Deep Learning in NLP

Homework P2

Solution to be sent (pdf, zip) to `waszczuk@phil.hhu.de` and `cwurm@phil.hhu.de` by 3.11.2019.

Preparation

First of all, download the archive with the code available on the course's website. Unpack it and open in VSCode (`File` → `Open Folder`, at least with English language settings). The folder contains two Python files:

- `moons.py`: module for creating the dataset we will be playing with
- `u2_lin.py`: a preliminary solution to the exercise

To have a look at the dataset, enter the IPython session in VSCode (`CTRL+‘` with English settings, `CTRL+ö` with German settings) and type the following:

```
run u2_lin
moons.plot_moons(X, Y) # X and Y are defined in u2_lin
```

If everything works as intended, you should get something similar to Fig. 1.

The **goal** is to design a PyTorch model which will be able to classify points $\mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$ according to whether they belong to the first moon (orange) or the second one (blue). To this end, we represent the orange (upper) moon with 0 and the blue (lower) moon with 1.

The **preliminary solution** to the exercise is based on a linear model with parameters $\mathbf{w} = (w_1, w_2) \in \mathbb{R}^2$ and (bias) $b \in \mathbb{R}$. Namely, given the input point $\mathbf{x} \in \mathbb{R}^2$, we „calculate the color" using:

$$y = w_1 x_1 + w_2 x_2 + b \tag{1}$$

Then we decide on the membership of the point based on the resulting value: the point belongs to the upper, orange moon if $y < 0.5$, and to the lower, blue moon otherwise.

Even though the model is linear, the implementation is built on top of the PyTorch library and follows the general principles of training in deep learning frameworks. Try to go through the code to understand what is going on. To see how well the model performs on the dataset, you can run the following in IPython:

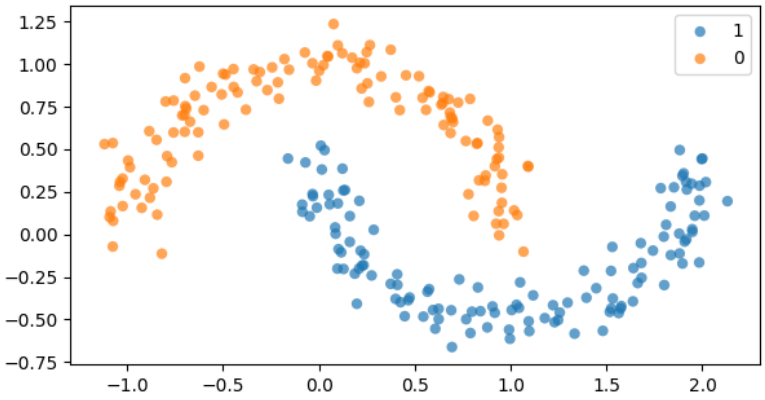

Figure 1: The moons dataset

```
moons.plot_moons(X, Y, model=apply_raw)
```

The parameters of the model are initialized randomly, so the fit won't be very good at first. To adapt the model to the dataset, run:

```
train()
```

After that, the fit should be better, but still not perfect due to deficiencies of the model (the dataset is not *linearly separable*, and therefore beyond what we a linear model can handle, at least without pre-processing).

Exercise 1

Note: the plan is to do most of this exercise (if not all) together in the class.

Implement a *feed-forward network* (FFN) and put it in place of the linear model. Given the size of the hidden layer h and the following parameters:

- $\mathbf{M}^1 \in \mathbb{R}^{h \times 2}$: matrix corresponding to the first linear transformation
- $\mathbf{b}^1 \in \mathbb{R}^h$: the first bias vector
- $\mathbf{M}^2 \in \mathbb{R}^{1 \times h}$: matrix corresponding to the second linear transformation
- $\mathbf{b}^2 \in \mathbb{R}$: the second bias vector

FFN (with one hidden layer) can be defined using the following pair of equations:

$$\mathbf{h} = \sigma(\mathbf{M}^1\mathbf{x} + \mathbf{b}^1) \tag{2}$$

$$y = \mathbf{M}^2\mathbf{h} + \mathbf{b}^2 \tag{3}$$

where $\mathbf{x} \in \mathbb{R}^2$ is the input vector, and σ is an element-wise sigmoid function.

The main missing piece is marked with `TODO`. You will have to identify the other missing parts yourself (they are mostly boilerplate, though, which we will try to avoid in future sessions).

Note: FFN is a so-called *universal approximator*, it can approximate any continuous function on a compact input domain to arbitrary accuracy (provided that h is large enough). This is something you will learn during the theoretical sessions, though.

Exercise 2

You can implement your own FFN in PyTorch, but it is possible to go even lower-level. The task of Ex. 2 is to implement your own matrix-vector product function `mv`. That is to say, given a matrix matrix $\mathbf{M} \in \mathbb{R}^{n \times m}$ and a vector $\mathbf{v} \in \mathbb{R}^m$, the result of $\texttt{mv}(\mathbf{M}, \mathbf{v})$ should be a vector $\mathbf{w} \in \mathbb{R}^n$ such that:

$$\mathbf{w}_{[k]} = \sum_{i=1}^{m} \mathbf{M}_{[k,i]} \mathbf{v}_{[i]} \tag{4}$$

where $\mathbf{v}_{[i]}$ is the i-th element of vector $\mathbf{v}$, and $\mathbf{M}_{[k,i]}$ is the element in the k-th row and i-th column of $\mathbf{M}$.

Use the code template in `ex2.py` to solve the exercise. This can be helpful. The code contains some additional checks to verify that the function you implement is correct.

Note: the goal of this exercise is to check your ability to manipulate tensors, and to show that it is possible to re-implement (some) higher-level tensor operations using more primitive operations. You can even replace `torch.mv` with your own `mv` function in the solution of Ex. 1, but this is not necessarily the best idea (why?).