Introduction to Machine Learning for Physicists

Problem Set 1
Lecturer: Fabian Ruehle
Github: https://github.com/ruehlef/ML_Oxford_Hilary2020/Problem_set_1/

1.0 Setup

Here we briefly explain how to set up a Python environment. If you have worked with Python before, just skip it.

Installing Python

First, you need a Python interpreter. There have been important changes between version 2 and version 3 of Python. Version 2 has been discontinued as of 2020, so I recommend using Python 3 (the subversion does not matter too much, I use 3.7 at the moment). Chances are there is already a python interpreter installed on your system. Try running

```
python3 --version
```

If the output looks something like

```
Python 3.7.3
```

your system has a Python environment and you can use that. Otherwise, install a Python 3 environment for your system, e.g. Miniconda[1].

Creating a virtual environment

Next, instead of directly using your system's Python installation, I strongly advice to create a "local copy" (known as a virtual environment) of your Python installation. This way, you can install and update Python packages for a specific project without changing the packages system-wide. Since libraries can change considerably and are not always necessarily backwards compatible, it is a good idea to save the virtual environment with each project, such that the code can be run with the Python and package versions for which it was developed, even if future projects will be using newer versions. For example, Tensorflow version 2 has important changes as compared to version 1. Similarly, PyTorch also made some crucial changes between their versions.
Creating a virtual environment is very easy. Executing

```
python3 -m venv ~/my_venv
```

will create a virtual environment `my_venv` in your home folder. That's all there is to it. Now you can use it via

```
source ~/my_env/bin/activate
```

Now, when you run python or the package manager pip to install new packages, it will use the virtual environment instead of your system's interpreter. You can verify this be executing

[1]https://docs.conda.io/en/latest/miniconda.html

```
which python
```

which should show the path to your virtual environment now. To deactivate the virtual environment simply execute `deactivate`, or close the terminal window.
If you installed Conda, the syntax is slightly different; see their documentation on how to set up a virtual environment.

Installing packages

Next, you should install the packages we will be using. Run[2]

```
pip install numpy
pip install scipy
pip install matplotlib
pip install torch
pip install tensorflow
pip install keras
pip install chainerrl
pip install scikit-learn
pip install future
```

The packages `numpy` and `scipy` provide many math functions. The package `matplotlib` is used for plotting the results. The packages `torch` and `tensorflow` are two of the currently most popular machine learning packages. The package `keras` provides some high-level functionalities for tensorflow, which makes coding in tensorflow easier. The package `chainerrl`

Running Python scripts

Python scripts are text files (usually with a ".py" extension). You can create the file in your favorite text editor (call it "test.py"), add some code (e.g. `print("Hello World")`) and run it via

```
python test.py
```

If you are planning to use Python more extensively, it might be worthwhile looking into using a Python Integrated Development Environment. There are many out there, see e.g. Wikipedia[3] for a list of available ones. I personally use PyCharm. These provide many features (they support setting up virtual environments, do syntax highlighting, code navigation, ...), most importantly, they make debugging easy (they provide a debugger with break points, variable watches, step-wise execution, stack traces, ...).

[2]If you are not using a virtual environment, at least use the "–user" option for `pip`.
[3]https://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments#Python

1.1 A simple classifier

We will start with a very simple classification task that serves to illustrate (by plotting) and understand the concepts behind NNs. I provide template files

- `exercise1_keras_template.py`
- `exercise1_torch_template.py`

with an outline of the code for these two libraries. You can use these and insert the code for the different parts of the exercise. Or you implement it from scratch. If you are completely new to Python, you might also just want to download the solutions

- `exercise1_keras_solution.py`
- `exercise1_torch_solution.py`

and play with the options and parameters a bit.
Given two particles with electric charge q_1 and q_2, we want to classify whether the force between them is attractive or not (effectively, this corresponds to an `XOR` operation). This problem is a binary one-vs-all classification task. The input to the NN will be $x^0 = (q_1, q_2)$, and the output can be either two nodes (one for the class "attractive" and one for the class "not attractive") with a `softmax` activation function, or a single node with a `logistic sigmoid` (or even `tanh`) activation function. In the latter case, the input would be mapped to 1 if the force is attractive and to 0 (or -1 in the case of `tanh`) else.
Feel free to use whatever library you want. I will provide template files for PyTorch and Keras. You should be able to complete the exercises by using the code you can find in these files. But it is also a good idea to get acquainted with the documentation of these libraries

- Keras: https://keras.io
- PyTorch: https://pytorch.org/docs/stable/torch.html

(a) First we need a training set, i.e. a labeled set of input-output-pairs. Implement a function in Python that computes the function $f : \mathbb{R}^2 \to \mathbb{Z}_2$,

$$f(q_1, q_2) = \begin{cases} 1 & \text{if } \operatorname{sign}(q_1) \neq \operatorname{sign}(q_2) \\ 0 & \text{else} \end{cases} \tag{1}$$

Use this function to create a training set for all pairs $-5 \leq q_i \leq 5$, $i = 1, 2$.

(b) Implement a neural network (call it `my_nn`) with two hidden layers with 10 nodes each and with `tanh` activation function. The final layer should have a `sigmoid` activation function. How many parameters does this NN have?

(c) Train the NN. We will discuss training in the next lecture, so just apply the functionality provided by your ML library for now. We will use `stochastic gradient descent` with `batch size` 1, `learning rate` 0.0001 and a `binary cross entropy` loss. Repeat this for 100 epochs (i.e. let the NN work through the entire training set 100 times). In Keras, you would do this in the following way

```
import numpy as np
from keras import optimizers

learning_rate = 0.01
batch_size = 1
loss = 'binary_crossentropy'
epochs = 100

sgd = optimizers.sgd(lr=learning_rate)
my_nn.compile(loss=loss, optimizer=sgd)
y_hat = my_nn.fit(np.array(x),
                  np.array(y),
                  epochs=epochs,
                  batch_size=batch_size,
                  verbose=1)
```

In PyTorch, this is done with

```
import numpy as np
import torch

learning_rate = 0.01
epochs = 100

sgd = torch.optim.SGD(my_nn.parameters(), lr=learning_rate)
criterion = torch.nn.BCELoss()

for epoch in range(epochs):
    for i in range(len(x)):
        sgd.zero_grad()
        y_hat = my_nn(x[i])
        loss = criterion(y_hat, y[i])
        loss.backward()
        sgd.step()
```

(d) DISCLAIMER: As we will discuss in the next section, one should never evaluate the performance of a NN with the same data that was used for training. Since we will only discuss this in the next lecture, we will make an exception and use the training set here.

Make a 3D plot of $(q_1, q_2, \hat{y})$ to visualize the output of the NN. Introduce a threshold of 0.5 for the decision (i.e. round the output of the NN to integer values) and plot again the output to visualize classification for this decision boundary. Study how the decision boundary forms by training the NN also for only 1, 5 or 10 episodes.

(e) Adapt the NN to allow for three classes, "attractive", "neutral", and "repulsive" (where neutral means that one of the particles is uncharged). Change the function from part (a) to assign class labels 0, 1, 2 to the three cases. Then, change the NN architecture of (b) to allow for three classes (now you will need three output nodes and a `softmax` layer as the output layer).

1.2 A simple regressor

In this exercise, we will train a NN for regression. Again, template files to fill in some code, as well as solutions, are available.
As input features for the regressor, we will use CMS data from $Z \to e\bar{e}$ decays. CERN provides quite a lot of data to play around with at http://opendata.web.cern.ch. We will be using the data set Zee.csv, which is available at http://opendata.web.cern.ch/record/545.
We want to train a NN that can predict the invariant mass of the Z boson from the transverse momenta $p_{T,1}, p_{T,2}$, the pseudo-rapidities η_1, η_2, and the azimuthal angles ϕ_1, ϕ_2 of the electron and the positron. In this case we know of course the exact formula, and this serves only to illustrate the concepts.
In fact, the invariant mass is not included in the data, so we need to compute it first such that we have labels for our training data. In terms of the quantities above, we have (in the relativistic approximation)

$$M_Z^2 = 2p_{T,1}p_{T,2}(\cosh(\eta_1 - \eta_2) - \cos(\phi_1 - \phi_2))\,. \tag{2}$$

(a) Implement a routine in Python that computes M_Z.

(b) The simplest approach is to just use $p_{T,i}$, η_i, and ϕ_i as input to compute M_Z. Set up a NN with 6 input nodes, two hidden layers with 100 nodes each, and a single output node. Which activation function should/shouldn't you choose for the output layer? And for the hidden layers?

(c) The code for training the regressor is essentially the same as for the classifier. As a loss function, we use the mean squared error between the labels and the predictions of the NN. In Keras, this is done simply by replacing the lines that specifies to use the BCE Loss in the classifier by

```
loss = 'mean_squared_error'
my_nn.compile(loss=loss, optimizer=sgd, metrics=['mse'])
```

In PyTorch, this is done by using

```
criterion = torch.nn.MSELoss()
```

Look at the error and the prediction of the NN (again, we should have performed a train:test split). What did the NN actually learn?

(d) NNs are more stable if the input and output are not too large. In particular, the input is often normalized to have zero mean and unit variance (with respect to the data available in the training set). Now, in our case, Δ_η and $\Delta_p hi$ already have approximately zero mean (why?), and their numerical values are not too large (why?). However, we might want to express $p_{T,i}$ and M_Z in units of 100GeV. Use this as input, train the NN again and see what happens.

(e) From the underlying Physics, we know that the result can only depend on $\Delta\eta = \eta_1 - \eta_2$ and $\Delta\phi = \phi_1 - \phi_2$. Moreover, it has to be symmetric in $p_{T,1}$ and $p_{T,2}$ and can thus, based on the mass dimensions, only depend on $p_{T,1} \cdot p_{T,2}$. So we can perform feature engineering and use these as input to the NN. Set it up such that it now only takes 3 inputs $p_{T,1} \cdot p_{T,2},\ \Delta_\eta,\ \Delta_\phi$. Which step was more important here, feature engineering or normalizing the input?