Stefano Quaggio

Heuristic for TSP Using Transformers

Dataset (20 pts)

- The dummy dataset consists of a set of graph-tour tuples. In detail we have:
  1. Graph class of the networkx library, within which we find a range of information such as having 20 nodes and their location (x, y).
  2. List with inside the sequence of nodes that refer to the tour of that graph.
- We have other information within the graph, such as having 190 edges, which are weighted, meaning they possess a cost of travel between nodes. Through the function `nx.get_edge_attributes(graph, "attribute name")` we can visualize attributes such as:
  1. **tour**: boolean value indicating whether the edge is part of the tour or not.
  2. **weight**: Euclidean distance assigned to the edge connecting 2 distinct nodes.
  3. **pos**: position (x,y) of a given graph node in space.
- Using the same function seen above, I extract the coordinates of each node and convert them to a tensor. Same with the tour starting and ending at zero (for the latter I am facilitated, since it is the second element of the tuple provided by the dataset).
- Once the datasets are loaded, I initialize the `TSPDataset` class with the path relative to the training, validation, and test datasets. Next I create a `Dataloader` for each dataset using a `batch_size` of 32.

Model (35 pts)

- I implemented the transformer following the structure given within the assignment. Relative to the layers used, I used `Transformer` encoder and decoder layers from the standard `Torch` library. The various hyperparameters were obtained after a variety of tests to try to lower both the training loss and to maximize the generalization capability of the model. Relative to the number of decoders and encoder layers, for example, I did not want to overdo the size because I was getting too high training times and increasing overfitting, as well as the number of head attention.

The increased feedforward size in contrast allowed me to improve accuracy slightly by not raising the time during training by too much.

- The masking I have used occurs on both encoder and decoder. Specifically, relative to the encoder it is used to prevent it from attending to padding tokens, ensuring that only valid tokens contribute to the attention mechanism. Relative to the decoder, masking is applied to prevent the model from attending to future positions in the target sequence. This prevents the decoder from cheating and seeing future solutions, since it already has ground truth. The two functions to create the triangular masks rely on the implementation seen in *Exercise 5* seen during the lectures, with a small adaptation related to the final padding mask for problems caused by the dataset.
- Positional encoding can be omitted for the encoder since we are not interested in their order within the sequence but in the relative positions that the nodes have with each other.
- In the decoder, on the other hand, the situation is a little different: since it generates the tour nodes, they must be placed within the positional encoder to maintain their order, allowing the path to continue based on the choices made previously and avoiding visiting nodes already seen.

Training (25 pts)

For the training phase, I used as a structure that within *Exercise 5* seen in class. An outer loop for epochs within which a training function is called that cycles over the various records in the dataset.

Standard Training (10 pts)

After several tests, I found `Adamw` as the best optimizer for this task, which allowed me to set the weight decay value so as to improve the generalization of the model. Regarding *betas* and *eps* I kept the values from the exercise, as they were also very good for my model. For the loss function I kept the `CrossEntropy`: initially I tried to insert the parameter for *label smoothing*, forcing the model not to focus only on one solution but to explore others as well, but the loss values for validation were getting worse and so I decided not to insert any parameter for this function. The learning rate was set to **0.0002**, as it was a good compromise value between avoiding overfitting and having maximum possible performance. To stay within the 10-minute range, I perform the training with 10 epochs, having about 1 minute per epoch. Concerning the train epoch function, I adapted the code of *Exercise 5* so that it is aligned with the structure for the TSP, that is, so that target input and output are shifted as shown in the model diagram. To avoid the gradient explosion, I also wanted to make sure to add gradient clipping during each weight update. Also in the training phase, I created an evaluate function to get the loss on the validation set and understand that the model is correctly decreasing the error and that we do not have an overfitting situation. The code is very similar to the training loop, the only difference being that it does not have the weight update. As can

be seen from the Figure 1, we have no overfitting within the 10-epoch window, with a final loss value for training of **1.975** and validation of **1.971**.

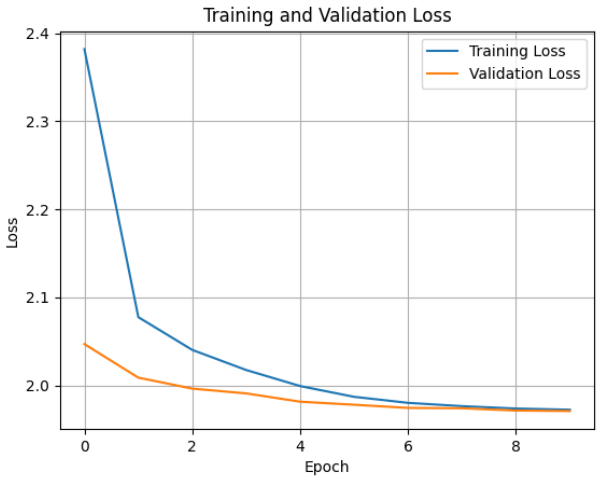

Figure 1: Training and validation loss for standard training

Training with Gradient Accumulation (15 pts)

For gradient accumulation training, I decided to keep the previous training as the basic structure, with the only difference being that the gradient is accumulated every `accumulation_steps`. In my case I found 4 as the ideal parameter, since it is possible to see the effect of accumulation in the short window of the 7 total epochs established, always taking advantage of the stability of the accumulated gradient. In fact, this allowed me to have more stability in the training, reducing the noise on the gradient and allowing me to increase the learning rate to a value of **0.0004**. Thanks mainly to the increase in the learning rate, I was able to reduce the number of epochs while maintaining the same loss value as the standard training for both train and validation. Again I use `AdamW` as an optimizer, which allows me to include parameters such as *weight decay* and *betas* to improve the performance of the transformer during training. As can be seen from Figure 2 also in this case there is no overfitting, as the training loss value remains above the loss value for validation during all 7 epochs. The effect of gradient accumulation can also be seen, with a more angular curve compared to the previous training. The final loss values are **1.974** for training and **1.971** for validation.

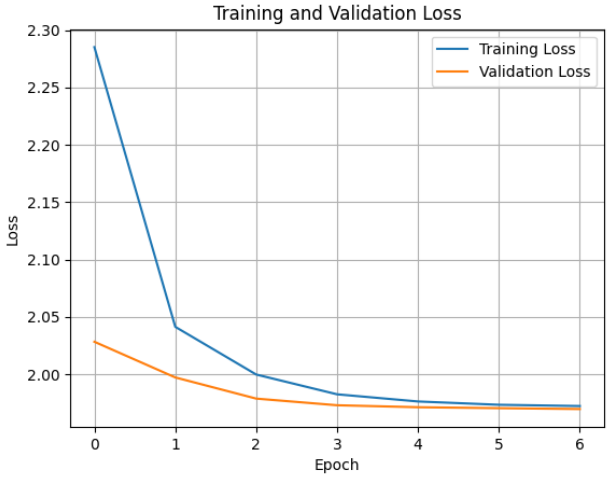

Figure 2: Training and validation loss for gradient accumulation training

Testing (15 pts)

- The already implemented function `transformer_tsp` allows me to go and evaluate the pre-trained model on a graph passed in as input, returning the tour length. Important to remember, the graph passed as input must be of the `graphx` type: therefore, when I pass the test dataset, I do not pass the previously created dataloader, since it is a collection of tensors that are not usable by the function at all. Regarding the function, it allows to predict the next node (starting from 0) based on the temporary tour that was created, mainly to avoid reuse of nodes. The logic is very similar to the greedy strategy used in the NLP task, with the difference being that once the k most likely nodes are taken, it starts from the most likely one and looks for the first node that has not yet been included in the created temporary tour.

- For this question, I created 4 box plots to visualize the performance of the 4 approaches:

  1. **Random**: the next node in the tour is drawn randomly.
  2. **Greedy**: the next node in the tour is drawn so that it is the nearest node not yet visited.
  3. **Model**: the next node in the tour is generated from the model with standard training.
  4. **Model (GA)**: the next node in the tour is generated from the model that had the training with gradient accumulation.

  The code iteratively invokes the pre-implemented function `compute_gap` which calculates the optimality gap for each instance and each approach described above. This gap is basically the difference between the solution generated by the approach and the optimal solution already shown in the graph. As can be seen from Figure 3, both models perform slightly better

than the random approach but are far from what the greedy algorithm achieved.

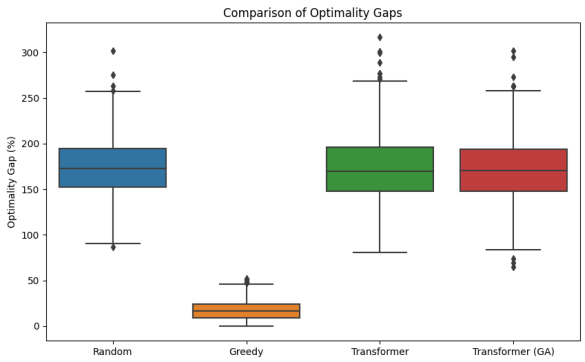

Figure 3: Gap distribution for the 4 methods

Critique (5 pts)

- The model created uses standard attention, but it does not take advantage of TSP-specific attention mechanisms, such as attention on graphs or attention on distances might be. It may in fact have problems in the case of using graphs without coordinates.
- Regarding the architecture of the model, tests with the provided dataset and dummy set could indicate the possibility of the model being more robust when dealing with large graphs, having difficulty generalizing due to the high dimensionality of the `d_e` and `d_d` parameters, causing overfitting.
- The choice the fine tune of parameters was made for the type of graph considered. Some additions may be needed both in the training parameters and in the parameters related to the model structure for any other graphs. Both have very similar performance, indicating that with the current batch size and graph we do not get much benefit from gradient accumulation.