STEFANO ANDREOTTI

Heuristic for TSP Using Transformers

Dataset (20 pts)

- **Load the dummy dataset**
  The single data item retrieved from the dummy dataset is a tuple containing two elements:
  - A `networkx.Graph` object representing the graph structure.
  - A list representing the tour: `[0, 3, 14, 2, 9, 6, 19, 13, 12, 16, 7, 18, 8, 17, 5, 11, 10, 15, 1, 4, 0]`.

  The Python type of the single data item is `tuple`.
- **Describe the edge attributes `tour` and `weight`**
  The dataset contains the following attributes:
  - Node attribute `pos`: this attribute stores the 2D coordinates of each node in the graph. For example the node 0 is: "Node 0: `(0.6049077053425551, 0.5748590937018008)`"
  - Edge attribute `weight`: This attribute stores the Euclidean distance between two nodes (the weight of the edge). For example the first edge weight is: "Edge (0, 1): `0.4287846201876535`"
  - Edge attribute `tour`: This attribute indicates whether the edge is part of the optimal tour (1 if it is, 0 otherwise). For example the first edge attribute part of the optimal tour is: "Edge (0, 3): `1`"
- **Implement a dataset class**
  The dataset class `TSPDataset` is implemented to return:
  - `X`: A tensor of node coordinates with size 20×2.
  - `y`: A tour starting from 0 and ending with 0, represented as a tensor of node indices.
- **Create Dataset objects**
  The dataset objects for training, validation, and testing areloaded from data files and then are created train and valid loader for the training part while the test loader is not needed:
  - `train_loader`: DataLoader for the training dataset, with batch size of 32.
  - `valid_loader`: DataLoader for the validation dataset, with batch size of 32.
  - `test_dataset`: Data for testing the model.

Model (35 pts)

- **Implement the transformer-based architecture.**
  The transformer-based architecture is designed to solve the Traveling Salesman Problem (TSP) by encoding the graph structure and decoding the optimal tour. The architecture was creating starting from the exercise about transformers, it consists of two main components: an encoder and a decoder.

  - **Encoder**: The encoder processes the node coordinates (2D positions) of the graph. Each node's coordinates are projected into a higher-dimensional space using a linear layer. The encoder then applies multiple layers of self-attention to capture the relationships between nodes. Since the node coordinates already contain spatial information, positional encoding is not required in the encoder.
  - **Decoder**: The decoder generates the tour sequentially, one node at a time. It uses masked self-attention to ensure that the model only attends to previously visited nodes, preventing it from "cheating" by looking at future nodes in the sequence. Positional encoding is added to the decoder's input to provide information about the order of nodes in the tour. The decoder also attends to the encoder's output (memory) to incorporate information about the graph structure.
  - **Final Layer**: The output of the decoder is passed through a feed-forward neural network (FFNN) to predict the next node in the tour. The FFNN outputs a probability distribution over all nodes, and the node with the highest probability is selected as the next step in the tour.

  The model is designed to handle graphs with 20 nodes, as specified in the assignment. The architecture and can be adapted to larger graphs by adjusting the input dimensions and the number of layers.

- **Explain where and why masking is used.**
  Masking is used in the decoder to prevent the model from attending to future nodes in the sequence. This is crucial for autoregressive tasks like TSP, where the model must generate the tour step by step. Without masking, the model could "cheat" by looking at future nodes, which would violate the sequential nature of the problem.

  Specifically:

  - A causal mask is applied to the decoder's self-attention mechanism. This mask ensures that each node in the sequence can only attend to itself and the nodes that come before it in the sequence.
  - Masking is not required in the encoder because the encoder processes all nodes simultaneously and does not generate a sequence.

- **Explain why positional encoding may be omitted in the encoder.**
  Positional encoding may be omitted in the encoder because the input to the encoder (node coordinates) already contains spatial information. The relative positions of the nodes are implicitly encoded in their coordinates, so additional positional encoding is not necessary.

- **Explain why positional encoding is necessary in the decoder.** Positional encoding is necessary in the decoder because the decoder generates the tour sequentially, one node at a time. Without positional encoding, the decoder would have no information about the order in which nodes are visited, which is critical for solving TSP. Positional encoding provides the decoder with information about the position of each node in the sequence.

Training (25 pts)

Standard Training (10 pts)

The model was trained for a maximum of approximately 10 minutes using the standard training procedure (without gradient accumulation). The training and validation losses were recorded at regular intervals of 3 epochs which is useful for having a light output and also always keep track of the loss. The results are as it follows: at epoch 3, the training loss was `2.019`, the validation loss was `2.002`. At epoch 6, the training loss was `1.994`, the validation loss was `1.984`. At epoch 9, the training loss was `1.986`, the validation loss was `1.984`. Training stopped after reaching the 10-minute time limit at epoch 10, with a total training time of `11.88 minutes`. The best model was obtained at epoch 10 with a training loss of `1.984` and a validation loss of `1.983`.
The model does not show signs of overfitting because the training and validation losses remain close to each other throughout the training process, as shown in the figure 1 that report the training and validation loss. This indicates that the model is generalizing well to unseen data. The small gap between the training and validation losses suggests that the model is not memorizing the training data but rather learning meaningful patterns that apply to the validation set.

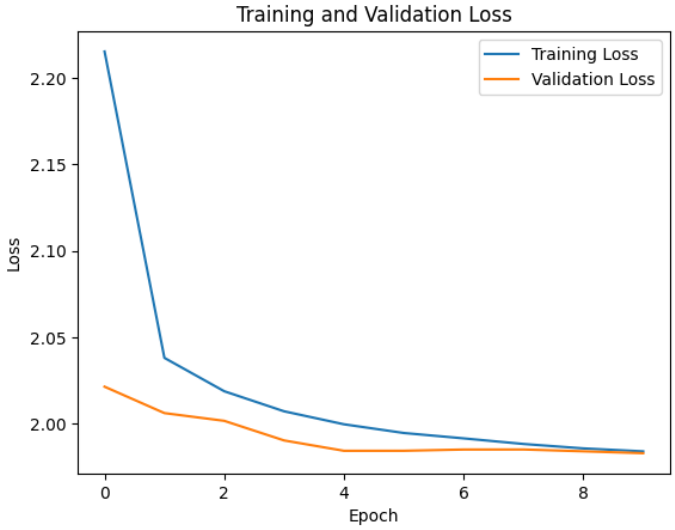

Figure 1: Training and validation loss for standard training

Training with Gradient Accumulation (15 pts)

The model was also trained using gradient accumulation, which allows for effective training with larger batch sizes by accumulating gradients over multiple smaller batches. The results are as follows: at epoch 3, the training loss was `2.038`, the validation loss was `2.004`. At epoch 6, the training loss was `2.009`, the validation loss was `1.992`. At epoch 9, the training loss was `1.997`, the validation loss was `1.985`. Training stopped after reaching the 10-minute time limit at epoch 10, with a total training time of `10.86 minutes`. The best model was obtained at epoch 10 with a training loss of `1.996` and a validation loss of `1.985`.
Similar to the standard training procedure, the model trained with gradient accumulation does not show signs of overfitting, as shown in figure 2. The training and validation losses remain close to each other, indicating good generalization. The use of gradient accumulation helps stabilize training by reducing the variance of gradient updates, which can lead to better convergence.

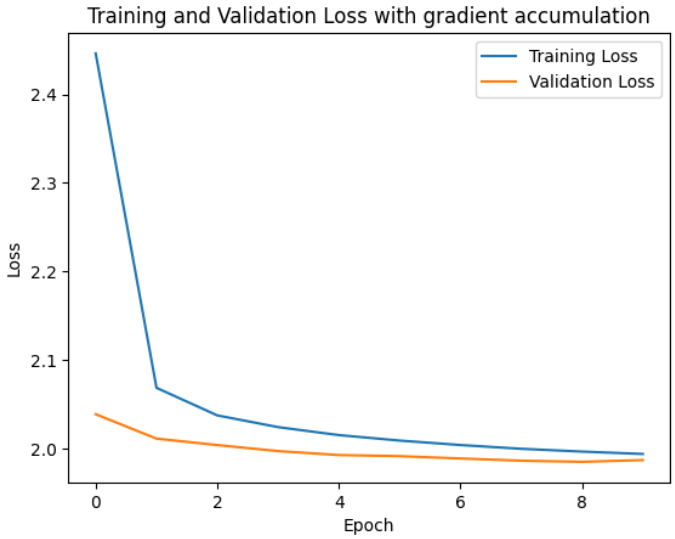

Figure 2: Training and validation loss for gradient accumulation training

Testing (15 pts)

- **Explain the function `transformer_tsp`**
  The `transformer_tsp` function is used to evaluate the trained model on a given graph, it is used with the test data. It takes as input a graph G and the trained model, and it predicts a tour by sequentially generating the next node in the sequence. The function works as follows:
  - The model is set to evaluation mode, and the node coordinates are extracted from the graph.
  - The tour starts at node 0, and the model predicts the next node in the sequence based on the current partial tour.

- The process continues until all nodes are included in the tour, and the final tour is returned.
- The function uses a greedy sampling strategy, where the node with the highest predicted probability is selected at each step. This is similar to the greedy algorithm used in natural language processing tasks, but the main difference is that the model is trained to predict the next node in the tour rather than the next word in a sentence.

- **Plot four boxplots showing the gap distribution**
  The figure below shows the gap distribution for four different methods:
  - **Random Tour**: Generates a random tour by randomly selecting nodes.
  - **Greedy Algorithm**: Constructs a tour by iteratively selecting the nearest unvisited node.
  - **Model**: Predicts a tour using the trained transformer model (without gradient accumulation).
  - **Model (GA)**: Predicts a tour using the trained transformer model with gradient accumulation.

  The figure 3 reveal the following insights:
  - The **Random Tour** method has the highest gap distribution, indicating that it performs the worst among the four methods. This is expected, as random tours do not consider the structure of the graph.
  - The **Greedy Algorithm** performs significantly better than random tours, with a lower gap distribution. However, it is still outperformed by the transformer-based models.
  - Both the **Model** and **Model (GA)** achieve the lowest gap distributions, demonstrating that the transformer-based approach is effective at solving TSP. The gap distributions for these two methods are very similar, suggesting that gradient accumulation does not significantly impact the model's performance in this case.

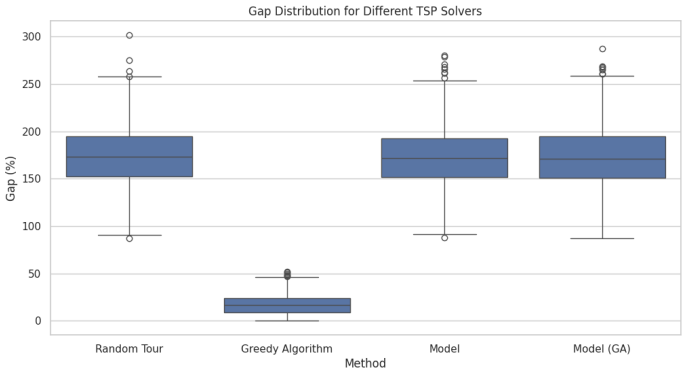

Figure 3: Boxplots showing gap distribution

Critique (5 pts)

- **Model Architecture**:
  The transformer-based architecture, while effective for small graphs (e.g., 20 nodes), may not scale well to larger graphs due to its quadratic complexity in self-attention. For graphs with 30 or 100 nodes, the model's memory and computational requirements could become prohibitive.

- **Dataset Size and Diversity**:
  The dataset is limited to graphs with 20 nodes, which restricts the model's ability to generalize to larger or more complex graphs. Additionally, the dataset is generated using a uniform distribution of nodes, which may not reflect real-world scenarios where nodes are distributed differently (e.g., clustered or along a path).

- **Generalizability**:
  The model is trained on Euclidean TSP instances, where distances are calculated using Euclidean geometry. It may struggle with non-Euclidean distances (e.g., road networks or flight paths) or graphs without coordinates. The model's performance on larger graphs (e.g., 30 or 100 nodes) is also untested.

- **Model Size**:
  The transformer model is relatively large, with multiple encoder and decoder layers. While this allows the model to capture complex patterns, it also increases the risk of overfitting and makes the model less practical for deployment in resource-constrained environments.

- **Hyperparameter Choices**:
  The choice of hyperparameters (e.g., learning rate, number of layers, dropout rate) may not be optimal for all datasets or problem instances. Further tuning could improve the model's performance, especially for larger or more diverse datasets.