AlphaTSP: Learning a TSP Heuristic using the AlphaZero Methodology

Felix Parker
Johns Hopkins University
fparker9@jhu.edu

Darius Irani
Johns Hopkins University
dirani2@jhu.edu

Abstract

Existing heuristic methods for solving combinatorial optimization problems generally perform and scale well, but rely on human intuition and decision making. Deep learning presents an opportunity to both automate the heuristic development process and learn policies that perform better than manually-specified strategies. AlphaTSP is a method inspired by AlphaZero for solving the Traveling Salesman Problem (TSP) that uses self-play reinforcement learning to learn a policy network to guide Monte Carlo tree search (MCTS). MCTS acts as an improvement operator to find a posterior policy given the policy of a neural network, which can be used as training for the network. A specialized graph neural network and graph construction approach are developed as a strong policy network. We demonstrate that AlphaTSP performs better than nearest neighbor and plain MCTS for solving small TSP instances on random Euclidean points, but is more computationally intensive.

1 Introduction

The traveling salesman problem (TSP) is an important and highly studied problem in combinatorial optimization for its applications in operations research and theoretical computer science. Let $\mathcal{G} = (V, E)$ be an undirected weighted graph where V represents a set of n nodes, E represents edges, and an associated cost matrix C_{ij}. C_{ij} satisfies the triangle inequality, meaning $c_{ij} + c_{jk} \geq c_{ik} \;\; \forall i, j, k \in V$, and provides the pairwise distances between edges [1]. Optimal TSP solutions consist of a tour (Hamiltonian circuit) that visits each node in the graph exactly once and minimizes the total edge weight along the tour. The TSP often arises in transportation and logistics applications such as arranging school bus routes to pick up students[1], computer wiring on a microchip, and punching holes within a circuit board. While there have been decades of research on this problem, finding an optimal solution is NP-hard. Therefore, most work has been dedicated to developing heuristics that guide search procedures and efficiently find good tours.

[1]http://www.math.uwaterloo.ca/tsp/apps/index.html

More recently, there has been considerable interest in applying machine learning to combinatorial optimization problems like the TSP [2]. Machine learning methods can be employed either to approximate slow strategies or to learn new strategies for combinatorial optimization. Reinforcement learning, which attempts to learn a strategy that can be used to maximize a reward function, is commonly used. Existing works have attempted to discover heuristics for the TSP using reinforcement learning [3, 4], but so far do not perform particularly well or scale well to large problems.

In this paper we propose to learn a heuristic method for solving the TSP by adapting the AlphaZero methodology, a framework for solving combinatorial games that has seen considerable success in other domains such as Chess, Go, and graph coloring. The specific contributions of our work are as follows:

1. We pose the nearest neighbor construction of a TSP solution as a Markov decision process, allowing us to apply strong methods in reinforcement learning to the problem.
2. We adapt the extremely successful AlphaZero methodology from [5] to solve TSP instances.
3. We develop a specialized representation of partial TSP solutions and a graph neural network to learn a policy for solution searching.
4. We demonstrate the success of our method on small random Euclidean TSP instances compared to other heuristic approaches.

The rest of this paper is structured as follows. In Section 2, we briefly survey related work in using deep learning techniques for combinatorial optimization problems. We also introduce the AlphaZero methodology and its application to other large combinatorial optimization problems like graph coloring. We present the technical details about using this methodology for the TSP, and formulating and training AlphaTSP in Section 3. In Section 4, we empirically evaluate AlphaTSP using generated TSP examples and present our results in Section 5. We conclude with a discussion of AlphaTSP and its contributions in Section 6 and highlight future directions for research in Section 7.

2 Related Work

Traveling Salesman Heuristics.
Heuristic methods for the TSP have been studied for decades resulting in good approximations of exact solvers in polynomial time. Popular heuristics include the greedy, nearest neighbor, insertion, and k-opt methods [6]. The simplest approach, and the one that AlphaTSP is based on, is nearest neighbor. This involves starting at a random node and iteratively selecting the nearest node that is not already in the tour. Insertion is similar, but instead of appending a node to the current partial solution, it determines the optimal location within the current tour to insert it.

In contrast with greedy and insertion methods that build up a tour from a start node, k-opt heuristics start with a random tour and find different ways to reconnect local neighborhoods between sub-tours that minimize the overall edge weight of the complete tour. This heuristic iteratively moves between tours until it arrives at a locally optimal (k-optimal) tour at which point it terminates. Popular choices for k are 2 and 3.

Deep Learning in Combinatorial Optimization.

Application of deep learning techniques to combinatorial optimization problems has been of extensive interest in recent years. Although traditional heuristics perform well and scale to large instances of these optimization problems, they are tedious to specify and require substantial domain expertise. Deep learning can automatically learns policies and these learned policies have the potential to perform better than manually specified heuristics. Two deep learning approaches that can be applied to combinatorial optimization applications are graph embedding networks and neural attention networks.

Dai et al. propose to learn greedy heuristics for difficult combinatorial optimization problems on graphs. They use a graph embedding network called `structure2vec` that captures properties of nodes in a graph within the context of its neighbors [7]. They use this to construct a greedy policy network that incrementally adds nodes to the feasible solution based on the graphical structure. `structure2vec` recursively computes a p-dimensional feature embedding $\mu_v \ \forall v \in V$ given a partial problem solution S by aggregating features x_v according to the graph's topology. They use fitted Q-learning to train the graph embedding network which parameterizes the policy. During each training iteration, the embeddings are synchronously updated and the information is propagated out to adjacent nodes.

Vinyals et al. contribute a neural attention approach to solving combinatorial optimization problems like Delaunay triangulation and the TSP [8]. A variable-sized output dictionary dependent on the length of the input sequence is a characteristic of many combinatorial optimization problems. This variability becomes problematic with traditional models used for sequence learning like Recurrent Neural Networks that require the output dictionary size to be fixed upon training. Vinyals et al. propose an architecture called Pointer Net which solves this problem of variable output dictionary size. Pointer Net uses neural attention, which enables it to focus on a subset of the input features, accomplished through pointer assignment from members of the input sequence to output [9]. The optimal assignment is found through a beam search procedure, which effectively filters out infeasible solutions. Pointer Net is trained in an offline supervised learning setting using example solutions.

AlphaZero

AlphaZero is the latest iteration on an approach to learn to solve complex combinatorial games such as Go with super-human performance, designed by DeepMind [5]. The first iteration, AlphaGo, was the first machine learning algorithm to defeat a world champion in Go and was trained using a combination of supervised learning from human expert moves and self-play reinforcement learning [10]. AlphaZero is an reinforcement learning algorithm trained solely through self-play, starting from random play, in which the algorithm plays an adversarial game against itself. The policy that performs better is rewarded, and over time, stronger policies are learned.

It uses a deep neural network with parameters θ, f_θ, that is trained to select a move and predict the outcome by incorporating a lookahead search inside the training loop. Instead of exhaustively searching the decision tree, AlphaZero's policy network samples from the tree using Monte Carlo tree search (MCTS), which guides the search towards better solutions. This methodology does not depend on human expert knowledge and avoids bias from human patterns. AlphaZero was able to rediscover the extensive Go knowledge accumulated over thousands of years by humans in the span of a few days.

GraphColorNet.
Huang et al. apply the method used in AlphaZero to graph coloring [11]. Graph coloring is another NP-hard combinatorial optimization problem, which asks what the minimal number of colors is such that no two adjacent nodes in the graph are colored the same. Their method, which they call FastColorNet, learns a graph embedding network through an extension of loopy belief propagation using an LSTM, and uses it to greedily assign color selections. Their method performs well compared to other heuristic methods for graph coloring, and scales to very large graphs, but also relies on using huge computer systems. FastColorNet demonstrates the applicability of the AlphaZero methodology to more complex combinatorial optimization problems.

3 Methods Overview

3.1 Traveling Salesman Problem as a Markov Decision Process

In order to apply reinforcement learning to the traveling salesman problem we pose the process of greedily growing a tour as a Markov decision process (MDP). A Markov decision process models a sequential decision process using S, the set of states, A_s, the set of actions available at $s \in S$, $T(s, a, s')$, the probability of reaching s' by taking a at s, and $R(s)$, the reward given in state s. The problem implied by a MDP is to find a policy, $\pi : S \to A$, such that the expected total reward is maximized [12].

To model a traveling salesman problem instance we define the following MDP:

- S is the set of partial tours
- A_s is the set of nodes not included in the partial tour at s
- $T(s, a, s') = 1$ if and only if s' represents the partial tour created by appending a to the partial tour in s
- $R(s)$ is the tour length if s is terminal (represents a full tour), otherwise 0

3.2 Monte Carlo Tree Search with Upper Confidence Bound

We use Monte Carlo tree search (MCTS) as the core of our method to solve the MDP. MCTS is well established as a method to solve Markov decision processes where S is very large. MCTS estimates the optimal policy function by randomly sampling from the decision

	TSP (n=20)	Chess	TSP (n=200)	Go	TSP (n=1000)	TSP (n=85,900)
MDP States	20	40	200	200	1000	85,00
Tree Size	10^{18}	10^{60}	10^{375}	10^{480}	10^{2567}	$10^{386,527}$

Table 1: Approximate number of MDP states and decision tree depth for TSP instances and the games of Chess and Go. While Chess and Go are significantly larger than the small TSP instances we study in this paper, much larger TSP instances are often used in practice and have much larger state spaces than Chess or Go making them more challenging to solve.

tree in order to estimate the value of taking each available action at a given state. As it is infeasible to sample exhaustively from the tree, given a fixed computational budget sampling techniques must balance the exploration of infrequently visited regions with the exploitation of moves that are known to perform well. A detailed discussion of MCTS can be found in [12]. The general procedure for performing MCTS given an input s_0 is summarized in Algorithm 1.[2]

Algorithm 1 MCTS(s_0): The general MCTS approach

```
1: Create root node $v_0$ with state $s_0$
2: while within computational budget do
3:     $v_l \leftarrow$ TREE-POLICY($v_0$)
4:     $\Delta \leftarrow$ DEFAULT-POLICY($s(v_l)$)
5:     BACKPROPAGATE($v_l, \Delta$)
6: return $a$(BESTCHILD($v_0$))
```

While inside of a pre-defined computational budget, MCTS incrementally builds a search tree. At termination, the best-performing root action is returned (a). The MCTS approach can be summarized into four steps: selection, expansion, simulation, and backpropagation. The TREE-POLICY algorithm summarizes the steps occurring during the selection stage. During the selection, MCTS applies a child selection policy (UCT, see Algorithm 6) which iterates down through the tree starting from the root and selects the most urgent non-terminal child (v_l) which contains unvisited children.

Algorithm 2 TREE-POLICY(v): Select most urgent child node

```
1: while $v$ is not a terminal leaf do
2:     if $v$ has unvisited children then
3:         return EXPAND($v$)
4:     else
5:         $v \rightarrow$ BEST-CHILD($v, C_p$)
6: return $v$
```

This node is then added to the current search tree during the expansion phase.

Once added, MCTS simulates the value (Δ) of this node by utilizing a default policy that randomly selects actions taken until a terminal leaf node is reached.

[2]All 6 algorithms presented here are formulated in [12], and the authors retain copyrights.

Algorithm 3 EXPAND(v): Expand search tree by adding node

```
1: Choose $a \in$ untried actions from $A(s(v))$
2: Add new child $v'$ to $v$
3: Set $s(v') = f(s(v), a)$
4: Set $a(v') = a$
5: return $v'$
```

Algorithm 4 DEFAULT-POLICY(s): Simulation

```
1: while $s$ is not a terminal leaf do
2:     Choose $a \in A(s)$ uniformly at random
3:     Set $s \leftarrow f(s, a)$
4:     $\Delta \leftarrow$ reward for state $s$
5: return $\Delta$
```

The outcome of this simulation is backpropagated back up the sequence of nodes from the selected child to the node to update each node's statistics.

Algorithm 5 BACKPROPAGATION(v, Δ): Backpropagation

```
1: while $v$ not null do
2:     Set $N(v) \leftarrow N(v) + 1$
3:     Set $Q(v) \leftarrow Q(v) + \Delta(v, p)$
4:     Set $v \leftarrow v.parent$
```

The problem of optimally balancing between exploration and exploitation is known as the multi-armed bandit problem. In the canonical description of the multi-armed bandit problem a gambler must iteratively choose between K slot machines, each with a different unknown distribution over rewards, in order to maximize their expected total payoff. [13] establish a simple algorithm, Upper Confidence Bound, which optimally converges to the best strategy. [14] extend this approach to MCTS with the Upper Confidence Bound for Trees algorithm, given by Algorithm 1. The key contribution of these methods is the UCT formula, given below, to score the value of taking any action in the search tree.

Within the MCTS+UCT framework, a child node j is selected that maximizes the following,

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

where X_j denotes the expected payoff at j, n denotes the number of times the current node, or parent to j, is visited, n_j gives the number of times child j is visited, and C_p is a positive constant to balance exploration and exploitation. Therefore, in MCTS+UCT we have a method for efficiently searching the decision tree of our TSP MDP that will converge to the optimal policy given a large enough computational budget. Child selection using this UCT policy is summarized in the following algorithm.

Algorithm 6 BEST-CHILD(v, c): Select child node that maximizes UCT

```
1: return $\underset{v' \in v.children}{\operatorname{argmax}} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln n}{n_j}}$
```

3.3 AlphaTSP

Following [10, 15, 5] we propose to learn a policy network to guide the MCTS search process towards better solutions using contextual information. We represent this contextual information as an undirected weighted graph on the nodes in the TSP instance, and use a graph neural network to predict which actions are likely to produce good tours.

3.3.1 Graph Construction

The context available to our policy network at each state s is the positions of the points in the TSP instance, the pairwise distances between points, the sequence of points in the partial tour, and the set of points that are remaining. As this information is naturally graph-structured, we construct a specialized graph from it that can be used by our policy network. The naive approach is simply to generate a graph where each node represents a point and edges join nodes that are consecutive in the current partial tour. However, due to the properties of graph convolutions which we use in our policy network, we instead construct the graph starting with a complete graph on the points and prune away edges that cannot be used in the final solution. Additionally, we define a feature vector on each node that contains its coordinates and an indicator of whether it is in the partial tour, and a weight for each edge that measures the distance between the nodes it connects.

3.3.2 Policy Network

The objective of the policy network is to learn a mapping from a state s in the AlphaTSP MDP, represented by a graph, to a multinomial distribution over the remaining nodes that can be used for MCTS sampling. In order to learn from this graph structured input we employ a graph neural network. Specifically, we design a three-layer graph convolutional network $f_\theta(s) = (\mathbf{p}, v)$ using GCN convolutions [16], described in Figure 1. The network has a base, consisting of two GCN layers with ReLU activations, and two heads. The first head uses a third GCN layer, and then performs a softmax operation on the nodes that are not included in the current tour, producing a distribution over the remaining nodes which can be used for sampling. The second head consists of a global pooling layer which averages features across all nodes, a fully connected layer, and a tanh activation to map the output to $[-1, 1]$ such that it can be used to predict whether the current partial solution is in a winning tour.

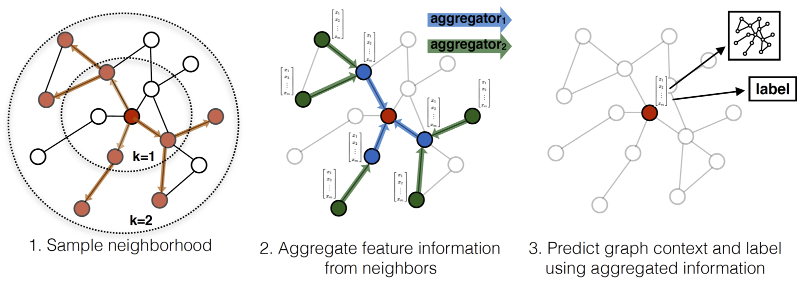
aggregator1
aggregator2
label
k=1
k=2
1. Sample neighborhood
2. Aggregate feature information from neighbors
3. Predict graph context and label using aggregated information

Figure 1: An overview of how graph convolutions work. At a given node, the operator first samples a fixed number of nodes from its neighborhood, applies a transformation function to the associated features, and aggregates over the neighborhood to produce an updated feature vector for the given node. Reprinted from [17][4].

3.3.3 Self Play

In order to train the policy network of AlphaTSP we use a method called self-play, in which the current policy network is trained adversarially against previous iterations of itself. To generate a training example, AlphaTSP first produces a full tour, t_0, by iteratively selecting the best node predicted by the policy network. Then, at each time step t, AlphaTSP runs the previous iteration of the policy network to produce π, a multinomial distribution over the unused nodes, and performs a MCTS search by sampling from this distribution. The MDP is played to completion, yielding a reward $z = +1$ if the resulting tour is shorter than t_0 and $z = -1$ otherwise. Then, examples are generated by storing the state s_t, the search probabilities π_t, and the final outcome z_t at each time step t.

The policy network is then trained on each of these generated examples by minimizing the following loss function.

$$\ell((\mathbf{p}, v), (\pi, z)) = (z - v)^2 + \sum_i (\pi_i - \mathbf{p}_i)^2 \text{ where } f_\theta(s) = (\mathbf{p}, v)$$

4 Experimental Evaluation

4.1 Implementation Details

We implement AlphaTSP in PyTorch using the PyTorch Geometric library. We use multi-threading to generate examples and train in parallel, allowing us to train on many examples. Using a computational budget of 3,000 samples for MCTS, our method can solve a TSP instance on 30 nodes in four seconds on an Intel Core i7 processor with 12 cores. Code is available at: `https://github.com/flixpar/AlphaTSP`.

4.2 Training Examples

We train on 20,000 graphs, saving examples from $n - 2$ time steps where n is the size of the graph. For our primary experiments we use n=30, resulting in 560,000 training examples. All examples consist of points sampled uniformly in the unit square.

[4]For this figure, the original authors retain their copyrights.

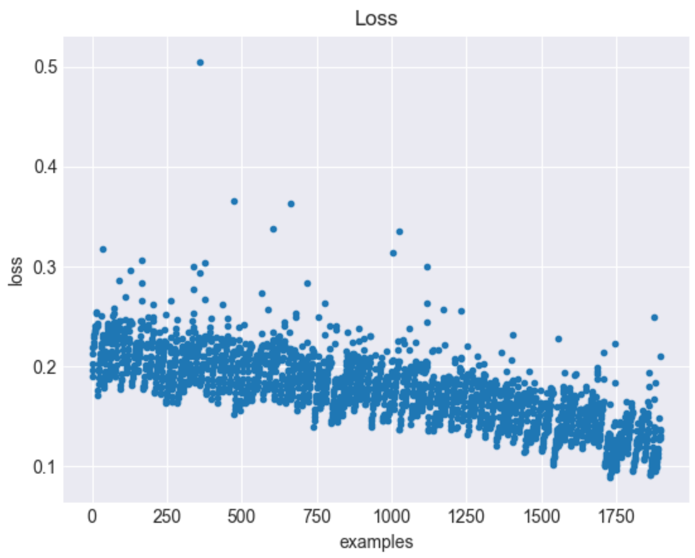

Figure 2: Policy network loss for a sample training run.

4.3 Additional Solvers

To evaluate the performance of AlphaTSP we compare its performance to existing heuristic methods as well as Concorde, an exact solver. The heuristics that we use include nearest neighbor, nearest insertion, and farthest insertion.

5 Results

Solver	Mean Tour Length	Approximation Ratio
AlphaTSP	5.326	1.175
MCTS	6.140	1.354
Nearest Neighbor	5.527	1.219
Exact	4.534	1.000

Table 2: Performance comparison of AlphaTSP to other heuristic solvers and the exact solution. Evaluation used 50 random graphs on 30 nodes. The exact solution was found with the Concorde solver. AlphaTSP performs well compared to other heuristics on these instances, but is still 17.5% worse than optimal.

6 Conclusions

The performance of AlphaTSP on small traveling salesman problem instances demonstrate that it is capable of effectively finding approximate solutions and is competitive with popular existing heuristics. While slower than existing heuristics on small instances, the AlphaTSP algorithm runs in linear time, and should therefore scale better than many heuristics and will scale much better than exact methods for sufficiently large examples.

7 Future Work

We are excited by this problem and optimistic about the potential of AlphaTSP given our initial findings, so we plan to continue working on the project.

Given time constraints, we were not able to do a careful empirical analysis of many of our design choices including graph construction, policy network architecture, graph convolution method, and training strategy. However, these are necessary steps to build a system that will outperform other heuristic strategies on the TSP. Therefore, we plan to begin future work by performing this analysis. In this vein, we are interested in further exploring graph neural networks. Graph neural networks are currently a hot topic of research and there is little consensus on which methods perform best for different problems, so we plan to do a comprehensive test to determine the best graph neural network design for our task. We also plan to continue investigating an alternative approach, node embedding networks, such as `node2vec`.

There has been extensive work on producing variants of Monte Carlo tree search which have different advantages over the base method. We are interested in investigating some of these variants and tailoring a method to our problem. There are also related methods which use Monte Carlo sampling rather than tree search to find good solutions without requiring an explicit graph, allowing them to scale to larger problems.

Finally, one of the attractive properties of AlphaTSP is that it is general and can easily be adapted. In particular, we chose to iteratively select nodes to append to a partial tour, but we plan to also investigate its effectiveness in the insertion heuristic method.

8 Acknowledgements

We would like to thank Professor William Cook for his guidance and feedback during this project.

References

[1] Gilbert Laporte. *The Traveling Salesman Problem: An overview of exact . . .* 1992.

[2] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon. *arXiv.org*, November 2018.

[3] Hanjun Dai, Bo Dai, and Le Song. Discriminative Embeddings of Latent Variable Models for Structured Data. *arXiv:1603.05629 [cs]*, March 2016. arXiv: 1603.05629.

[4] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural Combinatorial Optimization with Reinforcement Learning. *arXiv:1611.09940 [cs, stat]*, November 2016. arXiv: 1611.09940.

[5] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, December 2018.

[6] Sanchit Goyal. A Survey on Travelling Salesman Problem. page 10.

[7] Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning Combinatorial Optimization Algorithms over Graphs. *arXiv.org*, April 2017.

[8] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer Networks. *arXiv:1506.03134 [cs, stat]*, June 2015. arXiv: 1506.03134.

[9] Wouter Kool, Herke van Hoof, and Max Welling. Attention, Learn to Solve Routing Problems! *arXiv.org*, March 2018.

[10] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[11] Jiayi Huang, Mostofa Patwary, and Gregory Diamos. Coloring Big Graphs with AlphaGoZero. *arXiv.org*, February 2019.

[12] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[13] P Auer, N Cesa-Bianchi, P Fischer Machine learning, and 2002. Finite-time analysis of the multiarmed bandit problem. *Springer*.

[14] L Kocsis, C Szepesvári European conference on machine learning, and 2006. Bandit based monte-carlo planning. *Springer*.

[15] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, October 2017.

[16] T N Kipf, M Welling arXiv preprint arXiv 1609.02907, and 2016. Semi-supervised classification with graph convolutional networks. *arxiv.org*.

[17] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. *arXiv.org*, June 2017.