INM705 Deep Learning for Image Analysis

Lab 1 Instructions

1. Installation

The main objective of this lab is to introduce the students to some basic and advanced concepts in Pytorch necessary for operations with Computer Vision objects. Due to substantial differences between Camber and students' home computers, I'm providing two different ways of installing the virtual environment. For the local devices, I recommend using pip with Python3.6, for the Camber I recommend the provided virtual environment.

Camber (remote server)

For the installation of miniconda environment follow these instructions, except that I recommend using the newer environment:

```
wget https://repo.anaconda.com/miniconda/Miniconda3-py39_4.10.3-Linux-x86_64.sh
```

Once you have done that, create the provided alex2022 virtual environment:

```
conda env create -f alex2022.yml
conda activate alex2022
```

After that, you will have to install cv2 separately:

```
conda install -c conda-forge opencv
```

It is not very straightforward to run Jupyter notebooks on remote servers (we will get back to it later), so you can use the usual Python scripts, provided in this lab.

Local machine

The following instructions are for Windows computers. If you run Linux, use the instructions from Camber section, except that you actually can use Jupyter. Unfortunately I don't have the access to Apple OS. On Windows computers, you will have to use the command line, type cmd in the Windows menu.

1. Install python3.6.x (any subversion should do). For example, you can download v3.6.5.from here (Windows executable installer), check the 'add to path' checkbox. Note the directory of the installation (use cd in cmd to navigate to directories).

2. Download pip installer

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
```

Install pip:

python get-pip.py.

3. Use the provided requirements.txt file to install all the necessary libraries (I tested it on Windows):

pip install -r requirements.txt

2. Pytorch

1. Basic model

I leave basic Pytorch commands, e.g. tensor creation, leaf variables, etc to you. Wrt leaf variables, just keep in mind that the models' parameters are always leaf variables. Instead, we will focus on the pytorch's neural network abilities.

Run the provided notebook:

jupyter notebook DLIALab1.ipynb.

In this lab you will learn to use pytorch dataset, dataloader and model interfaces that allow you to sample the data, train, validate and test the model. This knowledge is essential to the rest of the course.

First, we will create a random data tensor and run it through a simple pytorch model, *BasicModel*, that has 2 fully connected layers. There are two ways to create ANNs in Pytorch:

1. Inherit ***nn.Sequential***: ordered dictionary of layers (or list), the order is defined in the dictionary (or list),

2. Inherit ***nn.Module***, that allows for a more sophisticated models (see below).

For now, we use nn.Sequential to create the model.

Accessing the model's layers, features and parameters

There are many ways to do this, depending on what you are after. For example, you can access layers and parameters using either generator, ordered dictionary or numbers (see Lab1 comments).

2. Advanced Models and Datasets

The dataset is the credit fraud classification downloaded from Kaggle. It consists of 28 input variables, and one output: Class 0 for non-fraud and Class 1 for fraud. This is a binary classification problem. The dataset of about 285K observations is split into three subsets: Train (75%), Validation (15%) and Test (10%) stored in three different files. A simple code ***split_data.py*** is provided to create other random splits.

In this lab we introduce two important interfaces:

3. Dataset + Dataloader

The main objective of this interface is to output a single data point. This interface inherits ***Dataset*** class from ***torch.utils.data*** library with two important 'magic' functions: ***__len__*** (total number of data points) and ***__getitem__*** (extract a datapoint indexed by ***idx*** input). The datapoint consists of the data ***X*** and label ***y***. In this case data is a vector of the predefined length and the label is the next value. Dataloader interface constructs a batch of predetermined size (hyperparameter) of these datapoints, e.g. for **batch_size=512** the data matrix **X** will be **512x28**, target vector **y 512x1**.

4. Model

Here we write pytorch interface for the model(net). It inherits ***torch.nn.Module*** superclass, and, in addition to the class constructor, where various layers are defined using pytorch's neural network library (***torch.nn***), it has a ***forward*** method that applies the neural network methodology to the input ***X***. The model outputs a single logit value y at training stage and a class index (y<threshold:0, y>threshold:1) at test stage. In pytorch there are two ways to compute the loss for the binary problem:

1) Logit + binary cross-entropy loss (***y+torch.nn.BCEWithLogitsLoss*** or ***torch.nn.functional.binary_cross_entropy_with_logits***),

2) Sigmoid value of the logit + cross-entropy (***torch.sigmoid(y) + torch.nn.BCELoss*** or ***torch.nn.functional.binary_cross_entropy***)

The loss value will be identical. Note that for the multilabel problem (Softmax).

We introduce a ***DataLoader*** class, also from the ***torch.utils.data*** library that samples these datapoints (resp. for training and validation).

These two interfaces are used throughout the whole course.

It's always useful to know how large your model is. We loop through the model's **state_dict** (ordered dictionary containing tuples of the form (layer name, layer parameters)) to get the total number of parameters in each layer (**.numel()** command)

Next, we introduce ***SGD*** optimizer from ***torch.optim*** library. This is Stochastic Gradient Descent (SGD) solver that updates the weights in the model. The loss function is mean squared error, MSE, which is also taken from ***torch.nn*** library. We set the model's hyperparameters learning rate (***lr***) and regularization (***weight_decay***) to 1e-4 and 1e-3 resp.

Using method ***save_checkpoint*** we save the model's checkpoint: model's weights (state_dict) are stored as the value of ***model_weights*** key and the optimizer's state (derivatives for each weight) as the value of ***optimizer_state key***. To load the

pretrained weights into the model, use ***load_checkpoint*** method. Note that not just the architecture, but also the layers' names must fully match the keys of the ordered dictionary in ***stat_dict*** in order to load without an error.

3. <u>Exercises (suggestions):</u>

1. In the provided script, add the code to get the test split loss.
2. Augment the provided model with more layers or blocks. Does the performance always improve?
3. Train the model using different hyperparameters (e.g., batch size, learning rate, etc). Save and load the model's checkpoint, evaluate it on the previously saved test splits.