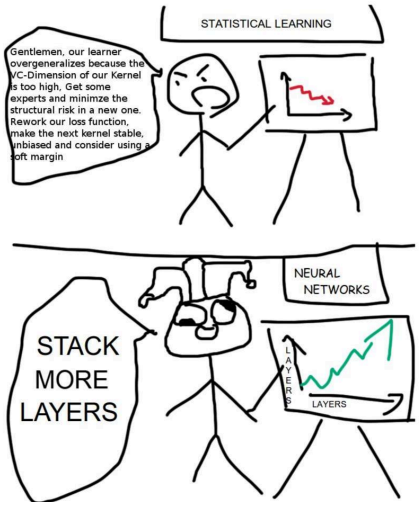
Introduction to Deep Learning 2022 Homework 1: Twitter Sentiment classification using Feed Forward Neural Networks

Due Date: 04.02.2022

Introduction

In the first homework assignment, you will implement a sentiment classifier for tweets using PyTorch. You will get a data set of tweets and some starter code. We ask you to implement a feed-forward neural network, and train it using PyTorch functions.



Getting Started

You should start by unpacking the archive `intro_to_dl_assignment_1.tgz`

```
$ tar -xzvf intro_to_dl_assignment_1.tgz
```

This results in a directory `intro_to_dl_assignment_1` with the following structure

```
--data/
----development.gold.txt
----development.input.txt
----test.gold.txt
----test.input.txt
----training.txt
--src/
----DL_hw1.py
----DL_hw1.ipynb
----data_semeval.py
----paths.py
```

The directory `data` contains the Twitter training, development and test sets. The skeleton code for the classifier, as well as code for data handling is located in `src`.

There are two skeleton code files: `DL_hw1.py` and `DL_hw1.ipynb`. The first one is plain Python code and the second one is a Juptyer notebook file. You can complete your assignment using either one. The rest of this document assumes that you are using `DL_hw1.py`.

You should start by copying `src/DL_hw1.py` to a new name `src/DL_hw_1_USERNAME.py`, where `USERNAME` is your university account name. This is the file that you will edit and finally upload to Moodle after you are done with the assignment.

Try to run the Python program. It requires PyTorch, NumPy and NLTK (for tokenization). You should see the output below.

```
$ python3 DL_hw1_username.py
[nltk_data] Downloading package punkt to /Users/username/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
epoch: 1, loss: 0.0000
epoch: 2, loss: 0.0000
...
epoch: 9, loss: 0.0000
epoch: 10, loss: 0.0000
TEST DATA: dec 21st 2012 will be know not as the end of the world but the
baby boom ! #2012shit, GOLD LABEL: neutral, GOLD CLASS 1, OUTPUT: -1
...
test accuracy: 0
```

The program will run ten epochs[1] of training algorithm and output accuracy on the

[1] An epoch is a complete run of over the entire training data.

test set. However, the implementation of the model is trivial (it does nothing) and the implementation of classification is also trivial (the classification simply returns neutral for all tweets). It is your task to actually implement the model, training and classification properly.

Twitter Data

We will be working with data sets from the 2014 SemEval shared task "Sentiment analysis in Twitter". If you are interested in background information or want to compare your own system to systems which participated in the competition, please have a look at:

S. Rosenthal, A. Ritter, P. Nakov and V. Stoyanov. *SemEval-2014 Task 9: Sentiment Analysis in Twitter*. SemEval. 2014

The started code will take care of reading and writing data for you. However, if you are interested in the data sets we are working with you can take a look at the `data` directory. Each data file consist of lines with three fields: a Twitter ID number, a sentiment class (positive, neutral or negative), and a tweet. Here are a few lines from `training.txt`

```
264183816548130816  positive  Gas by my house hit $3.39!!!! I'm going to Chapel Hill on Sat. :)
263939499426476032  positive  @jackseymour11 I may have an Android phone by the time I get back to school! :)
261965816906534912  neutral   @juice005 strange enough, I'm going to see Noel Gallagher in concert tomorrow night. ...
264160761117552640  negative  Just found a piece of candy that may have been injected with something. #Why
```

The files `development.input.txt` and `test.input.txt` do not show the correct sentiment label for the tweets. Instead, they have a dummy `unknwn` label in the second field. If you want to see the correct labels, you can look at `development.gold.txt` and `test.gold.txt`.

The Starter Code

Let's look at the main program in `DL_hw1.py`. It

1. reads in the Twitter sentiment data sets using `read_semeval_datasets()`,
2. generates bag-of-words representations for all tweets using `generate_bow_representations`,
3. initializes, a Feed-Forward Neural Network model, loss function, and optimizer, and
4. trains the model for 10 epochs on the training data and then uses the model to classify the test data.

The function `read_semeval_dataset` returns a Python `dict` object `data` which has five keys:

- `"training"` – the training set.

- `"development.input"` – the unlabeled development set.
- `"development.gold"` – the development set with gold standard labels.
- `"test.input"` – the unlabeled test set.
- `"test.gold"` – the test set with gold standard labels.

Each of these, for example `data["training"]`, is a list of tweets which are themselves Python `dict` objects, e.g.

```
{'ID': '264183816548130816',
'SENTIMENT': 'positive',
'BODY': ['Gas', 'by', 'my', 'house', 'hit', '$', '3.39', '!', '!',
         '!', '!', 'I', "'m", 'going', 'to', 'Chapel', 'Hill', 'on',
         'Sat', '.', ':)']}
```

where

- `tweet["ID"]` is the tweet ID.
- `tweet["SENTIMENT"]` is the the sentiment class (positive, negative, neutral).
- `tweet["BODY"]` the tokenized tweet.

The function `generate_bow_representations` transforms tweets into bag-of-word representations which are encoded as $1 \times |V|$ PyTorch tensors, where $|V|$ is the size of the vocabulary. You can access the tensor using `tweet['BOW']`.

1 Assignment: The `FFNN` Class

A feed-forward neural network model for tweet classification is parametrized by the weights and biases of its neurons. Eg., for a feed-forward network with two hidden layers (notation is modified from Goldberg, pg. 43):

$$\begin{aligned}
\text{FFNN}(x; W^1, b^1, W^2, b^2, W^3, b^3) &= y && (1)\\
h^1 &= g^1(xW^1 + b^1) && (2)\\
h^2 &= g^2(h^1W^2 + b^2) && (3)\\
y &= g^3(h^2W^3 + b^3) && (4)
\end{aligned}$$

where x is the BOW representations of the tweets, y is the output predicted by the network, W^n is the weight matrix of layer n, b^n is the bias vector of layer n, g^n is the activation function used in layer n, h^n denotes the output of the n^{th} hidden layer. [2]

The implementation of `FFNN` in the skeleton code looks like this:

```
#--- model ---
class FFNN(nn.Module):
    # Feel free to add whichever arguments you like here.
    def __init__(self, vocab_size, n_classes, extra_arg_1=None,
                                              extra_arg_2=None):
        super(FFNN, self).__init__():
        # WRITE CODE HERE
        pass

    def forward(self, x):
        # WRITE CODE HERE
        pass
```

There are two methods in the `FFNN` class. The first method `__init__` is supposed to initialize the model parameters. For representing the hidden layer(s), you will need weight matrices and bias vectors associated with that hidden layer. The most straightforward way is again using `torch.nn.Linear`.

Please use at least one hidden layer! We do not require a certain number of neurons in the hidden layer. It may be a good idea to start with a small number when developing your code, which will save some time in the initial experiments. Feel free with trying different numbers of hidden layers and neurons.

The second method `forward` computes the probabilities for our classes. In our case, the output y is a 1×3 tensor $(\log p_{\text{NEG}}, \log p_{\text{NEU}}, \log p_{\text{POS}})$. In this function, you can specify what kind of activation functions you would like your network to use for the hidden layers (ie., g^1 and g^2). You are welcome to play around with different activation functions! For g^3, since we are working on a classification task again, use the PyTorch function `torch.nn.functional.log_softmax`. [3]

[2] Note that the output layer does not have to employ an activation function, but in the case of a classification task, like in this assignment, it is makes things easier to use the `log_softmax`/`softmax` function.

[3] Once again, we could also return $(p_{\text{NEG}}, p_{\text{NEU}}, p_{\text{POS}})$ and use `torch.nn.softmax` but according to the PyTorch documentation, using `log_softmax` is preferable due to greater numerical stability.

2 Assignment: Loss Function and Optimizer

Now that the `FFNN` class is working, we need to implement a loss function and initialize the optimizer. You can do this in the main part of the program. The current skeleton code looks like this:

```
#--- set-up ---
model = FFNN(vocab_size, n_classes) #add extra arguments here if you use

# WRITE CODE HERE
loss_function = None
optimizer = None
```

You can add any extra arguments you would like to the `FFNN` class constructor, eg., one option would be:

```
model = FFNN(vocab_size, n_classes, n_hidden_1, n_hidden_2)
```

with the `n_hidden_k` argument denoting the number of neurons in the k^{th} hidden layer.

For the loss function, one option is to use the torch.nn.NLLLoss loss function, which will be appropriate for our classification problem. An alternative is the torch.nn.CrossEntropyLoss, however please note that this loss combines already the log_softmax and the NLLLoss, so unlike when using torch.nn.NLLLoss, you do not need to use a log_softmax function for the outputs when using the torch.nn.CrossEntropyLoss.

Please use a plain SGD optimizer for this homework.

3 Assignment: Training

The current training algorithm in the skeleton code looks like this:

```
#--- training ---
for epoch in range(N_EPOCHS):
    total_loss = 0
    # Generally speaking, it's a good idea to shuffle your
    # datasets once every epoch.
    random.shuffle(data['training'])

    for i in range(int(len(data['training'])/BATCH_SIZE)):
        minibatch = data['training'][i*BATCH_SIZE:(i+1)*BATCH_SIZE]
```

```
        # WRITE CODE HERE
        pass

    if ((epoch+1) % REPORT_EVERY) == 0:
        print('epoch: %d, loss: %.4f' %
              (epoch+1, total_loss*100/len(data['training'])))
```

The algorithm runs several epoch over the training data and trains on a single tweet at a time. Currently, the code does nothing.

Your task is to evaluate the loss for `tweet` given the current model parameters and the gold standard class `gold_class`. Then you need to perform a parameter update using `optimizer`. Don't forget to start by zeroing the gradient of `optimizer` so that your update will correctly reflect the current training example. You should also update the `total_loss` which tracks the cumulative loss over the entire training set. Just add the instance loss to `total_loss`.

The training algorithm assumes we are using minibatches for learning. Start with your `BATCH_SIZE` set to 1. We will increase this value in Section 5, once we make sure that the bare-bones implementation with a batch size of 1 is working correctly.

4 Assignment: Testing

Finally, now that you are able to train your model, you should implement classification. The current code for testing the model looks like this:

```
#--- test ---
correct = 0
with torch.no_grad():
    for tweet in data['test.gold']:
        gold_class = label_to_idx(tweet['SENTIMENT'])

        # WRITE CODE HERE
        # You can, but for the sake of this homework do not have to,
        # use batching for the test data.
        predicted = -1

        if IS_VERBOSE:
            print('TEST DATA: %s, GOLD LABEL: %s, GOLD CLASS %d, OUTPUT: %d' %
                 (' '.join(tweet['BODY'][:-1]), tweet['SENTIMENT'],
                  gold_class, predicted))

    print('test accuracy: %.2f' % (100.0 * correct / len(data['test.gold'])))
```

Currently, this simply returns -1 for every tweet. It also evaluates the accuracy on the held-out test set.

It is your task to use the `forward` function in the `FFNN` class to set `predicted` to the maximally probable class for each test example.

5 Assignment: Mini-Batch Training

Now, you should implement mini-batch training. In order to do this, you need to first increase `BATCH_SIZE`. Start by using something moderate like `10`. You can increase the batch size when everything is working properly.

Mini-batch training in PyTorch is almost trivial. You just need to pack your inputs (the examples in the list `minibatch`) into a tensor of dimension `BATCH_SIZE`$\times d_i$, where d_i is the regular input dimension for your model. Then you can simply call `model(x)` on your input batch `x`.

Once your batching is working properly, increase your `BATCH_SIZE` to 100. Using a larger batch size means you can (and generally do) increase your epoch count `N_EPOCHS` (since now the training is faster), and your learning rate `LEARNING_RATE` (since now the learning is more stable due to checking more than one data points at a time) as well. Experiment with these values.

In a bare–bones implementation, you can expect an eventual accuracy around or higher than 65%, although depending on your exact settings the number can also be different. If you get a too-low accuracy, check if everything runs as expected, and that your network is being trained properly. In particular, make sure that your loss is going down during the training procedure.

We hope you enjoy :)

Submission

Upload your `DL_hw1_USERNAME.py` or `DL_hw1_USERNAME.ipynb` file on Moodle.