Keras introduction

Romain Gautron

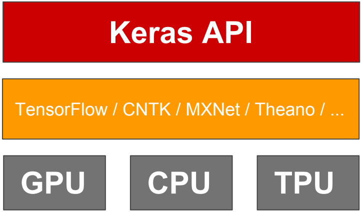
CIAT

March 5, 2019

Platform for
Big Data
in Agriculture

What is Keras? (©F. Chollet)

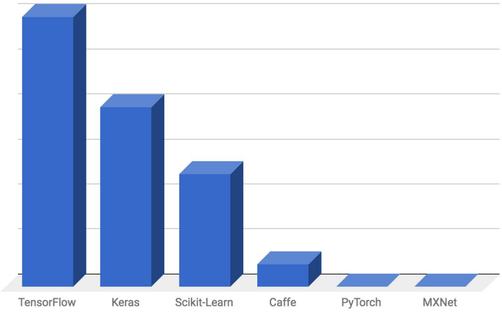
Keras: an API for specifying & training differentiable programs



Keras jobs (©F. Chollet)

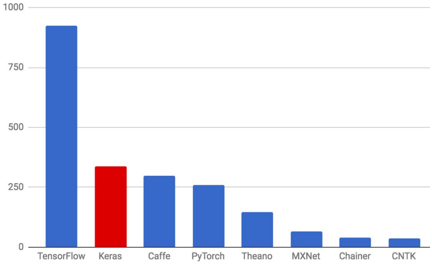
Startup-land traction

Hacker News jobs board mentions - out of 964 job postings

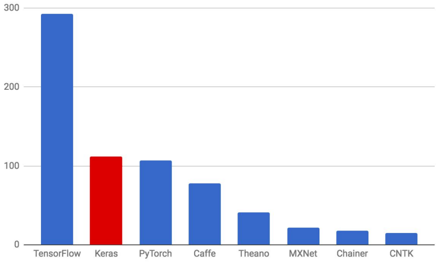


Keras in research (©F. Chollet)

Research traction



arXiv mentions as of 2018/03/07 (past 3 months)



arXiv mentions as of 2018/03/07 (past 1 month)

Keras philosophy (©F. Chollet)

The Keras user experience

Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.

This makes Keras easy to learn and easy to use. As a Keras user, you are more productive, allowing you to try more ideas than your competition, faster -- which in turn helps you win machine learning competitions.

This ease of use does not come at the cost of reduced flexibility: because Keras integrates with lower-level deep learning languages (in particular TensorFlow), it enables you to implement anything you could have built in the base language. In particular, as tf.keras, the Keras API integrates seamlessly with your TensorFlow workflows.

Three API styles

- The Sequential Model
  - Dead simple
  - Only for single-input, single-output, sequential layer stacks
  - Good for 70+% of use cases
- The functional API
  - Like playing with Lego bricks
  - Multi-input, multi-output, arbitrary static graph topologies
  - Good for 95% of use cases
- Model subclassing
  - Maximum flexibility
  - Larger potential error surface

Sequential API (©F. Chollet)

The Sequential API

```
import keras
from keras import layers

model = keras.Sequential()
model.add(layers.Dense(20, activation='relu', input_shape=(10,)))
model.add(layers.Dense(20, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

model.fit(x, y, epochs=10, batch_size=32)
```

Functional API (©F. Chollet)

The functional API

```python
import keras
from keras import layers

inputs = keras.Input(shape=(10,))
x = layers.Dense(20, activation='relu')(x)
x = layers.Dense(20, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs, outputs)
model.fit(x, y, epochs=10, batch_size=32)
```

Learning process (©F. Chollet)

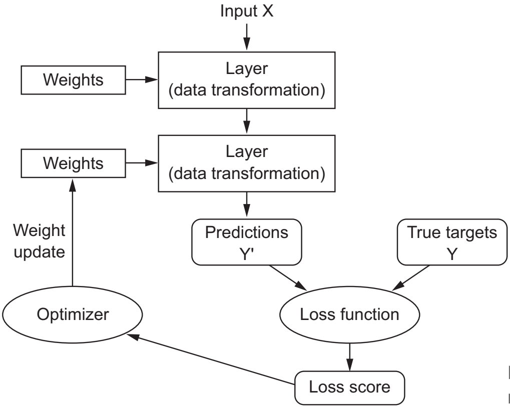
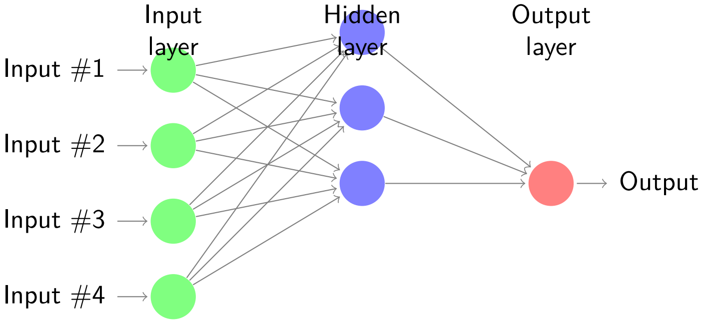


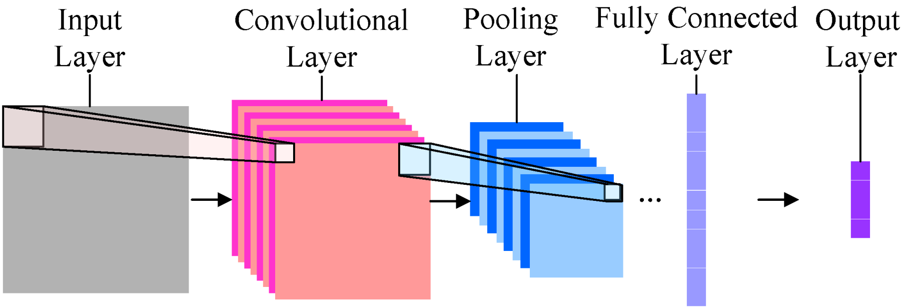
Figure 3.1 Relationship between the network, layers, loss function, and optimizer

1. Basic architectures
2. Layers types
3. Optimization
4. The use of callbacks

MLP


Input layer
Hidden layer
Output layer
Input #1
Input #2
Input #3
Input #4
Output

ConvNet general architecture



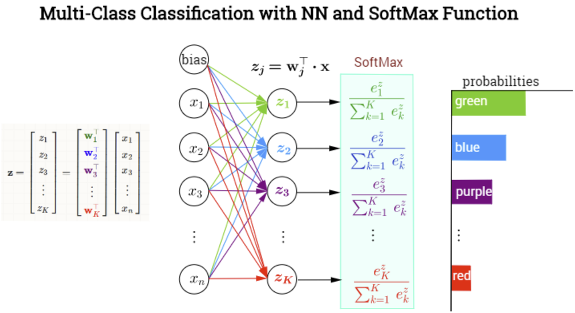
Classification VS Regression Output Layer

Classification (m classes)	Regression
$m == 2$: **1 neuron** with **sigmoid** activation $m > 2$: ***m*** **neurons** with **softmax** activation	**1 neuron** with **linear** activation

Softmax activation

Softmax

$$softmax(z_j) = \frac{\exp z_j}{\sum_{i=1}^{m} \exp z_i}, \mathbb{R} \rightarrow]0,1]$$



→ Extension of the logistic regression to a multiclass (>2) problem

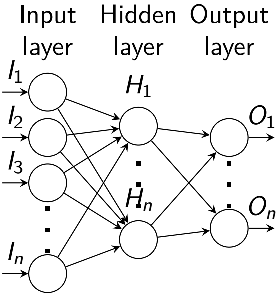
- ↪ Outputs interpreted as the probability of each class
- ↪ $\hat{y}$ is the maximum probability

Dense: fully connected layers

```
keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform')
```

Input Shape

(N_features,)



Output dim

2D

Multi-dimensional input data

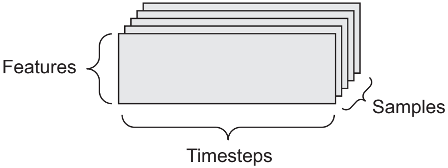


Figure 2.3 A 3D timeseries data tensor

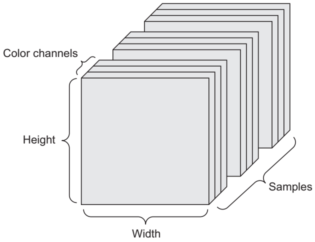


Figure 2.4 A 4D image data tensor (channels-first convention)

Convolutional layers

```
keras.layers.Conv1D(filters, kernel_size, strides=1, padding='valid')

keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid')
```

Input Shape (channel last)

Conv1D (steps, N_features)

Conv2D (height,width,channels)

For RGB images, channels = 3, for greyscale channels = 1

Output dim

Conv1D 3D

Conv2D 4D

Conv layers: padding & border effect (©F. Chollet)

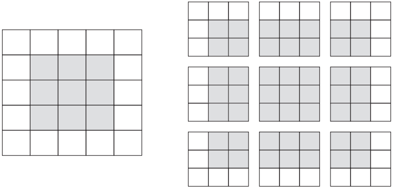


Figure 5.5 Valid locations of 3 × 3 patches in a 5 × 5 input feature map

Figure: 'valid' padding

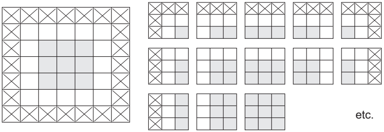


Figure 5.6 Padding a 5 × 5 input in order to be able to extract 25 3 × 3 patches

Figure: 'same' padding → same nb of patches than pixels

Conv layers: stride ©F. Chollet

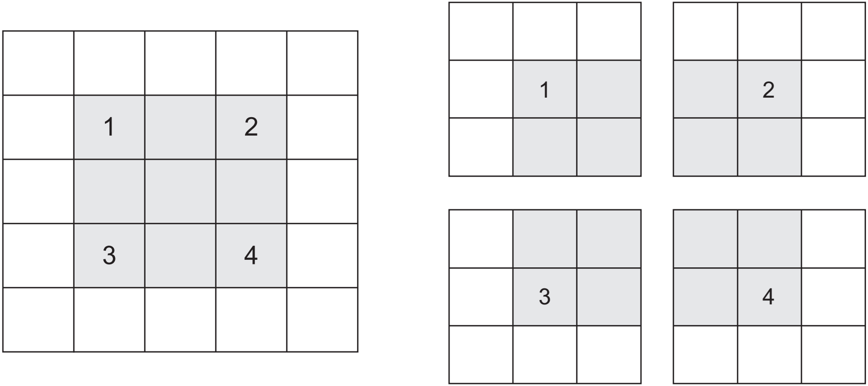


Figure 5.7 3 × 3 convolution patches with 2 × 2 strides

Maxpooling layers: downsampling

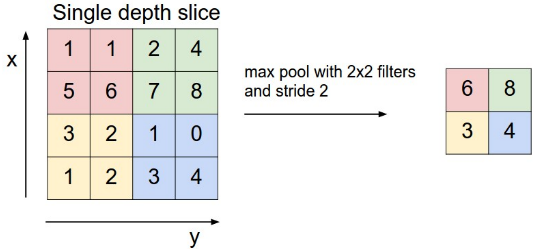
```
keras.layers.MaxPooling1D(pool_size=2, strides=None, padding='valid')

keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid')
```

Output dim

Conv1D 3D

Conv2D 4D

"strides: If None, it will default to pool_size"



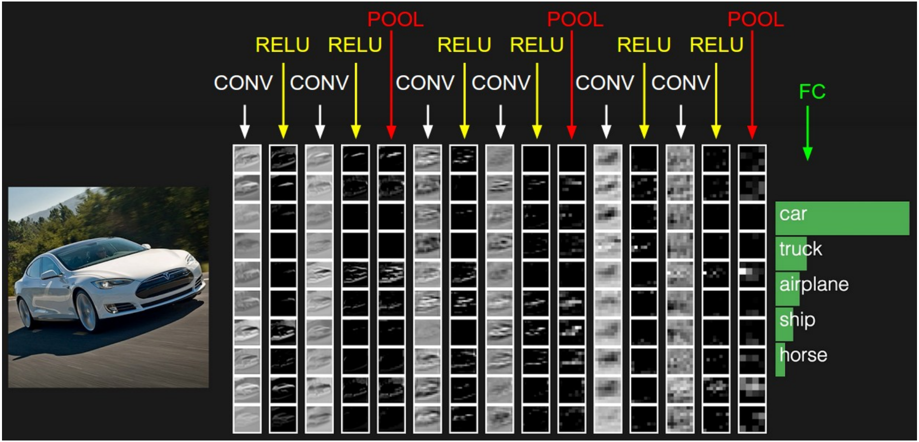
Flatten layer

```
keras.layers.Flatten()
```

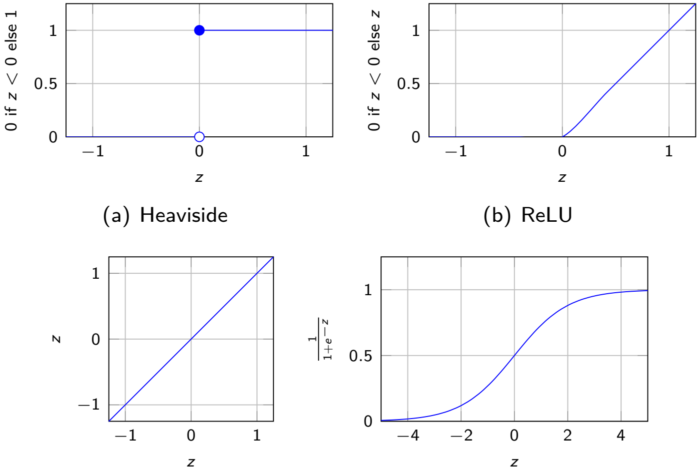
→ Flattens a N'-D input into a 2D output of shape (N_samples, N_features)

→ often between a pooling layer and a dense layer

ConvNet layer roles ©Standford University



Some NN activation functions



(a) Heaviside (b) ReLU

(c) Linear (d) Logistic sigmoid

Common layer activations in Keras

```
model.add(keras.layers.Dense(64))
model.add(keras.layers.Activation('tanh'))
# equivalent to:
model.add(keras.layers.Dense(64, activation='tanh'))
```

→ hyperbolic tangent: 'tanh'/keras.activations.tanh()

→ linear: None or 'linear'/ keras.activations.linear()

→ softmax: 'softmax'/keras.activations.softmax()

→ rectified linear unit: 'relu'/...

→ sigmoid: 'sigmoid'/...

Note: keras.layers.Activation('Name') ≡ keras.activations.name()

→ ReLu is a common choice for inner layers

BatchNormalization layer

```
keras.layers.BatchNormalization()
```

→ Applies an exponential moving average to normalize outputs of layers

→ Stabilizes learning

→ Used after Dense or Conv layers

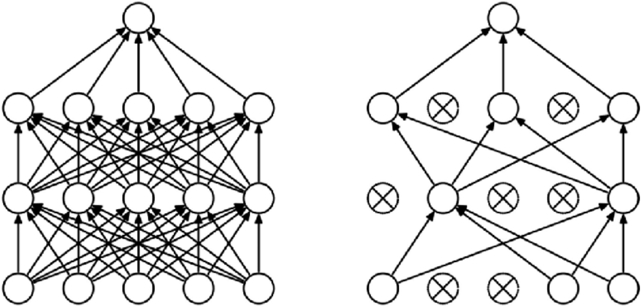
Dropout layer

```
keras.layers.Dropout(rate)
```

→ rate: probability of dropping one input to zero

→ avoid co-adaptation of units: regularizes the network

→ typically used between dense layers with rate>.5



(a) Standard Neural Net

(b) After applying dropout

Gradient descent

→ theoretically analytically possible to minimize directly loss according to weights

- ↪ intractable in practice
- ↪ we use an iterative method

Stochastic gradient descent

1. draw a random batch $(\mathbf{X}, \mathbf{y})$ from training set
2. predict $\hat{\mathbf{y}}$
3. compute loss between $\mathbf{y}$ and $\hat{\mathbf{y}}$
4. compute loss gradient regarding each weight (partial derivatives)
5. update weights in the opposite direction of gradient:
$w_{ijk}^{t+1} \leftarrow w_{ijk}^{t}(1 - lr * grad)$

Learning process (©F. Chollet)

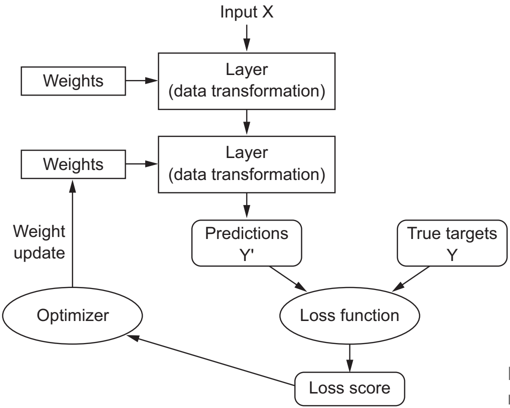


Figure 3.1 Relationship between the network, layers, loss function, and optimizer

Gradient descent (©F. Chollet)

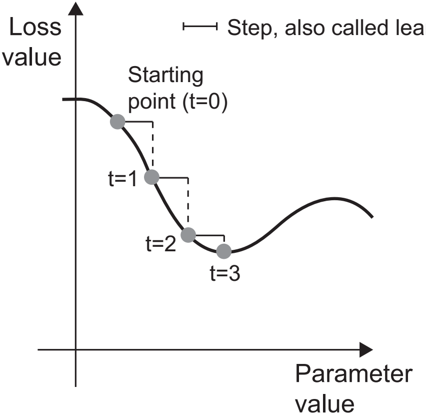


Figure 2.11 SGD down a 1D loss curve (one learnable parameter)

Gradient descent (©F. Chollet)

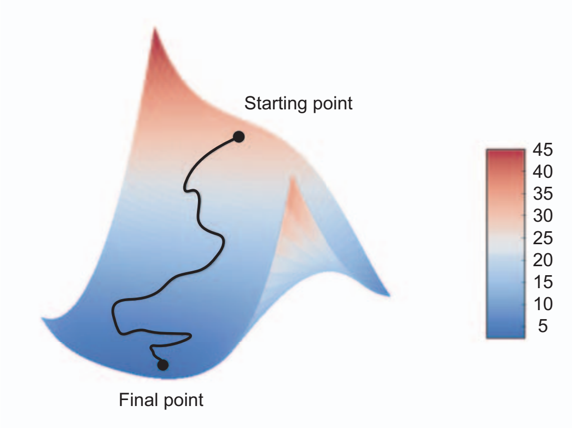


Figure 2.12 Gradient descent down a 2D loss surface (two learnable parameters)

Momentum notion (©F. Chollet)

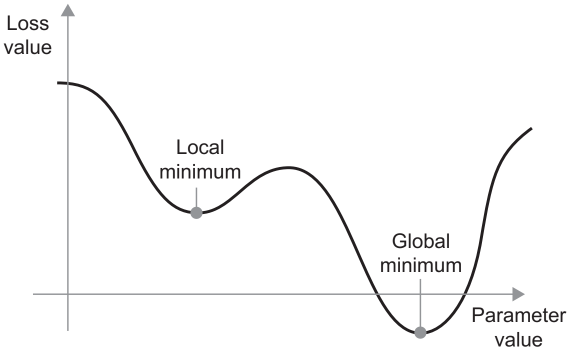


Figure 2.13 A local minimum and a global minimum

Keras common optimizers

```
keras.optimizers.SGD(lr=0.01)

keras.optimizers.RMSprop(lr=0.001)
```

Many variant based on SGD and momentum

→ SGD with momentum

→ RMSprop

→ ADAM

→ ...

↪ **RMSprop** works well in general case

↪ keep default parameters of Keras, **only adjust learning rate if needed**

↪ if called by string (ex: 'rmsprop'), comes with default values

Keras common losses

```
keras.losses.mean_squared_error(y_true, y_pred)

keras.losses.mean_absolute_error(y_true, y_pred)

keras.losses.categorical_crossentropy(y_true, y_pred)

keras.losses.binary_crossentropy(y_true, y_pred)
```

- → **Classification task**:
  - ↪ binary cross entropy ($\equiv$ log-loss): $-\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log\left(p_i\right) + \left(1-y_i\right)\log\left(1-p_i\right)\Big]$
  - ↪ $\mathbf{y} = [0, 1, 0, \cdots, 1]$
  - ↪ categorical cross entropy : $-\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{m} y_i^c \log p_i^c$
  - ↪ $\mathbf{y} = \left[[0,0,1], [0,1,0], [0,0,1], \cdots, [1,0,0]\right]$ (example of 3 classes)
- → **Regression task**:
  - ↪ Mean Squared Error: $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
  - ↪ Mean Absolute Error: $\frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$

Keras common metrics

```
keras.metrics.binary_accuracy(y_true, y_pred)

keras.metrics.categorical_accuracy(y_true, y_pred)
```

→ Not used for optimization

- ↪ measures to monitor performance for classification tasks
- ↪ can be used in with callbacks (see later)
- ↪ take max probability of predictions as predicted class

Keras steps

1. stack layers (sequential API) or manipulate tensors (functional API)
2. compile your model: specify optimizer, loss, callbacks (see later), ...
3. fit your model
4. (evaluate your model on hold test set)

Sequential vs Functional API

- **Sequential** API
  - model = keras.models.Sequential()
  - model.add(keras.layers.Dense(128,**input_shape=inputShape**))
  - model.add(keras.layers.Dense(1,activation='sigmoid'))
- **Functional** API
  - xInput = keras.Input(**shape=inputShape**)
  - x = keras.layers.dense(128)(x)
  - xOutput = keras.layers.Dense(1,activation='sigmoid')(x)
  - model = keras.models.Model(inputs=xInput,outputs=xOutput)

Model compilation: block assembling

```python
### For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
# if you need to tune the optimizer
model.compile(optimizer=RMSprop(lr=1e-4),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

### For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

### For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')
```

Model training

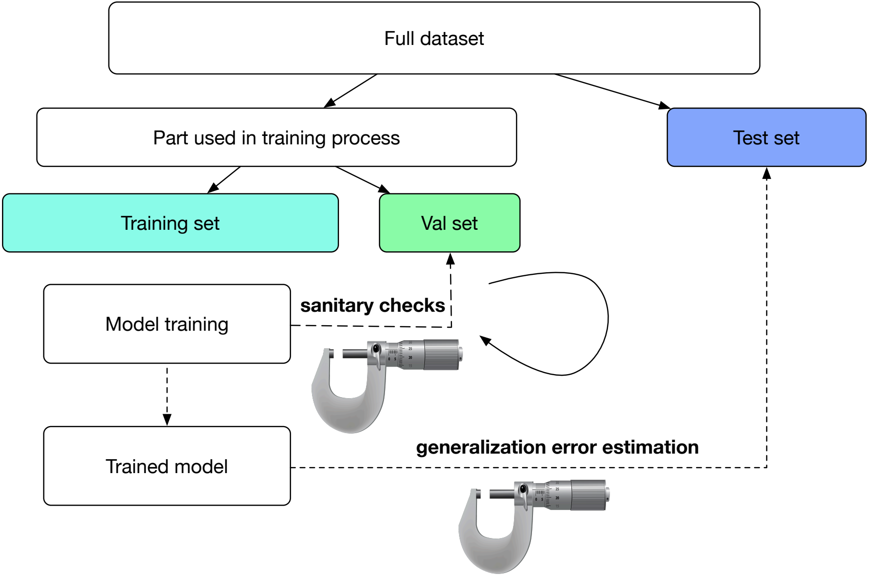
```
model.fit(x, y, batch_size, epochs=1, verbose=1,\
    callbacks=None, validation_split=0.0, validation_data=None, shuffle=True)
```

$\rightarrow$ x and y numpy.array objects, be careful with shaping

$\rightarrow$ if needed, validation_data is a tuple (x_{val}, y_{val})

$\rightarrow$ batch size: empirically better to keep it low (32 to 256)

Model training

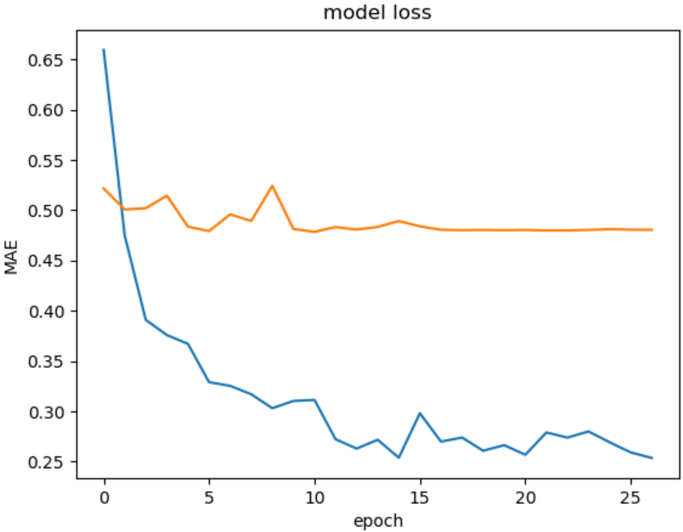


Model training

→ if you have computational resources: cross-validation

→ if you have a lot of computational resources: nested cv

Learning curves inspection



Evaluating the model

```python
## 1st way
model.evaluate(xTest,yTest)
# returns a tuple (lossValue,metricValue)
## 2nd way
yPredProb = model.predict(xTest,yTest)
# then convert yPredProd to classes
yPredClass = 1*(yPredProb>=.5) # binary
yPredClass = numpy.argmax(yPredProb, axis=-1) #multi-class
# then compute some score
score = keras.metrics.someMetric(yTest,yPred)
# or
score = keras.losses.someLoss(yTest,yPred)
```

Callbacks: very useful tools

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False)

keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=0, verbose=0, mode='auto')

keras.callbacks.TensorBoard(log_dir='./logs')

keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.001)
```

→ sanitary checkups at each epoch

→ **help prevent overfitting**

→ specified as a list within fit call