Fermilab Keras Workshop

Stefan Wunsch
stefan.wunsch@cern.ch

December 8, 2017

What is this talk about?

- Modern implemenation, description and application of neural networks
- Currently favoured approach:
  - **Keras** used for **high-level description** of neural networks models
  - **High-performance implementations** provided by backends, e.g., Theano or **TensorFlow** libraries

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

Outline

The workshop has these parts:

1. Brief introduction to **neural networks**
2. Brief introduction to **computational graphs** with TensorFlow
3. Introduction to **Keras**
4. **Useful tools** in combination with Keras, e.g., TMVA Keras interface

- In parts 3 and 4, you have to possibility to follow along with the examples on your laptop.

Assumptions of the tutorial:

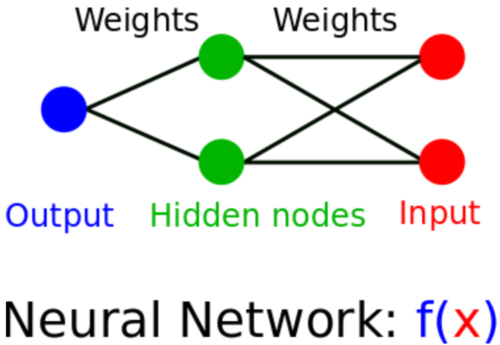
- You are not a neural network expert, but you know roughly how they work.
- You haven't used Keras before.
- You want to know why Keras is so popular and how you can use it!

You can download the slides and code examples from GitHub:

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

Brief Introduction to Neural Networks

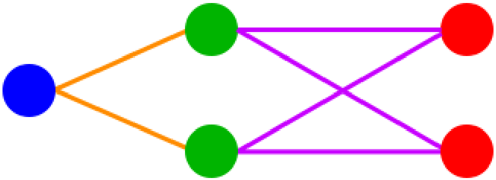
A Simple Neural Network



- **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.

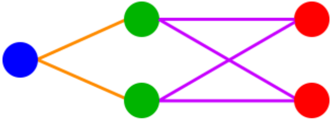
Mathematical Representation

- **Why do we need to know this?**
  → Keras backends TensorFlow and Theano implement these mathematical operations explicitely.
  → Basic knowledge to understand Keras' high-level layers



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

Mathematical Representation (2)



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

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

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

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

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

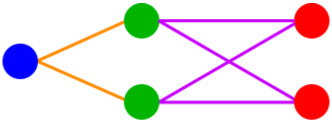
Activation is applied elementwise!

The “simple” neural network written as full equation:

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

- **How many parameters can be altered during training?**
  $\rightarrow$ 1+2+2+4=9 parameters

Training (Short Reminder)



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

Training:

1. **Forward-pass** of a **batch** of N inputs x_i calculating the outputs $f_{\mathrm{NN,i}}$
2. **Comparison of outputs** $f_{\mathrm{NN,i}}$ with true value $f_{\mathrm{Target,i}}$ using the **loss function** as metric
3. **Adaption of free parameters** to improve the outcome in the next forward-pass using the gradient from the **back-propagation** algorithm in combination with an **optimizer algorithm**

Common loss functions:

- Mean squared error: $\frac{1}{N}\sum_{i=1}^{N}(f_{NN,i} - f_{Target,i})^2$
- Cross-entropy: $-\sum_{i=1}^{N} f_{Target,i} \log(f_{NN,i})$

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

Brief Introduction to Computational Graphs With TensorFlow

Motivation

- Keras wraps and simplifies usage of libraries, which are optimized on efficient computations, e.g., TensorFlow.

- How do modern numerical computation libraries such as Theano and TensorFlow work?

Theano? TensorFlow?

- **Libraries for large-scale numerical computations**
- TensorFlow is growing much faster and gains more support (Google does it!).

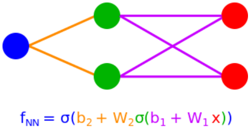
Theano / **Theano** Watch 512 Star 5,893 Fork 2,031

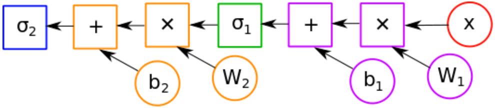
Theano *is a Python library that allows you to* ***define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently****.*

tensorflow / **tensorflow** Watch 4,641 Unstar 50,822 Fork 23,745

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.*

Computational Graphs



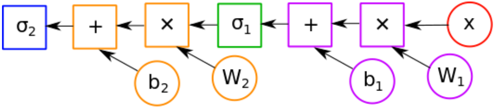


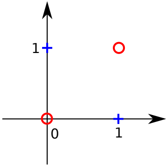
Example neural network $\rightarrow$ **According computational graph**

- 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.*

TensorFlow Implementation of the Example Neural Network





fermilab_keras_workshop/tensorflow/xor.py:

```
w1 = tensorflow.get_variable("W1", initializer=np.array([[1.0, 1.0],
                                                          [1.0, 1.0]]))
b1 = tensorflow.get_variable("b1", initializer=np.array([0.0, -1.0]))
w2 = tensorflow.get_variable("W2", initializer=np.array([[1.0], [-2.0]]))
b2 = tensorflow.get_variable("b2", initializer=np.array([0.0]))

x = tensorflow.placeholder(tensorflow.float64)
hidden_layer = tensorflow.nn.relu(b1 + tensorflow.matmul(x, w1))
y = tensorflow.identity(b2 + tensorflow.matmul(hidden_layer, w2))

with tensorflow.Session() as sess:
    sess.run(tensorflow.global_variables_initializer())
    x_in = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y_out = sess.run(y, feed_dict={x:x_in})
```

→ **Already quite complicated for such a simple model!**

TensorFlow Implementation of the Example Neural Network (2)

- Plain TensorFlow implements only the mathematical operations.
- Combination of these operations to a neural network model is up to you.
- Already quite complicated for a simple neural network model without definition of loss function, training procedure, ...

- **Solution 1:** Write your own framework to simplify TensorFlow applications
- **Solution 2:** Use wrapper such as Keras with predefined layers, loss functions, ...

Introduction to Keras

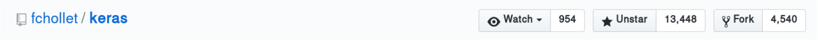
What is Keras?

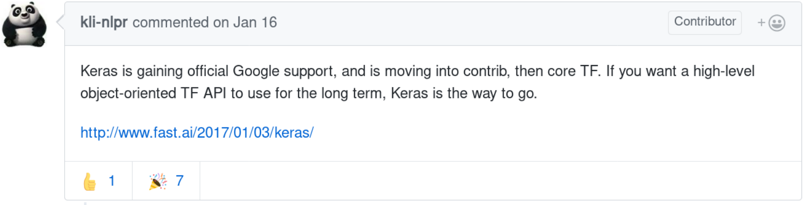
- Most popular tool to train and apply (deep) neural networks
- **Python wrapper around multiple numerical computation libaries**, e.g., TensorFlow
- Hides most of the low-level operations that you don't want to care about.
- **Sacrificing little functionality** for much easier user interface

- **Backends:** TensorFlow, Theano
- **NEW:** Microsoft Cognitive Toolkit (CNTK) added as backend

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.
- Looks 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.
- **But** you'll learn most by taking the examples as starting point and play around at home.

Download all files:

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

Set up the Python virtual environment:

```
cd fermilab_keras_workshop
bash init_virtualenv.sh
```

Enable the Python virtual environment:

```
# This has to be done in every new shell!
source py2_virtualenv/bin/activate
```

Keras Basics

Configure 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:

Terminal:

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

Inside a Python script:

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

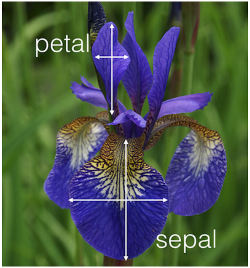
Example Keras config using TensorFlow as backend:

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

Example Using the Iris Dataset

- Next slides will introduce the basics of Keras using the example `fermilab_keras_workshop/keras/iris/train.py`.

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



Model Definition

- **Two types of models**: `Sequential` and the functional API
  - `Sequential`: Simply stacks all layers
  - Funktional API: You can do everything you want (multiple inputs, multiple outputs, ...).

```
# Define model
model = Sequential()

model.add(
    Dense(
        8, # Number of nodes
        kernel_initializer="glorot_normal", # Initialization
        activation="relu", # Activation
        input_dim=(4,) # Shape of inputs, only needed for the first layer!
    )
)

model.add(
    Dense(
        3, # Number of output nodes has to match number of targets
        kernel_initializer="glorot_uniform",
        activation="softmax" # Softmax enables an interpretation of the outputs as probabilities
    )
)
```

Model Summary

- **`model.summary()`** prints a description of the model
- **Extremely useful** to keep track of the number of free parameters

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 8)                 40
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 27
=================================================================
Total params: 67
Trainable params: 67
Non-trainable params: 0
_________________________________________________________________
```

Define Loss Function, Optimizer, Validation Metrics...

- Everything is set in a single function, called the **`compile`** step of the model.
- Validation is performed after each training epoch (next slides).

```
# Set loss, optimizer and evaluation metrics
model.compile(
        loss="categorical_crossentropy", # Loss function
        optimizer=SGD(lr=0.10), # Optimizer algorithm
        metrics=["accuracy",]) # Validation metric(s)
```

Data Preprocessing

- Some preprocessing steps are included in Keras, but mainly for text and image inputs.
- **Better option:** Using `scikit-learn` package (Link to `preprocessing` module)
- **Single input** (4 features)**:** [5.1, 3.5, 1.4, 0.2]
  - Needs to be scaled to the order of 1 to fit the activation function.
- **Single output** (3 classes)**:** [1 0 0]
- **Common preprocessing:** Standardization of inputs
  $\rightarrow$ Operation: $\frac{\text{input}-\text{mean}}{\text{standard deviation}}$

```
# Set up preprocessing
from sklearn.preprocessing import StandardScaler
preprocessing = StandardScaler()
preprocessing.fit(inputs)
inputs = preprocessing.transform(inputs)
```

Training

- Training is again a single call of the `model` object, called **`fit`**.

```
# Train
model.fit(
    inputs, # Preprocessed inputs
    targets_onehot, # Targets in 'one hot' shape
    batch_size=20, # Number of inputs used for
                   # a single gradient step
    epochs=10) # Number of cycles of the full
               # dataset used for training
```

That's it for the training!

Training (2)

```
Epoch 1/10
150/150 [==============================] - 0s 998us/step - loss: 1.1936 - acc: 0.2533
Epoch 2/10
150/150 [==============================] - 0s 44us/step - loss: 0.9904 - acc: 0.5867
Epoch 3/10
150/150 [==============================] - 0s 61us/step - loss: 0.8257 - acc: 0.7333
Epoch 4/10
150/150 [==============================] - 0s 51us/step - loss: 0.6769 - acc: 0.8267
Epoch 5/10
150/150 [==============================] - 0s 49us/step - loss: 0.5449 - acc: 0.8933
Epoch 6/10
150/150 [==============================] - 0s 53us/step - loss: 0.4384 - acc: 0.9267
Epoch 7/10
150/150 [==============================] - 0s 47us/step - loss: 0.3648 - acc: 0.9200
Epoch 8/10
150/150 [==============================] - 0s 46us/step - loss: 0.3150 - acc: 0.9600
Epoch 9/10
150/150 [==============================] - 0s 54us/step - loss: 0.2809 - acc: 0.9267
Epoch 10/10
150/150 [==============================] - 0s 49us/step - loss: 0.2547 - acc: 0.9200
```

Save 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)
```

Wrap-Up

Training:

```
# Load iris dataset
# ...

# Model definition
model = Sequential()
model.add(Dense(8, kernel_initializer="glorot_normal", activation="relu", input_dim=(4,)))
model.add(Dense(3, kernel_initializer="glorot_uniform", activation="softmax"))

# Preprocessing
preprocessing = StandardScaler().fit(inputs)
inputs = preprocessing.transform(inputs)

# Training
model.fit(inputs, targets_onehot, batch_size=20, epochs=10)

# Save model
model.save("model.h5")
```

Application:

```
# Load model
model = load_model("model.h5")

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

That's a full training/application workflow in less than ten lines of code!

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`

Advanced Usage of Keras

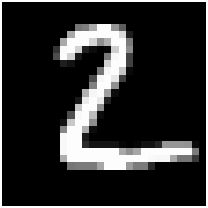
Example Using the MNIST Dataset

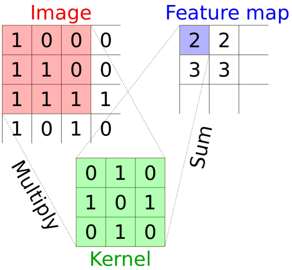
- Example in the repository:
  `fermilab_keras_workshop/keras/mnist/train.py`

- **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 greyscale 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]

Short Introduction to Convolutional Layers





- **Kernel:** Locally connected dense layer
- **Convolution:** Kernel moves similar to a sliding window over the image
- **Feature map:** Output "image" after application of the kernel

```
model = Sequential()

model.add(
    Conv2D(
        4, # Number of kernels/feature maps
        (3, # column size of sliding window used for convolution
        3), # row size of sliding window used for convolution
        activation="relu" # Rectified linear unit activation
    )
)
```

Model Definition

fermilab_keras_workshop/keras/mnist/train.py:

```
model = Sequential()

# First hidden layer
model.add(
    Conv2D(
        4, # Number of output filters or so-called feature maps
        (3, # column size of sliding window used for convolution
        3), # row size of sliding window used for convolution
        activation="relu", # Rectified linear unit activation
        input_shape=(28,28,1) # 28x28 image with 1 color channel
    )
)

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

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

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

Model Summary

- Detailed summary of model complexity with `model.summary()`

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 4)         40
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 4)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 676)               0
_________________________________________________________________
dense_1 (Dense)              (None, 16)                10832
_________________________________________________________________
dense_2 (Dense)              (None, 10)                170
=================================================================
Total params: 11,042
Trainable params: 11,042
Non-trainable params: 0
_________________________________________________________________
```

Training With Validation Metrics

- Validation metrics are evaluated after each training epoch.
- In `compile` step, multiple **predefined validation metrics** can be booked, e.g., `accuracy`.
- **Custom metrics** are possible.

Booking a predefined metric:

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

Training with validation data:

```
model.fit(inputs, targets, validation_split=0.2) # Use 20% of the data for validation
```

```
Epoch 1/10
30000/30000 [==============================] - 6s 215us/step - loss: 0.8095 - acc: 0.7565
- val_loss: 0.3180 - val_acc: 0.9085
Epoch 2/10
...
```

Training With Callbacks

- **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:

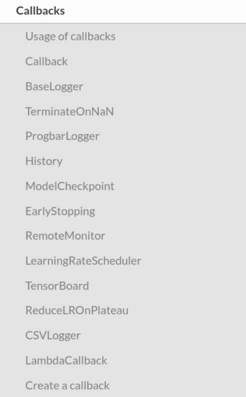
```
# Callback for model checkpoints
checkpoint = ModelCheckpoint(
        filepath="mnist_example.h5", # Output similar to model.save("mnist_example.h5")
        save_best_only=True) # Save only model with smallest loss
```

Register callback:

```
model.fit(inputs, targets, # Training data
        batch_size=100, # Batch size
        epochs=10, # Number of training epochs
        callbacks=[checkpoint]) # Register callbacks
```

Training With Callbacks (2)

- 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 Training Methods for 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, ...)
```

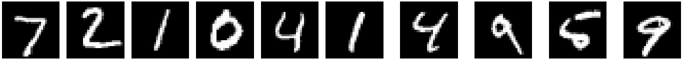
- Training with data from a **Python generator**:

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

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

Application on Handwritten Digits

- **PNG images of handwritten digits** are placed in `fermilab_keras_workshop/keras/mnist/example_images/`, have a look!



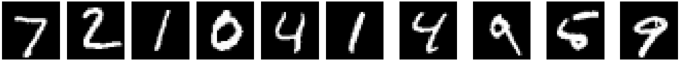
- Let's **apply our trained model** on the images:

```
./keras/mnist/apply.py keras/mnist/example_images/*.png
```

- **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)

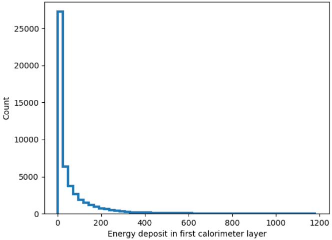
```
Predict labels for images:
    keras/mnist/example_images/example_input_0.png : 7
    keras/mnist/example_images/example_input_1.png : 2
    keras/mnist/example_images/example_input_2.png : 1
    keras/mnist/example_images/example_input_3.png : 0
    keras/mnist/example_images/example_input_4.png : 4
    keras/mnist/example_images/example_input_5.png : 1
    keras/mnist/example_images/example_input_6.png : 4
    keras/mnist/example_images/example_input_7.png : 9
    keras/mnist/example_images/example_input_8.png : 6
    keras/mnist/example_images/example_input_9.png : 9
```

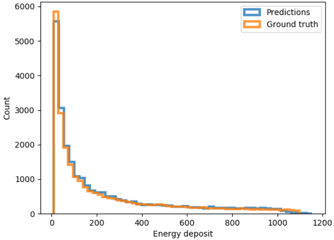


Examples with Physics Data

Toy Calorimeter

- Data represent measurements in a toy-calorimeter
  - **Inputs:** 13 calorimeter layers with energy deposits
  - **Target:** Reconstruction of total energy deposit
- Example in repository:
  `fermilab_keras_workshop/keras/calorimeter/train.py`





Implemented regression model:

```
model = Sequential()
model.add(Dense(100, activation="tanh", input_dim=(13,)))
model.add(Dense(1, activation="linear"))
```

- **Source:** Link

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)

Files:

- fermilab_keras_workshop/keras/HIGGS/train.py
- fermilab_keras_workshop/keras/HIGGS/test.py

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"))
```

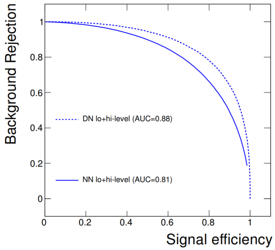
Training:

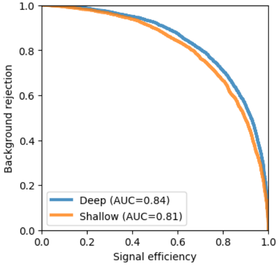
```
model.compile(loss="binary_crossentropy", optimizer=Adam(), metrics=["accuracy"])

model.fit(preprocessed_inputs, targets,
    batch_size=100, epochs=10, validation_split=0.25)
```

Deep Learning on the HIGGS Dataset (3)

- Weights of deep and shallow model are part of the repository.





- 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 130 lines of code**:

```
# Count lines of code
$ wc -l keras/HIGGS/*.py
  62 keras/HIGGS/test.py
  68 keras/HIGGS/train.py
 130 total
```

Useful Tools In Combination With Keras

TMVA Keras Interface

Prerequisites

- **Keras inteface integrated in ROOT/TMVA since v6.08**

- Example for this tutorial is placed here:
  `fermilab_keras_workshop/tmva/`

- You need ROOT with enabled PyROOT bindings. Easiest way to test the example is using CERN's **lxplus** machines:
  - `ssh -Y you@lxplus.cern.ch`
  - Source software stack LCG 91

How to source LCG 91 on lxplus:

```
source /cvmfs/sft.cern.ch/lcg/views/LCG_91/x86_64-slc6-gcc62-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?

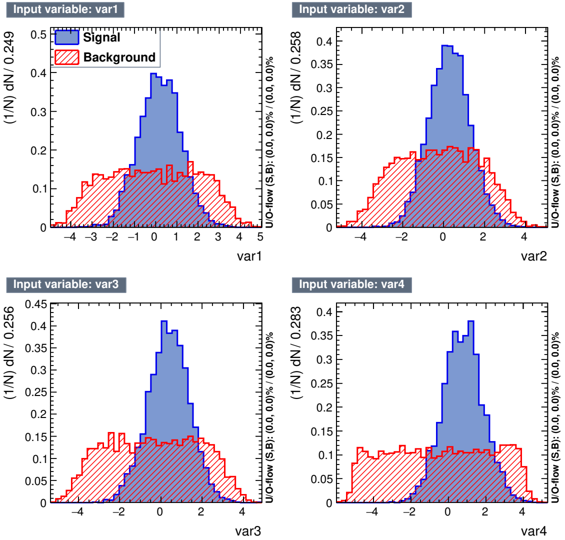
1. **Model definition** done in **Python** using **Keras**
2. **Data management**, **training** and **evaluation** within the TMVA framework
3. **Application** using the TMVA reader

- 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**:

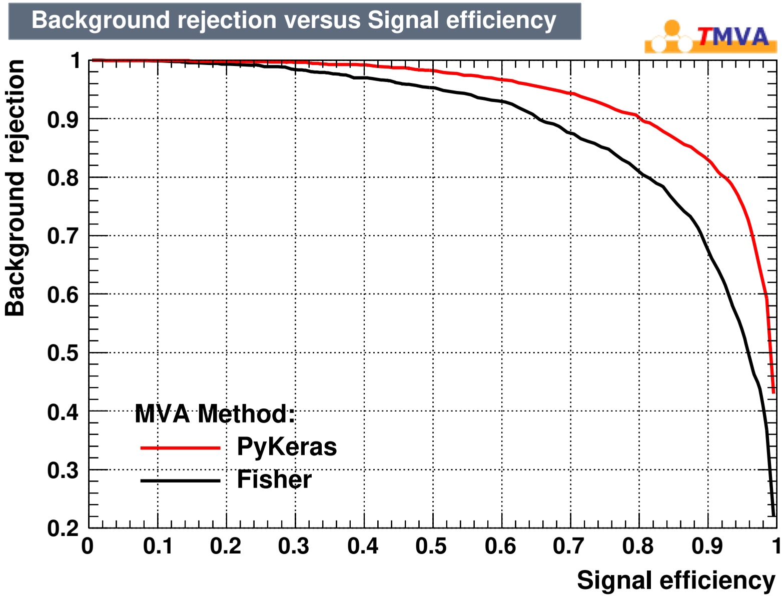
```
factory.BookMethod(dataloader, TMVA.Types.kPyKeras, 'PyKeras',
        'H:V:VarTransform=G:'+
        'Verbose=1'+\ # Training verbosity
        'FilenameModel=model.h5:'+\ # Model from definition
        'FilenameTrainedModel=modelTrained.h5:'+\ # Optional!
        'NumEpochs=10:'+\
        'BatchSize=32'+\
        '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!

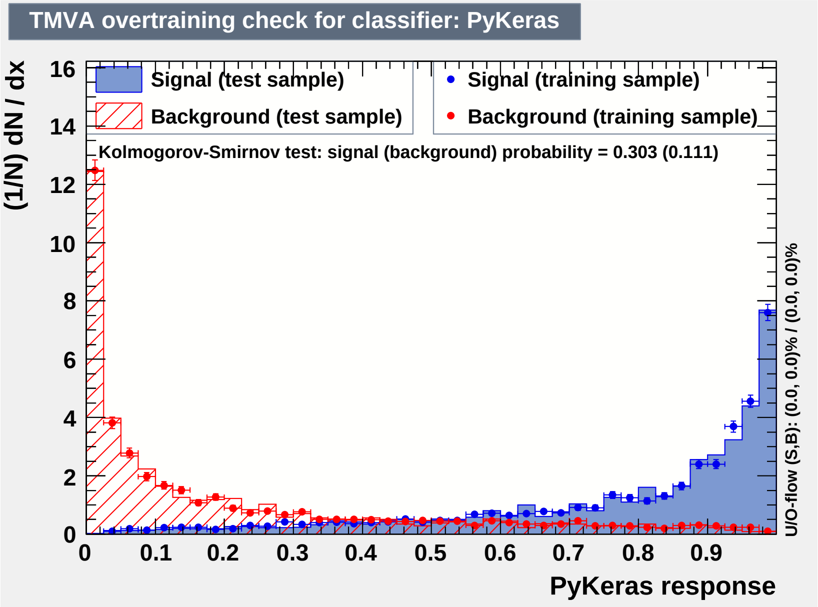
```
python tmva/BinaryClassification.py
```

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

Training Results: ROC



Training Results: Overtraining Check



Application

- **Does not differ from any other TMVA method!**

- **Example** application can be found here:
  `fermilab_keras_workshop/`
  `tmva/ApplicationBinaryClassification.py`

Application (2)

Run `python tmva/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
```

lwtnn with Keras

What is lwtnn?

- **Core problem:** TensorFlow and others are not made for event-by-event application!
- **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
- **Tutorial** can be found here: `https://github.com/stwunsch/iml_keras_workshop`

Load and Apply Model in C++ Using lwtnn

Convert trained Keras model to `lwtnn` JSON:

→ See the tutorial and README!

Load model:

```cpp
// 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:

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

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