Getting Started with DNN

Before we move to a more sophisticated ex. we will get familiar with Tensorflow and Keras. We will take a look at a first concrete example of a neural network, which makes use of the Python library Keras to learn to classify hand-written digits. The problem we are trying to solve here is to classify grayscale images of handwritten digits (28 pixels by 28 pixels), into their 10 categories (0 to 9). The dataset we will use is the MNIST dataset. It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s. You can think of “solving” MNIST as the “Hello World” of deep learning – it's what you do to verify that your algorithms are working as expected.

Firstly, we will import Keras (based on the Python/Tensorflow version):

```
from tensorflow import keras
...
import keras
```

If you are using the newest version of Tensorflow (keras is already included) please add additional functionality using:

```
from tensorflow import keras
from tensorflow.keras import XXX
```

The MNIST dataset comes pre-loaded in Keras, in the form of a set of four Numpy arrays:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Task 1 Please check the number of training and testing example.
Task 2 How many labels do we have?
Task 3 Check the image size and plot few examples

DNN workflow

- Build the neural network architecture.
- Train our neural network with the training data, train_images and train_labels. The network will then learn to associate images and labels.
- The network will produce predictions for test_images, and we will verify if these predictions match the labels from test_labels.

DNN architecture

- Our network will consist of a sequence of two Dense layers, which are densely-connected (also called “fully-connected”) neural layers.
- The second (and last) layer is a 10-way “softmax” layer, which means it will return an array of 10 probability scores (summing to 1). Each score will be the probability that the current digit image belongs to one of our 10 digit classes.

DNN training
To make our network ready for training, we need to pick three more things, as part of the "compilation" step:

- A **loss function**: This is how the network will be able to measure how good a job it is doing on its training data, and thus how it will be able to steer itself in the right direction.
- An **optimizer**: this is the mechanism through which the network will update itself based on the data it sees and its loss function.
- **Metrics**: to monitor during training and testing. Here we will only care about accuracy (the fraction of the images that were correctly classified).

Today, during our 3 examples we will use only the `Sequential class`. During our next meetings, I will introduce the functional API where we will be able to manipulate the data tensors that the model processes and apply layers to this tensors as if they were functions.

Network architecture

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Network Training

Once the model architecture is defined, the learning process can be configured in the compilation step. We specify the optimizer, loss function, and metrics.

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Data preparation
Before training, we will preprocess our data by reshaping it into the shape that the network expects, and scaling it so that all values are in the [0, 1] interval. Our training images are being stored in an array of shape (60000, 28, 28) of type uint8 with values in the [0, 255] interval. Please transform it into a float32 array of shape (60000, 28 * 28) with values between 0 and 1.
Necessary functions: `reshape` and `astype`.
Perform it for both train and test examples.

```
#TODO: reshape and transform the data
```

Because of we use `categorical_crossentropy` loss function we need to convert data format:

```
from keras.utils import to_categorical
```

```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Fit the model
To train the network we call the `fit` method of the network with parameters `epochs` and `batch_size`. Set `epochs` to 5, and `batch_size` to 128.

```
#TODO: fit the model
```

Network evaluation

Two quantities are being displayed during training: the “loss” of the network over the training data, and the accuracy of the network over the training data. We quickly reach an accuracy of 0.989 (i.e. 98.9%) on the training data. Now let's check that our model performs well on the test set too:

```
test_loss, test_acc = network.evaluate(test_images, test_labels)

print('test_acc:', test_acc)
```

Our test set accuracy turns out to be 97.8% – that's quite a bit lower than the training set accuracy. This gap between training accuracy and test accuracy is an example of “overfitting”, the fact that machine learning models tend to perform worse on new data than on their training data.

From:
https://home.agh.edu.pl/~mdig/dokuwiki/ - **MVG Group**

Permanent link:
https://home.agh.edu.pl/~mdig/dokuwiki/doku.php?id=teaching:data_science:dnn:lab1a

Last update: **2023/10/02 09:26**