Machine Learning for Natural Language Processing

Assignment 3: Linear classifier in PyTorch

Introduction

In this assignment, we will once again perform a binary classification task. This time we have a different dataset: we take data from the 'Conversations Gone Awry' corpus of the Cornell Conversation Analysis Toolkit (Zhang et al., 2018)[1]. This dataset concerns Wikipedia edit discussions that derailed into personal attacks. The tasks are to:

1. identify comments that contain personal attacks.
2. identify from comments whether the comment thread is likely to devolve into personal attacks (to pre-emptively stop it before it reaches that point).

You can choose either one of the two tasks. Each has its own quirks and challenges. In general, this is a very different dataset to the one in Lab 2. The main difference is that it is heavily skewed. Recall from Lab 1 that data skew can easily lead a classifier to maximise accuracy by only predicting the majority label. The same is true here. You will need to counteract this skew, and you will almost certainly find that the set of hyperparameters that worked well in Lab 2 does not work as well here.

(**Content warning:** this dataset contains profanities. If you would rather not see these, task 2 is much less likely to display them as top features. Alternatively, you can change the filepath in the `main()` function to `/local/course/ml/2019/assignment3/tokenised_conversations_filtered.json` for a filtered version of the dataset.)

Much of the code is the same. However, in this Assignment we will use PyTorch, a machine learning library. If you have not already done so, you may find it useful to familiarise yourself with some of the basic data structures and methods of PyTorch - in particular, Tensors. Parts 1–3 of this tutorial cover most of what we will do in this Assignment.[2] PyTorch is well-documented in general, and you can probably find a lot of help and inspiration from its `Docs` section.[3]

Preparation

As before, start by downloading the starter code to your working directory.

```
mkdir your/working/directory
cd your/working/directory
cp /local/kurs/ml/2019/assignment3/Assignment3.zip .
unzip Assignment3.zip
```

Also as before, there is a virtual environment with all dependencies for this assignment in place. Activate it as below.

```
source /local/kurs/ml/2019/assignment3/pytorch/bin/activate
```

Open `classifier.py` in your text editor and read through it carefully.

1 Defining the model

On line 35, you will find `class Net(torch.nn.Module)`. This is the model that we will iteratively train throughout this assignment. Note that it inherits from the `torch.nn.Module` superclass; this will be important

[1] https://github.com/CornellNLP/Cornell-Conversational-Analysis-Toolkit/blob/master/datasets/conversations-gone-awry-corpus/awry.README.v1.00.txt

[2] https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

[3] https://pytorch.org/docs/stable/index.html

later.

Implementation Your first task is to define the model's forward function. It takes as input a tensor of training examples, with dimensions $m \times n$, where m is the number of training examples and n is the number of features. (Note that unlike in Lab 2, this is a dense representation; that is, we don't exclude the zero values.)

$$input = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix}$$

The weights and bias are already defined for you. `weights` is a 1D tensor of length $n = nfeatures$, while `bias` is a scalar. Because weights has only one defined dimension, it will be compatible for dot product with the whole input tensor along dimension n.

Don't overthink this; it's really just the same as $x \cdot w + b$, but you can do it over the whole input tensor rather than line by line. The function for dot product in PyTorch is `torch.matmul`.

2 Autograd and Optimisation

Now that we have the forward function, the training loop will run. However, you'll notice that it never actually learns. We now need to do the following in the training loop (line 71):

- Get the derivatives of the weights with regard to the loss function (backpropagation).
- Perform the gradient descent step (optimisation).

Fortunately, PyTorch can do most of this for us automatically with little effort on our part.

Implementation

- First, find out how to automatically compute the gradients backwards from the loss function (backpropagation). This is done by PyTorch's `autograd` package.[4]
- Second, find out how to use PyTorch's optimiser class `torch.optim` to automatically update the weights.[5] There are many built-in optimisers that you could use, but for now let's stick to the simplest (and most familiar): Stochastic Gradient Descent (SGD).

Report Explain what changes you had to make to the code to backpropagate the loss gradients and perform the stochastic gradient descent.

3 Loss functions

On lines 163 and 170 you'll find our old loss functions, Logistic Loss and Hinge Loss. PyTorch has in-built loss functions such as `MSELoss`, which is in the hyperparameters. However, it's good practice to know how loss is computed using Tensors - especially as you may be writing your own loss function some day.

Implementation Complete the functions for logistic loss and hinge loss. Regularisation is not necessary (you can find those already implemented in `regularisers.py` if you want to use them later).

(**Note:** Logistic loss is actually implemented in PyTorch as `SoftMarginLoss`, so it may be good to compare your implementation to this gold standard. Also note that the PyTorch implementation uses mean reduction instead of log reduction as the constant; $\frac{1}{|\hat{Y}|}\log(1-\exp(-y\hat{y}))$ instead of $\frac{1}{\log 2}\log(1-\exp(-y\hat{y}))$, where $|\hat{Y}|$ is the number of elements in the vector of predicted labels.)

[4] https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

[5] https://pytorch.org/docs/stable/optim.html

4 Tuning the classifier

Now that you've implemented everything, you can begin the task of trying to achieve the best classifier possible by tuning the hyperparameters (and making any modifications you like). Go to the `main()` function (line 207).

Choose your task by selecting either `'comment'` (for comment level classification) or `'conversation'` (for thread-level classification) as `chosen_task`.

There is a new feature to the performance readout: I have added F1-score[6]. As mentioned before, data skew is a big problem in this dataset. You should therefore consider both F1 score and accuracy as performance metrics. I will leave it up to you to decide which to treat as most important when evaluating your models, but you should motivate your decision.

Report Report on the following:

- Give details of your best set of hyperparameters and any modifications you made. No need for graphs or tables, but describe your methodology and the motivations behind your decisions.
- Take a look at the most positive and negative features identified by your best hyperparameters. Do they make intuitive sense to you? Why or why not, and what do you think caused them?

5 Testing and reflection

When you're ready, enable test scoring and try on the test set.

Report Report on the following:

- Did the classifier generalise well to the test data? Why or why not?
- What other improvements do you think could have been made to the classifier - at any stage - to improve its performance further? Does PyTorch have tools to allow for these improvements?
- Overall, how did you find the experience of implementing the linear classifier in PyTorch compared to the native Python version?

6 VG Tasks (Optional)

The above is sufficient for a passing (G) grade. To get a VG on this Assignment, you need to complete all of the above tasks without major problems **and** complete at least one of the following tasks:

6.1 Minibatch training

The default classifier is based on batch training. It takes in all of the training data and calculates loss over the whole of the data size, and backpropagates the loss over the whole of the data size. Minibatch training, on the other hand, will take in the training data in subsets, updating after each subset, until it has read all of the training data. A more full explanation is here. [7]

Modify the training loop so that training data is fed in in minibatches, and loss is calculated and backpropagated over these smaller batch sizes. You will need to work out how to slice and shuffle the training example and label tensors. Experiment with different batch sizes (including the perceptron-style online batch size of 1) and report on the results.

6.2 Checkpointing/Early Stopping

A common problem in linear classifiers is that, after reaching an optimum, their validation loss suddenly shoots back up due to overfitting. You have probably seen this in the readout, where the final training epoch is not necessarily the best one. Even without this, it is possible for a model to converge early in its training, meaning that more training epochs are a waste of time and processing power.

Implement the following features in the classifier:

[6] For the sake of convenience, we will treat 'undefined' (what happens when Recall is zero) as 0. This avoids the '?' problem from Lab 1.

[7] https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/

- Stop training early if a chosen metric (typically validation loss or accuracy) doesn't improve. You should also include a patience parameter *p*, so the model will terminate early after *p* epochs without improvement.
- Save the best model parameters and load them at the end of training. (**Note:** If saving these into a file, you might consider placing them in the /tmp folder, or your /nobackup folder.)

Report on the results. How long does it usually take to reach the best parameters? What did you find was the best *p* value?

6.3 Multilayer Perceptron

Since our `Net` object inherits from the `nn.Module` class, we can use it as the basis for a multilayer perceptron (MLP). This is another term for a Feed Forward Neural Network (FFNN) with one or more hidden layers[8]. Modify the `Net` parameters and forward function to behave more like a MLP and run it on the dataset.

Report on the following:

- What changes did you have to make to the code in order for this to function as a MLP?
- What are the advantages and disadvantages of the MLP compared to the vanilla linear classifier?

Submission

Please submit a written report of no more than 5 pages, along with your modified `classifier.py` and any other files you needed to modify (if any) via Studentportalen. The first submission deadline is 23:59 on (2019-03-11); the resubmission deadline is 23:59 on 2019-05-20.

Your code should be able to be run from the command line using `python classifier.py` (or other filenames as appropriate). The code should be stable; i.e. it should not crash.

[8] And unrelated to the perceptron learning algorithm.