Using Pointer Networks to Solve the Traveling Salesman Problem

Rahul Kharse (rkharse1)
Alexander Chang (achang56)

3/23/19

Abstract

Sequence to sequence models are a powerful tool leverage for supervised deep learning tasks such as machine translation in order to develop more accurate models to transform one input to its equivalent output with as less noise as possible. Attention is an neural net augmentation that improves upon the success of sequence to sequence models to better focus the models task to transform more relevant data so that is not lost while going through further time steps. The combination of the two towards solving combinatorial optimization problems, where input sequence of graph data are transformed into optimized ordered sequences, such as shortest tour, have provided decent results as well. However, these problems suffer from the limitation of fixed input size. In this paper we explore the implementation of pointer networks, which creatively use attention to bypass this limitation and analyze the results specifically towards solving the TSP problem.

1 Introduction

RNNs are a natural generalization of feed-forward neural networks to sequences. The introduction of sequence-to-sequence (seq2seq) models uses RNNs to solve problems whose input and the output sequences have varying lengths and feature complicated and non-monotonic relationships. In the seq2seq model, one RNN maps the input sequence to a fixed-sized vector as an embedding representation, and then another RNN maps the vector to the target sequence. [1]

As an improvement upon the traditional seq2seq model [2], Bahdanau et. al. introduced a content-based attention mechanism that helps sequence-to-sequence models track long-term dependencies and translate long sentences correctly. This attention mechanism allows the decoder focus in on useful parts of the input. These developments have allowed RNNs to be successfully used to solve problems in multiple domains, from machine translation [1] to solving combinatorics problems [2].

In 2015, Vinyals et. al [2] showed that combinatorial optimization problems are solvable by using a supervised learning approach that uses a sequence-to-sequence model with attention. In the standard sequence-to-sequence with attention, outputs are chosen from a dictionary of fixed size. This results in the standard seq2seq model having the limitation that a new model would have to be trained for every input size possible for the combinatorial optimization problem (which is infinite). [2] proposed a simple yet clever solution to address this limitation by using the attention mechanism proposed in [3] as pointers to the input elements. This novel architecture, called a pointer network, has been shown to provide approximate solutions to a variety of combinatorial optimization problems such as the computation of planar convex hulls, solving Delaunay triangulations, and solving the symmetric planar Travelling Salesman Problem (TSP).

Our main focus will be to convey how to best train a pointer network to solve the 2D symmetric Traveling Salesman Problem (TSP). Since, TSP's output sequence length is dependent on the length of its input sequence, and outputs cannot be chosen from a known vocabulary (there are an infinite number of elements in $\mathbb{R}^2$) and must be chosen from the input sequence, pointer networks serve to be the best choice known today for solving TSP using a supervised deep learning approach.

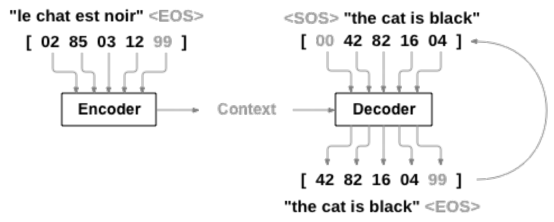
2 Models

In this project, we test sequence-to-sequence models described in [1] with the modification of the attention mechanism described in [2].

2.1 Sequence to Sequence

In a traditional sequence-to-sequence model, an encoder RNN maps the input sequence $x = \{x^{<1>}, ..., x^{<T_x>}\}$ to a fixed size vector, e, representing the input. This vector, e, is called the context vector. The context vector e fed into a decoder RNN network to start off the network, instead of a vector of all zeros in a regular language model.

Figure 1: An RNN encoder processes the input sequence and outputs a context vector that is used in the decoder to generate the output sequence



The sequence-to-sequence model can be thought of as a conditional probability model where the decoder is learning the probability distribution of the output sentence $y = \{y^{<1>}, ..., y^{<T_x>}\}$ conditioned on some input sentence using the probability chain rule.

$$p(y^{<1>}, ..., y^{<T_y>}|x^{<1>}, ..., x^{<T_x>}) = \Pi_{t=1}^{T_y} p(y^{<t>}|y^{<t-1>}, ..., y^{<1>}, e) \tag{1}$$

The way to train this would be to find the output sequence, y, that maximizes the conditional probability.

$$\underset{y}{\operatorname{argmax}}\, p(y^{<1>}, ..., y^{<T_y>}|x^{<1>}, ..., x^{<T_x>}) \tag{2}$$

In our project, we performed a test using two different optimization algorithms, Adam [4] and RMSProp [5] to maximize this probability. The Adam optimization algorithm is supposed to be a combination of the Momentum [6] and RMSProp algorithms. The intuition behind Momentum is that it smooths out oscillations when we are traversing hyper-ravines of the cost function. These oscillations slow down gradient descent and prevent us from using a larger learning rate - because we might oscillate out of the minima. By adding momentum, it allows the gradient to build up velocity in the direction where the gradient is gentle and consistent.

RMSProp also smooths out oscillations; however it is different because it is an adaptive learning rate method, where the learning rate changes based on magnitudes of past gradients. Although the algorithm remains unpublished, RMSProp has been shown empirically to provide good results when applied to machine translation [7]. By combining these two algorithms, the Adam optimization has been shown to be effective in training neural networks across multiple problems such as automatic generating of anime facial images [8] to machine translation [9].

Our project also explores how different RNN architectures affect the network in learning to model $p(y^{<1>}, ..., y^{<T_y>}|x^{<1>}, ..., x^{<T_x>})$. We explore using a Long Short Term Memory (LSTM) [10] to model this conditional probability just like in [1] and [2]. We also see how a sequence to sequence model using a gated recurrent unit (GRU) [11] can model the conditional probability. Both GRUs and LSTMs provide their own approach to mediating the vanishing gradient problem suffered by vanilla RNNs.

The vanishing gradient problem is the tendency of weights to go to zero and typically occurs in RNNs running over large sequences, since the repeated multiplication of probabilities (numbers that are less than 1 and greater than 0) results in diminishingly smaller values over time steps until floating point precision can no longer represent such small values. While neither GRUs or LSTMs fully resolve this issue, their use is standard in today's sequence-to-sequence models since they both greatly mediate it.

Lastly, we explore how well a bidirectional RNN [12] can model the conditional probability. Bidirectional RNNs function essentially the same as running an RNN on the reverse input sequence at the same time as running another RNN on normal input sequence. This gives the RNN both forward and backward sequence information to condition on, which can better embed input sequence information in the encoder. Of course, bidirectional RNNs cannot be used in the decoder since the future sequence elements are what we are trying to predict.

A typical implementation of a sequence-to-sequence would likely choose the output element at time t_i with the highest probability and repeat this selection strategy for every $t_i \in \{t_1, t_2, ..., t_n\}$ where at t_n the EOS string has the highest probability and is chosen. While this may seem like a reasonable strategy to achieve the highest probability sequence, it is fairly trivial to come up to a counter example. Consider, for instance, in a sequence-to-sequence model with vocabulary size 2 that at time t_i choice a has probability $P_t(w_i = a) = 0.6$ and choice b has probability $P_t(w_i = b) = 0.4$. The standard approach would choose $w_i = a$. But suppose if at time t_{i+1} choosing b at t_i would have resulted in $P_{t+1}(w_i = a) = 1$ where as the choice of a at t_i results in $P_{t+1}(w_i = a) = 0.5$ and $P_{t+1}(w_i = b) = 0.5$. Then clearly choosing b at t_i would have been the better choice.

The standard procedure, and the example of choosing a, is a greedy approach since it chooses the highest value option at each step. However, the greedy approach fails in situations such as those explained above. One solution to choosing the highest probability sequence would be to consider every possible branch to every other sequence to avoid missing considering situations such as choosing b. However, this solution is exponential in time complexity.

The alternative is to use a heuristic based approach known as beam search, in which only the top k highest probability items have their future states considered. This results in a linear time complexity in the length of the input sequence and k, since only k options are considered at each t_i. While this is still not guaranteed to give the highest probability sequence (since the best choice may have been lower than the top k elements at some t_i), it is a decent middle ground that generally give higher or equivalent results to the standard greedy approach. This is why we use beam search in out pointer net implementation, however, there are certain adjustments that need to be made for solving problems such as TSP that will be discussed in methods section 3.2 "Traveling Salesman Problem".

2.2 Attention and Pointers

Adding an attention mechanism to the vanilla sequence-to-sequence model has been shown to improve the neural network's ability to model the conditional probability during machine translation [3]. In machine translation, vanilla sequence-to-sequence models perform poorly when trying to translate long sequences. This is because when inputting in a long sentence, the encoder has to memorize the whole sentence in the form of the context vector before feeding it into the decoder, which translates the sentence. The attention mechanism alleviates this problem by using a separate internal neural network used to generate an attention vector to feed to the decoder. This attention vector allows the decoder to focus in on a particular part of the input.

The attention vector when applied our problem is computed as follows:

$$\begin{aligned} u^i_j &= v^T tanh(W_1 e_j + W_2 d_i) \;\; j \in (1, ..., n) \\ a^i_j &= softmax(u^i_j) \;\; j \in (1, ..., n) \\ d'_i &= \sum_{j=1}^{n} a^i_j e_j \end{aligned} \tag{3}$$

Here, n is the sequence length, the number of cities, of the input problem. The encoder hidden states are defined as $(e_1, ..., e_n)$ and the decoder hidden states are defined as $(d_1, ..., d_n)$. The equation for the neural network is on the first line, where W_1, W_2, and v are the parameters and $tanh$ is the neural network's activation function. After getting the output of the neural network, u^i_j, we use a softmax get the probabilities of which input to pay attention to. Lastly, we concatenate the encoder state and the attention vector to get d'_i which is then fed into the decoder.

However, in standard sequence-to-sequence, the target vocabulary size has to be defined beforehand. Therefore, the traditional sequence-to-sequence cannot generalize to problems where the output vocabulary size is dependant on the input. This is a problem when applying sequence-to-sequence to combinatorial optimization problems such as the traveling salesman problem (TSP).

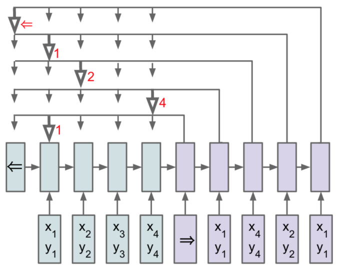
A simple modification to the attention mechanism detailed in [2] repurposes the attention mechanism to create pointers to positions in the input sequence.

The change is detailed below:

$$\begin{aligned} u^i_j &= v^T tanh(W_1 e_j + W_2 d_i) \;\; j \in (1, ..., n) \\ p(C_i | C_1, ..., C_{i-1}, \mathcal{P}) &= softmax(u^i_j) \;\; j \in (1, ..., n) \end{aligned} \tag{4}$$

Here $\mathcal{P} = \{P_1, ..., P_n\}$, the Cartesian coordinates of the points in the TSP problem. $C^{\mathcal{P}} = \{C_1, ..., C_{m^{\mathcal{P}}}\}$, are the index of points to visit in TSP. The modification to attention allows us to obtain the conditional probability of each index, given the input and previous sequence of indices.

Figure 2: In the pointer net model, the encoder processes the input and creates a context vector. The context vector is fed into the attention network along with the decoder's hidden state. Then the attention mechanism creates pointers to positions in the input sequence.



During inference, computing the *argmax* of the resulting attention vector gets us the pointers to the input sequence.

3 Methods

3.1 Traveling Salesman Problem Data

In this project, we will be focused on using pointer networks to solve planar symmetric TSP. In TSP, we are given a list of n Cartesian coordinates of cities, $\mathcal{P} = \{P_1, ..., P_n\}$. The goal is to find a tour, a path that goes through all the cities, that minimizes the total tour length. A tour, $C^{\mathcal{P}} = \{C_1, ..., C_n\}$ is a permutation of integers $1, ..., n$. In symmetric TSP, the distance between city $C_i \rightarrow C_j$ is the same as the distance from $C_j \rightarrow C_i$.

For our project we had two pipelines for obtaining the TSP data. Nevertheless, in both pipelines, the data was represented the same way. For our input data, we had a $m \times n \times 2$ array, where m was the number of examples, n was the number of cities, and the (x, y) coordinate of each city. The output data was represented as a $m \times n$ array of the tour. For each of our examples, we assumed that we start at city 0. We only trained on TSP problems with the same amount of cities, n.

The first method for obtaining the data was generating TSP solutions for tours of 10 cities. City coordinates were restricted to a square of width 20 centered at 0 and the graph connecting the cities was fully connected. We used the algorithm posted by mlalevic [13] at `https://gist.github.com/mlalevic/6222750` to solve for an optimal solution to the tour in exponential time.

The second method we obtained data was by downloading the dataset that [2] used located at `http://goo.gl/NDcOIG`. This dataset consisted of 2 solved combinatorial optimization problems: convex hull and TSP. We used this data for 5 and 20 cities. The TSP data was generated by randomly picking points within a $[0, 1] \times [0, 1]$ square. The data was split into a training and a testing set. The 5 city TSP training data consisted of 1 million examples while the testing data consisted of 10,000 examples. For the 5 city TSP, since we had so many training examples, we split data into 800,000 examples to train our data and 200,000 examples to validate our model. The 20 city TSP data consisted of 10,000 examples.

3.2 Architecture Search

We performed an architecture search to see how changing the RNN used for the encoder and decoder, the algorithm to minimize the loss function, the categorical cross entropy, and the decoding algorithm affected the network's ability to obtain an approximation to TSP.

First we compared how using a GRU or an LSTM RNN architecture affected results. We trained on 1000 10 city TSP examples from our generated data pipeline. For both models, we used the Adam optimizer with a learning rate of $1e-4$, with the rest of the parameters the default ones in PyTorch, to minimize the loss function. We used a batch size of 50 and trained for 1000 epochs. Our embedding size was 256. We used the same embedded input in our encoder and decoder. Our hidden size was also 256 and we had 1 layer in both our encoder and decoder.

Afterwards, we compared how using different optimizers, the Adam optimizer or RMSProp, affected the model's ability to learn. We used a LSTM. We trained on 800,000 5 city TSP examples from the dataset from [2] and validated on 200,000 examples. We trained both models using a learning rate of $1e-3$ and a batch size of 64 for 5 epochs. Our encoder and decoder had a embedding size of 256, a hidden size of 512, and 1 layer. We then got outputs from the RMSProp model run on the testing data, and calculated the average valid tour length to compare against the optimal tour length.

We then tested a sequence-to-sequence model using a bidirectional LSTM. We again trained on 800,000 5 city TSP examples from the dataset from [2] and validated on 200,000 examples. We used the Adam optimizer, with a learning rate of $1e-3$ and a batch size of 64 to train for 5 epochs. Our encoder and decoder had an embedding size of 256. We also used only 1 layer in our encoder and decoder. Because our encoder was bidirectional, we had a hidden size half that of the decoder, 256. The decoder had a hidden size of 512. The output hidden and cell states from the bidirectional encoder was $2 \times 64 \times 256$ tensor, which we concatenated the two tensors along the first dimension together into a $1 \times 64 \times 512$ tensor, which could then be fed into the decoder. We then got outputs from the model run on the testing data, and calculated the average valid tour length to compare against the optimal tour length.

3.2.1 Generating Feasible Solutions

While the pointernet model allows for output sequences to be generated that are constrained on the size of the input sequence, they do not enforce that that pointer to input item relation is one to one. For problems such as TSP in which all the cities need to be visited, this can results in certain cities being skipped and others being duplicated in the output sequence. Clearly, these type of outputs are not feasible since they do not constitute a tour. While input sequences of length 5 seemed not to be affected by this problem as the model was able to converge on optimal solutions, this problem occurred quite prevalently for input sequences of length 10 and above.

The first method of remediation for infeasible output solutions was to use a mask with our greedy approach for output sequence element selection. A mask was applied that set the attention weights at indexes of previously chosen output sequence elements to be $-\infty$. By doing so, the softmax applied immediately after to the attention weights would make the probability of choosing that element 0.

The second method of remediation for infeasible output solutions was to use a modified beam search that diregarded sequences of infeasible tours. This mandated that the beam search width k was equal to the length of the input sequence and that marginal probabilities be calculated for every probabilities of every element of the input sequence as the next element, since n total unique items must be chosen for n length output sequence as per the pidgeon hole principle. By computing the marginal probabilities we ensured that every item selected at every time t_i was from a valid probability distribution. Once probability distributions were computed for every input sequence element at every t_i, the highest probability sequence was taken using these probabilities that did not include duplicates by ignoring chosen input elements after they were already selected. This was again be done by setting all remaining probabilities for that input element to $-\infty$. In total, this gave an $O(n^2)$ time complexity for building up the probabilities using a beam search of length n and $k = n$ and $O(n^2)$ time complexity for choosing the highest probability sequence without duplicates.

4 Results

4.1 GRU vs. LSTM

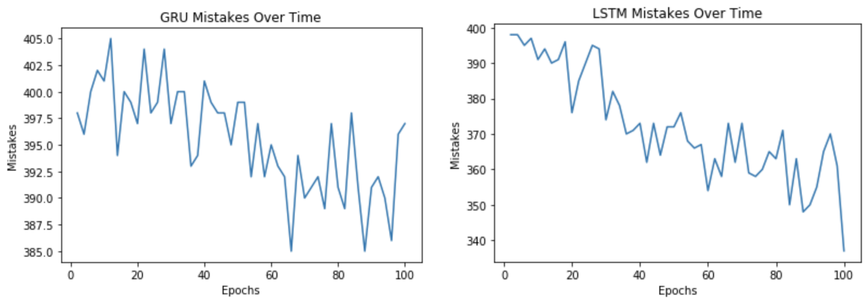


Figure 3: Mistakes of different RNNs over time: (a) GRU (b) LSTM

The plots demonstrate how the LSTM tended to learn and decrease its mistakes over time where as the GRU training was very unstable and only resulted in a minor decrease. This was likely due to the extra context vector the LSTM uses in addition to the hidden vector that both the LSTM and GRU have. Having this extra context vector likely helped the LSTM by giving the LSTM more context to make more accurate decisions from.

4.2 Adam vs. RMSProp

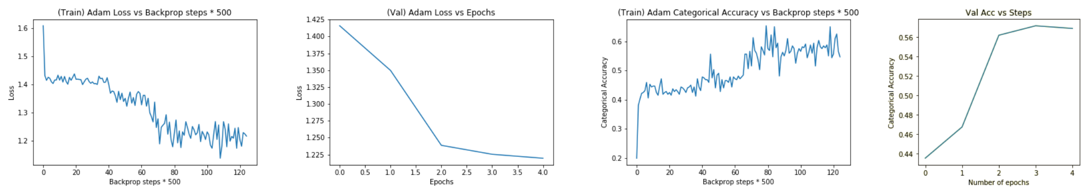


Figure 4: Results from using the Adam optimizer: (a) Training Loss (b) Validation Loss (c) Training Categorical Accuracy (d) Validation Categorical Accuracy

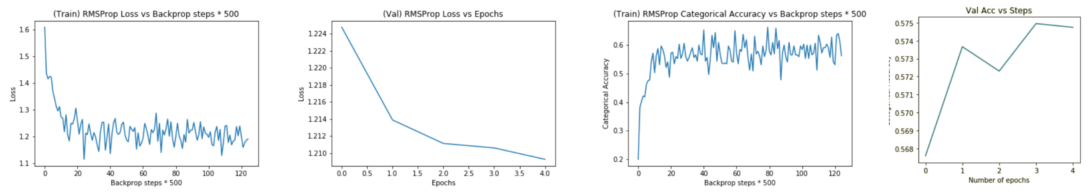


Figure 5: Results from using the RMSProp optimizer: (a) Training Loss (b) Validation Loss (c) Training Categorical Accuracy (d) Validation Categorical Accuracy

Comparing the loss plots of the results of using the Adam optimizer and the loss plots of the RMSProp optimizer shows that using the RMSProp optimizer resulted in faster convergence. Also the RMSProp optimizer didn't seem to get stuck on a local minimia during training, whereas the Adam optimizer did result in getting stuck in a local minima. However, both optimizers resulted in convergence to about the same loss. We got a categorical accuracy 57% for Adam and 59% for RMSProp. We then ran inference on the better model, the RMSProp to obtain the average length of outputted valid tours. The optimal average tour length was 2.1262, while our average tour was 2.333 giving us a 1.097-approximation.

4.3 Bidirectional RNN

Since we used the Adam optimizer when training the bidirectional RNN, we ran into the same problem where it was stuck at a local minima. However, using the bidirectional RNN resulted in faster convergence at the end than Adam and RMSProp, reaching around 57% categorical accuracy at about epoch 3. This is probably because having information about the reverse sequence could help with training and could possibly result in a better model if we had run it for more epochs. However, the resulting average valid tour length was 2.335, a 1.101-approximation, which is slightly worse than the RMSProp result.

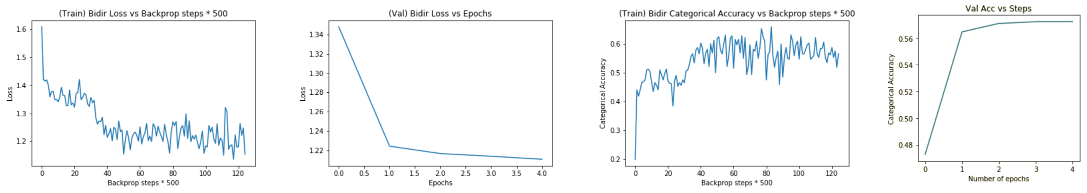
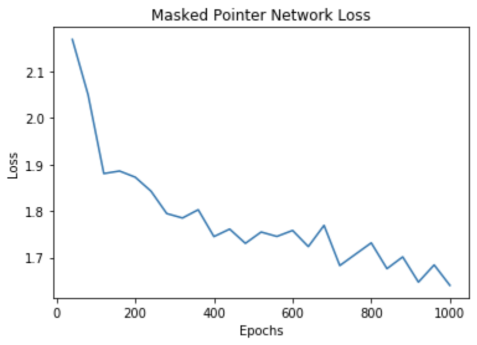


Figure 6: Results from using the bidirectional RNN: (a) Training Loss (b) Validation Loss (c) Training Categorical Accuracy (d) Validation Categorical Accuracy

4.4 Masked Pointer Network vs. Beam Search

Both the masked pointer network and the beam search performed very similarly with both giving on average 1.71-approximation tours as solutions to TSP. Both networks used a learning rate of $1e^{-}3$, hidden and embedding size of 256, batch size of 50, and 10000 examples per epoch.



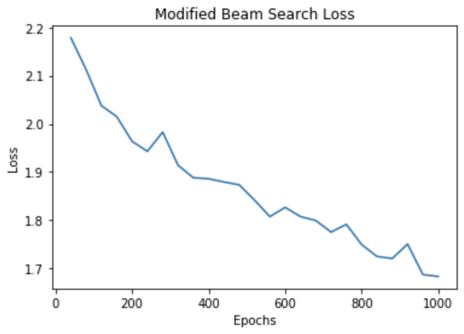


Figure 7: Loss plots of the two feasibility approaches: (a) Attention Masking (b) Modified k sequence length Beam Search

While these results do suggest there is not much advantage to doing a more expensive and technically intricate beam search over a simple greedy approach, we believe that perhaps some further optimizations were necessary in our neural network architecture to fully realize the benefits and drawbacks of using either approach. The loss plots demonstrate that both models converged and that a minima was reached. Likely this was a absolute minima, or the optimizer got stuck in a very difficult to escape local minima, since 1000 epochs were trained on and a starting learning rate of $1e^{-}3$ was used with the RMSProp adaptive learning rate optimizer.

4.5 Untrained Length Sequences

Untrained optimal length sequences of length 20 were tested using the greedy masked pointer network with the exact same hyper-parameters mentioned in the previous section (4.4). Tour sequences were obtained from datasets mentioned above instead of data generation since the data generation algorithm running in exponential time for sequences of length 20 was taking too long to generate even 100 examples. The dataset used gave 10000 examples for testing, and the results obtained gave on average 1.71-approximation tours as solutions to TSP. While these are not close to optimal distances, the fact that increasing the tour length did not increase the approximation ratio indicates that our pointer network architecture is capable of generalizing to sequences of greater length.

5 Conclusion

Overall, the results above indicate two seperate architecture types to be used, divided into categories of sequence length < 10 and sequence length ≥ 10. For sequence length of < 10 we found that it is best to use a bidirectional LSTM, embedding and hidden size of 256, RMSprop, 5 epochs, batch size of 64, and a learning rate of $1e^{-}3$. For sequence length of ≥ 10, we found that it is best to use a LSTM with 1 layer, learning rate of $1e^{-}3$, 10000 examples, batch size of 50, 1000 epochs, no dropout, hidden and embedding sizes of 256, a concatenation based attention mechanism, and a attention weight mask.

The results confirm that pointer networks are great tool for solving combinatorial optimization problems of variable length sequences and achieving reasonable approximate solutions. While adjustments can be made to the above architectures to fit specific needs of other combinatorial optimization problems, and other deep learning paradigms such as reinforcement learning can be used to improve upon these results, supervised learning based pointer networks provide a powerful baseline for solving said problems.

References

[1] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: *Proc. NIPS*. Montreal, CA, 2014. URL: `http://arxiv.org/abs/1409.3215`.

[2] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer Networks". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 2692–2700. URL: `http://papers.nips.cc/paper/5866-pointer-networks.pdf`.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2014. eprint: `arXiv:1409.0473`.

[4] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. eprint: `arXiv:1412.6980`.

[5] T. Tieleman and G. Hinton. *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning. 2012.

[6] Ilya Sutskever. "Training Recurrent Neural Networks". AAINS22066. PhD thesis. Toronto, Ont., Canada, Canada, 2013. ISBN: 978-0-499-22066-0.

[7] Parnia Bahar et al. "Empirical Investigation of Optimization Algorithms in Neural Machine Translation". In: *The Prague Bulletin of Mathematical Linguistics* 108.1 (2017). URL: `https://content.sciendo.com/view/journals/pralin/108/1/article-p13.xml`.

[8] Yanghua Jin et al. "Towards the Automatic Anime Characters Creation with Generative Adversarial Networks". In: *CoRR* abs/1708.05509 (2017). arXiv: `1708.05509`. URL: `http://arxiv.org/abs/1708.05509`.

[9] Jason Lee, Kyunghyun Cho, and Thomas Hofmann. "Fully Character-Level Neural Machine Translation without Explicit Segmentation". In: *CoRR* abs/1610.03017 (2016). arXiv: `1610.03017`. URL: `http://arxiv.org/abs/1610.03017`.

[10] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: `10.1162/neco.1997.9.8.1735`. eprint: `https://doi.org/10.1162/neco.1997.9.8.1735`. URL: `https://doi.org/10.1162/neco.1997.9.8.1735`.

[11] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* abs/1412.3555 (2014). arXiv: `1412.3555`. URL: `http://arxiv.org/abs/1412.3555`.

[12] M. Schuster and K. K. Paliwal. "Bidirectional recurrent neural networks". In: *IEEE Transactions on Signal Processing* 45.11 (Nov. 1997), pp. 2673–2681. ISSN: 1053-587X. DOI: `10.1109/78.650093`.

[13] mlalevic. *Simple Python implementation of dynamic programming algorithm for the Traveling salesman problem*. 2018. URL: `https://gist.github.com/mlalevic/6222750` (visited on 04/18/2019).