TP: Fundamentals of Deep learning with TensorFlow

TAs:

Marine Picot: *marine.picot@centralesupelec.fr*
Eduardo Dadalto: *eduardo.dadalto-camara-gomes@centralesupelec.fr*

April 2, 2021

Objectives of the Practical Class

Nowadays, Artificial Intelligence drives scientific and economic growth worldwide. This is primarily due to advances in Machine Learning (ML), notably in Deep Neural Networks (DNNs), which are essentially massive 'learning by experience/examples' systems. Their applications span and revolutionize almost every human activity. We use it every day–Siri, face or fingerprint recognition, translation, voice recognition, automatic subtitles, among others–and it tends to be used in more and more applications: early-stage cancer detection, management of electrical grids, chatbots, etc.

During this practical class, we are going to create, train and evaluate widely common and used deep learning structures applied to a few different fields:

- A simple Multi-Layer Perceptron used for regression.
- A fully connected AutoEncoder to learn a representation of images.
- A Convolutional Neural Network (CNN) used to classify images.
- A natural language classifier along with Word Embedding used to classify language.

To perform the simulations, you can either used Keras/TensorFlow directly on your computers (local) or create a notebook using Google Colab.

TensorFlow is an end-to-end open-source platform for machine learning developed by Google. It facilitates building, training, inference and evaluation of neural networks.

Keras is a deep learning API running on top of TensorFlow. It was developed with a focus to enable fast experimentation.

The main idea is that a deep learning model is usually a directed acyclic graph (DAG) of layers. So the functional API is a way to build graphs of layers.

It is extremely useful to fasten classical experimentation but lacks flexibility.

Installation of TensorFlow and related packages

To install TensorFlow on your laptop, you need to create virtual environments.

You can do the following steps to install TensorFlow on your laptop (local installation) by creating virtual environments :

- Install Anaconda `https://docs.anaconda.com/anaconda/install/` and follow the instructions according to your operating system.
- Create your virtual environment. First, go to Environment (on the left side of the screen) and press "+". Then, give your environment a name - Tensorflow, for example - and choose which programming language you want to use - here, we will use Python.
- Launch your virtual environment by pressing the "Play" button that appears next to the name of the virtual environment. Select the "Open on Terminal" option. A terminal is then opened.
- On this terminal, type "conda install tensorflow" to install TensorFlow on your computer.
- If you need more python packages to run your code (matplotlib to visualize plots for examples), you can install them on your virtual environment using the command "conda install *name-of-the-package*" on the previously opened terminal.

Once this installation is performed, you can use different interpreters to use run your codes. You can use Jupyter, Spyder, PyCharm, installed with Anaconda, or Visual Studio Code if you are more comfortable with it. You have to specify the virtual environment that you want your code to run on in any case.

Fundamentals of Google Colab

Google Colab is built on top of Jupyter notebooks and gives the user an online IDE for editing and running python code. It is especially useful for light and moderate machine learning work because it gives you access to a GPU for free, and it has most of the necessary packages already pre-installed.

Connection You can connect to Google Colab via your google account by going to `https://colab.research.google.com/`. Click on "New Notebook" and start working! Once you are in the new notebook, you need to choose what hardware accelerator (CPU, GPU, TPU) you want to use. So click on Edit → Notebook Settings and select a GPU when you need one. You will see a "Connect" button on top and tada... you have a free GPU now!

For more information on Google Colab, check out the document [1]

[1] `https://docs.google.com/document/d/1Kw20oFoEi4bD_QLpng5gF674gcRXMT5WB3GuMSZILZ4/edit`

Warning: the runtime limit is 12 hours. After this time, the sessions are automatically disconnected.

Keras Fundamentals to Create a Neural Network

The first step to use Keras, as for any code on Python, is to import all the required package to run your code.

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
```

Load your data

For the most commonly used datasets such that MNIST or CIFAR, Keras has loaders.

```
(x_train, y_train), (x_test, y_test) =
  keras.datasets.name_of_the_dataset.load_data()
```

You will have to load your data and transform them in NumPy with the right shape for other datasets.

Generate a model

Then, you will have to create your models. To do so, you have two possibilities. Either use keras.Sequential or create your model yourself.

```
model = keras.Sequential([
          keras.layers.Flatten(input_shape=(28, 28)),
          keras.layers.Dense(128, activation='relu'),
          keras.layers.Dense(10)
])
```

```
input = keras.Input(shape=(28, 28))
x = keras.layers.Flatten(input_shape=(28, 28))(input)
x = keras.layers.Dense(128, activation='relu')(x)
output = keras.layers.Dense(10)(x)
model = keras.Model(inputs=input, outputs=output)
```

will both create the same model.

Train, Evaluate and Infer

After the previous step is completed, you have to specify the loss, optimizer, and metric you want to use for your training. To do so, you have to use the compile function:

```
model.compile(loss=keras.losses.name_of_the_loss,
              optimizer=keras.optimizers.optimizer_name(),
              metrics=[name_of_the_metric]
)
```

Then, you need to train your model using fit:

```
model.fit(x_train, y_train, batch_size=64, epochs=2)
```

Finally, to determine the performances of your training scheme, you can use the function evaluate:

```
model.evaluate(x_test, y_test)
```

Numerical Experiments to be Performed

Regression problem using a multilayer perceptron (MLP)

Refresher. MLPs can solve problems stochastically, allowing for approximate solutions to highly complex problems such as regression from multiple features. Cybenko's theorem shows that they are universal function approximators that can be used to construct any mathematical model via regression analysis. For example, a simple linear regression model may propose:

$$f(\mathbf{x}_i; b, \mathbf{w}) = b + \mathbf{w}^T\mathbf{x}_i,$$

where $\mathbf{x}_i$ is a feature row, and b and $\mathbf{w}$ are the learnable parameters named as biases and weights, respectively. This equation models the regression part of a perceptron and is present on the interior of the neurons in an MLP.

A non-linear regressor takes an input a linear combination of parameters and passes through a non-linear function. These functions are called "activation functions". A widespread one is the ReLU function defined below:

$$f(\mathbf{x}) = \max(0, \mathbf{x}).$$

By connecting neuron outputs in a computational graph, you build a neural network. The MLP is a special case of feed-forward neural networks, where its computation graph is composed of one or multiple layers of perceptron-inspired neurons stacked together *sequentially.*

Exercise 1. Build a median house value for households within a block (measured in US Dollars) regressor using an MLP with 1 dense hidden layer with the ReLU activation function combined with a dropout layer. Optimize the problem with the Adam optimizer and a mean squared error (MSE) loss function.

How can you check if your model isn't over-fitting the data?

The dataset can be downloaded at [2].

To get started faster:

```
import pandas as pd
import tensorflow as tf
from tensorflow import keras
# Load the data
df = pd.read_csv("housing.csv")
# Drop null values
df.dropna(inplace=True)
# Drop categorical column as MLP cannot treat it as is
df = df.drop("ocean_proximity", axis=1)
X = df.drop("median_house_value", axis=1)
y = df["median_house_value"]
# Train test split
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test =
                        train_test_split(X, y, test_size=0.2)
```

Tip: for better performance, consider centering and standardizing the features.

Fully connected AutoEncoder

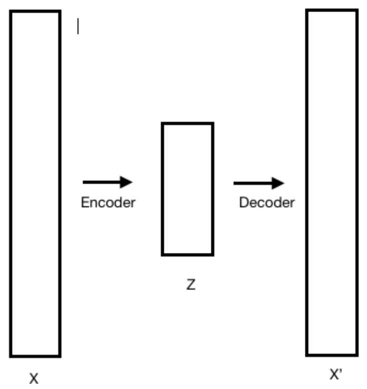
Refresher: An AutoEncoder is a deep neural network that encodes an input. The goal is to find meaningful representations of the input to reduce its dimensions while reconstructing the input from the representations. It is a non-supervised method.

If we consider $\mathbf{x} \in \mathcal{X}$ the input of the autoencoder, $z \in \mathcal{Z}$ the encoded representation and $x\prime \in \mathcal{X}$ the output of the autoencoder, the autoencoder can be seen as:

$$\mathbf{z} = \phi(\mathbf{x});\ x' = \psi(\mathbf{z}) = \psi \circ \phi(\mathbf{x}). \quad (1)$$

Exercise 2. Build a fully connected Autoencoder to encode the images from MNIST with representations of dimension 32, minimizing the reconstruction loss of your choosing. Explain your choice (by comparing it to other losses).

[2] `https://www.kaggle.com/camnugent/california-housing-prices`



Convolutional Neural Networks

Refresher A classical ConvNet is built through a sequence of layers, as shown in the list below. Each architecture combines these layers to obtain the best performance for a certain class of problems:

1. Input layer (Input).
2. Convolutional layer (most computationally demanding) (Conv2D).
3. Pooling layer (MaxPooling2D).
4. Dense layer (Dense).

Yann LeCun developed the first successful applications of Convolutional Networks in the '90s. LeNet-5 represents one of the simplest architectures, composed of 2 convolutional and 3 fully connected layers. This architecture has become the traditional "template": the stacking of convolutions and pooling layers and the termination of the network with one or more fully-connected layers. Usually, we want to control the output volume of the convolutional layer. For this, three hyperparameters are essential.

1. Depth (K): corresponds to the number of filters we would like to use.
2. Stride (S): the amount the filter slides. When the stride is 1 then we move the filters one pixel at a time.
3. Zero-padding (P): The amount of zeros that will be added around the border of the input volume.

Figure 1 illustrate well these parameters. Also, this calculator [3] can help you to design the convolutional layers.

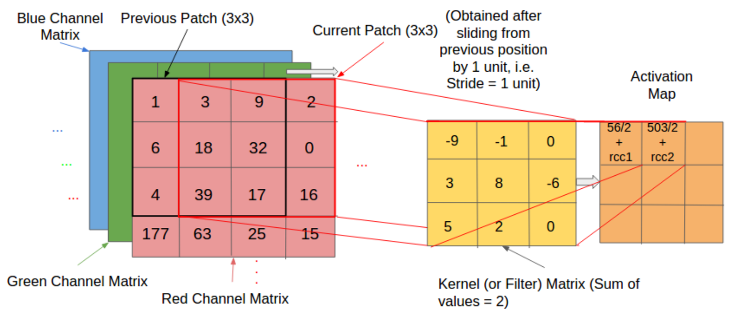


Figure 1: Illustration of a convolutional kernel operation in a neural network. Source: `https://blog.xrds.acm.org/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/`.

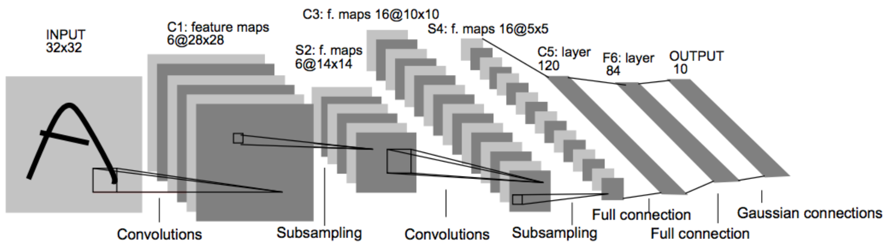


Figure 2: Illustration of a LeNet-5 ConvNet. Yan LeCun, MIT License.

Exercise 3. Code a LeNet-5 to classify the FashionMNIST dataset.
The expected architecture is illustrated in figure 2.
To get started faster:

```
import tensorflow as tf
from tensorflow import keras
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
import numpy as np
dataset, metadata = tfds.load("fashion_mnist",
                                as_supervised=True,
                                with_info=True)
train_dataset, test_dataset = dataset["train"], dataset["test"]
paddings = tf.constant([[2, 2], [2, 2], [0, 0]])
def normalize(images, labels):
  images = tf.pad(images, paddings)
  images = tf.cast(images, tf.float32)
```

[3] `https://madebyollin.github.io/convnet-calculator/`

```
  images /= 255
  return images, labels

# The map function applies the normalize function to each element
train_dataset = train_dataset.map(normalize)
test_dataset = test_dataset.map(normalize)
# Plot some image examples
plt.figure(figsize=(10,10))
i = 0
for (image, label) in test_dataset.take(25):
    image = image.numpy().reshape((32,32))
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(image, cmap=plt.cm.binary)
    i += 1
plt.show()
```

Natural Language Processing using Word Embedding

Refresher: In this exercise, we are going to determine whether a comment is positive or negative. To do so, we have to find a way for a classifier to “understand” a sentence. First of all, neural networks require NumPy vectors to work, so we need to find a way to “encode” the words as numbers. We can think of 3 ways. The first one is to use a one-hot encoder. Each word will be represented as a vector of n components full of 0 with a 1 at one spot. The place of the 1 will differ for every word. The second one is quite close since it consists of assigning a number to each word. These two methods are efficient, but we do not have any way to encode the fact the prior knowledge that 2 words can have a similar meaning. So the best solution is the 3rd method, called word embedding.

Word embedding gives us a way to use an efficient, dense representation in which similar words have a similar encoding. An embedding is a dense vector of floating-point values that are trainable parameters (weights learned by the model during training, in the same way a model learns weights for a dense layer). It is common to see word embeddings that are 8-dimensional (for small datasets), up to 1024-dimensions when working with large datasets. A higher dimensional embedding can capture fine-grained relationships between words but takes more data to learn.

Exercise 4. Code a classifier that determines whether a comment is positive (label 1) or negative (label 0) according to Figure 2. Next, you will find a few lines to help you.

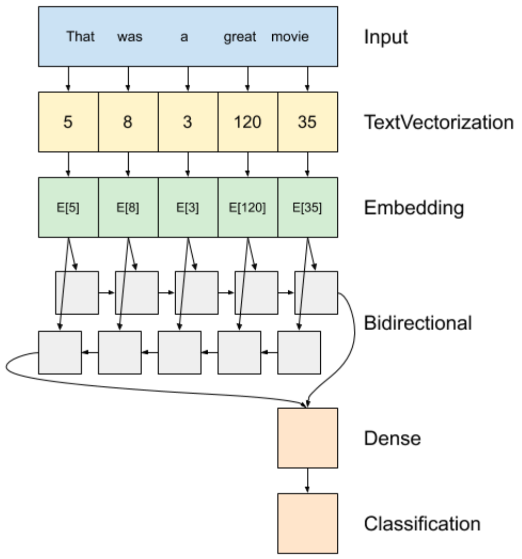


Figure 3: NLP classifier
.

Download the IMDb Dataset

```
import io
import os
import re
import shutil
import string
import tensorflow as tf
from tensorflow import keras
from datetime import datetime

url = "https://ai.stanford.edu/~amaas/data/
        sentiment/aclImdb_v1.tar.gz"
dataset = tf.keras.utils.get_file("aclImdb_v1.tar.gz", url,
                                  untar=True, cache_dir='.',
                                  cache_subdir='')
dataset_dir = os.path.join(os.path.dirname(dataset), 'aclImdb')
train_dir = os.path.join(dataset_dir, 'train')
remove_dir = os.path.join(train_dir, 'unsup')
shutil.rmtree(remove_dir)
seed = 123
train_ds = tf.keras.preprocessing.text_dataset_from_directory(
           'aclImdb/train', batch_size=batch_size,
          validation_split=0.2, subset='training', seed=seed)
val_ds = tf.keras.preprocessing.text_dataset_from_directory(
         'aclImdb/train', batch_size=batch_size,
         validation_split=0.2, subset='validation', seed=seed)

AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

.cache() keeps data in memory after it's loaded off disk. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache, which is more efficient to read than many small files.

.prefetch() overlaps data preprocessing and model execution while training.

Warning: you have to set a value for the batch size.

Preprocessing

```
# Create a custom standardization function to strip HTML break
# tags '<br />'.
def custom_standardization(input_data):
    lowercase = tf.strings.lower(input_data)
    stripped_html = tf.strings.regex_replace(lowercase,
                                             '<br />', ' ')
    return tf.strings.regex_replace(stripped_html,
           '[%s]' % re.escape(string.punctuation), ' ')

# Vocabulary size and number of words in a sequence.
vocab_size = 10000
sequence_length = 100

# Use the text vectorization layer to normalize, split, and map
# strings to integers. Note that the layer uses the custom
# standardization defined above.

# Set maximum_sequence length as all samples are not of the same
# length.

vectorize_layer = TextVectorization(
                     standardize=custom_standardization,
                     max_tokens=vocab_size,
                     output_mode='int',
                     output_sequence_length=sequence_length)

# Make a text-only dataset (no labels) and call adapt to build
# the vocabulary.
text_ds = train_ds.map(lambda x, y: x)
vectorize_layer.adapt(text_ds)
```

Embedding

```
 keras.layers.Embedding(vocab_size, embedding_dim,
                        name="embedding")
```

Warning: you have to set a value for the embedding dimension.

Classifier

Use 2 dense layers.

Optional : Visualize the embedding

```
weights = model.get_layer('embedding').get_weights()[0]
vocab = vectorize_layer.get_vocabulary()
out_v = io.open('vectors.tsv', 'w', encoding='utf-8')
out_m = io.open('metadata.tsv', 'w', encoding='utf-8')

for index, word in enumerate(vocab):
  if  index == 0: continue $\#$ skip 0, it's padding.
  vec = weights[index]
  out_v.write('\t'.join([str(x) for x in vec]) + "\n")
  out_m.write(word + "\n")\\
out_v.close()
out_m.close()
```

Then go to `http://projector.tensorflow.org/`, click on Load Data, and upload the two files you just created.

References

1. `https://keras.io/`
2. `https://keras.io/guides/functional_api/`
3. `https://www.tensorflow.org/tutorials/`