# Routing Paper

Wyatt Jones

University of Iowa

September 11, 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 much larger problem instances. 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. (Applegate, Bixby, Chvatal, & Cook, 2006) introduced the Concorde Algorithm which makes use of cutting plane algorithms is regarded as the current best TSP solver and ...

(Sutskever, Vinyals, & Le, 2014a)

(Sutskever, Vinyals, & Le, 2014b)

(Vinyals, Fortunato, & Jaitly, 2015)

(Bello, Pham, Le, Norouzi, & Bengio, 2016)

(Nazari, Oroojlooy, Snyder, & Takáč, 2018)

# 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

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

Let $x = (x_1, \ldots, x_T)$ 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, Frosio, Tyree, Clemons, and Kautz (2016).

Each method will be optimized using the ADAM optimizer as is standard in the literature with $\alpha$ 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., 2014a). 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, Cho, and Bengio (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

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

5

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 lead 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 ? 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 ? 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

6

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.

Figure 1: Plot of average rel len for 205 and 220

In Figure ? 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.
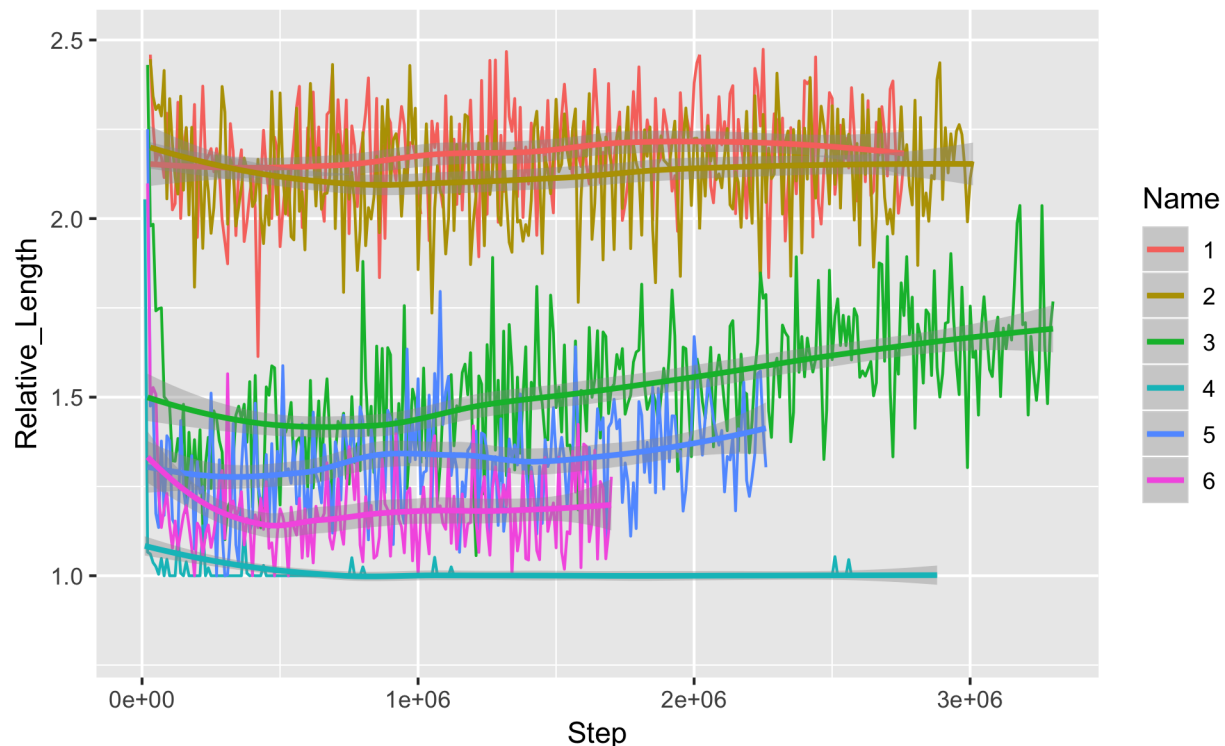
### 4.1.2 In Sample Variance



Figure 2: Multiple Training Runs of the Unidirectional Encoder/Decoder Attention=Luong, LR=1e-3, of cells=128
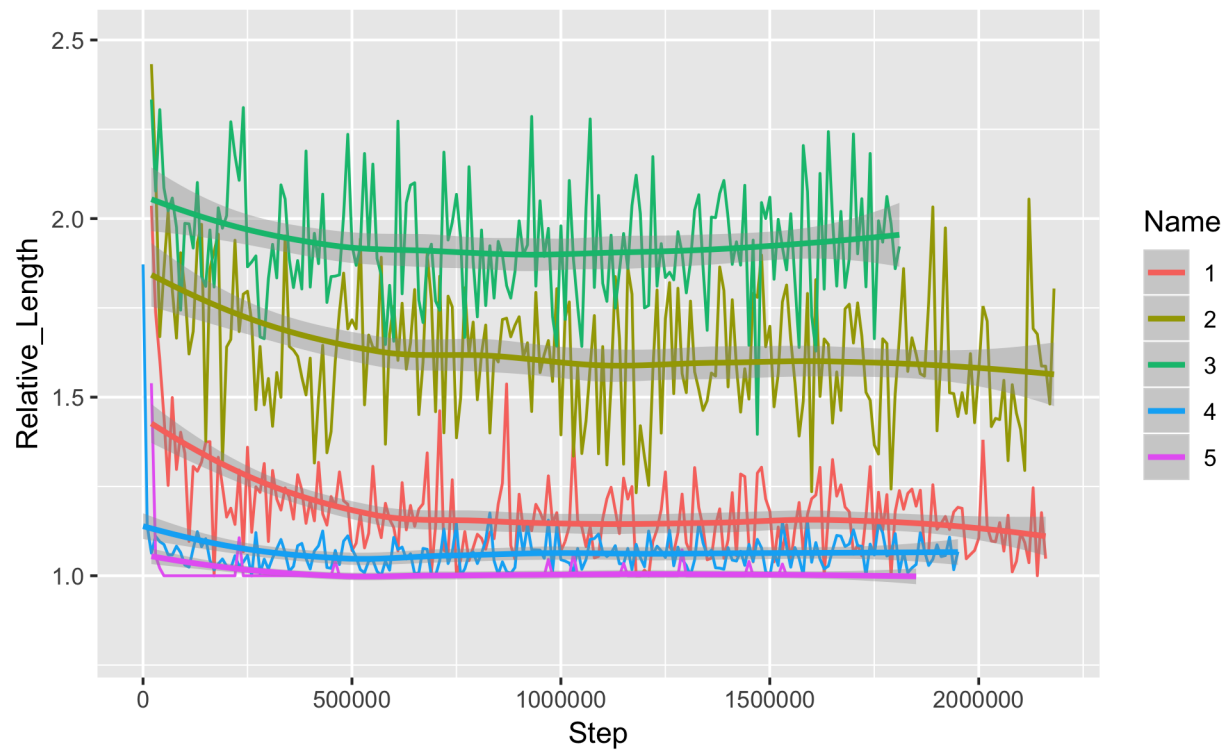
Figure 3: Multiple Training Runs of the Bidirectional Encoder/Decoder
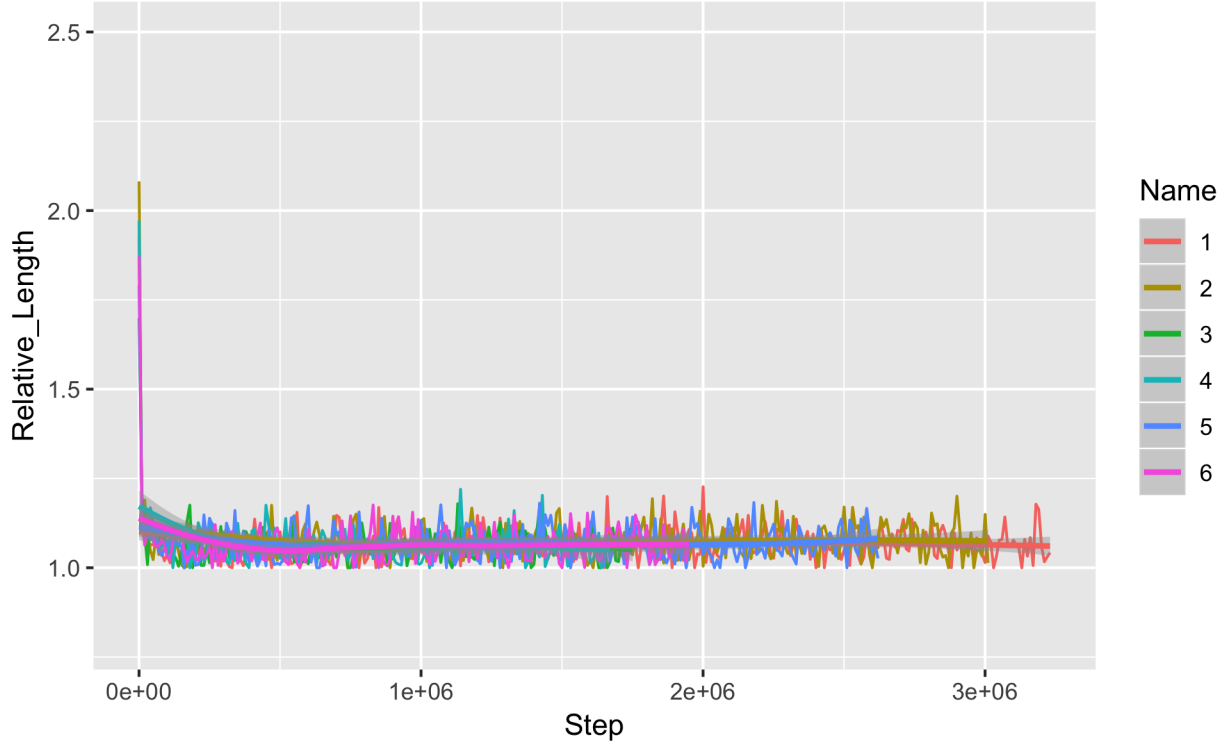Attention=Luong, LR=1e-3, of cells=128

Figure 4: Multiple Training Runs of the Pointer Network
Attention=Bah, LR=1e-3, of cells=128

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 ? where the relative length of 3 different runs of 205 are plotted over the time 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 is 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

never worked ideas PPO, moving average, sequence cost, use time, direction 4 or 6, beam search, PCA

## 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

## 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 $\pi_\theta$, the policy determined by $\theta$ 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 $\pi_\theta$ (Sutton, Barto, et al., 1998). This theorem is valuable since we can sample this expected value which is done in the REINFORCE algorithm where

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

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 $\beta$ is the discount parameter (Williams, 1992). Since software implementations of neural networks use Autograd which will automatically analytically compute

| 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 5: Table of Supervised Learning Experimental Results

| Architecture | Attention | # of cells | LR | Relative Length |
| --- | --- | --- | --- | --- |
| Unidirectional | Luong | 128 | 1e-2 | |
| Unidirectional | Luong | 128 | 1e-3 | |
| Unidirectional | Luong | 128 | 1e-4 | |
| Unidirectional | Luong | 32 | 1e-3 | |
| Unidirectional | Luong | 64 | 1e-3 | |
| Unidirectional | Bah | 128 | 1e-2 | |
| Unidirectional | Bah | 128 | 1e-3 | |
| Unidirectional | Bah | 128 | 1e-4 | |
| Unidirectional | Bah | 32 | 1e-3 | |
| Unidirectional | Bah | 64 | 1e-3 | |
| Bidirectional | Luong | 128 | 1e-2 | |
| Bidirectional | Luong | 128 | 1e-3 | |
| Bidirectional | Luong | 128 | 1e-4 | |
| Bidirectional | Luong | 32 | 1e-3 | |
| Bidirectional | Luong | 64 | 1e-3 | |
| Bidirectional | Bah | 128 | 1e-2 | |
| Bidirectional | Bah | 128 | 1e-3 | |
| Bidirectional | Bah | 128 | 1e-4 | |
| Bidirectional | Bah | 32 | 1e-3 | |
| Bidirectional | Bah | 64 | 1e-3 | |
| Pointer | Luong | 128 | 1e-2 | |
| Pointer | Luong | 128 | 1e-3 | |
| Pointer | Luong | 128 | 1e-4 | |
| Pointer | Luong | 32 | 1e-3 | |
| Pointer | Luong | 64 | 1e-3 | |
| Pointer | Bah | 128 | 1e-2 | |
| Pointer | Bah | 128 | 1e-3 | |
| Pointer | Bah | 128 | 1e-4 | |
| Pointer | Bah | 32 | 1e-3 | |
| Pointer | Bah | 64 | 1e-3 | |

Figure 6: Table of Reinforcement Learning Experimental Results

the gradient of the network with respect to its trainable parameters $\theta$ 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 $\phi$.

---

**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

Applegate, D., Bixby, R., Chvatal, V., & Cook, W. (2006). *Concorde tsp solver*.

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).

Nazari, M., Oroojlooy, A., Snyder, L. V., & Takáč, M. (2018). Deep reinforcement learning for solving the vehicle routing problem. *arXiv preprint arXiv:1802.04240*.

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. (2014a). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (pp. 3104–3112).

Sutskever, I., Vinyals, O., & Le, Q. V. (2014b). 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.