Natural Language Processing **Tel Aviv University**

Assignment 3: Transformers

Due Date: *March 1st EOD, 2025* Lecturer: Maor Ivgi, TA: Noa Mark

Instructions - Please Read

Questions 1 and 2 are mandatory. Additionally, you can choose to do question 3 as a bonus.

In this exercise, as well as in your final project, you will likely need to use HuggingFace's packages. Before you start, you watch Roi Cohen's tutorial (recorded in the fall semester). If you want to dive deeper, checkout their great course here.

1 Medical RAG

In this exercise, you will be asked to implement a **RAG** (Retrieval Augmented Generation) pipeline capable of answering users' questions about medical information.

Specifically, we will use MIRAGE (Medical Information Retrieval-Augmented Generation Evaluation) baseline to evaluate RAG performance on medical data.

You need to read the following tutorials:

1. Simple RAG for GitHub issues using Hugging Face Zephyr and LangChain
2. Advanced RAG on Hugging Face documentation using LangChain

And use their instruction to fill in the missing parts in this colab notebook.

Please copy this notebook to your drive, follow the instructions in the notebook, complete the missing parts, and run the cells. In the end of this exercise you should have a complete runnable notebook with the cell output nicely printed. Submit a copy of your notebook and include the theoretical answers in your PDF.

Preliminary: Complete the instruction in the 'setting' section.

1. When or why should you use RAG to access your data instead of fine-tuning?
2. Follow the instruction and complete the **Retriver** section. Load 'BAAI/bge-base-en-v1.5' embedding model from hugging face and initialize FAISS database (name it KNOWLEDGE_VECTOR_DATABASE).
3. Follow the instruction and complete the **Reader** section. Load 'HuggingFaceH4/zephyr-7b-beta model using BitsAndBytesConfig' as done in the turturial (name it model) and it's tokenizer.
4. Follow the instruction and complete the **Putting it all together** section. Define the rag-chain using langchain RunnableLambda, so it will get the query question and options and the context from the retriever.

5. Follow the instruction and complete the **Visualization** section. Visualize the search for the closest documents, of example_key, using PaCMAP (similar to Advanced RAG turturial)

   PaCMAP is high-demntionality visualization technique. We already seen example of using PCA for visualization in HW1 Q2. But there are many different visualization algorithms, such as t-SNE, UMAP and PaCMAP. Each algorithm preserve different properties of the data in the low demission. Generally speaking, t-SNE and UMAP preserve more local topology then PCA. (Fill free to read more about it.)

   Group according to the textbooks titles. What can you tell from the visualization?

   Include the question you visualize the search for and the plot in your PDF, as well as your full answer.

6. Now we have a simple pipeline for Medical QA using RAG. As you read in the tutorials, to improve our model we can tune the k hyper-parameter.

   - what is the maximal reasonable K for this case? (considering the chunk-size of our document and the max seq length of the model)
   - Is it always preferable to choose the largest k possible?

2 In-Context Learning

Through this question, you will experience using `GPT-2`, a transformer-based decoder only model, and explore its performance within a few and zero-shot learning settings. You will further compare its performance to that of `GPT-3`.

To evaluate that, we will use `TriviaQA` - a popular Question Answering benchmark, which is very commonly used to evaluate current LMs question answering abilities. This benchmark contains a few thousands of trivia questions, such as *"In which country is Lake Como?"* or *"In the British monarchy, who succeeded Queen Anne to the throne?"*, formulated as a multiple-choice question. We're though not gonna stick to this format, but apply our evaluation within an open domain closed-book questions. That is, we will rely on the model's current factual knowledge, to be extracted while attempting to answer a given question.

Please upload the `in_context.ipynb` file to google colab, and answer the following questions by filling in your code.

Important! Please enable a GPU in your Colab environment setting, and make sure you run your models on this.

1. **Random Demonstrations**. We're first gonna explore the performance given a random set of increasing sizes of in-context demonstrations.

   (a) `optional_in_context_demonstrations` is a list of 500 examples from `TriviaQA`. Each example is represented as a tuple, where the first entry is a question, as the second is a list of possible answers. Here you're required to randomly sample small sets of in-context examples, of sizes 3-8. Formally, you should have 6 different sets - the first one is of size 3, the second of size 4,

..., the last of size 8. Use each of these sets to construct an in-context prompt. This prompt should be phrases as follows:

Question: ¡first question¿
Answer: ¡one of its possible answers¿
...
Question: ¡last question¿
Answer: ¡one of its possible answers¿

Now, evaluate the performance of `GPT2-medium`, using each one of these prompts, on the validation set (`validation_set`), by using the function `check_answer_truthfulness`. In this section, please apply beam search decoding (`beam_search` function).

Please report the results (correctness average over the whole validation set), using `pandas.Dataframe`. You just have to print it :)

(b) Now apply the same process, but using sampling decoding (with temperature = 0.7). Try to explain the differences in results in two sentences.

2. **Demonstrations Retrieval**. In this section, we will retrieve the examples that are most "semantically" similar to the test question, out of `optional_in_context_demonstrations`, according to `BERT`.

   (a) Use `encode_question` to first encode the questions of all the examples in `optional_in_context_demonstrations`. Then, for each of the examples in the validation set, find the closest 8 examples in `optional_in_context_demonstrations` - here our measure to similarity is the dot product between the encoding vector of the test question, to this of the "train" question. Note that now, each of our validation examples will be accompanied with a different prompt. All that left is to evaluate and report the new results - both with beam search, and sampling decoding.

   (b) Explore a batch of examples from the validation set, and their new prompts. Can you explain why results have got better compared to the those of the random prompts from previous sections?

3. **Zero-shot**. Now we will try a zero-shot setting, where we don't give the model any demonstrations, but directly asking it the question.

   (a) Evaluate and report the results on the given validation set.

   (b) Have a look on the model's generations from the last section. Try to explain why the results are that low.

   (c) In order to solve this, we will use `LAMA`, which is a known sentence-completion benchmark, contains of many facts taken from Wikipedia. In This benchmark, the query is phrased as a fill-in-the-blank sentence, where the missing text is the target answer. For instance, *"It is found in north-western Russia, Ukraine, Romania, Bulgaria, Greece and"*, while the answer is *"Italy"*. Evaluate and report the results on `lama_validation_set`, which is a list of tuples composed the same way as the validation set of `TriviaQA`.

   (d) Do you think the results are actually higher than those you reported? Take a look on the model's generations and try to explain.

3 Pretrained Transformers (Bonus Question)

In this section you'll be training a Transformer to perform a task that involves accessing knowledge about the world — knowledge which isn't provided via the task's training data (at least if you want to generalize outside the training set). You'll find that it more or less fails entirely at the task. You'll then learn how to pretrain that Transformer on Wikipedia text that contains world knowledge, and find that finetuning that Transformer on the same knowledge-intensive task, might be helpful for enabling the model to access some of the knowledge learned at pretraining time. You'll find that this enables models to perform considerably OK on a held out validation set.

Through this section, you will fill in your code within the `pretrained_transformers` directory, and write your textual answers within the pdf file. You will probably want to develop on your machine locally, then run training on Slurm/Colab. You'll also probably need a few hours for the training, so plan your time accordingly!

1. **Check out the demo.**
   In the `mingpt-demo/` folder is a Jupyter notebook `play_char.ipynb` that trains and samples from a Transformer language model. Take a look at it (either locally on your computer or on Colab) to get somewhat familiar with how it defines and trains models. Some of the code you're writing below will be inspired by what you see in this notebook.

   Note that you do not have to write any code or submit written answers for this part.

2. **Read through `NameDataset` in `src/dataset.py`, our dataset for reading name-birthplace pairs.**
   The task we'll be working on with our pretrained models is attempting to access the birth place of a notable person, as written in their Wikipedia page. We'll think of this as a particularly simple form of question answering:

   Q: Where was [person] born?
   A: [place]

   From now on, you'll be working with the `src/` folder. **The code in `mingpt-demo/` won't be changed or evaluated for this assignment.** In `dataset.py`, you'll find the the class `NameDataset`, which reads a TSV (tab-separated values) file of name/place pairs and produces examples of the above form that we can feed to our Transformer model.

   To get a sense of the examples we'll be working with, if you run the following code, it'll load your `NameDataset` on the training set `birth_places_train.tsv` and print out a few examples.

   ```
   python src/dataset.py namedata
   ```

   Note that you do not have to write any code or submit written answers for this part.

3. **Implement finetuning (without pretraining).**
   Take a look at `run.py`. It has some skeleton code specifying flags you'll eventually need to handle as command line arguments. In particular, you might want to *pretrain*, *finetune*, or *evaluate* a model with this code. For now, we'll focus on the finetuning function, in the case without pretraining.

   Taking inspiration from the training code in the `play_char.ipynb` file, write code to finetune a Transformer model on the name/birthplace dataset, via examples from the `NameDataset` class. For now, implement the case without pretraining (i.e. create a model from scratch and train it on the birthplace prediction task from part (b)). You'll have to modify two sections, marked `[part c]` in the code: one to initialize the model, and one to finetune it. Note that you only need to initialize the model in the case labeled "vanilla" for now (later in section (g), we will explore a model variant). Use the hyperparameters for the `Trainer` specified in the `run.py` code.

Also take a look at the *evaluation* code which has been implemented for you. It samples predictions from the trained model and calls `evaluate_places()` to get the total percentage of correct place predictions. You will run this code in part (d) to evaluate your trained models.

This is an intermediate step for later portions, including Part 4, which contains commands you can run to check your implementation. No written answer is required for this part.

4. **Make predictions (without pretraining).**
Train your model on `birth_places_train.tsv`, and evaluate on `birth_dev.tsv`. Specifically, you should now be able to run the following three commands:

```
# Train on the names dataset
python src/run.py finetune vanilla wiki.txt \
        --writing_params_path vanilla.model.params \
        --finetune_corpus_path birth_places_train.tsv

# Evaluate on the dev set, writing out predictions
python src/run.py evaluate vanilla wiki.txt  \
        --reading_params_path vanilla.model.params \
        --eval_corpus_path birth_dev.tsv \
        --outputs_path vanilla.nopretrain.dev.predictions

# Evaluate on the test set, writing out predictions
python src/run.py evaluate vanilla wiki.txt  \
        --reading_params_path vanilla.model.params \
        --eval_corpus_path birth_test_inputs.tsv \
        --outputs_path vanilla.nopretrain.test.predictions
```

Training should take less than 30 minutes. Report your model's accuracy on the dev set (as printed by the second command above). Similar to assignment 4, we also have Tensorboard logging in assignment 5 for debugging. It can be launched using `tensorboard --logdir expt/`. Don't be surprised if it is well below 10%; we will be digging into why in Part 3. As a reference point, we want to also calculate the accuracy the model would have achieved if it had just predicted "London" as the birth place for everyone in the dev set. Fill in `london_baseline.py` to calculate the accuracy of that approach and report your result in your write-up. You should be able to leverage existing code such that the file is only a few lines long.

5. **Define a *span corruption* function for pretraining.**
In the file `src/dataset.py`, implement the `__getitem__()` function for the dataset class `CharCorruptionDataset`. Follow the instructions provided in the comments in `dataset.py`. Span corruption is explored in the T5 paper. It randomly selects spans of text in a document and replaces them with unique tokens (noising). Models take this noised text, and are required to output a pattern of each unique sentinel followed by the tokens that were replaced by that sentinel in the input. In this question, you'll implement a simplification that only masks out a single sequence of characters.

This question will be graded via autograder based on whether your span corruption function implements some basic properties of our spec. We'll instantiate the `CharCorruptionDataset` with our own data, and draw examples from it.

To help you debug, if you run the following code, it'll sample a few examples from your `CharCorruptionDataset` on the pretraining dataset `wiki.txt` and print them out for you.

```
python src/dataset.py charcorruption
```

No written answer is required for this part.

6. **Pretrain, finetune, and make predictions. Budget 2 hours for training.**
Now fill in the *pretrain* portion of `run.py`, which will pretrain a model on the span corruption task. Additionally, modify your *finetune* portion to handle finetuning in the case *with* pretraining. In particular, if a path to a pretrained model is provided in the bash command, load this model before finetuning it on the birthplace prediction task. Pretrain your model on `wiki.txt` (which should take approximately two hours), finetune it on `NameDataset` and evaluate it. Specifically, you should be able to run the following four commands: (Don't be concerned if the loss appears to plateau in the middle of pretraining; it will eventually go back down.)

```
# Pretrain the model
python src/run.py pretrain vanilla wiki.txt \
        --writing_params_path vanilla.pretrain.params

# Finetune the model
python src/run.py finetune vanilla wiki.txt \
        --reading_params_path vanilla.pretrain.params \
        --writing_params_path vanilla.finetune.params \
        --finetune_corpus_path birth_places_train.tsv

# Evaluate on the dev set; write to disk
python src/run.py evaluate vanilla wiki.txt  \
        --reading_params_path vanilla.finetune.params \
        --eval_corpus_path birth_dev.tsv \
        --outputs_path vanilla.pretrain.dev.predictions

# Evaluate on the test set; write to disk
python src/run.py evaluate vanilla wiki.txt  \
        --reading_params_path vanilla.finetune.params \
        --eval_corpus_path birth_test_inputs.tsv \
        --outputs_path vanilla.pretrain.test.predictions
```

Report the accuracy on the dev set (printed by the third command above). We expect the dev accuracy will be at least 10%, and will expect a similar accuracy on the held out test set.

7. succinctly explain why the pretrained (vanilla) model was able to achieve an accuracy of above 10%, whereas the non-pretrained model was not

Acknowledgements This question was adapted from Stanford's CS224n course. Their contributions are greatly appreciated.