Homework 2 Instructions

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

General Instructions

1. *Complete the provided Jupyter notebooks for each part (three in total). Each part is 30 points.*
2. *You can use material from the labs as a guide for this homework.*
3. *Cite any other open-source code which helps you to complete this assignment.*
4. *You can add written answers and plots directly to the notebook files.*
5. *As always, you can check CampusWire for any clarifications or questions.*
6. *Good luck! The assignment is due **November 4, 11:59 am (before the start of class)**.*
7. *We are giving you more time for the assignment for a reason: parts 2 and 3 require significant training time. **Please start early!***

Part 1: BoW-Based Natural Language Inference (30 pts)

For question 1, you will train a Bag-of-Words encoder to tackle the Stanford Natural Language Inference (SNLI) task. Because the SNLI data set is relatively large, we recommend starting the assignment early (training may take hours). Likewise, because you are training on only a subset of the full data and using a relatively simple BoW encoder, you should not expect to get accuracies comparable to the published leaderboard.

Dataset (6 pts)

The SNLI task poses the following problem: Given two pieces of text, a premise, and a hypothesis, you (or a model) have to choose one of the following:

1. The premise **entails** the hypothesis.
2. The premise **contradicts** the hypothesis.
3. The premise neither entails nor contradicts the hypothesis and is thus **neutral** to it.

For example:

1. *Premise:* A man inspects the uniform of a figure.
   Hypothesis: The man is sleeping.
   This is a **contradiction** since the man cannot be sleeping while inspecting the uniform.

2. *Premise*: A soccer game with multiple males playing.
   Hypothesis: Some men are playing a sport.

This is **entailment** because the hypothesis is true whenever the premise is true.

3. *Premise*: An older and younger man smiling.
   Hypothesis: Two men are swimming and laughing at the cats playing on the floor.
   This is **neutral** because the premise does not contain any information about cats.

Being based on natural text, there will be some ambiguity regarding the answers. Nevertheless, the NLI-style tasks have shown to be effective for training models to capture aspects of semantic information from text. Refer to `https://nlp.stanford.edu/projects/snli/`, or the original paper (`https://nlp.stanford.edu/pubs/snli_paper.pdf`) for more details. You will be provided with tokenized training and validation pickle files.

Model (8 pts)

At its core, the SNLI is a 3-class classification problem, where the input is two separate sentences (strings of text). Choosing what approach to use to integrate information from both sentences is where the problem becomes interesting from a modeling perspective, and a variety of approaches have been proposed. In our case, we will take the following simple approach:

1. We will use a bag-of-words (BoW) encoder to map each string of text to a fixed-dimensional vector representation. In other words, the representation of a sentence will be the average of the embedding vectors for the words in the sentence.[1]
   Note: HW1 and Lab 2 may be good references for this part.
2. We will then combine the vector representations. There are several ways to do this (sum, element-wise product, concatenation). You should try two of these methods.
3. Once we combine the vectors, we then apply a linear projection and softmax to produce a distribution over the three possible classes (`entailment`, `contradiction`, `neutral`).

Note: Unlike for HW1, we will require that you write a batched (i.e., vectorized) implementation of your neural network class. In other words, your network should be able to take in tensors representing multiple examples, where the first dimension is the batch size.

Training and Validation (16 pts)

Your task is to implement, perform hyperparameter tuning, and analyze the results of the model. Perform tuning over **one** of the hyperparameters, e.g.,

1. Embedding dimension
2. The optimizer itself (SGD vs. Adam)
3. Learning rate

Varying four different values for the hyperparameter, report the training and validation losses and accuracies (in plotted curves), as well as the number of trained parameters in each model. Take your best

[1] The `vocab_size` is the number of normal tokens, i.e., it does not count special tokens like `<pad>` or `<unk>`.

model based on validation performance and highlight 3 correct and 3 incorrect predictions in the validation set. Hypothesize why the model might have gotten the 3 incorrect predictions wrong.

Note: Make sure to choose hyperparameter values that are different enough to produce different loss curves. Exactly what this means depends on the hyperparameter, but it may be useful to make sure your options span several orders of magnitude for numerical hyperparameters.

Hand in: Training and validation loss curves, for two different combination methods and 4 different hyperparameter choices (8 training runs total). 3 correct and 3 incorrect predictions from your best model.

Part 2: Neural Language Modeling (30 pts)

LSTMs (10 pts)

In this part, you will train a neural recurrent language model on **Wikitext-2**. Lab 4 may be a useful reference. However, note that you will be using an LSTM (`nn.LSTM`) instead of a vanilla RNN (`nn.RNN`). There are several architectural hyperparameters that can have an effect on performance, including the hidden size and number of layers. In order to tune these hyperparameters, you should implement the following:

1. In the training code, implement validation-based early stopping, such that, if the validation performance does not improve for *m* epochs, terminate the training process. *m* is a hyperparameter called "max patience". Make sure that, **after early stopping, you have saved the model with the best validation performance, not the model at the last epoch**. (5 pts)

2. Choose **one** hyperparameter to vary. Pot training epochs vs. train loss, with 4 curves corresponding to different choices of the hyperparameter. Do the same for validation loss. (5 pts)

   Note: Training may be time-consuming, so we suggest you train your first model and use that to develop the rest of Q2 before coming back to tuning the hyperparameters. You should then rerun 2.2 and 2.3 using your best model by validation perplexity (this should be quick).

   Note: Make sure to choose hyperparameter values that are different enough to produce different loss curves. Exactly what this means depends on the hyperparameter, but it may be useful to make sure your options span several orders of magnitude for numerical hyperparameters.

Hand in: Training and validation loss curves (2 plots total, with 4 curves each).

Learned Embeddings (6 pts)

One way to understand the representations a neural network is learning is to analyze the geometry of its embedding space. In this part, you will analyze the embeddings learned in the final layer of your best model (by validation perplexity).

1. Choose a set of 4 works, e.g., {run, dog, where, quick}. For each word, find the 10 closest words and 10 furthest words in the trained LSTM model's final projection layer according to cosine similarity. You will need to implement your own cosine similarity function. (3 pts)
2. Use UMAP to visualize the model's projection layer. In particular, run UMAP on the entire `nn.Linear` weight matrix, and plot the points corresponding to the 100 words (5 ✕ (10 +10)) from the previous part. (3 pts)

Sampling (14 pts)

In this section, you will implement sampling for your recurrent language model. Here is pseudo-code for this procedure:

$h_0 \leftarrow \vec{0}$;
$x_0 \leftarrow$ <bos>;
while $x_t \neq$ <eos> **do**
  $h_t = \textbf{LSTM}(h_{t-1}, x_t)$;
  $p = \textbf{softmax}(\textbf{projection}(h_t))$;
  $x_{t+1} \sim \textbf{multinomial}(p)$;
end
Result: sampled sequence (<bos>, $x_1, x_2, x_3, ..., x_T$, <eos>)

1. Implement the sampling algorithm described here. (4 pts)

2. Sample 1,000 sequences from the best model you trained above and calculate their average log probability. Recall the following identities:

$$\log p(x_1, ..., x_T) = \log \prod_{t=1}^{T} p(x_t | x_{<t}) = \sum_{t=1}^{T} \log p(x_t | x_{<t}).$$

3. Compare the sampled sequences against 1,000 sequences from the validation set in terms of the number of unique tokens and length. (3 pts)
4. Choose 3 sampled sequences and discuss their characteristics. Does anything suggest these sequences are machine-generated rather than human-generated? Do these sequences stay on topic? Are they grammatically correct? (3 pts)

Hand in: Comparison of generated sequences and validation set, 3 selected generated sequences.

Part 3: Neural Machine Translation (30 pts)

Transformer Encoder (18 pts)

The goal of this part is to plug a transformer encoder into end-to-end neural machine translation instead of an RNN. You are not required to outperform the RNN-based baseline from Lab 5.

1. Build the encoder using the following steps:
    a. **Positional embeddings** should be added to the input embeddings. Please use the sinusoidal encoding as mentioned in the transformer paper.
    b. **Transformer encoder**: use the official PyTorch implementation of a transformer encoder.
    c. **Hyperparameters**: use the same hyperparameters as Lab 5: embedding and hidden size 512, 1 layer in the encoder with 2 attention heads.
2. Pair the full transformer encoder from step (1) with the following decoders from Lab 5 and train for 20 epochs:
    a. Basic RNN decoder (no attention)
    b. RNN decoder with encoder attention
    c. **(Optional, Extra credit)** RNN decoder with encoder attention and self-attention

Each architecture should take approximately 2 hours to train on Google Colab with a single GPU.

Hand in: Training and validation curves for both loss and BLEU.

Transformer Encoder (12 pts)

In this part, you will analyze the attention mechanisms in your model.

- Check the model code and find how attention weights are logged. Make sure to understand how each type of attention works.
- Implement a function that takes in the attention weights and plots the corresponding heat map.
- On the axes of the attention map, show the actual tokes from the sequences rather than numerical indices.

Hand in:

1. Select 3 examples from the validation set and print out the input sentence in the source language and the predicted sentence in the target language for each type of encoder.
2. For 3.2.B, print out the cross-attention heatmap.
3. For 3.2.C, print out the cross-attention heatmap and the decoder's self-attention heatmap.