705.641.81: Natural Language Processing Self-Supervised Models

Homework 2: Language modeling + count-based LMs  
+ preliminaries to neural networks

For homework deadline and collaboration policy, please see our Canvas page.  
Name: \_\_Madihah Shaik\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
Collaborators, if any: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Sources used for your homework, if any: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Homework goals:** After completing this homework, you should be able to be comfortable with various foundational concepts related to our course. Specifically, we hope you will gain the followings:

* Get more comfortable with linear algebra, specifically gradients and Jacobians!
* Gain a better grasp of count-based language models!
* Get comfortable with activation functions such as the Softmax function!
* Know how to develop your own classifier!

# 

# Concepts, intuitions and big picture

1. The -gram models we’ve shown make the -th order Markov assumption, i.e. that the distribution of words depends only on the previous words. What properties of language will it not capture? Discuss (briefly) several distinct ways in which this assumption is false for natural language.  
   A lot of words in natural language depend on other words that are farther away from them, rather than in the same window of n-1 previous words. This sliding window doesn’t always capture the full context to give information about the meaning of the word. Many clauses and phrases extend longer than the previous n-1 words.
2. Follow-up to previous question: Despite the Markov assumption, -gram models are remarkably good at predicting the next word. Discuss why this might be. What information is in the previous word(s) that makes these models perform so surprisingly well? In particular, what kinds of grammatical information do they capture?  
   Predicting the next word is done using the statistical probability of the training data. As the models are fed more training data, implicit grammar rules are learned. In phrases and sentences, many grammatical patterns repeat, and the order of parts of speech can be learned very well by these models.
3. Explain how are perplexity and cross-entropy loss related?  
   Cross-entropy loss is a measure of how close the model’s probability distribution is to the test data. Perplexity is a measure of how the model’s probability distribution matches the test data. Thus, perplexity uses the cross-entropy loss of each prediction to give a measure of the model’s average. Perplexity is the exponential of the average cross-entropy loss.
4. For a vocabulary of words, what would you expect perplexity to be if your model predictions were completely random? Compute the corresponding cross-entropy loss for and , and keep this in mind as a baseline.  
   If the model’s predictions are random, every word in the vocabulary has an equal chance of being selected: 1/V. The cross-entropy loss is . The perplexity would be . Cross-entropy loss for |V| = 2000 = log(2000) = 7.601. |V| = 10000 cross entropy loss = log(10000) = 9.21
5. Which of these are reasons for the recent wave of neural networks taking off? (check the options that apply.)  
    We have access to a lot more computational power.  
    Neural Networks are a brand new field.  
    We have access to a lot more data.  
    There has been significant improvements in benchmarks in speech recognition and NLP.
6. What does a neuron compute?  
    A neuron computes an activation function followed by a linear function ()  
    A neuron computes a linear function () followed by an activation function.  
    A neuron computes a function that scales the input linearly ().  
    A neuron computes the mean of all features before applying the output to an activation function.
7. What is the point of applying a Softmax function to the logits output by a sequence classification model?  
    It softens the logits so that they’re more reliable.  
    It applies a lower and upper bound so that they’re understandable.  
    The total sum of the output is then 1, resulting in a possible probabilistic interpretation.

# Linear Algebra Recap

## Gradients

Consider the following scalar-valued function:

1. Compute partial derivatives with respect to , and .
2. We can consider to be a function that takes a vector as input, where . Write the gradient as a vector and evaluate it at .

## Gradients of vectors

Let and . For the following parts, before taking any derivatives, identify what the derivative looks like (is it a scalar, vector, or matrix?) and how we calculate each term in the derivative. Then carefully solve for an arbitrary entry of the derivative, then stack/arrange all of them to get the final result.

* Show that .

+

* Show that .

* Show that .
* Show that .

#### Note:

The above equations follow the [numerator layout](https://en.wikipedia.org/wiki/Matrix_calculus#Layout_conventions) notation which is commonly following in linear algebra, but there are alternative notational conventions as well.

## Jacobian

Consider the following vector function from to :

1. What is the Jacobian matrix[[1]](#footnote-1) of ?
2. Evaluate the Jacobian matrix of at .

# -gram Language Models

## A toy -gram language model

Consider the following vocabulary, where BOS is the dummy token indicating the beginning of a sentence, and EOS indicates the end of a sentence. Consider the following training data:

BOS who EOS  
BOS who framed roger EOS  
BOS roger rabbit EOS  
BOS who framed roger rabbit EOS  
BOS roger framed who EOS  
BOS who framed the rabbit EOS

1. Compute the following probabilities: , , .
2. Briefly explain what the sparsity problem is in -gram language models?  
   When looking at vocabulary for a model, the n-gram is a sequence of n words from the vocabulary. In the roger rabbit example and with n=2, there are many possible n-grams from the vocabulary that do not exist in the training data, for example BOS EOS or rabbit roger. As n increases, the total number of possible n-grams also increases. If we were to count the existence of every n-gram in the training data, we would have a lot of 0s. This is the sparsity problem; the matrix of computed probabilities would be filled with mostly 0s.

# Challenges of linear classifiers of sentences

A simple way to compute a representation for a phrase is to add up the representations of the words in that phrase: , where are the word in and is the embedding for word .

1. Now, consider sentiment analysis on a phrase in which the predicted sentiments are

* for some choice of parameters . Note that here and “” is the “dot-product”. Prove that in such a model, the following inequality cannot hold for any choice of and word embeddings:
* Thereby, showing the inadequacy of this model in capturing negations.[[2]](#footnote-2)

1. **Extra Credit:** Construct another example of a pair of inequalities similar to the ones above that cannot both hold.
2. **Extra Credit:** Consider a slight modification to the previous predictive model:

* where ReLU (rectified linear function) is defined as:
* Given this choice of predictive function, show that it is possible to satisfy the above inequalities for some choice of . **Hint:** Show there exists parameters and word embeddings , and that the inequalities are satisfied.

1. Given the above result, explain (in 1-2 sentence) why the use of neural networks (which have more complexity than linear models)  
   Neural networks are able to capture non-linear relationships between words. In this example, a non-linear model will learn the difference between “not bad” and “not good” that is more complex than the linear explanation of (not+bad=not bad and not+good=not good). The ‘not’ word will be shown to negate the corresponding word, and the sentiment then depends on context, which is not captured linearly.

# Softmax function

Remember the Softmax functions from the class:

1. Prove that Softmax is invariant to constant offsets in the input, i.e., for any input vector  and any constant ,

* **Pro tip:** We make use of this property in practice to increase the numerical stability of our models. Specifically, using , i.e. subtracting its maximum element from all elements of would prevent numerical instability due to large values.

1. Softmax maintains the relative order of the elements in . In particular, show that the largest index is intact after applying Softmax:

1. Define the Sigmoid function as follows:

* Prove that Softmax is equivalent to the Sigmoid functions when the number of possible labels is two: . Specifically:

1. **Extra Credit:** Next, let’s extend (1) to prove the following inequality:

* **Hint:** Use the following inequality where each .

# Programming

In this programming homework, we will

* build a simple count-based (n-gram) language model.
* implement your own gradient descent optimization for the classifier you built in homework 1.

#### Skeleton Code and Structure:

The code base for this homework can be found at [this GitHub repo](https://github.com/JHU-CLSP/CS-601-471-671-Sp24-Public/) under the hw2 directory. Your task is to fill in the missing parts in the skeleton code, following the requirements, guidance, and tips provided in this pdf and the comments in the corresponding .py files. The code base has the following structure:

* ngram\_lm.py implements a n-gram language model on a subset of Wikipedia.
* gradient\_descent.py reuse the sentiment classifier on movie reviews you implemented in homework 1, with additional requirements to implement manual softmax, cross-entropy loss, and gradient updates.
* main.py provides the entry point to run your implementations in both ngram\_lm.py and backprop.py.
* hw2.md provides instructions on how to setup the environment and run each part of the homework in main.py

**TODOs** — Your tasks include 1) generate plots and/or write short answers based on the results of running the code; 2) fill in the blanks in the skeleton to complete the code. We will explicitly mark these plotting, written answer, and filling-in-the-blank tasks as **TODOs** in the following descriptions, as well as a **# TODO** at the corresponding blank in the code.  
**TODOs** (Copy from your HW1). We are reusing most of the model.py from homework 1 as the starting point for the gradient.py - you will see in the skeleton that they look very similar. Moreover, in order to make the skeleton complete, for all the **# TODO (Copy from your HW1)**, please fill in the blank below them by copying and pasting the corresponding implementations you wrote for homework 1 (i.e. the corresponding **# TODO** in homework 1.)

#### Submission:

Your submission should contain two parts: 1) plots and short answers under the corresponding questions below; and 2) your completion of the skeleton code base, in a .zip file

## Count-based LMs

In the lecture, we have learned language modeling as building probabilistic distribution (marginal and joint) over language. Moreover, we can estimate such distributions (e.g. ) by *counting*. For example

We are going to implement such a count-based LM, i.e. n-gram language model.

### Data Loading

We will use (a subset of) the [WikiText](https://huggingface.co/datasets/wikitext) dataset for building the n-gram LM. The WikiText language modeling dataset is a collection of over 100 million tokens extracted from the set of verified Good and Featured articles on Wikipedia. Spend a few minutes reading a few examples on Huggingface to get a better sense of what this dataset looks like. We will use Huggingface’s datasets library to download this dataset locally. For efficiency, we only use the first training samples to create our LM.

Read the load\_data function in ngram\_lm.py for how we obtain the data.

### Preprocessing: Sentence Split and Tokenization

As you have seen, each data sample consists of a Wikipedia paragraph. To perform our n-gram counting, we need first split each paragraph into sentences [[3]](#footnote-3) and then each sentence into a sequence of tokens (i.e. *tokenization* that we introduced in homework 1). In particular, we will use [an algorithm](https://aclanthology.org/W07-0733/) by our very own Philipp Koehn for sentence splitting; and use a *sub-word tokenizer* for tokenization. For now, we are using an existing sub-word tokenizer of an existing model from the Huggingface library, but in the future, we will also build our own tokenizer.

A *Sub-word tokenizer* which, as it should be clear from the name, splits each sentence into units smaller than words, based on their frequency. For example, the word "Potentials" is broken into four sub-words: ‘Po’, ’##ten’, ’##tial’, ’##s’‘, where ‘##‘ is a special symbol indicating that the sub-word is in the middle of a word.

Sub-word tokenization might initially seem like a bad idea since we are breaking up the word. But it brings it several benefits.

* Handling Out-of-Vocabulary (OOV) words: Sub-word tokenization allows the representation of rare or unseen words as a combination of sub-words or tokens that have been seen in the training data, thus reducing the OOV problem.
* Improved vocabulary size: It allows for a more compact vocabulary size, reducing the size of the language model, and increasing its efficiency.
* Better language model performance: Sub-word tokenization results in a better representation of morphologically rich languages, where words are formed by combining roots and affixes, leading to improved language model performance.
* Cross-lingual compatibility: Sub-word tokenization is language-agnostic, and models trained on sub-word tokens from one language can be applied to other languages, improving cross-lingual compatibility.

How are sub-tokenizers built? We will delve into that in a few weeks! For now, we will just use them!  
  
Read the sentence\_split\_and\_tokenize\_demo for how exactly an wikitext paragraph is converted into sentences and then tokens.

### Build the N-gram LM: Let’s Count!

**TODOs**: read the create\_ngrams function which iteratively processes each paragraph and counts the n-gram statistics, and completes the following lines under three **# TODO**s:

* tokenize the words in the sentence: that takes in a sentence and convert it into tokens as a list of tokens with the tokenizer.  
  **Hint**: check out how to do tokenization in sentence\_split\_and\_tokenize\_demo
* count n-gram statistics over each list of tokens, record them in
  + ngrams: the Counter for n-grams count
  + ngram\_context: the Counter for context (i.e. (n-1)-grams) count
  + next\_word\_candidates: the Dictionary that keeps track of all possible candidate next tokens given the context, i.e. all tokens that, concatenated with the (n-1)-gram context, constitutes an n-gram that exists in the data.
* for all the Counter and Dictionary, use tuple of n-gram or (n-1)-gram tokens as the keys.  
  **Hint**: also check the corresponding comments in the code for specific requirements
* computes the estimated probability of the next word (token) given the context.  
  **Hint**: use the formula in [6.1](#subsubsec:count-based lms), and also the corresponding comments in the code

Finally, the next\_word\_pred returned from the create\_ngrams, will record for each (n-1) gram context, the most probable next word continuation. For this homework, we will set so that to build a *trigram* language model.

### Visualize the N-gram distribution

**TODOs**: Complete the plot\_next\_word\_prob function in ngram\_lm.py, and run run\_ngram in main.py to plot the top-10 most probable next token given an input context. Paste the two plots for the two given contexts (provided in run\_ngram), and describe in 2-3 sentences your findings.  
**Hints**:

* you can check out [this tutorial](https://www.geeksforgeeks.org/bar-plot-in-matplotlib/) about how to plot bar chart with [matplotlib](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.bar.html).
* read the comments in the code carefully for more info about the input.

### Generation: Sample from the LM

Now that we have built our LM, given any prefix[[4]](#footnote-4), we can sample from the LM to generate a full completion. Specifically, at each timestamp, we take the last (n-1) tokens from the prefix as the context and query our LM to get the most probable next token, append it to the prefix, and continue until we sample the stop token our the maximum length is reached.

This sampling procedure is called greedy decoding, as we take the most probable next token each step, there are a few other sampling strategies like beam search , top-k, and top-p sampling . Similarly, we will get to them soon.  
  
Read the generate\_text function in ngram\_lm.py for details of the generation process described above.  
  
**TODOs**: Run run\_ngram in main.py and paste the completion of the two given prefixes (provided in run\_ngram), and describe in 2-3 sentences your findings.

## Optimizing the Sentiment Classifier with (Stochastic) Gradient Descent

In the second part of this programming homework, we will revisit the sentiment classifier we built in the last homework. Instead of relying on the PyTorch built-in loss functions, gradient calculations, and weights optimization, we will delve into them and implement our own version!

### Reuse Your HW1 Implementation

**TODOs** (Copy from your HW1): for all the **# TODO (Copy from your HW1)** in gradient\_descent.py, please fill in the blank below them by copying and pasting the corresponding implementations you wrote for homework 1 (i.e. the corresponding **# TODO** in the model.py in homework 1).

### Softmax Function

Remember the [nn.CrossEntropyLoss](https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html) we used in homework 1, it can be further decomposed into that 1) normalize the real-value scores of each class (e.g. the logits) into a probability distribution using softmax function, and calculate the cross entropy loss of this probability distribution against the ground truth binary distribution (In practice, PyTorch instead provides the combination of [LogSoftmax](https://pytorch.org/docs/stable/generated/torch.nn.LogSoftmax.html#torch.nn.LogSoftmax) with [negative log likelihood loss](https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html#torch.nn.NLLLoss)). We will first implement the softmax function, which you have been familiar with in the lecture, and [5](#sec:softmax).  
  
**TODOs**: Complete the softmax function of the SentimentClassifier class in gradient\_descent.py. Note: you must implement the optimized for numerical stability version described in the **Pro tip** in [[subsec:5.1]](#subsec:5.1) of [5](#sec:softmax).  
**Hint**: check the comments in the code for specific input-output requirements.  
A correct implementation should pass the test\_softmax in gradient\_descent.py.  
  
With the softmax function, we turn our neural network into a classifier that assigns a probability distribution over the 2 sentiment classes. Specifically, denoting our input feature vector as , the nn.Linear layer transforms into a **logit score** vector using a weight matrix and a bias vector :

This logit score has one element per class, so the weight matrix must have a size , where is the number of classes (output labels) and is the number of dimensions of the input space (features). The bias vector has elements (one per class).

The logit score is turned into probabilities using the **softmax** operator:

### Gradients on Cross Entropy Loss

We will start by defining an objective function that defines "goodness" for our classifier. A common choice for classification is [categorical cross-entropy loss](https://en.wikipedia.org/wiki/Cross_entropy) or negative log-likelihood.

A discussion or derivation of cross-entropy loss is beyond the scope of this class but a good introduction to it can be [found here](https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/). A discussion of what makes it superior to MSE for classification can be [found here](https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/). We will just focus on its properties instead.

Letting denote the ground truth value of class , and be our prediction of class , the cross-entropy loss is defined as:

If the number of classes is 2 (which is the case here), we can expand this:

Notice that as our probability for predicting the correct class approaches 1, the cross-entropy approaches 0. For example, if , then as , . If our probability for the correct class approaches 0 (the exact wrong prediction), e.g. if and , then .

This is true in the more general -class cross-entropy loss as well, , where if our prediction is very close to the true label, then the entropy loss is close to 0, whereas the more dissimilar the prediction is to the true class, the higher it is.

**Practical tip**: in practice, a very small is added to the log, e.g. to avoid which is undefined.  
  
To optimize this objective, we will compute its gradients with respect to parameters and

Before doing that, let’s redefine cross-entropy loss in matrix form. With a minibatch of input features where is the number of instances per batch (batch size), and is the dimension of feature vectors, and each input is a row vector of dimension . And the corresponding output from our network and label matrix , with row vectors as one-hot encoding of the class label, and as the predicted probabilities assigned to each class. Then the loss can be expressed as

where denotes the summation over all elements of the vector, and are all element-wise.

After doing the derivations, we obtain the gradient with respect to and :

Verify the correctness of this gradient in your own time! :), or check out this [great tutorial](https://jmlb.github.io/ml/2017/12/26/Calculate_Gradient_Softmax/) for the derivation. Note that because is a matrix, too. is therefore the **outer product** between the error vector ( elements) and the input vector ( elements).

For more efficiency, we can write the above expression in matrix form:

Now, let’s put all these together and implement a function for the gradient and loss calculation:  
  
**TODOs**: Read and complete the missing lines in gradient\_loss function of the SentimentClassifier class in gradient\_descent.py to calculate the gradients of the weights and biases of the linear layer, as well as the cross entropy loss.  
**Hint**: refer to [[eq:grad\_w]](#eq:grad_w) and [[eq:grad\_b]](#eq:grad_b) for the gradient calculation formulas, and [[eq:ce]](#eq:ce) for cross-entropy loss calculation. Also, note that the weight matrix that is stored in nn.Linear is actually in the shape of , so you probably need to take a transpose of your gradient calculation.

### Gradient Descent

Remember the gradient descent calculation we learned in class, with the gradient and loss calculation we just implemented, we can now optimize our classifier with gradient descent.  
  
**TODOs** Read and complete the missing lines in train function in gradient\_descent.py, to perform gradient updates on weights and biases, with the given learning\_rate. Once you finish, run the single\_run\_back\_prop in main.py to train the model and paste the plot here.  
**Hint**: you can access the weights and biases of the linear layer with model.linear.weight and model.linear.bias, respectively.

# Optional Feedback

Have feedback for this assignment? Found something confusing? We’d love to hear from you!

1. <https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant> [↑](#footnote-ref-1)
2. Question credit: “Introduction to Natural Language Processing” by J. Eisenstein. [↑](#footnote-ref-2)
3. we count n-gram status within sentence boundary [↑](#footnote-ref-3)
4. in our case of trigram LM, we require the prefix to have tokens [↑](#footnote-ref-4)