IML TensorFlow and Keras Workshop

Stefan Wunsch
stefan.wunsch@cern.ch

April 10, 2018

What is this workshop about?

- Modern description, implementation and application of neural networks
- Introduction to the currently favored packages:
  - **TensorFlow:** Low-level implementation of operations needed to implement neural networks in multi-threaded CPU and multi GPU environments
  - **Keras:** High-level convenience wrapper for backend libraries, e.g. TensorFlow, to implement neural network models

TensorFlow

tensorflow / **tensorflow**	Watch	7,714	Unstar	94,917	Fork	60,643
keras-team / **keras**	Watch	1,621	Unstar	27,781	Fork	10,187

Outline

The workshop has these parts:

1. Very brief introduction to **neural networks**
2. Modern implementation of neural networks with **computational graphs** using **TensorFlow**
3. **Rapid development** of neural network applications using **Keras**

Assumptions of the tutorial:

- You are not a neural network expert, but you know roughly how to work.
- You don't know how TensorFlow and Keras works and how they play together.
- You want to know why TensorFlow and Keras are so popular and how you can use it!

Disclaimer:

- You won't learn how to use TensorFlow or Keras in one hour.
- **This tutorial tries to provide you with a good start and all information you need to become an expert!**

Set up your system

Clone the repository with the notebooks and slides:

`git clone https://github.com/stwunsch/iml_tensorflow_keras_workshop`

Using SWAN (favored solution):

1. Log in on `swan.cern.ch` and select the software stack `LCG 93`
2. Open a terminal with `New->Terminal` and clone the repository as shown above
3. Browse to the notebooks as indicated on the following slides

Using your own laptop:

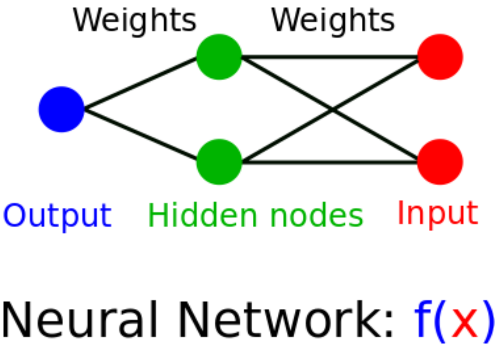
1. Clone the repository as shown above
2. Run the script `init_virtualenv.sh`
3. Source the virtual Python environment with `source py2_virtualenv/bin/activate`
4. Start a jupyter server with `jupyter notebook` and browse to the notebooks

Using lxplus:

1. Log in to lxplus with `ssh -Y your_username@lxplus.cern.ch`
2. Clone the repository as shown above
3. Source the software stack `LCG 93` with `source /cvmfs/sft.cern.ch/lcg/views/LCG_93/x86_64-slc6-gcc62-opt/setup.sh`
4. Convert the notebooks to Python scripts with `jupyter nbconvert --to python input.ipynb output.py`

(Very) brief introduction to neural networks

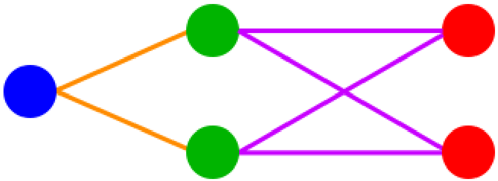
A simple neural network



- **Important:** A neural network is only a mathematical function. No magic involved!
- **Training:** Finding the best function for a given task, e.g., separation of signal and background.

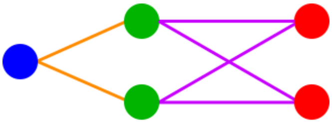
Mathematical representation

- **Why do we need to know this?**
  → TensorFlow implements these mathematical operations explicitly.
  → Basic knowledge to understand Keras' high-level layers.



$$f_{NN} = \sigma(b_2 + W_2\sigma(b_1 + W_1 x))$$

Mathematical representation (2)



$f_{NN} = \sigma(b_2 + W_2\sigma(b_1 + W_1 x))$

$$\text{Input} : x = \begin{bmatrix} x_{1,1} \\ x_{2,1} \end{bmatrix}$$

$$\text{Weight} : W_1 = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}$$

$$\text{Bias} : b_1 = \begin{bmatrix} b_{1,1} \\ b_{2,1} \end{bmatrix}$$

$$\text{Activation} : \sigma(x) = \tanh(x) \ \ \text{(as example)}$$

Activation is applied elementwise!

The "simple" neural network written as full equation:

$$f_{\mathrm{NN}} = \sigma_2 \left(\begin{bmatrix} b^2_{1,1} \end{bmatrix} + \begin{bmatrix} W^2_{1,1} & W^2_{1,2} \end{bmatrix} \sigma_1 \left(\begin{bmatrix} b^1_{1,1} \\ b^1_{2,1} \end{bmatrix} + \begin{bmatrix} W^1_{1,1} & W^1_{1,2} \\ W^1_{2,1} & W^1_{2,2} \end{bmatrix} \begin{bmatrix} x_{1,1} \\ x_{2,1} \end{bmatrix} \right) \right)$$

Further reading: Deep Learning Textbook

Free textbook written by Ian Goodfellow, Yoshua Bengio and Aaron Courville: http://www.deeplearningbook.org/

- Written by leading scientists in the field of machine learning
- **Everything you need to know** about modern machine learning and deep learning in particular.

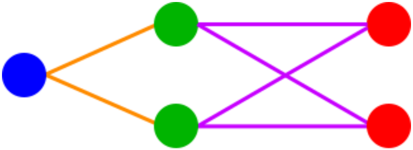
- Part I: Applied Math and Machine Learning Basics
  - 2 Linear Algebra
  - 3 Probability and Information Theory
  - 4 Numerical Computation
  - 5 Machine Learning Basics
- II: Modern Practical Deep Networks
  - 6 Deep Feedforward Networks
  - 7 Regularization for Deep Learning
  - 8 Optimization for Training Deep Models
  - 9 Convolutional Networks
  - 10 Sequence Modeling: Recurrent and Recursive Nets
  - 11 Practical Methodology
  - 12 Applications
- III: Deep Learning Research
  - 13 Linear Factor Models
  - 14 Autoencoders
  - 15 Representation Learning
  - 16 Structured Probabilistic Models for Deep Learning
  - 17 Monte Carlo Methods
  - 18 Confronting the Partition Function
  - 19 Approximate Inference
  - 20 Deep Generative Models

Computational graphs with TensorFlow

What is TensorFlow?

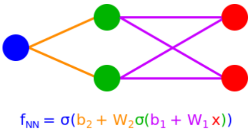
TensorFlow *is an open source software library for* ***numerical computation using data flow graphs****. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.*

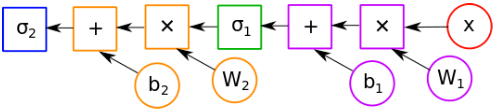
- **In first place:** TensorFlow is not about neural networks.
- But it is a **perfect match** to implement neural networks efficiently!



$$f_{NN} = \sigma(b_2 + W_2\sigma(b_1 + W_1 x))$$

Computational graphs





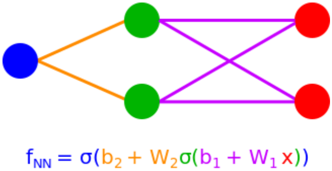
Example neural network → **According computational graph**

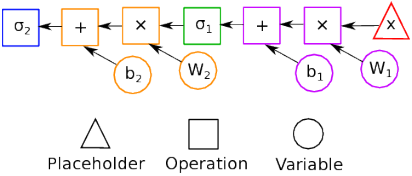
- TensorFlow implements all needed **mathematical operations for multi-threaded CPU and multi GPU** environments.
- Computation of neural networks using data flow graphs is a perfect match!

> ***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.*

Basic blocks to build graphs in TensorFlow

- **Basic blocks:**
  - **Placeholders:** Used for injecting data into the graph, e.g., the inputs x of the neural network
  - **Variables:** Free parameters of the graph, e.g., the weight matrices W of the neural network
  - **Operations:** Functions that operate on data in the graph, e.g., the matrix multiplication of W_1 and x





Run the graph in a TensorFlow session

- A **graph** in TensorFlow can be run inside a **session**.
- Following example calculates $y = W \cdot x$ using TensorFlow:

Computational graph:

$$y = W \cdot x = \begin{pmatrix} 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} = 11$$

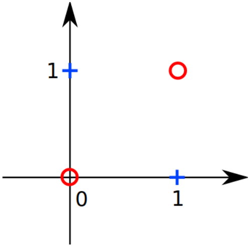
TensorFlow code:

```
import tensorflow as tf
import numpy as np

# Build the graph y = W * x
x = tf.placeholder(tf.float32) # A placeholder
W = tf.get_variable("W", initializer=[[1.0, 2.0]]) # A variable
y = tf.matmul(W, x) # An operation

with tf.Session() as sess: # The session
    sess.run(tf.global_variables_initializer()) # Initialize variables
    result = sess.run(y, feed_dict={x: [[3.0], [4.0]]}) # Run graph
```

Example: XOR-solution with TensorFlow

Path to notebook: `tensorflow/xor.ipynb`

Scenario: Solving the separation of the blue crosses and red circles using a neural network implemented in TensorFlow



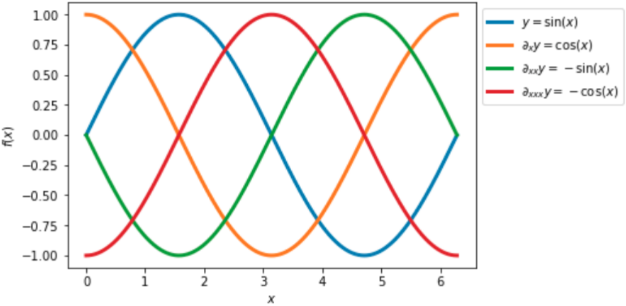
Content:

- Usage of placeholders, variables and operations to build a graph
- Run the graph in a session

Automatic differentiation

- XOR example covers only the inference (forward-pass) part of TensorFlow.
- Training includes optimization of weights using the back-propagation algorithm.
- Excessive use of gradients during training!

How can we compute the gradient of a graph?



Automatic differentiation (2)

- (Almost) each operation in TensorFlow is shipped with an inbuilt gradient.
- Computation of full gradient using the chain-rule of derivatives:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

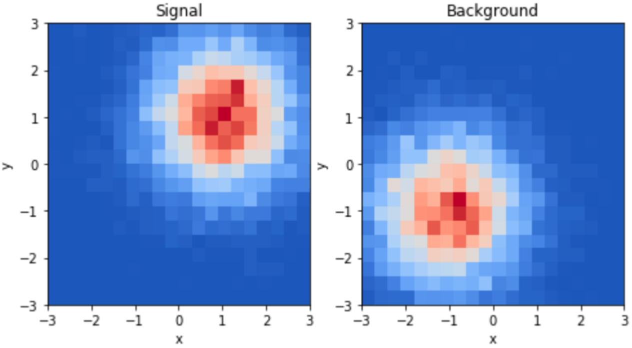
- Explicit TensorFlow call: `tensorflow.gradients(z, x)`

Path to notebook:

`tensorflow/automatic_differentiation.ipynb`

Example: Full training tool-chain in TensorFlow

Path to notebook: `tensorflow/gaussian.ipynb`



Let's try to identify following steps:

1. Definition of neural network model
2. Implementation of loss function and optimizer algorithm
3. Training loop

Advanced: Efficient input pipelines in TensorFlow

- TensorFlow is designed to perform **highly-efficient computations** and ships **many useful features** (Documentation).
- Pick out one of the most-frequently needed: **Data-loading**

- Data-loading often bottleneck if not all data fits in memory (very common for image processing!)
- TensorFlow provides **input piplines** directly **inbuilt in the graph**.
- **Full utilization of CPU/GPU** by loading data form disk in **queues** in memory concurrently

Path to notebook: `tensorflow/queues.ipynb`

And many more features ...

Further reading: Stanford course about TensorFlow

- Very well done and highly entertaining course!
- Lecturer working in the field (OpenAI, DeepMind, Google, ...)
- Small Keras part held by Francois Chollet (author of Keras!)

Link: https://web.stanford.edu/class/cs20si/syllabus.html

Rapid development of neural network applications using Keras

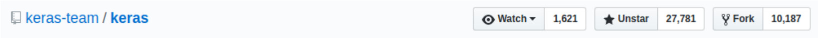
What is Keras?

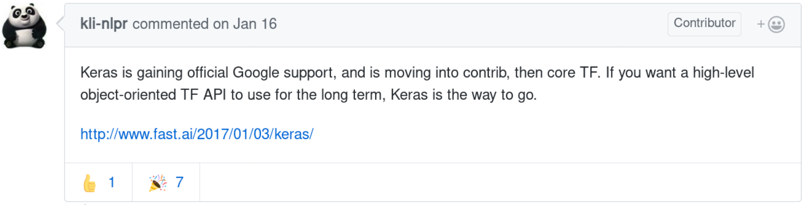
- (Most) popular tool to train and apply neural networks
- **Python wrapper around multiple numerical computation libaries**, e.g., TensorFlow
- Hides most of the low-level operations that you don't want to care about.
- **Sacrificing little functionality** for much easier user interface
- **Backends:** TensorFlow, Theano, CNTK

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

Why Keras and not one of the other wrappers?

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





- Read the full story here: Link

Comparison of TensorFlow and Keras

Same model set up in TensorFlow and Keras:

TensorFlow:

```
def model(x):
    with tf.variable_scope("model") as scope:
        w1 = tf.get_variable('w1', shape=(2, 100), dtype=tf.float64,
                initializer=tf.random_normal_initializer(stddev=0.1))
        b1 = tf.get_variable('b1', shape=(100), dtype=tf.float64,
                initializer=tf.constant_initializer(0.1))
        w2 = tf.get_variable('w2', shape=(100, 1), dtype=tf.float64,
                initializer=tf.random_normal_initializer(stddev=0.1))
        b2 = tf.get_variable('b2', shape=(1), dtype=tf.float64,
                initializer=tf.constant_initializer(0.1))

    l1 = tf.nn.relu(tf.add(b1, tf.matmul(x, w1)))
    logits = tf.add(b2, tf.matmul(l1, w2))
    return logits, tf.sigmoid(logits)

x = tf.placeholder(tf.float64, shape=[None, 2])
logits, f = model(x)
```

Keras:

```
model = Sequential()
model.add(Dense(100, activation="relu", input_dim=2))
model.add(Dense(1, activation="sigmoid"))
```

Compare following notebooks for full code example:

Path to TensorFlow notebook: `tensorflow/gaussian.ipynb`

Path to Keras notebook: `keras/gaussian.ipynb`

Configure the Keras backend

Two ways to configure Keras backend (Theano, TensorFlow or CNTK):

1. Using **environment variables**
2. Using **Keras config file** in `$HOME/.keras/keras.json`

Example setup using environment variables:

Shell:

```
export KERAS_BACKEND=tensorflow
python your_script_using_keras.py
```

Inside a Python script:

```
from os import environ
environ['KERAS_BACKEND'] = 'tensorflow'
```

Example Keras config using TensorFlow as backend:

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

Model definition with Keras

Path to notebook: `keras/gaussian.ipynb`

Model definition can be performed with two APIs:

Sequential model: Stacking layers sequentially

```
model = Sequential()
model.add(Dense(100, activation="relu", input_dim=2))
model.add(Dense(1, activation="sigmoid"))
```

Functional API: Multiple input/output models, ...

```
inputs = Input(shape=(2,))
hidden_layer = Dense(100, activation="relu")(inputs)
outputs = Dense(1, activation="sigmoid")(hidden_layer)
model = Model(inputs=inputs, outputs=outputs)
```

model.summary()

Path to notebook: `keras/gaussian.ipynb`

Very useful convenience method:

```
model.summary()
```

```
=================================================================
dense_1 (Dense)              (None, 100)               300
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 101
=================================================================
Total params: 401
Trainable params: 401
Non-trainable params: 0
```

Easy to keep track of **model complexity**.

Setting optimizer, loss and validation metrics

Path to notebook: `keras/gaussian.ipynb`

Single line of code:

```
model.compile(
        loss="binary_crossentropy", # Loss function
        optimizer="adam",           # Optimizer algorithm
        metrics=["accuracy"]        # Validation metric
        )
```

Available layers, losses, optimizers, . . .

- There's **everything you can imagine**, and it's **well documented**.
- Have a look: www.keras.io

Training in Keras

Path to notebook: `keras/gaussian.ipynb`

Again, **single line of code**:

```
model.fit(data_train, labels_train,
          validation_data=(data_val, labels_val),
          batch_size=100,
          epochs=100
          )
```

Save, load and apply the trained model

Save model:

- Models are **saved as `HDF5` files**: `model.save("model.h5")`
  - Combines description of weights and architecture in a single file
- **Alternative**: Store weights and architecture separately
  - Store weights: `model.save_weights("model_weights.h5")`
  - Store architecture: `json_dict = model.to_json()`

Load model:

```
from keras.models import load_model
model = load_model("model.h5")
```

Apply model:

```
predictions = model.predict(inputs)
```

Full example using the MNIST dataset

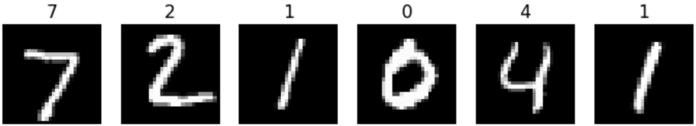
Path to notebook: `keras/mnist_train.ipynb`

- **MNIST dataset?**
  - **Task:** Predict the number on an image of a handwritten digit
  - **Official website:** Yann LeCun's website (Link)
  - Database of **70000 images of handwritten digits**
  - 28x28 pixels in gray-scale as input, digit as label

- **Data format:**
  - **Inputs:** 28x28 matrix with floats in [0, 1]
  - **Target:** One-hot encoded digits, e.g., 2 $\rightarrow$ [0 0 1 0 0 0 0 0 0 0]

Application on handwritten digits

Path to notebook: `keras/mnist_apply.ipynb`



If you are bored on your way home:

1. Open with GIMP `keras/your_own_digit.xcf`
2. Dig out your most beautiful handwriting
3. Save as PNG and run your model on it

Training with callbacks

Path to notebook: `keras/mnist_train.ipynb`

- **Callbacks** are executed before and/or after each training epoch.
- Numerous **predefined** callbacks are available, **custom** callbacks can be implemented.

Definition of model-checkpoint callback:

```
# Callback for model checkpoints
checkpoint = ModelCheckpoint(
        filepath="mnist_example.h5", # Output similar to model.save("mnist_example.h5")
        save_best_only=True) # Save only model with smallest loss
```

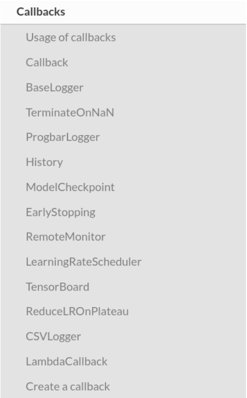
Register callback:

```
model.fit(inputs, targets,
        batch_size=100,
        epochs=10,
        callbacks=[checkpoint]) # Register callbacks
```

Training with callbacks (2)

Path to notebook: `keras/mnist_train.ipynb`

- Commonly used callbacks for improvement, debugging and validation of the training progress are implemented, e.g., **`EarlyStopping`**.
- Powerful tool: **`TensorBoard`** in combination with TensorFlow
- Custom callback: **`LambdaCallback`** or write callback class extending base class `keras.callbacks.Callback`



Advanced: Customize Keras

Path to notebook:

`keras/custom_loss_metric_callback.ipynb`

- Keras is highly customizable!
- Easily define **own loss function, metrics and callbacks**

```
import keras.backend as K

def custom_loss(y_true, y_pred):
    return K.mean(K.square(y_pred - y_true), axis=-1)

def custom_metric(y_true, y_pred):
    return K.mean(K.square(y_pred - y_true), axis=-1)

model.compile(
        loss=custom_loss,
        metrics=[custom_metric],
        optimizer="adam")
```

Advanced: Training on “big data”

- The call `model.fit(inputs, targets, ...)` expects all `inputs` and `targets` to be already loaded in memory.
  $\rightarrow$ Physics applications have often data on Gigabyte to Terabyte scale!

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(inputs, targets, ...)
```

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

```python
def generator_function():
    while True:
        inputs, labels = custom_load_next_batch()
        yield inputs, labels

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

Path to notebook: `keras/fit_generator.ipynb`

Note: Data preprocessing in Keras applications

- Some preprocessing methods are included in Keras, but mainly for text and image inputs.
- **Better option:** Using `scikit-learn` package (Link to preprocessing module)

```
from sklearn.preprocessing import StandardScaler
preprocessing = StandardScaler()
preprocessing.fit(inputs)
preprocessed_inputs = preprocessing.transform(inputs)
```

Deep learning on the HIGGS dataset

One of the most often cited papers about deep learning in combination with a physics application:

Searching for Exotic Particles in High-Energy Physics with Deep Learning
Pierre Baldi, Peter Sadowski, Daniel Whiteson

- **Topic:** Application of deep neural networks for separation of signal and background in an exotic Higgs scenario
- **Results:** Deep learning neural networks are more powerful than “shallow” neural networks with only a single hidden layer.

Let’s reproduce this with minimal effort using Keras!

Deep learning on the HIGGS dataset (2)

Path to notebook: `keras/HIGGS_train.ipynb`

Dataset:

- Number of events: 11M
- Number of features: 28

Shallow model:

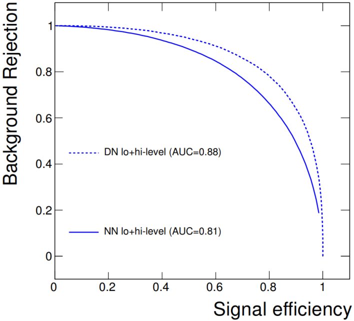
```
model_shallow = Sequential()
model_shallow.add(Dense(1000, activation="tanh", input_dim=(28,)))
model_shallow.add(Dense(1, activation="sigmoid"))
```

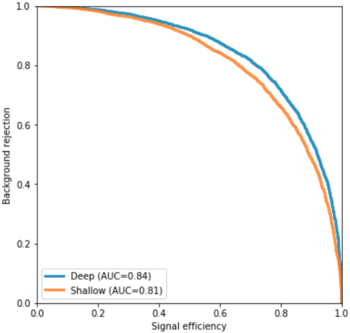
Deep model:

```
model_deep = Sequential()
model_deep.add(Dense(300, activation="relu", input_dim=(28,)))
model_deep.add(Dense(300, activation="relu"))
model_deep.add(Dense(300, activation="relu"))
model_deep.add(Dense(300, activation="relu"))
model_deep.add(Dense(300, activation="relu"))
model_deep.add(Dense(1, activation="sigmoid"))
```

Deep learning on the HIGGS dataset (3)

Path to notebook: `keras/HIGGS_test.ipynb`





- Shallow model matches performance in the paper, but deep model can be improved.
  $\rightarrow$ **Try to improve it!** But you'll need a decent GPU. . .
- Keras allows to **reproduce this result with a total of about 130 lines of code.**

Further reading: `keras.io` examples

- **Repository:** `http://www.github.com/keras-team/keras`
- **Folder:** `keras/examples/`

Recommendations:

- **`addition_rnn.py`**: Application of a RNN parsing strings such as `"535+61"` and returning the actual result 596, runs on a consumer CPU
- **`neural_style_transfer.py`**: Transfers the style of a reference image to an input image, needs a decent GPU