EUDAT Collaborative Data Infrastructure

PRACE

Introduction to Keras TensorFlow

Marco Rorro
m.rorro@cineca.it
CINECA – SCAI SuperComputing Applications and Innovation Department

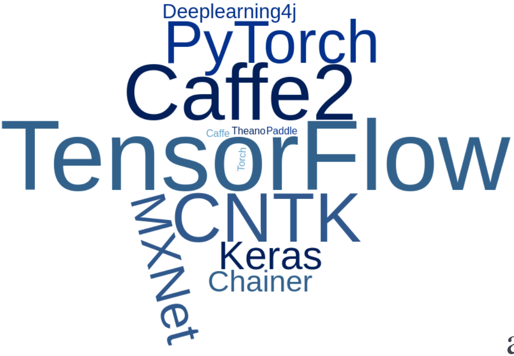
Table of Contents

Introduction

Keras

Distributed Deep Learning

Deep learning frameworks



and others

https://skymind.ai/wiki/comparison-frameworks-dl4j-tensorflow-pytorch

TensorFlow

- Google Brain's second generation machine learning system

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful data-flow graphs

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful data-flow graphs
- automatic differentiation capabilities

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful data-flow graphs
- automatic differentiation capabilities
- optimization algorithms: gradient and proximal gradient based

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful data-flow graphs
- automatic differentiation capabilities
- optimization algorithms: gradient and proximal gradient based
- code portability (CPUs, GPUs, on desktop, server, or mobile computing platforms)

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful data-flow graphs
- automatic differentiation capabilities
- optimization algorithms: gradient and proximal gradient based
- code portability (CPUs, GPUs, on desktop, server, or mobile computing platforms)
- Python interface is the preferred one (Java, C and Go also exist)

TensorFlow

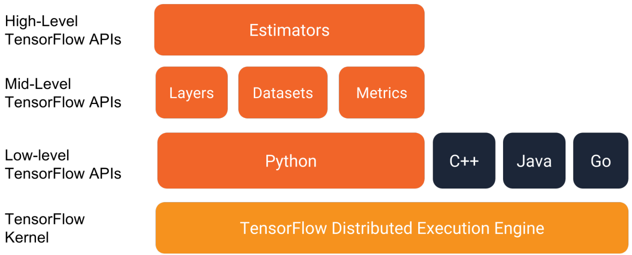
- Google Brain's second generation machine learning system
- computations are expressed as stateful data-flow graphs
- automatic differentiation capabilities
- optimization algorithms: gradient and proximal gradient based
- code portability (CPUs, GPUs, on desktop, server, or mobile computing platforms)
- Python interface is the preferred one (Java, C and Go also exist)
- installation through: pip, Docker, Anaconda, from sources

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful data-flow graphs
- automatic differentiation capabilities
- optimization algorithms: gradient and proximal gradient based
- code portability (CPUs, GPUs, on desktop, server, or mobile computing platforms)
- Python interface is the preferred one (Java, C and Go also exist)
- installation through: pip, Docker, Anaconda, from sources
- Apache 2.0 open-source license

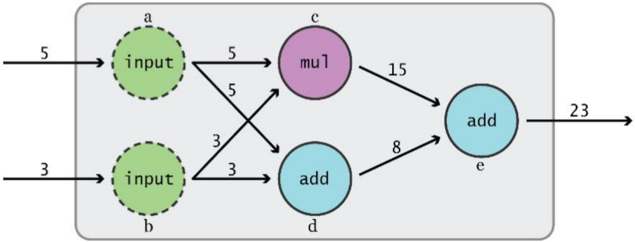
Tensorflow

- Tensorflow is a computational framework for building machine learning models
  - High-level, object-oriented API (tf.estimator)
  - Libraries for common model components (tf.layers/tf.losses/tf.metrics)
  - Lower-level APIs (TensorFlow)



TensorFlow dataflow graph

- TensorFlow 1 separates definition of computations from their execution
  - Phase 1: assemble a graph
  - Phase 2: use a session to execute operations in the graph.
- not true in eager mode (default in TensorFlow 2)



Introduction

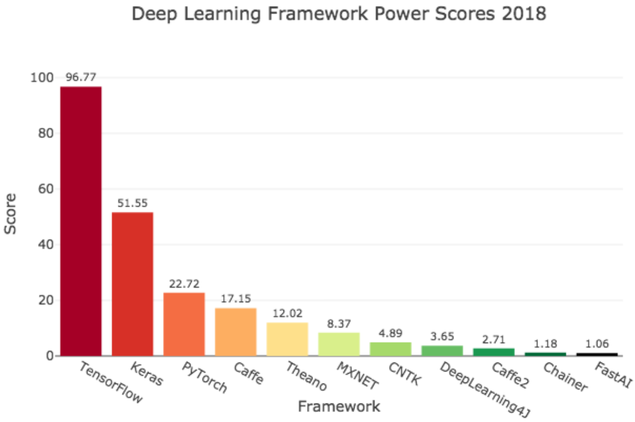
Keras

Distributed Deep Learning

Keras

- Keras is a high-level neural networks API, written in Python, developed with a focus on enabling fast experimentation.
- Keras offers a consistent and simple API, which minimizes the number of user actions required for common use cases, and provides clear and actionable feedback upon user error.
- Keras is capable of running on top of many deep learning backends such as TensorFlow, CNTK, or Theano. This capability allows Keras model to be portable across all these backends.
- Kesas is one of the most used Deep Learning Framework used by researchers, and is now part of the official TensorFlow Higher Level API as tf.keras
- Keras models can be trained on CPUs, Xeon Phi, Google TPUs and any GPU or OpenCL-enabled GPU like device.
- Keras is the TensorFlow's high-level API for building and training deep learning models.

Keras



Deep Learning Framework Power Scores 2018
Score
100
80
60
40
20
0
96.77
51.55
22.72
17.15
12.02
8.37
4.89
3.65
2.71
1.18
1.06
TensorFlow
Keras
PyTorch
Caffe
Theano
MXNET
CNTK
DeepLearning4J
Caffe2
Chainer
FastAI
Framework

Building models with Keras

- The core data structure of Keras is the Model which is basically a container of one or more Layers.
- There are two main types of models available in Keras: the Sequential model and the Model class, the latter used to create advanced models.
- The simplest type of model is the Sequential model, which is a linear stack of layers. Each layer is added to the model using the .add() method of the Sequential model object.
- The model needs to know what input shape it should expect. The first layer in a Sequential model (and only the first) needs to receive information about its input shape, specifing the input_shape argument. The following layers can do automatic shape inference from the shape of its predecessor layer.

Collaborative EUDAT Data Infrastructure

PRACE

Model build

```
import tensorflow as tf
from tensorflow import keras

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation

model = Sequential()
# Adds to the model a densely-connected
# layer with 32 units with input shape 16:
model.add(Dense(32, input_shape=(16,)))
# Adds another layer with 16 units,
# each connected to 32 outputs of previous layer
model.add(Dense(16))
# Last layer with 8 units,
# each connected to 16 outputs of previous layer
model.add(Dense(8, activation='softmax'))
```

Activation functions

- The activation argument specifies the activation function for the current layer. By default, no activation is applied.
- The softmax activation function normalize the output to a probability distribution. Is commonly used in the last layer of a model. To select a single output in a classification problem the most probable one can be selected.
- The ReLU (Rectified Linear Unit), $\max(0,x)$, is commonly used as activation function for the hidden layers.
- Many other activation functions are available or easily defined as well as layer types.

Model compile

- Once the model is built, the learning process is configured by calling the compile method. The compile phase is required to configure the following (mandatory) elements of the model:
  - optimizer: this object specifies the optimization algorithm which adapt the weights of the layers during the training procedure;
  - loss: this object specifies the function to minimize during the optimization;
  - metrics: [optional] this objects measure the performance of your model and is used to monitor the training

```
# Configure the model for mean-squared error regression.
model.compile(optimizer='sgd', # stochastic gradient descent
loss='mse', # mean squared error
metrics=['accuracy']) # an optional list of metrics
```

Model compile

- Once the model is compiled, we can check its status using the summary and get precious information on model composition, layer connections and number of parameters.

```
model.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 32)                544
_________________________________________________________________
dense_1 (Dense)              (None, 16)                528
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 136
=================================================================
Total params: 1,208
Trainable params: 1,208
Non-trainable params: 0
_________________________________________________________________
```

Model training

- The .fit method trains the model against a set of training data, and reports loss and accuracy useful to monitor the training process.

```
import numpy as np

# generate synthetic training dataset
x_train = np.random.random((1000, 16))
y_train = np.random.random((1000, 8))

# generate synthetic validation data
x_valid = np.random.random((100, 16))
y_valid = np.random.random((100, 8))

# fit the model using training dataset
# over 10 epochs of 32 batch size each
# report training progress against validation data
model.fit(x=x_train, y=y_train,
          batch_size=32, epochs=10,
          validation_data=(x_valid, y_valid))
```

Model evaluation and prediction

- Once the training process has completed, the model can be evaluated against the validation dataset. The evaluate method returns the loss value and, if the model was compiled providing also a metrics argument, the metric values.

```
model.evaluate(x_valid, y_valid, batch_size=32)
```

- The predict method can finally be used to make inference on new data

```
model.predict(x_valid, batch_size=128)
```

Collaborative EUDAT Data Infrastructure

PRACE

Model saving and restore

- A trained model can be saved and stored to a file for later retreival. This allows you to checkpoint a model and resume training later without rebuiling and training from scratch.
- Files are saved in HDF5 format, with all weight values, model's configuration and even the optimizer's configuration.

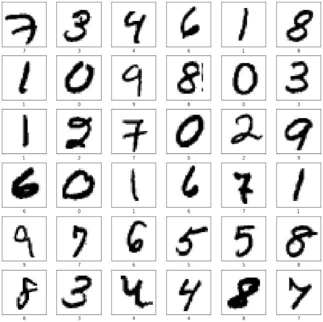
```
save_model_path='saved/intro_model'
model.save(filepath=save_model_path, include_optimizer=True)
```

```
model = tf.keras.models.load_model(filepath=save_model_path)
```

Try it out

The MINIST dataset

- The MNIST data set is a standard set of handwritten numerical digits from 0 to 9 which is commonly used as the "Hello World" test for Deep Learning classification problem.
- Each sample is a 28×28 grayscale image.



Loading MNIST

- Keras comes with many dataset built in and automatically splits the data into a training and validation set.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.
    load_data()
```

Defining a model

```
model.add(tf.keras.layers.Conv2D(filters=64, kernel_size=2, padding='
    same', activation='relu', input_shape=(??,??,?)))
model.add(tf.keras.layers.MaxPooling2D(pool_size=2))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Conv2D(filters=32, kernel_size=2, padding='
    same', activation='relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=2))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(?, activation='softmax'))
```

Compiling and Training

- The categorical cross entropy, $-\sum p(x)q(x)$, with p the true distribution and q the expected one.
- Adam is adaptive learning rate optimization algorithm.

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

```
model.fit(x_train,
          y_train,
          batch_size=64,
          epochs=10,
          validation_data=(x_valid, y_valid),
          )
```

- Try it out

Callbacks

- What if we want to stop if accuracy is > 0.01 ?
- define a callback

```
class myCallback(keras.callbacks.Callback)
    def on_epoch_end(self,epoch,logs={}):
        if(logs.get('acc')>0.01):
            print("\nAccuracy exceeds threshold, Stop train!")
            self.model.stop_training =True
```

- the install it

```
mycallbacks=myCallBack()
model.fit(train_images,train_labels,epoch=100,callbacks=[mycallbacks
     ])
```

Callbacks

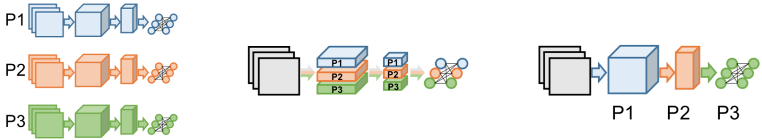
- Keras provides some predefined callbacks to feed in, among them for example:
  - **TerminateOnNaN()**: that terminates training when a NaN loss is encountered
  - **ProgbarLogger()**: that prints metrics to stdout
  - **ModelCheckpoint(filepath)**: that save the model after every epoch
  - **EarlyStopping**: which stop training when a monitored quantity has stopped improving
  - **LambdaCallback**: for creating simple, custom callbacks on-the-fly
- You can select one or more callback and pass them as a list to the callback argument of the fit method.
- You can also create a callback object from scratch, customizing its behaviour overloading the base methods of the Callback Keras class:
  - **on_epoch_begin** and **on_epoch_end**
  - **on_batch_begin** and **on_batch_end**
  - **on_train_begin** and **on_train_end**
- A callback has access to its associated model through the class property self.model, so that you can monitor and access many of the quantities which are in the optimization process.

Introduction

Keras

Distributed Deep Learning

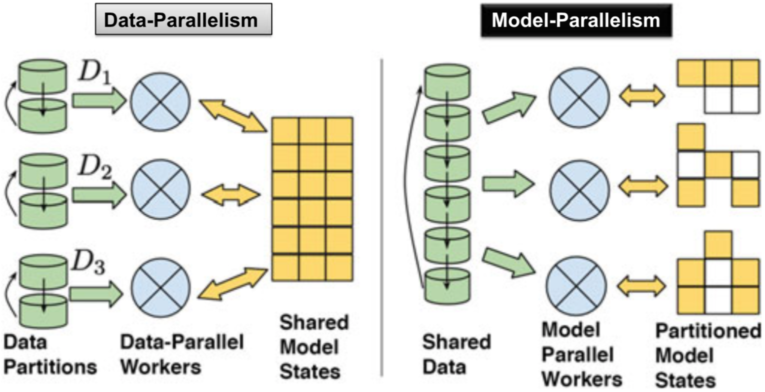
Neural Network concurrency



(a) Data Parallelism (b) Model Parallelism (c) Layer Pipelining

Tal Ben-Nun and Torsten Hoefler, Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis, 2018

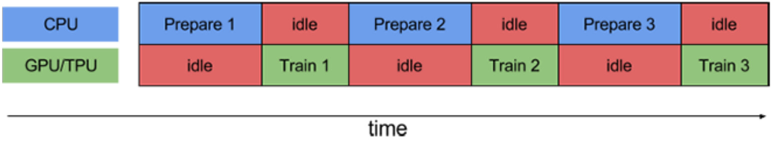
Data Parallelism vs Model Parallelism

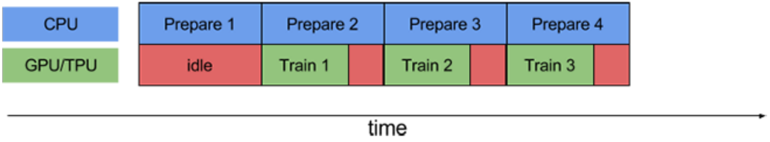


Hardware and Libraries

- It is not only a matter of computational power:
  - CPU (MKL-DNN)
  - GPU (cuDNN)
  - FPGA
  - TPU
- Input/Output
  - SSD
  - Parallel file system (if you run in parallel)
- Communication and interconnection too, if you are running in distributed mode
  - MPI
  - gRPC + verbs (RDMA)
  - NCCL

Data Input Pipeline





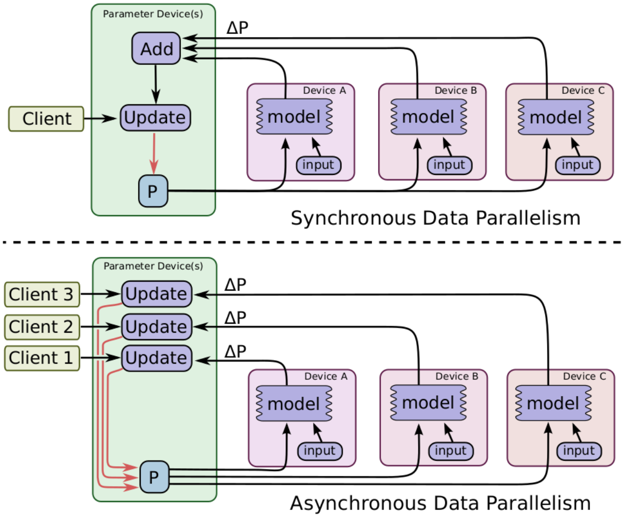
CPU optimizations

- Built from source with all of the instructions supported by the target CPU and the MKL-DNN option for Intel® CPU.
- Adjust thread pools
  - *intra_op_parallelism_threads*: Nodes that can use multiple threads to parallelize their execution will schedule the individual pieces into this pool. (*OMP_NUM_THREADS*)
  - *inter_op_parallelism_threads*: All ready nodes are scheduled in this pool

```
config = tf.ConfigProto()
config.intra_op_parallelism_threads = 44
config.inter_op_parallelism_threads = 44
tf.session(config=config)
```

- The MKL is optimized for NCHW (default NHWC) data format and use the following variables to tune performance: *KMP_BLOCKTIME*, *KMP_AFFINITY*, *OMP_NUM_THREADS*

Synchronous and asynchronous data parallel training



Keras GPUs Parallel Model

```
model = keras.Sequential()
...
gpus=4
parallel_model = keras.utils.multi_gpu_model(model, gpus=gpus)

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

parallel_model.fit(x_train,
         y_train,
         batch_size=batch_size,
         epochs=epochs,
         validation_data=(x_valid, y_valid),
         )
```

Collaborative EUDAT Data Infrastructure

PRACE

Keras + Uber/Horovod

```
...
import horovod.tensorflow.keras as hvd
...
#Horovod: initialize Horovod.
hvd.init()

opt = tf.keras.optimizers.Adam(0.001 * hvd.size())
opt = hvd.DistributedOptimizer(opt)

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

callbacks = [
    # Horovod: broadcast initial variable states from rank 0 to all
         other processes.
    hvd.callbacks.BroadcastGlobalVariablesCallback(0),
]

model.fit(x_train,
         y_train,
         batch_size=batch_size,
         callbacks=callbacks,
         epochs=epochs,
         validation_data=(x_valid, y_valid)
         )
```

Hands-on

- Use $keras - mnist.py$
- Define the input shape in the first layer and the output shape in the last layer
- Run it and play with hyperpameters

Hands-on

- Use *keras – mnist.py*
- Define the input shape in the first layer and the output shape in the last layer
- Run it and play with hyperpameters
- Use *keras – mnist – mgpu.py*
- Try the multi GPU on Galileo. Play with the number of GPU and the hyperparameter (*batchsize* and *epochs*)

Hands-on

- Use $keras-mnist.py$
- Define the input shape in the first layer and the output shape in the last layer
- Run it and play with hyperpameters
- Use $keras-mnist-mgpu.py$
- Try the multi GPU on Galileo. Play with the number of GPU and the hyperparameter ($batchsize$ and $epochs$)
- Use $keras-mnist-hvd.py$. Play with the number of GPU, the number of nodes and the hyperparameters ($batchsize$ and $epochs$)

Other References

- Horovod
- NCCL
- MNIST
- CIFAR datasets
- Deeplearning.ai youtube channel