Stanford MS&E 236 / CS 225: Lecture 2
The traveling salesman problem and recurrent neural networks

Ellen Vitercik[*]

April 17, 2024

In these notes, we will begin with an overview of the canonical traveling salesman problem (TSP) and then discuss tools that we will use to integrate machine learning into solving TSP.

1 Traveling salesman problem

In TSP, the input is a network with n nodes representing a map with n cities. We denote the distance from node i to node j as $c_{i,j} \geq 0$. The goal in TSP is to find the shortest distance tour passing through each node exactly once. We can represent a tour as a permutation $\pi : [n] \to [n]$, where $\pi(1)$ is the first node we visit, $\pi(2)$ is the second node we visit, and so on. Thus, our goal is to find the permutation that minimizes

$$c_{\pi(n),\pi(1)} + \sum_{i=1}^{n-1} c_{\pi(i),\pi(i+1)}.$$

TSP is one of the most famous NP-hard problems[1] and has served as a "challenge" problem in both theory and practice for over 70 years. Due to this problem's difficulty, many polynomial-time, efficient heuristics have been developed for this problem. For example, Algorithm 1 is pseudo-code for the *nearest insertion* heuristic. If, in Step 3, we chose the

Algorithm 1 Nearest insertion TSP heuristic.

```
Input: $n$ cities with distances $c_{i,j}$ between each pair $(i, j)$
1: Start with subtour that only consists of the first city
2: while the subtour is not a complete tour do
3:     Among all cities not in the subtour, choose the one that is closest to any city in the subtour
4:     Insert it into the subtour at whatever position causes the smallest increase in tour length
Output: Return the complete tour of all $n$ cities
```

city that is *farthest* from any city in the subtour, we would have the *farthest insertion* heuristic. See this blogpost for animations of these heuristics and others.

[*]These notes are course material and have not undergone formal peer review. Please feel free to send me any typos or comments.

[1]In theory, this means that we cannot find an optimal tour much faster than trying out all $(n-1)!$ tours. To see why there are $(n-1)!$ tours, suppose, without loss of generality, that we start at node 1. There are $n-1$ choices for the second node we visit, $n-2$ choices for the node we visit after that, and so on. Thus, in total, there are $(n-1)!$ total tours.

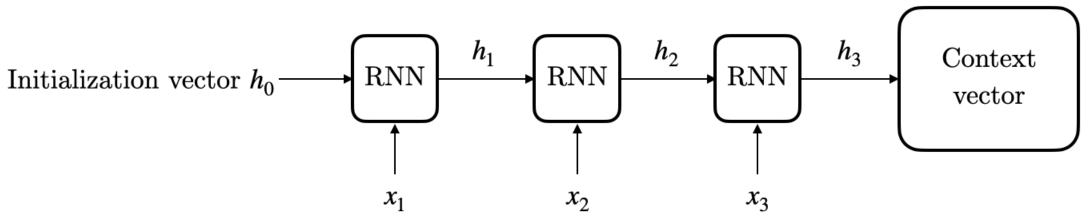

Figure 1: Encoder RNN

As we will see in later classes, these hand-designed heuristics can work well on certain TSP instances. However, they are based on simple intuition, and it is reasonable to hope that with an extensive search, we could find stronger, efficient heuristics that are based on more complex calculations of the input network. Our goal in the next few classes will be to use machine learning to discover heuristics that are optimized for the particular type of TSP instances they are trained on (corresponding to, for example, particular types of robotics tasks, or the specific road networks that a shipping company navigates day after day).

Our initial investigation into using ML for TSP will be based on papers by Vinyals et al. [2] and Bello et al. [1], who used sequence-to-sequence models for this task. These early papers illustrated the potential benefits of using deep learning for combinatorial optimization. In future classes, we will expand on these building blocks as we discuss more modern architectures, such as transformers. Moreover, we will compare and contrast sequence-to-sequence models with other architectures, such as graph neural networks.

2 Sequence-to-sequence recurrent neural networks (RNNs)

Sequence-to-sequence models[2] have primarily been developed for natural language processing, but we will see their utility for other tasks in the next few classes. The input is a sequence of vectors $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n \in \mathbb{R}^{d_0}$ which could be, for example, word embeddings or cities on a map with $d_0 = 2$. The output is a sequence of vectors $\boldsymbol{y}_1, \dots, \boldsymbol{y}_m \in \mathbb{R}^{d_1}$. This output could correspond to a translation of the original sequence into a foreign language or, as we will see, a tour of the n input cities.

The first step of a sequence-to-sequence model is to feed the input into an *encoder recurrent neural network (RNN)*, which is illustrated in Figure 1. The encoder RNN is based on an initialization vector $\boldsymbol{h}_0 \in \mathbb{R}^{d_h}$, which is a trainable parameter. Using $\boldsymbol{h}_0$ and $\boldsymbol{x}_1$, we compute the *hidden state* $\boldsymbol{h}_1 \in \mathbb{R}^{d_h}$, which is the output of a simple, single-layer neural network

$$\boldsymbol{h}_1 = \phi\left(W^{(hh)}\boldsymbol{h}_0 + W^{(hx)}\boldsymbol{x}_1\right),$$

where ϕ is a non-linearity, and $W^{(hh)} \in \mathbb{R}^{d_h \times d_h}$, $W^{(hx)} \in \mathbb{R}^{d_h \times d_0}$ are trainable weight matrices. In a similar fashion, we compute the next hidden state

$$\boldsymbol{h}_2 = \phi\left(W^{(hh)}\boldsymbol{h}_1 + W^{(hx)}\boldsymbol{x}_2\right),$$

and so on. In general, the t^{th} hidden state is computed as

$$\boldsymbol{h}_t = \phi(W^{(hh)}\boldsymbol{h}_{t-1} + W^{(hx)}\boldsymbol{x}_t).$$

[2]This section closely follows Stanford's CS244N notes on (1) RNNs and (2) sequence-to-sequence models and attention mechanisms.

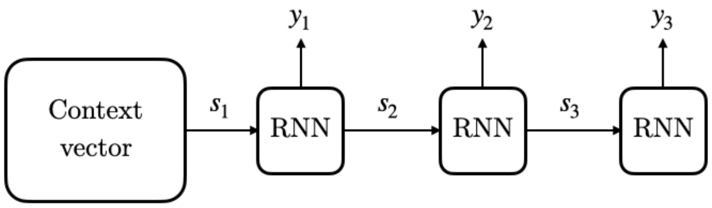

Figure 2: Decoder RNN

After all inputs have been processed, we have a *context vector*.

Given the context vector, we compute the outputs $\boldsymbol{y}_1, \dots, \boldsymbol{y}_m$ in a similar fashion, as illustrated in Figure 2. We will denote the initial context vector as $\boldsymbol{s}_1$, and the output $\boldsymbol{y}_1$ is computed as

$$\boldsymbol{y}_1 = \text{softmax}(W^{(sy)}\boldsymbol{s}_1),$$

where $W^{(sy)}$ is a trainable weight matrix. We then compute the next hidden state

$$\boldsymbol{s}_2 = \phi(W^{(ss)}\boldsymbol{s}_1),$$

where $W^{(ss)}$ is a trainable weight matrix and ϕ is a non-linearity. Similarly, the t^{th} output is defined as

$$\boldsymbol{y}_t = \text{softmax}(W^{(sy)}\boldsymbol{s}_t)$$

and the t^{th} hidden state is defined as

$$\boldsymbol{s}_t = \phi(W^{(ss)}\boldsymbol{s}_{t-1}).$$

With this, we have the basic encoder-decoder RNN.

Crucially, this architecture can transform sequences of arbitrary length to sequences of arbitrary length since the weight matrices are shared across all indices. However, information from early in the sequence may be lost later in the sequence and RNNs suffer from exploding and vanishing gradients (see, for example, these lecture notes from Stanford's CS244N course). These issues, among others, motivate *long-short-term-memories (LSTMs)*, which we will cover next. Moreover, in the context of TSP, the output will almost certainly not form a tour of the n cities, which motivates *pointer networks*. We will discuss pointer networks next class.

2.1 Long-short-term-memories (LSTMs)

The key idea motivating LSTMs is to introduce a *cell state* (in addition to the hidden states $\boldsymbol{h}_t$ and $\boldsymbol{s}_t$), which can be thought of as a conveyor belt that runs down the entire chain of computations. It is easy for information to flow along this conveyor belt with minimal change. A handful of gates will regulate how much information is added to or removed from the cell state. For example, if the cell state includes the gender of a previous subject, we should, intuitively, forget that gender if we encounter a new subject. These gates are illustrated in Figure 3 and can be interpreted as follows:

1. Generate **new memory:** This step works much like the vanilla RNNs we saw above. We combine the prior hidden state $\boldsymbol{h}_{t-1}$ and the input $\boldsymbol{x}_t$ to generate a new memory:

$$\tilde{\boldsymbol{c}}_t = \tanh\left(U^{(c)}\boldsymbol{h}_{t-1} + W^{(c)}\boldsymbol{x}_t\right).$$

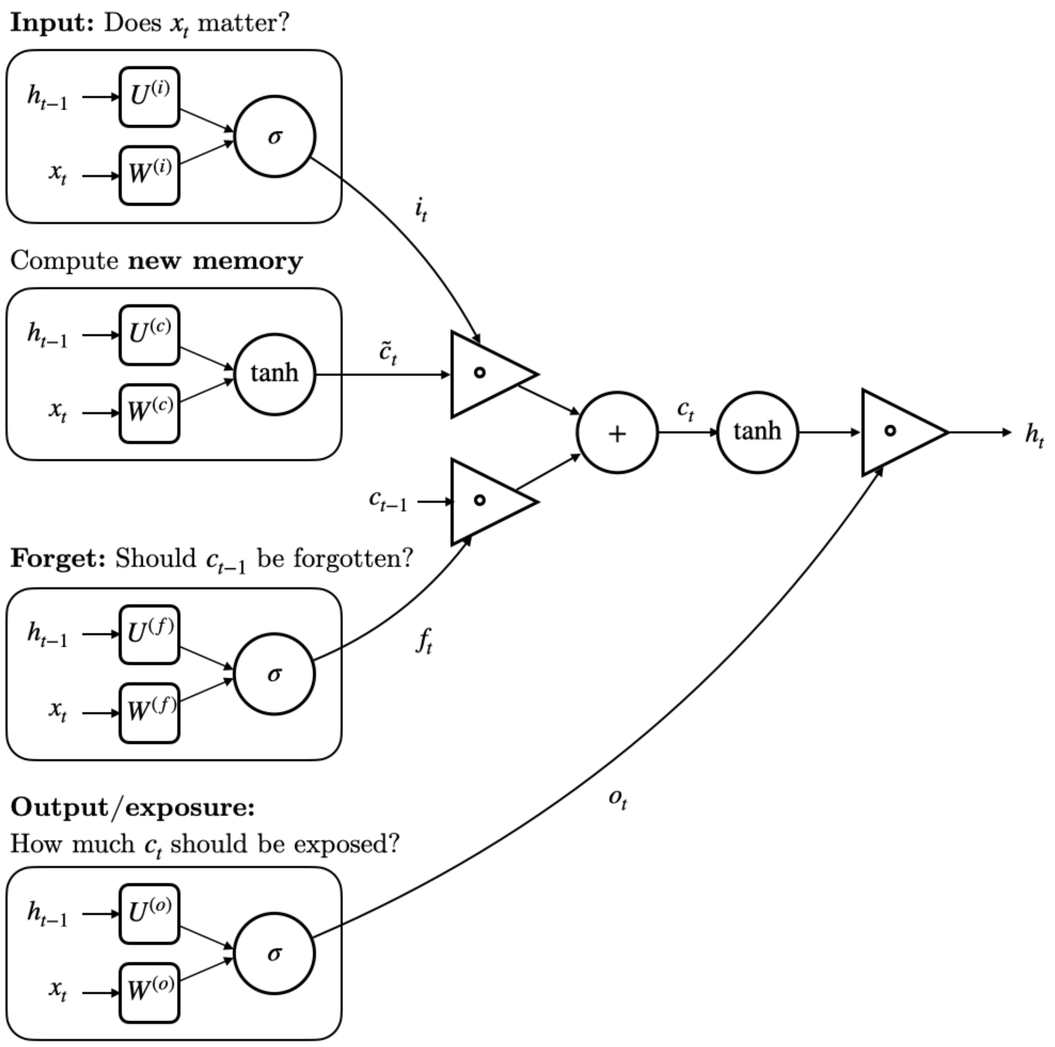

Figure 3: LSTM

For all gates, the matrices U and W (with some superscript) are trainable parameters.

2. **Input gate:** In the step above, we do not check if the new memory is "important" before we generate it. The input gate determines whether the input $\boldsymbol{x}_t$ is important and, thus, whether information about it is worth preserving, as encoded by the vector

$$\boldsymbol{i}_t = \sigma\left(U^{(i)}\boldsymbol{h}_{t-1} + W^{(i)}\boldsymbol{x}_t\right),$$

where σ is a sigmoid function. Intuitively, if i_t is close to zero, the memory is not worth preserving.

3. **Forget gate:** This gate is similar to the input gate, except that it is meant to determine whether the previous cell state $\boldsymbol{c}_{t-1}$ is worth preserving. Its computation is

$$\boldsymbol{f}_t = \sigma\left(U^{(f)}\boldsymbol{h}_{t-1} + W^{(f)}\boldsymbol{x}_t\right).$$

4. Generate **new cell state:** The new cell state is generated using the advice of the input and forget gates. In particular, the new cell state is

$$\boldsymbol{c}_t = f_t \circ \boldsymbol{c}_{t-1} + i_t \circ \tilde{\boldsymbol{c}}_t.$$

5. **Output/exposure gate:** This gate is meant to delineate the hidden state $\boldsymbol{h}_t$ from the new cell state $\boldsymbol{c}_t$. The hidden state $\boldsymbol{h}_t$ is used as input to every LSTM gate, and the output/exposure gate determines which parts of the new cell state $\boldsymbol{c}_t$ should be saved in the hidden state as encoded by the vector

$$\boldsymbol{o}_t = \sigma\left(U^{(o)}\boldsymbol{h}_{t-1} + W^{(o)}\boldsymbol{x}_t\right).$$

   Finally, the hidden state is defined as

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh(\boldsymbol{c}_t).$$

2.2 Attention

The last concept we will briefly discuss today is *attention*, which will come up again in later classes on. Here, we note that the decoder RNN only uses the single context vector to produce outputs. However, two outputs $\boldsymbol{y}_t$ and $\boldsymbol{y}_{t'}$ will likely depend on the input in varying ways. Under an attention mechanism, the output $\boldsymbol{y}_t$ will depend on the decoder hidden state $\boldsymbol{s}_t$ as well as the entire sequence of encoder hidden states $\boldsymbol{h}_1, \dots, \boldsymbol{h}_n$. We will see an example of this in the next class.

3 Next time

Next time, we will discuss pointer networks, which use LSTMs to predict *permutations*, such as the order of nodes in a TSP tour.

References

[1] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *Workshop track of the International Conference on Learning Representations (ICLR)*, 2017.

[2] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2015.