COL 774: Machine Learning. Assignment 4

Due Date: 11:59 pm, Tuesday, 25th November, 2025. Total Points: 100

Notes:

- You should submit all your code as well as any graphs that you might plot.
- Submit all the code in .py files and all answers to theory questions; all graphs should be submitted in a PDF. Do not submit any thoery in the code files.
- Do not submit the datasets.
- Include a **single write-up (pdf) file** which includes a brief description for each question explaining what you did. Include any observations and/or plots required by the question in this single write-up file.
- You should use Python for all your programming solutions.
- Your code should have appropriate documentation for readability. This will influence your marks. Demo will also be conducted in which you will have to explain your code
- You will be graded based on what you have submitted as well as your ability to explain your code.
- The readability and presentation in your report will also carry marks. While there is no specific format, you are advised to be clear, concise and complete in your answers with explanations wherever required
- Refer to the course website for assignment submission instructions.
- This assignment is supposed to be done individually. You should carry out all the implementation by yourself.
- Your assignment will be run on autograder scripts. Detailed instructions regarding this and submission guidelines of code will follow soon. Till then, please start working on writing the functions for your code.
- We plan to run Moss on the submissions. We will also include submissions from previous years since some of the questions may be repeated. Any cheating will result in a zero on the assignment, an addtional penalty of the negative of the total weightage of the assignment and possibly much stricter penalties (including a **fail grade** and/or referring to a **DisCo**). Please note that using any **AI tools is not allowed**. If the software detects any plagiarism due to it, this will be penalised equivalently.
- Many of the problems below have been adapted from the Machine Learning course offered by Andrew Ng at Stanford.
- For all the parts, the allowed libraries are pandas, numpy, matplotlib, argparse, and sys, along with any other library specifically for plotting. Please note that sklearn or scikit-learn is not allowed. If there are any confusion on whether a library is allowed, feel free to ask on Piazza. We may penalize submission which uses a library outside the above permitted libraries and those approved on piazza.

1 (100 points) Learning to Solve Mazes

1.1 Overview

In this assignment, you'll explore how sequence to sequence models can be used to solve spatial reasoning tasks - specifically, predicting the path through a maze given the start and end points. This assignment is inspired by the recent work, ***Structured World Representations in Maze-Solving Transformers*** (https://arxiv.org/pdf/2312.02566). We will first describe the dataset, task specification, and then specific implementation requirements.

1.2 Dataset Details

A Maze can be seen as a 2D-grid of cells with coordinates `(x,y)`. An edge is defined between two adjacent grid cells as `(x_1,y_1) <--> (x_2,y_2)`. In our representation, any two adjacent grid cells with no edge between them are considered to be separated by a wall. Accordingly, we define the maze structure as a sequence of edges separated by `;`. Given the maze-structure (adjacency list) along with an origin and a target grid-cell, the goal is to predict the sequence of grid coordinates (**the path**) that connects between the two. The data consists of two kinds of mazes, forked and fork-less. Forked mazes contain multiple branches and require choice/decision points to find the single optimal path, whereas fork-less mazes offer a unique, unambiguous path between start and end-point. An example of 3x3 tokenized input and output sequence is given below:

```
<ADJLIST_START> (1,2) <--> (0,2) ; (1,2) <--> (1,1) ; (0,0) <--> (1,0) ; (2,1) <-->
(2,2) ; (1,2) <--> (2,2) ; (0,1) <--> (1,1) ; (2,1) <--> (2,0) ; (2,0) <--> (1,0) ;
<ADJLIST_END>
<ORIGIN_START> (1,2) <ORIGIN_END>
<TARGET_START> (0,0) <TARGET_END>
<PATH_START>
```

Output:

```
(1,2), (2,2) (2,1), (2,0), (1,0), (0,0) <PATH_END>
```

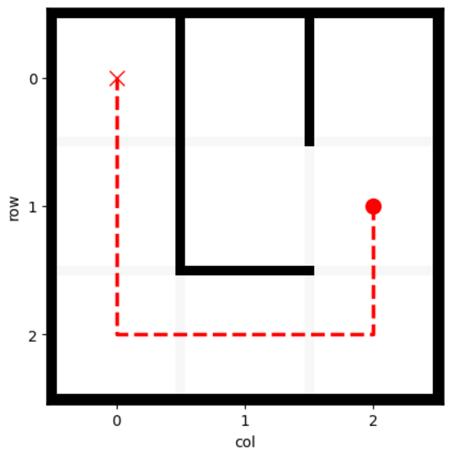

Figure 1: Example of 3x3 forked maze. ● represents starting point, × marks the end.

In our problem, we'll deal with the following input-output tokens:

1. **Coordinate Tokens:** Strings like `'(1,2)'`, `'(2,0)'`, representing grid locations. There's a unique token for each grid-cell.
2. **Special Tokens:**
   - **Maze Structure Tokens:** `<ADJLIST_START>`, `<-->`, `;`, `<ORIGIN_START>`, `<TARGET_END>`, etc., which collectively define the maze connectivity, the starting point (Origin), and the destination (Target)

- **Sequence Delimiters:** `<PATH_START>` (serves as the boundary between the description of the problem and the solution path), `<PATH_END>` (marks the end of the correct path).

The data for the assignment is available here. You are provided with a collection of total 100K (forked and fork-less) **6x6 mazes** split across train and test set. If required, you may further split the train set, to get a slightly smaller training set, and a separate validation set. **We are also planning to provide you an unseen test set to evaluate your models at a later stage**. The CSV file contains columns for `input_sequence`, `output_path` and `maze_type` stored as strings. An additional python file is also provided to help you visualize mazes with ground-truth/predicted paths. The input and output in sequences in the CSV files are valid lists stored as strings. You can use python's inbuilt `eval()` function to obtain the list of tokens, an example is provided in `visualize.ipynb` notebook.

1.3 Task Definition

Your task is to build two models that takes the maze connectivity, origin and target grid cells as input and predicts the path between them. Specifically,

- Implement an RNN-based sequence2sequence with attention.
- Implement a Transformer based model for the same task and compare the two.

2 (50 points) Recurrent Neural Network (RNN)

2.1 Introduction to Recurrent Neural Networks (RNN) - with Attention

Recurrent Neural Networks (RNNs) are sequential neural networks where connections between nodes form a directed chain, enabling them to process temporal data. For the sequence-to-sequence task like ours, an RNN is typically configured as an **Encoder-Decoder** model as done in the class.

- The **Encoder** reads the entire Input sequence (**X**), which contains the maze's structural description, as well as the start and end point of the desired path. It processes this sequence token-by-token and compresses all the information into a single, fixed-size vector known as the **context vector**(the final hidden state).
- The **Decoder** takes this fixed context vector as its latent representation and sequentially generates the output sequence (**Y**) the optimal path.
- We'll follow the implementation of RNN with **Bahdanau Attention** as described in this paper (`https://arxiv.org/pdf/1409.0473`). Specifically, refer to section 1.2 and section 2 of the Appendix.

We will use an **Embedding** layer, which is a learnable matrix used to convert high-dimensional input data like words/sentences into low-dimensional vectors that can be easily processed by neural-network. In PyTorch, an embedding layer can be implemented using the nn.Embedding module, which takes as input a tensor of indices representing the words or tokens in the input text, and outputs a tensor of embedding vectors.

2.2 Task: Implement and Train the RNN with Attention Pathfinding Model

Implement an Encoder-Decoder RNN with Attention using PyTorch's `nn.RNN` class in pytorch with the following hyper-parameters. Additionally implement the Bahdanau Attention with a one-hidden layer MLP as described in the paper. In the decoder of RNN, we'll follow eq(3) of the Bahdanau paper (see more details below).

```
BATCH-SIZE = 32
EPOCHS = 20
LEARNING-RATE = 1e-4
EMBEDDING-DIMENSION = 128
HIDDEN-DIMENSION = 512
NUMBER-OF-RNN-LAYERS = 2
OPTIMIZER = ADAM
```

Points to note:

1. **Model Implementation:** Since the maze description and outputs vary in length for each sample. You must think about handling variable-length input sequences, read about techniques like padding, masking, or sequence packing during training. Use the Cross-Entropy Loss, ensuring padding tokens are correctly ignored.

2. **Training Setup:** Recall we used Eq (3) Specifically we pass the y_{i-1} (i.e., output of the previous timestep) to the decoder for generating current timestep's output. There are two ways to do this, one is to pass the generated token ($\hat{y_{i-1}}$) (by taking an argmax over the generated probability vector) or use the ground-truth token (y_{i-1}), both at the previous time-step. Using the ground-truth token is called teacher-forcing. In each case, we use the corresponding embedding vector as input to the decoder at the current time step. You should use Teacher-forcing during training to help stabilize learning. A typical value for teacher-forcing ratio is `0.5`, i.e, half the training batches (randomly chosen) are trained with teacher forcing, and other half without it.

3. **Evaluation metric:** We'll focus on two primary metrics. **[1] Sequence Accuracy(Exact match)** i.e output is correct iff the entire generated path matches the ground-truth path. **[2] F1-Score over Predicted Tokens** to measure partial matches, read about F1-here here, also refer to A3.

Report and Analysis

- Plot the loss and accuracy (both token and sequence) curves for both training/val/test sets across epochs.
- Visualize predictions for any 5 random mazes in the validation set.

3 (50 points) Transformer Architecture

3.1 Introduction of the Transformer Architecture

The Transformer architecture, introduced in the paper **"Attention Is All You Need"** (`https://arxiv.org/pdf/1706.03762`) (please read this carefully), entirely replaces recurrence with **Attention Mechanisms**. The Transformer implements a full **Encoder-Decoder Transformer** architecture, utilizing multi-head attention to model dependencies regardless of the distance between tokens, making it well-suited for tasks involving long-range reasoning in structured data like mazes.

- **Encoder Stack (Self-Attention):** Takes Input $\mathbf{X}$ (the maze description, start and end points). Allows every token in the input to attend to all other tokens in $\mathbf{X}$, creating a rich, contextual representation of the input.
- **Decoder Stack (Masked Self-Attention and Cross-Attention):** Takes Target $\mathbf{Y}$ (shifted [1]).
  - **Masked Self-Attention:** Maintains the auto-regressive nature by only allowing the current prediction to attend to previous path tokens.
  - **Cross-Attention:** Critically, this allows the decoder to attend to the entire Encoder's representation enabling it to selectively focus on the most relevant parts of the maze description (e.g., walls, forks, goal location) when generating the next move.
- **Embedding Layer** similar to an RNN, an embedding layer is used to convert the token indices into a vector-representation. One key point to note is that, this matrix is shared between both the encoder and decoder. Refer to section (3.4) of the paper.
- **Positional Encoding:** These are added to the token-embeddings to incorporate sequential order information (since the model has no inherent ordering now due to attention only architecture). Refer to section (3.5) of the paper to learn about positional embeddings. Specifically, we will work with sinusoidal positional embedding where each dimension of the positional encoding corresponds to a sinusoid with a different wavelength (related via a geometric progression). Refer to the relevant sections in the paper for more details.

[1]this is done to enforce causal dependencies in time, read Section 3.2.3 in the Transformer paper, and if required you should explore/read more about this from additional sources

3.2 Task: Implement and Train the Transformer Pathfinding Model

Implement and train a Transformer-based Encoder-Decoder model using **PyTorch's** available classes like, `nn.TransformerEncoderLayer`, `nn.TransformerEncoder`, `nn.TransformerDecoderLayer`, `nn.TransformerDecoder` with the following hyper-parameters

```
D-MODEL = 128
NHEAD = 8
NUM-LAYERS = 6
DIM-FEEDFORWARD = 512
DROPOUT = 0.1
```

Other hyper-params remains the same the the RNN part.

Points to note:

1. **Model Implementation:** Carefully implement shifting of output sequence to enforce causal attention. You must correctly handle Positional Encoding.

Report and Analysis

- Plot the same loss and accuracy curves as done for RNN.
- Visualize and compare the Transformer's performance with RNN based model on randomly selected 5 mazes from validation set.

4 Assignment Scoring

Note that in this assignment your score will significantly depend not only on the correctness of your implementation but also on overall-accuracy obtained on the (un)seen test set. Additional bonus points (up to 10%) will also be awarded to submissions which perform particularly well on the (un)seen test.

5 Submission Instructions

1. You must submit all your code as ***.py*** files.
2. The .zip file must be named as `ENTRY_NO.zip` eg. `2022AIZ8170.zip`, unzipping it should create a directory with your entry number containing all the required files.
3. You must submit an **eval.py** which should take two command-line arguments, [**1**] The path to your pre-trained model, [**2**] The path to a .txt file containing an example input of a maze. Upon execution it must print and visualize the predicted path by your model.