Proceedings of Machine Learning Research 95:726-739, 2018 ACML 2018

Boosting Dynamic Programming with Neural Networks for Solving NP-hard Problems

Feidiao Yang[1,2,3] YANGFEIDIAO@ICT.AC.CN
Tiancheng Jin[4] JINTIA@UMICH.EDU
Tie-Yan Liu[3] TYLIU@MICROSOFT.COM
Xiaoming Sun[1,2] SUNXIAOMING@ICT.AC.CN
Jialin Zhang[1,2,✉] ZHANGJIALIN@ICT.AC.CN
[1] *Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China*
[2] *University of Chinese Academy of Sciences, Beijng, China*
[3] *Microsft Research Asia, Beijing, China*
[4] *University of Michigan, Ann Arbor, MI, USA*

Editors: Jun Zhu and Ichiro Takeuchi

Abstract

Dynamic programming is a powerful method for solving combinatorial optimization problems. However, it does not always work well, particularly for some NP-hard problems having extremely large state spaces. In this paper, we propose an approach to boost the capability of dynamic programming with neural networks. First, we replace the conventional tabular method with neural networks of polynomial sizes to approximately represent dynamic programming functions. And then we design an iterative algorithm to train the neural network with data generated from a solution reconstruction process. Our method combines the approximating ability and flexibility of neural networks and the advantage of dynamic programming in utilizing intrinsic properties of a problem. This approach can significantly reduce the space complexity and it is flexible in balancing space, running time, and accuracy. We apply the method to the Travelling Salesman Problem (TSP). The experimental results show that our approach can solve larger problems that are intractable for conventional dynamic programming and the performances are near optimal, outperforming the well-known approximation algorithms.

Keywords: combinatorial optimization, NP-hard, dynamic programming, neural network

1. Introduction

Dynamic programming is a powerful method for solving combinatorial optimization problems. By utilizing the properties of optimal substructures and overlapping subproblems, dynamic programming can significantly reduce the search space and efficiently find an optimal solution. A representative example is the chain matrix multiplication problem (Cormen et al., 2009). The dynamic programming algorithm takes only a polynomial time complexity of $O(n^3)$ while the naive brute-force method takes at least exponential number of enumerations. Dynamic programming can even provide efficient algorithms for NP-hard problems. For instance, the famous 0-1 knapsack problem can be solved in pseudo-polynomial time complexity of $O(cn)$ with sophisticated dynamic programming, which is much faster than the naive enumeration with $O(2^n)$ time complexity.

However, dynamic programming does not always work well, since the tabular method, which will be discussed later, may take exponential space and time for some NP-hard problems. For example, the Held-Karp algorithm, a dynamic programming algorithm to solve the Travelling Salesman Problem (TSP) proposed independently by Bellman (Bellman, 1962) and by Held and Karp (Held and Karp, 1962), has the time complexity of $O(2^n n^2)$ and space complexity of $O(2^n n)$. Although it is faster than the brute-force method that examines all $O(n!)$ cycles, the algorithm is actually not able to solve relatively large problems in practice more than 20 points because it still takes exponential time and space.

Recently, with success of deep learning, people consider an interesting idea of integrating powerful machine learning methods on solving hard combinatorial optimization problems. Neural network technology may be the most promising one for its powerful approximating ability and flexibility. Vinyals et al. (Vinyals et al., 2015) designed a new recurrent neural network (RNN) architecture called Pointer Network to learn a mapping from sequences to index sequences for learning approximation algorithms to solve combinatorial optimization problems such as convex hull, Delaunay triangulation, and TSP. Milan et al. (Milan et al., 2017) further extended the Pointer Network architecture to another long short-term memory (LSTM) model by considering the original objects. And they applied the method to solve some NP-hard problems including the quadratic assignment problem and TSP. Their work showed the interesting and strong capability of deep neural network to learn an algorithm from input-output data.

However, there are some limitations in these work. First, these approaches do not utilize the intrinsic properties of the problem, and they use neural network as a black box to learn from a lot of labelled data generated from an approximation algorithm. That is, the learning target is approximated solutions rather than optimal solutions and their performance are reasonably behind the approximation algorithms, let alone the optimal solutions. Second, because of the learning difficulty, these methods only learn instances of fixed sizes or limited sizes, for example, TSP instances of 50 points to 100 points. For test cases of larger sizes, the performance drops dramatically. In addition, the sequence to sequence learning frameworks can only handle specific problem types. For example, they can only process planar or Euclidean TSP instances with each element in the sequence being the coordination of a point in the Euclidean space, rather than general TSP problems.

Another important field in machine learning closely related to dynamic programming is reinforcement learning. Reinforcement learning is to learn the best interaction with an environment for getting reward as much as possible. The core problem in reinforcement learning is to solve a Markov decision process (MDP) and work out the optimal policy (action function). The typical way to derive the optimal policy is to learn the value function or Q-function of the MDP, which look like the dynamic programming function in combinatorial optimization. For model-based scenarios (fully known information about the MDP), the basic method is a recursive and iterative method, which is also called dynamic programming approach. For model-free problems (unknown MDP), the method is similar with additionally employing Monte-Carlo simulation. In addition, for solving problems with large scale or even continuous state space or action space, people propose the idea of value function approximation by replacing the tabular value function representation with other more flexible function representation approaches (Sutton and Barto, 2017). Recently, considering the universal approximating ability and success of deep neural networks, people propose

the idea of representing the value function or Q-function with neural networks and design a series of training algorithms (Sutton and Barto, 2017; Riedmiller, 2005; Mnih et al., 2013, 2015). This emerging direction is called deep reinforcement learning.

Inspired by the previous work, we consider if we can combine the advantages of dynamic programming that utilizes the intrinsic properties of a problem and the strong approximating ability and flexibility of neural networks. Comparing with the rigid tabular method, introducing the neural network technique can let the algorithm be more flexible and powerful. On the other hand, comparing with the previous black box usage of neural network, utilizing the problem dependent properties can lead to a better performance.

In this paper, we propose an approach to boost the capability of dynamic programming with neural network technology. This approach approximately represents a dynamic programming function with a neural network and it trains the neural network with an iterative data-driven algorithm. Specifically, the main contributions of this paper are as follows.

- We propose the idea of approximately representing a dynamic programming function with a neural network of polynomial size instead of the conventional memorization table that may require exponential space. This method can significantly reduce the space complexity and it is flexible in balancing the space, running time, and accuracy.
- We design an iterative update algorithm to train the neural network with data generated from a solution reconstruction process. This algorithm is flexible on running time control with respect to the accuracy tolerance and an intermediate result can also provide a fairly good suboptimal solution to the problem.
- We apply the approach to the Held-Karp algorithm to solve TSP and conduct a series of experiments. The experimental results show that our method can solve larger problems that are intractable for conventional dynamic programming. The performances are near optimal, outperforming the well-known approximation algorithms.

We would like to point out that there are differences between dynamic programming for solving MDP and combinatorial optimization problems. First, in MDP, the state transition is non-deterministic but the state transition in combinatorial optimization is usually deterministic. Second, though the state transition in MDP is uncertain, but one state can only jump to another single state with one simulation step. On the contrary, one state may go to multiple sub-states in combinatorial optimization. For example, in chain matrix multiplication problem, the algorithm will face two sub-problems after selecting one split point. Therefore, our approach has to employ a data structure to organize the state visiting.

Another thing we would like to mention is that the intent of this work is not to challenge the best solution of TSP. The value of this work is to provide an idea of leveraging the powerful machine learning methods to solve a new real-world problem in the case that one can design a dynamic programming but it has an extremely large state space. We choose TSP as our experimental benchmark for some reasons. First, it is quite simple to demonstrate our approach so that the readers are easy to follow the key idea but the complicated business details. Second, TSP is a well-studied problem and the Held-Karp dynamic programming algorithm is a natural baseline of our experiments.

The rest of this paper is organized as follows. The second section revisits the basic knowledge of dynamic programming and the Travelling Salesman Problem. The third sec-

tion introduces and discusses our approach in detail. Specifically, one subsection proposes the idea of representing a dynamic programming function with a neural network and the other subsection introduces the training algorithm. In the fourth section, we apply our approach to TSP and report the experimental results. In the last section, we summarize our work and discuss some future exploration.

2. Preliminaries

2.1. Revisiting dynamic programming

To solve a combinatorial optimization problem with dynamic programming, we often consider it as a multi-step decision-making problem. At each step, we face a subproblem and it turns to some smaller subproblems after making the current decision. Subproblems are also called states as they actually form a search state space of a simple search algorithm that inefficiently examines all solutions. One kind of such algorithms is to compute a state value function $f : S \to \mathbb{R}$, in which S is the state set, mapping states to the corresponding optimal values. This function is also called dynamic programming function in the context of dynamic programming algorithm design and analysis.

Dynamic programming can be considered as an improvement to the basic search method under the condition that the search space has the properties of optimal substructures and overlapping subproblems.

Like the divide-and-conquer method, the optimal substructures property means that the solution to a problem could be constructed by combining the solutions to its subproblems. This property gives us a top-down perspective to understand the relationship between a problem and its subproblems and to formulate a recursive equation connecting the solutions to a problem and its subproblems. Usually, suppose selecting a decision a for a state s will turn to a set of sub-states $\delta(s, a)$, the dynamic programming equation for a minimization problem looks like

$$f(s) = \min_a \left\{ v(a) + \sum_{s' \in \delta(s,a)} f(s') \right\}, \tag{1}$$

where a goes over all feasible decisions for state s and $v(a)$ is the value of making this decision.

Unlike the divide-and-conquer method in which subproblems to a problem are disjoint, the overlapping subproblems property says that subproblems to a problem share some subsubproblems. A simple top-down divide-and-conquer method will cause a lot of unnecessary computation by repeatedly solving the common subproblems. A key idea of dynamic programming is to solve each subproblem only once and then save the answer in a table. We call such a method tabular method, which significantly reduces the computation time. Although dynamic programming starts from a top-down view, an algorithm often runs in a bottom-up fashion. That is, the algorithm starts by filling the table with some edge cases and then fills other cells according to the dynamic programming equation.

After the computation of a dynamic programming function, it is easy to find out an optimal solution. We start from the state representing the original problem and make decisions and go through the states with a depth first or breadth first order. At each

step, we just make the decision that optimizes the current subproblem and turn to the next corresponding subproblems accordingly. Specifically, for a current state s, we make a decision a according to the equation

$$a = \arg\min_{a'} \left\{ v(a') + \sum_{s' \in \delta(s,a')} f(s') \right\}. \tag{2}$$

Finally, all such decisions form the solution naturally.

2.2. Travelling Salesman Problem and Held-Karp algorithm

Travelling Salesman Problem (TSP) is a well-known combinatorial optimization problem in computer science and graph theory. The problem is to find a minimum cost cycle in a complete graph with nonnegative edges, visiting every vertex exactly once (Vazirani, 2001). It is an important problem closely related to many practical problems.

The most direct solution is to examine all vertex permutations and find out the one with the shortest length. The running time of this brute-force method is obviously $O(n!)$ and it is inefficient.

Actually, TSP is NP-hard and a reasonable approach is to find suboptimal solutions with approximation algorithms. However, generally TSP cannot be approximated unless P = NP. But for metric and symmetric TSP (with undirected graphs satisfying the triangle inequality), the Christofides algorithm (Christofides, 1976) can solve it with $O(n^3)$ time and produce a suboptimal solution less than 1.5 times the optimum. This is the best algorithm with approximation ratio guarantee known so far. Asymmetric TSP is much harder and the best known approximation ratio is $O(\log n / \log \log n)$ (Asadpour et al., 2010).

Another early effort of solving TSP with dynamic programming is the Held-Karp algorithm (Bellman, 1962; Held and Karp, 1962). Without loss of generality, assume there are n vertices and we always start from vertex 1. It is easy to understand that the construction of a cycle is a multi-step decision-making process. We define a state with a pair (P, c), in which P is a subset of $[n] = \{1, \ldots, n\}$ denoting the vertices to be visited and c is the current starting point. Let $f(P, c)$ denote the shortest length of the path visiting all vertices in P, starting from c and finally returning to vertex 1, as shown in figure 1. The original problem corresponds to the state that $P = [n]$ and $c = 1$.

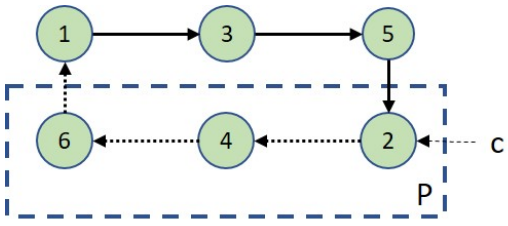

Figure 1: An illustration to the Held-Karp algorithm, where $P = \{2, 4, 6\}$ and $c = 2$. The solid lines denote the visited path and the dashed lines denote a potential path of visiting all vertices in P, starting from c and returning to vertex 1.

Based on the state presentation design, we have the following dynamic programming equation,

$$f(P, c) = \min_{c' \in P; c' \neq c} \{d(c, c') + f(P \backslash \{c\}, c')\}, \tag{3}$$

where $d(c, c')$ is the edge cost from c to c'. Because there are n vertices and 2^n subsets of vertex, the total number of possible states is $O(2^n n)$, which is the space complexity of the dynamic programming algorithm. In addition, as it needs to examine all the left vertices to compute one $f(P, c)$, the time complexity is $O(2^n n^2)$.

It is easy to construct a solution from a given f. Suppose $c_1, \ldots, c_i$ are the first i vertices of the solution and initially $c_1 = 1$. The $(i+1)$-th vertex of the solution is as follows except some edge cases.

$$c_{i+1} = \arg\min_{c \notin \{c_1, \ldots, c_i\}} \{d(c_i, c) + f([n] \backslash \{c_1, \ldots, c_i\}, c)\}. \tag{4}$$

3. Approximating dynamic programming with neural networks

3.1. Approximately representing dynamic programming functions with neural networks

As discussed in last section, the essence of the tabular method is to use a table to represent a dynamic programming function. The advantage of this method is that it can precisely save the information of a function if the state space is finite no matter how complicated the function is. However, it may cost too much space if the state space is extremely large though it is finite.

In addition to the tabular method, there are many ways to represent a function. Among them, neural network is one powerful alternative. According to the universal approximation theorem (Cybenko, 1989; Hornik, 1991), a feed-forward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function (Balázs, 2001).

For representing a dynamic programming function with neural network, we first need to encode states to numerical feature vectors as we usually do in machine learning. Since a state may have several components, our idea is to concatenate different parts to form a vector with each part corresponding to one component. The encoding method for each component may be different according to its type.

- If the component is a numerical scale, we leave it as it is as a component in the vector;
- If the component is an id or order in a set of size n, the part is a one-hot binary vector of size n with all 0s but the corresponding element as 1;
- If the component is a subset of a set of size n, the part is a binary vector of size n having 1 denoting the corresponding element in the subset or 0 otherwise.

For the example in figure 1, the state $(\{2, 4, 6\}, 2)$ could be encoded as a feature vector $(0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0)$, with the first part $(0, 1, 0, 1, 0, 1)$ corresponding to the subset $P = \{2, 4, 6\}$ and the last part $(0, 1, 0, 0, 0, 0)$ corresponding to the starting vertex $c = 2$.

After the quantization of the state (input) space, a dynamic programming function can in principle be extended to a continuous function with interpolation techniques without losing any precision though it is discrete. And then we can further approximately represent the dynamic programming function with a neural network of finite size within any precision.

One may wonder if the size of the neural network will be even larger than the required size of the tabular method. Despite it being possible, we argue that it is still practicable to approximate a dynamic programming function with a much smaller neural network. First, a dynamic programming function does not need to be absolutely precise as it is a tool to construct the optimal solution whose value could be computed directly. Second, not all states are equally useful for solving the problem. We may let the useful states to be more precise than the useless states. That is, the neural network is not necessary to fit all states uniformly. Third, a suboptimal solution is adequate in practice. It is valuable to find a suboptimal solution efficiently with an inaccurate dynamic programming function. With these reasons, the function could be much smoother and could be approximated with a simple way. Comparing with the rigid tabular method, the neural network method is more powerful and flexible. A much smaller neural network could approximate a large model with tolerable accuracy. Certainly, for having a polynomial time algorithm, we employ a neural network with polynomial size, trading off the space, running time, and accuracy.

3.2. An iterative algorithm and solution reconstruction process to train the neural network

A conventional way of computing a dynamic programming function is to fill the table in a specific topological order of states in a bottom-up fashion. However, it is difficult to fill a value to a neural network without affecting other values as it is a parameterized function of all states. In addition, it is also impractical to visit all states if the state space is extremely large. The traditional scheme of training a neural network is supervised learning, which trains the neural network with a lot of sampled $(s, f(s))$ pairs and hopes the generalization works. However, this method is impractical because the computation of even a single exact $f(s)$ may take exponential time.

Similar to the algorithm of deep Q-learning (Mnih et al., 2013, 2015), we design an iterative algorithm to update the neural network with training data generated from a solution reconstruction process, as shown in algorithm 1 for a minimization problem.

Rather than fitting the state values directly, we alternatively turn to the idea of making the model look like a dynamic programming function as much as possible. Specifically, suppose $f(s;\theta)$ is the neural network, where s is the input state vector and θ is the model parameter. Ideally, if f is exactly the dynamic programming function, equation (1) should hold for all states. However, there is usually a gap between the left term and the right term. A natural idea is to make the gaps as close as possible and the learning object is to minimize the loss function $J(\theta)$ defined as

$$J(\theta) = \sum_{s\in\mathcal{S}} \left(f(s;\theta) - \min_a \left\{ v(a) + \sum_{s'\in\delta(s,a)} f(s';\theta) \right\} \right)^2, \tag{5}$$

Algorithm 1 Training the neural network with solution reconstruction

Initialize neural network $f(s;\theta)$ with model parameter θ, maybe with pre-training on edge cases
Initialize the data pool D
Initialize the exploration parameter $\varepsilon_t = 1.0$ and the learning rate η_t appropriately
for each iteration $t = 1, 2, \ldots$ **do**
  Initialize the state list S, the state scheduling data structure Q, and the state visit-marking data structure V
  Add the state s_0 representing the original problem to Q and V
  while Q is not empty **do**
    s = Q.pop()
    S.Add(s)
    With probability ε_t:
      select a random feasible decision a
    with probability $1 - \varepsilon_t$:
      select decision a according to equation (2)
    for each sub-state $s' \in \delta(s, a)$ **do**
      if cannot find s' in V **then**
        Add s' to Q and V
      end if
    end for
  end while
  Generate a mini-batch training data B=S+Sample(D)
  Preform a gradient descent step on data B to close the gap according to equations (6) and (7)
  Add S to D with a weight being the value of the solution
  Decay ε_t and η_t if necessary
end for

where $\mathcal{S}$ in principle is the set of all states. A common solution to the optimization problem is the gradient descent algorithm. The gradient with respect to θ is

$$\nabla J(\theta) = 2 \sum_{s \in \mathcal{S}} \left(f(s;\theta) - \min_a \left\{ v(a) + \sum_{s' \in \delta(s,a)} f(s';\theta) \right\} \right) \left(\nabla f(s;\theta) - \nabla \min_a \left\{ v(a) + \sum_{s' \in \delta(s,a)} f(s';\theta) \right\} \right). \tag{6}$$

In each iteration the model parameter is updated by a gradient descent step with learning rate η_t as

$$\theta_{t+1} = \theta_t - \eta_t \nabla J(\theta_t). \tag{7}$$

However, it is impractical to calculate the gradient in a single iteration with all states. Based on the idea of mini-batch technique in stochastic gradient descent, we use a small sample of states instead to estimate the gradient. In our algorithm, we generate training

states by an iterative solution reconstruction process. In each iteration, we construct a solution based on current function f as if it is the correct dynamic programming function, and we use the collected states as training data. This method has advantages over random sampling. First, as f gets better and better, the generated states are more likely to be on the optimal path or near the optimal path. That is, these states have likely more potential. Second, we can give a weight to such states according to the generated solution. Such weights could be further utilized for training data sampling, which is different from the uniform data sampling in (Mnih et al., 2013, 2015).

For avoiding the model falls into local optima too early, particularly in the early stage that f is inaccurate, we introduce the exploration strategy in solution construction. We maintain an exploration parameter ε_t. At each step in iteration t, we make the "optimal" decision according to equation (2) with probability $1 - \varepsilon_t$, or choose a random feasible decision with probability ε_t. Parameter ε_t decay over time, from 100% to a small number such as 5%.

For collecting more data for training and breaking the correlation of consecutive iterations, we maintain a data pool. In each iteration, we sample moderate data according to their weights from the pool along with the generated data as a mini-batch to update the model.

State scheduling is another feature in our algorithm, which is a key difference from the standard Q-learning. In the process of solution construction, one state may lead to multiple sub-states, so we need to visit the states in a specific order with the help of some appropriate data structures. For example, we may visit the states in the depth first order with a stack or in the breadth first order with a queue. In addition, if sub-states will be overlapped, we will need an extra data structure such as a hash set or binary set to mark if a state has been visited otherwise it will cause a lot of unnecessary computation. Certainly, if the states to a solution just form a chain, it is unnecessary to have such scheduling data structures.

4. Experiments in TSP

4.1. Experimental setting

We choose TSP as the experimental setting for some reasons. On one hand, among the elegant NP-hard problems such as the satisfiability problem (SAT) and the set cover problem in theoretical computer science, TSP is the most practical one as many real-world problems can be modeled with TSP and it is easy to understand with less background knowledge. On the other hand, among the NP-hard problems in the real world such as the vehicle routing problem (VRP) and the bin packing problem, TSP is the simplest one and it is easy to follow, avoiding complicated conditions.

We carry out experiments on a subset of TSPLIB (Reinelt, 1991), a data set of sample instances from real-world problems, including symmetric and asymmetric problems. TSPLIB provides the best-known solutions to its problems that are from more than ten years' efforts of the human.

We implement four algorithms. The first is our approach, named as NNDP, short for neural network dynamic programming. The second is the Held-Karp algorithm. The third is the Christofides algorithm. And the last is the nearest neighbour greedy algorithm, which

is also a well-known approximation algorithm since it is quite simple, taking only $O(n)$ or $O(n^2)$ time, though it does not have the theoretical guarantee as good as Christofides does.

Following are some experimental details to our algorithm.

For a given instance of TSP with n vertices, we construct the dynamic programming function as a fully connected feed-forward neural network. The input layer has $2n$ nodes, separated into two parts. The first part is a binary vector of size n, corresponding to P, denoting the vertex subset to be visited. The other part is a one-hot binary vector of size n, corresponding to c, denoting the starting vertex. The neural network has two hidden layers, each having $4n$ hidden nodes with sigmoid activation function. We choose such a setting for balancing the training time and the accuracy. And finally the output layer has only one node and it is a linear combination of the outputs of the previous layer. Therefore, there are $O(n^2)$ parameters in the neural network.

We develop our code based on Theano and run the experiments on a server with Intel Xeon CPU (E5-2695 v2, 2.40GHz) and 128 GB memory. For each instance, we run the algorithm with 10000 iterations. We maintain the exploration parameter ε_t with an exponential decay that $\varepsilon_{t+1} = \max\{0.995\varepsilon_t, 0.05\}$. We keep a small constant learning rate $\eta = 0.001$. The data pool is a limited queue that holds at most 1000 paths and the data too old will be thrown away. A mini-batch contains 10 paths, with 1 path generated from the current iteration and 9 ones sampled from the data pool.

4.2. Experimental results

The experimental results are shown in table 1. The number appended to the name of an instance means the problem size. For example, Gr17 is an instance having 17 vertices. The first 7 cases are symmetric TSP instances and the last 3 cases are asymmetric TSP instances. In the table, we show the absolute value to a solution as well as the approximation ratio over the best-known value for clear comparison.

Seeing from the experimental results, although the Held-Karp algorithm can guarantee the optimal solution, it cannot extend to relatively large problems due to its space complexity issue. Our algorithm overcomes the space problem and it can solve larger problems that are intractable for conventional dynamic programming.

Our approach also performs well. The relative errors of our approach are almost within 10%, outperforming the Christofides algorithm and the greedy algorithm. Our approach even has good performance in asymmetric TSP. We do not directly compare our approach with previous neural network based method in (Vinyals et al., 2015) and (Milan et al., 2017) because they can only solve Euclidean TSP. But the instances in TSPLIB are not necessary in such case. However, the experimental results in these papers show that their methods do not catch up with the approximation algorithms, Christofides algorithm specifically, so it is safe to conclude that our approach performs better than the previous work.

The running time depends on the problem size and the training data size. As we design the neural network with $O(n^2)$ parameters, the total time complexity to one iteration is $O(n^4)$. In our server, the running time to one iteration varies from 0.01 seconds to 0.1 seconds, for problems from about 20 vertices to about 100 vertices. Our approach converges reasonably and it is flexible with respect to accuracy tolerance. Figure 2 illustrates an

example of the convergence of the case Bayg29. It shows that the algorithm can reach a nearly optimal solution with about 500 iterations.

Table 1: Experimental results in TSPLIB

Data	**Best**	**NNDP**		**Held-Karp**		**Christofides**		**Greedy**	
Gr17	2085	**2085**	**1**	**2085**	**1**	2287	1.1607	2178	1.0446
Bayg29	1610	**1610**	**1**	NA	NA	1737	1.0789	1935	1.2019
Dantzig42	699	**709**	**1.0143**	NA	NA	966	1.382	863	1.2346
HK48	11461	**11539**	**1.0068**	NA	NA	13182	1.1502	12137	1.059
Att48	10628	**10868**	**1.0226**	NA	NA	15321	1.4416	12012	1.1302
Eil76	538	**585**	**1.0874**	NA	NA	651	1.1128	598	1.1115
Rat99	1211	**1409**	**1.1635**	NA	NA	1665	1.3749	1443	1.1916
Br17	39	**39**	**1**	**39**	**1**	NA	NA	56	1.435
Ftv33	1286	**1324**	**1.0295**	NA	NA	NA	NA	1589	1.2002
Ft53	6905	**7343**	**1.0634**	NA	NA	NA	NA	8584	1.169

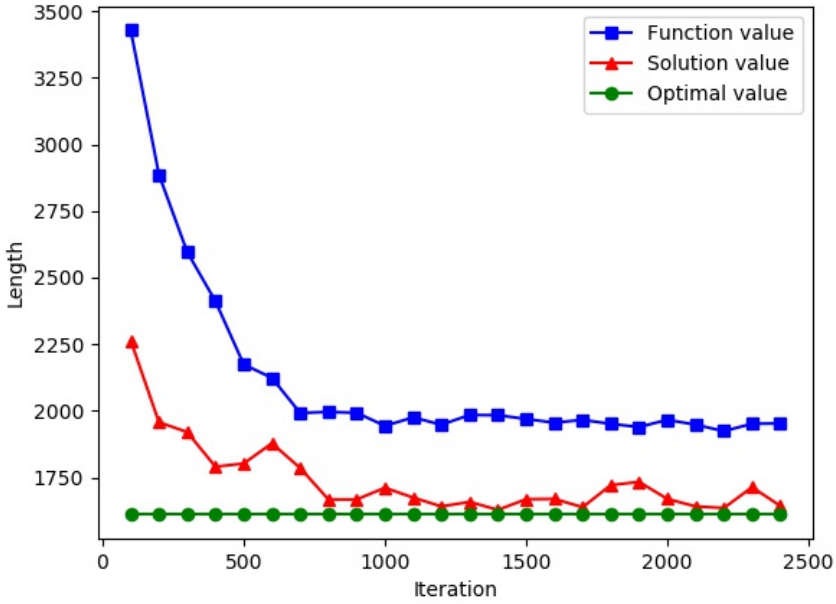

Figure 2: Convergence of the case Bayg29. The function values are the values the model outputs; the solution values are the exact values of the solutions constructed from the model.

5. Conclusions and future work

Based on the neural network technology and the similar idea from reinforcement learning method, in this paper we propose an approach to boost the capability of dynamic programming with neural networks for solving NP-hard combinatorial optimization problems. First, we replace the tabular method with a neural network of polynomial size to approximately represent a dynamic programming function. And then we design an iterative algorithm to train the neural network with data generated from a solution reconstruction process. Our method combines the approximating ability and flexibility of neural networks and the advantage of dynamic programming in utilizing intrinsic properties of a problem. Our approach can significantly reduce the required space complexity for some NP-hard problems and it is flexible in balancing space, running time, and accuracy.

There are differences in our approach from the standard deep reinforcement learning. First, we aim at solving the NP-hard combinatorial optimization problems while reinforcement learning is to solve MDP. Moreover, we introduce some scheduling data structure to handle the cases that one state has multiple subsequent substates. Last but not least, we assign each data point a weight for sampling from the data pool rather than uniform sampling.

We apply our approach to the Bellman-Held-Karp algorithm, a dynamic programming algorithm for solving TSP. The experimental results show that our method can handle larger problems that are intractable for the conventional dynamic programming algorithms. As an approximation algorithm, our approach also outperforms other well-known approximation algorithms.

There may be some future work along this direction. First, we may consider applying this approach to more theoretical problems such as the weighted set cover problem and practical problems such as the express delivering problem. Second, the neural network dynamic programming function could be considered as a heuristic function in the search process. So it is possible to generalize the idea to replace the handcrafted heuristic function with a neural network and to automatically learn a better heuristic function in search algorithm design though the search spaces do not have the good properties. In addition, a possible extension of our approach is to solve combinatorial optimization problems under non-deterministic environments.

Acknowledgments

We would like to thank Guang Yang from Institute of Computing Technology, Chinese Academy of Sciences, for his inspirational discussion. We would also like to thank Weidong Ma from Microsoft Research Asia for his reading through an early draft of this paper and his helpful advice on polishing the language. We deeply appreciate the valuable comments from the reviewers, especially the pointing to the valuable reference "Neuro-Dynamic Programming" Bertsekas et al. (1997) which we omitted before and should be an important reference for the future work. This work is supported in part by the National Natural Science Foundation of China Grant 61433014, 61761136014, 61872334, 61502449, 61602440, the 973 Program of China Grants No.2016YFB1000201.

References

Arash Asadpour, Michel X. Goemans, Aleksander Madry, Shayan Oveis Gharan, and Amin Saberi. An o(log n/ log log n)-approximation algorithm for the asymmetric traveling salesman problem. *In Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'10*, pages 379–389, 2010.

Csanád Csáji Balázs. Approximation with artificial neural networks. Master's thesis, Eötvös Loránd University, 2001.

Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1):61–63, January 1962. ISSN 0004-5411. doi: 10.1145/321105.321111. URL http://doi.acm.org/10.1145/321105.321111.

Dimitri P Bertsekas, John N Tsitsiklis, and A Volgenant. Neuro-dynamic programming. *Third World Planning Review*, 1997.

Nocos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem, 1976.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

G. Cybenko. Approximations by superpositions of sigmoidal functions. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.

Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962. doi: 10.1137/0110015. URL http://dx.doi.org/10.1137/0110015.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(91)90009-T. URL http://www.sciencedirect.com/science/article/pii/089360809190009T.

Anton Milan, S. Rezatofighi, Ravi Garg, Anthony Dick, and Ian Reid. Data-driven approximations to np-hard problems. *AAAI Conference on Artificial Intelligence*, 2017.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A Riedmiller. Playing atari with deep reinforcement learning. *arXiv: Learning*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin A Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015.

Gerhard Reinelt. Tsplib – a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991. doi: 10.1287/ijoc.3.4.376. URL http://dx.doi.org/10.1287/ijoc.3.4.376.

Martin Riedmiller. Neural fitted q iteration-first experiences with a data efficient neural reinforcement learning method. *ECML*, 3720, 2005.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction, Second Edition*. The MIT Press, 2017.

Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015. URL http://papers.nips.cc/paper/5866-pointer-networks.pdf.