Deep Learning - lab 3

TensorFlow and Keras

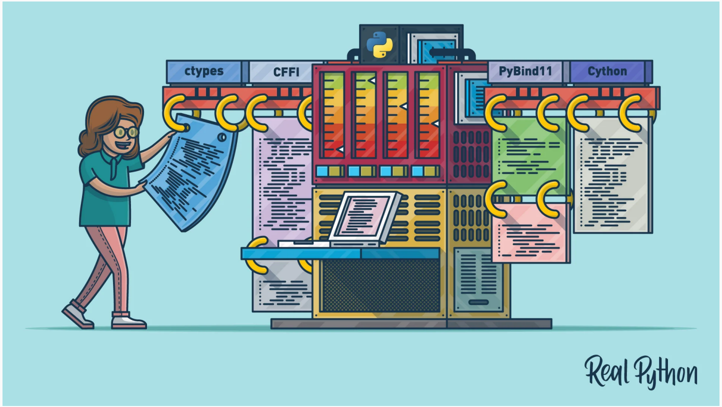
Prof. Stefano Carrazza

University of Milan and INFN Milan

Extending Python

Extending Python with compilation

- ctypes
- PyObjects
- CFFI
- pybind11
- Cython
- Numba



Extending Python with compilation

Does deep learning require low level programming knowledge?

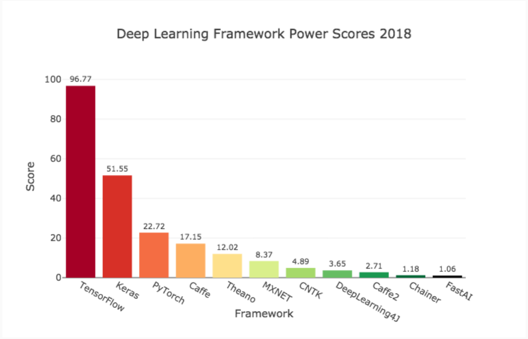
Depends:

- No: we can use frameworks based on compiled code.
- Yes: we can extend frameworks with custom operations (operators).

Which DL framework?

Make a decision based on:

- learning curve
- development pace
- community size
- papers implemented in the framework
- stability and long-term growth
- performance for specific task and hardware setup



TF variables and gradients

Placing variables and tensors

```
import tensorflow as tf

with tf.device('CPU:0'): # places tensors and execute on CPU memory
  a = tf.Variable([[2.0, -3.0, 1.0], [1.0, 5.0, 3.0]])
  b = tf.constant([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]])
  c = tf.matmul(a, b)
print(c)

"""Output
tf.Tensor(
  [[-1. -1.]
   [20. 20.]], shape=(2, 2), dtype=float32)
"""
```

Placing variables and tensors

```
import tensorflow as tf

with tf.device('CPU:0'): # places tensors and execute on CPU memory
  a = tf.Variable([[2.0, -3.0, 1.0], [1.0, 5.0, 3.0]])
  b = tf.constant([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]])

with tf.device('GPU:0'): # objects and operations move to GPU
  c = tf.matmul(a, b)
print(c)

"""Output
tf.Tensor(
  [[-1. -1.]
   [20. 20.]], shape=(2, 2), dtype=float32)
"""
```

TF gradients

Automatic gradient

```python
import tensorflow as tf

x = tf.Variable(3.0)

with tf.GradientTape() as tape:
  y = x ** 2 + x

# dy/dx = 2x + 1
dy_dx = tape.gradient(y, x)
print(dy_dx.numpy())
```

Automatic gradient

```
import tensorflow as tf

w = tf.Variable(tf.random.normal((3, 2)), name='w')
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
x = [[1., 2., 3.]] # automatic cast to tf.constant

with tf.GradientTape() as tape:
  y = tf.nn.sigmoid(x @ w + b)
  loss = tf.reduce_mean(tf.math.square(y))

[dl_dw, dl_db] = tape.gradient(loss, [w, b])
print(dl_dw, dl_db)
```

Automatic gradient

```
x0 = tf.Variable(3.0, name='x0') # A trainable variable
x1 = tf.Variable(3.0, name='x1', trainable=False) # Not trainable
x2 = tf.Variable(2.0, name='x2') + 1.0 # Not a Variable: + tensor = a tensor.
x3 = tf.constant(3.0, name='x3') # Not a variable

with tf.GradientTape() as tape:
  y = (x0**2) + (x1**2) + (x2**2)

grad = tape.gradient(y, [x0, x1, x2, x3])
"""
tf.Tensor(6.0, shape=(), dtype=float32)
None
None
None
"""
```

TensorFlow Module

Custom Module

```
class SimpleModule(tf.Module):
  def __init__(self, name=None):
    super().__init__(name=name)
    self.a_variable = tf.Variable(5.0, name="train_me")
    self.non_trainable_variable = tf.Variable(5.0, trainable=False)

  def __call__(self, x):
    return self.a_variable * x + self.non_trainable_variable

simple_module = SimpleModule(name="simple")

simple_module(tf.constant(5.0))
```

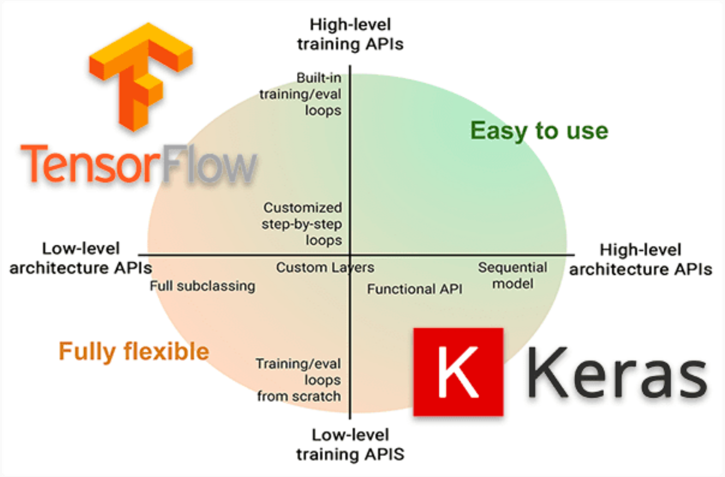
Custom Module

```
class Dense(tf.Module):
  def __init__(self, in_features, out_features, name=None):
    super().__init__(name=name)
    self.w = tf.Variable(
      tf.random.normal([in_features, out_features]), name='w')
    self.b = tf.Variable(tf.zeros([out_features]), name='b')

  def __call__(self, x):
    y = tf.matmul(x, self.w) + self.b
    return tf.nn.relu(y)
```

Keras

Keras


High-level
training APIs
Built-in
training/eval
loops
Easy to use
TensorFlow
Customized
step-by-step
loops
Low-level
architecture APIs
High-level
architecture APIs
Custom Layers
Full subclassing
Functional API
Sequential
model
Fully flexible
Training/eval
loops
from scratch
Keras
Low-level
training APIS

Sequential model

```python
import tensorflow as tf
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(2, activation="relu"))
model.add(tf.keras.layers.Dense(3, activation="relu"))
model.add(tf.keras.layers.Dense(1))

# call model
y = model(tf.ones((1,1)))
"""
<tf.Tensor: shape=(1, 1), dtype=float32,
 numpy=array([[0.09049725]], dtype=float32)>
"""
```

Sequential model

```
# print model summary
model.summary()
"""
  Model: "sequential"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 2)                 4
dense_1 (Dense)              (None, 3)                 9
dense_2 (Dense)              (None, 1)                 4
=================================================================
Total params: 17
Trainable params: 17
Non-trainable params: 0

_________________________________________________________________
"""
```

Sequential model

```python
import tensorflow as tf
model = tf.keras.Sequential()
model.add(...)

# compile model
model.compile(...)

# train model
model.fit(...)
```

Sequential model

A Sequential model is not appropriate when:

- Your model has multiple inputs or multiple outputs
- Any of your layers has multiple inputs or multiple outputs
- You need to do layer sharing
- You want non-linear topology (e.g. a residual connection, a multi-branch model)

Solution $\rightarrow$ use the functional API

The Functional API

```python
import tensorflow as tf

# build model step by step
inputs = tf.keras.Input(shape=(5,))
x1 = tf.keras.layers.Dense(64, activation="relu")(inputs)
x2 = tf.keras.layers.Dense(64, activation="relu")(x1)
outputs = tf.keras.layers.Dense(1)(x2)

# build model from functional API
model = tf.keras.Model(inputs=inputs, outputs=outputs, name="mymodel")
```

The Functional API

```python
import tensorflow as tf
# ...
# load model and data
# ...
model = tf.keras.Model(inputs=inputs, outputs=outputs, name="mymodel")

# compile model with custom loss and optimizer
model.compile(loss="mean_squared_error",
              optimizer=tf.keras.optimizers.SGD())

# train and save history
history = model.fit(x_train, y_train, batch_size=64, epochs=1000)

# evaluate model performance on separate test set
test_loss, test_accuracy = model.evaluate(x_test, y_test)
```