



Deep Learning Libraries

Digital Systems M, Module 2
Matteo Poggi, Università di Bologna

Deep Learning Libraries

Deep Learning is one of the hottest topics in the last decade.

It allowed for unthinkable progresses in several fields such as image processing, text analysis, audio processing, etc.

Although deep learning itself is quite old (late '80), its popularity exploded in ~2014.

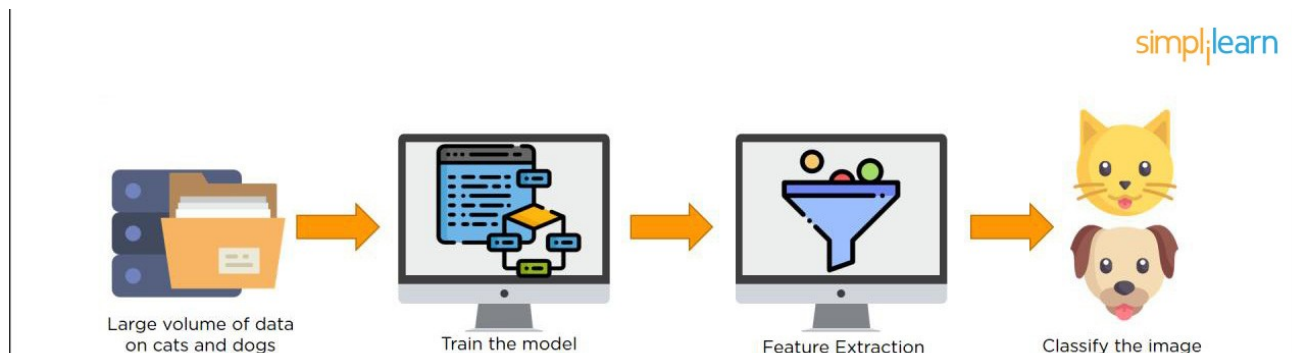


Deep Learning Libraries

Convolutional Neural Networks are among the most popular frameworks leveraging deep learning and can learn to solve several problems, broadly divided into **classification** and **regression** problems

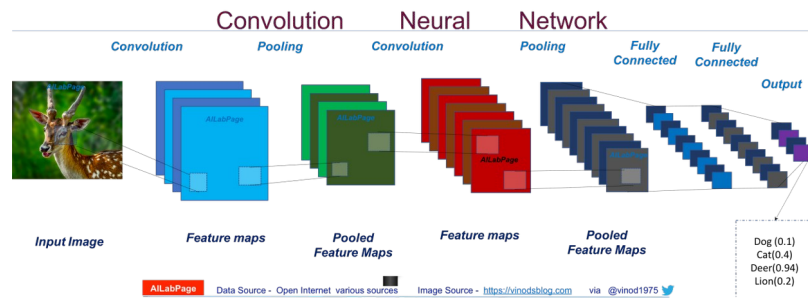
This is done through two steps: **training** and **testing** phases

During training, annotated images are iteratively provided to the network to learn how to solve the problem. During testing, the network is asked to process any possible image and to provide the correct answer to the specific problem



Deep Learning Libraries

A network is built by a sequence of modules, each one characterized by a set of parameters, called **weights**. Such weights define the behavior of each single module (e.g., the parameters in a convolutional kernel)



During the iterative training, the network processes an image and predicts a value, by passing it through any module sequentially (**forward pass**). Such value is compared to the real **label** assigned to such image.

By measuring the *difference* between the predicted value and the label, we can quantify the **error** and **optimize the network to minimize it**

Deep Learning Libraries

So, given a **differentiable** loss L , computed between the output (function of the weights of the network) and the label, we are able to minimize it iteratively by estimating the gradient of the loss and “moving” in the opposite direction.

In other terms, given the current state of the weights, we have to measure the error of the network prediction (this error is function of the weights), then update each weight in the opposite direction of the gradient.

$$\theta_j = \theta_j - \alpha \frac{\delta}{\delta \theta_j} \mathcal{L}(\theta) \quad \forall \theta_j \in \theta$$

α is the **learning rate**, the hyper-parameter that rules the “intensity” of the change.

Deep Learning Libraries

We can obtain automatically the partial derivatives that compose the gradient exploiting the chain rule of derivatives: given two functions of x , f and g , then the derivative of $f(g(x))$ is $f'(g(x)) * g'(x)$.

In other terms:

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta g} \frac{\delta g}{\delta x}$$

We can use this property to obtain the partial derivative of each weight with respect to the loss function and update any weight (**backward pass**)

Handwritten digit recognition

The very first problem tackled with Convolutional Neural Networks consists into recognizing handwritten digit.

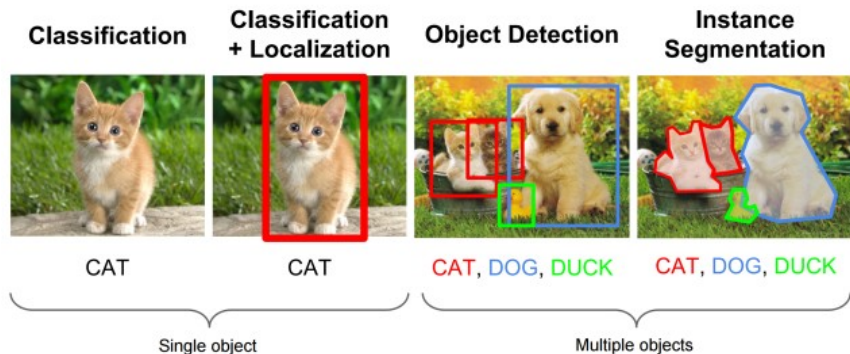
The network is trained to classify handwritten digit into 10 classes, representative of numbers between 0 and 9. Since the number of possible outputs is finite, this represents a **classification problem**

In this case, the network is trained to predict N outputs (N=classes), each one encoding the probability of the input to belong to a certain class

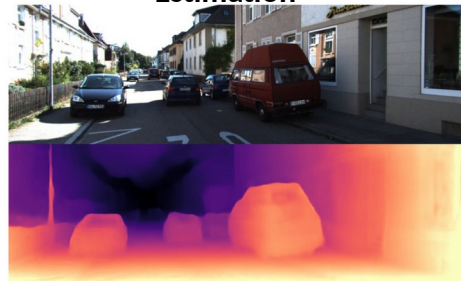


Deep Learning Libraries

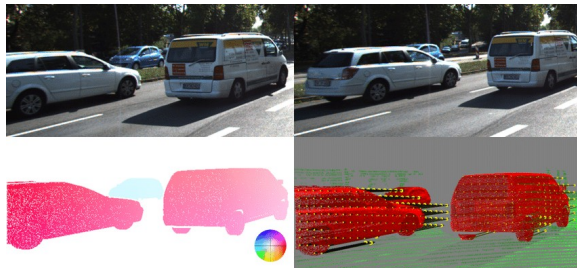
From the two broad families of problems, **classification** vs **regression**, we can distinguish a number of different tasks



Depth
Estimation



Motion (optical flow)
Estimation



Deep Learning Libraries

A number of software frameworks allowing for implementation of deep learning solutions exists

In this lecture, we will have a look to two of the most popular libraries, **Tensorflow** and **PyTorch**

We will see how to use basic functionalities in both libraries, how to implement some very basic training routines and how to use off-the-shelf networks, already trained by others for us



Tensorflow

Deep Learning Libraries

Tensorflow is an open source machine learning and deep learning framework developed by Google.

It allows to develop machine learning models, and it has a great support also for deploying.

Nowadays widely used in productions not only by Google but also by many other companies

Installed via pip



Deep Learning Libraries

TensorFlow offers API for Python, Java and C/C++

Two main versions of Tensorflow exist: 1.x and 2.x

In TensorFlow 1.x, the writing of code was divided into two phases: building the computational **graph** and then creating a session to **execute** it.

Tensorflow 2.x uses **Eager Execution** to avoid the need for the two phases.

Models trained with TensorFlow may be ported on mobile devices (iOS and Android) using a lightweight version of TensorFlow called **tf-lite**

What is a tensor?

Informally, a tensor is a multi-dimensional array

- A 0-D tensor is a scalar
- A 1-D tensor is an array
- A 2-D tensor is a matrix
- A 3-D tensor is an array of matrices
- ...

Deep Learning Libraries



We can think an image as a 3-D tensor with height H , width W and channel dimensions C

Graph

The graph specifies what operations perform, and how the tensors have to flow from the inputs in order to generate the desired output

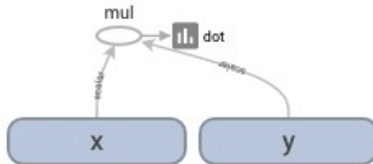
Internally, the compiler can exploit pruning operations to cut off unused branches of the graph, saving memory and computational time

Deep Learning Libraries

Suppose we have two scalar tensors, **x** and **y**, and we need to compute the dot product between them

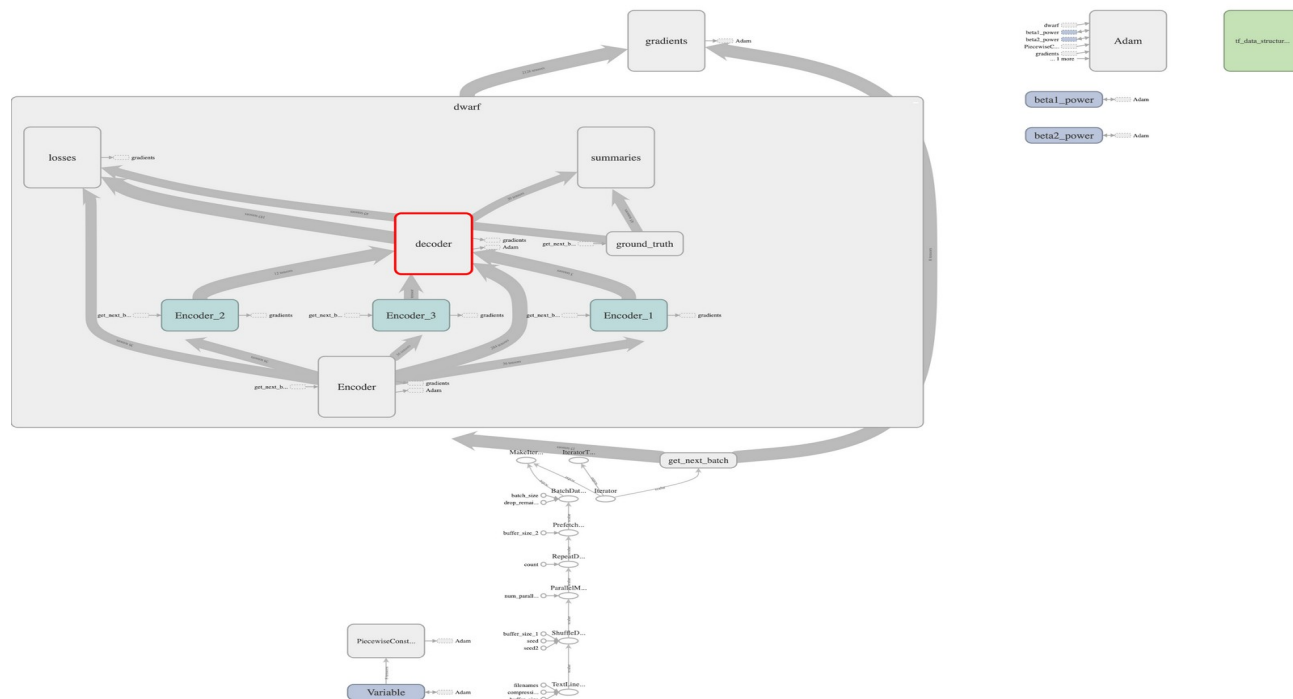
```
x = tf.Variable(5, dtype=tf.float32, name='x')  
y = tf.Variable(2, dtype=tf.float32, name='y')  
dot = x * y
```

The resulting graph is



Deep Learning Libraries

Of course, we can build more complex graphs



Deep Learning Libraries

In the graph, we have edges and nodes:

- edges: they represent tensors that flow between nodes
- nodes: they are `tf.Operation` added to the graph. They take zero or more input tensors and generate zero or more output tensors.

In the dot example, **x,y** and **dot** are Tensors, while **mul** is an Operation

Deep Learning Libraries

In the previous example we used the `tf.Variable` to represent both **x** and **y**

In TensorFlow we can represent data using various types of tensors (extending a common class, **Tensor**):

- **Variable**: tensor whose value may change by running operations on it
- **Constant**: it creates a constant tensor
- **Placeholders** (TF 1.x only): used to feed the graph with input values during the execution

Deep Learning Libraries

The **rank** of a Tensor is the number of dimensions, while its **shape** is the number of elements in each dimension

Rank	Entity	Shape	Example
0	Scalar	[]	A single number
1	Array	[N0]	A list of numbers
2	Matrix	[N0,N1]	A matrix
3	3-Tensor	[N0, N1, N2]	An image
4	4-Tensor	[N0, N1, N2, N3]	A batch of images

Deep Learning Libraries

```
x = tf.Variable(5, dtype=tf.float32, name='x')  
y = tf.Variable(2, dtype=tf.float32, name='y')
```

What is the rank of x? And of y?

To answer these questions, just print the two ranks and look at the results!

```
rank_x = tf.rank(x)  
rank_y = tf.rank(y)  
print('rank x:{}'.format(rank_x)) # rank x:Tensor("Rank:0", shape=(), dtype=int32)  
print('rank y:{}'.format(rank_y)) # rank y:Tensor("Rank_1:0", shape=(), dtype=int32)
```

Deep Learning Libraries

Although not what we expected, the previous result makes sense

Indeed, we have asked to TensorFlow the rank of the two Tensors, and it added to the graph the operation to get it

What we printed out was the shape of the resulting ranking Tensor (generated by the rank operation) and **not the rank value**!

To get the rank, we need to **execute** the graph: at the end of the execution, the ranking tensor will assume the rank value

Execution

In TensorFlow 1.x, the execution of the graph have to be performed inside a Session.

The Session takes a graph and it executes all the Operations from the starting point up to the desired set of Tensor we want to evaluate.

If no graph is selected, the default graph would be used.

Deep Learning Libraries

In TF 1.x, we can create a new Session using the **with** statement, in order to open and close automatically the Session (releasing resources at the end):

```
x = tf.Variable(5, dtype=tf.float32, name='x')
y = tf.Variable(2, dtype=tf.float32, name='y')
dot = x * y

config = tf.ConfigProto(allow_soft_placement=True)

with tf.Session(config=config) as session:
    session.run(tf.global_variables_initializer())
    session.run(tf.local_variables_initializer())
    dot_value = session.run(dot)
    print(dot_value) # 10.0
    print(dot) # Tensor("mul:0", shape=(), dtype=float32)
    print(type(dot_value)) # <class 'numpy.float32'>
    print(type(dot)) # <class 'tensorflow.python.framework.ops.Tensor'>
```


Deep Learning Libraries

In TF 2.x, Sessions are **gone**. By calling for an operation between variables, we directly process it thanks to Eager Execution.

```
x = tf.Variable(5, dtype=tf.float32, name='x')
y = tf.Variable(2, dtype=tf.float32, name='y')
dot = x * y
```

We can print the result as a TF Tensor...

```
>>> print(y)
tf.Tensor(25, shape=(), dtype=int32)
>>>
```

... or directly convert it into numpy structures!

```
>>> print(y.numpy())
25
>>>
```

Deep Learning Libraries

To summarize:

- TensorFlow has got **two phases**: the graph creation and its execution
- In 1.x, some operations are performed during the first phase, while others during the second one.
- In 2.x, Eager execution allows to run the graph without an explicit session

An example of this dualism is a piece of code inside an **if statement**: in case of a static graph construction (1.x) if we evaluate the if construct during the graph construction, that block will add or not a branch in the graph, but at runtime the new branch will be always executed (if it has been added) or never (if the condition was not satisfied during the graph creation).

If you need to check a condition at runtime, you have to use the **tf.cond** in 1.x. In 2.x, the graph is dynamic so we can use standard if constructs from python.

Shapes

Another example of this dualism concerns the shape: to get the shape of a Tensor we can run the following operations

```
t = tf.Variable([[[5.4],[1.0]]], dtype=tf.float32, name='t')
tf_shape = tf.shape(t)
t_shape = t.shape
```

Why two operations? Are them the same? Let's look what they print out

```
t = tf.Variable([[[5.4],[1.0]]], dtype=tf.float32, name='t')
tf_shape = tf.shape(t)
t_shape = t.shape
print('tf.shape: {}'.format(tf_shape)) # tf.shape: Tensor("Shape:0", shape=(3,), dtype=int32)
print('t.shape: {}'.format(t_shape))   # t.shape: (1, 2, 1)
```

So, they are not the same since the first returns a Tensor while the other a list

Deep Learning Libraries

However, running the Tensor we are getting the right shape

```
shape = session.run(tf_shape)
print('shape:{} of type {}'.format(shape, type(shape))) # shape:[1,2,1] of type <class 'numpy.ndarray'>
```

What is happening here is that we are asking for two different shapes: the static and the dynamic

The former is the shape used at graph creation, while the latter is the shape assumed by the tensors when we run the graph in the session.

Indeed, the two may **change** during different runs of the same code (while the tensor **rank** remains constant)

Creating a network

Tensorflow itself comes together with another library, **Keras**.

Keras is a front-end library providing high-level functions to implement several components to be used to build a neural network (convolutions, pooling, etc)

It is compatible with several back-end frameworks (Theano, or Tensorflow itself)

```
from tensorflow import keras
```

We can implement networks using Tensorflow low-level functions as well, however we need to handle several more details (we won't see this in the lecture)

Deep Learning Libraries

Some examples of layers ready for use:

- `tf.keras.layers.Conv2D(filters, kernel_size)`
- `tf.keras.layers.AveragePooling2D(pool_size)`
- `tf.keras.layers.MaxPool2D(pool_size)`
- `tf.keras.layers.Dense(units)`
- ...

Deep Learning Libraries

We can implement a neural network as a sequence (**tf.keras.models.Sequential** object) of several components (**tf.keras.Module**)

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

In these few lines, we implemented a network made of two fully connected layers, with 512 and 10 neurons, followed respectively by a ReLU and a Softmax layer.

Deep Learning Libraries

We then need to define a few details about training, such as the optimization method, the loss function and which metrics to compute to measure performance. Then, we can train the network giving the input data (`x_train`) and their labels (`y_train`) by calling the **fit** function (do you remember scipy?)

```
model.compile(optimizer='adam',  
              loss='sparse_categorical_crossentropy',  
              metrics=['accuracy'])  
  
model.fit(x_train, y_train, epochs=5)
```

We can then evaluate the network accuracy with the **evaluate** function

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

To run inference during deployment, we can use **model.predict**

Deep Learning Libraries

We can also handle training at lower level, defining our own **train cycle**. First, we create our network, define the optimizer and the loss function

```
# Get model
inputs = keras.Input(shape=(784,), name="digits")
x = layers.Dense(64, activation="relu", name="dense_1")(inputs)
x = layers.Dense(64, activation="relu", name="dense_2")(x)
outputs = layers.Dense(10, name="predictions")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

# Instantiate an optimizer to train the model.
optimizer = keras.optimizers.SGD(learning_rate=1e-3)

# Instantiate a loss function.
loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

Then, we iterate over training data and update our model

```
epochs = 2
for epoch in range(epochs):
    print("\nStart of epoch %d" % (epoch,))
    start_time = time.time()

    # Iterate over the batches of the dataset.
    for step, (x_batch_train, y_batch_train) in enumerate(train_dataset):
        with tf.GradientTape() as tape:
            logits = model(x_batch_train, training=True)
            loss_value = loss_fn(y_batch_train, logits)
            grads = tape.gradient(loss_value, model.trainable_weights)
            optimizer.apply_gradients(zip(grads, model.trainable_weights))
```

Deep Learning Libraries

In alternative, we can define our network and use it through more advanced paradigms. We can define our model as a class extending **tf.keras.Model**. When initialized, we define any layer in it

```
class MyModel(tf.keras.Model):  
    def __init__(self, num_classes=10):  
        super(MyModel, self).__init__(name='my_model')  
        self.dense_1 = layers.Dense(32, activation='relu')  
        self.dense_2 = layers.Dense(num_classes, activation='sigmoid')  
  
    def call(self, inputs):  
        # Define your forward pass here,  
        x = self.dense_1(inputs)  
        return self.dense_2(x)
```

Then, we implement the **call** function to specify the operations processed during inference. This paradigm is used by **PyTorch** as well

Finally, we can train our network as in the previous case through a custom cycle

Deep Learning Libraries

Once we have trained our model, we can save its weights with **`tf.keras.models.save_model`**, by specifying the model to save and a path where to store it

We can load a pre-trained model as well, with **`tf.keras.models.load_model`**

We may want to load a pre-trained model both to use it at deployment, as well as to **train it again** and improve its performance

Pre-existing networks

Since 2014, a few very popular network architectures become standard in most applications (they are also known as **foundation models**)

Keras provides easy access to most of them through the **applications** module

- `tf.keras.applications.VGG16`
- `tf.keras.applications.ResNet50`
- `tf.keras.applications.MobileNet`
- ...

Transfer Learning

We can exploit the knowledge learned from large datasets in our applications, by creating a foundation model and loading the weights saved after a training on **ImageNet**

```
base_model = tf.keras.applications.SequentialMobileNetV2(  
    input_shape=(160, 160, 3),  
    include_top=False,  
    weights='imagenet')  
  
base_model.trainable = False
```

We can change the final layer to fit with our problem and train it alone

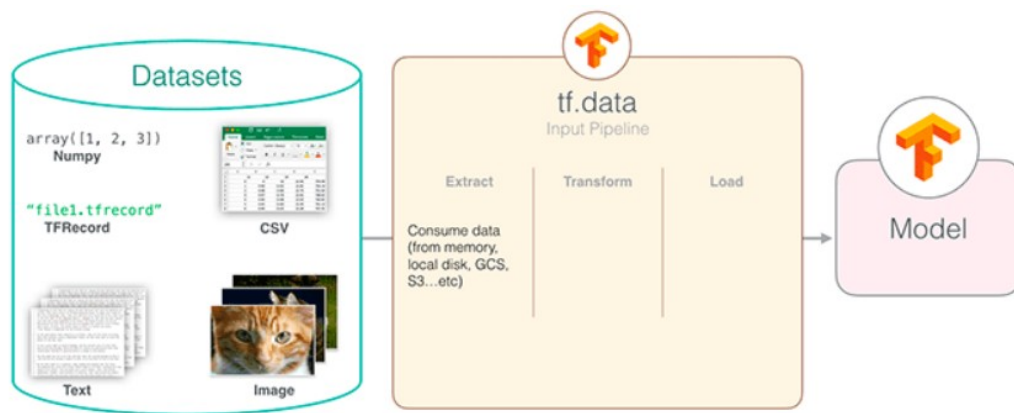
```
model = tf.keras.models.Sequential([  
    base_model,  
    tf.keras.layers.GlobalAveragePooling2D(),  
    tf.keras.layers.Dense(1)  
])  
# Compile and fit
```

Deep Learning Libraries

Reading data

Tensorflow provides **tf.data** to build data pipeline, comprising data reading, transformation and loading into Tensorflow models

Keras also provides **ImageGenerator** interface for the same purpose



Deep Learning Libraries

Given a list of training inputs and labels, we can create our training pipeline as

```
dataset = tf.data.Dataset.from_tensor_slices((trainX, trainY))
# build the data input pipeline
print("[INFO] creating a tf.data input pipeline..")
dataset = (dataset
    .shuffle(1024)
    .cache()
    .repeat()
    .batch(32)
    .prefetch(AUTOTUNE)
)
```

In alternative, we can pass a list of file paths, read them from disk and apply transformations on them within a **map** call

```
trainDS = tf.data.Dataset.from_tensor_slices(trainPaths)
trainDS = (trainDS
    .shuffle(len(trainPaths))
    .map(load_images, num_parallel_calls=AUTOTUNE)
    .map(augment, num_parallel_calls=AUTOTUNE)
    .cache()
    .batch(32)
    .prefetch(AUTOTUNE)
)
```

```
def load_images(imagePath):
    # read the image from disk, decode it, convert the data type to
    # floating point, and resize it
    image = tf.io.read_file(imagePath)
    image = tf.image.decode_png(image, channels=3)
    image = tf.image.convert_image_dtype(image, dtype=tf.float32)
    image = tf.image.resize(image, config.IMAGE_SIZE)
    # parse the class label from the file path
    label = tf.strings.split(imagePath, os.path.sep)[-2]
    label = tf.strings.to_number(label, tf.int32)

    # return the image and the label
    return (image, label)
```

```
def augment(image, label):
    # perform random horizontal and vertical flips
    image = tf.image.random_flip_up_down(image)
    image = tf.image.random_flip_left_right(image)
    # return the image and the label
    return (image, label)
```



PyTorch

Deep Learning Libraries

PyTorch is an open source machine learning and deep learning framework developed by Facebook AI.



Launched in 2016, it represents the natural evolution of another major deep learning framework, called **Torch**.

PyTorch extends Torch with Python interfaces (Torch used LUA as main language).

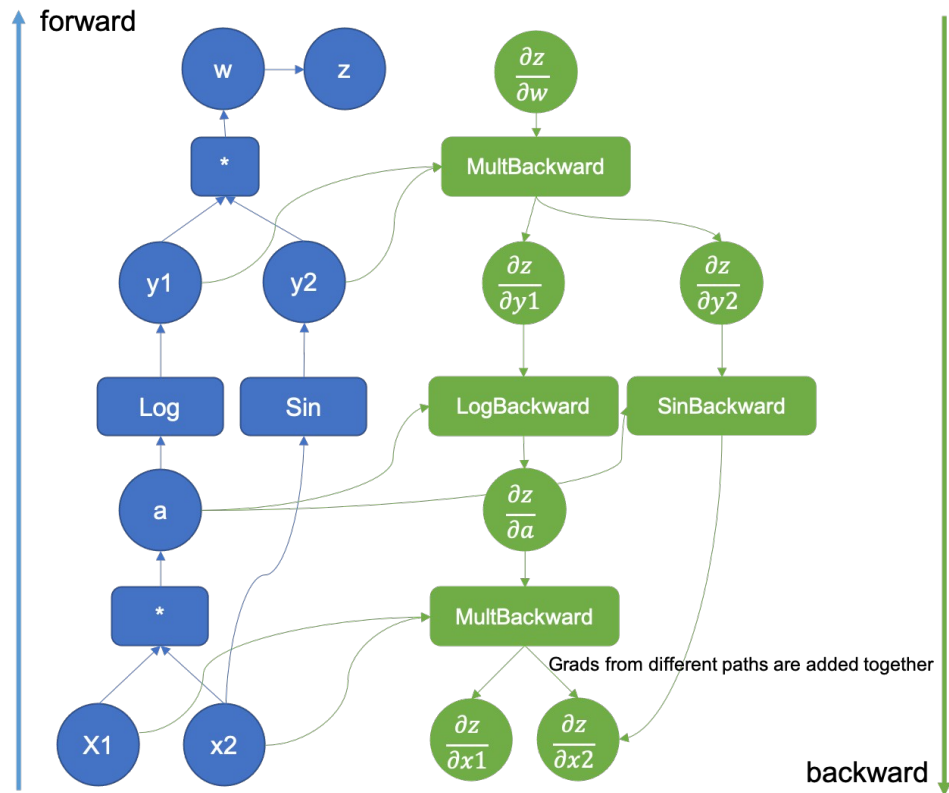
PyTorch builds a **dynamic graph**, not requiring the two phases as defined in Tensorflow 1.x. It also allows for portability to mobile devices using **torchscript**

Installed via pip

Deep Learning Libraries

PyTorch itself relies on a computational graph.

It is implemented similarly to Tensorflow graph, allowing to easily keep track of dependencies and for automatic gradient computation and propagation



Deep Learning Libraries

Most of PyTorch functionalities are very similar to what available in Tensorflow 2.x (actually, PyTorch came first...)

PyTorch main data structure is the **Tensor** class. We can operate on tensors as we do in Tensorflow (e.g., summing them, printing their shape, converting them into numpy data)

```
>>> a = torch.Tensor([3])
>>> b = torch.Tensor([4])
>>> print(a+b)
tensor([7.])
>>>
```

```
>>> a.shape
torch.Size([1])
>>>
>>> a.size()
torch.Size([1])
>>>
```

```
>>> print((a+b).numpy())
[7.]
>>>
```

No session in PyTorch!

Creating a network

PyTorch does not provide high-level APIs equivalent to **Keras**.

Most layers are implemented under **torch.nn** package:

- `nn.conv2D(in_channels, out_channels, kernel_size)`
- `nn.AvgPool2D(kernels_size)`
- `nn.MaxPool2D(kernel_size)`
- `nn.Linear(in_features, out_features)`
- ...

Deep Learning Libraries

Similarly to Keras, to define a network we implement a class extending **nn.Module**

In the class constructor, we define the network architecture. We can stack layers one after the other inside a **nn.Sequential** object

The **forward** function defines the operations performed by the network during inference (as the **call** function does in Keras)

```
class LeNet5(nn.Module):

    def __init__(self, n_classes):
        super(LeNet5, self).__init__()

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, stride=1),
            nn.Tanh()
        )

        self.classifier = nn.Sequential(
            nn.Linear(in_features=120, out_features=84),
            nn.Tanh(),
            nn.Linear(in_features=84, out_features=n_classes),
        )
```

```
def forward(self, x):
    x = self.feature_extractor(x)
    x = torch.flatten(x, 1)
    logits = self.classifier(x)
    probs = F.softmax(logits, dim=1)
    return logits, probs
```

Deep Learning Libraries

Some specific layers behave differently during **training** and **testing**

For instance, some **normalization layers** accumulate statistics during training, then they fix them and keep them unchanged once the training is over

We can easily configure a model for training or testing behaviors (and, automatically, configure any of its layer accordingly) by calling the **train()** and **eval()** of the model object itself

Deep Learning Libraries

Implementing a training loop is, again, very similar to what we have already seen with Tensorflow

Torch.optim provides standard optimizers, while torch.nn.functional (F) already implements several standard loss functions

```
optimizer
params = [encoder.parameters(), decoder.parameters()]
optimizer = torch.optim.Adam(params, lr=1e-3)

TRAIN LOOP
model.train()
num_epochs = 1
for epoch in range(num_epochs):
    For train_batch in mnist_train:
        x, y = train_batch
        x = x.cuda(0)
        x = x.view(x.size(0), -1)
        z = encoder(x)
        x_hat = decoder(z)
        loss = F.mse_loss(x_hat, x)
        print('train loss: ', loss.item())

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Some higher-level utils, implemented by **PyTorch lightning**, allow to define training/validation logic when writing the network class, by extending **LightningModule** instead of **nn.Module**

```
model = LitAutoEncoder()
trainer = pl.Trainer()
trainer.fit(model, mnist_train, mnist_val)
```

Deep Learning Libraries

Once we have trained our model, we can save it (in particular, its **state_dict**, containing its weights) with **torch.save** by specifying the model state dictionary (**model.state_dict**) and a path to a specific file

The model itself can be restored when starting a new training or when running the testing phase. To this aim, we load the dictionary file with **torch.load** and set it as the state dictionary of a model object through the **load_state_dict** function

```
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
```

Higher level loading functions are provided by **PyTorch lightning** or **Torchscript**

Deep Learning Libraries

Torchvision is a library part of the PyTorch project, providing several interfaces to popular datasets, image transformation functionalities and **model architectures**

From **torchvision.models** we can use foundation models as well

- `models.vgg16`
- `models.resnet18`
- ...

```
""" Resnet18
"""

model_ft = models.resnet18(pretrained=use_pretrained)
set_parameter_requires_grad(model_ft, feature_extract)
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, num_classes)
input_size = 224
```

We can instantiate a model, replace its output and train it on our own data!

Deep Learning Libraries

PyTorch also allows to retrieve model definitions (and weights) remotely thanks to **torch.hub**

We can do this by means of the **load** function and specifying repository and model names, as well as if we want a pretrained model or not. By default, torch.hub retrieves repositories from github

```
# Model
model = torch.hub.load('ultralytics/yolov5', 'yolov5s', pretrained=True)

# Images
imgs = ['https://ultralytics.com/images/zidane.jpg'] # batch of images

# Inference
results = model(imgs)
```

Deep Learning Libraries

To implement data pipelines, PyTorch provides **Dataset** and **Dataloader** classes.

To implement our own pipeline, we extend the Dataset class and implement the `__getitem__` method

Then, we create an object with our own Dataset class and then pass it to a DataLoader object, which will give us any element by iterating over it

```
class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

```
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```



Deep learning utilities

Deep Learning Libraries

A few tools can greatly ease our life when training neural networks

Google colab (colab.research.google.com) is a cloud service allowing to run python code on your browser. It gives access to hardware resources (GPUs, TPUs, etc.) and to shared files (over Google Drive), with an interface similar to Jupyter Notebook

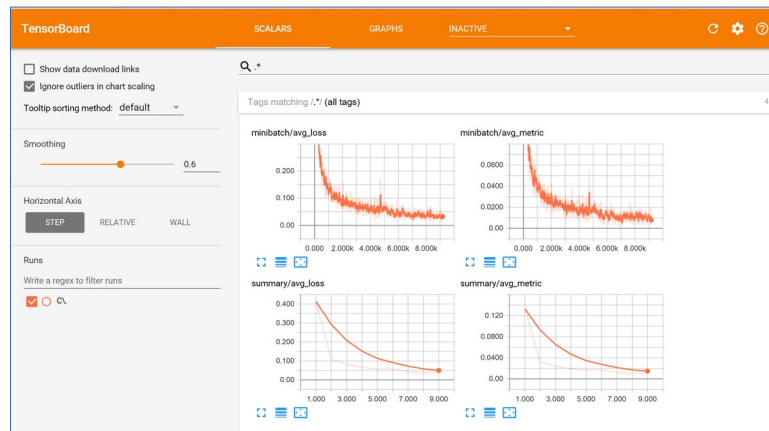
Kaggle (kaggle.com) is a large-scale repository collecting code and data. It allows for using more than 50K public datasets and 400K notebooks to play with machine learning and data

Deep Learning Libraries

Tensorboard is a python package, coupled with Tensorflow, allowing to easily keep track of the training behavior of a network

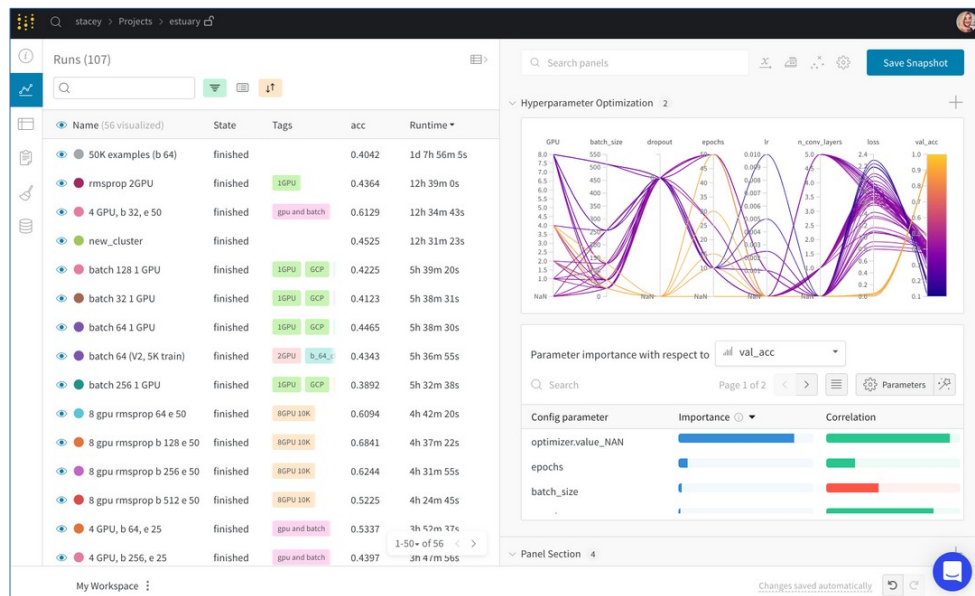
It provides functions to plot statistics (loss values, accuracy) as well as to store image examples. Recently, it has been added to PyTorch as well, in the **utils** package

Logging is performed directly from python code by means of a **SummaryWriter** object, then a Tensorboard server can be launched and visualized over a web browser



Deep Learning Libraries

Weights & Biases (wandb.ai) allows to keep track of many experiments in a centralized server. As for Tensorboard, logging is performed through python APIs provided by the **wandb** package, after logging in. Then, logs are directly hosted on wandb.ai





Mobile Deep Learning

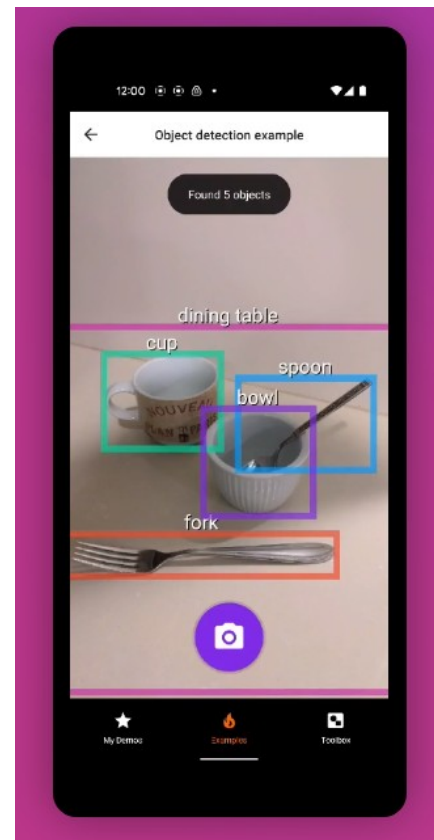
Deep Learning Libraries

In the last 5-6 years deep learning applications appeared in several forms (advertisement, augmented reality apps, etc.)

Inevitably, they landed on the most popular consumer devices, **mobile phones**

Applications leveraging deep learning on consumer come as products – thus, we expect to install and run them without requiring additional efforts

The focus is on the **testing phase**, while the networks are still trained on standard hardware (desktop PC With GPUs, cloud servers, etc.)



Deep Learning Libraries

Support to **inference** on mobile devices is increasing over time

Both Tensorflow and PyTorch provides packages specifically designed for deployment on mobile phones

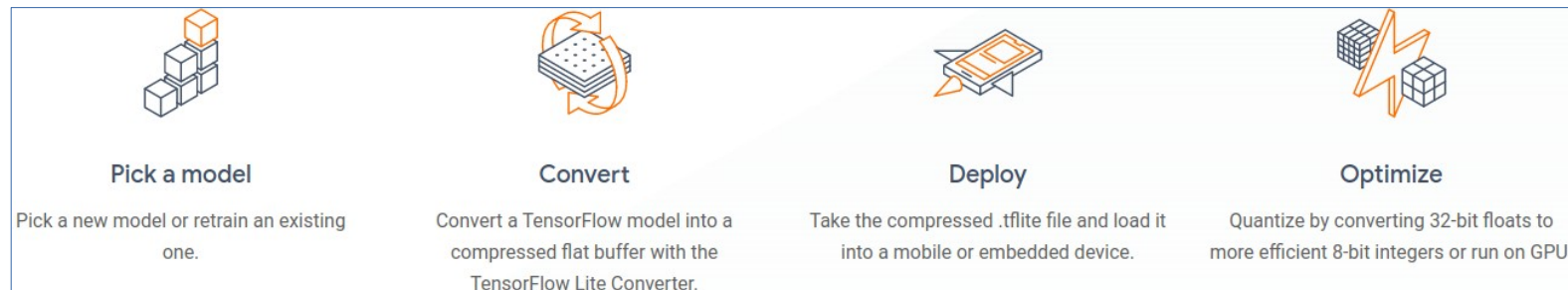
Right now, the standard frameworks used to this aim are **Tensorflow Lite** and **PyTorch Mobile**

This field is particularly vivid and new solutions might appear soon
(Facebook – now Meta – announced **Pytorch-live** on November 25...)

Deep Learning Libraries

Tensorflow Lite is a deep learning framework, tightly coupled with Tensorflow, designed for on-device inference

It provides functionalities specific for the **deployment** stage (after training!) to improve portability and efficiency



Deep Learning Libraries

The first step consists into converting a Tensorflow model into lite format

This can be done for both networks written with low level APIs or using Keras

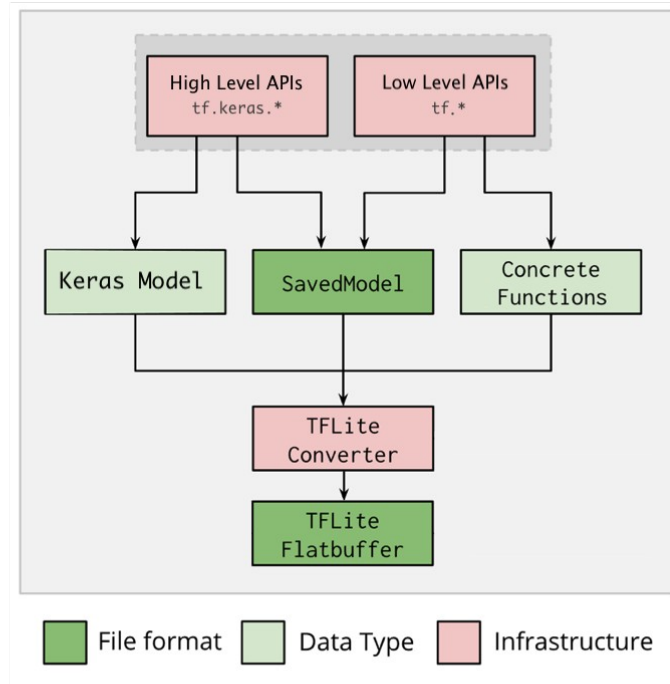
Concerning Keras models, we can either convert a model from disk or one we have already loaded

```
import tensorflow as tf

# Convert the model
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
    f.write(tflite_model)
```

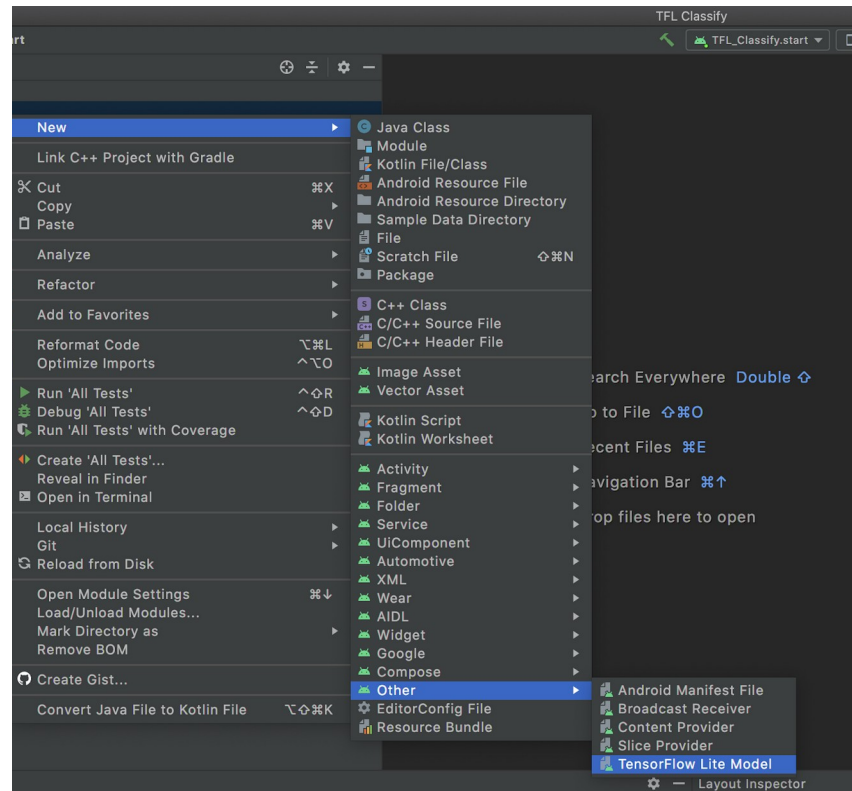
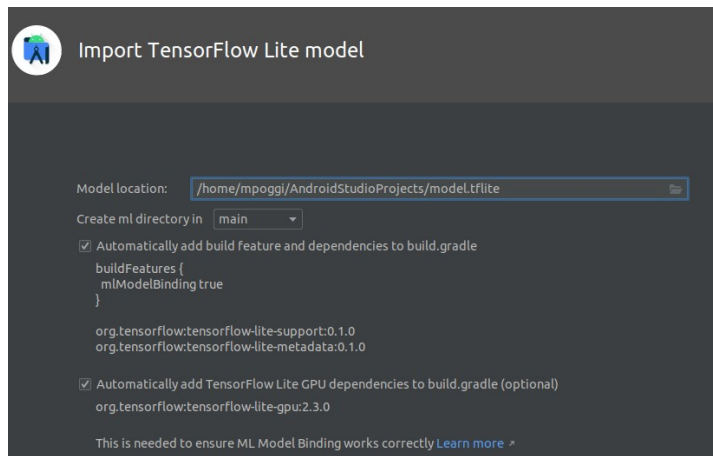
```
# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
```



Deep Learning Libraries

Tensorflow Lite **wizards** are available in the latest Android Studio versions

We can import models directly from the IDE and automatically update gradle files



Deep Learning Libraries

It also provides us a simple template to prepare data for inference, running the network and retrieving the results

Sample Code

```
Kotlin  Java
try {
    Model model = Model.newInstance(context);

    // Creates inputs for reference.
    TensorBuffer inputFeature0 = TensorBuffer.createFixedSize(new int[]{1, 224, 224, 3}, DataType.FLOAT32);
    inputFeature0.loadBuffer(byteBuffer);

    // Runs model inference and gets result.
    Model.Outputs outputs = model.process(inputFeature0);
    TensorBuffer outputFeature0 = outputs.getOutputFeature0AsTensorBuffer();

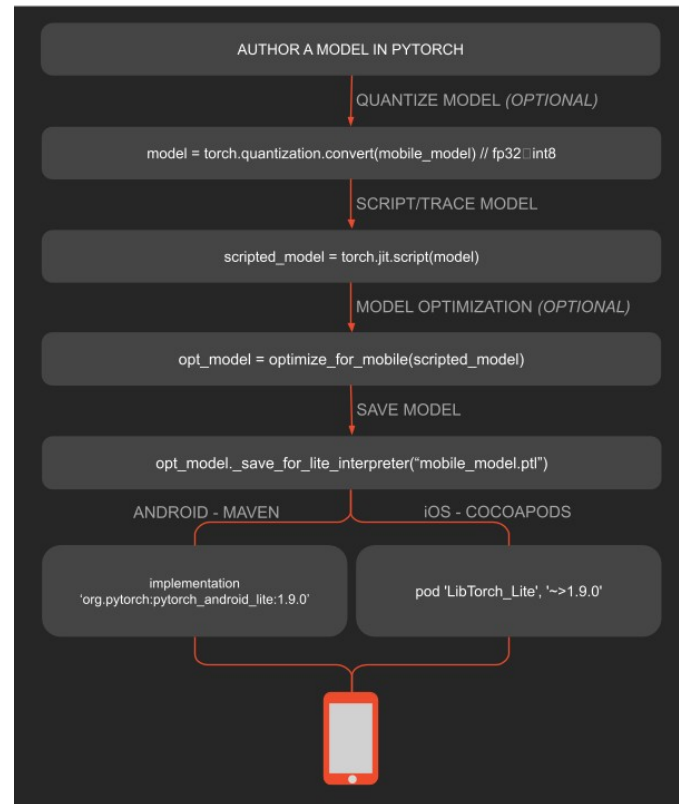
    // Releases model resources if no longer used.
    model.close();
} catch (IOException e) {
    // TODO Handle the exception
}
```

Of course, depending on the network and the task, our output **TensorBuffer** will have different shapes (1xN for classification, 1xHxWxN for segmentation , etc.)

Deep Learning Libraries

PyTorch mobile runtime allows for seamless deployment of trained models on mobile devices while remaining in the PyTorch ecosystem

Similarly to TFLite, it provides functionalities to optimize models, as well as efficient interpreters (**lite interpreter**)



Deep Learning Libraries

The first step, again, consists into converting a model into one optimized for mobile devices

A **mobile_optimizer** is provided by PyTorch to optimize the network and save it in a format compatible with the lite interpreter

```
import torch
import torchvision
from torch.utils.mobile_optimizer import optimize_for_mobile

model = torchvision.models.mobilenet_v2(pretrained=True)
model.eval()
example = torch.rand(1, 3, 224, 224)
traced_script_module = torch.jit.trace(model, example)
traced_script_module_optimized = optimize_for_mobile(traced_script_module)
traced_script_module_optimized._save_for_lite_interpreter("app/src/main/assets/model.ptl")
```

Then, we can embed our model into our Android application

Deep Learning Libraries

First, we add PyTorch dependency to the build.gradle file (and torchvision in case we use one of the foundation models provided by it)

```
implementation 'org.pytorch:pytorch_android:1.6.0'  
implementation 'org.pytorch:pytorch_android_torchvision:1.6.0'
```

as well as jcenter() among repositories

Deep Learning Libraries

On Java side, we create a **Module** object and load the converted model from the assets

```
try {
    this.module = Module.load(assetFilePath(context, this, "model.ptl"));
} catch (IOException e) {
    e.printStackTrace();
}
```

```
/**
 * Copies specified asset to the file in /files app directory and returns this file absolute path.
 *
 * @return absolute file path
 */
public static String assetFilePath(Context context, String assetName) throws IOException {
    File file = new File(context.getFilesDir(), assetName);
    if (file.exists() && file.length() > 0) {
        return file.getAbsolutePath();
    }
    try (InputStream is = context.getAssets().open(assetName)) {
        try (OutputStream os = new FileOutputStream(file)) {
            byte[] buffer = new byte[4 * 1024];
            int read;
            while ((read = is.read(buffer)) != -1) {
                os.write(buffer, 0, read);
            }
            os.flush();
        }
        return file.getAbsolutePath();
    }
}
```

Deep Learning Libraries

Finally, we can run the network by converting a **Bitmap** to a **Tensor** object and then passing it to the module

```
Tensor inputTensor = TensorImageUtils.bitmapToFloat32Tensor(conv,  
    TensorImageUtils.TORCHVISION_NORM_MEAN_RGB, TensorImageUtils.TORCHVISION_NORM_STD_RGB);  
  
Tensor outputTensor = this.module.forward(IValue.from(inputTensor)).toTensor();
```

Again, the shape of the output tensor depends on the task

Deep Learning Libraries

In case our network is a classification model, the output will contain N probability scores for the N classes. Then, we can look for the one class having the highest probability


```
float[] scores = outputTensor.getDataAsFloatArray();

float maxScore = -Float.MAX_VALUE;
int maxScoreIdx = -1;
for (int i = 0; i < scores.length; i++) {
    if (scores[i] > maxScore) {
        maxScore = scores[i];
        maxScoreIdx = i;
    }
}

String className = this.IMAGENET_CLASSES[maxScoreIdx];
```

We can check the class corresponding to the maximum probability id by looking over a look-up table

```
public static String[] IMAGENET_CLASSES = new String[]{
    "tench, Tinca tinca",
    "goldfish, Carassius auratus",
    "great white shark, white shark, man-eater, man-eating shark, Carcharodon carcharias",
```



Optimizations:
weights quantization

Model optimization consists into converting the set of operations performed by the trained network into others better suited for efficient inference

Several optimization strategies exist to reduce the computational cost required to run a neural network, with different degrees of complexity in terms of implementation (e.g., separable convolutions)

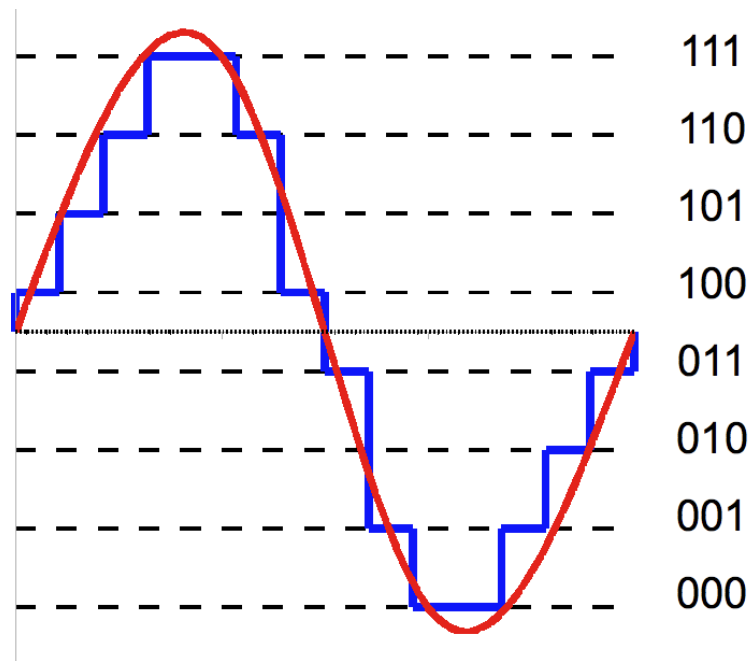
A standard (and simple) approach to optimize models consists into applying **quantization** to the weights

Deep Learning Libraries

Quantization is the process of constraining an input from a larger set of possible values (for instance, real numbers) to a smaller one (for instance, integers)

The main advantage of quantization is to require less data to store information (the fewer the possible values, the fewer the bits required to store them)

Moreover, **fixed point** arithmetic operations result much more efficient than **floating point** ones



Deep Learning Libraries

As we reduce the granularity of the values used by our network, its accuracy may reduce as well

Quantization supported by both **Tensorflow** and **PyTorch**. It is aimed at accelerating inference and is usually supported for **forward pass only**

A quantized model results in a faster model if appropriate hardware support is available (e.g., quantizing a model to float16 without proper support to float16 operations will have no impact on speed)

Tensorflow Lite supports **post-training quantization** in three ways

Dynamic range quantization is the simplest form of optimization, statically converting only weights from float32 to int8 (floating to fixed)

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_quant_model = converter.convert()
```

At inference, weights are converted from int8 to float32 (conversion is done once and cached to reduce latency)

Some operators (called **dynamic**) can quantize their activations as well based on a **dynamic range**, and thus run with 8bits operations

Full integer quantization converts the entire set of operations in the networks to int8 quantized operations

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
tflite_quant_model = converter.convert()
```

This requires to **calibrate** the min-max range of all floating point tensors in the model, which are **static**

For **dynamic tensors** (such as results of the inference), we cannot anticipate the min-max range, unless we have a **representative dataset** to estimate it

Float16 quantization converts to 16bit floating point values, allowing for minimal loss in accuracy and 2x inference speed

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]
tflite_quant_model = converter.convert()
```

Some delegates (e.g., GPU) run naively on float16 data, resulting much faster. However, if float16 hardware support is not available (e.g., on CPU), values are dequantized to float32, losing any gain. In general, the gain in terms of speed is lower than the one achieved by full int quantization

More to come (16bit activations with 8bit weights, ...)

PyTorch supports three main quantization strategies
(more details at <https://pytorch.org/docs/stable/quantization.html>)

Dynamic quantization stores pre-quantized weights and dynamically quantizes activations during inference

```
# create a quantized model instance  
model_int8 = torch.quantization.quantize_dynamic(  
    model_fp32, # the original model  
    {torch.nn.Linear}, # a set of layers to dynamically quantize  
    dtype=torch.qint8) # the target dtype for quantized weights
```

Deep Learning Libraries

Static quantization quantizes the weights and activations of the model, also fusing activations into preceding layers where possible

It requires **calibration** with a representative dataset to determine optimal quantization parameters for activations

```
# define a floating point model where some layers could be statically quantized
class M(torch.nn.Module):
    def __init__(self):
        super(M, self).__init__()
        # QuantStub converts tensors from floating point to quantized
        self.quant = torch.quantization.QuantStub()
        self.conv = torch.nn.Conv2d(1, 1, 1)
        self.relu = torch.nn.ReLU()
        # DeQuantStub converts tensors from quantized to floating point
        self.dequant = torch.quantization.DeQuantStub()

    def forward(self, x):
        # manually specify where tensors will be converted from floating
        # point to quantized in the quantized model
        x = self.quant(x)
        x = self.conv(x)
        x = self.relu(x)
        # manually specify where tensors will be converted from quantized
        # to floating point in the quantized model
        x = self.dequant(x)
        return x

# create a model instance
model_fp32 = M()

# model must be set to eval mode for static quantization logic to work
model_fp32.eval()

# attach a global qconfig, which contains information about what kind
# of observers to attach. Use 'fbgemm' for server inference and
# 'qnnpack' for mobile inference. Other quantization configurations such
# as selecting symmetric or asymmetric quantization and MinMax or L2Norm
# calibration techniques can be specified here.
model_fp32.qconfig = torch.quantization.get_default_qconfig('fbgemm')

# Fuse the activations to preceding layers, where applicable.
# This needs to be done manually depending on the model architecture.
# Common fusions include 'conv + relu' and 'conv + batchnorm + relu'
model_fp32_fused = torch.quantization.fuse_modules(model_fp32, [['conv', 'relu']])

# Prepare the model for static quantization. This inserts observers in
# the model that will observe activation tensors during calibration.
model_fp32_prepared = torch.quantization.prepare(model_fp32_fused)

# calibrate the prepared model to determine quantization parameters for activations
# in a real world setting, the calibration would be done with a representative dataset
input_fp32 = torch.randn(4, 1, 4, 4)
model_fp32_prepared(input_fp32)

# Convert the observed model to a quantized model. This does several things:
# quantizes the weights, computes and stores the scale and bias value to be
# used with each activation tensor, and replaces key operators with quantized
# implementations.
model_int8 = torch.quantization.convert(model_fp32_prepared)

# run the model, relevant calculations will happen in int8
res = model_int8(input_fp32)
```

Quantization-aware training (QAT) allows for modeling the effect of quantization while training the network, improving the final accuracy

During training, all calculations are performed with float32 values and some modules simulate the effect of int8 operations by clamping and/or rounding values

It usually leads to more accurate networks compared to static quantization

```
import torch

# define a floating point model where some layers could benefit from QAT
class M(torch.nn.Module):
    def __init__(self):
        super(M, self).__init__()
        # QuantStub converts tensors from floating point to quantized
        self.quant = torch.quantization.QuantStub()
        self.conv = torch.nn.Conv2d(1, 1, 1)
        self.bn = torch.nn.BatchNorm2d(1)
        self.relu = torch.nn.ReLU()
        # DeQuantStub converts tensors from quantized to floating point
        self.dequant = torch.quantization.DeQuantStub()

    def forward(self, x):
        x = self.quant(x)
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        x = self.dequant(x)
        return x

# create a model instance
model_fp32 = M()

# model must be set to train mode for QAT logic to work
model_fp32.train()

# attach a global qconfig, which contains information about what kind
# of observers to attach. Use 'fbgemm' for server inference and
# 'qnnpack' for mobile inference. Other quantization configurations such
# as selecting symmetric or asymmetric quantization and MinMax or L2Norm
# calibration techniques can be specified here.
model_fp32.qconfig = torch.quantization.get_default_qat_qconfig('fbgemm')

# fuse the activations to preceding layers, where applicable
# this needs to be done manually depending on the model architecture
model_fp32_fused = torch.quantization.fuse_modules(model_fp32,
                                                    [['conv', 'bn', 'relu']])

# Prepare the model for QAT. This inserts observers and fake_quants in
# the model that will observe weight and activation tensors during calibration.
model_fp32_prepared = torch.quantization.prepare_qat(model_fp32_fused)

# run the training loop (not shown)
training_loop(model_fp32_prepared)

# Convert the observed model to a quantized model. This does several things:
# quantizes the weights, computes and stores the scale and bias value to be
# used with each activation tensor, fuses modules where appropriate,
# and replaces key operators with quantized implementations.
model_fp32_prepared.eval()
model_int8 = torch.quantization.convert(model_fp32_prepared)
```