IML Keras Workshop

Stefan Wunsch
stefan.wunsch@cern.ch

March 22, 2017

Outline

The workshop has three parts:

1. **Introduction to Keras** using the MNIST dataset
2. Tutorial for the **ROOT/TMVA Keras interface**
3. (Optional) Usage of **lwtnn with Keras**

Assumptions and targets of the tutorial:

- You haven't used Keras before.
- You want to know why Keras is so popular and how it works!

You can download the slides and all examples running this:
`git clone https://github.com/stwunsch/iml_keras_workshop`

Part 1: Keras Tutorial

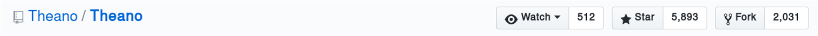
What is Keras?

- Tool to train and apply (deep) neural networks
- **Python wrapper around Theano and TensorFlow**
- Hides many low-level operations that you don't want to care about.
- **Sacrificing little functionality** of Theano and TensorFlow for much easier user interface

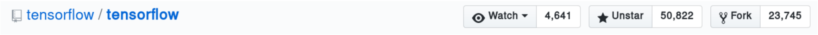
Being able to go from idea to result with the least possible delay is key to doing good research.

Theano? TensorFlow?

- **They are doing basically the same.**
- TensorFlow is growing much faster and gains more support (Google does it!).



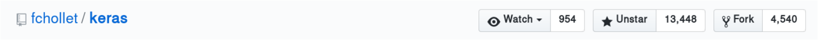
Theano *is a Python library that allows you to* ***define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently****.*

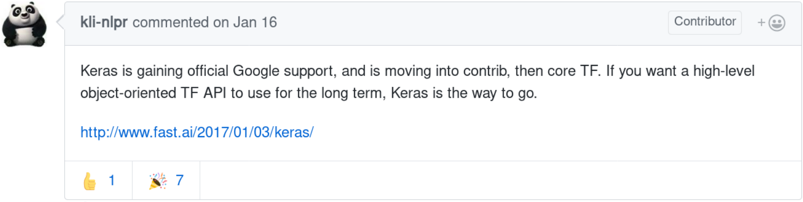


TensorFlow *is an open source software library for* ***numerical computation using data flow graphs****. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.*

Why Keras and not one of the other wrappers?

- There are lot of alternatives: TFLearn, Lasagne, ...
- None of them are as **popular** as Keras!
- Will be **tightly integrated into TensorFlow** and officially supported by Google.
- Look like a **safe future for Keras**!





- Read the full story here: Link

Let's start!

- **How does the tutorial works?** You have the choice:
    1. You can just listen and learn from the code examples on the slides.
    2. You can follow along with the examples on your own laptop.

Using **lxplus** (straight forward way, recommended):

```
ssh -Y you@lxplus.cern.ch

# Download the files
git clone https://github.com/stwunsch/iml_keras_workshop

# Set up the needed software from CVMFS
source iml_keras_workshop/setup_lxplus.sh
# or:
source /cvmfs/sft.cern.ch/lcg/views/LCG_88/x86_64-slc6-gcc49-opt/setup.sh
```

Using **SWAN** (backup plan):

- Open a terminal using the `New` button

Using **your own laptop** (if you have some experience with this):

```
# Install all needed Python packages using pip
pip install theano
pip install keras=="1.1.0"
pip install h5py
```

Configure Keras Backend

- Two ways to configure Keras backend (Theano or TensorFlow):
  1. Using **environment variables**
  2. Using **Keras config file** in `$HOME/.keras/keras.json`

Example setup using environment variables:

```
# Select Theano as backend for Keras using enviroment variable `KERAS_BACKEND`
from os import environ
environ['KERAS_BACKEND'] = 'theano'
```

Example Keras config using Theano as backend:

```
$ cat $HOME/.keras/keras.json
{
    "image_dim_ordering": "th",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "theano"
}
```

MNIST Example

- **File in examples:** `example_keras/mnist_train.py`
- **MNIST dataset?**
  - **Official website:** Yann LeCun's website (Link)
  - Database of **70000 images of handwritten digits**
  - 28x28 pixels in greyscale as input, digit as label

- **Inputs and targets:**
  - Input: 28x28 matrix with floats in [0, 1]
  - Output: One-hot encoded digits, e.g., 2 $\rightarrow$ [0 0 1 0 0 0 0 0 0 0]

```
# Download MINST dataset using Keras examples loader
x_train, x_test, y_train, y_test = download_mnist_dataset()
```

Define the Neural Network Architecture

- Keras offers **two ways** to define the architecture:
  1. **Sequential model for simple models**, layers are stacked linearily
  2. **Functional API for complex models**, e.g., with multiple inputs and outputs

Sequential model example: Binary classification with 4 inputs

```
model = Sequential()
# Fully connected layer with 32 hidden nodes
# and 4 input nodes and hyperbolic tangent activation
model.add(Dense(32, activation='tanh', input_dim=4))
# Single output node with sigmoid activation
model.add(Dense(1, activation='sigmoid'))
```

Define the Neural Network Architecture (2)

Example model for handwritten digit classification:

```
model = Sequential()

# First hidden layer
model.add(Convolution2D(
        4, # Number of output feature maps
        2, # Column size of kernel used for convolution
        2, # Row size of kernel used for convolution
        activation='relu', # Rectified linear unit
        input_shape=(28,28,1))) # 28x28 image with 1 channel

# All other hidden layers
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.5))

# Output layer
model.add(Dense(10, activation='softmax'))
```

Model Summary

```
# Print model summary
model.summary()
```

- Well suitable to get an idea of the **number of free parameters**, e.g., number of nodes after flattening.

```
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
convolution2d_1 (Convolution2D)  (None, 27, 27, 4)     20          convolution2d_input_1[0][0]
____________________________________________________________________________________________________
maxpooling2d_1 (MaxPooling2D)    (None, 13, 13, 4)     0           convolution2d_1[0][0]
____________________________________________________________________________________________________
flatten_1 (Flatten)              (None, 676)           0           maxpooling2d_1[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 16)            10832       flatten_1[0][0]
____________________________________________________________________________________________________
dropout_1 (Dropout)              (None, 16)            0           dense_1[0][0]
____________________________________________________________________________________________________
dense_2 (Dense)                  (None, 10)            170         dropout_1[0][0]
====================================================================================================
Total params: 11,022
Trainable params: 11,022
Non-trainable params: 0
```

Loss Function, Optimizer and Validation Metrics

- After definition of the architecture, the model is compiled.
- **Compiling** includes:
  - Define **loss function**: Cross-entropy, mean squared error, ...
  - Configure **optimizer algorithm**: SGD, AdaGrad, Adam, ...
  - Set **validation metrics**: Global accuracy, Top-k-accuracy, ...

```python
# Compile model
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])
```

$\Rightarrow$ **That's it!** Your model is ready to train!

Available Layers, Losses, Optimizers, ...

- There's **everything you can imagine**, and it's **well documented**.
- Possible to **define own layers** and **custom metrics** in Python!
- Check out: `www.keras.io`

Callbacks and Training

- **Callbacks** are executed before or after each training epoch.
- Custom callbacks are possible!

```
# Set up callbacks
checkpoint = ModelCheckpoint(
        filepath='mnist_example.h5',
        save_best_only=True)
```

- Training is only a single line of code.

```
# Train
model.fit(images, labels, # Training data
          batch_size=100, # Batch size
          nb_epoch=10, # Number of training epochs
          validation_split=0.2, # Use 20% of the train dataset
                                # for validation
          callbacks=[checkpoint]) # Register callbacks
```

That's all you need. Try the `mnist_train.py` script!

Callbacks and Training (2)

- Output looks like this:

```
Train on 30000 samples, validate on 30000 samples
Epoch 1/10
30000/30000 [==============================] - 4s - loss: 1.5575 - acc: 0.4324
Epoch 2/10
30000/30000 [==============================] - 4s - loss: 1.0883 - acc: 0.6040
Epoch 3/10
30000/30000 [==============================] - 4s - loss: 0.9722 - acc: 0.6419
Epoch 4/10
30000/30000 [==============================] - 4s - loss: 0.9113 - acc: 0.6620
Epoch 5/10
30000/30000 [==============================] - 4s - loss: 0.8671 - acc: 0.6787
Epoch 6/10
30000/30000 [==============================] - 4s - loss: 0.8378 - acc: 0.6836
Epoch 7/10
30000/30000 [==============================] - 4s - loss: 0.8105 - acc: 0.6918
Epoch 8/10
30000/30000 [==============================] - 4s - loss: 0.8023 - acc: 0.6901
Epoch 9/10
30000/30000 [==============================] - 4s - loss: 0.7946 - acc: 0.6978
Epoch 10/10
30000/30000 [==============================] - 4s - loss: 0.7696 - acc: 0.7049
```

Advanced Training Methods

These methods can be used to train on data that does not fit in memory.

- Training on **single batches**, performs a single gradient step:

```
model.train_on_batch(x, y, ...)
```

- Training with data from a **Python generator**:

```
def generator_function():
    while True:
        yield custom_load_next_batch()

model.fit_generator(generator_function, ...)
```

Store Model to File

Again, Keras offers **two ways** to do so:

- Store architecture and weights **in one file**:

```
model = Sequential()
...
model.save('path/to/file')
```

- Store **architecture** as JSON or YAML file and the **weights separately**:

```
model = Sequential()
...
json_string = model.to_json()
model.save_weights('path/to/file')
```

Load and Apply a Trained Model

Look at the file `mnist_apply.py`!

- **Single line of code and your full model is back**, if you've used the `model.save()` method:

```
model = load_model('mnist_example.h5')
```

- Otherwise, it's not much more complicated:

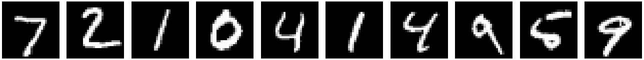
```
model = model_from_json(json_string)
model.load_weights('path/to/file')
```

- **Application** is an one-liner as well:

```
prediction = model.predict(some_numpy_array)
```

Application on Handwritten Digits

- **PNG images of handwritten digits** are placed in `example_keras/mnist_example_images`, have a look!



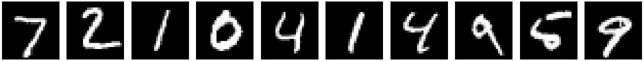
- Let's **apply our trained model** on the images:

```
pip install --user pypng
./mnist_apply.py mnist_example_images/*
```

- **If you are bored on your way home:**
  1. Open with GIMP `your_own_digit.xcf`
  2. Dig out your most beautiful handwriting
  3. Save as PNG and run your model on it

Application on Handwritten Digits (2)

```
Predict labels for images:
    mnist_example_images/example_input_0.png : 7
    mnist_example_images/example_input_1.png : 2
    mnist_example_images/example_input_2.png : 1
    mnist_example_images/example_input_3.png : 0
    mnist_example_images/example_input_4.png : 4
    mnist_example_images/example_input_5.png : 1
    mnist_example_images/example_input_6.png : 4
    mnist_example_images/example_input_7.png : 9
    mnist_example_images/example_input_8.png : 4
    mnist_example_images/example_input_9.png : 9
```



Part 2: TMVA Keras Interface

Prerequisites

- **Keras inteface integrated in ROOT since v6.08**
- Example for this tutorial is placed here: `example_tmva/BinaryClassification.py`
- To try the example, it's recommended to use **lxplus**:
  - `ssh -Y you@lxplus.cern.ch`
  - Source software stack 88 or bleeding edge

How to source LCG 88 on lxplus:

```
source /cvmfs/sft.cern.ch/lcg/views/LCG_88/x86_64-slc6-gcc49-opt/setup.sh
```

Why do we want a Keras interface in TMVA?

1. **Fair comparison** with other methods
   - Same preprocessing
   - Same evaluation
2. **Try state-of-the-art DNN performance in existing analysis**/application that is already using TMVA
3. **Access data** in **ROOT files** easily
4. Integrate Keras in your **application** using **C++**
5. **Latest DNN algorithms in the ROOT** framework with **minimal effort**

How does the interface work?

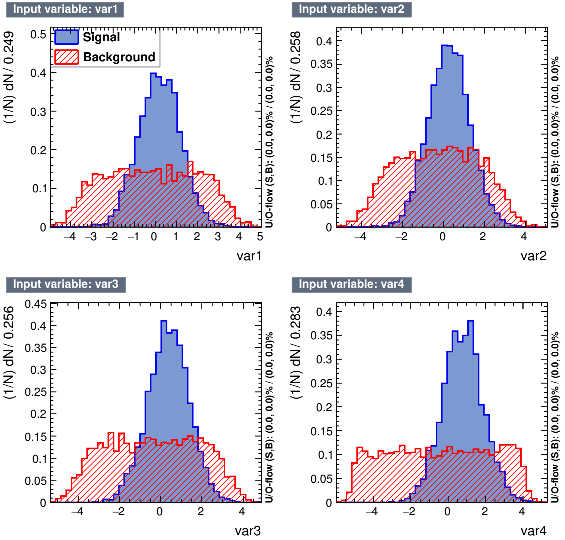
1. **Model definition** done in **Python** using **Keras**
2. **Data management**, **training** and **evaluation** within the TMVA framework
3. **Application** using the TMVA reader or plain Keras

- The interface is implemented in the optional **PyMVA** part of TMVA:

```
# Enable PyMVA
ROOT.TMVA.PyMethodBase.PyInitialize()
```

Example Setup

- **Dataset** of this example is standard ROOT/TMVA test dataset for binary classification



Model Definition

- Setting up the model does not differ from using plain Keras:

```
model = Sequential()
model.add(Dense(64, init='glorot_normal', activation='relu', input_dim=4))
model.add(Dense(2, init='glorot_uniform', activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer=Adam(), metrics=['accuracy',])
model.save('model.h5')
```

- For **binary classification** the model needs **two output nodes**:

```
model.add(Dense(2, activation='softmax'))
```

- For **multi-class classification** the model needs **two or more output nodes**:

```
model.add(Dense(5, activation='softmax'))
```

- For **regression** the model needs a **single output node**:

```
model.add(Dense(1, activation='linear'))
```

Training

- **Training options** defined in the **TMVA booking options**:

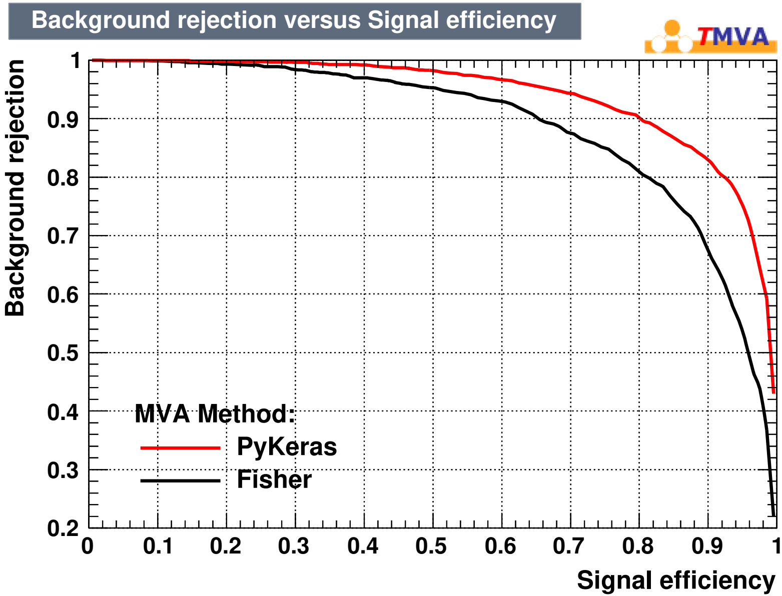
```
factory.BookMethod(dataloader, TMVA.Types.kPyKeras, 'PyKeras',
        'H:V:VarTransform=G:'+
        'Verbose=1'+\ # Training verbosity
        'FilenameModel=model.h5:'+\ # Model from definition
        'FilenameTrainedModel=modelTrained.h5:'+\ # Optional!
        'NumEpochs=10:'+\
        'BatchSize=32'+\
        'ContinueTraining=false'+\ # Load trained model again
        'SaveBestOnly=true'+\ # Callback: Model checkpoint
        'TriesEarlyStopping=5'+\ # Callback: Early stopping
        'LearningRateSchedule=[10,0.01; 20,0.001]')
```

That's it! You are ready to run!

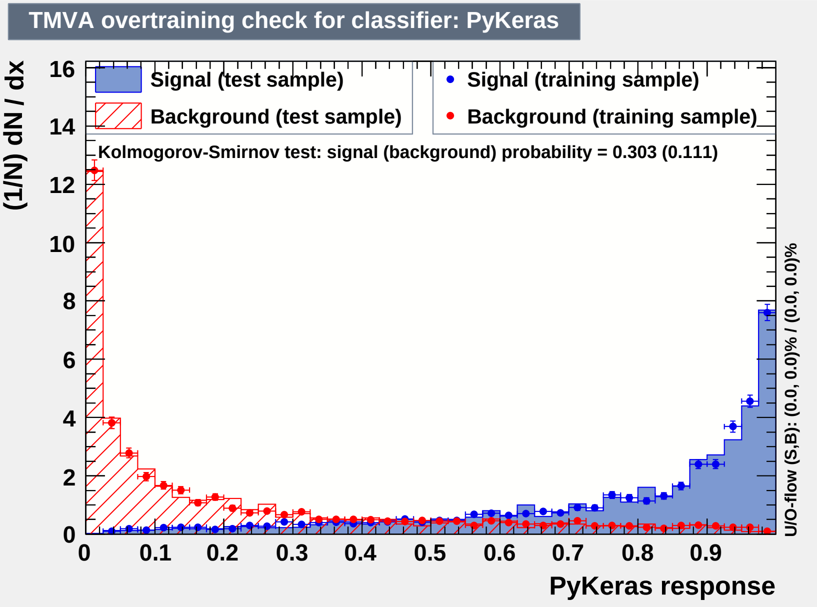
```
python BinaryClassification.py
```

Run TMVA GUI to examine results: `root -l TMVAGui.C`

Training Results: ROC



Training Results: Overtraining Check



Application

- **Does not differ from any other TMVA method!**
- **Example** application is set up here: `example_tmva/ApplicationBinaryClassification.py`
- You can use **plain Keras** as well, just load the file from the option `FilenameTrainedModel=trained_model.h5`.

```
model = keras.models.load_model('trained_model.h5')
prediction = model.predict(some_numpy_array)
```

Application (2)

Run `python ApplicationBinaryClassification.py`:

```
# Response of TMVA Reader
                          : Booking "PyKeras" of type "PyKeras" from
                          : BinaryClassificationKeras/weights/TMVAClassification_PyKeras.weights.xml.
Using Theano backend.
DataSetInfo               : [Default] : Added class "Signal"
DataSetInfo               : [Default] : Added class "Background"
                          : Booked classifier "PyKeras" of type: "PyKeras"
                          : Load model from file:
                          : BinaryClassificationKeras/weights/TrainedModel_PyKeras.h5

# Average response of MVA method on signal and background
Average response on signal:     0.78
Average response on background: 0.21
```

Part 3: lwtnn with Keras

What is lwtnn?

- **C++ library** to apply neural networks
  - Minimal dependencies: C++11, Eigen
  - Robust
  - Fast

- **“Asymmetric” library:**
  - **Training** in any language and framework on any system, e.g., **Python and Keras**
  - **Application** in **C++** for real-time applications in a limited environment, e.g., high-level trigger

- **GitHub:** `https://github.com/lwtnn/lwtnn`
- **IML talk about lwtnn by Daniel Guest:** Link

How does it work?

- **Example** in this tutorial:
  - **Iris dataset**: Classify flowers based on their proportions
  - **4 features**: Sepal length/width and petal length/width
  - **3 targets** (flower types): Setosa, Versicolour, and Virginica

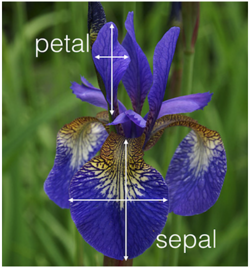


Image source

1. **Train** the model in **Python** using **Keras**
2. **Convert** the architecture and weights to JSON format
3. **Load** and **apply** the model using lwtnn in **C++** using the JSON file

Set up lwtnn

- **Run `setup_lwtnn.sh`**:
  - Downloads Eigen v3.3.0
  - Downloads lwtnn v2.0
  - Builds lwtnn

- **Does not work on lxplus**, because of bug in Boost library...

Training

- **We don't focus on this for now!**
- **Script:** `example_lwtnn/train.py`
  - **Loads iris dataset** using scikit-learn
  - **Trains** a simple three-layer feed-forward network
  - **Saves model** weights and architecture separately

```
# Save model
model.save_weights('weights.h5', overwrite=True)

file_ = open('architecture.json', 'w')
file_.write(model.to_json())
file_.close()
```

Convert to lwtnn JSON

- **Conversion** script of lwtnn takes these inputs:
  - **Keras architecture** as from training
  - **Keras weight file** from training
  - **Variable description** JSON file

Variable description:

```
{
  "class_labels": ["setosa", "versicolor", "virginica"],
  "inputs": [
    {
      "name": "sepal length",
      "offset": 0.0,
      "scale": 1.0
    },
    ...
```

- Puts all **together in a single JSON**, that can be read by lwtnn in C++

Convert to lwtnn JSON (2)

- **Run** `LWTNN_CONVERT`

```
python lwtnn/converters/keras2json.py architecture.json \
        variables.json weights.h5 > lwtnn.json
```

- Output `lwtnn.json` contains all information needed to run the model:

```
{
  "inputs": [
    {
      "name": "sepal length",
    ...

  "layers": [
    {
      "activation": "rectified",
      "weights": [
        -0.01653989404439926,
        ...
```

Load and Apply Model in C++ Using lwtnn

Have a look at `apply.cpp`!

Load model:

```
// Read lwtnn JSON config
auto config = lwt::parse_json(std::ifstream("lwtnn.json"));

// Set up neural network model from config
lwt::LightweightNeuralNetwork model(
        config.inputs,
        config.layers,
        config.outputs);
```

Apply model:

```
// Load inputs from argv
std::map<std::string, double> inputs;
...

// Apply model on inputs
auto outputs = model.compute(inputs);
```

Load and Apply Model in C++ Using lwtnn (2)

- **Compile `apply.cpp`:** Just type `make`
- Tell your system where it can find the `lwtnn.so` library: `export LD_LIBRARY_PATH=lwtnn/lib/`
- **Print data** of iris dataset: `python print_iris_dataset.py`

```
Inputs: 5.10 3.50 1.40 0.20 -> Target: setosa
Inputs: 7.00 3.20 4.70 1.40 -> Target: versicolor
Inputs: 6.30 3.30 6.00 2.50 -> Target: virginica
```

- **Run your model** on some examples:

```
# Apply on features of Setosa flower
./apply 5.10 3.50 1.40 0.20

# Class: Probability
setosa:     0.958865
versicolor: 0.038037
virginica:  0.003096
```