[5LSL0] Keras tutorial

Rik Vullings
Ruud van Sloun
Bart van Erp

July 29, 2019

1 Introduction

Machine learning

In many scientific fields the introduction of artificial intelligence has proven to be extremely useful. Artificial intelligence is the field of study where the goal is to make computers behave like intelligent creatures. Machine learning is a subfield of artificial intelligence, where the main focus is to make computers learn from previously observed data. The goal of machine learning is to mimic a process, which transforms a set of data into another.

Fig. 1 gives a very generic overview of the machine learning process. First a data set is required of a specific process that we can mimic. This process can be as simple as a XOR-operation or as complicated as detecting malignant tissue in medical images. For the given process it is required to collect input data x and the corresponding output data y. Through a series of mathematical operations, called a model or network, the goal is the manipulate our input data (e.g. video recording of a self-driving vehicle) to our desired output data (e.g. steering action). This manipulation depends on many free parameters or weights in the various mathematical operations. After processing the input data through the network a prediction of the desired output$\hat{y}$ is established. From the difference between the predicted output and the desired output an error or cost function J can be defined, which gives a measure of the similarity between the outputs. Optimization of this error function allows us to alter the weights of the network in order to reduce this error and to make the model mimic the underlying process as good as possible.

This reader will discuss the main workflow of training a machine learning network for any given problem. This workflow will be described in the Python programming language. First the data preparation is discussed, after which a network is created using the Keras library. After this the network is trained and evaluated. Finally the reader will discuss some commonly made mistakes that should help you make sense of most error messages.

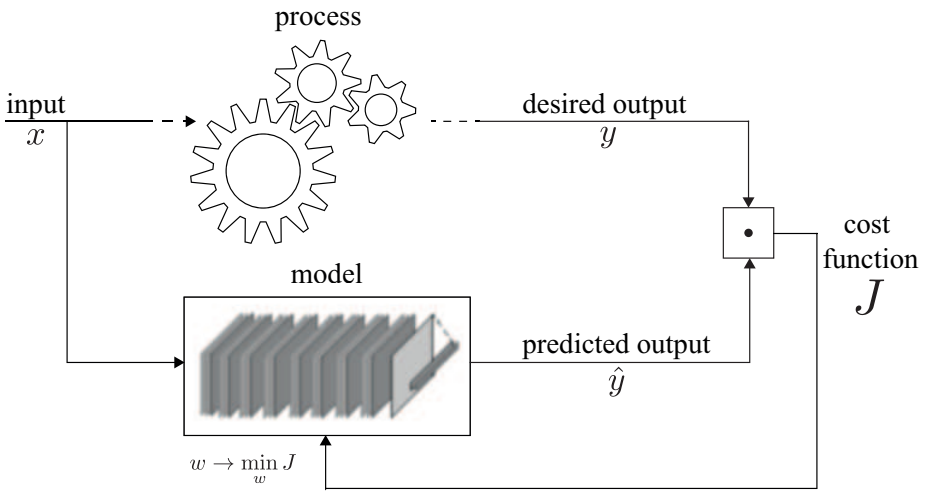

Figure 1: General overview of the machine learning workflow.

Tensorflow

Most machine learning networks contain tens to hundreds of layers and can have millions of parameters to train. In order to perform all mathematical operations efficiently and speed up the training of network significantly Tensorflow is used. Tensorflow is a Python library which allows for the exploitation of parallelism. In modern CPU's and especially in GPU's parallelism is used intensively in order to perform multiple mathematical at the same time. The name Tensorflow could be regarded as the optimization of the flow (i.e. the processing efficiency) of tensor (which are multi-dimensional matrices) operations on a computer.

Keras

Where Tensorflow allows for the efficient processing of tensor operations, Keras allows for the simple conversion of high-level model design and manipulations to all tensor operations that are involved in order to do so.

2 Data preparation

For the training of the network data has to be fed into the network and the corresponding prediction has to be compared to the desired output. In order to do so, the network is trained for several epochs, which can be regarded as a single iteration on the entire data set. During an epoch several batches of data are fed to the network. A batch is a finite subset of the data set. Usually batches are significantly smaller than the data set, because of memory limitations. Two main approaches can be distinguished for feeding the data into the network.

```python
import os
import random
import numpy as np

def generator():

    # set parameters of generator
    DATA_PATH = "C:/path/to/data/folder"
    data_files = os.listdir(DATA_PATH)
    batch_size = 32
    data_size = 1000

    # create generator by an infinite loop
    while True:
        # create an empty batch of data
        batch = np.empty(shape=(batch_size, data_size))
        # fill the batch variable with the data files
        for batch_item in range(batch_size):
            # get the index of a random data file
            index = random.randint(0, len(data_files)-1)
            # load the random data file
            batch[batch_item] = np.load(DATA_PATH+"/"+data_files[index])
        # return the batch of data
        yield batch
```

Listing 1: A simple generator function.

Approach 1: loading all data

The first approach involves first loading the entire data set into the memory and then selecting batches of this data set for the training of the network. Because the data is already loaded into memory (RAM) it will speed up training significantly, since the algorithm does not need to access slower memory devices (such as HDD's). However, this method has the problem that usually a large memory is required to hold the entire data set.

Approach 2: creating a generator

A second approach that does not involve loading all data at once can be more useful for memory-limited devices. In Python a generator object can be created, which can be regarded as a function that returns a batch of (processed) data when called. Each time the generator function is called, a new batch of data is provided. Suppose you have a directory with all your data files. From this directory you would like create a batch of data consisting out of 32 random files.

Listing 1 shows an example code of a very simple generator function. First the function creates an array of all possible data files. Then the function will jump into an infinite loop (this is intentional!) and create a batch of 32 randomly loaded files. The only function that might be new is the `yield` function. This function can be regarded as the return function, however, the function does not automatically return a batch. It requires another external function to be called on the function before returning a batch.

Listing 2 shows an example code on how the generator is called. First the generator function is imported and assigned to a variable. Now in order to get

```
from generator_file import generator

# initialize generator
gen = generator()

# fetch a batch of data from generator
batch1 = next(gen)
batch2 = next(gen)
...
```

Listing 2: The call of a generator function.

a batch of data the `next()` function is called on the generator function. This will activate the `yield` function of the generator and return a batch of data, after which a new batch is automatically being processed.

Creating batches

The batch sizes that are returned can be picked arbitrarily. Large batch sizes are characterized by a more smooth training curve, because gradients are calculated over more data, but require larger memory. Similarly small batch sizes are characterized by a more noisy training curve and by smaller memory usage.

It is important to format the batches correctly in terms of size for the training to go properly. The shape of a batch of data (can be determined by `np.shape(batch)`) is a tuple of the batch size followed by the data shape. In order to satisfy this condition the functions `np.expand_dims()` and `np.squeeze()` might prove useful.

Advanced generators

Preprocessing

It has been proven in multiple scientific papers that the pre-processing of the data shows significant improvements in the performance of the network. Furthermore it might be possible that the data is not ideal and should be conditioned before use (e.g. the data lengths should be equal). These additional steps can be implemented in the generator itself by simply extending the code in Listing 1.

Keras generator class

One of the downsides of the generator in Listing 1 is that there is no guarantee that the generator will iterate on the data set once per epoch, before returning data files for the second time. It might now be possible that a particular data file is return more often than another.
In order to prevent this from happening, the online available Keras generator class can be used in order to add functionality to a generator.

```
# instantiate model object
model = Sequential()
# add layers
model.add(Dense(16, input_dim=5, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

Listing 3: A simple sequential network.

3 Creating your network

Now the basis of data preparation has been discussed, it is time to start creating machine learning networks. The Keras library makes it extremely easy to build a network from scratch. Two different types of networks exist in the Keras terminology.

The sequential network

The first type of networks concerns all networks where mathematical operations are solely applied one after another to the same set of data without any combinations of the streams of data in the network. This type of network is called the sequential network.

Listing 3 shows a piece of code where a simple model is created. As can be seen, first a model object is instantiated. After this several layers are added in a chronological fashion using the `.add()` function. It is compulsory for the first layer to indicate the shape of the input data (without batch size indication). Activation functions can be included after the layers using the `activation` argument. The Keras documentation elaborates more on the other type of layers with their respective arguments.

Using the `model.summary()` function, a summary of the model is printed. It is always important to check that the output dimensions of the network are the same as the dimensions of the desired output data. If this is not the case then the model should be adapted in order to satisfy this requirement. In some cases the use of the `Flatten` layer may prove to be useful, since it reduces the dimensionality of the data.

More advanced networks

The sequential networks are limited by their simplicity. Sometimes the use of for example residual connections might prove to be useful in the model design. Since these residual connections require us to interconnect several flows of data in the network, another approach has to be taken. In Keras the options exists to create these models, but the syntax is slightly different than for the case of the sequential models. This functional API also allows for the use of multiple inputs in case this may be convenient.

```
# define layers
first_layer = Input(shape=(8,))
second_layer = Dense(64)(first_layer)
third_layer = Dense(64)(second_layer)
fourth_layer = Add()([second_layer, third_layer])
# create model
model = Model(inputs=first_layer, outputs=fourth_layer)
```

Listing 4: A network using the functional API.

```
# load data in memory
x = np.load("input_data.npy")
y = np.load("output_data.npy")

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

# train model
history = model.fit(x, y)
```

Listing 5: Training of a model when all the data is loaded into memory.

Listing 4 shows a basic network with a residual connection using the functional API. The functional API works as follows: several layers are defined. The layer preceding another layer is put in between brackets after the function. In this way, a sequential model can be defined. However, now additional layers may be used such as the `Add` layer. This layer adds two flows of data in the network. In Listing 4 this layer is used to add the output of the second layer to the output of the third layer. Similarly to the sequential model, we are required to define an input layer and to specify its shape. Once all layers are interconnected, the model can be created using the `Model()` function by specifying the first and last layer.

4 Training your network

Now that the data has been processed nicely in batches and the network has been correctly defined, with input and output dimensions that correspond to the data. It is time to start training the network. The training of a network involves many different options to choose from. Some important options will be discussed after the general training procedure has been explained.

The training of a network can be performed using two methods. Listing 5 shows the method were all data is loaded in the memory and Listing 6 uses a data generator. The input data is denoted by `x` and the desired output data is denoted by `y`. The data generator outputs a tuple of the input data and the corresponding desired output data. Both models are compiled, where the settings of the training process can be altered. The type of optimizer is in this case chosen as the mean squared error. The loss is the function which the

```
# create data generator (returns (x,y))
traingen = generator()

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

# train model
history = model.fit_generator(traingen)
```

Listing 6: Training of a network using a data generator.

training procedure tries to minimize. Metrics can be regarded as a measure of the performance of the network. The metrics are not being optimized by the network, but are just there to compare performance over time.

The training of the network itself is performed by the `.fit()` or `.fit_generator()` functions. In this function the data or data generator is passed as an argument. The number of epochs can also be given, which resembles the amount of iterations over the entire train set. For generators the Keras class needs to be used in order to iterate first over the entire set before the second iteration.

The training of the network returns a history object. This object contains the loss as well as the metrics for each epoch. After the training has been completed this can be plotted to show the training curves.

Callbacks

It usually is desired to change parameters during training or to fetch intermediate performance to closely monitor the training of the network. The functions and options that need to be executed during the training are called callbacks. Listing **??** shows how callbacks are implemented in the train functions. The most important callbacks include: ModelCheckpoint, which saves a copy of the model after a certain interval of time; LearningRateScheduler, which allows you to create a schedule to change the learning rate over time; TensorBoard, which saves intermediate results that can be plotted in real-time by the Tensorboard visualization tool.

Evaluation

During training usually not only the training loss and metrics need to be monitored, but also the testing loss and metrics, which give a performance indication of the network on never-before-seen data. By monitoring this the underfitting and overfitting of the network can be recognized. The training data or data generator can simply be passed as an argument to the `.fit()` and `.fit_generator()` functions.

If the evaluation of the network needs to be performed after the training has completed, the `evaluate()` and `evaluate_generator()` functions can be used.

Similarly if the predicted output of a certain set of data needs to be calculated, the `predict()` function can be used.

5 Commonly encountered problems

This section will briefly discuss some of the problems that you might encounter when implementing your own machine learning network.

Tensorflow vs. Tensorflow GPU

When installing Tensorflow you will likely also come across Tensorflow GPU. Both libraries will do the exact same job, however, Tensorflow GPU will perform all calculations on your GPU instead of your CPU. This usually has the benefit that a GPU will show significant improvements in training speed and that the CPU remains free for other tasks. However, the GPU version requires you to install several software programs in order to get everything working. Usually in terms of performance Tensorflow GPU is the best way to go, however, it requires some knowledge in order to get everything nicely installed. When unsure whether you can install the GPU version in your first encounter with machine learning, just install the normal Tensorflow version first and upgrade later on if you feel confident enough.

Keras vs. Tensorflow.Keras

When loading all necessary functions from the Keras library in order to build and train models, the Keras library needs to be imported. This library is a library on its own, but it is also a part of the entire Tensorflow library. Because of the heavy dependence of the Keras library on the Tensorflow library it is usually safer to import Keras using `from Tensorflow import Keras`. Not doing this could cause some very abstract errors which you would like to prevent.

Layers vs. functional API

The Keras library has several layers whose name is also used similarly in the functional API. An example of this is the add layer. The `Add()` function is a layer and the `add()` function is a function. The nuance in capital letter requires a slightly different syntax when adding the layer to the network. Listing 7 shows an example of the different syntaxes used.

```
# create additive layer 1
add_layer_1 = Add()([layer_before_1, layer_before_2])

# create additive layer 2
add_layer_2 = add([layer_before_1, layer_before_2])
```

Listing 7: Difference in syntax for the Add and add() functions.