CS 601.471/671 NLP: Self-supervised Models

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

For homework deadline and collaboration policy, check the calendar on the course website[*]

Name: __________Zichen Fu____________

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!

How to hand in your written work: Via Gradescope as before.

[*] https://self-supervised.cs.jhu.edu/sp2024/

1 Concepts, intuitions and big picture

1. The n-gram models we've shown make the n-th order Markov assumption, i.e. that the distribution of words depends only on the previous $n-1$ words. What properties of language will it not capture? Discuss (briefly) several distinct ways in which this assumption is false for natural language.
   Answer: *The Markov assumption makes prediction about next word based on the current state (i.e. the current n-gram) and each word is generated only depending on previous n words. This will lose the longer-term context in language. For example, the n-gram (n < 10) model could not know what verb is the most likely to occur calling back to the subject showed 10 words before. This may cause grammatical errors on sentence level, such as run-on. Second, only focusing on local level word combinations may lead to deviations from the desired semantic meanings.*

2. Follow-up to previous question: Despite the Markov assumption, n-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?
   Answer: *n-gram model is good at recognizing the common combinations of words, for example "pros and cons" is commonly used together as a trigram, since this type of common agreement usage in a language can be captureed well by probability. Second, the property of the previous word (is it a noun, verb, or adjective, etc.) grammatically makes expectation to next words (i.e. adjective expects a following noun, a verb does not expect a following verb). At the local level, the n-gram model captures these patterns well, thus the model looks good from the grammatical perspective. Third, by marking the punctuation as words, the n-gram model learns that after several specific words, the comma or period is likely to occur, which implicitly makes it learn the sentence structure that follows the grammatical rules.*

3. Explain how are perplexity and cross-entropy loss related?
   Answer: *The perplexity has the formula of* $ppl = P(w_1, w_2, \ldots, w_n)^{-\frac{1}{n}}$, *which its logit form is* $ppl((w_1, w_2, \ldots, w_n) = 2^H, H = -\frac{1}{n}\sum_{i=1}^{n} log_2(P(w_i|w_1, \ldots, w_{i-1}))$. *This formula is close to the cross entropy loss formula* $H(p,q) = -\sum_{x \in X} p(x) log(q(x))$. *Here* $p(x)$ *represents the true distribution of natural language. Even though it is not known, but by calculating the mean of the articulates of the natural language (the articles used for training), the* $\frac{1}{n}$ *term in the perplexity formula is a valid empirical estimation to the true natural language distribution, thus the formula can be considered as the cross-entropy loss between the model's distribution and the true natural language distribution (or say the difference between these 2 distributions).*

4. For a vocabulary of $|V|$ words, what would you expect perplexity to be if your model predictions were completely random? Compute the corresponding cross-entropy loss for $|V| = 2000$ and $|V| = 10000$, and keep this in mind as a baseline.
   Answer: *For a completely random prediction, the perplexity would be* $|V|$, *meaning the model could not distinguish which word is more possible among all these* $|V|$ *words. Since the prediction is completely random, the probability of each words would be uniform (i.e. the probability of each word is the same as* $\frac{1}{|V|}$*), thus the formula of cross-entropy loss is then* $H = -\frac{1}{|V|} log_2(\frac{1}{|V|})$. *For* $|V| = 2000$, *the cross entropy loss is* $H = \frac{1}{2000} log_2(\frac{1}{2000}) \approx 0.00548$; *for* $|V| = 10000$, *the cross entropy loss is* $H = \frac{1}{10000} log_2(\frac{1}{10000}) \approx 0.00133$.

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.
   Answer: *AC*

6. What does a neuron compute?
   □ A neuron computes an activation function followed by a linear function ($z = Wx + b$)
   □ A neuron computes a linear function ($z = Wx + b$) followed by an activation function.
   □ A neuron computes a function g that scales the input x linearly ($Wx + b$).
   □ A neuron computes the mean of all features before applying the output to an activation function.
   Answer: *BC*

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.
Answer: *BC*

2 Linear Algebra Recap

2.1 Gradients

Consider the following scalar-valued function:

$$f(x,y,z) = x^2y + \sin(z+6y).$$

1. Compute partial derivatives with respect to x, y and z.
   Answer:

$$\begin{aligned}\frac{\partial f}{\partial x} &= 2xy\\ \frac{\partial f}{\partial y} &= x^2 + 6cos(z+6y)\\ \frac{\partial f}{\partial z} &= cos(z+6y)\end{aligned}$$

2. We can consider f to be a function that takes a vector $\theta \in \mathbb{R}^3$ as input, where $\theta = [x,y,z]^\top$. Write the gradient as a vector and evaluate it at $\theta = [3, \pi/2, 0]^\top$.
   Answer: $\nabla f = \begin{bmatrix} 2*3*\pi/2 \\ 3^2 + 6cos(0+6*\pi/2) \\ cos(0+6*\pi/2) \end{bmatrix} = \begin{bmatrix} 3\pi \\ 3 \\ -1 \end{bmatrix}$

2.2 Gradients of vectors

Let $\mathbf{x}, \mathbf{c} \in \mathbb{R}^n$ and $\mathbf{A} \in \mathbb{R}^{n\times n}$. 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 $\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^\top \mathbf{c}) = \mathbf{c}^\top$.
  Answer: *Derivative: vector*
  Method of calculation: expand out the dot product to the summation

$$\begin{aligned}\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^\top \mathbf{c}) &= \frac{\partial}{\partial \mathbf{x}}\left(\sum_{i=1}^n x_i c_i\right)\\ &= \left(\frac{\partial}{\partial x_1}(\sum_{i=1}^n x_i c_i), \frac{\partial}{\partial x_2}(\sum_{i=1}^n x_i c_i), \ldots, \frac{\partial}{\partial x_n}(\sum_{i=1}^n x_i c_i)\right)\\ &= (c_1, c_2, \ldots, c_n)\\ &= \mathbf{c}^\top\end{aligned}$$

- Show that $\frac{\partial}{\partial \mathbf{x}}\left(\|\mathbf{x}\|_2^2\right) = 2\mathbf{x}^\top$.
  Answer: *Derivative: vector*
  Method of calculation: expand out the dot product/magnitude to the summation

$$\begin{aligned}\frac{\partial}{\partial \mathbf{x}}(\|\mathbf{x}\|_2^2) &= \frac{\partial}{\partial \mathbf{x}}\left(\sum_{i=1}^n x_i^2\right)\\ &= \left(\frac{\partial}{\partial x_1}(\sum_{i=1}^n x_i^2), \frac{\partial}{\partial x_2}(\sum_{i=1}^n x_i^2), \ldots, \frac{\partial}{\partial x_n}(\sum_{i=1}^n x_i^2)\right)\\ &= (2x_1, 2x_2, \ldots, 2x_n)\\ &= 2\,(x_1, x_2, \ldots, x_n)\\ &= 2\mathbf{x}^\top\end{aligned}$$

- Show that $\frac{\partial}{\partial \mathbf{x}}\left(\mathbf{A}\mathbf{x}\right) = \mathbf{A}$.
  Answer: *Derivative: matrix*
  Method of calculation: expand out the matrix multiplication to the summation, and also use the previously proved $\frac{\partial}{\partial \mathbf{x}}\left(\mathbf{x}^\top \mathbf{c}\right) = \mathbf{c}^\top$

  Denoting the i-th row in A as a_i, *thus* $A = \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_n \end{bmatrix}$

$$\begin{aligned}\frac{\partial}{\partial \mathbf{x}}\mathbf{A}\mathbf{x} &= \frac{\partial}{\partial \mathbf{x}}\left(\sum_{i=1}^{n} x_i a_i\right) \\ &= \left(\frac{\partial}{\partial x_1}(\sum_{i=1}^{n} x_i a_i), \frac{\partial}{\partial x_2}(\sum_{i=1}^{n} x_i a_i), \ldots, \frac{\partial}{\partial x_n}(\sum_{i=1}^{n} x_i a_i)\right) \\ &= (a_1, a_2, \ldots, a_n) \\ &= \mathbf{A}\end{aligned}$$

- Show that $\frac{\partial}{\partial \mathbf{x}}\left(\mathbf{x}^\top \mathbf{A}\mathbf{x}\right) = \mathbf{x}^\top(\mathbf{A} + \mathbf{A}^\top)$.
  Answer: *Derivative: matrix*
  Method of calculation: chain rule
  Denote $w = Ax$, *then*

$$\begin{aligned}\frac{\partial}{\partial \mathbf{x}}\left(\mathbf{x}^\top \mathbf{A}\mathbf{x}\right) &= \frac{\partial}{\partial \mathbf{x}}\left(\mathbf{x}^\top w\right) \\ &= x^\top \frac{\partial w}{\partial \mathbf{x}} + w^\top \frac{\partial x}{\partial \mathbf{x}} \\ &= x^\top A + w^\top \\ &= x^\top A + x^\top A^\top = x^\top(A + A^\top)\end{aligned}$$

2.3 Jacobian

Consider the following vector function from $\mathbb{R}^3$ to $\mathbb{R}^3$:

$$\mathbf{f}(\boldsymbol{\theta} = [x_1, x_2, x_3]) = \begin{cases} \sin(x_1 x_2 x_3) \\ \cos(x_2 + x_3) \\ \exp(-\frac{1}{2}x_3^2) \end{cases}$$

1. What is the Jacobian matrix[*] of $\mathbf{f}(\boldsymbol{\theta})$?
   Answer: $\begin{bmatrix} x_2x_3cos(x_1x_2x_3) & x_1x_3cos(x_1x_2x_3) & x_1x_2cos(x_1x_2x_3) \\ 0 & -sin(x_2+x_3) & -sin(x_2+x_3) \\ 0 & 0 & -x_3exp(-\frac{x_3^2}{2}) \end{bmatrix}$
2. Evaluate the Jacobian matrix of $\mathbf{f}(\boldsymbol{\theta})$ at $\boldsymbol{\theta} = [1, \pi, 0]$.
   Answer: $\begin{bmatrix} 0 & 0 & \pi \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

[*] https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant

3 n-gram Language Models

3.1 A toy n-gram language model

Consider the following vocabulary, $V = \{\texttt{BOS}, \texttt{EOS}, \texttt{who}, \texttt{framed}, \texttt{roger}, \texttt{rabbit}, \texttt{the}\}$ 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: $P(\texttt{rabbit})$, $P(\texttt{rabbit}|\texttt{roger})$, $P(\texttt{EOS}|\texttt{rabbit})$.
   Answer: $n_{total} = 23$ *including EOS.*

$$P(\texttt{rabbit}) = \frac{3}{23} \approx 0.1304$$

$$P(\texttt{rabbit}|\texttt{roger}) = \frac{count(\texttt{roger rabbit})}{count(\texttt{roger})} = \frac{2}{4} = 0.5$$

$$P(\texttt{EOS}|\texttt{rabbit}) = \frac{count(\texttt{rabbit EOS})}{count(\texttt{rabbit})} = \frac{3}{3} = 1;$$

2. Briefly explain what the sparsity problem is in n-gram language models?
   Answer: *The sparsity refers to the lower probability as the testing n-gram grows longer (i.e. n is larger) since no matter how large the training dataset is, it may not cover every variation of word combinations. Therefore, when the n-gram calculating the conditional probability, the unseen n-grams would all lead to close-to-zero probabilities, which makes the model less likely to distinguish which word is more likely to co-occur with the existing sentence, thus this sparsity makes the model's prediction worse.*

4 Challenges of linear classifiers of sentences

A simple way to compute a representation for a phrase s is to add up the representations of the words in that phrase: $\mathrm{repr}(s) = \sum_{w \in s} v_w$, where $w \in s$ are the word in s and $v_w \in \mathbb{R}^d$ is the embedding for word w.

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

$$f(s; \theta) = \theta \cdot \mathrm{repr}(s),$$

for some choice of parameters θ. Note that here $\theta \in \mathbb{R}^d$ and "$\cdot$" is the "dot-product". Prove that in such a model, the following inequality cannot hold for any choice of θ and word embeddings:

$$f(\texttt{good}; \theta) > f(\texttt{not good}; \theta)$$
$$f(\texttt{bad}; \theta) < f(\texttt{not bad}; \theta)$$

Thereby, showing the inadequacy of this model in capturing negations.[†]
Answer: *Denoting the embedding of "good" as v_{good}, "not" as v_{not}, "bad" as v_{bad}*
From the given f and inequalities:

$$\theta \cdot (v_{good}) > \theta \cdot (v_{good} + v_{not}) = \theta \cdot v_{good} + \theta \cdot v_{not} \tag{1}$$
$$\theta \cdot (v_{bad}) < \theta \cdot (v_{bad} + v_{not}) = \theta \cdot v_{bad} + \theta \cdot v_{not} \tag{2}$$

From (1) we get $0 > \theta \cdot v_{not}$ while from (2) we get $0 < \theta \cdot v_{not}$. These 2 inequality contradict with each other. In conclusion, there is an inadequacy of this model in capturing negations.

2. **Extra Credit:** Construct another example of a pair of inequalities similar to the ones above that cannot both hold.
Answer: *Still let $f(s; \theta) = \theta \cdot repr(s)$. For sentiment that measured with larger f means more positive, in English the inequality holds true for:*

$$f(\texttt{charming}); \theta) < f(\texttt{more charming}); \theta) \tag{3}$$
$$f(\texttt{disliked}); \theta) > f(\texttt{more disliked}); \theta) \tag{4}$$

From (3) we get $0 < \theta \cdot$ v_more while from (4) we get $0 > \theta \cdot$ v_more. These 2 inequality contradict with each other. When the same adjective "more" is added in front of words, it magnifies the extent of sentiment that it pushes positive words to more positive while negative words to more negative; however, this means contradicted arithmetic operations. In conclusion, there is an inadequacy of this model in capturing magnified extents.

3. **Extra Credit:** Consider a slight modification to the previous predictive model:

$$f(s; \theta) = \theta \cdot \mathrm{ReLU}(\mathrm{repr}(s)),$$

where ReLU (rectified linear function) is defined as:

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0. \end{cases}$$

Given this choice of predictive function, show that it is possible to satisfy the above inequalities for some choice of θ. **Hint:** Show there exists parameters θ and word embeddings $v_{\texttt{good}}$, $v_{\texttt{bad}}$ and $v_{\texttt{not}}$ that the inequalities are satisfied.
Answer: *If using ReLU function,*

$$\theta \cdot ReLU(v_{good}) > \theta \cdot ReLU(v_{good} + v_{not})$$
$$\theta \cdot ReLU(v_{bad}) < \theta \cdot ReLU(v_{bad} + v_{not})$$

Assume $d = 2$ such that the embeddings and the θ are all vectors with 2 elements. Then let

$$v_{\texttt{good}} = \begin{bmatrix} 2 \\ -2 \end{bmatrix} \qquad v_{\texttt{not}} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$
$$v_{\texttt{bad}} = \begin{bmatrix} -2 \\ 2 \end{bmatrix} \qquad \theta = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

[†] Question credit: "Introduction to Natural Language Processing" by J. Eisenstein.

Thus

$$ReLU(v_{\texttt{not good}}) = ReLU(\begin{bmatrix} 3 \\ -3 \end{bmatrix}) = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$
$$ReLU(v_{\texttt{not bad}}) = ReLU(\begin{bmatrix} -1 \\ 1 \end{bmatrix}) = \begin{bmatrix} 0 \\ 1 \end{bmatrix})$$

The inequalities are now not contradicting that

$$\theta \cdot ReLU(v_{good}) = -2 > \theta \cdot ReLU(v_{\texttt{not good}}) = -3$$
$$\theta \cdot ReLU(v_{bad}) = -2 < \theta \cdot ReLU(v_{\texttt{not bad}}) = -1$$

4. Given the above result, explain (in 1-2 sentence) why the use of neural networks (which have more complexity than linear models)
   Answer: *First, given the complex nature of language, the decision boundary of the language may not be perfectly suitable for linear separation as for the cases shown above. Using neural networks could add non-linearity to the decision boundary thus improve the prediction accuracy of the model.*

5 Softmax function

Remember the Softmax functions from the class:

$$\text{Softmax: } \sigma(\mathbf{z}) \triangleq [\sigma(\mathbf{z})_1, \ldots, \sigma(\mathbf{z})_K] \text{ s.t. } \sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K)$$

1. Prove that Softmax is invariant to constant offsets in the input, i.e., for any input vector $\mathbf{z}$ and any constant c,

$$\sigma(\mathbf{z}) = \sigma(\mathbf{z} + c)$$

   Pro tip: We make use of this property in practice to increase the numerical stability of our models. Specifically, using $c = -\max_{i \in \{1 \ldots K\}} z_i$, i.e. subtracting its maximum element from all elements of $\mathbf{z}$ would prevent numerical instability due to large values.
   Answer:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

$$\sigma(\mathbf{z} + \mathbf{c})_i = \frac{(e^{z_i + c})}{\sum_{j=1}^{K}(e^{(z_j + c)})} = \frac{e^c \cdot e^{z_i}}{\sum_{j=1}^{K}(e^c \cdot e^{z_j})} = \frac{e^c \cdot e^{z_i}}{e^c \cdot \sum_{j=1}^{K}(e^{z_j})} = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

   Since $c \to e^c$ is always canceled out, any constant c added to the original input vector z would not affect the Softmax result.

2. Softmax maintains the relative order of the elements in $\mathbf{z}$. In particular, show that the largest index is intact after applying Softmax:

$$\arg\max_{i \in \{1 \ldots K\}} z_i = \arg\max_{i \in \{1 \ldots K\}} \sigma(\mathbf{z})_i$$

   Answer: *Given* $\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
   we have $\arg\max_{i \in \{1 \ldots K\}} \sigma(\mathbf{z})_i = \arg\max_{i \in \{1 \ldots K\}} \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
   Since $\sum_{j=1}^{K} e^{z_j}$ *is constant,* $\arg\max_{i \in \{1 \ldots K\}} \sigma(\mathbf{z})_i = \arg\max_{i \in \{1 \ldots K\}} z_i$

3. Define the Sigmoid function as follows:

$$\text{Sigmoid: } S(x) \triangleq \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} = 1 - S(-x).$$

Prove that Softmax is equivalent to the Sigmoid functions when the number of possible labels is two: $K = 2$. Specifically:

$$\sigma([z_1, z_2])_1 = S(z_1 - z_2)$$

Answer:

$$\sigma([z_1, z_2]) = [\frac{e^{z_1}}{e^{z_1} + e^{z_2}}, \frac{e^{z_2}}{e^{z_1} + e^{z_2}}]$$
$$S(z_1 - z_2) = \frac{e^{z_1 - z_2}}{e^{z_1 - z_2} + 1} = \frac{e^{z_1}/e^{z_2}}{e^{z_1}/e^{z_2} + 1} = \frac{e^{z_1}}{e^{z_1} + e^{z_2}}$$
$$\sigma([z_1, z_2])_1 = S(z_1 - z_2) = \frac{e^{z_1}}{e^{z_1} + e^{z_2}}$$

Therefore, when $K = 2$, $\sigma([z_1, z_2])_1 = S(z_1 - z_2)$, the Softmax is equivalent to the Sigmoid functions.

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

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \geq \prod_{\substack{j=1 \\ j \neq i}}^K \frac{1}{1 + e^{-(z_i - z_j)}} = \prod_{\substack{j=1 \\ j \neq i}}^K S(z_i - z_j)$$

Hint: Use the following inequality $(1 + \sum_i \alpha_i) \leq \prod_i (1 + \alpha_i)$ where each $\alpha_i \geq 0$.
Answer:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} = \frac{e^{z_i}/e^{z_i}}{(\sum_{j=1}^K e^{z_j})/e^{z_i}}$$
$$= \frac{1}{e^{z_i}/e^{z_i} + \sum_{j=1}^K e^{-(z_i - z_j)}} = \frac{1}{1 + \sum_{j=1}^K e^{-(z_i - z_j)}}$$
$$\because (1 + \sum_i \alpha_i) \leq \prod_i (1 + \alpha_i), \alpha_i \geq 0$$
$$\therefore e^{z_i} \geq 0 \rightarrow (1 + \sum_i e^{z_i}) \leq \prod_i (1 + e^{z_i})$$
$$\therefore \frac{1}{1 + \sum_{j=1}^K e^{-(z_i - z_j)}} \leq \prod_{\substack{j=1 \\ j \neq i}}^K \frac{1}{1 + e^{-(z_i - z_j)}}$$
$$\because K = 2, \sigma([z_1, z_2])_1 = S(z_1 - z_2)$$
$$\therefore \prod_{\substack{j=1 \\ j \neq i}}^K \frac{1}{1 + e^{-(z_i - z_j)}} = \prod_{\substack{j=1 \\ j \neq i}}^K S(z_i - z_j)$$

6 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 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

6.1 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. $\mathbf{P}(X_t|X1, \cdots, X_{t-1})$) by *counting*. For example

$$\mathbf{P}(\text{mat}|\text{the cat sat on the}) \approx \frac{\text{count("the cat sat on the mat")}}{\text{count("the cat sat on the")}}$$

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

6.1.1 Data Loading

We will use (a subset of) the 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 1e5 training samples to create our LM.

Read the `load_data` function in `ngram_lm.py` for how we obtain the data.

6.1.2 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 [‡] 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 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.

6.1.3 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 subsection 6.1, 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 $n = 3$ so that to build a *trigram* language model.

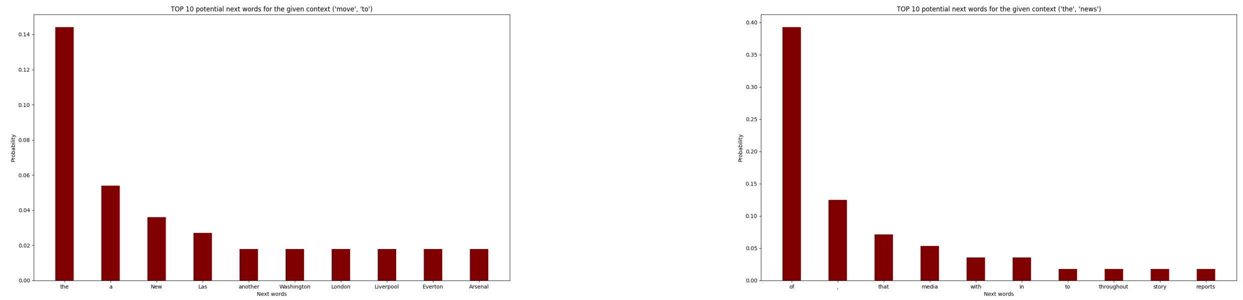
[‡] we count n-gram status within sentence boundary

6.1.4 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 about how to plot bar chart with matplotlib.
- read the comments in the code carefully for more info about the input.

your plots and answer:



(a) top 10 next words of "move to"

(b) top 10 next words of "the news"

Figure 1: The probabilities of top 10 next words of two bi-gram contexts

Answer: *The situation looks common that the most probable next word would have a much higher probability than that of the other words, similar to the "long tail effect" used in economics. For the less-likely options of next words, their probabilities are close to each other (and also close to zero).*

6.1.5 Generation: Sample from the LM

Now that we have built our LM, given any prefix[§], 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 (Meister et al., 2020), top-k, and top-p sampling (Holtzman et al., 2019). 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.
Answer:
Completion 1: According to the report
"According to the report, the first time in the United States, and the other hand, the first time in the United States, and the other hand, the first time"
Completion 2: The president of the association
"The president of the association ' s " The One I Love You ' re not here to Connacht side of the game ' s " The One I Love You ' re not
Findings: Even though the sentence makes sense locally within the neighboring words, but the overall sentence has repetition and does not have solid semantic meaning. Also, the punctuation seems problematic, and it shows a relatively stronger pattern of meaningless repetition.

[§] in our case of trigram LM, we require the prefix to have ≥ 2 tokens

6.2 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!

6.2.1 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).

6.2.2 Softmax Function

Remember the nn.CrossEntropyLoss 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 with negative log likelihood loss). We will first implement the softmax function, which you have been familiar with in the lecture, and section 5.

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 item 1 of section 5.
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 **x**, the `nn.Linear` layer transforms **x** into a **logit score** vector **z** using a weight matrix W and a bias vector **b**:

$$\mathbf{z} = \mathbf{x}W + \mathbf{b}$$

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

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

$$\hat{y}_j = \mathbf{P}(\text{class = j}) = \frac{\exp(z_j)}{\sum_k \exp(z_k)}$$

6.2.3 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 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. A discussion of what makes it superior to MSE for classification can be found here. We will just focus on its properties instead.

Letting y_i denote the ground truth value of class i, and $\hat{y}_i$ be our prediction of class i, the cross-entropy loss is defined as:

$$CE(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i$$

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

$$CE(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

Notice that as our probability for predicting the correct class approaches 1, the cross-entropy approaches 0. For example, if $y = 1$, then as $\hat{y} \to 1$, $CE(y, \hat{y}) \to 0$. If our probability for the correct class approaches 0 (the exact wrong prediction), e.g. if $y = 1$ and $\hat{y} \to 0$, then $CE(y, \hat{y}) \to \infty$.

This is true in the more general M-class cross-entropy loss as well, $CE(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i$, 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 ϵ is added to the log, e.g. $\log(\hat{y} + \epsilon)$ to avoid $\log 0$ which is undefined.

To optimize this objective, we will compute its gradients with respect to parameters W and $\mathbf{b}$

Before doing that, let's redefine cross-entropy loss in matrix form, for all classes:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{N} \sum_i [\mathbf{y}_i \cdot \log \hat{\mathbf{y}}_i] \tag{5}$$

where the summation is over N many input-output instances, $\mathbf{y}_i \in \{0,1\}^c$ is a one-hot encoding of the class label, and $\hat{\mathbf{y}}_i \in [0,1]^c$ is the matrix of probabilities assigned to each class.

After doing the derivations, we obtain the gradient with respect to W and $\mathbf{b}$:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{N} \sum_i [\mathbf{x}_i^\top (\hat{\mathbf{y}}_i - \mathbf{y}_i)] \tag{6}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{1}{N} \sum_i [\hat{\mathbf{y}}_i - \mathbf{y}_i] \tag{7}$$

Verify the correctness of this gradient in your own time! :), or check out this great tutorial for the derivation. Note that because W is a (d, c) matrix, $\frac{\partial \mathcal{L}}{\partial W}$ too. $\mathbf{x}_i^\top (\mathbf{y}_i - \hat{\mathbf{y}}_i)$ is therefore the **outer product** between the error vector $\mathbf{y}_i - \hat{\mathbf{y}}_i$ (c elements) and the input vector $\mathbf{x}$ (d elements).

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

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{N} \mathbf{X}^\top (\hat{\mathbf{Y}} - \mathbf{Y}) \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{1}{N} (\hat{\mathbf{Y}} - \mathbf{Y}) \tag{9}$$

where $\mathbf{X} \in \mathbb{R}^{N \times d}$ is the feature matrix of all our instances, $\mathbf{Y} \in \{0,1\}^{N \times c}$ is the matrix of gold labels (one-hot vector for each instance), $\mathbf{Y} \in [0,1]^{N \times c}$ is the matrix of probabilities for all of our instances. 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 Equation 8 and Equation 9 for the gradient calculation formulas, and Equation 5 for cross-entropy loss calculation.
A correct implementation should pass the `test_gradient_loss` in `gradient_descent.py`.

6.2.4 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.
your plot:

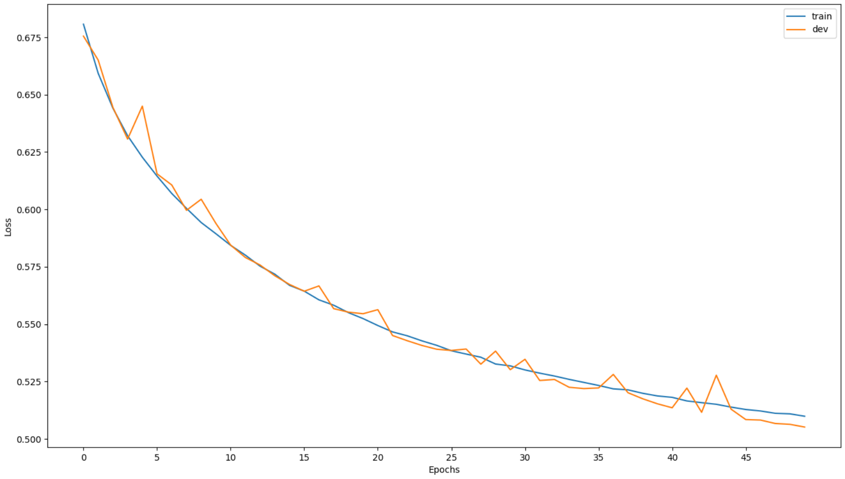


Figure 2: train and dev loss

Optional Feedback

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

References

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. (2019). The curious case of neural text degeneration.

Meister, C., Cotterell, R., and Vieira, T. (2020). If beam search is the answer, what was the question?