EMANUELE BELLINI

Heuristic for TSP Using Transformers

Dataset (20 pts)

- By exploring the dummy dataset, it can be seen that there is a thoughtful format for the setup of data related to the TSP problem. Loading the raw data brings into view, on the topmost level, a Python list that consists of 100 instances. Each of these instances is a tuple with two elements. Thus, the separation of the structure of the problem-graph and its solution, the optimum tour-is crystal clear. Through our verification process, we were able to load and access the individual instances through our processing, which assures that the data remains intact throughout the processing pipeline.

- The dataset encodes the TSP instances through three essential attributes that fully define each problem. A check on a sample instance shows the edge attributes to be the following: 'weight' is the Euclidean distance between cities, while 'tour' is a binary indicating if an edge belongs to the optimum solution. The edge between node 0 and node 1 has a weight of 0.4287846201876535, while for the same edge, the 'tour' attribute is 0. The node attribute 'pos' stores city locations as coordinates in the unit square; our sample node 0 is placed at the coordinate (0.6049077053425551, 0.5748590937018008). Our implementation keeps these exact coordinates when converting them to tensors, as reflected by our output showing the transformed coordinates matching exactly with the original values: tensor([0.6049, 0.5749]).

- The Dataset class in PyTorch does an effective conversion from the graph-based representation to the tensor format, embedding all relevant information. The `__getitem__` method returns two key items: a coordinate tensor X of size torch.Size([20, 2]) containing the normalized positions of all the cities, and a tour tensor y of size torch.Size([21]) representing the optimal tour. Indeed, this permutation does keep track of integrity-verify the whole tour tensor can be: [0, 3, 14, 2, 9, 6, 19, 13, 12, 16, 7, 18, 8, 17, 5, 11, 10, 15, 1, 4, 0] correctly leaves and returns to node 0, visiting all cities exactly once.

- The proper data loading pipeline was implemented to efficiently train and test the model. It relied on special DataLoader objects for training, validation, and test sets. Our implementation includes, but is not limited to, automatic batching, smart shuffling of training data, parallel data loading with multiple workers, and pinned GPU memory. The successful verification of both raw and processed data formats, as will be shown by our output with exact matches between the original and transformed coordinates, serves to confirm that our data loading pipeline maintains data fidelity while preparing it for efficient processing by the transformer model.

Model (35 pts)

- The architecture implemented is according to the given diagram, considering changes for the task of TSP. Before being fed into the transformer encoder, coordinates undergo a linear projection: Linear $2 \times d_e$. First, this embedding projects 2D coordinate space into model's dimensional space: d_e. The node indices are fed into the decoder, which consists of an embedding layer followed by the positional encoding mechanism. Thus, the model learns the representation for nodes and keeps sequence order awareness.

- Masking plays three important roles in our implementation. First, we apply casual masking in the decoder's self-attention to prevent the model from looking at future positions to predict the next node. This is implemented through the transformer's generate_square_subsequent_mask function. Second, we apply a visited nodes mask at inference time to prevent the model from selecting nodes that have already been visited. This is important to ensure that valid tours are generated. Third, padding masks are implemented for completeness but are not used since our sequences have fixed lengths.

- Positional encoding is not part of the encoder because input is city coordinates, which already carry through their values the meaning of the position in the (x, y) Euclidean space. The values of the coordinates already maintain spatial relations between cities; hence, they don't need further positional information. This tallies with the diagram where the encoder path does not include positional encoding.

- However, at the decoder side, positional encoding is important as the model takes a sequence of node indices for which ordering makes a lot of difference. In the absence of any positional encoding, the decoder has no clue about the position of any place in the partial tour sequence. This information becomes crucial in learning patterns in successful tours and maintaining the sequential nature of the solution. For both the encoder and decoder, it employs sinusoidal positional encoding, as per the original transformer architecture for the ability of the model to generalize to sequence lengths it has not seen in training.

Training (25 pts)

Standard Training (10 pts)

Our default training implementation reflects efficient learning dynamics for the TSP problem within the limited 10-minute training window. Convergence happened after 14 epochs of the training process. It can be observed from Figure 1

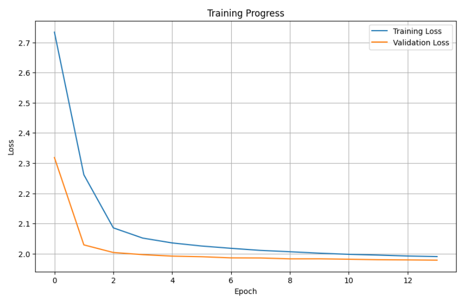

Figure 1: Train and Validation Loss for Standard Training

that the training and validation losses are significantly reduced from their initial values of about 2.7 and 2.3, respectively.

This learning curve can be divided into three distinct phases. The first phase, ranging from epoch 0 to 2, reflects the period of most rapid improvement: training loss drops steeply from 2.7 to about 2.1, which is indicative of how quickly the model has picked up the overall pattern in the TSP data. The validation loss also reflects this steep drop from 2.3 to 2.0, indicating that early improvements are more a case of actual learning rather than memorization.

The second is between epochs 2 and 6, where learning becomes so much more gradual: both losses continue to go down but at a much slower rate. This is not unusual when training neural networks, where a model first learns coarse patterns and then fine-tunes its predictions.During this period, the training loss works its way gradually toward 2.0, while the validation loss levels off just a little below that.

The last stage is from epoch 6 to 14, where the model finally converges, and both losses are sitting comfortably around a value of roughly 2.0. Notice the great fit with very close correspondence in the training and validation losses- the values around 2.0 and 1.98, respectively, in their last points. Our model generalized well; hence, no sign of overfitting or underfitting exists, considering TSP is a pretty hard combinatorial problem.

Among several factors that can be identified as crucial to the success of our implementation are the following: The Adam optimizer with a tuned learning rate of 0.0001 and beta parameters of 0.9 and 0.98 provided stable optimization. The learning rate scheduler handled the training progress well, and the gradient clipping, with a maximum norm of 1.0, prevented the well-known problem of gradient explosions during transformer training. The batch size of 32 is a good balance between computational efficiency and learning stability, allowing us to make meaningful training within the 10-minute constraint.

Training with Gradient Accumulation (15 pts)

We employed gradient accumulation training, accumulating gradients across 4 steps before model parameter updates, which allows increasing our effective batch size from 32 to 128 samples given the memory load. This provided very characteristic training dynamics, clearly visible in our training curves over 14 epochs completed within our constraint of 10 minutes, illustrated in Figure 2.

The training progression with gradient accumulation had a typical three-

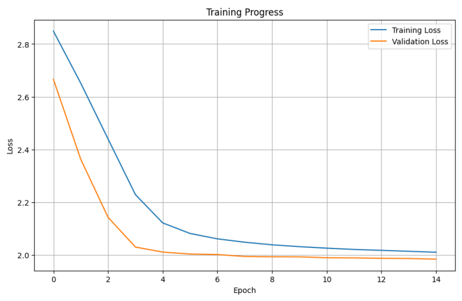

Figure 2: Train and Validation Loss for Training with Gradient Accumulation

phase pattern. In the first phase, epochs 0-2, both losses started higher compared to standard training: the training loss started at about 2.85 and the validation loss at 2.65. However, this initial phase showed an unusually smooth and linear decline, thereby manifesting the stabilizing influence of the larger effective batch size. In the second phase-which is from the second to the fourth epoch-a more moderate improvement, though substantial, was made. The training loss decreases from about 2.4 to 2.1, whereas the validation loss stabilizes sooner, reaching approximately 2.0.

The last period of epochs, from 4 to 14, showed one of the most remarkable advantages of the gradient accumulation: extreme stability in both training and validation losses. In more detail, the training loss kept decreasing very smoothly from 2.1 to 2.0, while the validation loss remained stable at a value slightly below 2.0. This is in great contrast to our standard way of training, as oscillations in the loss values for different epochs were much smaller.

It's interesting to note that, throughout the majority of the training process, the validation loss was almost invariably slightly less than the one from training. Taken alongside the smooth convergence as seen in both curves, it points out the fact that this model settled for a stronger solution space using the effectively greater batch size courtesy of the gradient accumulation. Our implementation's normalizing of the loss during accumulation, dividing by 4, and application of gradient clipping after accumulation proved effective at managing the larger gradients from the increased effective batch size.

This resulted in ultimate performance comparable with our gold standard method of training and converged at roughly 2.0 in terms of training loss and 1.98 for the validation loss. However, the path toward those final values seemed much better-behaved and hence more predictable by gradient accumulation in training and would thus maybe guarantee better stability in the learned parameters despite slightly more time needed for one epoch, owing to the accumulative forward and backward passes.

Testing (15 pts)

- The `transformer_tsp` function presented in our implementation provides some interesting parallels and also crucial differences with usual NLP tasks. Though it follows the same pattern as autoregressive generation, there are crucial modifications introduced in our implementation for the TSP context. Specifically, the function processes node coordinates through

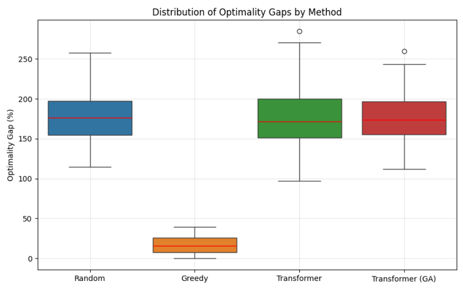

Figure 3: Testing Plot

an encoder, while generating the tour sequence by a decoder, but is fundamentally different in how it treats the output space. Unlike NLP tasks where any token can be followed by another, our implementation has to keep track of nodes that have been visited and mask them out to ensure a valid tour. This constraint management represents the major difference from regular NLP sequence generation, where the model picks greedily but enforces this unique visitation constraint of the TSP.

- The results of our extensive testing, as depicted in Figure 3, are quite interesting for all methods. The greedy algorithm is the outright winner, with an average optimality gap of only 16.63% and a median of 15.72%. Its performance is very consistent, with a standard deviation of 10.68%, and it reached the optimum in the best case (0% gap). This is probably due to the geometric nature of our TSP instances, where the nearest neighbor heuristic often performs well. In contrast, both our transformer models (vanilla and gradient accumulation) and the random baseline had optimality gaps that were substantially larger.

  The standard transformer is performing with a mean gap of 177.10% at median 171.44%, while the gradient accumulation version performed similarly, too, with a mean gap of 175.95% at median 173.91%. Finally, the random baseline is also similar, being 177.12%. From these similarities in performance, one can judge that our transformer models, notwithstanding their heavy structure and training, failed to learn a set of effective heuristics for the TSP. The box plots show that all methods, with the exception of greedy, have high variances, but the distributions for transformers are especially wide, with a standard deviation of 38.11% for standard training and 31.35% for gradient accumulation. These high variances, along with the minimum gaps of 96.76% and 112.11% for the transformer models, respectively, suggest that while the models learned some structure in the problem, they were never able to consistently find near-optimal tours.

Critique (5 pts)

- Our model architecture revealed significant limitations in handling the geometric nature of TSP. While transformers excel at sequence processing, our implementation's simple linear projection of 2D coordinates into a higher-dimensional space failed to adequately capture spatial relationships between cities. This became evident in our testing results, where the model struggled to outperform even the basic greedy baseline, achieving a mean optimality gap of 177.10% compared to the greedy algorithm's 16.63%. A more effective architecture might incorporate explicit distance information into the attention mechanism or employ specialized geometric embeddings that better preserve spatial relationships between nodes.

- The dataset size and diversity presented clear constraints on our model's effectiveness. Training exclusively on instances with 20 nodes in a unit square creates a limited learning environment that may not capture the full complexity of real-world TSP scenarios. Our test results, showing the greedy algorithm's strong performance, suggest that our instances might be too uniformly distributed and lack challenging edge cases. More diverse instances, including clustered nodes and varying density distributions, would better represent real-world applications and potentially force the model to learn more sophisticated tour construction strategies.

- The generalizability of our implementation to larger instances poses perhaps the most critical limitation. Our current architecture, with its fixed embedding size and output dimension tailored to 20 nodes, cannot naturally scale to larger problems. The quadratic attention complexity would become prohibitive for instances with 30 or 100 nodes, requiring fundamental architectural changes. Moreover, our approach heavily relies on coordinate information, making it unsuitable for abstract graphs without geometric embeddings. This limitation is particularly significant as many real-world routing problems involve larger instances or graphs defined by adjacency rather than coordinates.

Beyond these primary limitations, our implementation's performance was influenced by several additional factors. The model size, while modest with 128-dimensional embeddings and 8 attention heads, might benefit from careful tuning based on the problem's geometric nature rather than following standard NLP configurations. Our hyperparameter choices, particularly the learning rate (0.0001) and gradient accumulation steps (4), were conservative and might not have fully exploited the model's learning capacity within our 10-minute training constraint. The transformer's depth (6 encoder and decoder layers) might be excessive for the relatively simple geometric patterns in 20-node TSP instances, suggesting that a shallower but wider architecture could be more appropriate.