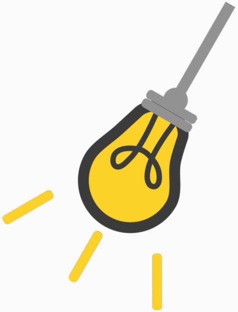
浙江大学爱丁堡大学联合学院
ZJU-UoE Institute

Lecture 15 - Using Keras to build a CNN

Nicola Romanò - nicola.romano@ed.ac.uk

Learning objectives

- Describe tools commonly used to build a CNN.
- Use Keras for building and training a "*LeNet-5 style*" CNN.
- Use Keras for transfer learning.



The tools

Python tools for deep learning

TensorFlow

- *"An end-to-end open source platform for machine learning"*
- Developed by Google

PyTorch

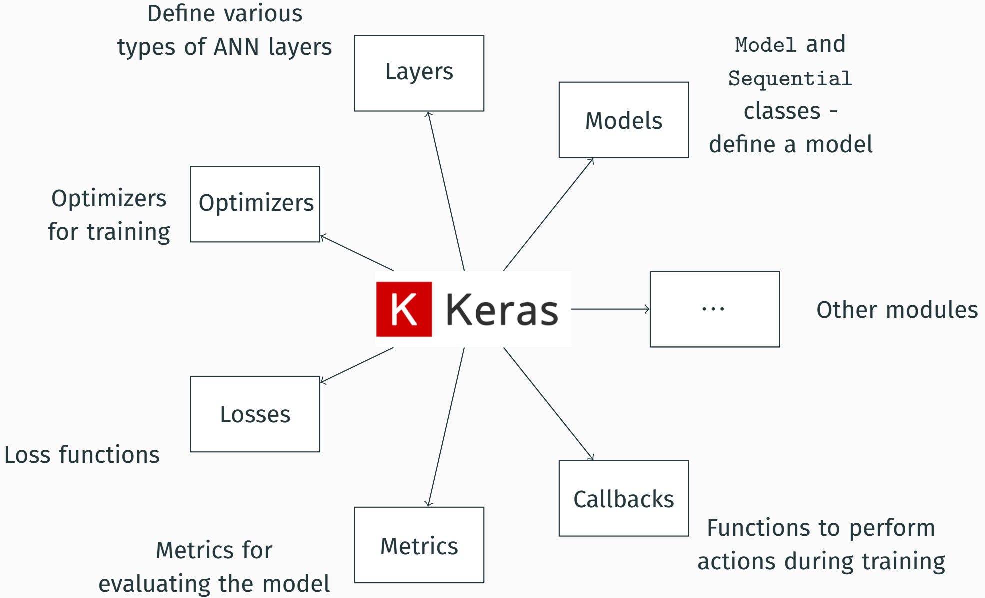
- *"An open source machine learning framework"*
- Developed by Facebook

Keras

- *"A deep learning framework"*
- Developed by François Chollet (package `keras`).
- The `keras` package also supports other "backends" (like JAX or Pythorch).

For this course we will use **Keras version 3**, but feel free to explore PyTorch as well!

A very brief overview of Keras



Layers

Keras makes it easy to define layers.

Several classes are available, such as `Conv2D`, `MaxPooling2D`, `Dense`, etc.

Keras makes it easy to define layers.

Several classes are available, such as `Conv2D`, `MaxPooling2D`, `Dense`, etc.

Convolutional layer

- 32 filters
- Size of 3x3, stride of 1, valid padding
- ReLU activation

```
layer = keras.layers.Conv2D(
   filters=32,
   kernel_size=3, strides=(1,1),
   padding='valid',
   activation='relu')
```

Layers

Keras makes it easy to define layers.

Several classes are available, such as `Conv2D`, `MaxPooling2D`, `Dense`, etc.

Convolutional layer

- 32 filters
- Size of 3x3, stride of 1, valid padding
- ReLU activation

```
layer = keras.layers.Conv2D(
   filters=32,
   kernel_size=3, strides=(1,1),
   padding='valid',
   activation='relu')
```

Dense layer

- 128 units
- Sigmoid activation

```
layer = keras.layers.Dense(
   units=128, activation='sigmoid')
```

Two ways to build a model.

Sequential API

- A sequential model is a linear stack of layers.
- You can add layers one at a time using the `add` method.

```
model = keras.models.Sequential()
model.add(layer)
model.add(layer2)
```

Functional API

- For non-linear, more complex models
- Allows multiple inputs and outputs

```
input_img = keras.Input(shape=(28, 28, 3))
FC = keras.layers.Dense(units=50)(input_img)
out = keras.layers.Dense(units=5)(FC)
model = keras.Model(inputs = input_img,
    outputs = out)
```

Compiling the model

Once the model has been created it needs to be *compiled*. This allows us to choose the optimizer, the loss function and the metrics to monitor during training.

For example, for a classification problem, we might decide to use stochastic gradient descent* as the optimizer, cross entropy as the loss function and accuracy as the metric.

* Note: Adam (ADAptive Movement estimation algorithm, Diederik et al, 2014), is an implementation of the stochastic gradient descent algorithm often used in deep learning.

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

Great! We're all set for training!

Training

At training time, we need to feed the model with data. We will have defined some **training** and **validation** set.

Now we need to set:

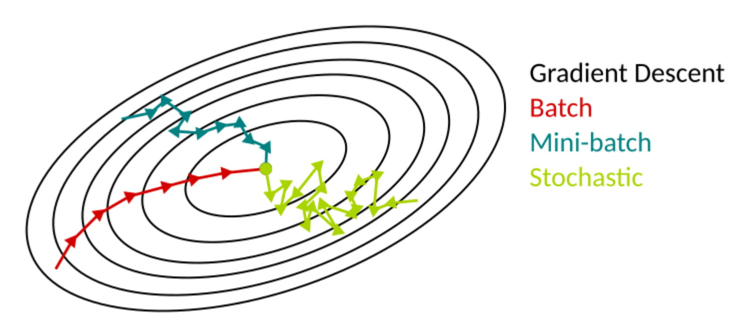
- **Epochs**. How many times to go through all the training data.

Training

At training time, we need to feed the model with data. We will have defined some **training** and **validation** set.

Now we need to set:

- **Epochs**. How many times to go through all the training data.
- **Batch size**. Training with all the data at once (*batch training*) is computationally expensive. Using **mini-batches** is faster, but might need more epochs.



At training time, we need to feed the model with data. We will have defined some **training** and **validation** set.

Now we need to set:

- **Epochs**. How many times to go through all the training data.
- **Batch size**. Training with all the data at once (*batch training*) is computationally expensive. Using **mini-batches** is faster, but might need more epochs.
- Example: 1000 training samples, `batch_size = 100`. It will take 10 **iterations** to complete one epoch.

Training

At training time, we need to feed the model with data. We will have defined some **training** and **validation** set.

Now we need to set:

- **Epochs**. How many times to go through all the training data.
- **Batch size**. Training with all the data at once (*batch training*) is computationally expensive. Using **mini-batches** is faster, but might need more epochs.
- Example: 1000 training samples, `batch_size = 100`. It will take 10 **iterations** to complete one epoch.
- The special case of `batch_size=1` is **stochastic gradient descent** (SGD).

Training

At training time, we need to feed the model with data. We will have defined some **training** and **validation** set.

Now we need to set:

- **Epochs**. How many times to go through all the training data.
- **Batch size**. Training with all the data at once (*batch training*) is computationally expensive. Using **mini-batches** is faster, but might need more epochs.
- Example: 1000 training samples, `batch_size = 100`. It will take 10 **iterations** to complete one epoch.
- The special case of `batch_size=1` is **stochastic gradient descent** (SGD).
- A forward and a backward pass are run for each iteration.

Training - code

```
history = model.fit(x_train, y_train,
    batch_size=32,
    epochs=10,
    validation_data=(x_val, y_val))
```

Note:

- The `fit` method takes as input the training data, the labels and the number of epochs to train for.
- The `fit` method returns a history object, which contains the loss and accuracy values for each epoch.

And now... predict!

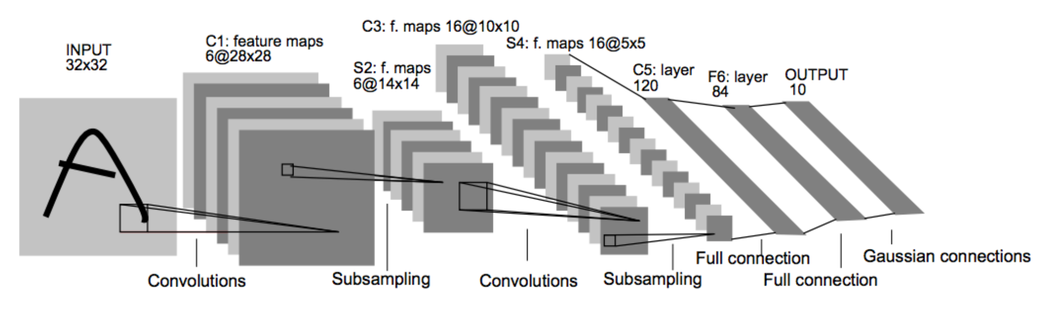
Prediction is as simple as calling the `predict` method on the model.

```
predictions = model.predict(x_test)
```

Example 1 - A “*LeNet-5 style*” CNN

Example 1 - A simple CNN

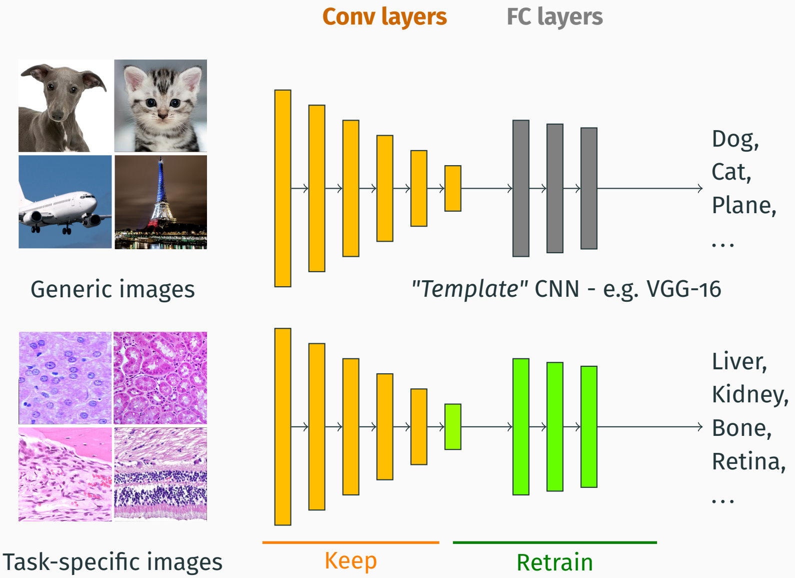
Remember the LeNet-5 CNN architecture



We are going to build a similar version, to train on the MNIST dataset. We are "remodernising" it by using ReLU activations, max pooling and a softmax output layer.

Example 2 - Transfer learning

Transfer learning



Transfer learning - VGG16 on CIFAR-10

We are going to use the pretrained VGG16 weights to classify the CIFAR-10 dataset.

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class.