COSC490LLMs

Homework 4: Neural Language Modeling + Fixed-Window LMs

Name: Dustin O'Brien
Collaborators, if any: None
Sources used for your homework, if any: Currently None

This assignment focuses on language modeling, while continuing to build up on our prior knowledge of neural networks. We will review several aspects about training neural nets and also extend it to modeling sequences in language.

Homework goals: After completing this homework, you should be comfortable with:

- thinking more deeply about training neural networks; debugging your neural network
- getting more engaged in using PyTorch for training NNs
- training your first neural LM

How to hand in your written work: via MyClasses.

Collaboration: Make certain that you understand the course collaboration policy, described on the course website. You may discuss the homework to understand the problems and the mathematics behind the various learning algorithms, but you are **not allowed to share problem solutions with any other students. You must write the solutions individually**.

Typesetting: We strongly recommend typesetting your homework, especially if you have sloppy handwriting. We will provide a LaTeX template for homework solutions. Type your answers into the corresponding solution field under each question.

1 Concepts, intuitions and big picture

1.1 Multiple-choice questions.

1. Select the sentence that best describes the terms “model”, “architecture”, and “weights”.
☐ Model and weights are the same; they form a succession of mathematical functions to build an architecture.
✓ An architecture is a succession of mathematical functions to build a model and its weights are those functions parameters.
Answer:

2. During forward propagation, in the forward function for a layer l you need to know what is the activation function in a layer (Sigmoid, tanh, ReLU, etc.). During Backpropagation, the corresponding backward function does not need to know the activation function for layer l since the gradient does not depends on it.
☐ True
✓ False
Answer:

3. You have built a network using the tanh activation for all the hidden units. You initialize the weights to relative large values, using `randn(..,..)*1000`. What will happen?
☐ It doesn't matter. So long as you initialize the weights randomly gradient descent is not affected by whether the weights are large or small.
☐ This will cause the inputs of the tanh to also be very large, thus causing gradients to also become large. You therefore have to set the learning rate to be very small to prevent divergence; this will slow down learning.
☐ This will cause the inputs of the tanh to also be very large, causing the units to be “highly activated” and thus speed up learning compared to if the weights had to start from small values.
✓ This will cause the inputs of the tanh to also be very large, thus causing gradients to be close to zero. The optimization algorithm will thus become slow.
Answer:

4. What is the “cache" used for in our implementation of forward propagation and backward propagation?
☐ It is used to cache the intermediate values of the cost function during training.
✓ We use it to pass variables computed during forward propagation to the corresponding backward propagation step. It contains useful values for backward propagation to compute derivatives.
☐ It is used to keep track of the hyperparameters that we are searching over, to speed up computation.
☐ We use it to pass variables computed during backward propagation to the corresponding forward propagation step. It contains useful values for forward propagation to compute activations.
Answer:

5. Among the following, which ones are "hyperparameters"? (Check all that apply.)
✓ size of the hidden layers.
✓ learning rate
✓ number of iterations
✓ number of layers in the neural network
Answer:

6. True/False? Vectorization allows you to compute forward propagation in an L-layer neural network without an explicit for-loop (or any other explicit iterative loop) over the layers $l = 1, 2, \ldots, L$.
☐ True
✓ False
Answer:

7. True or false? A language model usually does not need labels annotated by humans for its pretraining.
✓ True ☐ False
Answer:

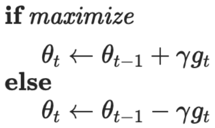
8. What is the order of the language modeling pipeline?
☐ First, the model, which handles text and returns raw predictions. The tokenizer then makes sense of these predictions and converts them back to text when needed.
☐ First, the tokenizer, which handles text and returns IDs. The model handles these IDs and outputs a prediction, which can be some text.

✓ The tokenizer handles text and returns IDs. The model handles these IDs and outputs a prediction. The tokenizer can then be used once again to convert these predictions back to some text.
Answer:

1.2 Short answer questions

1. Look at the definition of [Stochastic] Gradient Descent in PyTorch: `https://pytorch.org/docs/stable/generated/torch.optim.SGD.html`. You will see that this is a bit more complex than what we have seen in the class. Let's understand a few nuances here.

   (a) Notice the `maximize` parameter which controls whether we are running a maximization or minimization. In the algorithm the effect of this parameter is shown as essentially a change in the sign of the gradients: How do you think this change leads to the desired outcome (maximization vs minimization)?



```
if maximize
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$
```

   Answer: *This will attempt to maximize the cost function rather than minimizing it as itll lead to this could be possibly practical for utility maximization problems such as it the parameter was profit based on some inputs the stochastic gradient descent would attempt to maximize profit instead of its usually minimization patterns*

   (b) The next set of parameters `momentum`, `dampening`, `weight_decay`. What do you think the impact of these parameters are? Read the documentations and interpret their roles.
   Answer: *weight_decay is a parameter which pulls the weights closer to 0 and therefore keeps them relatively small to reduce overfitting. It works by subtracting the parameter model size by the weight_decay therefore if weight decay is high itll pull it significantly closer to 0 and if small not much. momentum and dampening are quite related terms momentum is the accumilation of previous descent directions ability to impact the direction of the current angle of descent allowing for descent to sort of break through small decreases in cost and allow for potentially more optimal solutions and also allow for faster learning while dampening is the influence each node has on the current momentum.*

2. What are few benefits to using Fixed-Window-LM over n-grams language models? (limit your answer to less then 5 sentences). Answer: *Firstly, by not having a fixed window and instead a variable window n gram model would result in significantly more possible n-grams and drmatically increase the sparcity problem. Another benefit of the fixed window size is by allowing for smaller windows and thereofre less entries reduces size of matrixes and therefore shrinking amount of memory needing to be used to hold this.*

3. When searching over hyperparameters, a typical recommendation is to do random search rather than a systematic grid search. Why do you think that is the case? (less than 2 sentences). Answer: *This is a better approach as it firstly allows for a more globally optimal set of hyperparameters rather than a locally optimal set. It also prevents overfitting since using something systematic like grid search will result in the model slowly tuning to best parameters for the given dataset rather than true distibution adding randomness fixes this.*

4. Remember the normalization layer that we saw during the class. Here is the corresponding PyTorch page: `https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html`. In the normalization formula, why do we use epsilon ϵ in the denominator?
   Answer: *The fear is given an variance of 0 the following formula will simply not work by adding a small epsilon value to the equation to make sure that this error doesn't occur and instead sets values likely to a very large number.*

2 Softmax Smackdown: Squishing the Competition

2.1 Softmax gradient

You might remember in the midterm exam that we saw a neural network with a Softmax and a cross-entropy loss:

$$\hat{\mathbf{y}} = \text{Softmax}(\mathbf{h}),\ J = \text{CE}(\mathbf{y}, \hat{\mathbf{y}}), \qquad \mathbf{h}, \mathbf{y}, \hat{\mathbf{y}} \in \mathbb{R}^d.$$

Basically here $\hat{\mathbf{y}}$ is a d-dimensional probability distribution over d possible options (i.e. $\sum_i \mathbf{y}_i = 1$ and $\forall i : \mathbf{y}_i \in [0, 1]$). $\mathbf{y}$ is a d-dimensional one-hot vector, i.e., all of these values are zeros except one that corresponds to the correct label.

$$\nabla_{\mathbf{h}} J = \hat{\mathbf{y}} - \mathbf{y}.$$

Prove the above statement. In your proof, use the statement that you proved in HW3 about gradient of Softmax function: $\frac{d\hat{\mathbf{y}}_i}{d\mathbf{h}_j} = \hat{\mathbf{y}}_i(\delta_{ij} - \hat{\mathbf{y}}_j)$, where δ_{ij} is the Kronecker delta function and $\hat{\mathbf{y}}_i$ is the value in the i-th index of $\hat{\mathbf{y}}$.

Answer: *Using the cross entropt formula for CE we get the following*

$$-\sum_i y \log\left(\sigma\left(\mathbf{h}\right)\right)$$

Since y is a one hot encoding it is only 1 at one place and at the rest it is 0 where the previous formula will evaluate to 0

$$0 \cdot \log\left(\sigma\left(\mathbf{h}\right)\right) = 0$$

Therefore we only need to consider the index where $y = 1$

$$-1 \cdot \log\left(\sigma\left(\mathbf{h}\right)\right) = -\log\left(\sigma\left(\mathbf{h}\right)\right)$$

Taking the derivative in reference to h we get

$$\frac{\partial \mathbf{y}}{\partial \mathbf{h}} = -\frac{1}{\sigma\left(\mathbf{h}\right)}\sigma\left(\mathbf{h}\right)\left(\delta_{ij} - \sigma\left(\mathbf{h}\right)\right) = -\left(\delta_{ij} - \sigma\left(\mathbf{h}\right)\right) \rightarrow \sigma\left(\mathbf{h}\right) - \delta_{ij}$$

Since the one hot encoding is 1 when both are same the Kronecker delta is same as y at this point so substiuting back in y for delta and converting sigma notation back to original $\hat{\mathbf{y}}$ *we get*

$$\hat{\mathbf{y}} - \mathbf{y}$$

2.2 Softmax temperature

Remember the softmax function, $\sigma(\mathbf{z})$? Here we will add a *temperature* parameter $\tau \in \mathbb{R}^+$ to this function:

$$\text{Softmax: } \sigma(\mathbf{z};\tau)_i = \frac{e^{z_i/\tau}}{\sum_{j=1}^K e^{z_j/\tau}} \quad \text{for } i = 1, \ldots, K$$

Show the following:

1. In the limit as temperature goes to zero $\tau \to 0$, softmax becomes the same as greedy action selection, `argmax`.
   Answer: *Since the largest exponential will end up dominating the answer and therefore the* $\max\left(z_i\right)$ *will become significantly larger than the rest of the terms and will increase as signifcantly more than the rest of the terms. using this we know*

$$\lim_{\tau \to 0} \frac{e^{z_i/\tau}}{\sum_{j=0}^K e^{z_j/\tau}} \approx \frac{e^{z_i/\tau}}{e^{\max(z_i)/\tau}} \approx 0$$

   while

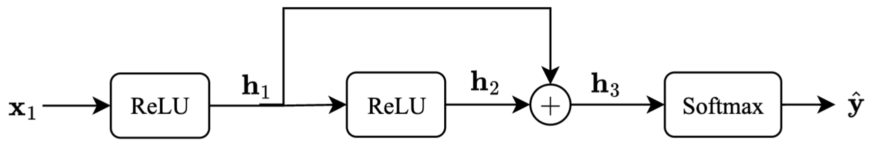
$$\lim_{\tau \to 0} \frac{e^{\max(z_i)/\tau}}{\sum_{j=0}^K e^{z_j/\tau}} \approx \frac{e^{\max(z_i)/\tau}}{e^{\max(z_i)/\tau}} = 1$$

   therefore the max will approach a probability of 1 while the rest of terms will approach 0

2. In the limit as temperature goes to infinity $\tau \to +\infty$, softmax gives equiprobable selection among all actions.
   Answer: *As* $\tau \to +\infty e^{z_i/\tau} \to 1$ *using this we can derive the following*

$$\lim_{\tau \to +\infty} \frac{e^{z_i/\tau}}{\sum_{j=1}^K e^{z_j/\tau}} = \frac{1}{\sum_{j=1}^K 1} = \frac{1}{K}$$

   this is a constant and therefore as temperature approaches infinity the probability of all words approach the same number



$$\mathbf{x} \in \mathbb{R}^d,\ \mathbf{W}_{1,2} \in \mathbb{R}^{d\times d},\ \hat{\mathbf{y}} \in \mathbb{R}^d$$

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x}_1,\ \mathbf{h}_1 = \mathrm{ReLU}(\mathbf{z}_1),$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{h}_1,\ \mathbf{h}_2 = \mathrm{ReLU}(\mathbf{z}_2),$$

$$\mathbf{h}_3 = \mathbf{h}_1 + \mathbf{h}_2,\ \hat{\mathbf{y}} = \mathrm{Softmax}(\mathbf{h}_3),\ J = \mathrm{CE}(\mathbf{y}, \hat{\mathbf{y}})$$

3 Backprop Through Residual Connections

As you know, When neural networks become very deep (i.e. have many layers), they become difficult to train due to the vanishing gradient problem – as the gradient is back-propagated through many layers, repeated multiplication can make the gradient extremely small, so that performance plateaus or even degrades. An effective approach is to add skip connections that skip one or more layers. See the provided network.

1. You have seen this in the quiz, but lets try again. Prove that: $\frac{\partial}{\partial \mathbf{z}_i}\mathrm{ReLU}(\mathbf{z}) = 1\{\mathbf{z}_i > 0\}$ where $1\{x > 0\} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$. More generally, $\nabla_{\mathbf{z}}\mathrm{ReLU}(\mathbf{z}) = \mathrm{diag}\left(1\{\mathbf{z} > 0\}\right)$ where $1\{.\}$ is applied per each dimension and $\mathrm{diag}(.)$ turns a vector into a diagonal matrix.

   Answer: *The defintion of the Relu Function is*

$$Relu(x) = \max{(0, x)}$$

   for all $x > 0$ $Relu(x) = x$ else $x = 0$
   Considering the first section where $x > 0$ we know the following

$$\frac{d}{dx}Relu(x) = \frac{d}{dx}x = 1$$

   and for second portion $x < 0$

$$\frac{d}{dx}Relu(x) = \frac{d}{dx}0 = 0$$

   Using this we can extend this to vectors

   Consider the vector $\mathbf{z} = \begin{bmatrix} z_0 \\ z_1 \\ \cdots \\ z_n \end{bmatrix}$

$$\frac{\partial}{\partial z}\mathbf{z} = \begin{bmatrix} \frac{\partial}{\partial z}\mathbf{z}_0 \\ \frac{\partial}{\partial z}\mathbf{z}_1 \\ \cdots \\ \frac{\partial}{\partial z}\mathbf{z}_n \end{bmatrix} = \frac{\partial}{\partial z}ReLU(\mathbf{z}) = diag(1\{\mathbf{z} > 0\})$$

2. As you see, a single variable ($\mathbf{h}_2$ in the example) feeds into two different layers of the network. Here we want to show that the two upstream signals stemming from this node are merged as summation during Backprop. Specifically prove that:

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_1} = \mathbf{I} + \frac{\partial \mathbf{h}_2}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial \mathbf{h}_1}$$

   Answer: *consider the equation*

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_1} = \boldsymbol{I} + \frac{\partial \mathbf{h}_2}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial \mathbf{h}_1} \rightarrow$$

$$\partial \mathbf{h}_3 = \left(\boldsymbol{I} + \frac{\partial \mathbf{h}_2}{\partial \mathbf{z}_2} \frac{\partial \mathbf{z}_2}{\partial \mathbf{h}_1} \right) \partial \mathbf{h}_1 = \left(\boldsymbol{I} + \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \right) \partial \mathbf{h}_1 \rightarrow \int \partial \mathbf{h}_3 = \int \left(\boldsymbol{I} + \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \right) \partial \mathbf{h}_1$$

$$\int \partial \mathbf{h}_3 = \int \partial \mathbf{h}_1 + \partial \mathbf{h}_2 = \mathbf{h}_3 = \mathbf{h}_1 + \mathbf{h}_2$$

Since we know that the final statement there is true and the equivalence relation holds throughout the math that the original statement must be true

3. In the given neural network your task is to **compute the gradient** $\frac{\partial J}{\partial \mathbf{x}}$. You are allowed (and highly encouraged) to use variables to represent intermediate gradients.

   Hint 1: Compute these gradients in order to build up your answer: $\frac{\partial \mathbf{J}}{\partial \mathbf{h}_3}$, $\frac{\partial \mathbf{J}}{\partial \mathbf{h}_2}$, $\frac{\partial \mathbf{J}}{\partial \mathbf{z}_2}$, $\frac{\partial \mathbf{J}}{\partial \mathbf{h}_1}$, $\frac{\partial \mathbf{J}}{\partial \mathbf{z}_1}$, $\frac{\partial \mathbf{J}}{\partial \mathbf{x}}$. Show your work so we are able to give partial credit! **Hint 2:** Recall that *downstream* = *upstream* * *local*. Answer:

$$\frac{\partial J}{\partial h_3} = \hat{\mathbf{y}} - h_3$$

$$\frac{\partial h_3}{\partial h_2} = 1$$

$$\frac{\partial h_3}{\partial h_1} = 1$$

$$\frac{\partial h_2}{\partial z_2} = diag\left(1\left\{z_2 > 0\right\}\right)$$

$$\frac{\partial z_2}{\partial h_1} = w_2$$

$$\frac{\partial h_1}{\partial z_1} = diag\left(1\left\{z_1 > 0\right\}\right)$$

$$\frac{\partial z_1}{\partial x} = w_1$$

   Following through with the back progprgation we get the Following

$$\frac{\partial J}{\partial x} = \hat{\mathbf{y}} - \left(\frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x} + \frac{\partial h_3}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x} \right) \rightarrow$$

$$\hat{\mathbf{y}} - \left(w_1 diag\left(1z_1 > 0\right) w_2 diag\left(1z_2 > 0\right) + w_1 diag\left(1z_1 > 0\right)\right)$$

4. Based on your derivations, explain why residual connections help mitigate vanishing gradients. Answer: *We can see that vanishing gradients will be mitigated as we are reintroducting the derivative and therefore increasing the amount of influence the first term has on the final outcome.*

4 Programming

In this programming homework, we will

- implement your own subword tokenizer.
- implement fixed-window MLP language models

Skeleton Code and Structure: The code base for this homework can be found at MyClasses Files under the `hw3` 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:

- `tokenization.py` implements the Byte-Pair Encoding algorithm for subword tokenization.
- `mlp_lm.py` implements a fixed-window MLP-based language model on a subset of Wikipedia.
- `main.py` provides the entry point to run your implementations in both `tokenizatiton.py` and `mlp_lm.py`.
- `hw4.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.

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

4.1 Tokenization Algorithms

All NLP systems have *at least* three main components that help machines understand natural language:

1. Tokenization
2. Embedding
3. Model architectures

Models like BERT, GPT-2 or GPT-3 all share the same components but with different architectures that distinguish one model from another. We are going to focus on the first component of an NLP pipeline which is **tokenization**. An often overlooked concept but it is a field of research in itself. Though we have SOTA algorithms for tokenization it's always a good practice to understand the evolution trail.

Here's what we'll cover:

- What is tokenization?
- Why do we need a tokenizer?
- Types of tokenization - Word, Character and Subword.
- Byte Pair Encoding Algorithm
- Other tokenization algorithms:
  - Unigram Algorithm
  - WordPiece - BERT transformer
  - SentencePiece - End-to-End tokenizer system

4.1.1 What is Tokenization?

Tokenization is the process of representing raw text in smaller units called tokens. These tokens can then be mapped with numbers to further feed to an NLP model.
Read and run the `full_word_tokenization_demo` in `tokenization.py` for an overly simplified example of what a tokenizer does.

This is a very simple example where we just split a text string in to "whole words" on white spaces, and we have not considered grammar, punctuation, or compound words (like "test", "test-ify", "test-ing", etc.).

Problems with word tokenization:

- **Missing words in the training data**: With word tokens, your model won't recognize the variants of words that were not part of the data on which the model was trained. So, if your model has seen 'foot' and 'ball' in the training data but the final text has "football", the model won't be able to recognize the word and it will be treated with a "<UNK>" token. Similarly, punctuations pose another problem, "let" or "let's" will need individual tokens and it is an inefficient solution. This will **require a huge vocabulary** to make sure you've every variant of the word. Even if you add a **lemmatizer** to solve this problem, you're adding an extra step in your processing pipeline.
- **Not all languages use space for separating words**: For a language like Chinese, which doesn't use spaces for word separation, this tokenizer will fail.

4.1.2 Character-based tokenization

To resolve the problems associated with word-based tokenization, an alternative approach of character-by-character tokenization was tried. This solves the problem of missing words as now we are dealing with characters that can be encoded using ASCII or Unicode and it could generate embedding for any word now.

Every character, be it space, apostrophes, or colons, can now be assigned a symbol to generate a sequence of vectors. But this approach had its cons. Character-based models will treat each character and will lead to longer sequences. For a 5-word long sentence, you may need to process 30 tokens instead of 5 word-tokens.

4.1.3 Subword Tokenizaiton

To address the earlier issues, we could think of breaking down the words based on a set of prefixes and suffixes. For example, we can build a system that can identify subwords like "##s", "##ing", "##ify", "un##" etc., where the position of double hash "##" denotes prefix and suffixes. So, a word like "unhappily" is tokenized using subwords like "un##", "happ", and "##ily".

BPE (Byte-Pair Encoding) was originally a data compression algorithm that is used to find the best way to represent data by identifying the common byte pairs. It is now used in NLP to find the best representation of text using the least number of tokens.

Here's how it works:

1. Add a "</w>" at the end of each word to identify the end of a word and then calculate the word frequency in the text.
2. Split the word into characters and then calculate the character frequency.
3. For a predefined number of iterations (set by you), count the frequency of the consecutive tokens and merge the most frequently occurring pairings.
4. Keep iterating until you have reached the iteration limit or if you have reached the token limit.

We will now go through this algorithm step by step.

Read the `get_word_freq` function in `tokenization.py` that implements the step 1 above to preprocess each word and count its frequency.

TODOs: Read and complete the missing lines in the `get_pairs` function in `tokenization.py` to implement the step 2 above. This function takes the word frequency record returned from `get_word_freq` as input, split the words into tokens (characters at this initial step), and counts token-pairs frequency.

Hint: follow the comments in the code for specific requirements.

Read the `get_most_frequent_pair` and `merge_byte_pairs` functions in `tokenization.py` that implements the step 3 above, merge the top-1 frequent token-pairs in all the words.

In practise, after iterating over the input text for a desired number of times, we extract the resulting tokenization as our subword dictionary. For this purpose, we provide `get_subword_tokens` function in `tokenization.py`

With all these functions, we can now run one step of the BPE algorithm!

Read and run the `test_one_step_bpe.py` in `main.py` to test the BPE algorithm for one step. A correct implementation of `get_pairs` should pass all the tests.

4.1.4 Complete the BPE Algorithm

With the above implementations, we can complete all the steps of the BPE algorithm that iterates over the input corpus.

TODOs: Read and complete the missing lines in the `exract_bpe_subwords` function in `tokenization.py` that implements the full BPE algorithm. Once you finished, run the `test_bpe` function in `main.py`, a correct implementation of `exract_bpe_subwords` should pass all the tests.
Hint: follow the comments in the code and you can reuse the helper functions provided/you implemented above.

So as we iterate with each best pair, we merge (concatenating) the pair and you can see as we recalculate the frequency, the original character token frequency is reduced and the new paired token frequency pops up in the token dictionary.

4.1.5 Running BPE on a subset of Wikipedia

Now, let's run this algorithm on a larger scale. In particular, we will run on many Wikipedia paragraphs. As usual, let's use the Huggingface library to download the data (the `load_bpe_data` function).

TODOs: Read and run the `bpe_on_wikitext` in `main.py` that iterates the BPE algorithm on a subset of Wikipedia, and interprets the final subwords extracted from the Wikipedia documents. In particular, identify examples of subwords that correspond to

1. complete words
2. part of a word
3. non-English words(including other languages or common sequence special characters)

Explain why these subwords have come about. (no more than 10 sentences)

4.1.6 Additonal Readings: Other Tokenization Algorithms

WordPiece
WordPiece is the subword tokenization algorithm used for BERT, DistilBERT, and Electra. The algorithm was outlined in (**?**) and is very similar to BPE. WordPiece first initializes the vocabulary to include every character present in the training data and progressively learns a given number of merge rules. In contrast to BPE, WordPiece does not choose the most frequent symbol pair, but the one that maximizes the likelihood of the training data once added to the vocabulary.

So what does this mean exactly? Referring to the previous example, maximizing the likelihood of the training data is equivalent to finding the symbol pair, whose probability is divided by the probabilities of its first symbol followed by its second symbol is the greatest among all symbol pairs. E.g. "u", followed by "g" would have only been merged if the probability of "ug" divided by "u", "g" would have been greater than for any other symbol pair. Intuitively, WordPiece is slightly different from BPE in that it evaluates what it *loses* by merging two symbols to ensure it's *worth it*.

Unigram
Unigram is a subword tokenization algorithm introduced in (**?**). In contrast to BPE or WordPiece, Unigram initializes its base vocabulary to a large number of symbols and progressively trims down each symbol to obtain a smaller vocabulary. The base vocabulary could for instance correspond to all pre-tokenized words and the most common substrings. Unigram is not used directly for any of the models in the transformers, but it's used in conjunction with SentencePiece(section 4.1.6).

At each training step, the Unigram algorithm defines a loss (often defined as the log-likelihood) over the training data given the current vocabulary and a unigram language model. Then, for each symbol in the vocabulary, the algorithm computes how much the overall loss would increase if the symbol was to be removed from the vocabulary. Unigram then removes p (with p usually being 10% or 20%) percent of the symbols whose loss increase is the lowest, i.e. those symbols that least affect the overall loss over the training data. This process is repeated until the vocabulary has reached the desired size. The Unigram algorithm always keeps the base characters so that any word can be tokenized.

Because Unigram is not based on merge rules (in contrast to BPE and WordPiece), the algorithm has several ways of tokenizing new text after training. As an example, if a trained Unigram tokenizer exhibits the vocabulary:
`[‘‘b’’, ‘‘g’’, ‘‘h’’, ‘‘n’’, ‘‘p’’, ‘‘s’’, ‘‘u’’, ‘‘ug’’, ‘‘un’’, ‘‘hug’’],`

"hugs" could be tokenized both as `[‘‘hug’’, ‘‘s’’]`, `[‘‘h’’, ‘‘ug’’, ‘‘s’’]` or `[‘‘h’’, ‘‘u’’, ‘‘g’’, ‘‘s’’]`. So which one to choose? Unigram saves the probability of each token in the training corpus on top of saving the vocabulary so that the probability of each possible tokenization can be computed after training. The algorithm simply picks the most likely tokenization in practice but also offers the possibility to sample a possible tokenization according to their probabilities.

Those probabilities are defined by the loss the tokenizer is trained on. Assuming that the training data consists of the words $x_1, \ldots, x_N$ and that the set of all possible tokenizations for a word x_i is defined as $S(x_i)$, then the overall loss is defined as

$$\mathcal{L} = -\sum_{i=1}^{N} \log \left(\sum_{x \in S(x_i)} p(x) \right)$$

SentencePiece
All tokenization algorithms described so far have the same problem: It is assumed that the input text uses spaces to separate words. However, not all languages use spaces to separate words. One possible solution is to use language-specific pre-tokenizers, e.g. XLM uses a specific Chinese, Japanese, and Thai pre-tokenizer). To solve this problem more generally, **?** treats the input as a raw input stream, thus including the space in the set of characters to use. It then uses the BPE or unigram algorithm to construct the appropriate vocabulary.

The XLNetTokenizer uses SentencePiece for example, which is also why in the example earlier the "_" character was included in the vocabulary. Decoding with SentencePiece is very easy since all tokens can just be concatenated and "_" is replaced by a space.

All transformer models in the library that use SentencePiece use it in combination with unigram. Examples of models using SentencePiece are ALBERT XLNet,Marian, and T5.

4.2 Fixed-Window MLP Language Models

In the second part of the homework, we will build, train, and evaluate fixed-window MLP language models as we learned in class: given the context words in the fixed-size window on the left hand side, predict the next word continuation.

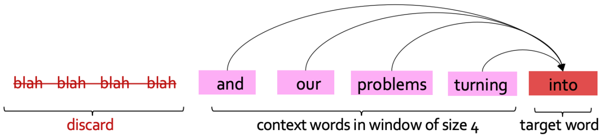


Figure 1: Fixed-window LM

4.2.1 Data Loading and Preprocessing

We will be using the same WikiText dataset we used for building n-gram LM in homework 2. We follow similar preprocessing steps to split paragraphs into sentences followed by tokenization (Now we have learned more about subword tokenization in class and the previous question!).

TODOs read the `preprocess_data` function in `ngram_lm.py` and complete the missing lines to prepare input-output pairs for training fixed-window LMs.
Hint: follow the requirements in the code comments.

4.2.2 Build our LM

Let's now turn to building our model. Here, we are following the footsteps of the neural probabilistic language model (NPLM) (**?**), which is extending earlier studies such as **?**.

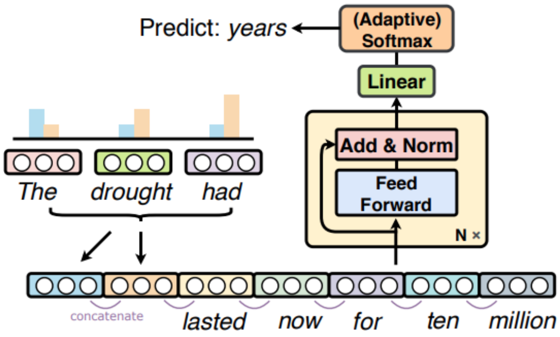


Figure 2: NPLM Architecture

We will follow a modular design for our implementation to make it more interpretable. In particular, will implement 3 layers:

- The **input layer** which loops up word embeddings, concatenates them, and transforms them into a hidden representation.
- The **middle layer** transforms the representation with a non-linearity.
- The **output layer** which transforms a hidden vector to a probability distribution over the words

Let's get to work!

TODOs: read and complete the `forward` functions for the following classes

- `NPLM_first_block`: input layer that embeds the input token ids into embedding vectors, concatenates the embeddings, applies linear transformation, layer normalization, and dropout
- `NPLM_block`: middle layers with linear transformation, tanh activation, residual connection, layer normalization, and dropout.
- `NPLM_final_block`: output layer that transforms hidden representation to log probability over the vocabulary with log softmax.

Hint: check out nn.LayerNorm and nn.Dropout, torch.tanh and nn.functional.log_softmax for more details on these layers and operations. For embedding concatenation, you may find torch.Tensor.view useful. Also, follow the step-by-step comments in the code for the requirements of the implementation.

TODOs: read and complete the `__init__` and `forward` functions for the `NPLM` class, which is the final model that stacks all the above layers.
Hint: remember to apply ReLU non-linear activation after each middle layer, details at nn.functional.relu.

4.2.3 Train and Evaluate the LM

Now it's time to train and evaluate it! Like the previous homework, we will use cross-entropy loss between the predictions of our language model and actual words that appeared in our training data. Note that we applied log softmax to the final output of the LM, as described in homework to, we will use negative log-likelihood loss as the training criterion to calculate the cross-entropy loss.

As introduced in the class, we will use **Perplexity** to evaluate our LM, as a measure of predictive quality of a language model.

TODOs Read and complete the missing lines in `train` and `evaluate` functions in `mlp_lm.py` to calculate the perplexity.
Hint: remember in the class we discussed the connection between perplexity and cross-entropy loss, and note that in our implementation we took natural logarithm instead of the base of 2.

Note we are using Adam **(Adaptive Moment Estimation)** (**?**), which is a popular optimization algorithm used in deep learning and machine learning. It is a stochastic gradient descent (SGD) optimization algorithm that is well suited for training deep neural networks. The algorithm has some internal estimates to dynamically adjust the learning rates for each parameter based on its past gradients, which can result in faster convergence and improved performance compared to traditional SGD.

TODOs Once you finished all the implementations, run `load_data_mlp_lm` and `single_run_mlp_lm` in `main.py` to train and evaluate the model and paste the plots of loss and Perplexity here.
your plots

4.2.4 Sample From a Pre-trained LM

Note that compared with the original NPLM implementation, we reduce the model size by shrinking both the width and depth of our model, and we only use a small subset of the data. It is expected to take an hour or so to train the model on the CPU of your PC. Training the above LM with full size model and data might take hours to days. To save you time, we have trained a language model for you to play with. All you have to do is to download its weight parameters. Download the pre-trained weights and copy it to your local directory under `/hw4/`.

Now that we have the model weight downloaded, we can instantiate a model with these parameters. This will work in two steps.

- First we will need to create a new model (with potentially random weights).
- Then, we will copy the parameter values to the model using load_state_dict function.

We then evaluate the loaded model on the dev set. Now that we have loaded a pre-trained LM, how can we sample from it? We will implement two strategies, greedy decoding and top-p sampling (**?**) that we briefly discussed in homework 2.

For greedy, we select the most probable words (argmax).

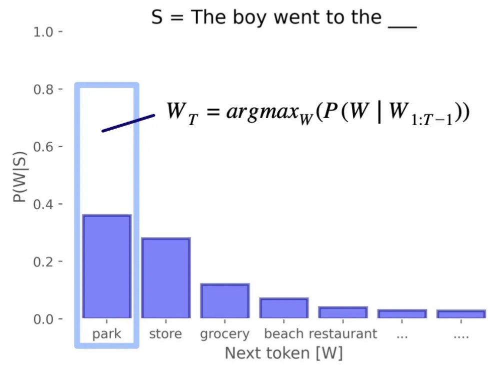


Figure 3: Greedy Decoding

For top-p, we sample from the distribution proportional to word probabilities. Words with higher probabilities will be more likely to be sampled. However, we also have an option of filtering the low-probability tokens, and instead retaining tokens that constitute texttttop_p probability.
Read `generate_text` and `sample_from_mlp_lm` in `mlp_lm.py` for more details about how to load the pre-trained model, evaluate on dev set and perform greedy/top-p sampling. You can learn more about top-p sampling implementation at TopPLogitsWarper.

TODOs Run the `sample_from_trained_mlp_lm` in `main.py` to sample from the pre-trained LM, paste the completion to the prefix under different sampling strategies here, and describe in 2-3 sentences your findings.

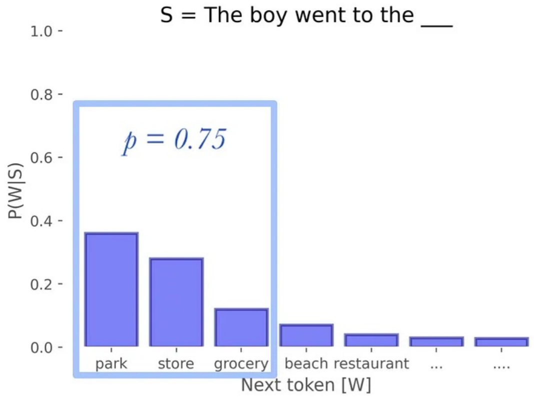


Figure 4: Greedy Decoding

Hint: compare the output of the above generations, which one is your favorite? Explain why this choice of sampling leads to better text generation.

Answer:

greedy: The best perks of living on the east
top-p sampling (p=0.0): The best perks of living on the east
top-p sampling (p=0.3): The best perks of living on the east
top-p sampling (p=1.0): The best perks of living on the east