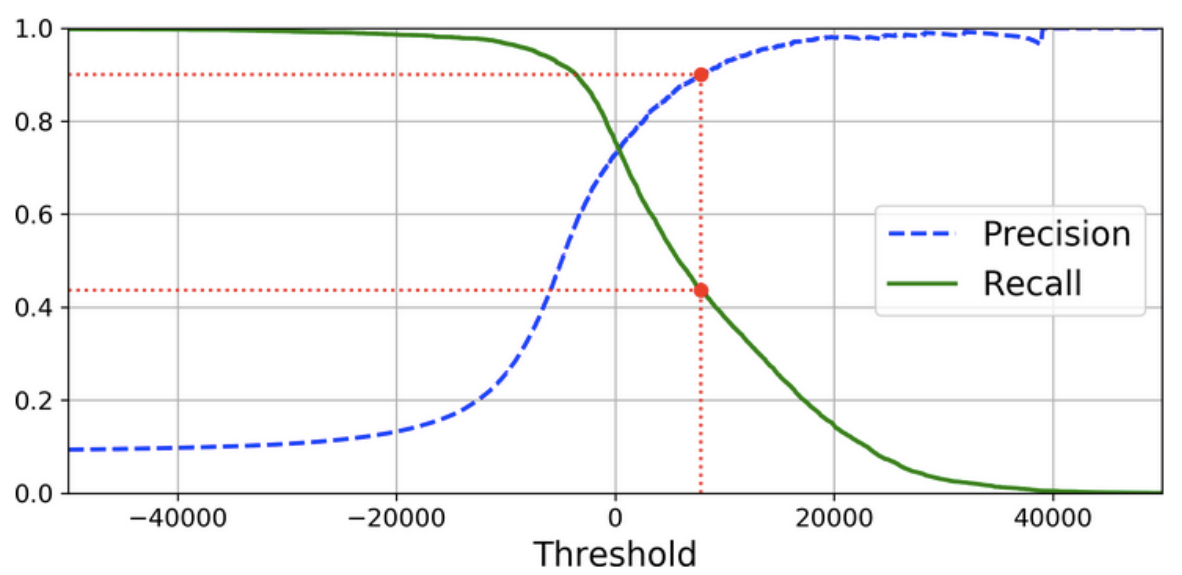
Remember from AMLBas?



SGDClassifier, for each instance, computes a score based on a decision function, and if that score is greater/smaller than a threshold, it assigns the instance to the positive/negative class

Looking at various thresholds, it is evident that when precision increases then recall reduces, and vice versa. This is called the **precision/recall tradeoff**

"How do I choose the threshold?".



Receiver Operating Characteristic (ROC)

Remember from AMLBas?

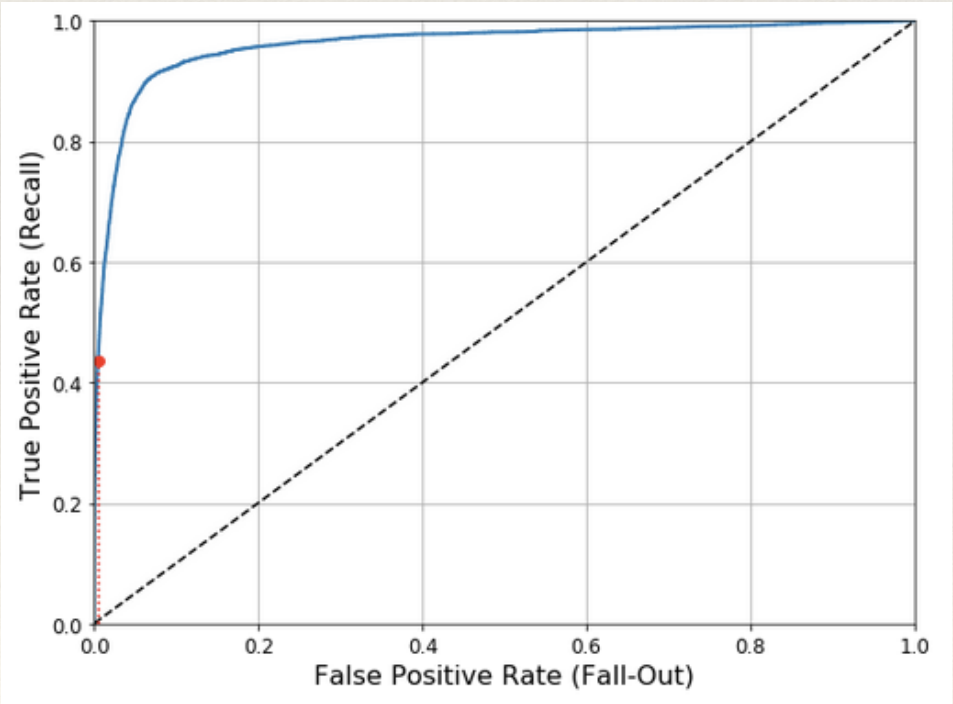
The **Receiver Operating Characteristic** (**ROC**) curve is another very common tool used with binary classifier

It is very similar to the precision/recall curve, but:

- it plots the **TPR** (= recall) against the **FPR** (FPR = ratio of negative instances that are incorrectly classified as positive), which is FPR=1-**TNR** (TNR = ratio of negative instances that are correctly classified as negative - also called **specificity**). In other words, the ROC curve plots sensitivity (recall) versus 1 – specificity.

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

.. and then plot:

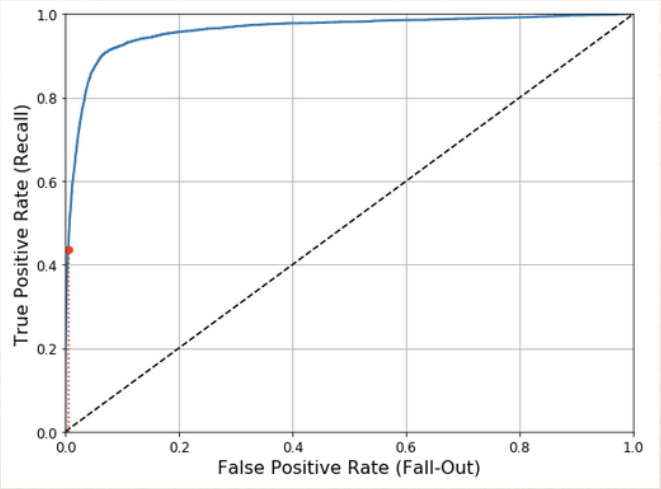


Area Under the Curve (AUC)

Remember from AMLBas?

Observations on the ROC:

- the higher (lower) the TPR, the more (fewer) false positives FPR the classifier produces
- the dotted line represents the ROC curve of a purely random classifier
- a good classifier stays as far away from that line as possible, toward the top-left corner



To compare classifiers you need a number: this could be then the **Area Under the** ROC **Curve** (AUC)

- a perfect classifier will have AUC = 1
- a purely random classifier will have AUC = 0.5.

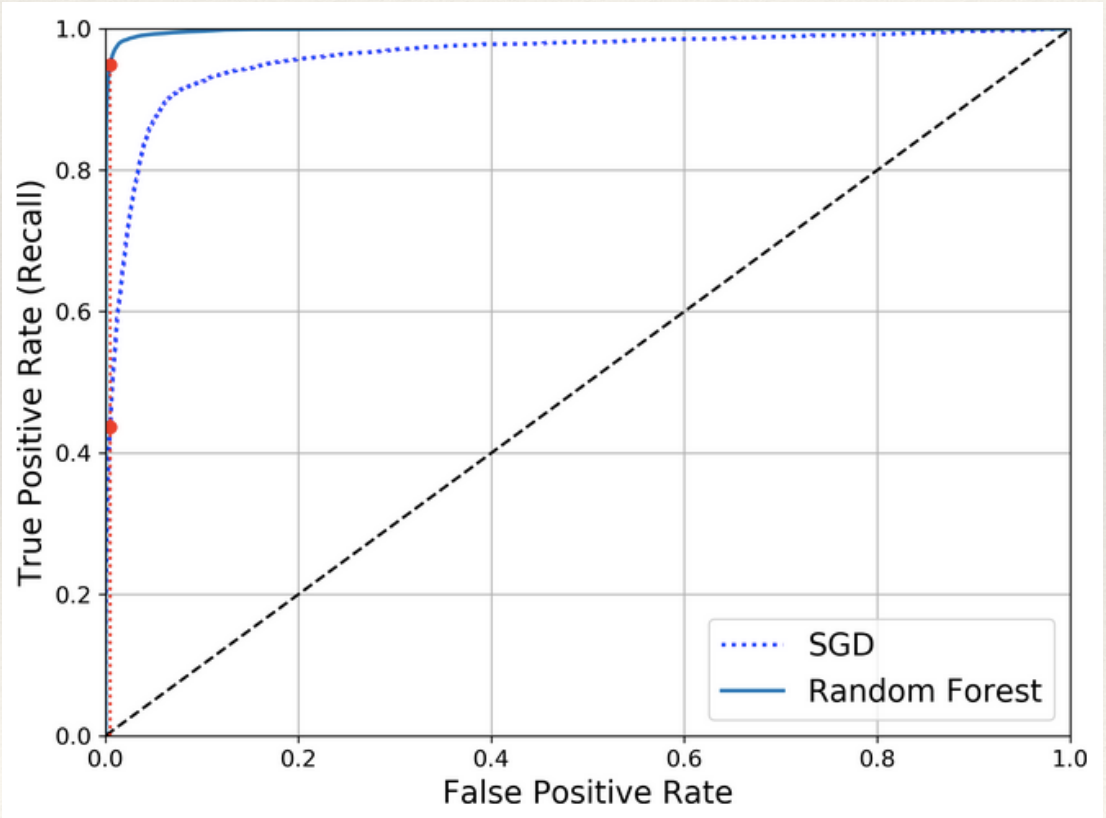
```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

```
0.9611778893101814
```

Model comparison using AUC

Use ROC+AUC as performance metrics. Get them for all models, and you can compare them.

- e.g. (not in the notebook) if one trains a **RandomForestClassifier** and compare its ROC curve and ROC AUC score to the **SGDClassifier**



RandomForestClassifier's ROC curve looks much better than the SGDClassifier's. AUC scores also show this (below)

SGDClassifier

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

```
0.9611778893101814
```

RandomForestClassifier

```
roc_auc_score(y_train_5, y_scores_forest)
```

```
0.9983436731328145
```

MNIST recap so far

Now you recapped a bit how to:

- train a **binary classifier**
- choose the appropriate metric for your task
- evaluate your classifiers using CV
- select the precision/recall tradeoff that fits your needs, and compare various models using ROC curves and ROC AUC scores

One can continue and try to detect more than just the 5s…

— bonus material —

From Binary to **Multi-class** classifiers

A binary classifier distinguishes between two classes.

A **multi-class classifier** (aka "multinomial classifier") can distinguish between >2 classes.

Train a **multi-class** classifier

Which classifier can I use?

- some algos (such as SGD classifiers, Random Forest classifiers, naive Bayes classifiers) are capable of handling multiple classes directly
- others (such as Logistic Regression or Support Vector Machine classifiers) are strictly binary classifiers

However, there are various strategies that you can use to **perform multiclass classification using multiple binary classifiers**

- e.g. **one-versus-one** (**OvO**): train a binary classifier for every pair of digits (one to distinguish 0s and 1s, another to distinguish 0s and 2s, ..)
  - for N classes, you train N×(N–1)/2 classifiers → for MNIST, 45 binary classifiers! When you want to classify an image, you have to run the image through all 45 classifiers and see which class wins the most duels.
  - advantage: each classifier needs to be trained only on the part of the training set for the two classes that it must distinguish. E.g. Some algos (such as SVM classifiers) scale poorly with the size of the training set, OvO is preferred (it is faster to train many classifiers on small training sets than training few classifiers on large training sets)
- e.g. **one-versus-the-rest** (**OvR**) or **one-versus-all** (**OvA**): train 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on)
  - when you want to classify an image, you get the decision score from each classifier for that image and select the class whose classifier outputs the highest score
  - see comment above for SVM: for most binary classification algorithms, instead, OvR is preferred

Should I implement one of these 2 strategies into my code myself? Fortunately not.

- Scikit-Learn detects when you try to use a binary classification also for a multiclass classification task, and it automatically runs OvR or OvO

Try this with a Support Vector Machine classifier, using the sklearn.svm.SVC class:

```
>>> from sklearn.svm import SVC
>>> svm_clf = SVC()
>>> svm_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> svm_clf.predict([some_digit])
array([5], dtype=uint8)
```

- you train the SVC on the training set using the original target classes from 0 to 9 (y_train), instead of the 5-versus-the-rest target classes (y_train_5).
- then, it makes a prediction (a correct one in this case).

Under the hood, Scikit-Learn actually used the OvO strategy: it trained 45 binary classifiers, got their decision scores for the image, and selected the class that won the most duels.

If you call the decision_function() method, you will see that it returns 10 scores per instance (instead of just 1):

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores
array([[ 2.92492871,  7.02307409,  3.93648529,  0.90117363,  5.96945908,
         9.5       ,  1.90718593,  8.02755089, -0.13202708,  4.94216947]])
```

- That's one score per class
- The highest score is indeed the one corresponding to class 5

```
>>> np.argmax(some_digit_scores)
5
>>> svm_clf.classes_
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
>>> svm_clf.classes_[5]
5
```

If you want to force Scikit-Learn to use OvO or OvR, you can use the OneVsOneClassifier or OneVsRestClassifier classes:

- simply create an instance and pass a classifier to its constructor
  - it does not even have to be a binary classifier

E.g. this creates a multiclass classifier using the OvR strategy, based on an SVC:

```
>>> from sklearn.multiclass import OneVsRestClassifier
>>> ovr_clf = OneVsRestClassifier(SVC())
>>> ovr_clf.fit(X_train, y_train)
>>> ovr_clf.predict([some_digit])
array([5], dtype=uint8)
>>> len(ovr_clf.estimators_)
10
```

Error analysis on a full, multi-class classification for MNIST

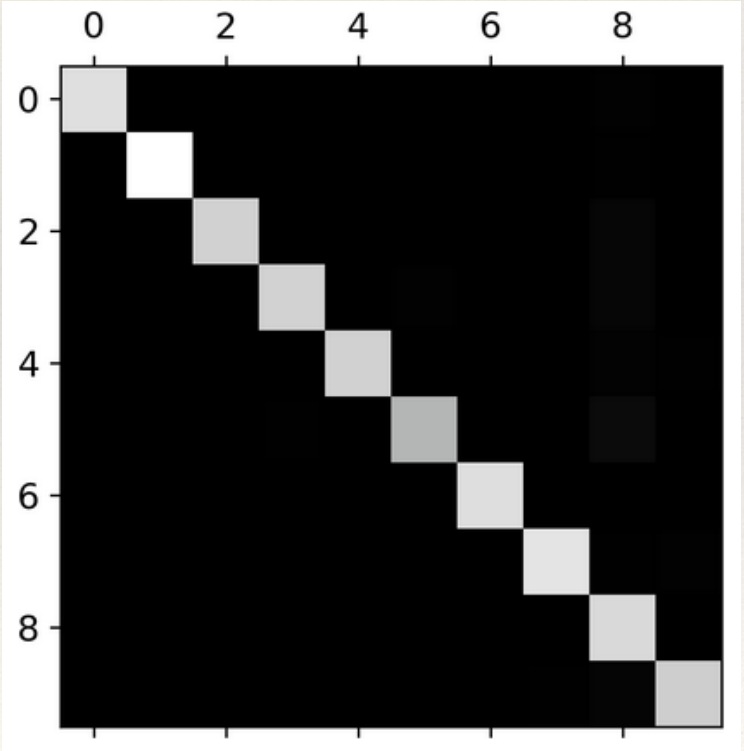
Assume you selected and trained your model. Now you want to improve it: you need first to **analyse the type of errors it makes**

- start by checking the confusion matrix
- note it is not 2x2 now, but 10x10 better to visualize it (matplotlib)

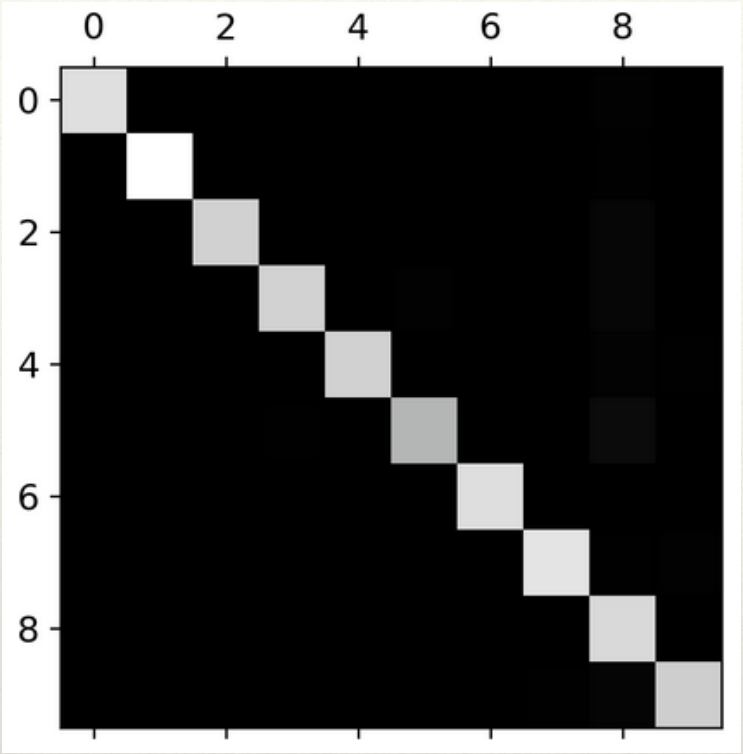
```
array([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [   0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

Fairly good, since most images are on the main diagonal, which means that they were classified correctly.

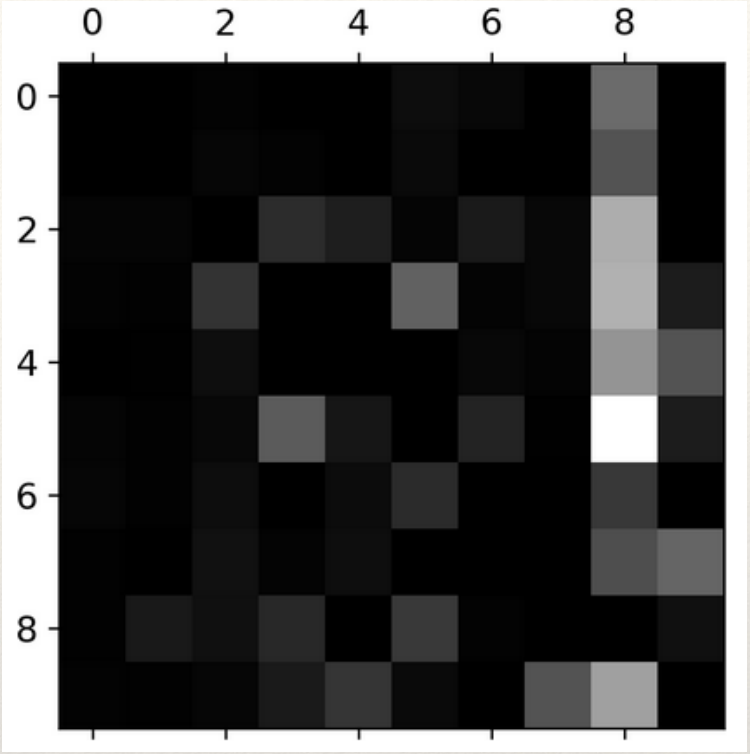
5s is a bit darker though. Less statistics for 5s? classifier performs worse on 5s wrt other numbers?



Error analysis on a full, multi-class classification for MNIST



Normalise each value in the confusion matrix to the number of images in the corresponding class



Observations and possible actions: many images get misclassified as 8s. 5s are often misclassified as 8s and as 3s

- **you should reduce the false 8s** (gather more training data that looks like 8s but are not? engineer new features that would help the classifier (e.g. count # closed loops in digits, preprocess the image to highlight more e.g. loops, ..)
- **you should improve on 3s vs 5s** (too similar: try a non-linear model? insist on image preprocessing to ensure that they are well centered and not too rotated? ..)

MNIST recap

Now you know how to:

- get the data and inspect it
- train a binary classifier
- measuring performance using CV → accuracy
- confusion matrix
- precision, recall and their trade-off
- F1 score
- ROC, AUC
- train a multi class classifier
- error analysis

Well done!

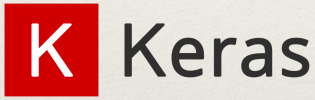
Tools for the hands-on's

(the past ones, and the future ones)

ML tools and frameworks

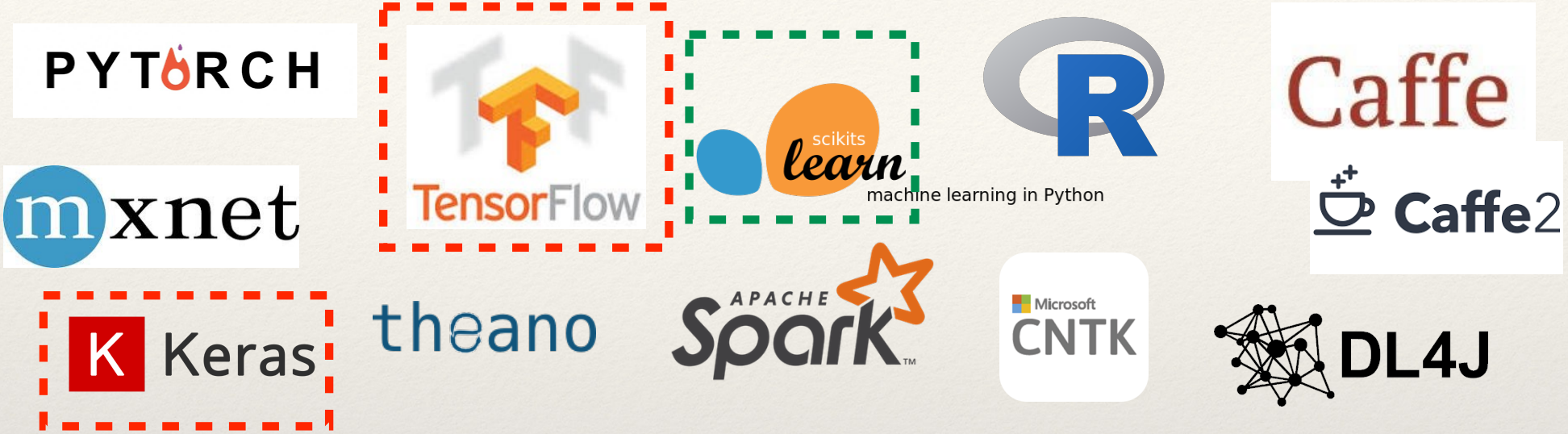
[DISCLAIMER: not an exaustive list..]

machine learning in Python



ML tools and frameworks

[DISCLAIMER: not an exaustive list..]



Nowadays, often tools define *what* you do and not only *how*…

- get familiar with Sklearn + Keras/Tensorflow + Pytorch + fast.ai + …

Together: **scikit-learn + Keras/Tensorflow**

Scikit-learn

machine learning in Python

https://scikit-learn.org

Tensorflow and Keras

Adopt something that provides you with a modern description, implementation and application of learning algorithms, including neural networks (of course!)

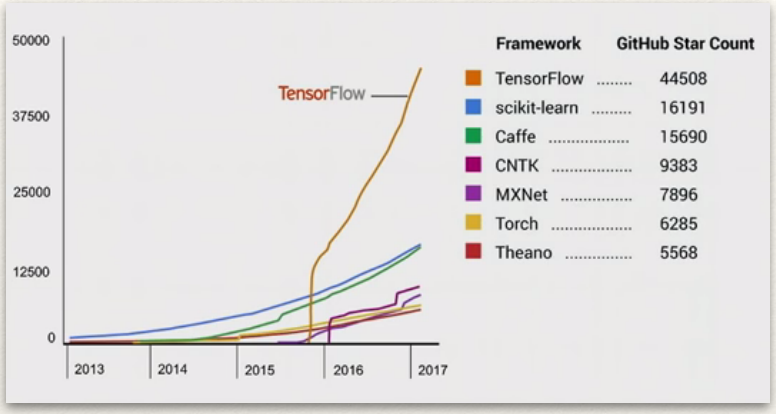
a good backend choice

TensorFlow: Low-level implementation of operations needed to implement (e.g.) neural networks in multi-threaded CPU and multi GPU environments (*basically, all this.. transparently!*)

Keras: High-level convenience wrapper for backend libraries, e.g. TensorFlow, to implement neural network models

on top of (e.g.) Tensorflow

Both quite popular, and widely adopted



Bump in 2015 as TF became public

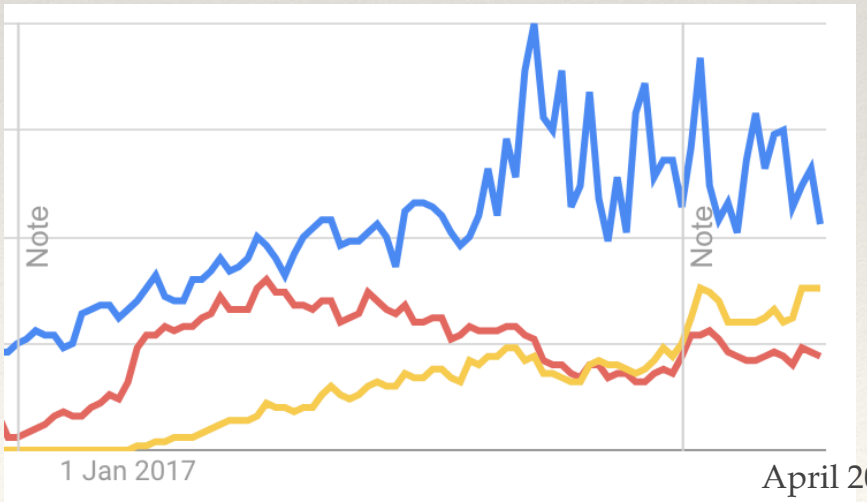
Recently:

Interest over time

Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means that there was not enough data for this term.

[Disclaimer: plenty of caveats in such comparisons..]

[google trends]



Keras

PyTorch

Tensorflow

Tensorflow

TensorFlow is an open source (since 2015) software library by the Google Brain team for **numerical computation using data flow graphs**.

- **Nodes** in the graph represent mathematical operations..
- .. while the graph edges represent the multidimensional data arrays (**tensors**) communicated between them.

In first approx: Tensorflow is not only about NNs..

.. but it is a perfect match to implement NNs efficiently.

Keras

(Most) popular tool to train and apply NNs

Python wrapper around multiple numerical computation libraries

- e.g. TensorFlow
- but, backends: TensorFlow, Theano, CNTK , ..

Pros:

- Hides most of the low-level operations that you don't want to care about

Cons:

- Sacrificing little functionality for much easier user interface

Main asset: **being able to go from idea to result with the least possible delay**

Keras + Tensorflow

Keras

keras.io

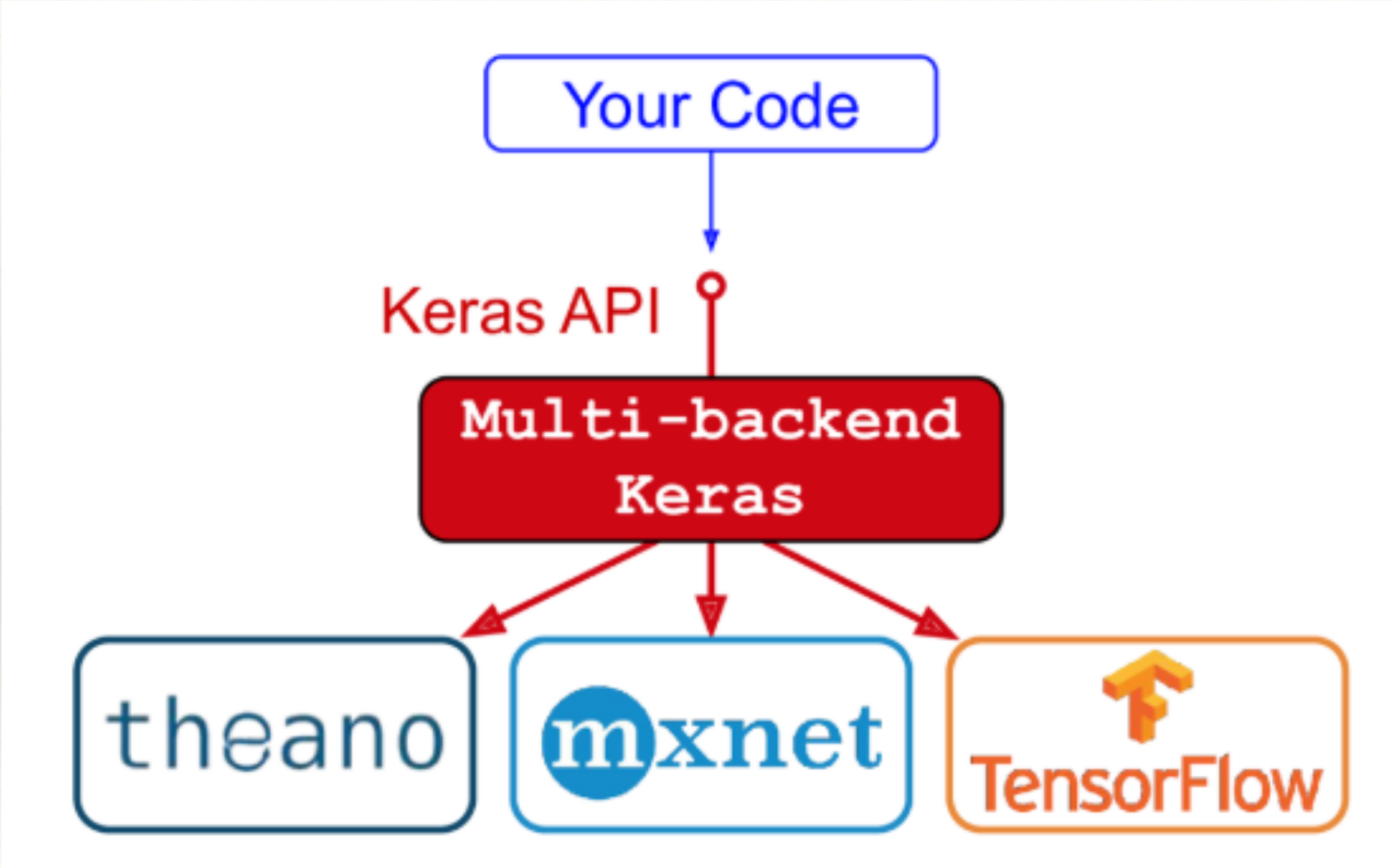
- a high-level DL API that allows to build, train, evaluate, execute all sorts of NN
- Developed by François Chollet as part of a research project called ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System). Released as an open source project in March 2015
- Quickly gained popularity, thanks to ease-of-use, flexibility, beautiful design

Keras cannot work "alone"

- for the heavy computations required by NNs, the keras implementation relies on a computation backend
- As of 2015, you could choose from 3 popular open source DL libs:
  - Google TensorFlow, Microsoft Cognitive Toolkit (CNTK), Theano (caveats apply here)

We will refer to this keras reference implementation as a **multi-backend Keras**.

Origin of **Keras**



More on **Keras**

Not the end of the story, though..

Late 2016: other implementations have been released, enabling users to run Keras on:

- Apache MXNet
- Apple's Core ML
- Javascript/Typescript (to run Keras code in a web browser)
- PlaidML (which can run on all sorts of GPU devices, not just Nvidia)
- …

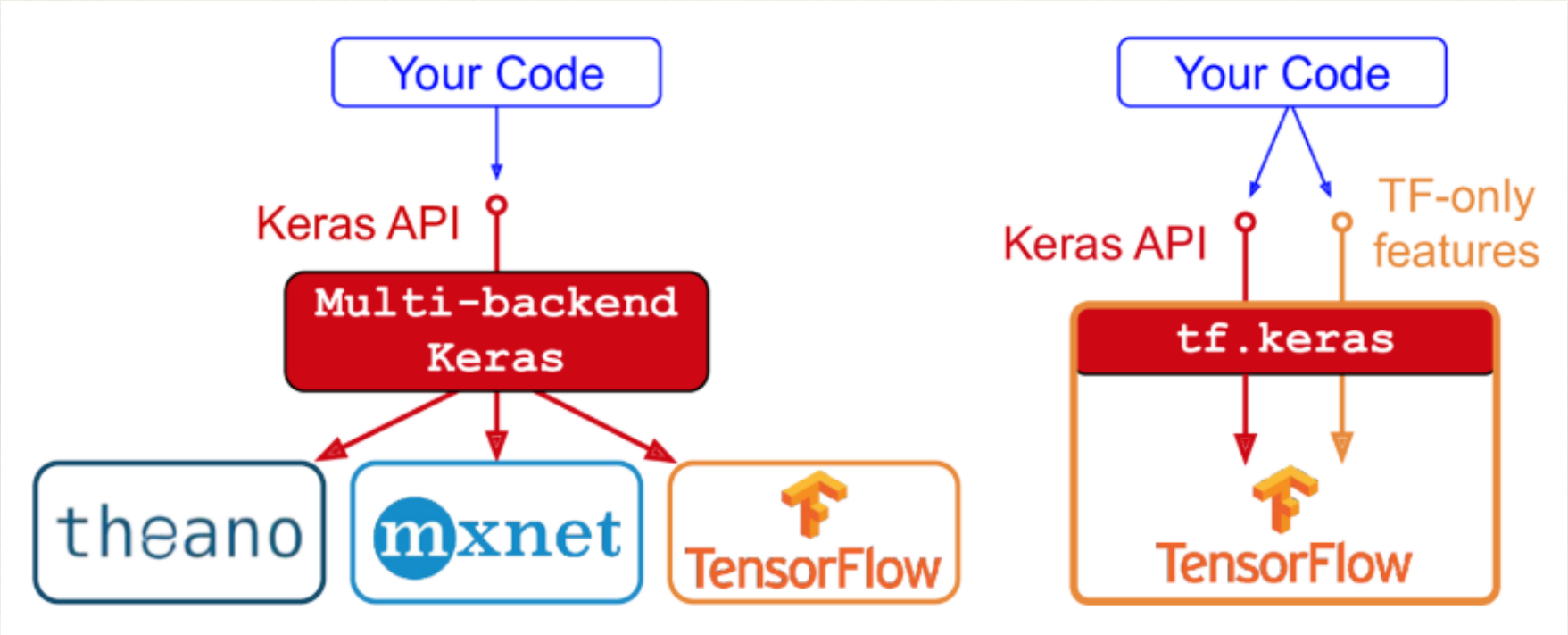
Moreover, changes on the Google Tensorflow side happened too..
TensorFlow itself now comes **bundled** with its own Keras implementation, called **tf.keras**

- Limitation: it only supports TensorFlow as the backend
- Advantage: it offers some very useful extra features (see next)

tf.keras

E.g. tf.keras supports TensorFlow's Data API, which makes it easy to load and preprocess data efficiently

- One might argue about this choice for many reasons.. anyway tf.keras supporters state that if one wants to use Keras on TF, using tf.keras is currently just the best option



NOTE for this course: in all examples, an attempt will be made to avoid using any of the TF-specific features, so the code should run just fine on other Keras implementations as well (at least in Python), with only minor modifications, such as changing the imports.

PyTorch

Another popular DL library, in parallel with Keras + TensorFlow, is the **PyTorch** library.

- its API is quite similar to Keras
  - this in part because both APIs were inspired by Scikit-Learn, among others..
- "once you know one, it is quicker to switch to the other": is it true?

Note that PyTorch's popularity **grew exponentially since 2018**, largely thanks to its simplicity and excellent documentation

- these were NOT AT ALL TensorFlow 1.x's main strengths
- However, TensorFlow 2 is now arguably just as simple as PyTorch, as it has adopted Keras as its official high-level API and has greatly simplified and cleaned up the rest of the API, the documentation has been completely re-organized, etc.
- Similarly, PyTorch's main weaknesses (e.g., limited portability and no computation graph analysis) have been largely addressed in PyTorch 1.0 (and beyond, now..)

A competition (if healthy..) is beneficial to end-users!

Keras interface for a shallow NN

(+ wrapping sklearn into it, now that we are at it)

Among the possible **Keras** interfaces (e.g. Sequential, Functional) we focus on the **Sequential** interface at this stage.

The Keras Sequential interface defines a neural network as a sequence of steps.

add()

https://keras.io/api/models/sequential/

add method

```
Sequential.add(layer)
```

Adds a layer instance on top of the layer stack.

Arguments

- **layer**: layer instance.

Raises

- **TypeError**: If `layer` is not a layer instance.
- **ValueError**: In case the `layer` argument does not know its input shape.
- **ValueError**: In case the `layer` argument has multiple output tensors, or is already connected somewhere else (forbidden in `Sequential` models).

```
model.add(layer)
```

Keras Sequential → add()

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

All these model are a good starting point for the MNIST exercise

And then:

①

```
model1 = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax')])
```

Input_shape *needs to be specified in the input layer to allocate proper size tensors, ..*

Keras Sequential → add()

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

All these model are a good starting point for the MNIST exercise

And then:

①

②

```
model1 = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax')])
```

Input_shape *needs to be specified in the input layer to allocate proper size tensors, ..*

The most popular →

```
model2 = Sequential()
model2.add(Dense(32, input_dim=784))
model2.add(Activation('relu'))
model2.add(Dense(10))
model2.add(Activation('softmax'))
```

.. then in subsequent layers it is determined by previous layers

Keras Sequential → add()

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

All these model are a good starting point for the MNIST exercise

And then:

①

```
model1 = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax')])
```

Input_shape *needs to be specified in the input layer to allocate proper size tensors, ..*

②

The most popular →

```
model2 = Sequential()
model2.add(Dense(32, input_dim=784))
model2.add(Activation('relu'))
model2.add(Dense(10))
model2.add(Activation('softmax'))
```

.. then in subsequent layers it is determined by previous layers

③

The shortest (and most readable, perhaps?) →

```
model3 = Sequential(
    [Dense(32, input_shape=(784,), activation='relu'),
    Dense(10, activation='softmax')])
```

summary()

https://keras.io/api/models/model/

summary method

```
Model.summary(line_length=None, positions=None, print_fn=None)
```

Prints a string summary of the network.

Arguments

- **line_length**: Total length of printed lines (e.g. set this to adapt the display to different terminal window sizes).
- **positions**: Relative or absolute positions of log elements in each line. If not provided, defaults to `[.33, .55, .67, 1.]`.
- **print_fn**: Print function to use. Defaults to `print`. It will be called on each line of the summary. You can set it to a custom function in order to capture the string summary.

Raises

- **ValueError**: if `summary()` is called before the model is built.

```
model.summary(line_length=None, positions=None, print_fn=None)
```

summary()

```
model3 = Sequential(
    [Dense(32, input_shape=(784,), activation='relu'),
    Dense(10, activation='softmax')])
```

```
modelX.summary()
```

①②

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 32)                25120
_________________________________________________________________
activation_4 (Activation)    (None, 32)                0
_________________________________________________________________
dense_7 (Dense)              (None, 10)                330
_________________________________________________________________
activation_5 (Activation)    (None, 10)                0
=================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
_________________________________________________________________
```

Legenda: red (matrix), green (bias vector)

For each layer, you can see the output shape *and the #* of params

784 * 32 + 32 = **25120**
32 * 10 + 10 = **330**

And at the end the total # of *(trainable)* params

25120 + 330 = **25450**

③

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_10 (Dense)             (None, 32)                25120
_________________________________________________________________
dense_11 (Dense)             (None, 10)                330
=================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
_________________________________________________________________
```

The # of learnable parameters is one way to characterise the degrees of freedom of the NN and the level of complexity.

Tiny NN.. but already found 2 dozens of thousands of learnable params!

compile()

https://keras.io/api/metrics/

The `compile()` method takes a `metrics` argument, which is a list of metrics:

```
model.compile(
    optimizer='adam',
    loss='mean_squared_error',
    metrics=[
        metrics.MeanSquaredError(),
        metrics.AUC(),
    ]
)
```

Before one starts learning the model, you specify an optimizer, the loss function to use, and the metrics that you want to log during training.

```
model.compile(optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_mode=None,
weighted_metrics=None, target_tensors=None, **kwargs)
```

compile()

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

# or:
#model3.compile("adam", "categorical_crossentropy", metrics=['accuracy'])
```

The adam optimiser as a good default choice (many more options..)

For classification problems. Needed in this case, where you have softmax in the output layer

The accuracy metric is just fine to monitor the convergence of the NN training process

compile() → optimizer

https://keras.io/api/optimizers/

Available optimizers

- SGD
- RMSprop
- Adam
- Adadelta
- Adagrad
- Adamax
- Nadam
- Ftrl

(in keras.io today)

The list of available optimisers may change (update) frequently.

Prepare your data

To fit anything, i.e. before invoking any fit(), you need to prepare your data:

```
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()

print('Before reshaping:', X_train.shape)
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
print('After reshaping:',X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

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
import keras
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
Before: (60000, 28, 28)
After: (60000, 784)
60000 train samples
10000 test samples
```

Several actions needed:

- import
- reshape
- type conversion
- rescaling
- OHE of the targets
  - multi (10) class classification → the target vector needs to be in the right shape, it must be a vector of length 10 to match the size of the output layer

fit()

Similar to sklearn fit method, but in Keras it takes a bunch of other arguments, not only training and test sets.

```
model.fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None,
validation_split=0.0, validation_data=None, shuffle=True, class_weight=None,
sample_weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None,
validation_freq=1, max_queue_size=10, workers=1, use_multiprocessing=False, **kwargs)
```

fit()

```
model.fit(X_train, y_train, batch_size=128, epochs=10, verbose=1)
```

```
Epoch 1/10
60000/60000 [==============================] - 1s 21us/step - loss: 0.5091 - accuracy: 0.8605
Epoch 2/10
60000/60000 [==============================] - 1s 20us/step - loss: 0.2610 - accuracy: 0.9249
Epoch 3/10
60000/60000 [==============================] - 1s 20us/step - loss: 0.2189 - accuracy: 0.9369
Epoch 4/10
60000/60000 [==============================] - 1s 20us/step - loss: 0.1926 - accuracy: 0.9449
Epoch 5/10
60000/60000 [==============================] - 1s 20us/step - loss: 0.1732 - accuracy: 0.9496
Epoch 6/10
60000/60000 [==============================] - 1s 20us/step - loss: 0.1587 - accuracy: 0.9538
Epoch 7/10
60000/60000 [==============================] - 1s 21us/step - loss: 0.1469 - accuracy: 0.9582
Epoch 8/10
60000/60000 [==============================] - 1s 22us/step - loss: 0.1353 - accuracy: 0.9608
Epoch 9/10
60000/60000 [==============================] - 1s 20us/step - loss: 0.1268 - accuracy: 0.9635
Epoch 10/10
60000/60000 [==============================] - 1s 21us/step - loss: 0.1188 - accuracy: 0.9653
```

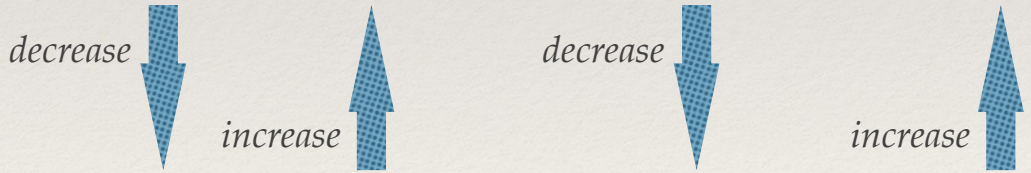
Acc 96.5% (note: *not* validated): not so brilliant for MNIST anyway.. but it is a tiny NN, only 10 epochs, etc..

fit() with validation

Static TV splitting: training on 54k, validate on 6k

```
model.fit(X_train, y_train, batch_size=128, epochs=10, verbose=1, validation_split=.1)
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 1s 22us/step - loss: 0.5365 - accuracy: 0.8548 - val_loss: 0.2368 - val_accuracy: 0.9362
Epoch 2/10
54000/54000 [==============================] - 1s 20us/step - loss: 0.2647 - accuracy: 0.9257 - val_loss: 0.1918 - val_accuracy: 0.9478
Epoch 3/10
54000/54000 [==============================] - 1s 20us/step - loss: 0.2202 - accuracy: 0.9381 - val_loss: 0.1690 - val_accuracy: 0.9523
Epoch 4/10
54000/54000 [==============================] - 1s 21us/step - loss: 0.1936 - accuracy: 0.9443 - val_loss: 0.1529 - val_accuracy: 0.9572
Epoch 5/10
54000/54000 [==============================] - 1s 21us/step - loss: 0.1751 - accuracy: 0.9498 - val_loss: 0.1408 - val_accuracy: 0.9593
Epoch 6/10
54000/54000 [==============================] - 1s 22us/step - loss: 0.1595 - accuracy: 0.9544 - val_loss: 0.1361 - val_accuracy: 0.9620
Epoch 7/10
54000/54000 [==============================] - 1s 20us/step - loss: 0.1472 - accuracy: 0.9577 - val_loss: 0.1308 - val_accuracy: 0.9637
Epoch 8/10
54000/54000 [==============================] - 1s 21us/step - loss: 0.1362 - accuracy: 0.9602 - val_loss: 0.1243 - val_accuracy: 0.9648
Epoch 9/10
54000/54000 [==============================] - 1s 21us/step - loss: 0.1270 - accuracy: 0.9631 - val_loss: 0.1202 - val_accuracy: 0.9652
Epoch 10/10
54000/54000 [==============================] - 1s 21us/step - loss: 0.1194 - accuracy: 0.9654 - val_loss: 0.1223 - val_accuracy: 0.9657
```



Acc **~96.5%** (w/ validation). Accuracies (T vs V) very similar. Tiny NN, no symptoms of overfitting. Trend in loss and acc indicate that training for more epoch might help (5x more epochs might give you up to +0.5%). *Note: this is NOT the accuracy you might expect to generalise to.*

evaluate

Similar to score in sklearn

```
score = model.evaluate(X_test, y_test, verbose=0)
print("Test loss: {:.3f}".format(score[0]))
print("Test Accuracy: {:.3f}".format(score[1]))
```

```
Test loss: 0.137
Test Accuracy: 0.959
```

Output is a tuple with various score metrics

Acc **~96%** (slightly lower w.r.t last slide). *This is the accuracy you should expect to generalise to* if you run the model against previously unseen samples.

What to do next? Probably most additional gain might come from designing a more complex NN.

Probably one wants to see in some more detail what happened in this process → next

loggers and callbacks

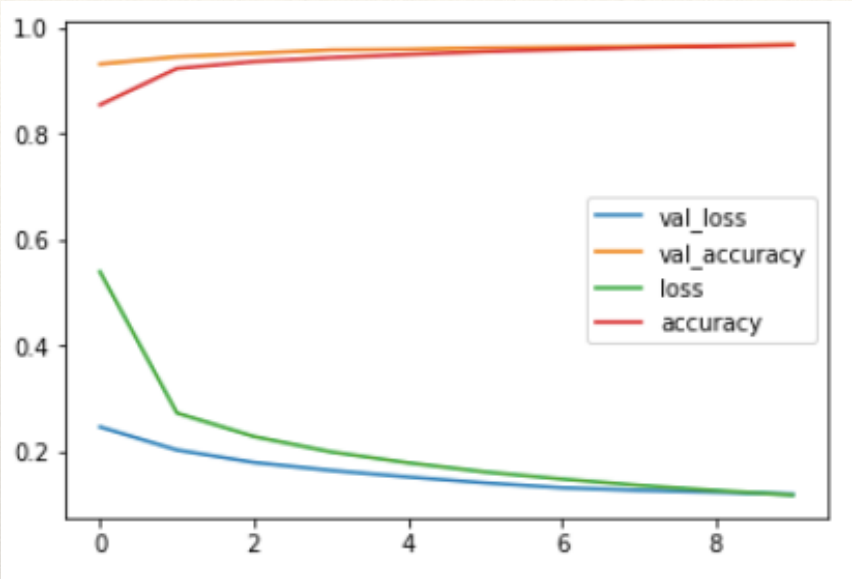
```
import pandas as pd
```

```
#%%time
history_callback = model.fit(X_train, y_train,
                             batch_size=128,
                             epochs=10,
                             verbose=1,
                             validation_split=.1)
pd.DataFrame(history_callback.history).plot()
```

The fit() method returns a callback, that you can exploit. The history() methods returns a dictionary, that you can e.g. convert into a Pandas Dataframe and plot.

loggers and callbacks

epochs=**10**

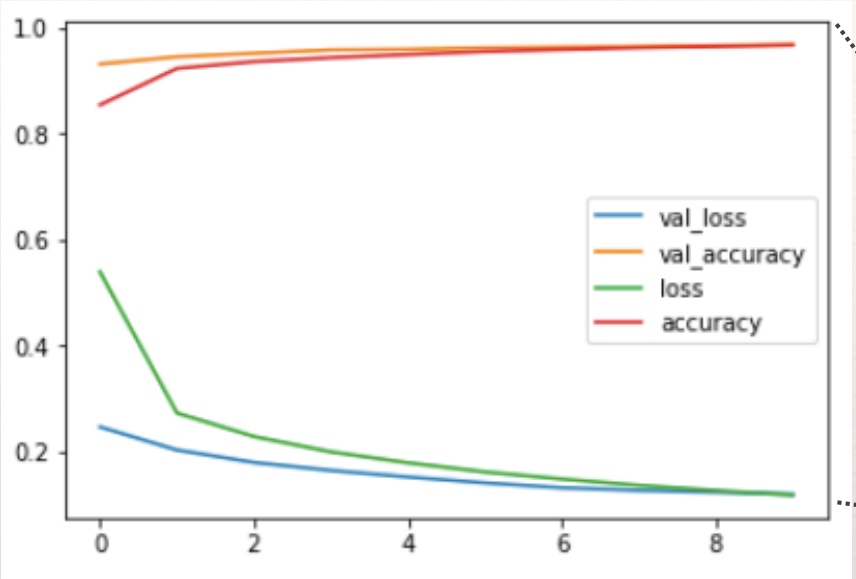



```
Test loss: 0.134
Test Accuracy: 0.960
```

Everything looks fine..
and no overfitting..

loggers and callbacks

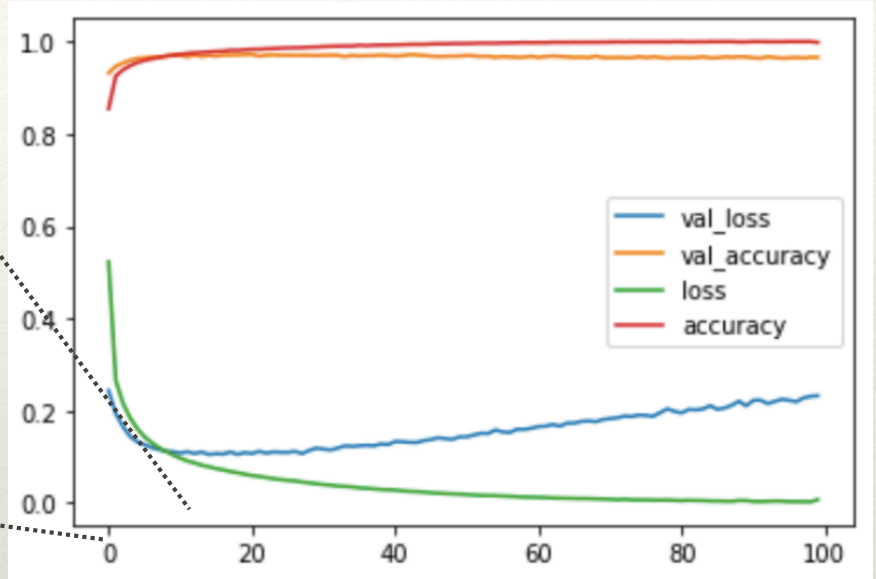
epochs=**10**




```
Test loss: 0.134
Test Accuracy: 0.960
```

Everything looks fine..
and no overfitting..

epochs=**100**




```
Test loss: 0.223
Test Accuracy: 0.966
```

While the training loss is decreasing,
the validation loss is increasing..
synthoms of overfitting..

Keras wrappers for sklearn

For those disappointed that we are not using sklearn any more..

```
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV

def make_model(optimizer="adam", hidden_size=32):
    model = Sequential([
        Dense(hidden_size, input_shape=(784,)),
        Activation('relu'),
        Dense(10),
        Activation('softmax'),
    ])
    model.compile(optimizer=optimizer,loss="categorical_crossentropy",
                  metrics=['accuracy'])
    return model

clf = KerasClassifier(make_model)

param_grid = {'epochs': [1, 5, 10],
              'hidden_size': [32, 64, 256]}
grid = GridSearchCV(clf, param_grid=param_grid, return_train_score=True)
grid.fit(X_train, y_train)
```

As Keras Sequential's and sklearn's interfaces are different, solution is that sklearn is wrapped into a "KerasClassifier" (and KerasRegressor..)

define a function that makes, compiles and returns the model (note the parameters as function's arguments)

make_model is then wrapped, and the clf object will be treated as a sklearn model (e.g. GridSearchCV..)

We used Keras to make the model, then treated it like any other sklearn model! Cool!

performance

The training will be quite verbose and hard to read with clarity. Best is to insert *all CV results* into a data frame and visualise what we are mostly interested in

```
res = pd.DataFrame(grid.cv_results_)
res.pivot_table(index=["param_epochs", "param_hidden_size"],
                values=['mean_train_score', "mean_test_score"])
```

		mean_test_score	mean_train_score
param_epochs	param_hidden_size		
1	32	0.928467	0.933608
	64	0.939750	0.947508
	256	0.959333	0.966833
5	32	0.957233	0.971304
	64	0.965467	0.981850
	256	0.974000	0.993288
10	32	0.961900	0.982208
	64	0.969633	0.991838
	256	0.976333	0.996846

Training for 10 epochs, with an hidden layer size of 256, a very good training accuracy can be obtained (99.7%), with a 97.6% performance on the test set. With more fine tuning, one could perhaps increase the latter to ~98-99%.