Goals. The main purpose of this assignment is to familiarize you with the practical details of implementing, training, and evaluating language models. It is also meant to acquaint you with some of the challenges of working with relatively large datasets, as is common today. You will implement an n-gram language model with smoothing, as well as an RNN and (optionally) LSTM language model. At the end, you will answer a few conceptual questions about embeddings.

Resources

Dependencies. As in the previous assignment, please use Python 3.8 or higher. This assignment allows the following Python packages: `torch`, `torchvision`, `numpy`, `nltk`, `spacy`. Any non-standard packages not in this list might cause the autograder to crash.

You should already have installed PyTorch from doing the previous assignment; if so, you can reuse your conda environment from HW0. If not, follow these instructions: `https://pytorch.org/get-started/locally/`. GPUs are recommended but not required for this assignment, so don't worry too much about CUDA support. If you'd like to use GPUs, we recommend Google Colab. Some helpful Google Colab guides are linked in the footnotes below.[1][2]

As before, I highly recommend using conda to manage Python dependencies.

Data. The data and code can be found at this Google Drive link.[3] For this assignment, we'll be using a snippet of C4 (Colossal Cleaned Common Crawl), a text dataset consisting of documents that were scraped from the internet and then filtered and cleaned for quality purposes. We'll be training language models of varying types on these datasets, and then comparing their performance.

As in the previous assignment, the data that you have access to has been split into a training set, development set, and test set.

Code structure. You will see several Python scripts in the provided repository. The main files you'll be modifying are as follows:

- `tokenizer.py`: Implements a basic word-level tokenizer. You will be implementing a BPE tokenizer here for Q1.
- `lm.py`: Implements the core functionality of n-gram and RNN-based language models. This is the main script you will be working in from Q3 onward.

General tips. Many of these functions will take more than a few minutes to run. For example, training the `BPETokenizer` with 5,000 merges takes about 10–15 minutes on my machine, and training the RNN takes about 5–10 minutes. To make things faster during debugging, you can subsample the training data to just a few examples, and then go back to working with the full dataset at the very end when you're getting your final scores.

Assignment

Download the code and data, and unzip the file. Change into the `hw1/code/` directory.

[1] Google Colab guide
[2] Google Colab cheat sheet
[3] `https://drive.google.com/file/d/1YzABlZGC8lM__q0OABIQSWs9nBCUV75R/view?usp=sharing`

Task 1: Subword Tokenization and N-gram Language Modeling (22 points)

Your first task will be to implement a subword tokenizer, and then to build an n-gram language model over the subword vocabulary.

We will use byte-pair encoding (BPE) as a way to reduce the size of our vocabulary while also yielding fewer unknown tokens. The skeleton of this tokenizer has been provided. Extend the `BPETokenizer` class to split each word into character tokens, and then iteratively merge the most frequent token pairs into new tokens that get added to the vocabulary.

You will then implement a trigram (3-gram) language model with Laplace smoothing. You will write n-gram language model training code. This involves computing the conditional probability of each trigram given its bigram prefix. As we are using trigrams, this involves counting the number of instances of each triplet of tokens. Recall that we generally want to compute $p(x_i|x_{1:i-1})$. Computing this in exact form is intractible, so we instead make a Markov assumption and only look at the previous tokens within the n-gram:

$$p(x_i|x_{1:i-1}) \approx p(x_i|x_{i-n+1:i-1}) \tag{1}$$

where $p(x_i|x_{i-n+1:i-1}) = \frac{C(x_{i-n+1:i-1}x_n)}{C(x_{i-n+1:i-1})}$; in words, this is the count of the trigram divided by the count of its bigram prefix.

With Laplace smoothing, we add a constant offset k to the numerator, and add the vocab size (weighted by the offset) $k \cdot |V|$ to the denominator.

Q1 (8 points). Complete the `BPETokenizer` class. Once you've implemented it, set the vocab size to a value close to 5,000 (the default), and try training it on the provided dataset. Training will probably take a while—this took about 10–15 minutes on my machine.[4]

Q2 (2 points). Using a vocab size of 5,000 or so, try inspecting your BPE tokenizer's vocab or manually tokenizing a few words using `tokenizer.tokenize` (and optionally using `tokenizer.inverse_vocab` so you can understand what these tokens are). Give at least 3 examples of words that are split into subwords or characters by the tokenizer, and at least 3 examples of words that are not split by the tokenizer. In 1–3 sentences, what trends do you notice about the kinds of words that get split or not?

After this, for the sake of efficiency, you can reduce the vocab size to a low number like 1,000 for the rest of the assignment.

Q3 (7 points; AUTOGRADED). Implement the n-gram language model by extending `NGramLM` in `lm.py`. This is a trigram language model, so you will need to store the counts of all trigrams and bigrams, and use these to compute the conditional probabilities of all trigrams following the formula above. Note that there is a smoothing term k; use this to implement Laplace smoothing when computing your probabilities such that unseen n-grams have nonzero probabilities.

You cannot compare perplexities across tokenizers; this metric is only comparable across models that use the exact same token vocabulary. Thus, we will not ask you to tune your vocab size to optimize perplexity. Instead, to receive full credit, **your n-gram model should achieve a dev perplexity of less than 15 if you use a vocab size of about 1,000, or 0 merges** (i.e., a character-level tokenizer). **Your model should train in less than 5 minutes**, not including the time it takes to train the tokenizer. Our implementation obtained a perplexity of 8.49 on the dev set, and finished training in 40.43 seconds without a GPU.[5] If you use a vocab size of 5,000, you will probably get a perplexity in the hundreds.

[4] If you cannot get this working, you can use the provided `WordTokenizer` to do Q3 onward.

[5] When using the WordTokenizer instead of the BPETokenizer, we obtained a dev perplexity of 60.7 and finished training in 28.60 seconds without a GPU. We needed to reduce the vocab size to 1000 to get this perplexity.

Of course, as-is, we can't yet measure perplexity, because we don't have an implementation of it! This is what you will implement next.

Q4 (5 points). Now, you will implement the evaluation metric we'll be using to compare language models: **perplexity**. Recall that the perplexity of a model M given a dataset $\mathcal{D}$ consisting of tokens $x_i \in \mathcal{D}$ is defined as follows:

$$\text{ppl}(M, \mathcal{D}) = \exp\Big(-\frac{1}{|D|} \sum_{i=1}^{|D|} \log p_M(x_i | x_{1:i-1}) \Big), \tag{2}$$

where M is your language model, p_M is the probability of an n-gram according to your model, D is the evaluation dataset, and $|D|$ is the number of tokens in the evaluation dataset.

Extend the `perplexity` method in `NGramLM` (`lm.py`) to implement this metric. Once you've implemented it, evaluate your n-gram language model on the development set, and report its perplexity in your written report (along with your vocab size). Please feel free to use outside libraries to double-check your implementation, but your implementation in the `perplexity` function must be written from scratch, using only the standard Python libraries (the `math` library may be helpful here).

Task 2: Implementing a Neural Language Model (22 points)

Now, you will implement a far better language model based on a recurrent neural network (RNN). This will be a very basic version of an RNN with no attention nor any kind of gating mechanisms.

You will be given one method here: `get_next_token_prob`. It takes a context and returns a log-probability distribution over the next tokens given the context. The log-probability distribution is a vector of length equal to the vocabulary size.

Q5 (20 points; AUTOGRADED). Implement an RNN language model by extending the provided `RNNLM` class in `lm.py`. Once you've done this, measure the perplexity of your model on the dev set. (The perplexity function has been provided for you.) Report this perplexity in your written report, as well as the vocab size you used to get it.

Assuming a character-level tokenizer with a vocab size of 1,000, your final model should **achieve a perplexity of less than 12, and train in less than about 15 minutes**, not including tokenizer training time.[6] Our implementation obtained a perplexity of 6.25 in 6.5 minutes of training without a GPU. This is a very unoptimized implementation with no batching, and you can probably train even faster by using a GPU and/or by implementing batching.

Debugging tips. Note that your model will be evaluated on perplexity and likelihoods, rather than correctness of predictions. As such, **you must implement a language model.** This means that the model must return a log-probability distribution over tokens given the prior context. Be sure to check that your language model's output is, in fact, a log-probability distribution over all tokens in the vocabulary.

If you run into errors, it's common to use print statements in your code to figure out where the issue is. A very helpful and common practice is printing the shapes of all your tensors using `.shape` to make sure everything matches what you expect.

For faster debugging, we recommend making sure that your model can overfit a very small training set quickly. You could construct a smaller debugging training set by subsampling the provided training set. To

[6] We will use a much smaller training dataset on the Gradescope backend than you have used. Treat these targets as approximate heuristics for how good and efficient your model should be to pass the autograder, rather than exact targets.

make sure things are training properly, you can add some code to inspect the training loss at the end of each epoch; it should go down after every epoch. Tune your learning rate carefully.

Consider using small embedding sizes and hidden sizes so that your model trains quickly and doesn't take much memory. For this dataset, you should be able to get away with surprisingly small embedding sizes and small hidden sizes (e.g., close to 100).

If you have access to a GPU, you can implement batching to make training much faster. You can do this by creating a new dimension in your tensors; this makes the tensor of size B $\times$ L $\times$ S, where B is the number of examples in your batch, L is the number of tokens of each sequence of the batch, and S is the size of the hidden representations of the network. The rest of your code shouldn't need to change much to accomodate this addition. Note that you do not need to do this to complete the assignment, but it may be helpful for debugging to be able to run your code more quickly.

Q6 (2 points). Now compare your basic n-gram model to your RNN. *Be sure to use the exact same vocabulary, or the perplexities will not be comparable.* How do the perplexities of these models compare? How about training times?

In 1–3 sentences, which of the models you trained do you think would generalize best to novel data in a very different domain? In other words, which model would probably get a lower perplexity on test data from a very different source? Why do you think so?

Task 3: Understanding Word Vectors and Embeddings (24 points)

There is no coding in this part of the assignment.

Q7. (4 points) Suppose we have the following token embeddings:

```
the: [0.8, 0.2, 0.1, 0.0]
a:   [0.8, 0.2, 0.4, 0.0]
cat: [0.0, 0.5, 0.8, 0.9]
dog: [0.0, 0.4, 1.1, 0.7]
```

Which other token in this vocabulary has the highest cosine similarity with `dog`? The token cannot be the `dog` token itself. Provide the cosine similarity you computed, and state which token has the highest cosine similarity.

Q8. Consider co-occurrence–based word vectors. Here, every word vector is a vector of size $|V|$, where V is the vocabulary size. A word w's vector representation is the vector containing its co-occurrence counts with all words in the vocabulary given some dataset.

a. (2 points) Take this dataset:

```
feed the cat
feed the dog
the dog eats
```

Using a context window size of ± 1 word (one word to the left and one word to the right), construct the co-occurrence word vector for "the". Use the following indices in your vector: 0 for "feed", 1 for "the", 2 for "cat", 3 for "dog", 4 for "eats".

b. (5 points) Take this table of co-occurrence counts:

```
Paris  | city x3, government x2, French x10
France | French x10
Italy  | Italian x10
Rome   | city x3, government x2, Italian x10
```

A common word vector analogy involves computing the capital of a new country via, for example, the formula $\mathbf{v}_{\text{Paris}} - \mathbf{v}_{\text{France}} + \mathbf{v}_{\text{Italy}} = \mathbf{v}_{\text{Rome}}$. Apply the vector creation method from Q8(a) to this co-occurrence table to get embeddings for "Paris", "France", "Italy", and "Rome". (Feel free to only include indices for "city", "government", "French", and "Italian" in these embeddings, and to assume that the other co-occurrences are all 0.) Then, show that the mentioned vector analogy holds by applying the mentioned formula to these embeddings.

Q9. The next few subquestions will focus on skip-gram embeddings. The skip-gram model computes the probability $p(+|w, c)$ that c is a real context word for w:

$$p(+|w, c) = \sigma(\mathbf{w} \cdot \mathbf{c}) \tag{3}$$

where w and c are the word and context, and $\mathbf{w}$ and $\mathbf{c}$ are their vector representations, each of dimensionality d. Each vocabulary entry has independent vectors for the word and context vectors, so each entry really has two vectors associated with it.

Assume a context window size of ± 1 words. We are using skip-grams with negative sampling (SGNS) with $k = 2$ negative samples per positive word.

Consider the following dataset:

```
Paris France
Rome Italy
in France
in Italy
```

a. (3 points) Write down all the positive training examples (w, c_+) in this dataset.

b. (2 points) For the positive example (w=France, c_+=Paris), write down two possible negative examples that could be sampled.

c. (8 points) Suppose the dimensionality of the word and context vectors is $d = 2$. Provide word $\mathbf{w}$ and context $\mathbf{c}$ vectors for all vocabulary items that would assign high likelihood to the dataset. As a reminder, the log-likelihood of the training data is $\log \sigma(\mathbf{w} \cdot \mathbf{c}_+) + \sum_{i=1}^{k} \log \sigma(-\mathbf{w} \cdot \mathbf{c}_-)$. Your vectors should satisfy these conditions:

1. $p(+|w, c_+) > 0.9$ for all positive pairs
2. $p(-|w, c_-) < 0.1$ for all negative pairs

Extra Credit

Implementing an LSTM (22 points)

If you choose, you can implement a far better version of the RNN language model call the Long Short-term Memory (LSTM) network. The LSTM was the state of the art in language modeling for many years before the Transformer came about, and is still used by many computational linguists to model human language processing.

Implementing the LSTM will involve extending the RNN class you wrote. Given the context vector from the previous position $\mathbf{c}_{i-1}$ and input representation $\mathbf{x}_i$, the LSTM adds the following gates to each layer:

Forget gate:

$$\mathbf{f}_i = \sigma(U_{\text{forget}}\mathbf{h}_{i-1} + W_{\text{forget}}\mathbf{x}_i) \tag{4}$$

$$\mathbf{k}_i = \mathbf{c}_{i-1} \odot \mathbf{f}_i, \tag{5}$$

where $\odot$ is the Hadamard product (i.e., an elementwise multiplication), and σ is the sigmoid function.

Input gate:

$$\mathbf{g}_i = \tanh(U_{\text{g}}\mathbf{h}_{i-1} + W_{\text{g}}\mathbf{x}_i) \tag{6}$$

$$\mathbf{i}_i = \sigma(U_{\text{input}}\mathbf{h}_{i-1} + W_{\text{input}}\mathbf{x}_i) \tag{7}$$

$$\mathbf{j}_i = \mathbf{g}_i \odot \mathbf{i}_i \tag{8}$$

We use the add gate to derive the updated context vector:

$$\mathbf{c}_i = \mathbf{j}_i + \mathbf{k}_i \tag{9}$$

Finally, we have the output gate:

$$\mathbf{o}_i = \sigma(U_{\text{output}}\mathbf{h}_{i-1} + W_{\text{output}}\mathbf{x}_i) \tag{10}$$

$$\mathbf{h}_i = \mathbf{o}_i \odot \tanh(\mathbf{c}_i) \tag{11}$$

E1 (20 points). Implement the LSTM. You will extend the `LSTMLM` class to do this. Once you've implemented it, measure the perplexity of your language model on the dev data. Report the final dev perplexity in your written report, alongside the vocab size you used to get it.

E2 (2 points). Compare the dev perplexity and training time of your LSTM LM to your RNN LM. *Again, be sure to use the exact same vocabulary during this comparison!* Which trains faster, and which performs better? In 1–2 sentences, why do you think so?

Submission

Upload the following two files to Gradescope separately:

1. Upload your written responses as a PDF with numbered responses to **HW1: Language Modeling [Written]** on Gradescope.
2. Upload a .zip file containing your `lm.py` and `tokenizer.py` scripts to **HW1: Language Modeling [Code]** on Gradescope.

Sanity-check your implementations by running the following commands before you upload:

```
python lm.py --model NGRAM
python lm.py --model RNN
```

If your submission runs on the autograder, but fails on the perplexity threshold, try setting the vocab size to 1,000.

If you ended up using the `WordTokenizer`, your models may not pass the autograder. This is okay; we will take off points for Q1, but we will assess whether what your language model implementations are acceptable and reallocate points appropriately.