Politecnico di Torino

Tensorflow

Prof. Diego Valsesia

diego.valsesia@polito.it

Academic Year 2023-2024

Politecnico di Torino

What is tensorflow?

- Open-source library by Google for **numerical computation** (not just deep learning!)
- Build a **computational graph** with the mathematical operations you want to perform
- Core **backend** developed in **C++** for efficiency:
    - supports multiple CPUs, multiple GPUs with use of libraries (**CUDA, ROCm**) for GPU acceleration
- Interface with the backend using APIs for **Python**, Java, Go, …
- Multiplatform: Linux, MacOS, Windows, Android, iOS, embedded devices, …
- Main competitor: PyTorch (Meta / Linux foundation)

Tensorflow vs. Pytorch

TensorFlow

\+ Keras high-level API: NNs in the least amount of code

\+ Very good for production environments

\+ TFLite: de-facto standard for embedded and mobile

\- Annoying and convoluted for custom implementations

\- Difficult to find implementations of latest publications

Use **Tensorflow** if:

- you want to quickly write a simple, standard NN
- you want to use it in a real-world environment/device

PYTORCH

\+ Standard for researchers

\+ Easy and powerful customization

\+ Latest features and publications

\- No good support for deployment on production or devices

Use **Pytorch** if:

- you are a researcher and want to invent new NNs

What is a tensor?

Tensor = multidimensional array

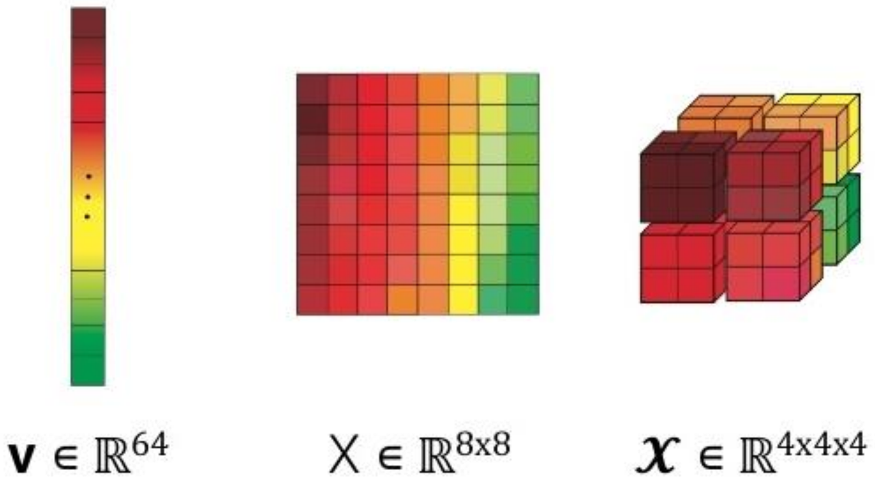
- 1D tensor: vector
- 2D tensor: matrix
- 3D tensor : RGB image

Example: a sequence of images as a 4D tensor

(100, 256, 256, 3)

Number of **images**	Number of **rows**	Number of **columns**	Number of color **channels**

*channels-last notation



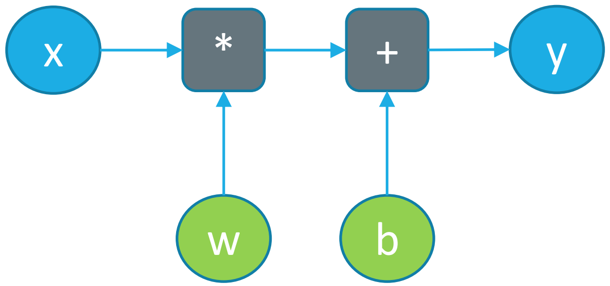
What is a computational graph?

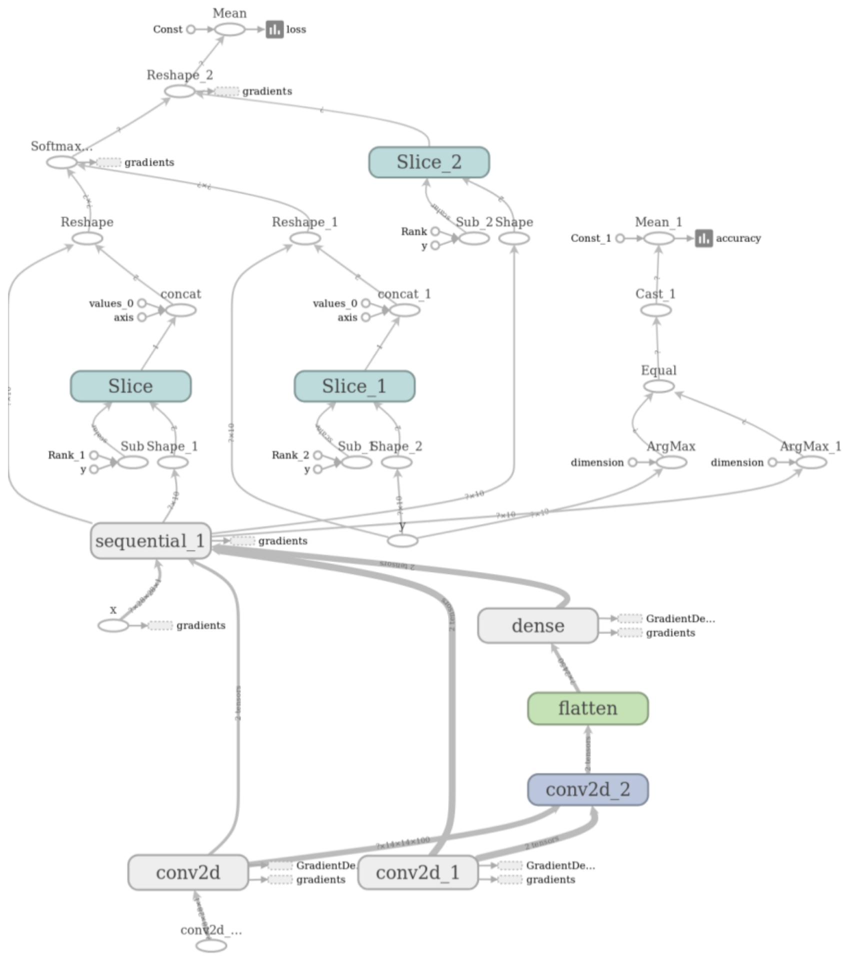
Computational graph

=

sequence of mathematical operations connected to each other as a graph of nodes

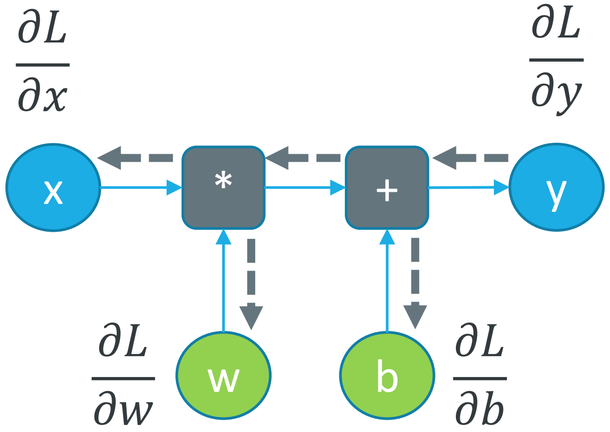
Example: $y = w * x + b$

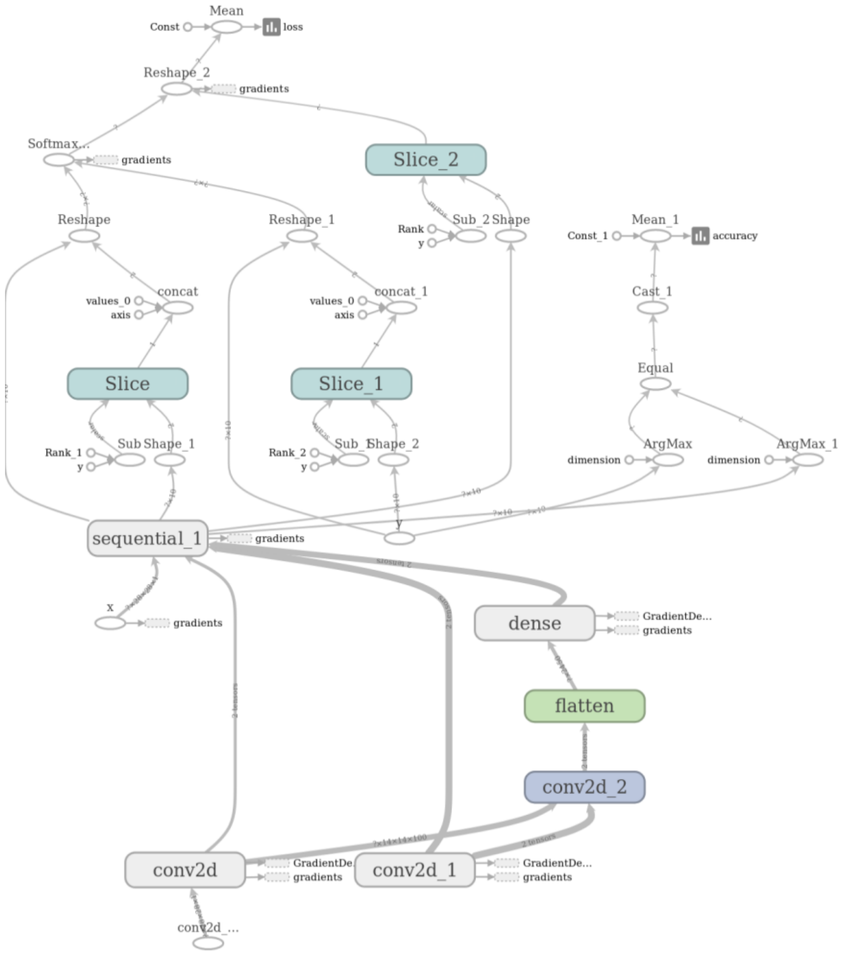




Automatic differentiation

- Tensorflow knows the **derivative** of each operation in the computational graph
- It can **backpropagate** derivatives through the computational graph





Guide to the Python API

```
import tensorflow as tf
```

Import the tensorflow Python module

Commonly used shorthand

Guide to the Python API

- Tensorflow APIs are confusing
  - rapidly evolving*
  - several different ways of doing the same thing

- 3 main levels of abstraction:
  - High-level: **tf.keras** models, tf.estimator
  - Mid-level : tf.layers (deprecated), losses, metrics, datasets
  - Low-level: tf.nn, sessions, …

Wrappers for comfortable model training and testing

Reusable components for common operations

Low-level operations used to define custom layers, models, …
Explicitly manage the computational graph

*: many online resources can be outdated presenting deprecated or more complex ways of doing things (e.g., sessions, feed_dict, manual variable management, …)

Implementing a neural network with Keras

- Create a Keras model

```
model = tf.keras.models.Sequential([…])
```

- Compile the model

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

- Train the model

```
model.fit(x_train, y_train, batch_size=…., epochs=…)
```

- Evaluate/Use the model

```
model.evaluate(x_test, y_test)
y_hat = model.predict(x)
```

Reference: https://www.tensorflow.org/guide/keras/overview

Keras models

- A **Sequential model** is a linear stack of layers
  - the output of a layer is the input of the next layer

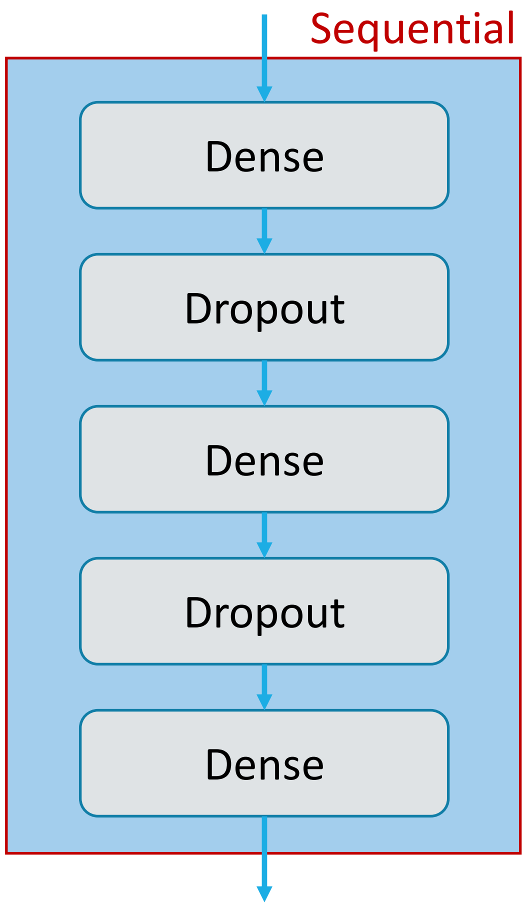
```
model = tf.keras.models.Sequential([

        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(10, activation='softmax')

])
```

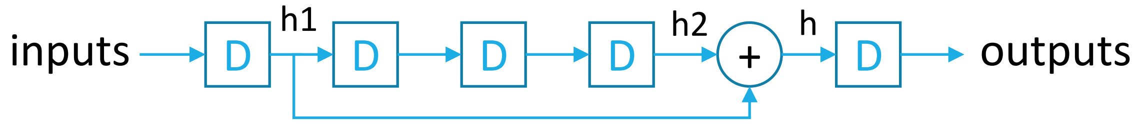
Argument is a list of layers in input to output order

Use any layer from **tf.keras.layers**



Keras models

- Arbitrarily complicated arrangements of layers can be defined with the **functional API**




```
inputs = tf.keras.Input(shape=(784,))
h1 = tf.keras.layers.Dense(64, activation='relu’)(inputs)
h2 = tf.keras.layers.Dense(64, activation='relu’)(h1)
h2 = tf.keras.layers.Dense(64, activation='relu’)(h2)
h2 = tf.keras.layers.Dense(64, activation='relu’)(h2)
h = h1+h2
outputs = tf.keras.layers.Dense(10 , activation=‘softmax’)(h)

model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Input layer returns the tensor with the input data

Use layers as functions operating on the tensors

Use any operation

Model packs everything into a Keras Model

Keras models

- Keras models have extra methods and attributes. summary() will print a summary of the model with all the tensor sizes and number of trainable parameters

```
model.summary()
```

```
Layer (type)                    Output Shape
            Param #
=================================================================
dense_1 (Dense)                 (None, 64)
            1344
_________________________________________________________________
dropout_1 (Dropout)             (None, 64)
            0
_________________________________________________________________
dense_2 (Dense)                 (None, 64)
            4160
_________________________________________________________________
dropout_2 (Dropout)             (None, 64)
            0
_________________________________________________________________
dense_3 (Dense)                 (None, 10)
            650
=================================================================
Total params: 6,154
Trainable params: 6,154
Non-trainable params: 0
```

Layers, Initializers, Regularizers

- Visit https://www.tensorflow.org/api_docs/python/tf/keras/ for a comprehensive list.

```
tf.keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

tf.keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

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

- For the initializer parameters you can use the Keras initializer functions:

```
tf.keras.initializers.Zeros()
tf.keras.initializers.Ones()
tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)
tf.keras.initializers.GlorotNormal(seed=None)
```

- For the regularizers you can use:

```
tf.keras.regularizers.l1(0.01)
tf.keras.regularizers.l2(0.01)
tf.keras.regularizers.l1_l2(0.01)
```

Model compilation

- The *compile* procedure configures the model for training

```
model.compile(optimizer=tf.keras.optimizers.Adam(0.01), loss='categorical_crossentropy', metrics=['accuracy'])
```

Any optimizer from **tf.keras.optimizers**
Their parameters include the learning rate

```
tf.keras.optimizers.SGD(0.01)
tf.keras.optimizers.Adam(0.01)
tf.keras.optimizers.RMSProp(0.01)
```

Any loss function from **tf.keras.losses**
Aliases are available as strings for common losses

```
'categorical_crossentropy'          # one-hot label
'sparse_categorical_crossentropy'   # integer labels
'mse'
```

A list of metrics to evaluate the model, from **tf.keras.metrics**

Model training

- The *fit* procedure trains the model

```
model.fit(x_train, y_train, batch_size=32, epochs=10)
```

Input training data

Training labels

Size of minibatch

Length of training.
1 epoch = as many iterations as to use the whole dataset once = N / batch_size iterations

Model evaluation

- The *evaluate* procedure tests the model with the metrics in *compile* and prints their values

```
model.evaluate(x_test, y_test)
```

Input test data

Test labels

Model prediction

- The *predict* procedure runs the trained model and returns the value of the output tensor for a given input

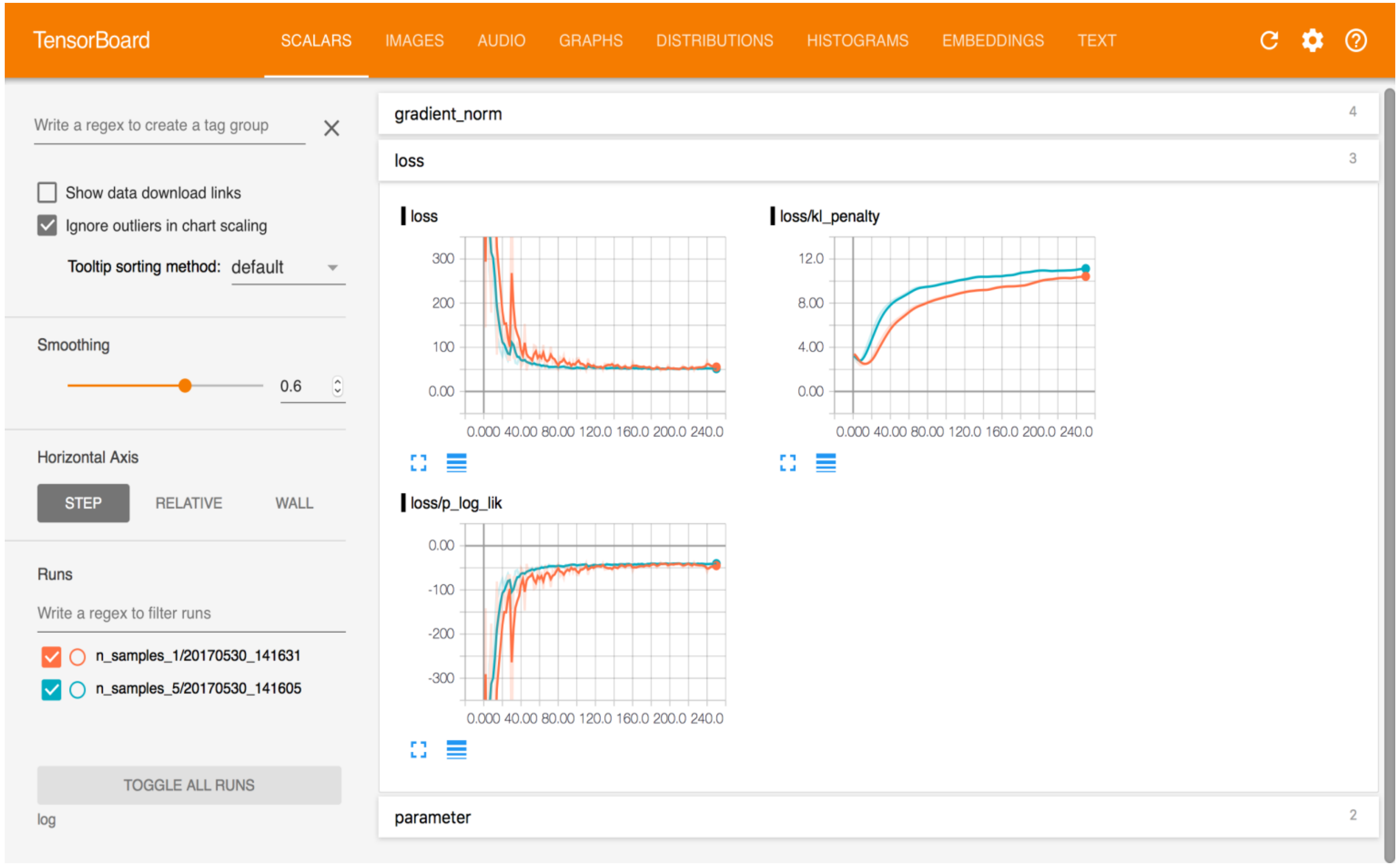
```
y = model.predict(x)
```

Value of output tensor

Input test data

Tensorboard

- Tensorboard is a **graphical interface** to visualize tensor values, statistics, the computational graph , …
- **Very useful for debugging**: keep track of the loss function during a long training, visualize a histogram of weight values
- You must **write instructions** in your program to **save** information to an **Event File** that is read by Tensorboard for visualization



Tensorboard

- To use Tensorboard to monitor training of Keras models you need to add a callback to the *fit* method

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir = logdir, update_freq = ‘batch’)
```

string with the directory where to save the log file

```
model.fit(x_train, y_train, batch_size=32, epochs=10, callbacks=[tensorboard_callback])
```

Data management

- How can I provide data to my model?

1. ***Keep all of them in RAM as Numpy arrays***

```
import numpy as np

x_train = np.random.random((1000, 128, 128, 3))
y_train = np.random.random((1000, 10))



model.fit(x_train, y_train, batch_size=16, epochs=10)
```

Data management

2. Use Tensorflow Datasets to build loading pipelines:

- ***Read data from disk as they are needed (or access RAM numpy arrays)***
- ***Shuffle data***
- ***Apply preprocessing operations (e.g. normalization, augmentation, …)***

```
dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
dataset = dataset.batch(16)

model.fit(dataset, epochs=10)
```

.from_tensor_slices() imports numpy arrays already in RAM

.batch() prepares an object represing a minibatch

Fit the Keras model passing the dataset batch object. Notice the batch size is already specified

Reading JPG images - Example

- Directory with many JPG images to be used for training:

Img.0001.4.jpg

Image number

Class label

STEP 1: create a tf.data.Dataset that knows which files to load

```
img_list = tf.data.Dataset.list_files('my_dir/*.jpg')
```

Reading JPG images - Example

STEP 2: create a loader function returning one image and its label

```
def loader(filename):

  label = int( tf.strings.split(filename, '.')[-2] )

  image = tf.io.read_file(filename)
  image = tf.image.decode_jpeg(image)
  image = tf.image.convert_image_dtype(image, tf.float32)

  image = tf.image.resize(image, [128, 128])

  return image, label
```

Parse the file name to extract the class label

Read jpeg file and convert it to tf.float32 tensor

Resize the image to a standard size

Reading JPG images - Example

STEP 3: *map* the loader onto the dataset and get a batch object

```
img_dataset = img_list.map(loader).batch(16)
```

.map() is Dataset method that allows to specify per-element transformations.
In this case: what to do with each file name (load, get label and resize image)

STEP 4: use it to train the model

```
model.fit(img_dataset, epochs=10)
```

Saving and loading Keras models

- **Saving a model in SavedModel format**
  - Creates a directory *my_model* containing some files
  - *saved_model.pb*: architecture, training config (optimizer, losses, …)
  - *variables*: weights values

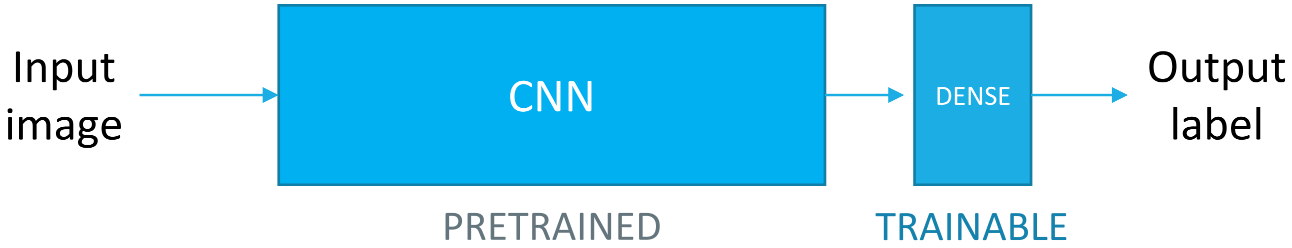
```
model.save('my_model')
```

- **Loading a Keras model**
  - The SavedModel format already has everything that is needed to run the model, no need to redefine layers, …

```
model = tf.keras.load_model('my_model')
```

Transfer learning

- Finetune a **pretrained** network to work on your specific problem
- Example:
  - take a CNN classifier trained on the huge ImageNet dataset
  - remove the last layer (softmax classifier)
  - replace it with a new layer for your problem
  - train only the new last layer



Pretrained models

- Tensorflow provides many pretrained models that can be easily imported

```
base_model = tf.keras.applications.MobileNetV2(input_shape=(128,128,3), include_top=False, weights='imagenet')
```

Import the MobileNetV2 pretrained network as a Keras Model

Size of the input tensors (no batch size)

False: remove the final classification layer

Pretrained on Imagenet dataset

```
base_model.trainable = False
```

Disable training of the weights

Check tf.keras.applications for more: https://www.tensorflow.org/api_docs/python/tf/keras/applications

Pretrained models

- Once imported use the Keras Models functional API to modify them

```
base_model = tf.keras.applications.MobileNetV2(input_shape=(128,128,3), include_top=False, weights='imagenet')
base_model.trainable = False
```

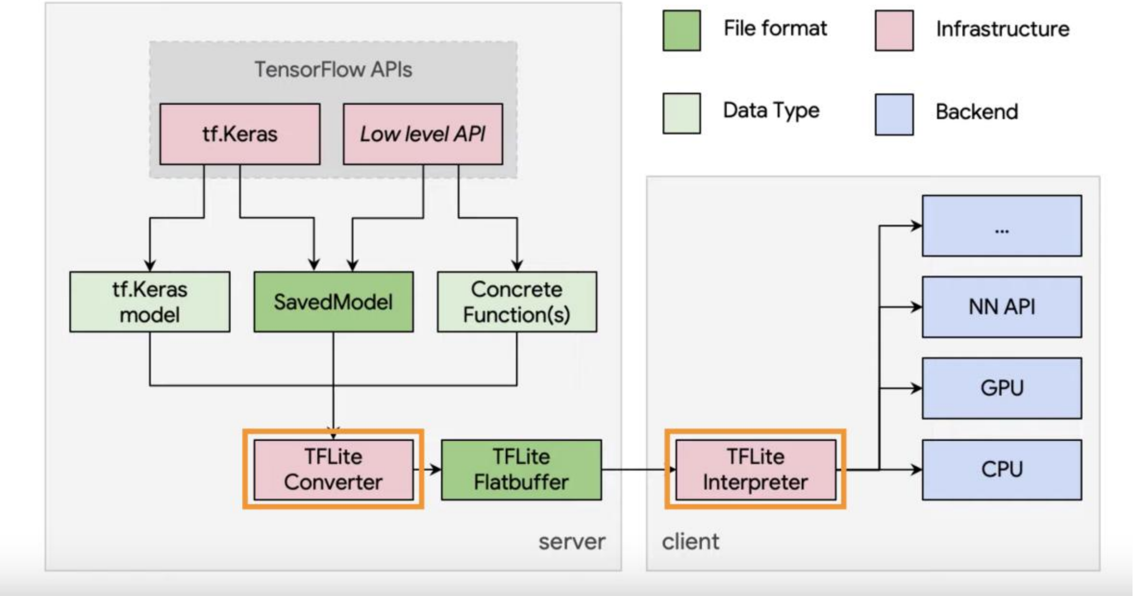
```
x = base_model.input
y = …
new_model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Get input tensor
Add more layers with functional API
Pack everything into a new model

TFLite

- Subset of Tensorflow with tools to use NNs on **embedded devices**

- **Tensorflow Lite model:** special efficient and portable format for models
    - .tflite extension
    - optimized for speed and storage size
    - a normal tensorflow/keras model can be converted to a TFLite model



Note: some Tensorflow functions cannot be ported to TFLite to keep the library small and efficient.
Rare case for some non-standard functions.

Converting a Keras model to TFLite

- Convert from a keras model object

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converted_model = converter.convert()
with open("converted_model.tflite", "wb") as f:
    f.write(converted_model)
```

- Convert from a saved keras model (SavedModel format)

```
converter = tf.lite.TFLiteConverter.from_saved_model(‘my_model_dir’)
converted_model = converter.convert()
with open("converted_model.tflite", "wb") as f:
    f.write(converted_model)
```

Model Quantization

- TFLite converter can also implement **weight quantization** via its optimizations options

- **Post-training quantization** example:
  - 8-bit quantization of only weights (float inference)

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
quantized_model = converter.convert()
```

Model Quantization

- **Fully integer quantization** (both weights and activations are int8)
  - Needed for embedded devices like Google Coral
  - Requires calibration data

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
quantized_model = converter.convert()
```

Example of how to provide calibration data.
Consult https://www.tensorflow.org/lite/performance/post_training_quantization

```
def representative_dataset():
  for data in dataset:
    yield {
      "image": data.image,
      "bias": data.bias,
    }
```

TFLite Inference

- How do I run the converted model?
- The TFLite model is executed through an **interpreter** (a tiny subset of the Tensorflow framework that can fit small devices)

```
interpreter = tf.lite.Interpreter(model_content = converted_model)
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

interpreter.set_tensor(input_details[0]['index'], input_data)

interpreter.invoke()

results = interpreter.get_tensor(output_details[0]['index'])
```

→ Create the interpreter and load the model

→ Prepares space for input and output tensors

→ Feed your input_data into the input tensor

→ Run the model

→ Read the output tensor

TFLite Inference – full example with dataset

- How to implement an equivalent of Keras "evaluate" (test over whole test set):

```
def evaluate(interpreter):

  input_index = interpreter.get_input_details()[0]["index"]
  output_index = interpreter.get_output_details()[0]["index"]
  input_format = interpreter.get_output_details()[0]['dtype']

  squared_error = []
  for tt in range(val_images.shape[0]):

    test_image = test_images[tt:(tt+1)]
    test_label = test_labels[tt:(tt+1)]

    interpreter.set_tensor(input_index, test_image)

    interpreter.invoke()
    output = interpreter.get_tensor(output_index)

    squared_error.append((test_label-output)**2)

  mean_squared_error = np.mean(squared_error)

  return mean_squared_error

interpreter = tf.lite.Interpreter('quantized_model.tflite')
interpreter.allocate_tensors()
test_mse = evaluate(interpreter)
```

Test set
test_images: (Nimages, H, W, C)
test_labels: (Nimages,)

Note
TFLite interpreter can only process **one sample** at a time, not a batch!

Quantization-aware training

- QAT can be easily performed with the **tensorflow_model_optimization** library

```
!pip install tensorflow_model_optimization
import tensorflow_model_optimization as tfmot

quant_aware_model = tfmot.quantization.keras.quantize_model(model)
quant_aware_model.fit(…)

converter = tf.lite.TFLiteConverter.from_keras_model(quant_aware_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
quantized_aware_model = converter.convert()
```

In Google Colab you need to install it

This can still be used as a normal Keras model

QAT

Convert to TfLite to actually quantize