CSE 256 Fall 2024, UCSD: PA1 (100 points)

Deep Averaging Networks | Byte Pair Encoding | Skip-Gram

Released: October 4, 2024 —— Due: October 18, 2024

The main goal of this assignment is for you to get experience training neural networks for text tasks. You'll see the standard pipeline used in many NLP tasks (reading in data, preprocessing, training, and testing). You will also get experience with sub-word tokenization. Lastly, you will work on conceptual exercises in order to get a better understanding of the skip-gram word embedding algorithm.

Preliminaries

Software. In this course, assignments will be in Python. We will work with **Python 3.6 or later**. We will be using **PyTorch**, a Python-based machine learning framework. To set up PyTorch on your machine, follow the instructions at `https://pytorch.org/get-started/locally/`. This assignment is small-scale and can be completed using CPU only, so there's no need to install CUDA or set up GPU support unless you choose to. Installing via **Anaconda** is typically the easiest method, especially if you are using OS X. While working in a virtual environment is recommended for better dependency management, it is not required.

First, create a new virtual environment with Python 3:

```
conda create -n my-virtenv python=3
```

where `my-virtenv` can be any name you choose.

Then, if you're running Linux, install PyTorch with:

```
conda install -n my-virtenv -c pytorch pytorch-cpu
```

If you're on Mac, use:

```
conda install -n my-virtenv -c pytorch pytorch
```

Data. You'll be using the dataset of Socher et al. [4], which consists of movie review snippets taken from Rotten Tomatoes. The task is a positive/negative binary sentiment classification of sentences. The data files given to you have newline-separated sentiment examples, consisting of a label (0 or 1) followed by a tab, followed by the sentence, which has been tokenized but not lowercased. The data has been split into a train, and development (dev) set.

Code. The following files are provided to you:

- `sentiment_data.py` handles data reading. This also defines a SentimentExample object, which wraps a list of words with an integer label (0/1), as well as a WordEmbeddings object, which contains pre-trained word embeddings for this dataset.
- `utils.py` implements an Indexer class, which can be used to maintain a mapping between indices and words in the vocabulary.
- `BOWmodels.py` contains simple discrete Bag-of-Words models for the sentiment classification task on the provided dataset. You can use this as a reference for implementing your Deep Averaging Network (DAN) in part 1.
- `DANmodels.py` an empty file where you will implement your DAN models.
- `main.py` shows how to evaluate the Bag-of-Words models on the sentiment classification task. You can use this as a reference for evaluating your DAN models.

Getting Started. Expand the zip file and change into the directory. To confirm everything is working properly, run:

```
python main.py --model BOW
```

This loads the data, trains the models defined in `BOWModels.py`, and evaluates them on the training and dev sets. The reported dev accuracy should be around 0.5. This BOW models count the number of times each word appears in the sentence and does not use pretraining word embeddings. The vocabulary size is truncated to 512 for BOW dimensionality. This model should be easy to beat with the DAN model you will implement.

Part 1: Deep Averaging Network (DAN) (50 Points)

In this part, you'll implement a deep averaging network as discussed in the lecture and in [1]. If our input $s = (w_1, \ldots, w_n)$, then we use a feedforward neural network for prediction with input

$$\frac{1}{n}\sum_{i=1}^{n} e\left(w_i\right)$$

, where e is a function that maps a word w to its real-valued vector embedding. Implement the DAN model in `DANmodels.py`.
You'll need the WordEmbeddings class. This class wraps a matrix of word vectors and an Indexer in order to index new words. The Indexer contains two special tokens: `PAD` (index 0) and `UNK` (index 1). You'll want to use `get_initialized_embedding_layer` to get a `torch.nn.Embedding` layer that can be used in your network. This layer is trainable (if you set frozen to False), but is initialized with the pre-trained embeddings.

Pretrained Embeddings

You are given two sources of pretrained embeddings you can use: `glove.6B.50d-relativized.txt` and `data/glove.6B.300d-relativized.txt`. These are trained using GloVe by Pennington et al. [2]. These vectors have been relativized to your data, meaning that they do not contain embeddings for words that don't occur in the train, or dev data. This is purely for runtime and memory optimization.

Implementation And Debugging Tips

- Following the example of the Bag-of-Words models, you should implement a `DAN` class that inherits from `torch.nn.Module`. This class should have an `__init__` method that initializes the layers of the network and a `forward` method that defines how input is passed through the network. Your module should take a list of word indices as input and embed them using a `nn.Embedding` layer initialized appropriately.
- Compute your classification loss based on the prediction. In class, we saw using the negative log probability of the correct label as the loss. You can do this directly, or you can use a built-in loss function like `NLLLoss` or `CrossEntropyLoss`. Pay close attention to what these losses expect as inputs (probabilities, log probabilities, or raw scores).
- You should print training/dev loss at certain intervals of training epochs; this will give you an idea of how the learning process is proceeding. Much like it is done in `main.py` for the Bag-of-Words models.
- You should also print the accuracy of your model on the dev set as training progresses. This will give you an idea of how well your model is generalizing to unseen data. Here again, you can refer to `main.py`.
- If you see NaNs in your code, it's likely due to a large step size. log(0) is the main way these arise.
- Google/Stack Overflow and the PyTorch documentation are your friends. Although you should not seek out prepackaged solutions to the assignment itself, you should avail yourself of the resources out there to learn the tools.

Deliverables

1a) Implement the deep averaging network (DAN). Your implementation should consist of averaging vectors and using a feedforward network, but otherwise you do not need to exactly reimplement what's discussed in Iyyer et al. [1]. Things you can experiment with, and can discuss in your report include varying the number of layers, the hidden layer sizes, which source of embeddings you use (50d or 300d), your optimizer (Adam is a good choice), the nonlinearity, whether you add dropout layers (after embeddings? after the hidden layer?), and your initialization. **Briefly describe what you did and report your results in the writeup.**

You should get **at least** 77% **accuracy on the development set** you should aim for your model to train in less than 5 minutes or so (and you should be able to get good performance in 2 minutes on a modern laptop).

1b) **Randomly initialized embeddings**. Remove the GloVe initialization from the model; just initialize the embedding layer with random vectors, which should then be updated during learning. How does this compare to using GloVe? Discuss how training embeddings from scratch affects model convergence and performance.

Part 2: Byte Pair Encoding (30 Points)

In this part you will modify the Deep Averaging Network (DAN) to use subword tokenization in place of word-level tokenization. Implement the **Byte Pair Encoding (BPE)** algorithm as discussed in class, and in Sennrich et al. [3] and train BPE on 'train.txt'. Use the sub-word vocabulary generated by BPE in your DAN, instead of the word-level vocabulary.

- **Vocabulary Size**: Experiment with different vocabulary sizes for the subword-based model and observe how it impacts model performance and computational efficiency.

Since you will be using subword units instead of full words, you will not have access to pre-trained word embeddings. You must **initialize the embeddings randomly** using the `torch.nn.Embedding` layer (as in Part 1b) and train them from scratch.

Deliverables

2a) Compare the performance of your subword-based DAN model to the word-level DAN model you implemented earlier. Experiment with different vocabulary sizes for the subword-based model and observe how it impacts model performance, and discuss your findings.

Part 3: Understanding Skip-Gram (20 Points)

Consider the skip-gram model, defined by

$$P(context = y \mid word = x) = \frac{\exp\left(\mathbf{v}_x \cdot \mathbf{c}_y\right)}{\sum_{y'} \exp\left(\mathbf{v}_x \cdot \mathbf{c}_{y'}\right)}$$

where x is the "center word", y is the "context word" being predicted, and $\mathbf{v}, \mathbf{c}$ are d-dimensional vectors corresponding to words and contexts, respectively. Note that each word has independent vectors for each of these, so each word really has two embeddings.

Q1 (10 points) Consider the following sentences:

```
the dog
the cat
a dog
```

Assume a window size of $k = 1$. The skip-gram model considers the neighbors of a word to be words on either side. So with these assumptions, the first sentence above gives the training examples

($x = the$, $y = dog$) and ($x = dog, y = the$). The skip-gram objective, log likelihood of this training data, is $\sum_{(x,y)} \log P(y \mid x)$, where the sum is over all training examples.

Suppose that we have word and context embeddings of dimension $d = 2$, the context embedding vectors w for *dog* and *cat* are both $(0, 1)$, and the context embedding vectors w for *a* and *the* are $(1, 0)$. Treat these as fixed.

3a) If we treat the above data as our training set, what is the set of probabilities $P(y \mid \text{the})$ that maximize the data likelihood? Note: this question is asking you about what the optimal probabilities are independent of any particular setting of the skip-gram vectors.

Hint: when we have a biased coin and observe the pattern heads, heads, heads, tails, our maximum likelihood estimate of P (heads) $= \frac{3}{4}$ *and P(tails)* $= \frac{1}{4}$*. That is, the likelihood is maximized by having the model match the empirical distribution of outcomes observed in the data.*

3b) Can these probabilities be nearly achieved with a particular setting of the word vector for *the*? Give a nearly optimal vector (returns probabilities within 0.01 of the optimum) and a description of why it is optimal. (Truly optimizing models with softmaxes in them typically requires infinitely-large weights. You are free to use such weights, but you can also just use large constants instead.)

Q2 (10 points) For this question, consider the following sentences:

- `the dog`
- `the cat`
- `a dog`
- `a cat`

Now suppose the dimensionality of the word embedding space $d = 2$. Write down the following:

3c) The training examples derived from these sentences

3d) A set of both word and context vectors that nearly optimizes the skip-gram objective. These vectors should give probabilities within 0.01 of the optimum.

Submission Instructions

Submit on Gradescope (more details on Gradescope to be provided.).

- **Code:** You will submit your code together with a neatly written README file with instructions on how to run your code. We assume that you always follow good practice of coding (commenting, structuring), thus these factors are not central to your grade.

  Allow us to run your code with a single command for each part, e.g.,

  ```
  python main.py --model DAN
  python main.py --model SUBWORDDAN
  ```

  Any other options for running your model(s) should be documted in your README.

- **Report:** Your writeup should be 5 pages or less in PDF (with reasonable font sizes). Focus on reporting your results and describing what you did, mentioning any distinctive aspects of your implementation.

  You won't be penalized for exceeding the page limit, but quality is preferred over quantity. Avoid including low-level code documentation, which belongs in comments or the README. Use tables or figures to present results instead of copying the entire system output.

  For part 3, show your work along with your answer.

References

[1] Mohit Iyyer, Varun Manjunatha, Jordan L. Boyd-Graber, and Hal Daumé III. Deep unordered composition rivals syntactic methods for text classification. In *ACL*, pages 1681–1691. The Association for Computer Linguistics, 2015.

[2] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *EMNLP*, pages 1532–1543. ACL, 2014.

[3] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Katrin Erk and Noah A. Smith, editors, *ACL*, pages 1715–1725, 2016.

[4] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *ACL*, pages 1631–1642. ACL, 2013.