CSCI 59000 NLP Spring 2024 Homework 4

March 5, 2024

Instructions

We will be using Canvas to collect your assignments. Please read the following instructions to prepare your submission.

1. Submit your solution in a pdf file and a zip file (<yourFirstName_LastName>.pdf/zip). Your write-up must be in pdf. Your code must be in the zip file.
2. In your pdf file, the solution to each problem should start on a new page.
3. Latex is strongly encouraged to write your solutions, e.g., using Overleaf (https://www.overleaf.com/). However, scanned handwritten copies are also acceptable. Hard copies will not be accepted.
4. You need to add screen captures of your code in your write-up.
5. You may discuss the problems and potential directions for solving them with another student. However, you need to write your own solutions and code separately, and not as a group activity. Please list the students you collaborated with on your submission.

Problem 1 (Programming involved, 50 points)

This homework problem is sourced from Professor Greg Durrett.

Please use up-to-date versions of Python and PyTorch. The only installed dependencies are the latest versions of numpy, torch, scipy, and torchvision. **The autograder will check if you use any unsupported dependencies.**

Data The dataset for this paper is the `text8`[1] collection. This is a dataset taken from the first 100M characters of Wikipedia. Only 27 character types are present (lowercase characters and spaces); special characters are replaced by a single space and numbers are spelled out as individual digits (*20* becomes *two zero*). A larger version of this benchmark (90M training characters, 5M dev, 5M test) was used in [1].

Framework code The framework code you are given consists of several files. We will describe these in the following sections.

`lm_classifier.py` contains the driver for Part 1 and handles the classification task. `models.py` contains skeletons in which you will implement your models for both parts and their training procedures.

Part 1: RNNs for Classification

In this first part, you will do a simplified version of the language modeling task: binary classification of fixed-length sequences to predict whether the given sequence is followed by a consonant or a vowel. You will implement the entire training and evaluation loop for this model.

Data `train-vowel-examples.txt` and `train-consonant-examples.txt` each contain 5000 strings of length 20, and `dev-vowel-examples.txt` and `dev-consonant-examples.txt` each contain 500. The task is to predict whether the first letter following each string is a vowel or a consonant. The consonant file (for both train and test) contains examples where the next letter (in the original text, not shown) was a consonant, and analogously for the vowel file.

Getting started Run:

```
python lm_classifier.py
```

This loads the data for this part, learns a `FrequencyBasedClassifier` on the data, and evaluates it. This classifier gets 71.4% accuracy, where random guessing gets you 50%. `lm_classifier`.py contains the driver code, and the top of `models.py` contains the skeletal implementation for this classifier.

Q1 (50 points) Implement an RNN classifier to classify segments as being followed by consonants or vowels. This will require defining a PyTorch module to do this classification, implementing training of that module in `train_rnn_classifier`, and finally completing the definition of `RNNClassifier` appropriately to use this module for classification.

Your final model should get **at least 75% accuracy.**[2]

[1] Original site: `http://mattmahoney.net/dc`

[2] Our reference implementation can get 76.3% accuracy in 2 minutes and 78.2% in 6 minutes of training with an unoptimized, unbatched implementation.

Network structure The inputs to your network will be sequences of character indices. You should first embed these using a `nn.Embedding` layer and then feed the resulting tensor into an RNN. Two effective types of RNNs to use are `nn.GRU` and `nn.LSTM`. Make sure you initialize their weights before the start of training! You may want to use the Glorot initializer (`nn.init.xavier_uniform_`).

You should take the output of the RNN (the last hidden state) and use it for binary classification with a softmax layer. You can add one or more feedforward layers before the softmax layer if you want. You can make your own `nn.Module` that wraps the embedding layer, RNN, and classification layer.

Code structure First, you need a function to go from the raw string to a PyTorch tensor of indices. Then loop through those examples, zero your gradients, pick up an example, compute the loss, run backpropagation, and update parameters with your optimizer. You should implement this training in `train_classifier`.

Using RNNs LSTMs and GRUs can be a bit trickier to use than feedforward architectures. First, these expect input tensors of dimension [sequence length, batch size, input size]. You can use the `batch_first` argument to switch whether the sequence length dimension or batch dimension occurs first. If you're not using batching, you'll want to pad your sentence with a trivial 1 dimension for the batch. `unsqueeze` allows you to add trivial dimensions of size 1, and `squeeze` lets you remove these.

Second, an LSTM takes as input a pair of tensors representing the state, h and c. Each is of size [num layers * num directions, batch size, hidden size]. To start with, you probably want a 1-layer RNN just running in the forward direction, so once again you should use `unsqueeze` to make a 3-tensor with first dimension length of 1. GRUs are similar but only have one hidden state.

Tensor manipulation `np.asarray` can convert lists into numpy arrays easily. `torch.from_numpy` can convert numpy arrays into PyTorch tensors. `torch.FloatTensor(list)` can convert from lists directly to PyTorch tensors. `.float()` and `.int()` can be used to cast tensors to different types.

General tips As always, make sure you can overfit a very small training set as an initial test. If not, you probably have a bug. Then scale up to train on more data and check the development performance of your model.

Consider using small values for hyperparameters so things train quickly. In particular, with only 27 characters, you can get away with small embedding sizes for these, and small hidden sizes for the RNN may work better than you think!

Make sure that the following command works before you submit your code `models.py`:

```
python lm_classifier.py --model RNN
```

This command should run without error and train.

Problem 2 (Programming involved, 50 points)

In this assignment, we will build a classifier using a pre-trained language model. We will use python packages, `Hugging Face`, `PyTorch`, `Scikit-Learn`, and `NumPy`. Please use up-to-date versions of the packages for this assignment.

Data

We will use the *polarity dataset v2.0* which was used for Homework 3. You can download the dataset from this website. When uncompressing the file, you will find data in two folders: *pos* and *neg*.

We will prepare our training and test datasets as in Homework 3, but with 9:1. Sort the files in each folder in ascending order using the filenames. Move the last 100 samples in each folder to new folders (`pos_test`, `neg_test`), which will be our test data. Change the original folder names into `pos_train` and `neg_train`, respectively. Now your `pos_train` and `neg_train` have 900 samples each, and `pos_test`, `neg_test` have 100 samples each. The files in _train and _test folders must be different.

`pos` and `neg` will be the labels for the documents. When you read the documents in your code, your code will have to remember the label for each document.

Coding Instructions

We will refer to this Hugging Face tutorial to build your own classifier. Write code following the steps below, referring to the tutorial. Include screenshots of your code in your write-up for each step.

(a) Implement a function called `read_data(...)`. The input of this function will be paths to `pos_train`, `neg_train`, `pos_test`, and `neg_test`. The output (return values) will be train_texts, train_labels, test_texts, and test_labels. We will tokenize the texts using the `DistilBert` tokenizer in the following steps.

(b) We will create a development dataset. Using the `train_test_split` function in *sklearn*, split the training data into training data (`train_texts` and `train_labels`) and development data (`dev_texts` and `dev_labels`). Make the dev dataset contain 100 positive samples and 100 negative samples. You can do this by changing the parameters of the function.

(c) Write code to tokenize the texts in the three datasets, using the `DistilBertTokenizerFast`.

(d) Now convert the tokenized data and labels into a Dataset object. Declare a class `PolarityDataset`, and create PolarityDataset objects for each dataset (i.e., training, dev, and test)

(e) Write code for fine-tuning with native PyTorch, using the DistilBert pre-trained model. You can refer to the tutorial. Explain each line of the code using comments.

(f) Add code to print the training loss and validation loss (i.e., loss on the dev set) every epoch.

(g) Add code to obtaining the performance of the trained model on the test data. Add code to print classification accuracy, precision, recall, and f1 score. [precision, recall, and f1 score from the positive ('pos') side]

(h) Run the code with AdamW optimizer, learning rate 5e-5, and batch size 16, and the number of epochs 3. What is the training loss and validation loss on epoch 3? What is the test performance when applying the trained model to the test data? Answer these questions in your write-up.

(i) Explore different learning rate, batch size, and the number of epochs. State the resulting changes together with what you changed and how you changed. Discuss the resulting changes (in terms of how and why).

(j) Make one change in the network architecture (not the optimizer/hyperparameters). State what change you made. Discuss the performance change (in terms of how and why).

References

[1] Tomas Mikolov et al. "Subword Language Modeling with Neural Networks". In: *Online preprint*. 2012.