Routing Paper

Wyatt Jones

University of Iowa

September 18, 2018

1 Introduction

(Silver et al., 2016)

(Silver et al., 2017)

(Hessel et al., 2017)

Talk about how RL is very general method to solve DP problems, is really cool, and can do amazing things (GO, ATARI) Talk about application to OR with TSP and how it is a big problem, talk about how feas constraint is annoying, and large cont state space, very applicable to a lot of problems Why isnt it applied more? Discuss the problems that arise when doing RL research initial policy parameterization matters, many hyperparamters, hard to select network architecture, hard to evaluate if the architecture is capable of learning the policy (SR vs RL), local min, sensitive to random seed, many different RL algorithms with new advances happening frequently, training is computationally intensive and so cant try everything, people dont write about what didnt work.

A major complication when trying to select the appropriate architecture for your neural network that will be trained using RL is that there is often not the necessary feedback to iteratively improve the architecture until the network is training optimally. This is due to the subset of architectures that will effectively train on a given problem is very small compared to the number of potential architectures and that when the architecture is not close enough the feedback that the researcher observes is that the network simply doesn't improve while training is taking place. This makes it difficult to evaluate the value of different changes to NNA when the NN is just not learning.

The researcher can use SL to evaluate the value of a given NNA with the hope that if a given NNA can be trained using SL on a smaller problem then it will work using RL on the larger problem that is of importance. In order to study this I used SL to train several different NNA and then used RL to try and train the same NNA and compared the results.

2 Previous Work

The Traveling Salesman Problem (TSP) is a well studied combinatorial optimization problem and many exact and approximate methods have been developed to solve it. The problem was first formulated in 1930 and it is often used as a benchmark for many optimization methods. The current best exact dynamic programming algorithm for TSP has a complexity of $\mathcal{O}(2^n n^2)$ which makes it infeasible on problems larger than 40 points. While calculating the optimal route exactly is infeasible for most problems many approximate methods that rely on heuristic approaches have been able to perform well on problem instances with much more than 40 points. Modern TSP solvers, which rely on handcrafted heuristics that determine how to navigate the space of feasible policies efficiently are able to solve TSP instances with thousands of points.

Google's vehicle routing problem solver (OR-Tools) relies on a combination of local search algorithms and metaheuristics specifically crafted for the TSP. OR-Tools applies hand-engineered heuristics such as 2-opt to move from one potential solution to the next and then uses a metaheuristic such as guided local search, which escapes local minima by putting a penalty on solution features that are known to decrease performance (Bello, Pham, Le, Norouzi, & Bengio, 2016).

While these methods are very successful in solving the TSP the search heuristics are often not successful for newly encountered problems. Since search algorithms have the same performance when averaged over all problems, each application must carefully select the appropriate heuristics for each problem. The difficulty of finding the best heuristics for each problem has lead to research into finding a more general approach.

Motivated by recent success in using recurrent neural networks (RNN) to train on problems with a sequence for input and output researchers have studied the application of RNN's to the TSP (Sutskever, Vinyals, & Le, 2014).

Major advancements in the field of neural machine translation has yielded neural network architectures that are able to translate a sentences which are encoded as sequences from one language to another (Bahdanau, Cho, & Bengio, 2014).

Training a artificial neural network is most commonly done using label of the correct output however, for the TSP the use of labels would rely on another method that is close to optimal to determine the route that would then be used as a label. Reinforcement learning offers an alternative method to train the network which does not require labels and instead is trained by interacting with an environment and attempting to update parameters so that the reward obtained from the network's policy is maximized. Recent improvements in this field such as increasing performance by combining multiple methods (Hessel et al., 2017), increasing the ability to use available computational resources (Babaeizadeh, Frosio, Tyree, Clemons, & Kautz, 2016), and improving sample efficiency (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017) has lead to researchers attempting to use reinforcement learning on new problems.

Recent research on the use of reinforcement learning to solve the TSP began with (Vinyals, Fortunato, & Jaitly, 2015) who introduced a network architecture called a Pointer Network. (Bello et al., 2016) further improved the performance of the architecture presented by (Vinyals et al., 2015) through several different small architecture changes and various post processing steps.

3 Experimental Design

In this section I propose an experimental design in order to study the affect that Neural Net architecture and hyperparameter choice has on both supervised and reinforcement learning. I will describe the set of Neural Net architectures and hyperparameters that will be evaluated and the methods used to evaluate the performance of each method. In order to parameterize the policy function I will use a Recurrent Neural Network with LSTM cells. This method is advantageous since it is able construct the policy iteratively rather than determining the route in one step. This is done by letting the probability of a given route be determined by the following

$$p_\theta(\pi|s) = p_\theta(y_1, \dots, y_T|x_1, \dots, x_T) = \prod_{t=1}^{T} p_\theta(y_t|y_1, \dots, y_{t-1}, c)$$

Let $x = (x_1, \dots, x_T)$ be a sequence where x_i is the x-y coordinates for point i, y_t is the id of the point traveled to at time t and c is a context vector. In order to prevent the policy from choosing a previously chosen location I store in memory a record of each location chosen and then penalize the probability of moving to that location so that it cannot be chosen.

For each Neural Net architecture and each hyperparameter choice the method will be trained using both supervised and reinforcement learning. The loss for supervised learning will be the cross entropy loss between the parameterized policy and an near optimal policy provided by using the TABU search from Google's OR-Tools. Reinforcement learning will be done using the GA3C algorithm Babaeizadeh et al. (2016).

Each method will be optimized using the ADAM optimizer as is standard in the literature with α varying between 1e-2, 1e-3, and 1e-4, and fixing $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon =$ 1e-8. I also examine how the number of LSTM cells affects the training process by training the neural networks with 32, 64, and 128 LSTM cells.

The methods will be evaluate by determining how many times longer the parameterized policies route is compared to the length of a route found using TABU search. I will use the average relative length for the last 10,000 problem instances that the method evaluated after training for 36 hours using parallelized using 16 cores from Intel's Haswell CPU architecture. Each method is also run three times using different initial RNG seeds.

3.1 Unidirectional Encoder/Decoder

The first major Neural Network architecture that I use is directly from the Neural Machine Translation (NMT) literature (Sutskever et al., 2014). This architecture which is designed to be trained using supervised learning to translate sentences from one language to another uses a paramterization that consists of an encoder and decoder. The encoder maps the input sequence x into a context vector c and then the decoder maps the context vector to the final output. To construct the context vector c the following formula is used

$$h_t = f(x_t, h_{t-1})$$

where f is an LSTM and $c = h_T$. Once the encoder has constructed the context vector c the decoder then determines the output probability at each step using the formulas

$$p_\theta(y_t|y_1, \ldots, y_{t-1}, c) = g(y_{t-1}, s_t, c)$$

$$s_t = f(s_{t-1}, y_{t-1}, c_t)$$

where $g(y_{t-1}, s_t, c)$ is the softmax of the output of the decoder's LSTM cells and $f(s_{t-1}, y_{t-1}, c_t)$ is how the LSTM cell updates its hidden state. In this paper $p_\theta(y_t|y_1, \ldots, y_{t-1}, c)$ is the probability that the route moves to the location y in step t.

3.2 Bidirectional Encoder/Decoder

The second major Neural Network architecture that I use the bidirectional Encoder/Decoder framework presented in Bahdanau et al. (2014). The Bidirectional encoder consists of forward and backward RNN's. The forward RNN reads the input sequence in order from $t = 1$ to T and the backward reads the input sequence from T to $t = 1$. The hidden state h_j is then constructed by concatenating the two vectors $h_j = [\overrightarrow{h}_j^\top; \overleftarrow{h}_j^\top]$

3.3 Pointer Network

The final architecture is from Vinyals et al. (2015) where there is no longer an encoder and the probability of moving to the next point is simplified to the following

$$p_\theta(y_t|y_1, \ldots, y_{t-1}, c) = \text{softmax}(c)$$

where h_t is the LSTM's hidden state at step t and c is the context vector. The context vector is calculated using an attention layer.

3.4 Attention Layer

The NMT literature found that introducing a mechanism that will focus on part of the input sequence while decoding would increase performance and Vinyals et al. (2015) found that due to the spatial nature of the TSP that ... NOT SURE HOW TO INTRODUCE THIS

In this paper I will study the two most common attention mechanisms and how the impact the performance of each method. The output of the attention mechanism is a new context vector c that is then input into the decoder. Each attention mechanism is constructed with the following formulas

$$c_t = \sum_{j=1}^{T} \alpha_{tj} h_j$$

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{T} \exp(e_{tk})}$$

$$e_{tj} = a(s_{t-1}, h_j)$$

where t represents the current step during decoding, k represents the location in the input sequence, and a is a specific attention mechanism. The first specific attention mechanism that I use is the Bahanadu Attention Mechanism from Bahdanau et al. (2014) where e_{tj} is computed by

$$e_{tj} = v_\alpha^\top \tanh(W_\alpha s_{t-1} + U_\alpha h_j)$$

where $v_\alpha \in \mathbb{R}^n, W_\alpha \in \mathbb{R}^{n\times n}, U_\alpha \in \mathbb{R}^{n\times 2n}$ are weight matricies.

The second attention mechanism is the Luong Attention Mechanism from Luong, Pham, and Manning (2015) where e_{tj} is computed by

$$e_{tj} = s_{t-1}^\top W_\alpha h_j$$

where $W_\alpha \in \mathbb{R}^{n\times n}$ is a weight matrix.

4 Results

I this section I analyze the main experimental results. First I show that when training using supervised learning that as the Neural Network Architecture becomes more specialized to the problem of interest there is an increase in performance per hour holding all else constant. Then I show that the quicker that the method trained with supervised learning leads to an increase in performance when trained using reinforcement learning and that many methods that could be trained using supervised learning were not able to be trained using reinforcement learning.

4.1 Supervised Learning

4.1.1 Hyperparameter and Architecture Choice

In Figure 1 the results from training the 35 different methods is presented. It shows that while the methods that used a unidirectional and bidirectional encoder were able to achieve near optimal results on the TSP 20 that they had very inconsistent performance when hyperparameters were slightly changed. This can be seen by comparing 200, 202, and 205 where the learning rate was changed and the performance was changed significantly.

Figure 1 also shows that unidirectional encoder/decoder was only able to train using the Luong Attention mechanism, the bidirectional encoder/decoder was able to be trained with both attention mechanisms and the pointer network was only able to be trained using the Bahanadu attention mechanism. While the experimental evidence shows that the best run for the unidirectional and bidirectional encoder out performed the best run for the pointer network, the pointer network was more robust to changes in the hyper parameters and was more consistently able to achieve a performance within 10% of the optimal route length.

In Figure 1 we can see that while the best run that used a unidirectional encoder/decoder architecture was able to achieve optimal results the average performance improvement per hour of training time is less the average for methods that used a pointer network architecture. This suggests that the pointer network is more consistently able to achieve performance improvements to paramaterized policy during the course of training than the encoder/decoder framework is.

4.1.2 In Sample Variance

The experiment also provided evidence of significant in sample variance. Identical runs of a method often yielded very different results when only the initial random seed was changed. This can be seen in Figure 3 where the relative length of multiple different runs of the best performing Unidirectional Encoder/Decoder architecture are plotted over the time that the method was trained. This plot shows that a method that can yield optimal performance can have difficulty reproducing the same performance when rerun. This in sample variance is also present in the best performing Bidirectional Encoder/Decoder architecture which can be seen in Figure 4. While the Unidirectional and Bidirectional Encoder/Decoder architectures exhibited a high degree of in sample variance the pointer network does not. Figure 5 shows that the best performing Pointer network architecture is able to train consistently to a low relative length but did not have a run that performed as well as the best Unidirectional or Bidirectional Encoder/Decoder architectures.

The in sample variance that can occur with these methods can be a major problem for reproducibility since even if the architecture and hyperparameters are the same different seeds for the RNG can cause significant differences in performance for the method. The in sample variance also is a problem for further research in this field since the evaluation of a newly proposed architecture can be noisy which makes tuning hyperparameters difficult. These problems could be avoided by performing many sample runs and reporting both the best case and average case for the method however, this can be extremely computationally intensive and many researchers choose to not report the frequency for which their architecture trained successfully.

4.2 Reinforcement Learning

In Figure 2 the average relative length of the best run for each method when trained using reinforcement learning is presented. This table shows that the Unidirectional and Bidirectional Encoder/Decoder networks failed completely to train using reinforcement learning for any hyperparameter setting. Figure 6 and Figure 7 show that while these methods were able to be trained using supervised learning their performance when first initialized is that of a random policy and that there is no improvement throughout the course of training. This behavior is consistent for all the different hyperparameter choices for the Unidirectional and Bidirectional Encoder/Decoder architectures. These results show that evaluating the affect that different hyperparameters have on an architecture's performance is difficult to determine. This suggests that hyperparameter tuning is not possible when the neural network architecture is not specialized enough for the problem of interest so that reinforcement learning is possible. This can be a significant barrier for research in this area since when evaluating the performance of a newly proposed network architecture it is not possible to determine if poor performance is due to bad hyperparameter choices or if the issue is with the network architecture. Additionally, the high computational cost of each sample run can prohibit using the average of multiple runs to evaluate the effect that hyperparameters have on performance.

Figure 8 presents the relative length of multiple different runs for the best performing Pointer Network architecture. This figure shows that once the neural network architecture is specialized enough for a given problem and the hyperparameters are tuned reinforcement learning can be used to learn the optimal policy function in combinatorial optimization problems. These results replicate the findings of Bello et al. (2016).

Figure 9 shows that a training is not always monotonic and that the sample variance observed in the supervised learning is present again for reinforcement learning. This shows that a single training run of a neural network may not be representative of the potential performance and so multiple runs for each architecture are needed to evaluate performance.

5 Conclusion

(Main Idea is to show how it is necessary to build a specific architecture to solve specific problems) while the method is general it's implementation is anything but general (Sub Ideas SL can be used to narrow subset but not all SL will work for RL)

Tables

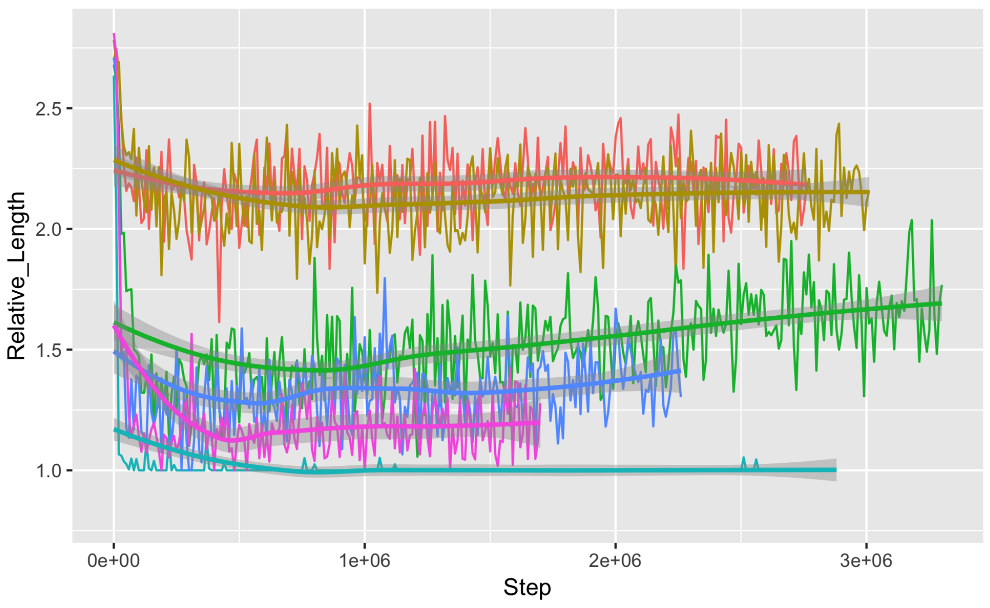

Figure 3: Multiple Training Runs of the Unidirectional Encoder/Decoder Using Supervised Learning

Attention=Luong, LR=1e-3, of cells=128

Architecture	Attention	# of cells	LR	Relative Length
Unidirectional	Luong	128	1e-2	2.77
Unidirectional	Luong	128	1e-3	1.00
Unidirectional	Luong	128	1e-4	1.99
Unidirectional	Luong	32	1e-3	2.13
Unidirectional	Luong	64	1e-3	1.01
Unidirectional	Bah	128	1e-2	2.76
Unidirectional	Bah	128	1e-3	2.43
Unidirectional	Bah	128	1e-4	2.45
Unidirectional	Bah	32	1e-3	2.66
Unidirectional	Bah	64	1e-3	2.61
Bidirectional	Luong	128	1e-2	2.28
Bidirectional	Luong	128	1e-3	1.00
Bidirectional	Luong	128	1e-4	2.08
Bidirectional	Luong	32	1e-3	1.50
Bidirectional	Luong	64	1e-3	1.01
Bidirectional	Bah	128	1e-2	2.77
Bidirectional	Bah	128	1e-3	1.50
Bidirectional	Bah	128	1e-4	2.17
Bidirectional	Bah	32	1e-3	1.00
Bidirectional	Bah	64	1e-3	2.38
Pointer	Luong	128	1e-2	2.22
Pointer	Luong	128	1e-3	1.12
Pointer	Luong	128	1e-4	1.54
Pointer	Luong	32	1e-3	1.51
Pointer	Luong	64	1e-3	1.57
Pointer	Bah	128	1e-2	1.21
Pointer	Bah	128	1e-3	1.06
Pointer	Bah	128	1e-4	1.07
Pointer	Bah	32	1e-3	1.11
Pointer	Bah	64	1e-3	1.07

Figure 1: Table of Supervised Learning Experimental Results

Architecture	Attention	# of cells	LR	Relative Length
Unidirectional	Luong	128	1e-2	2.75
Unidirectional	Luong	128	1e-3	2.74
Unidirectional	Luong	128	1e-4	2.84
Unidirectional	Luong	32	1e-3	2.64
Unidirectional	Luong	64	1e-3	2.74
Unidirectional	Bah	128	1e-2	2.76
Unidirectional	Bah	128	1e-3	2.73
Unidirectional	Bah	128	1e-4	2.74
Unidirectional	Bah	32	1e-3	2.71
Unidirectional	Bah	64	1e-3	2.74
Bidirectional	Luong	128	1e-2	2.77
Bidirectional	Luong	128	1e-3	2.75
Bidirectional	Luong	128	1e-4	2.71
Bidirectional	Luong	32	1e-3	2.73
Bidirectional	Luong	64	1e-3	2.74
Bidirectional	Bah	128	1e-2	2.69
Bidirectional	Bah	128	1e-3	2.74
Bidirectional	Bah	128	1e-4	2.75
Bidirectional	Bah	32	1e-3	2.74
Bidirectional	Bah	64	1e-3	2.75
Pointer	Luong	128	1e-2	2.65
Pointer	Luong	128	1e-3	1.03
Pointer	Luong	128	1e-4	1.12
Pointer	Luong	32	1e-3	1.15
Pointer	Luong	64	1e-3	1.04
Pointer	Bah	128	1e-2	1.11
Pointer	Bah	128	1e-3	
Pointer	Bah	128	1e-4	
Pointer	Bah	32	1e-3	
Pointer	Bah	64	1e-3	

Figure 2: Table of Reinforcement Learning Experimental Results

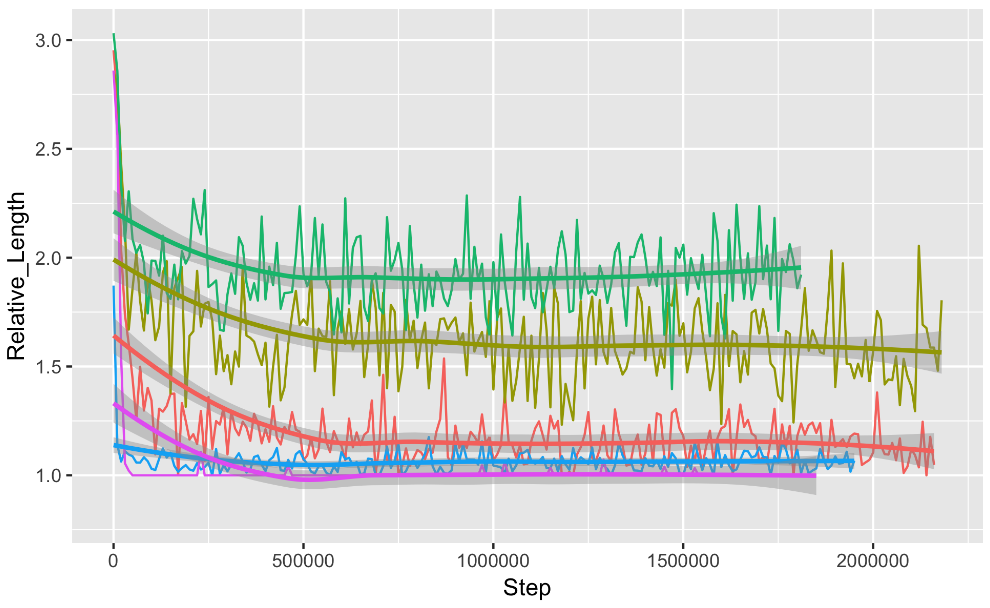

Figure 4: Multiple Training Runs of the Bidirectional Encoder/Decoder Using Supervised Learning

Attention=Luong, LR=1e-3, of cells=128

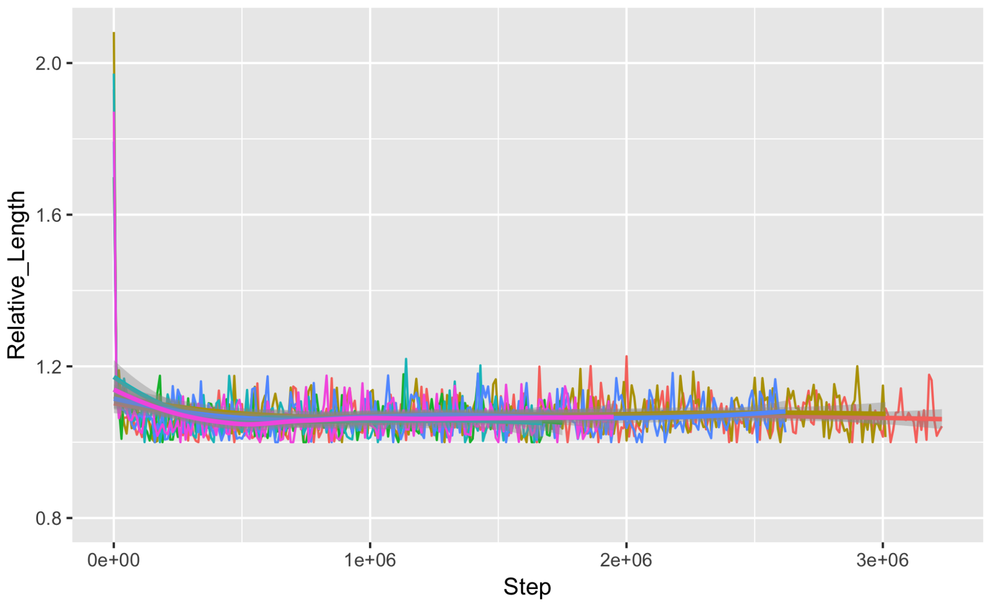

Figure 5: Multiple Training Runs of the Pointer Network Using Supervised Learning

Attention=Bah, LR=1e-3, of cells=128

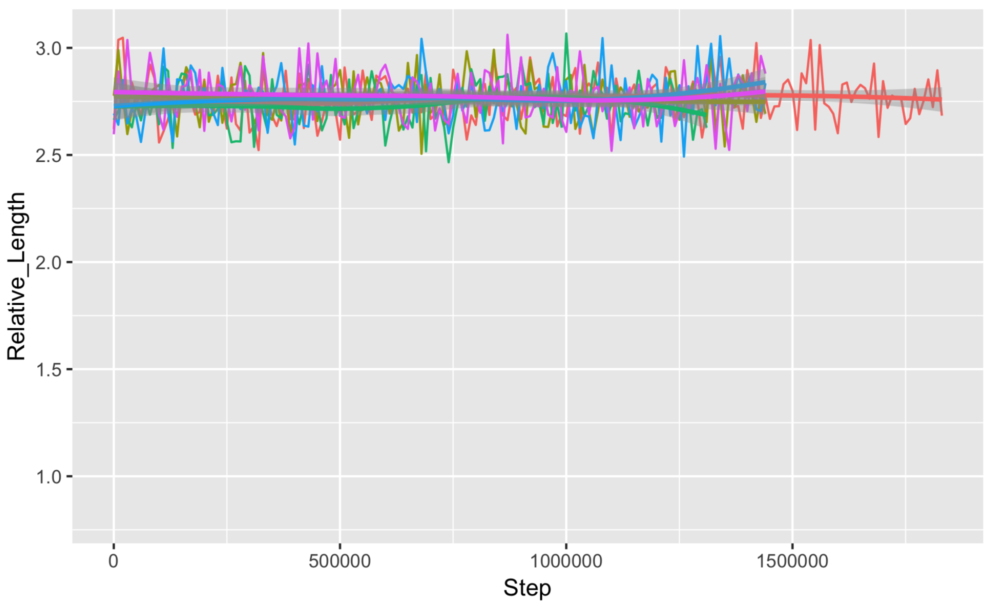

Figure 6: Multiple Training Runs of the Unidirectional Encoder/Decoder Using Reinforcement Learning

Attention=Luong, LR=1e-3, of cells=128

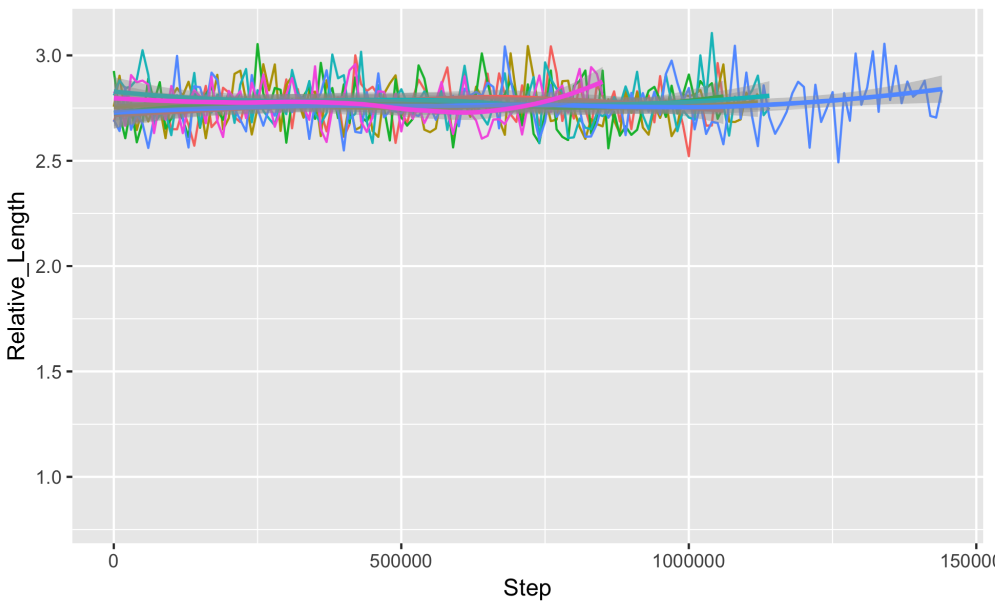

Figure 7: Multiple Training Runs of the Bidirectional Encoder/Decoder Using Reinforcement Learning

Attention=Luong, LR=1e-3, of cells=128

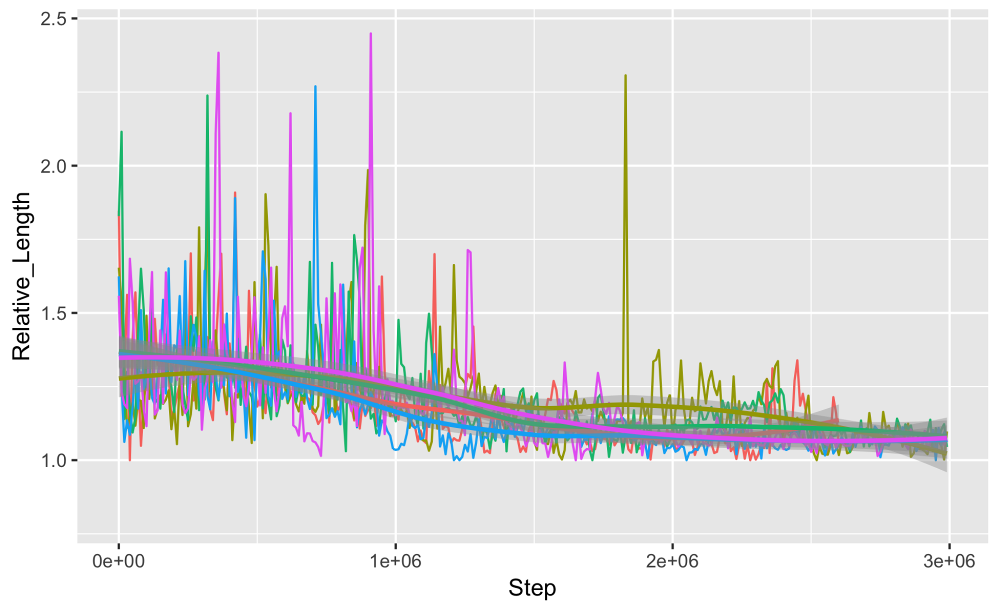

Figure 8: Multiple Training Runs of the Pointer Network Using Reinforcement Learning
Attention=Luong, LR=1e-3, of cells=128

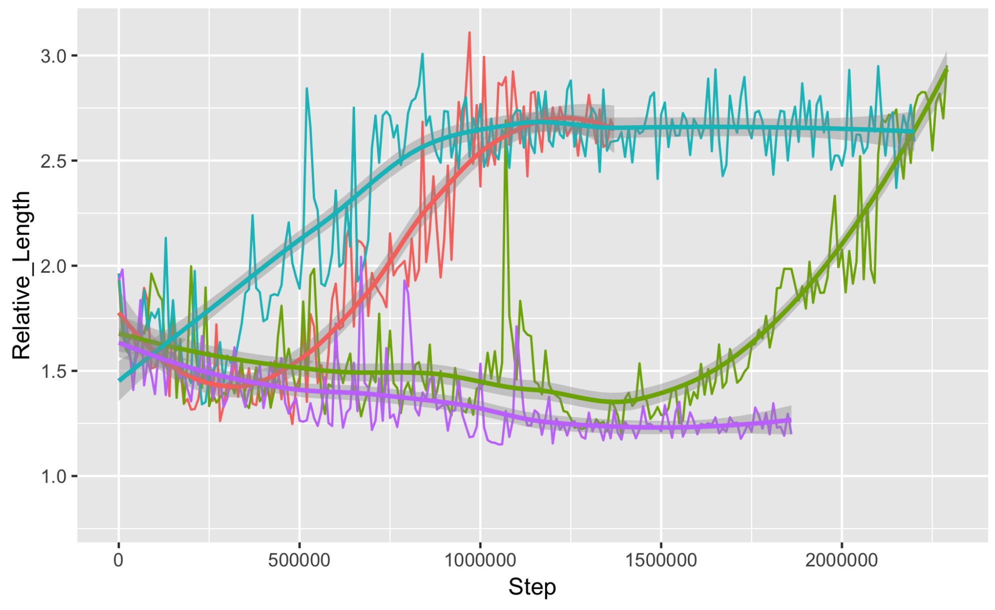

Figure 9: Multiple Training Runs of the Pointer Network Using Reinforcement Learning
Attention=Luong, LR=1e-4, of cells=128

Appendix 1: The Policy Gradient Method

The policy gradient theorem states that if we let

$$J(\theta) \doteq v_{\pi_\theta}(s_0)$$

denote the true value function for π_θ, the policy determined by θ then

$$\nabla J(\theta) = \sum_s \mu_\pi(s) \sum_a \nabla \pi_\theta(a|s) r_{\pi/theta}(s,a)$$

where $r_\pi(s,a)$ is the reward from taking action a in state s, and $\mu_\pi(s)$ is the expected number of time steps t on which $S_t = s$ given s_0 and following π_θ (Sutton, Barto, et al., 1998). This theorem is valuable since we can sample this expected value which is done in the REINFORCE algorithm where

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\beta^t R_t \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}\Big]$$

where R_t is the observed reward from step t to the end, A_t is the sampled action in the sampled state S_t, and β is the discount parameter (Williams, 1992). Since software implementations of neural networks use Autograd which will automatically analytically compute the gradient of the network with respect to its trainable parameters θ through backpropagation this equation is now easy to compute.

Improvements to this algorithm include the addition of a separate trainable network called a critic network. This network is used to decrease the variance of the policy parameter update. The addition of a critic network changes the gradient calculation to

$$\nabla_\theta J(\theta|s) = \mathbb{E}_{\pi \sim p_\theta(.|s)}\Big[(L(\pi|s) - \hat{v}_\phi(s))\nabla_\theta \log p_\theta(\pi|s)\Big]$$

where $\hat{v}_\phi(s)$ is a neural network parameterized by ϕ.

Algorithm 1 Actor-Critic Method

```
1: procedure Given $p_\theta(.|s)$, $\hat{v}_\phi(s)$ and step sizes $\alpha_1, \alpha_2$
2:     Initialize policy parameter $\theta$ and state-value parameters $\phi$
3:     while Training do
4:         Generate an episode $S_0, A_0, r_1, \ldots, S_{T-1}, A_{T-1}, r_T$ following $\pi_\theta(\cdot|\cdot)$.
5:         for $t = 1, \ldots, T$ do
6:             $R_t \leftarrow$ return from step $t$
7:             $\phi \leftarrow \phi + \alpha_1(R_t - \hat{v}_\phi(S_t))\nabla_\phi \hat{v}_\phi(S_t)$
8:             $\theta \leftarrow \theta + \alpha_2 \beta^t (R_t - \hat{v}_\phi(S_t))\nabla_\theta \log(\pi_\theta(A_t|S_t))$
9:         end for
10:    end while
11: end procedure
```

This method can be use parallelization in order to more efficiently use multiple CPU threads. This method was introduced in Mnih et al. (2016) and is called Asynchronous Advantage Actor-Critic (A3C) and has been shown to increase the speed for which the policy can be trained and the parallelization has also been shown to help stabilize the parameter updates. See (Sutton et al., 1998) for a more detailed introduction to the REINFORCE, and Actor-Critic methods and see (Mnih et al., 2016) for a detailed description of A3C.

Appendix 2: Recurrent Neural Networks

A recurrent neural network (RNN) is a class of artificial neural networks where output of a cell is directed back as input in to the cell again. This allows the network to exhibit temporal dynamic behavior for a sequence. Unlike feedforward neural networks RNN's maintain a hidden state h_t that is changed as the network iteratively processes input which allows the network to process sequences.

In this paper I use long short-term memory (LSTM) as the central component of my RNN. LSTM cells have parameterized functions for processing the input x_t, updating the cell's hidden state h_t, and determining the output of the cell y_t (Hochreiter & Schmidhuber, 1997). These functions are called the input gate, forget gate, and output gate respectively. The input gate is a sigmoid layer which decides which elements will be potentially added to the current hidden state as a function of the previous hidden state and the current input. Next the forget gate is composed of a sigmoid layer that decides which components of the cell's hidden state to keep for the next step as a function of the previous cell's hidden state and the current input. Finally, the output gate determines the output of the cell as a function of the input and the new hidden state. The parameters of these functions are then trained to minimize a loss function using a gradient decent optimizer such as ADAM.

Please see Understanding LSTM Networks for an introduction to LSTM Networks.

References

Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., & Kautz, J. (2016). Ga3c: Gpu-based a3c for deep reinforcement learning. *CoRR abs/1611.06256*.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., & Bengio, S. (2016). Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928–1937).

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... others (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... others (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (pp. 3104–3112).

Sutton, R. S., Barto, A. G., et al. (1998). *Reinforcement learning: An introduction.* MIT press.

Vinyals, O., Fortunato, M., & Jaitly, N. (2015). Pointer networks. In *Advances in neural information processing systems* (pp. 2692–2700).

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, *8*(3-4), 229–256.