CS371N Assignment 2: Feedforward Neural Networks

Academic Honesty: Please see the course syllabus for information about collaboration in this course. While you may discuss the assignment with other students, **all code you write and your writeup must be your own!**

Goals The main goal of this assignment is for you to get experience training neural networks over text. You'll play around with feedforward neural networks in PyTorch and see the impact of different sets of word vectors on the sentiment classification problem from Assignment 1.

Code Setup

Please use Python 3.5+ and a recent version of PyTorch for this project.

The list of installed packages in the autograder is: `numpy`, `nltk`, `spacy`, `torch`, `scipy`, `matplotlib`, `torchvision`.

Installing PyTorch You will need PyTorch for this project. **If you are using CS lab machines, PyTorch should already be installed and you can skip this step.** To get it working on your own machine, you should follow the instructions at `https://pytorch.org/get-started/locally/`. The assignment is small-scale enough to complete using CPU only, so don't worry about installing CUDA and getting GPU support working unless you want to.

Installing via anaconda is typically easiest, especially if you are on OS X, where the system python has some weird package versions. Installing in a virtual environment is recommended but not essential. Once you have anaconda installed, you can create a virtual environment and install PyTorch with the following commands:

```
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 torchvision-cpu
```

If you're on Mac, use:

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

Part 1: Optimization (25 points)

In this part, you'll get some familiarity with function optimization. **Note that although you will do some coding here, you do not need to submit your code and there is no autograder for this part.**

Q1 (15 points) First we start with `optimization.py`, which defines a quadratic with two variables:

$$y = (x_1 - 1)^2 + 8(x_2 - 1)^2$$

This file contains a manual implementation of SGD for this function. Run:

```
python optimization.py --lr 1
```

to optimize the quadratic with a learning rate of 1. However, the code will crash, since the gradient hasn't been implemented.

a) Implement the gradient of the provided quadratic function in `quadratic_grad`. `sgd_test_quadratic` will then call this function inside an SGD loop and show a visualization of the learning process. **Note: you should not use PyTorch for this part!** Report the gradient in your writeup; this code will not be autograded.

b) With a step size of 0.01, how many steps are needed to get within a distance of 0.1 of the optimal solution (that is, within a distance of 0.1 from the point $(1, 1)$, where the function is minimized)? You can round your answer to the nearest 10 steps; you don't need an exact answer.

c) What is the "tipping point" of the step size parameter, where step sizes larger than that cause SGD to diverge rather than find the optimum? You can measure this empirically; you don't need an analytical solution. You can round your answer to the nearest 0.01.

d) When initializing at the origin, what is the best step size to use? Measure this in terms of number of iterations to get with 0.1 of the optimum. You can round your answer to the nearest 0.01. **Include a plot of optimization with this best step size using the provided contour plot machinery.**

Q2 (10 points) In this part, you'll get some additional familiarity with optimization, this time using PyTorch. We define another function in `rosenbrock.py`. The Rosenbrock function is a classic test for optimization algorithms:

$$y = \sum_{i=1}^{n-1} \left(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right)$$

It attains its minimum at $x = (1, 1, \ldots, 1)$, all ones, but is very difficult to optimize with gradient-based methods.[1]

The main function in `rosenbrock.py` will optimize this using either SGD or a variant of SGD called Adam[2]:

```
python rosenbrock.py --method SGD --lr 1.0
python rosenbrock.py --method ADAM --lr 1.0
```

Try a few different step sizes for SGD and Adam on the Rosenbrock function. For each optimization technique, describe the range of behaviors you see depending on step size. (Hint: try varying step sizes by orders of magnitude, like 1, 0.1, 0.001, etc.)

Part 2: Deep Averaging Network (50 points)

In this part, you'll implement a deep averaging network as discussed in lecture and in Iyyer et al. (2015). If our input $s = (w_1, \ldots, w_n)$, then we use a feedforward neural network for prediction with input $\frac{1}{n}\sum_{i=1}^{n} e(w_i)$, where e is a function that maps a word w to its real-valued vector embedding.

Getting started Download the code and data; the data is the same as in Assignment 1. Expand the tgz file and change into the directory. To confirm everything is working properly, run:

```
python neural_sentiment_classifier.py --model TRIVIAL --no_run_on_test
```

[1] Wikipedia has a nice figure and description: `https://en.wikipedia.org/wiki/Rosenbrock_function`

[2] Discussed in the neural net optimization video.

This loads the data, instantiates a `TrivialSentimentClassifier` that always returns 1 (positive), and evaluates it on the training and dev sets. Compared to Assignment 1, this runs an extra word embedding loading step.

Framework code The framework code you are given consists of several files. `neural_sentiment_classifier.py` is the main class. As before, you cannot modify this file for your final submission, though it's okay to add command line arguments or make changes during development. You should generally not need to modify the paths. The `--model` and `--feats` arguments control the model specification. The main method loads in the data, initializes the feature extractor, trains the model, and evaluates it on train, dev, and blind test, and writes the blind test results to a file.

`models.py` is the file you'll be modifying for this part, and `train_deep_averaging_network` is your entry point, similar to Assignment 1. Data reading in `sentiment_data.py` and the utilities in `utils.py` are similar to Assignment 1. However, `read_sentiment_examples` **now lowercases the dataset**; the GloVe embeddings do not distinguish case and only contain embeddings for lowercase words.

`sentiment_data.py` also additionally contains a `WordEmbeddings` class and code for reading it from a file. 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). `UNK` can stand in words that aren't in the vocabulary, and `PAD` is useful for implementing batching later. Both are mapped to the zero vector by default.

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, which will be slower) but is initialized with the pre-trained embeddings.

Data You are given two sources of pretrained embeddings you can use: `data/glove.6B.50d-relativized.txt` and `data/glove.6B.300d-relativized.txt`, the loading of which is controlled by the `--word_vecs_path`. These are trained using GloVe (Pennington et al., 2014). These vectors have been *relativized* to your data, meaning that they do not contain embeddings for words that don't occur in the train, dev, or test data. This is purely a runtime and memory optimization.

PyTorch example `ffnn_example.py`[3] implements the network discussed in lecture for the synthetic XOR task. It shows a minimal example of the PyTorch network definition, training, and evaluation loop. Feel free to refer to this code extensively and to copy-paste parts of it into your solution as needed. Most of this code is self-documenting. **The most unintuitive piece is calling `zero_grad` before calling backward!** Backward computation uses in-place storage and this must be zeroed out before every gradient computation.

Implementation Following the example, the rough steps you should take are:

1. Define a subclass of `nn.Module` that does your prediction. This should return a log-probability distribution over class labels. Your module should take a list of word indices as input and embed them using a `nn.Embedding` layer initialized appropriately.

2. Compute your classification loss based on the prediction. In lecture, 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

[3] Available under "Readings" on the course website.

function like `NLLLoss` or `CrossEntropyLoss`. Pay close attention to what these losses expect as inputs (probabilities, log probabilities, or raw scores).

3. Call `network.zero_grad()` (zeroes out in-place gradient vectors), `loss.backward` (runs the backward pass to compute gradients), and `optimizer.step` to update your parameters.

Implementation and Debugging Tips Come back to this section as you tackle the assignment!

- You should print training loss over your models' epochs; this will give you an idea of how the learning process is proceeding.
- You should be able to do the vast majority of your parameter tuning in small-scale experiments. Try to avoid running large experiments on the whole dataset in order to keep your development time fast.
- If you see NaNs in your code, it's likely due to a large step size. $\log(0)$ is the main way these arise.
- For creating tensors, `torch.tensor` and `torch.from_numpy` are pretty useful. For manipulating tensors, `permute` lets you rearrange axes, `squeeze` can eliminate dimensions of length 1, `expand` or `repeat` can duplicate a tensor across a dimension, etc. You probably won't need to use all of these in this project, but they're there if you need them. PyTorch supports most basic arithmetic operations done elementwise on tensors.
- To handle sentence input data, you typically want to treat the input as a sequence of word indices. You can use `torch.nn.Embedding` to convert these into their corresponding word embeddings.
- Google/Stack Overflow and the PyTorch documentation[4] 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.

Q3 (35 points AUTOGRADED, 15 points WRITEUP)

a) Implement the deep averaging network. 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. (2015). Things you can experiment with 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.**

b) Implement batching in your neural network. To do this, you should modify your `nn.Module` subclass to take a batch of examples at a time instead of a single example. You should compute the loss over the entire batch. Otherwise your code can function as before. You can also try out batching at test time by changing the `predict_all` method. **Try at least one batch size greater than one. Briefly describe what you did, any change in the results, and what speedup you see from running with that batch size.**

Note that different sentences have different lengths; to fit these into an input matrix, you will need to "pad" the inputs to be the same length. If you use the index 0 (which corresponds to the `PAD` token in the indexer), you can set `padding_idx=0` in the embedding layer. For the length, you can either dynamically choose the length of the longest sentence in the batch, use a large enough constant, or use a constant that isn't quite large enough but truncates some sentences (will be faster).

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

Requirements You should get least **77% accuracy** on the development set in less than **10 minutes** of train time on a CS lab machine (and you should be able to get good performance in 3-5 minutes). Your final implementation that you submit can be either batched or unbatched, but **you should implement both**.

If the autograder crashes but your code works locally, there is a good chance you are taking too much time. You may need to reduce the runtime below 10 minutes to get the autograder to complete, as the autograder VMs are underpowered and have some variance in performance. Try reducing the number of epochs so your code runs in 3-4 minutes and resubmitting to at least confirm that it works.

c) Try removing 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? (You do not need to submit these changes to your code.)

Part 3: Understanding Word Embeddings (25 points)

Consider the skip-gram model, defined by

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

where x is the "main 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.

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

Q4 (5 points) Consider the following sentences:

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

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.

a) If we treat the above data as our training set, what is the set of probabilities $P(y|\textit{the})$ that maximizes 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 $P(\text{heads}) = \frac{3}{4}$ and $P(\text{tails}) = \frac{1}{4}$. That is, likelihood is maximized by having the model match the empirical distribution of outcomes observed in the data.

b) 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.[5]

Q5 (10 points) Now 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:

a) The training examples derived from these sentences

b) 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.

Q6 (10 points) Suppose our word embedding space is V dimensions where V is the number of tokens. Each context vector is a one-hot vector: a vector of all zeroes with a single 1 at the corresponding index in the vector space. So word 1 has the context vector $(1, 0, 0, \ldots, 0)$.

a) Suppose word w_4 occurs with word w_1 twice, word w_2 once, word w_3 once, and no other words. What is the optimal word vector choice for w_4? (Note that there are multiple possible answers.)

b) Using the same context embeddings and the optimal word vectors from above, suppose you have the following co-occurence statistics for the word *king*, *man*, *woman*, and *queen*:

king	*royal* x4, *decree* x3, *he* x10
man	*he* x10
woman	*she* x10
queen	*royal* x4, *decree* x3, *she* x10

(There may be other words in the vocabulary besides these, but they do not affect the results.)

One regularity in embedding spaces frequently reported in the literature is $\mathbf{v}_{king} - \mathbf{v}_{man} + \mathbf{v}_{woman} = \mathbf{v}_{queen}$. Applying your vector creation method from part (a), you should be able to construct a set of optimal embeddings[6] for these vectors that exhibits this regularity. Write down these embeddings, and explain why this regularity holds here.

c) Explain whether you think this regularity would hold using real counts from real text. What differences in co-occurrence patterns might cause this to break down?

[5] Truly optimizing models with softmaxes in them typically requires infinitely-large weights. You're free to use such weights, but you can also just use large constants instead.

[6] If your embeddings from part (a) have infinitely large values, clip these to the range [-100, 100] for the purposes of this question.

Deliverables and Submission

You will submit both your code and writeup to Gradescope. These are submitted as **two separate uploads** to Gradescope.

Written Submission You should upload to Gradescope a PDF or text file of your answers to the questions. This can be handwritten and scanned/photographed if that works best for you.

Note that you can submit the written assignment independently of the code. If you are unable to get the code fully working, please write up what you did and answer as many questions as possible, even partially, so we can assign you appropriate partial credit.

Please put your name on the assignment, and select a page with your name on it as the "question page" for any autograded question parts.

Code Submission Your code in `models.py` will be evaluated by our autograder on several axes:

1. Execution: your code should train and evaluate within the time limits without crashing
2. Accuracy on the development set of your deep averaging network model using 300-dimensional embeddings
3. Accuracy on the blind test set: this is not explicitly reported by the autograder but we may consider it, particularly if it differs greatly from the dev performance (by more than a few percent)

Note that we will only evaluate your code with 300-dimensional embeddings.

Make sure that the following command works before you submit:

```
python neural_sentiment_classifier.py
```

References

Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. 2015. Deep Unordered Composition Rivals Syntactic Methods for Text Classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.