Homework 1 Instructions

NLP with Representation Learning, Fall 2022
Instructor: Tal Linzen
TAs: Rui Chen, Abhinav Gupta, Will Merrill, Ying Wang

General Instructions

1. *This document describes what you need to do for each part of the assignment. You will be writing code in various Python files and some short responses in README.md.*
2. *In each code file, the part you need to fill in will be marked with "`TODO`".*
3. *If you are instructed to provide a short response to a question, the answer should go in README.md under the corresponding section.*
4. *Good luck! Any questions about the assignment can be posted on CampusWire.*
5. *Check this CampusWire thread for a list of corrections and clarifications.*

Environment Setup

This assignment requires the Python packages `requests`, `numpy`, and `torch`. You can install these using pip or conda. Our recommended way to do this is within a conda environment:

```
conda create -n cds-nlp python=3.8
conda activate cds-nlp
pip install numpy requests
pip install torch
```

Depending on the machine you are using, you may need to tweak the last command using the PyTorch installation widget.

Part 1: N-Gram Models

In this section, you will implement a basic n-gram model with no smoothing. The core challenge in doing this is counting the number of occurrences of different n-grams in the training data.

a. In ngrams_vanilla.py, implement the function `estimate`. The purpose of this function is to count the frequencies of different n-grams in the training data. See the docstring of the `estimate` function for more details about how it is supposed to work. Run the following command and paste the output in README.md:

```
python fit_ngrams.py vanilla
```

Part 2: Additive Smoothing

In this section, you will extend your n-gram model with additive smoothing.

a. In your own words, what is the purpose of smoothing in an n-gram model?

b. In ngrams_additive.py, implement the function `ngram_prob` with additive smoothing. You can run the script with the following to test your implementation:

```
python fit_ngrams.py additive --delta=0.005
```

In README.md, report the output of running the command above.

c. You can tweak the hyperparameter n, which controls the n-gram size, and `delta`, which controls the strength of the additive smoothing, by modifying the command above. Is 0.005 a good value for `delta`? Compare the model's validation perplexity varying `n` over {2, 3} and across several orders of magnitude of `delta` (e.g., 0.0005, 0.005, 0.05). What is the best-performing pair of hyperparameters `(n, delta)` in your search?

Part 3: Interpolation Smoothing

In this section, you will implement an n-gram model with interpolation smoothing. Interpolating smoothing works by fitting several n-gram models with n between 1 and N and then taking a weighted sum of their predictions. Importantly, the weights together must sum to 1.

a. In ngrams_interpolation.py, implement the function `ngram_prob`. You can run the script as follows to test your implementation:

```
python fit_ngrams.py interpolation
```

In README.md, report the output of running the command above.

b. The parameters `lambda1`, `lambda2`, and `lambda3` are the weights assigned to unigram, bigram, and trigram model. Try 5 different settings of these weights, respecting the constraint that they sum to 1. What setting performs the best?

Part 4: Backoff

Like smoothing, backoff is a method for dealing with the sparsity in the training data with an n-gram language model. With backoff, any n-gram that has a count of 0 in the training data is estimated with a smaller n-gram.

For example, imagine we are trying to estimate the probability of "*house"* given "*the green*". A vanilla n-gram model would divide the count of "*the green house*" by the count of "*the green*" to

compute this probability. With backoff, though, the count of "*the green house*" is 0 in the training data, so we instead divide the count of "*green house*" by the count of "*green*".

Note that in the case where the count of "*green house*" is 0, we instead back off to a unigram model, i.e., divide the count of "*house*" by the count of the empty string (total number of tokens).

a. In ngrams_backoff.py, implement the function `ngram_prob`. You can run the script as follows to test your implementation:

   ```
   python fit_ngrams.py backoff
   ```

   In README.md, report the output of running the command above.

Part 5: Test Set Evaluation

Now that you have fit hyperparameters on the validation set, you can compute the perplexity of each model type on the test set using the `--test` flag, for example:

```
python fit_ngrams.py additive --delta=0.005 --test
```

b. Select the best hyperparameters for each model type (vanilla, additive smoothing, interpolation smoothing, and backoff). Report perplexity for each model on the test set. Which model performs the best overall?

Part 6: A Taste of Neural Networks

This part of the assignment is independent from everything that came before.[1] You will be training a neural network text classifier to perform the task of sentiment analysis. Sentiment analysis involves processing an input sentence to classify whether its sentiment is positive (1) or negative (0). To get a better sense for this task setup, you can browse through the data in part6/data/train.txt.

Provided Code and Data

The framework code you are given consists of several files. neural_sentiment_classifier.py is the main class. You cannot modify this file for your final submission, although you can run it with different command line arguments. 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.

[1] Thank you to Greg Durett, from whose course this section was adapted.

Steps

You can test your code by running neural_sentiment_classifier.py, but the only file you should change is models.py. You will need to implement the following:

1. **`NeuralNet`**: Implement the constructor as well as `NeuralNet.forward:` the function that runs a forward pass of the neural net classifier. This should take a list of word indices as input, embed them through an embedding layer, pass them through a deep averaging network, and return a tensor representing a log-probability distribution over class labels. Details of the deep averaging network are described in the code comments.

2. **`NeuralSentimentClassifier`**: Implement the constructor and `predict`. The constructor should take advantage of the provided functions `read_word_embeddings` and `WordEmbeddings.get_initialized_embedding_layer` from sentiment_data.py to load pretrained embeddings and create a `NeuralNet` object. `predict` should map a list of word indices to a (0/1) prediction from a trained model by calling `forward` on the trained `NeuralNet`.

3. **`train_deep_averaging_network`**: This function creates a `NeuralSentimentClassifier` and implements the full training loop for the neural net it contains. The number of epochs should be `args.num_epochs` with a batch size of `args.batch_size` and learning rate `args.lr`.

   For each batch, compute classification loss based on the prediction using 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). 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. Return the created `NeuralSentimentClassifier.`

Ensure your model trains successfully, and report the accuracy of your model on the train set and development set for the final epoch in README.md.