# Homework 1, CMSC848O Spring 2025

This is due on **February 25th, 2025**, submitted via Gradescope as a PDF (File>Print>Save as PDF). 100 points total.

**IMPORTANT**: After copying this notebook to your Google Drive, please paste a link to it below. To get a publicly-accessible link, hit the *Share* button at the top right. Change the access permissions from "Restricted" to "Anyone with the link" and then click the "Copy link" button. Paste the result below. If you fail to do this, you will receive no credit for this homework!

***LINK: https://colab.research.google.com/drive/1SlyBWm9jNSFgp5HwfjS7yxeCZ7QvtTBg?usp=sharing****

---

*How to do this problem set:*

- Some questions require writing simple Python code and computing results, and the rest of them have written answers. For coding problems, you will have to fill out all code blocks that say `YOUR CODE HERE`.
- For text-based answers, you should replace the text that says "Write your answer here..." with your actual answer.
- This assignment is designed so that you can run all cells almost instantly. If it is taking longer than that, you have made a mistake in your code.
- There is no penalty for using AI assistance on this homework as long as you fully disclose it in the final cell of this notebook (this includes storing any prompts that you feed to large language models). That said, anyone caught using AI assistance without proper disclosure will receive a zero on the assignment (we have several automatic tools to detect such cases). We're literally allowing you to use it with no limitations, so there is no reason to lie!

---

*How to submit this problem set:*

- Write all the answers in this Colab notebook. Once you are finished, generate a PDF via (File -> Print -> Save as PDF) and upload it to Gradescope.
- **Important:** check your PDF before you submit to Gradescope to make sure it exported correctly. If Colab gets confused about your syntax, it will sometimes terminate the PDF creation routine early.
- **Important:** on Gradescope, please make sure that you tag each page with the corresponding question(s). This makes it significantly easier for our graders to grade submissions. We may take off points for submissions that are not tagged.
- When creating your final version of the PDF to hand in, please do a fresh restart and execute every cell in order. Then you'll be sure it's actually right. One handy way to do this is by clicking `Runtime -> Run All` in the notebook menu.

---

*Academic honesty*

- We will audit the Colab notebooks from a set number of students, chosen at random. The audits will check that the code you wrote actually generates the answers in your PDF. If you turn in correct answers on your PDF without code that actually generates those answers, we will consider this a serious case of cheating. See the course page for honesty policies.
- We will also run automatic checks of Colab notebooks for plagiarism. Copying code from others is also considered a serious case of cheating.

---

## Question 1.1 (5 points)

You have a unigram language model with a vocabulary of just two words: {cat,dog}. The model always predicts "cat" with probability 0.7 and "dog" with probability 0.3, regardless of any context. You have a small test sequence that is exactly four words long: "cat cat dog cat".

Compute the perplexity of the model on this test sequence.

---

**Write your answer here!** If you want partial credit, please include any intermediate equation(s) formatted in LaTeX in your answer. You can add LaTeX to your answer by wrapping it in $ signs; see the above cell for examples. If you've never used LaTeX before, please check out this notebook to get the hang of it!

```
from math import exp, log
round(exp(-(log(0.7)+log(0.7)+log(0.3)+log(0.7))/4), 3)
```

```
1.766
```

## Give Information:

- Vocab: {cat, dog}
- P(cat) = 0.7
- p(dog) = 0.3
- sentence: cat cat dog cat
- N = 4
- Unigram model hence $P(t_i|t_{1,2,\cdots i}) = P(t_i)$ that is the words are independent

$$\text{Perplexity} = exp(-\frac{1}{4}\sum_{i=1}^{4} logP(t_i|t_{1,2\cdots 4})) = exp(-\frac{1}{4}logP(t_i)) \because \text{its an unigram model}$$

### Step 1

$$\begin{aligned} logP(t_i) &= log(P(cat) \times P(cat) \times P(dog) \times P(cat)) \\ &= log(P(cat)) + log(P(cat)) + log(P(dog)) + log(P(cat)) \\ &= log(0.7) + log(0.7) + log(0.3) + log(0.7) \\ &= -0.357 - 0.357 - 1.24 - 0.357 \\ &= -2.274 \end{aligned}$$

### Step 2

$$\text{Perplexity} = exp(-\frac{1}{4} \times -2.274) = 1.766$$

## Question 1.2 (5 points)

Let's keep going with the same language model from the previous question. Now, you receive a new test sequence: "cat dog fish cat". What is the perplexity of the model on this new test sequence?

---

**Write your answer here!** As before, if you want partial credit, please include intermediate computations in LaTeX here.

- Sentence: cat dog fish cat

Over here Fish is (out of vocabulary word) hence we attach a very small probability with the word fish $P(fish) \rightarrow 0$ now since its tends to 0 $log(P(fish)) = \infty$

## Calculation

$$\begin{aligned} \text{Perplexity} &= exp(-\frac{1}{4}log(P(t_i)) \\ &= exp(-\frac{1}{4}(log(P(cat)) + log(P(dog)) + log(P(fish)) + log(P(cat)))) \\ &= exp(-\frac{1}{4}(log(0.7) + log(0.3) + log(0) + log(0.7)) \\ &= exp(-\frac{1}{4}(-0.357 - 1.24 + \infty - 0.357)) \\ &= exp(-\frac{1}{4}(-1.917 + \infty)) \\ &= \infty \end{aligned}$$

## Question 1.3 (5 points)

Here is a simple way to build a language model: for any prefix $w_1, w_2, \ldots, w_{i-1}$, retrieve all occurrences of that prefix in some huge text corpus (such as the Common Crawl) and keep count of the word $w_i$ that follows each occurrence. Now, use these counts to estimate the

conditional probability $P(w_i|w_1, w_2, \ldots, w_{i-1})$ for any prefix. Explain why this method is completely impractical!

---

**Write your answer here!** Please keep it brief (i.e., 2-3 sentences).

This approach is impractical because the number of possible prefixes grows exponentially, leading to massive memory requirements and severe data sparsity, making it impossible to store or compute probabilities efficiently. Additionally, the model would struggle to generalize, as it relies solely on exact prefix matches from the training corpus and cannot handle unseen word sequences or out-of-vocabulary words. The computational cost of searching for occurrences in a large corpus like Common Crawl is prohibitively high, making this method infeasible for real-world applications.

## Question 2.1 (5 points)

Let's switch over to some simple coding! The below coding cell contains the opening paragraph of Daphne du Maurier's novel *Rebecca*. Write some code in this cell to compute the number of unique word **types** and total word **tokens** in this paragraph. Use a whitespace tokenizer to separate words (i.e., split the string on white space using Python's split function, don't worry about handling punctuation properly). Be sure that the cell's output (i.e., after running it) is visible in the PDF file you turn in on Gradescope.

---

```
paragraph = '''Last night I dreamed I went to Manderley again. It seemed to me
that I was passing through the iron gates that led to the driveway.
The drive was just a narrow track now, its stony surface covered
with grass and weeds. Sometimes, when I thought I had lost it, it
would appear again, beneath a fallen tree or beyond a muddy pool
formed by the winter rains. The trees had thrown out new
low branches which stretched across my way. I came to the house
suddenly, and stood there with my heart beating fast and tears
filling my eyes.'''.lower() # lowercase normalization is often useful in NLP

types = 0
tokens = 0

# YOUR CODE HERE! POPULATE THE types AND tokens VARIABLES WITH THE CORRECT VALUES!
token_list = paragraph.split()
types = len(set(token_list))
tokens = len(token_list)
# DO NOT MODIFY THE BELOW LINE!
print('Number of word types: %d, number of word tokens: %d' % (types, tokens))
```

```
Number of word types: 76, number of word tokens: 100
```

## Question 2.2 (5 points)

Now let's look at the most frequently used word **types** in this paragraph. Write some code in the below cell to print out the ten most frequently-occurring types. We have initialized a Counter object that you should use for this purpose. In general, Counters are very useful for text processing in Python. Be sure that the cell's output (i.e., after running it) is visible in the PDF file you turn in on Gradescope.

---

```
from collections import Counter
c = Counter(token_list)

# DO NOT MODIFY THE BELOW LINES!
for word, count in c.most_common(10):
  print(word, count)
```

```
i 6
the 6
to 4
a 3
and 3
my 3
it 2
that 2
was 2
with 2
```

## Question 2.3 (5 points)

What do you notice about these words and their linguistic functions (i.e., parts-of-speech)? These words are known as "stopwords" in NLP and are often removed from the text before any computational modeling is done. Why do you think that is?

---

**Write your answer here!** Please keep it brief (i.e., 2-3 sentences).

The most frequent word are: 'i', 'the', 'to', 'a', 'and', 'my', 'it', 'that', 'was', 'with'. These words serve grammatical meaning rather than carrying significant meaning, we can also not differentiate one document from another as these words are frequently used, removing them allow us to focus on the content also reducing the dimensionality hence reducing the computations faster and more memory efficient.

## Question 3.1 (10 points)

In *neural* language models, we represent words with low-dimensional vectors also called *embeddings*. We use these embeddings to compute a vector representation $x$ of a given prefix, and then predict the probability of the next word conditioned on $x$. In the below cell, we use PyTorch, a machine learning framework, to explore this setup. We provide embeddings for the prefix "Alice talked to"; your job is to combine them into a single vector representation $x$ using element-wise vector addition. Be sure that the cell's output (i.e., after running it) is visible in the PDF file you turn in on Gradescope.

*TIP: if you're finding the PyTorch coding problems difficult, you may want to run through the 60 minutes blitz tutorial!*

---

```
import torch
torch.set_printoptions(sci_mode=False)
torch.manual_seed(0)

prefix = 'Alice talked to'

# spend some time understanding this code / reading relevant documentation!
# this is a toy problem with a 5 word vocabulary and 10-d embeddings
embeddings = torch.nn.Embedding(num_embeddings=5, embedding_dim=10)
vocab = {'Alice':0, 'talked':1, 'to':2, 'Bob':3, '.':4}

# we need to encode our prefix as integer indices (not words) that index
# into the embeddings matrix. the below line accomplishes this.
# note that PyTorch inputs are always Tensor objects, so we need
# to create a LongTensor out of our list of indices first.
indices = torch.LongTensor([vocab[w] for w in prefix.split()])
prefix_embs = embeddings(indices)
print('prefix embedding tensor size: ', prefix_embs.size())

# okay! we now have three embeddings corresponding to each of the three
# words in the prefix. write some code that adds them element-wise to obtain
# a representation of the prefix! store your answer in a variable named "x".

### YOUR CODE HERE!
x = torch.sum(prefix_embs, dim=0)

### DO NOT MODIFY THE BELOW LINE
print('embedding sum: ', x)
```

```
prefix embedding tensor size:  torch.Size([3, 10])
embedding sum:  tensor([-0.1770, -2.3993, -0.4721,  2.6568,  2.7157, -0.1408, -1.8421, -3.6277,
         2.2783,  1.1165], grad_fn=<SumBackward1>)
```

## Question 3.2 (5 points)

Modern language models do not use element-wise addition to combine the different word embeddings in the prefix into a single representation (a process called *composition*). What is a major issue with element-wise functions that makes them unsuitable for use as composition functions?

---

**Write your answer here!** Please keep it brief (i.e., 2-3 sentences).

One major issue with element-wise functions like addition is that they are permutation invariant. This means that the function treats the inputs the same regardless of their order. In language, word order is crucial for conveying meaning, so a composition function that ignores order cannot capture the syntactic and semantic structure of a sentence effectively.

## Question 3.3 (10 points)

One very important function in neural language models (and for basically every task we'll look at this semester) is the softmax, which is defined over an $n$-dimensional vector $< x_1, x_2, \dots, x_n >$ as $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{1 \le j \le n} e^{x_j}}$. Let's say we have our prefix representation $\boldsymbol{x}$ from before. We can use the softmax function, along with a linear projection using a matrix $W$, to go from $\boldsymbol{x}$ to a probability distribution $p$ over the next word: $p = \text{softmax}(W\boldsymbol{x})$. Let's explore this in the code cell below. Be sure that the cell's output (i.e., after running it) is visible in the PDF file you turn in on Gradescope.

```
torch.manual_seed(0)

# remember, our goal is to produce a probability distribution over the
# next word, conditioned on the prefix representation x. This distribution
# is thus over the entire vocabulary (i.e., it is a 5-dimensional vector).
# take a look at the dimensionality of x, and you'll notice that it is a
# 10-dimensional vector. first, we need to **project** this representation
# down to 5-d. We'll do this using the below matrix:

W = torch.rand(10, 5)

# use this matrix to project x to a 5-d space, and then
# use the softmax function to convert it to a probability distribution.
# this will involve using PyTorch to compute a matrix/vector product.
# look through the documentation if you're confused (torch.nn.functional.softmax)
# please store your final probability distribution in the "probs" variable.

### YOUR CODE HERE
logits = torch.matmul(x, W)
probs = torch.nn.functional.softmax(logits, dim=0)


### DO NOT MODIFY THE BELOW LINE!
print('probability distribution', probs)

```

```
probability distribution tensor([0.5722, 0.3568, 0.0414, 0.0206, 0.0089], grad_fn=<SoftmaxBackward0>)
```

## Question 3.4 (15 points)

So far, we have looked at just a single prefix ("Alice talked to"). In practice, it is common for us to compute many prefixes in one computation, as this enables us to take advantage of GPU parallelism and also obtain better gradient approximations. This is called *batching*, where each prefix is an example in a larger batch. Here, you'll redo the computations from the previous cells, but instead of having one prefix, you'll have a batch of two prefixes. The final output of this cell should be a 2x5 matrix that contains two probability distributions, one for each prefix.
**NOTE: YOU WILL LOSE POINTS IF YOU USE ANY LOOPS IN YOUR ANSWER!** Your code should be completely vectorized (a few large computations is faster than many smaller ones). Be sure that the cell's output (i.e., after running it) is visible in the PDF file you turn in on Gradescope.

```
torch.manual_seed(0)

# for this problem, we'll just copy our old prefix over two times
# to form a batch. in practice, each example in the batch would be different.
batch_indices = torch.cat(2 * [indices]).reshape((2, 3))
batch_embs = embeddings(batch_indices)
print('batch embedding tensor size: ', batch_embs.size())

# now, follow the same procedure as before:
# step 1: compose each example's embeddings into a single representation
# using element-wise addition. HINT: check out the "dim" argument of the torch.sum function!
batch_composed = torch.sum(batch_embs, dim=1)

#
# step 2: project each composed representation into a 5-d space using matrix W
batch_logits = torch.matmul(batch_composed, W)
# step 3: use the softmax function to obtain a 2x5 matrix with the probability distributions

# please store this probability matrix in the "batch_probs" variable, which is
# currently initialized with random numbers.

batch_probs = torch.nn.functional.softmax(batch_logits, dim=1)


```

```
### DO NOT MODIFY THE BELOW LINE
print("batch probability distributions:", batch_probs)
```

```
batch embedding tensor size:  torch.Size([2, 3, 10])
batch probability distributions: tensor([[0.5722, 0.3568, 0.0414, 0.0206, 0.0089],
        [0.5722, 0.3568, 0.0414, 0.0206, 0.0089]], grad_fn=<SoftmaxBackward0>)
```

## Question 4 (30 points)

Find one academic paper about long-context language models (use e.g., Semantic Scholar to search for relevant papers) that is of interest to you. Then, write a summary in your own words of the paper you chose. Your summary should answer the following questions: what is its motivation? Why should anyone care about it? How does it work? Were there things in the paper that you didn't understand at all? What were they? Fill out the below cell, and make sure to write 2-4 paragraphs for the summary to receive full credit!

**Title of paper**: Can Long-Context Language Models Subsume Retrieval, RAG, SQL, and More?

**Authors**: Jinhyuk Lee et.al

**URL**: https://www.semanticscholar.org/reader/9e320d3a55b5a05a0eff3c2b28cdc2d707346930

**Your summary**:

The paper **"Can Long-Context Language Models Subsume Retrieval, RAG, SQL, and More?"** by Jinhyuk Lee et al. explores whether **Long-Context Language Models (LCLMs)** can replace specialized retrieval systems, databases, and complex reasoning pipelines. Traditionally, AI models have relied on external retrieval tools and databases to process large amounts of information, but LCLMs—capable of handling contexts up to **millions of tokens**—offer an alternative approach. To evaluate this, the authors introduce **LOFT (Long-Context Frontiers Benchmark)**, which consists of six tasks spanning text, visual, and audio retrieval, retrieval-augmented generation (RAG), SQL-like reasoning, and many-shot in-context learning. The study assesses models like **Gemini 1.5 Pro, GPT-4o, and Claude 3 Opus** to determine how well LCLMs can perform these tasks compared to specialized systems.

The results show that **LCLMs can rival state-of-the-art retrieval models in some cases**, particularly in **text and multimodal retrieval**. However, they struggle with **multi-step compositional reasoning**, such as SQL-style querying, where structured data must be processed efficiently. A key insight from the study is that **prompting strategies significantly impact performance**—techniques like **Chain-of-Thought (CoT) reasoning** help LCLMs perform better when dealing with long-context queries. The authors also introduce **Corpus-in-Context (CiC) Prompting**, where entire corpora are placed within the model's context rather than relying on external retrieval. While this approach improves certain tasks, the study finds that **retrieval quality degrades when context lengths exceed 1 million tokens**, highlighting current scalability limitations.

One aspect of the paper that I found difficult to fully understand is **how LCLMs internally prioritize information within a long context window**. The study suggests that **the position of relevant documents within the context strongly affects retrieval performance**, but it is unclear why certain placements improve or degrade results. Does this mean LCLMs have implicit biases toward earlier sections of the input? If so, could reordering information within the prompt significantly change retrieval accuracy? Additionally, while the paper discusses LCLMs' ability to process structured data like SQL databases, it is not entirely clear whether these models can generalize across different database schemas without fine-tuning. Could LCLMs eventually replace formal query languages, or will they always require specialized adaptations for handling structured information?

Overall, this paper presents an **exciting but cautious** perspective on LCLMs, showing that while they can **replace some retrieval tasks**, they are **not yet a complete replacement for structured databases and complex reasoning systems**. The **LOFT benchmark** provides a strong foundation for evaluating LCLMs as they continue to scale, and future research will likely focus on improving their **efficiency, accuracy, and ability to handle structured reasoning** at scale.

## AI Disclosure

- Did you use any AI assistance to complete this homework? If so, please also specify what AI you used.
    - ChatGPT
    - Perplexity

---

*(only complete the below questions if you answered yes above)*

- If you used a large language model to assist you, please paste *all* of the prompts that you used below. Add a separate bullet for each prompt, and specify which problem is associated with which prompt.
    - Give me a tutorial in Pytorch Basic, while drawing parallels to tensorflow?
    - Give me a tutorial on NLP using Pytorch also draw parallels with tensorflow?
    - Summarize the research paper and be as detailed as possible
    - Check my assignment and grade it according to the marks given alongside it

- **Free response**: Describe your overall experience with the AI. How helpful was it? Did it just directly give you a good answer, or did you have to edit it? Was its output ever obviously wrong or irrelevant? Did you use it to get the answer or check your own answer?
    - I Have to edit it, the codes it was giving was correct but irrespective of models the output for some sample data was always wrong