Stanford MS&E 236 / CS 225: Lecture 3
Pointer networks for the traveling salesman problem

Ellen Vitercik*

April 17, 2024

This class introduces *pointer networks* [2], which use LSTMs to predict *permutations*, such as the order of a set of cities in a tour. We will also see how to use policy gradients, i.e, the classic REINFORCE algorithm [3] to train pointer networks to (approximately) solve the traveling salesman problem (TSP). The focus of this lecture is TSP, but the ideas apply to other combinatorial problems as well. For example, Bello et al. [1] use this framework to compute high-value solutions to the knapsack problem.

1 Pointer networks

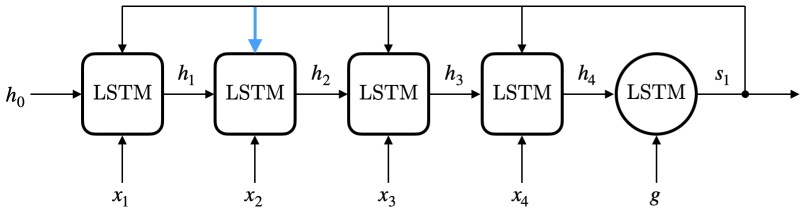
The input to a pointer network is a set $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subset \mathbb{R}^{d_0}$. In the context of the traveling salesman problem, this input could encode n cities on a map with $d_0 = 2$ (or $d_0 > 2$ if embedded in a higher-dimensional space). The output is a distribution over permutations $\pi : [n] \to [n]$ (a pointer network's output could be more general than this, but we will stick to permutations for this class). A permutation π corresponds to a tour of these n cities that visits city $\pi(1)$ first, city $\pi(2)$ second, and so on. We use the notation $\mathbb{P}[\pi \mid X]$ to denote the probability the pointer network places on permutation π given the input X.

In more detail, a pointer network uses long-short-term-memories (LSTMs) to compute individual conditional probabilities $\mathbb{P}[\pi(i) \mid \pi(< i), X]$, which is the probability distribution over the i^{th} city in the tour, conditioned on the first $i-1$ cities in the tour, which we denote as $\pi(< i)$. To obtain the joint distribution $\mathbb{P}[\pi \mid X]$, we use the chain rule:

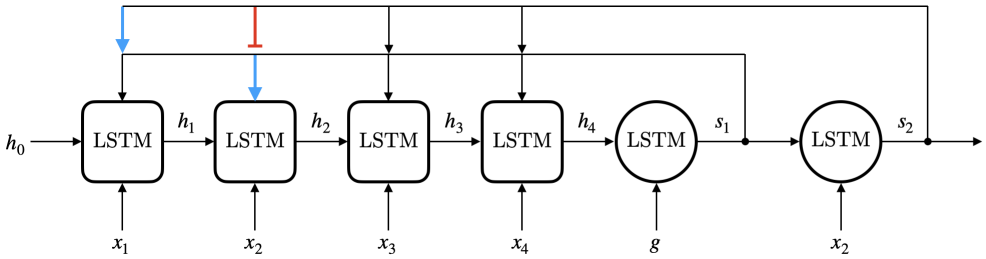
$$\mathbb{P}[\pi \mid X] = \prod_{i=1}^{n} \mathbb{P}[\pi(i) \mid \pi(< i), X].$$

We will begin by carefully working through how we compute $\mathbb{P}[\pi(1) \mid X]$. (Although choosing the first city in the tour is somewhat irrelevant, it will help us build intuition for the full pointer network.) First, the input is encoded using an encoder LSTM, which computes hidden states $\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_n$ (where the initial hidden state $\boldsymbol{h}_0$ is a trainable parameter). The input to the first decoder LSTM is the hidden state $\boldsymbol{h}_n$ together with a trainable parameter $\boldsymbol{g}$. This LSTM returns the first decoder hidden state $\boldsymbol{s}_1$. To compute $\mathbb{P}[\pi(1) \mid X]$, the pointer network defines a vector $\boldsymbol{u} \in \mathbb{R}^n$ where $u_t = a(\boldsymbol{h}_t, \boldsymbol{s}_1)$ is the output of an *attention function* a. The value $a(\boldsymbol{h}_t, \boldsymbol{s}_1) \in \mathbb{R}$ is meant to capture how relevant the t^{th} input is

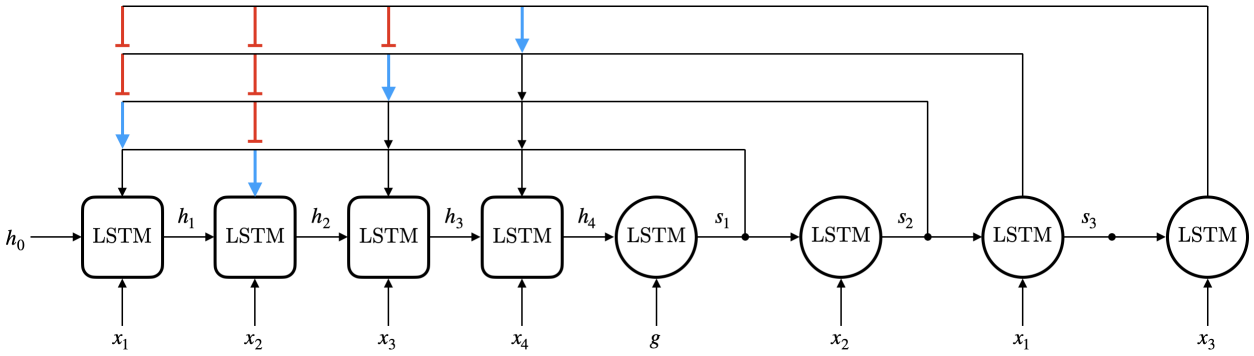
*These notes are course material and have not undergone formal peer review. Please feel free to send me any typos or comments.



(a) The blue arrow indicates that city 2 is chosen as the first city of the tour.



(b) City 1 is chosen as the second city of the tour. The red line indicates that it is not possible to choose city 2 again.



(c) The full pointer network.

Figure 1: Illustration of a pointer network that selects the tour $2 \to 1 \to 3 \to 4 \to 2$.

to the first output. For example, in a seminal paper, Bello et al. [1] define $a(\boldsymbol{h}, \boldsymbol{s}) = \boldsymbol{v}^\top \cdot \tanh\left(W_1 \boldsymbol{h} + W_2 \boldsymbol{s}\right)$, where $\boldsymbol{v}, W_1$, and W_2 are trainable parameters. Finally, we define $\mathbb{P}[\pi(1) \mid X] = \text{softmax}(\boldsymbol{u}) \in [0,1]^n$, and we sample a city from this distribution, as illustrated in Figure 1a. If city j was sampled as the first city in the tour, the pointer network uses $\boldsymbol{x}_j$ as the input to the next decoder LSTM, which computes the next hidden state $\boldsymbol{s}_2$.

To compute $\mathbb{P}[\pi(2) \mid \pi(1), X]$, the pointer network next defines the vector $\boldsymbol{u} \in [0,1]^n$ as follows: for all $t \in [n]$,

$$u_t = \begin{cases} a(\boldsymbol{h}_t, \boldsymbol{s}_2) & \text{if } t \neq \pi(1) \\ -\infty & \text{else.} \end{cases}$$

We again define $\mathbb{P}[\pi(2) \mid \pi(1), X] = \text{softmax}(\boldsymbol{u}) \in [0,1]^n$, and we sample a city from this distribution, as illustrated in Figure 1b. By setting $u_t = -\infty$ if $t = \pi(1)$, we ensure that we do not sample the same city twice. Once again, if city j was sampled as the second city, the pointer network uses $\boldsymbol{x}_j$ as the input to the next decoder LSTM, which computes the next hidden state $\boldsymbol{s}_3$.

More generally, to compute $\mathbb{P}[\pi(i) \mid \pi(< i), X]$, the pointer network defines the vector $\boldsymbol{u}$ as follows: for all $t \in [n]$,

$$u_t = \begin{cases} a(\boldsymbol{h}_t, \boldsymbol{s}_i) & \text{if } t \text{ hasn't been visited, i.e., } t \neq \pi(j) \text{ for } j < i \\ -\infty & \text{else.} \end{cases}$$

As before, $\mathbb{P}[\pi(i) \mid \pi(< i), X] = \text{softmax}(\boldsymbol{u})$. The full pointer network is illustrated in Figure 1c. Check out the paper by Bello et al. [1] for additional bells and whistles that can further improve the performance of pointer networks.

There are several key insights:

1. By sampling first from $\mathbb{P}[\pi(1) \mid X]$, then $\mathbb{P}[\pi(2) \mid \pi(< 1), X]$, and so on, we are guaranteed to obtain a valid tour.
2. The pointer network can be tested on instances of a different size than those it was trained on. Unlike vanilla LSTMs, we are not restricted to outputs from vocabularies of a fixed size.

2 Training pointer networks for TSP

Next, we will discuss training pointer networks for TSP. Given an input set of cities $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subset \mathbb{R}^2$, permutation $\pi : [n] \to [n]$, the length of the tour defined by π is

$$L(\pi \mid X) = \sum_{i=1}^{n-1} \left\|\boldsymbol{x}_{\pi(i)} - \boldsymbol{x}_{\pi(i+1)}\right\|_2 + \left\|\boldsymbol{x}_{\pi(n)} - \boldsymbol{x}_{\pi(1)}\right\|_2.$$

We will abstract away the specifics of the pointer network and simply denote its parameters by $\boldsymbol{\theta}$. We use $p_{\boldsymbol{\theta}}(\pi \mid X)$ to denote the probability the pointer network places on the permutation π. The pointer network's *loss* given X is the expected length of the tour it outputs, which we denote as

$$J(\boldsymbol{\theta} \mid X) = \mathop{\mathbb{E}}_{\pi \sim p_{\boldsymbol{\theta}}(\cdot \mid X)} [L(\pi \mid X)].$$

We will train and test on TSP instances sampled from a distribution $\mathcal{D}$, where $p_{\mathcal{D}}(X)$ is the probability assigned to instance X. For simplicity in these notes, we assume that $\mathcal{D}$ has finite support, but this is not a necessary assumption.

With this notation in place, we can state our overall goal, which is to find a parameter vector $\boldsymbol{\theta}$ that minimizes the overall loss

$$J(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{X \sim \mathcal{D}} [J(\boldsymbol{\theta} \mid X)].$$

Our method for achieving this goal is gradient descent: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. We will use policy gradient (i.e., the REINFORCE algorithm [3]) to estimate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. To explain the logic behind this algorithm, we first write $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ in terms of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} \mid X)$:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \left(\sum_X J(\boldsymbol{\theta} \mid s) p_{\mathcal{D}}(X) \right) = \sum_X \nabla_{\boldsymbol{\theta}} \left(J(\boldsymbol{\theta} \mid X) \right) p_{\mathcal{D}}(X) = \mathop{\mathbb{E}}_{X \sim \mathcal{D}} [\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} \mid X)].$$

Algorithm 1 Vanilla REINFORCE algorithm for TSP

Input: Number of training steps T, batch size B, learning rate α

1: **for** $t = 1, \ldots, T$ **do**
2: Draw B samples: $X_i \sim \mathcal{D}$ for $i \in \{1, \ldots, B\}$
3: Sample a tour for each instance: $\pi_i \sim p_{\boldsymbol{\theta}}(\cdot \mid X_i)$ for $i \in \{1, \ldots, B\}$
4: Estimate the gradient of $J(\boldsymbol{\theta})$ as $g_{\boldsymbol{\theta}} \leftarrow \frac{1}{B} \sum_{i=1}^{B} L(\pi_i \mid X_i) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\pi_i \mid X_i)$
5: Gradient step $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha g_{\boldsymbol{\theta}}$

Output: Parameters $\boldsymbol{\theta}$

Next, given an instance X,

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} \mid X) &= \nabla_{\boldsymbol{\theta}} \mathop{\mathbb{E}}_{\pi \sim p_{\boldsymbol{\theta}}(\cdot \mid X)} [L(\pi \mid X)] && \text{(by definition of } J(\boldsymbol{\theta} \mid X)\text{)} \\
&= \nabla_{\boldsymbol{\theta}} \left(\sum_{\pi} L(\pi \mid X) p_{\boldsymbol{\theta}}(\pi \mid X) \right) && \text{(expanding the expectation)} \\
&= \sum_{\pi} \nabla_{\boldsymbol{\theta}} \left(L(\pi \mid X) p_{\boldsymbol{\theta}}(\pi \mid X) \right) && \text{(moving the gradient inside the sum)} \\
&= \sum_{\pi} L(\pi \mid X) \nabla_{\boldsymbol{\theta}} \left(p_{\boldsymbol{\theta}}(\pi \mid X) \right). && (L(\pi \mid X) \text{ doesn't depend on } \boldsymbol{\theta})
\end{aligned}$$

We will next employ a useful fact from calculus:

$$\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\pi \mid X) = \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\pi \mid X)}{p_{\boldsymbol{\theta}}(\pi \mid X)}.$$

This implies that

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{\pi} L(\pi \mid X) \nabla_{\boldsymbol{\theta}} \left(\log p_{\boldsymbol{\theta}}(\pi \mid X) \right) p_{\boldsymbol{\theta}}(\pi \mid X) = \mathop{\mathbb{E}}_{\pi \sim p_{\boldsymbol{\theta}}(\cdot \mid X)} \left[L(\pi \mid X) \nabla_{\boldsymbol{\theta}} \left(\log p_{\boldsymbol{\theta}}(\pi \mid X) \right) \right].$$

The takeaway from this derivation is that to compute $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} \mid X)$, only need to compute gradients of the (logarithm of the) policy, $\log p_{\boldsymbol{\theta}}(\pi \mid X)$. This leads naturally to the REINFORCE algorithm, Algorithm 1, which runs gradient descent using Monte Carlo estimates of $\nabla_{\boldsymbol{\theta}} \left(\log p_{\boldsymbol{\theta}}(\pi \mid X) \right)$.

REINFORCE with a baseline

A typical issue with the vanilla REINFORCE algorithm is that if $L(\pi \mid X)$ is often large, the gradient estimates $g_{\boldsymbol{\theta}}$ can have high variance. A common variance-reduction strategy is to train a baseline $b(X)$ to estimate the expected loss $\mathbb{E}_{\pi \sim p_{\boldsymbol{\theta}}(\cdot \mid X)} [L(\pi \mid X)]$. This baseline is included when estimating the gradient in Step 4 of Algorithm 1 by instead computing the estimate

$$g_{\boldsymbol{\theta}} \leftarrow \frac{1}{B} \sum_{i=1}^{B} \left(L(\pi_i \mid X_i) - b(X_i) \right) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\pi_i \mid X_i).$$

See the paper by Bello et al. [1] and Chapter 17 of the CS 229 lecture notes for more details.

3 Next time

Next time, we will analyze the performance of this ML approach to TSP!

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.

[3] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.