CMSC 25700/35100 Winter 2026 Assignment 3
Attention and Prompting

Due: 11:59 PM CST, Friday, February 6, 2026

This assignment examines the classic attention mechanism, both in standard Transformers and as a general deep learning technique, and prompting techniques for modern large language models (LLMs). It covers math questions about mathematical properties of Transformer self-attention, and coding tasks for customized attention architectures as well as prompting LLMs. The whole assignment is adapted from various sources, including the CS224N: Natural Language Processing with Deep Learning course offered by Stanford University [9], and the Dive into Deep Learning textbook [10].

Here is a quick summary:

1. **Mathematical exploration of Transformer Attention:** What kinds of operations can self-attention easily implement? Why should we use fancier things like multi-headed self-attention? This section illustrates motivations of self-attention and Transformer networks mathematically.
2. **Attention as a General Deep Learning Technique:** Moving beyond standard self-attention in Transformers, you will gain some hands-on experience with designing and implementing a customized attention-based architecture that is tailored to natural language inference. You will also gain a deeper understanding that attention is a *general* deep learning technique that computes a selective summary of the values dependent on a query.
3. **Prompting Methods for Modern Language Models:** Besides mechanism-level understanding, you will eventually analyze the behaviors of a much more complex system, LLMs, which have demonstrated remarkable capabilities in learning from the information in user prompts without parameter updates. You will explore several common prompting methods (direct, chain-of-thought, and few-shot prompting), and how their effectiveness may vary between two different types of NLP tasks: classification and generation.

Submission Instructions. There are two types of questions in this assignment. Each question is annotated as either "written" or "coding" at the beginning of its specification.

For all the **written** questions, we have provided a latex-based report template, `a3_report_template.zip`. You should download the template files, typeset and render your answer in the provided `\ifans` command or the `solutionblock` environment after each question. The final compiled **PDF** report should be submitted to **`Assignment 3 - Written`** on GradeScope.

For all the **coding** questions, you should build your work on top of the provided source code by completing all the `TODOs` according to the specification of each question. After that, you should execute `a3_attention_snli_dist/test_a3_q2.py` and `a3_prompting_dist/test_a3_q3.py` to generate `A3-Q2.json` and `A3-Q3.json` respectively.

Eventually, you should zip together `a3_attention_snli_dist/` and `a3_prompting_dist/`, which contain both your completed source code and the newly generated .json files, into **a single .zip file**. This .zip file should be submitted to **`Assignment 3 - Code`** on GradeScope. NOTE that inside both folders, **the source code and data should be organized in the same file structure as they are originally distributed, and the json files should remain in their default locations after they are generated.** The submitted .zip file should have the following structure:

```
A3.zip
├── a3_attention_snli_dist/
│   ├── model.py (TODOs completed)
│   ├── train.py
│   ├── utils.py
│   ├── test_a3_q2.py
│   └── A3-Q2.json (generated)
└── a3_prompting_dist/
    ├── data/
    ├── prompting_gsm.py
    ├── prompting_gsm.sh
    ├── prompting_snli.py
    ├── prompting_snli.sh
    ├── utils.py (TODOs completed)
    ├── test_a3_q3.py
    └── A3-Q3.json (generated)
```

Important: Do not include any additional folder layer (e.g., `A3/`) between the `.zip` root and the two `_dist` folders. The two folders should be at the top level when the zip is extracted.

GPU Usage Instructions. For Section 2 and 3 of this assignment, you may need to use the GPU resources provided by Google Colab. Since we are not using jupyter notebooks this time, you will need to upload your code files to your Google Drive, and mount the drive to a Colab notebook in order to execute the files. We have provided a Colab notebook template for you to run your code. Please refer to the detailed instructions in that Colab notebook.

1. Mathematical Exploration of Transformer Attention (30 points)

We will start this assignment with simplified mathematical intuitions about the **standard multi-head self-attention mechanism in Transformers**. In this question, we will practice working with the self-attention equations, and motivate why multi-headed self-attention can be preferable to single-headed self-attention.

Recall that attention can be viewed as an operation on a *query* vector $\mathbf{q} \in \mathbb{R}^d$, a set of *value* vectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_n\}, \mathbf{v}_i \in \mathbb{R}^d$, and a set of *key* vectors $\{\mathbf{k}_1, \ldots, \mathbf{k}_n\}, \mathbf{k}_i \in \mathbb{R}^d$, specified as follows:

$$\mathbf{c} = \sum_{i=1}^{n} \mathbf{v}_i \alpha_i \tag{1}$$

$$\alpha_i = \frac{\exp(\mathbf{k}_i^\top \mathbf{q})}{\sum_{j=1}^{n} \exp(\mathbf{k}_j^\top \mathbf{q})} \tag{2}$$

with $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)$ termed the "attention weights". Note that the output $\mathbf{c} \in \mathbb{R}^d$ is an average over the value vectors weighted with respect to $\boldsymbol{\alpha}$.

(a) (5 points) **[written]: Copying in attention.** One advantage of attention is that it's particularly easy to "copy" a value vector to the output c. In this problem, we'll motivate why this is the case.

i. (2 points) The distribution $\boldsymbol{\alpha}$ is typically relatively "diffuse"; the probability mass is spread out between many different α_i. However, this is not always the case. **Describe** (in one sentence) under what conditions the categorical distribution $\boldsymbol{\alpha}$ puts almost all of its weight on some α_j, where $j \in \{1, \ldots, n\}$ (i.e., $\alpha_j \gg \sum_{i \neq j} \alpha_i$). What must be true about the query $\mathbf{q}$ and/or the keys $\{\mathbf{k}_1, \ldots, \mathbf{k}_n\}$?

ii. (3 points) Under the conditions you gave in (i), **describe** the output $\mathbf{c}$.

(b) (5 points) **[written]: An average of two.** Instead of focusing on just one vector $\mathbf{v}_j$, a Transformer model might want to incorporate information from *multiple* source vectors.

Consider the case where we instead want to incorporate information from **two** vectors $\mathbf{v}_a$ and $\mathbf{v}_b$, with corresponding key vectors $\mathbf{k}_a$ and $\mathbf{k}_b$. Assume that (1) all key vectors are orthogonal, so $\mathbf{k}_i^\top \mathbf{k}_j = 0$ for all $i \neq j$; and (2) all key vectors have norm 1. **Find an expression** for a query vector $\mathbf{q}$ such that $\mathbf{c} \approx \frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$, and **justify your answer**.* (Recall what you learned in part (a).)

(c) (10 points) **[written]: Drawbacks of single-headed attention:** In the previous part, we saw how it was *possible* for a single-headed attention to focus equally on two values. The same concept could easily be extended to any subset of values. In this question we'll see why it's not a *practical* solution.

Consider a set of key vectors $\{\mathbf{k}_1, \ldots, \mathbf{k}_n\}$ that are now randomly sampled, $\mathbf{k}_i \sim \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)$, where the means $\boldsymbol{\mu}_i \in \mathbb{R}^d$ are known to you, but the covariances Σ_i are unknown (unless specified otherwise in the question). Further, assume that the means $\boldsymbol{\mu}_i$ are all perpendicular; $\boldsymbol{\mu}_i^\top \boldsymbol{\mu}_j = 0$ if $i \neq j$, and unit norm, $\|\boldsymbol{\mu}_i\| = 1$.

i. (4 points) Assume that the covariance matrices are $\Sigma_i = \alpha I, \forall i \in \{1, 2, \ldots, n\}$, for vanishingly small α. Design a query $\mathbf{q}$ in terms of the $\boldsymbol{\mu}_i$ such that as before, $\mathbf{c} \approx \frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$, and provide a brief argument as to why it works.

ii. (6 points) Though single-headed attention is resistant to small perturbations in the keys, some types of larger perturbations may pose a bigger issue. In some cases, one key vector $\mathbf{k}_a$ may be larger or smaller in norm than the others, while still pointing in the same direction as $\boldsymbol{\mu}_a$.† As an example, let us consider a covariance for item a as $\Sigma_a = \alpha I + \frac{1}{2}(\boldsymbol{\mu}_a \boldsymbol{\mu}_a^\top)$ for vanishingly small α (as shown in figure 1). This causes $\mathbf{k}_a$ to point in roughly the same direction as $\boldsymbol{\mu}_a$, but with large variances in magnitude. Further, let $\Sigma_i = \alpha I$ for all $i \neq a$.

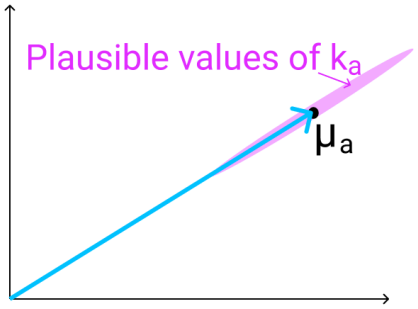

Figure 1: The vector $\boldsymbol{\mu}_a$ (shown here in 2D as an example), with the range of possible values of $\mathbf{k}_a$ shown in red. As mentioned previously, $\mathbf{k}_a$ points in roughly the same direction as $\boldsymbol{\mu}_a$, but may have larger or smaller magnitude.

When you sample $\{\mathbf{k}_1, \ldots, \mathbf{k}_n\}$ multiple times, and use the $\mathbf{q}$ vector that you defined in part i., what do you expect the vector $\mathbf{c}$ will look like qualitatively for different samples? Think about how it differs from part (i) and how $\mathbf{c}$'s variance would be affected.

(d) (6 points) **[written]: Benefits of multi-headed attention:** Now we'll see some of the power of multi-headed attention. We'll consider a simple version of multi-headed attention which is identical to single-headed self-attention as we've presented it, except two query vectors ($\mathbf{q}_1$ and $\mathbf{q}_2$) are defined, which leads to a pair of vectors ($\mathbf{c}_1$ and $\mathbf{c}_2$), each the output of single-headed attention given its respective query vector. The final output of the multi-headed attention is their average, $\frac{1}{2}(\mathbf{c}_1 + \mathbf{c}_2)$.

*Hint: while the softmax function will never *exactly* average the two vectors, you can get close by using a large scalar multiple in the expression.

†Unlike the original Transformer, some newer Transformer models apply layer normalization before attention. In these pre-layernorm models, norms of keys cannot be too different which makes the situation in this question less likely to occur.

As in question 1(c), consider a set of key vectors $\{\mathbf{k}_1, \ldots, \mathbf{k}_n\}$ that are randomly sampled, $\mathbf{k}_i \sim \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)$, where the means $\boldsymbol{\mu}_i$ are known to you, but the covariances Σ_i are unknown. Also as before, assume that the means $\boldsymbol{\mu}_i$ are mutually orthogonal; $\boldsymbol{\mu}_i^\top \boldsymbol{\mu}_j = 0$ if $i \neq j$, and unit norm, $\|\boldsymbol{\mu}_i\| = 1$.

i. (2 points) Assume that the covariance matrices are $\Sigma_i = \alpha I$, for vanishingly small α. Design $\mathbf{q}_1$ and $\mathbf{q}_2$ in terms of $\boldsymbol{\mu}_i$ such that $\mathbf{c}$ is approximately equal to $\frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$. Note that $\mathbf{q}_1$ and $\mathbf{q}_2$ should have different expressions.

ii. (4 points) Assume that the covariance matrices are $\Sigma_a = \alpha I + \frac{1}{2}(\boldsymbol{\mu}_a \boldsymbol{\mu}_a^\top)$ for vanishingly small α, and $\Sigma_i = \alpha I$ for all $i \neq a$. Take the query vectors $\mathbf{q}_1$ and $\mathbf{q}_2$ that you designed in part i. What, qualitatively, do you expect the output $\mathbf{c}$ to look like across different samples of the key vectors? Explain briefly in terms of variance in $\mathbf{c}_1$ and $\mathbf{c}_2$. You can ignore cases in which $\mathbf{k}_a^\top \mathbf{q}_i < 0$.

(e) (4 points) Based on part (d), briefly summarize how multi-headed attention overcomes the drawbacks of single-headed attention that you identified in part (c).

2. Attention as a General Deep Learning Technique (30 points)

In the previous question, we explored the mechanisms of standard transformer self-attention. Actually, attention is a general deep learning technique that can be applied to many architectures and many tasks. In this question, we will gain some hands-on experience with **designing and implementing a "non-standard" attention-based architecture for natural language inference**.

Recall that in Assignment 2 we have encountered the natural language inference (NLI) task, which studies whether a *hypothesis* can be inferred from a *premise*, where both are text sequences. NLI determines the logical relationship between a pair of text sequences. Such relationships fall into three types:

- **Entailment**: the hypothesis can be inferred from the premise.
- **Contradiction**: the negation of the hypothesis can be inferred from the premise.
- **Neutral**: all other cases.

We will continue using the Stanford Natural Language Inference (SNLI) Corpus [1] as Assignment 2 did, whose training set has about 550,000 labeled English sentence pairs and the test set has about 10,000 pairs. The three labels (entailment, contradiction, neutral) are balanced in both sets. We will also use the pretrained word embeddings from GloVe [7] to represent the input tokens in these text sequences, as we did in Assignment 2. We have provided a codebase to build your work on as follows. **All coding efforts inside this section assume `a3_attention_snli_dist/` as the root dir.**

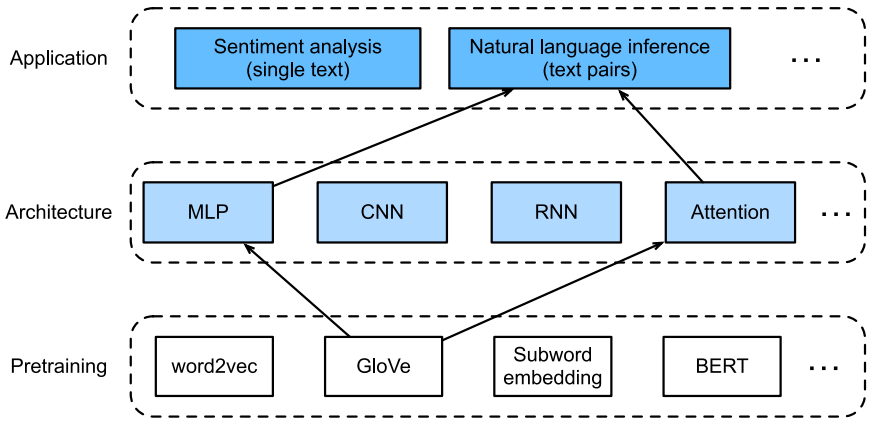

Figure 2: System Structure Overview

File Overview

We continue to use the same data loading and training utilities as Assignment 2, so that the performance obtained from this section can be compared with earlier results. Specifically, all the data loading utilities are reorganized into `utils.py`, where we load the SNLI dataset from Huggingface using `load_dataset("stanfordnlp/snli")`, and the 100-dimensional GloVe word embeddings via `gensim.downloader.load("glove-wiki-gigaword-100")`.

The complete training pipeline is also prepared in `train.py`, where you import the attention-based network class from `model.py`. That is to say, for this section, all your coding efforts go to `model.py`, and you should be able to correctly run `train.py` after implementing all the TODOs in `model.py`.

Design and Implementation of the Attention-based Model (18 points)

In view of many models that are based on complex and deep architectures, [6] proposed to address natural language inference with attention mechanisms and called it a "**decomposable attention** model". This results in a model **without** recurrent or convolutional layers, achieving the best result at the time on the SNLI dataset with much fewer parameters. In this section, we will describe and implement this attention-based architecture with simple MLPs for the NLI task.

Simpler than preserving the order of tokens in premises and hypotheses, we can just **align** tokens in one text sequence to **every** token in the other, and vice versa, then **compare** and **aggregate** such information to predict the logical relationships between premises and hypotheses. Similar to alignment of tokens between source and target sentences in machine translation, the alignment of tokens between premises and hypotheses can be neatly accomplished by attention mechanisms.

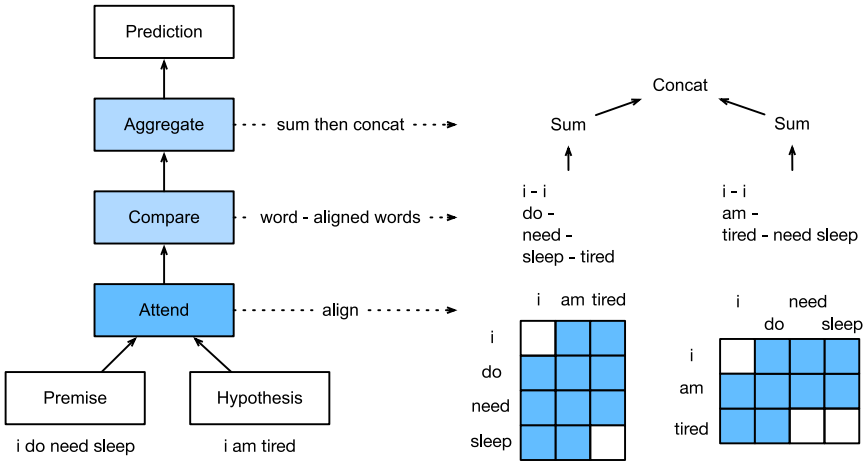

Figure 3: NLI Using Decomposed Attention

Figure 3 depicts the attention-based architecture with a running example. At a high level, it consists of three jointly trained steps: attending, comparing, and aggregating. We will illustrate and implement them step by step in the following.

Throughout this section, for all references of MLP function, you should use the provided `mlp` helper function with the right dimensions.

(a) (8 points) **[coding]: Attending**

As illustrated in Figure 3, we compute soft alignment between premise and hypothesis tokens by first computing pairwise attention weights between all token pairs, then using these weights to create

aligned representations. For each premise token, we compute a weighted representation of the hypothesis that captures which hypothesis words are most relevant. Similarly, for each hypothesis token, we compute a weighted representation of the premise that captures which premise words are most relevant. This bidirectional alignment allows the model to understand the relationship between the two sequences.

Input: Premise embeddings $\mathbf{A} \in \mathbb{R}^{m \times d}$ and hypothesis embeddings $\mathbf{B} \in \mathbb{R}^{n \times d}$, where m and n are sequence lengths and d is the embedding dimension.

Output: Aligned hypothesis representations $\boldsymbol{\beta} \in \mathbb{R}^{m \times d}$ where row $\boldsymbol{\beta}_i$ is the weighted average of hypothesis tokens aligned with premise token i, and aligned premise representations $\boldsymbol{\alpha} \in \mathbb{R}^{n \times d}$ where row $\boldsymbol{\alpha}_j$ is the weighted average of premise tokens aligned with hypothesis token j.

Algorithm:

$$E = f(\mathbf{A}) \cdot f(\mathbf{B})^\top \in \mathbb{R}^{m \times n}$$

$$\boldsymbol{\beta} = \text{softmax}(E) \cdot \mathbf{B} \in \mathbb{R}^{m \times d}$$

$$\boldsymbol{\alpha} = \text{softmax}(E^\top) \cdot \mathbf{A} \in \mathbb{R}^{n \times d}$$

where function f is an MLP function, and $f(\mathbf{A}) \in \mathbb{R}^{m \times d'}$, $f(\mathbf{B}) \in \mathbb{R}^{n \times d'}$, with d' being the MLP output dimension. First, apply MLP f to each token in $\mathbf{A}$ and $\mathbf{B}$ separately, then compute attention matrix E via matrix multiplication. Next, compute $\boldsymbol{\beta}$ where softmax is applied row-wise to E (over hypothesis tokens), so that each row i of $\boldsymbol{\beta}$ represents which hypothesis words are most relevant to premise word i. Finally, compute $\boldsymbol{\alpha}$ where softmax is applied row-wise to the transposed attention matrix $E^\top$ (over premise tokens), equivalently applying column-wise softmax to E, so that each row j of $\boldsymbol{\alpha}$ represents which premise words are most relevant to hypothesis word j.

Implement the `Attend` class in `model.py`. It would make your life significantly easier to keep track of the **tensor shape evolution** as you dive deeper into the implementation.

Hint: You may find `torch.bmm` useful for batched matrix multiplication when computing the attention-weighted sums.

(b) (2 points) **[coding]: Comparing**

We compare each token in one sequence with its aligned representation from the other sequence.

Input: Premise embeddings $\mathbf{A} \in \mathbb{R}^{m \times d}$, hypothesis embeddings $\mathbf{B} \in \mathbb{R}^{n \times d}$, aligned hypothesis representations $\boldsymbol{\beta} \in \mathbb{R}^{m \times d}$ from Attending Step B, and aligned premise representations $\boldsymbol{\alpha} \in \mathbb{R}^{n \times d}$ from Attending Step C.

Output: Comparison vectors $\mathbf{V}_A \in \mathbb{R}^{m \times d'}$ for premise and $\mathbf{V}_B \in \mathbb{R}^{n \times d'}$ for hypothesis, where d' is the MLP output dimension.

Algorithm:

$$\mathbf{V}_A = g([\mathbf{A}, \boldsymbol{\beta}]) \in \mathbb{R}^{m \times d'}$$

$$\mathbf{V}_B = g([\mathbf{B}, \boldsymbol{\alpha}]) \in \mathbb{R}^{n \times d'}$$

where function g is an MLP, and $[\cdot, \cdot]$ denotes concatenation along the feature dimension (resulting in dimension $2d$). The MLP g processes each concatenated token pair independently. Each row of $\mathbf{V}_A$ compares a premise token with its softly aligned hypothesis tokens, and each row of $\mathbf{V}_B$ compares a hypothesis token with its softly aligned premise tokens.

Implement the `Compare` class in `model.py`.

(c) (4 points) **[coding]: Aggregating**

We aggregate the comparison vectors from both sequences to produce the final classification logits.

Input: Comparison vectors $\mathbf{V}_A \in \mathbb{R}^{m \times d'}$ for premise and $\mathbf{V}_B \in \mathbb{R}^{n \times d'}$ for hypothesis (from Comparing).

Output: Classification logits $\hat{\mathbf{y}} \in \mathbb{R}^3$ representing scores for entailment, contradiction, and neutral.

Algorithm:

$$\mathbf{v}_A = \sum_{i=1}^{m} \mathbf{V}_{A,i} \in \mathbb{R}^{d'}$$

$$\mathbf{v}_B = \sum_{j=1}^{n} \mathbf{V}_{B,j} \in \mathbb{R}^{d'}$$

$$\hat{\mathbf{y}} = h([\mathbf{v}_A, \mathbf{v}_B]) \in \mathbb{R}^3$$

where function h is an MLP, and $\mathbf{V}_{A,i}$ denotes row i of $\mathbf{V}_A$. Apply sum pooling over the sequence dimension to aggregate comparison vectors into fixed-size representations $\mathbf{v}_A$ and $\mathbf{v}_B$, which are independent of the original sequence lengths m and n. Concatenate the two aggregated vectors (resulting in dimension $2d'$) and feed through MLP h to produce 3 output logits for the three NLI classes.

Implement the `Aggregate` class in `model.py`.

(d) (4 points) **[coding]: Putting them All Together**
In `model.py`, implement the `DecomposableAttention` class, which integrates all three steps with an embedding layer for input word vectors.

Training, Evaluation, and Analysis (12 points)

(a) (6 points) **[written]: Performance Report**
Run `train.py` to train and evaluate your model. Report your final test accuracy, along with the validation accuracy for each epoch.

(b) (6 points) **[written]: Analysis of Decomposable Attention**
Answer the following questions about the model architecture that you have just implemented.

i. (2 points) What is the most prominent purpose of introducing MLPs in this attention-based architecture?

ii. (4 points) What are the major drawbacks of the decomposable attention model for natural language inference? Consider aspects such as **what information might be lost or insufficiently encoded**, and **how it compares to canonical sequential models such as RNNs and Transformers**.

3. Prompting Methods for Modern Language Models (40 points)

In the previous two questions, we focused on *designing* simple attention-based architectures. For this question, we will shift our attention to **behavioral testing and analysis** of Large Language Models (LLMs). In recent years, LLMs have demonstrated remarkable capabilities in **In-Context Learning (ICL)**, where the model learns to solve a task given background information or a few examples ("demonstrations") in the prompt ("context") without updating its parameters. In this question, we will explore a variety of common prompting methods and their effectiveness across two types of NLP tasks: **classification** (SNLI) and **generation** (GSM-Symbolic). Specifically, you will explore:

- **Direct prompting**: Asking the model to produce the answer directly
- **Chain-of-thought (CoT) prompting** [4, 8]: Encouraging the model to reason step-by-step before giving the final answer
- **Few-shot prompting** [2, 8]: Providing example demonstrations to guide the model's responses

Through hands-on experiments, you will gain practical experience with prompting techniques and develop intuition for when different methods are most effective. We have provided a codebase to build your work on as follows. **All coding efforts inside this section use `a3_prompting_dist/` as the root directory.**

Task Descriptions and Basic Setup

Classification Task

We will continue using the SNLI dataset for classification. For this part, we provide:

- **Test set**: 100 examples uniformly sampled from the SNLI test split
- **Few-shot pool**: 200 examples from the training split for selecting few-shot demonstrations

Generation Task

GSM-Symbolic [5] is a recent dataset released by Apple as a variant of the well-known GSM8K benchmark [3]. GSM8K consists of grade-school level math word problems that require multi-step reasoning to solve. However, because GSM8K has been widely used and likely mixed into the training data of most modern LLMs, models may achieve high performance through memorization rather than genuine reasoning.

To address this limitation, GSM-Symbolic systematically perturbs the **numbers** and **natural language entities** (e.g., names, objects) in the original GSM8K problems while preserving the underlying mathematical structure. This allows us to test whether language models can truly reason about mathematical problems or are simply pattern-matching against memorized examples.

For this part, we provide:

- **Test set**: 100 examples uniformly sampled from the GSM-Symbolic test split
- **Few-shot pool**: 200 examples for selecting few-shot demonstrations

Data Files

All data files are provided in the `data/` folder and are ready for use:

- `snli_test_100.jsonl` — SNLI test set
- `snli_fewshot_pool_200.jsonl` — SNLI few-shot example pool
- `gsm_symbolic_test_100_student.jsonl` — GSM-Symbolic test set
- `gsm_symbolic_fewshot_pool_200_student.jsonl` — GSM-Symbolic few-shot example pool

Model and Decoding

Throughout this part, you will use `gemma-3-1b-it`, a modern, lightweight, and open-source instruction-tuned language model released by Google and available on HuggingFace. You will need to register a Huggingface account, review and accept Google's usage license on the homepage of `gemma-3-1b-it`, and create a User Access Token in order to download and use this model.

You will use **greedy decoding** for all experiments. Greedy decoding selects the most likely token at each step and thus produces deterministic outputs. We will cover decoding strategies in more detail in future lectures; for now, just treat greedy decoding as the default and most deterministic approach.

Chat Templates (6 points)

Modern instruction-tuned language models are trained on conversations formatted with specific templates that structure the interaction between user and assistant. These **chat templates** wrap the raw prompt text in special tokens or markers (e.g., `<start_of_turn>user`, `<end_of_turn>`) that the model learned to recognize during instruction tuning.

When prompting an instruction-tuned model, it is essential to apply the appropriate chat template. Using raw prompts without the expected formatting can lead to significantly degraded performance, as the model may not recognize that it should respond as an assistant.

HuggingFace tokenizers provide utilities to handle this formatting automatically. Learn more about chat templates in the HuggingFace documentation.

(a) (6 points) **[Coding]** Complete the `apply_chat_template` function in `utils.py` to apply the chat template to a prompt string.

Prompting Methods for Classification: SNLI (16 points)

(a) (6 points) **[Coding]** Complete the function `build_snli_prompt` in `utils.py` to implement the following three prompting methods for the SNLI task:

- 0-shot direct prompting
- 0-shot chain-of-thought prompting
- Few-shot direct prompting

Examples of their prompt templates are shown below. Lines starting with `//` are comments to help you understand the structure, and should not be treated as actual prompts. **You need to follow these templates exactly in order to receive full points.**

0-shot Direct Prompting for SNLI

```
// Direct prompting requires the model to output the final answer only.
You are given a premise and a hypothesis. Classify their relationship as one of: entailment,
neutral, or contradiction. Provide only the final label.

// Target question
Premise: A woman is chatting as she drinks her coffee.
Hypothesis: The woman is silent.
Answer: [TO BE COMPLETED BY MODEL]
```

0-shot Chain-of-Thought Prompting for SNLI

```
// Chain-of-thought prompting encourages the model to think step by step before giving the
final answer.
You are given a premise and a hypothesis. Classify their relationship as one of: entailment,
neutral, or contradiction, after thinking step by step.

// Target question, with ``Let's think step by step'' to trigger chain-of-thought reasoning.
Premise: A woman is chatting as she drinks her coffee.
Hypothesis: The woman is silent.
Answer: Let's think step by step. [TO BE COMPLETED BY MODEL]
```

Few-shot Direct Prompting for SNLI

```
// Few-shot prompting provides k examples as demonstrations before the target question.
```

```
You are given a premise and a hypothesis. Classify their relationship as one of: entailment,
neutral, or contradiction. Provide only the final label.

// Example 1 (randomly selected in your case)
Premise: A man wearing a white shirt with a black stripe in khaki pants posing in front of a
fence guarding a building and a red flag.
Hypothesis: The man is climbing the flag pole to get ontop of the building.
Answer: contradiction

// ...

// Example k (randomly selected in your case)
Premise: A big dog drinks as a little dog gets down off a deck.
Hypothesis: There are animals drinking.
Answer: entailment

// Target question
Premise: A woman is chatting as she drinks her coffee.
Hypothesis: The woman is silent.
Answer: [TO BE COMPLETED BY MODEL]
```

(b) (4 points) **[Written]** Experiment with these three prompting methods to evaluate their performance on the SNLI task. For few-shot direct prompting, experiment with 1, 2, 4, and 8 shots. For each shot count, randomly select 3 different sets of examples from the pool to observe variance in performance. Report your results in Table 1, using accuracy as the only metric on the percentage scale (i.e., filling numbers on a range from 0 to 100 without the percentage unit). Note that only one number should be put in each row for 0-shot performance, and three numbers should be put in each of the remaining rows.

Note that these functionalities are already implemented in the provided codebase. You only need to run the provided shell script `prompting_snli.sh` after completing the required functions in `utils.py`.

method		max	min	avg
0-shot	direct			
	cot			
1-shot	direct			
2-shot	direct			
4-shot	direct			
8-shot	direct			

Table 1: Performance of different prompting methods on the SNLI classification task.

(c) (6 points) **[Written]** Compare the performance of these prompting methods on SNLI:

1. How does 0-shot CoT prompting compare with 0-shot direct prompting?
2. How does few-shot direct prompting compare with 0-shot direct prompting?
3. What is the impact of the number of shots on performance? What variance do you observe across different example selections for few-shot prompting?

Prompting Methods for Generation: GSM-Symbolic (18 points)

(a) (6 points) **[Coding]** Similar to above, complete the function `build_gsm_prompt` in `utils.py` to implement the same suite of three prompting methods for the GSM-Symbolic task. Examples of their prompt templates are shown below. **You need to follow these templates exactly in order to receive full points.**

0-shot Direct Prompting for GSM-Symbolic

```
Solve the following math problem. Provide only the final numeric answer.

Problem: Jin saw a 45-foot dolphin with 18 9-inch remoras attached to it. But 1/3 of the
remoras go away. What percentage of the dolphin's body length is the combined length of the
remaining remoras?
Answer: [TO BE COMPLETED BY MODEL]
```

0-shot Chain-of-Thought Prompting for GSM-Symbolic

```
Solve the following math problem step by step, and provide the final numeric answer.

Problem: Jin saw a 45-foot dolphin with 18 9-inch remoras attached to it. But 1/3 of the
remoras go away. What percentage of the dolphin's body length is the combined length of the
remaining remoras?
Answer: Let's think step by step. [TO BE COMPLETED BY MODEL]
```

Few-shot Direct Prompting for GSM-Symbolic

```
Solve the following math problem. Provide only the final numeric answer.

// Example 1 (randomly selected in your case)
Problem: Sanjay saw a 60-foot dolphin with 16 12-inch remoras attached to it. But a quarter
of the remoras go away. What percentage of the dolphin's body length is the combined length
of the remaining remoras?
Answer: 20

// ...

// Example k (randomly selected in your case)
Problem: A juggler can juggle 800 balls. 1/4 of the balls are tennis balls, and 1/2 of the
tennis balls are indigo of which 1/10 are marked. How many marked indigo tennis balls are
there?
Answer: 10

// Target question
Problem: Jin saw a 45-foot dolphin with 18 9-inch remoras attached to it. But 1/3 of the
remoras go away. What percentage of the dolphin's body length is the combined length of the
remaining remoras?
Answer: [TO BE COMPLETED BY MODEL]
```

(b) (4 points) **[Written]** Similar to the experimental procedure for SNLI, experiment with these prompting methods to evaluate their performance on the GSM-Symbolic task, and report your results in Table 2 similarly.

Note: In addition to the three prompting methods above, there is a fourth method called **few-shot chain-of-thought prompting** [8] that combines few-shot examples with step-by-step reasoning.

	method	max	min	avg
0-shot	direct			
	cot			
1-shot	direct			
2-shot	direct			
4-shot	direct			
8-shot	direct			

Table 2: Performance of different prompting methods on the GSM-Symbolic generation task.

We do not require you to implement or experiment with it in this assignment, but you may optionally explore it for extra practice.

(c) (8 points) **[Written]** Compare the performance of these prompting methods on GSM-Symbolic:

1. How does 0-shot CoT prompting compare with 0-shot direct prompting?
2. How does few-shot direct prompting compare with 0-shot direct prompting?
3. How do your observations on GSM-Symbolic compare with those on SNLI? What might explain the differences in trends between the two tasks?

References

[1] Bowman, S. R., Angeli, G., Potts, C., and Manning, C. D. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (Lisbon, Portugal, Sept. 2015), Association for Computational Linguistics, pp. 632–642.

[2] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems 33* (2020), 1877–1901.

[3] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

[4] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners. *Advances in neural information processing systems 35* (2022), 22199–22213.

[5] Mirzadeh, S. I., Alizadeh, K., Shahrokhi, H., Tuzel, O., Bengio, S., and Farajtabar, M. GSM-symbolic: Understanding the limitations of mathematical reasoning in large language models. In *The Thirteenth International Conference on Learning Representations* (2025).

[6] Parikh, A., Täckström, O., Das, D., and Uszkoreit, J. A decomposable attention model for natural language inference. In *Proceedings of the 2016 conference on empirical methods in natural language processing* (2016), pp. 2249–2255.

[7] Pennington, J., Socher, R., and Manning, C. D. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)* (2014), pp. 1532–1543.

[8] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems 35* (2022), 24824–24837.

[9] Yang, D., and Hashimoto, T. Cs224n: Natural language processing with deep learning (winter 2025): Lecture materials. https://web.stanford.edu/class/cs224n/, 2025. Accessed: 2025-12-27.

[10] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. *Dive into Deep Learning*. Cambridge University Press, 2023. https://D2L.ai.