Deep Neural Network Approximated Dynamic Programming for Combinatorial Optimization

Shenghe Xu,[1] **Shivendra S. Panwar,**[1] **Murali Kodialam,**[2] **T.V. Lakshman**[2]
[1]Department of Electrical and Computer Engineering, NYU Tandon School of Engineering, Brooklyn, NY
[2]Nokia Bell Labs, Crawford Hill, NJ
{shenghexu, panwar}@nyu.edu, {murali.kodialam, tv.lakshman}@nokia-bell-labs.com

Abstract

In this paper, we propose a general framework for combining deep neural networks (DNNs) with dynamic programming to solve combinatorial optimization problems. For problems that can be broken into smaller subproblems and solved by dynamic programming, we train a set of neural networks to replace value or policy functions at each decision step. Two variants of the neural network approximated dynamic programming (NDP) methods are proposed; in the value-based NDP method, the networks learn to estimate the value of each choice at the corresponding step, while in the policy-based NDP method the DNNs only estimate the best decision at each step. The training procedure of the NDP starts from the smallest problem size and a new DNN for the next size is trained to cooperate with previous DNNs. After all the DNNs are trained, the networks are fine-tuned together to further improve overall performance. We test NDP on the linear sum assignment problem, the traveling salesman problem and the talent scheduling problem. Experimental results show that NDP can achieve considerable computation time reduction on hard problems with reasonable performance loss. In general, NDP can be applied to reducible combinatorial optimization problems for the purpose of computation time reduction.

Introduction

Dynamic programming (DP) is a widely used method for solving various optimization problems (Bellman 1966). For a problem that can be reduced to sub-problems with similar structures, each corresponding to a stage of decision making, DP finds the optimal solution for each sub-problem and achieves global optimal solution. Since the problem is broken into sub-problems, DP can efficiently reduce search space as compared with naive exhaustive search over all the possible combinations.

However, for many of the combinatorial optimization problems, the size of the search space grows exponentially or by factorial order of the problem size. Even if problems are broken into sub-problems to reduce search space, the complexity of algorithms using DP can still be high for NP-hard problems. For example, for the traveling salesman problem (TSP), simple exhaustive search has time complexity of $\mathcal{O}(n!)$. The Bellman-Held-Karp algorithm (Bellman 1962; Held and Karp 1962) based on DP can achieve time complexity of $\mathcal{O}(2^n n^2)$ and space complexity of $\mathcal{O}(2^n n)$. The exponential growing complexity can still be relatively high for large problem sizes, making it unsuitable for time or memory critical applications.

Methods based on neural networks (NNs) have been proposed as solutions to combinatorial problems since decades ago (Looi 1992; Smith 1999). Recent advancements in deep neural networks (DNNs) have lead to more efficient schemes using novel network architectures and new training procedures of NNs (Bello et al. 2016; Khalil et al. 2017; Yang et al. 2018). However, many of the previously proposed methods focus on specific classes of problems such as graph based problems (Khalil et al. 2017) and routing problems (Kool, van Hoof, and Welling 2018). Alternatively, they rely on specific network architectures and specific reinforcement learning training procedures (Bello et al. 2016). Though in (Yang et al. 2018) a NN based dynamic programming method was proposed, it requires training on each testing instance of the problem, which is impractical for time critical tasks.

In this paper, we propose a deep neural network approximated dynamic programming approach to solve general combinatorial optimization problems. The main contributions of this paper are as follows:

- We propose a general framework of replacing policy or value function calculation process with NNs called neural network approximated dynamic programming (NDP). The framework is simple and robust and can be combined with different NN architectures.
- An unsupervised training procedure is proposed for NDP. It consists of a pre-training step and a fine-tuning step. Robust performance improvement is achieved in the training procedure.
- Experimental results on the linear sum assignment problem (LSAP) and the TSP show that compared with previous methods, NDP can be an alternative to achieve a balance between computation time and solution quality. When applied to the talent scheduling problem, NDP is

able to achieve considerable reduction of computation time, with a reasonable gap from the optimal solution.

Related Work

Recent developments of deep neural networks has enabled machine learning based methods to achieve state-of-the-art results in various tasks(LeCun, Bengio, and Hinton 2015; He et al. 2016). By adopting various techniques and NN architectures, several methods have also been developed for combinatorial optimization problems and achieved close to optimal results (Bello et al. 2016; Khalil et al. 2017; Yang et al. 2018; Kool, van Hoof, and Welling 2018).

In (Bello et al. 2016), a pointer network (Vinyals, Fortunato, and Jaitly 2015) based method was proposed to solve the traveling salesman problem (TSP) and the knapsack problem. To overcome the difficulty of generating high-quality labeled data for NP-hard problems, the pointer network was trained with reinforcement learning, with an actor-critic based procedure. Combined with an active search process at testing stage, the pointer network based method was able to achieve near optimal solutions with less computation time compared with previous methods.

An algorithm called S2V-DQN using graph embedding networks was proposed in (Khalil et al. 2017). The algorithm focuses on solving combinatorial optimization problems with graph structures, especially TSP. S2V-DQN uses a strcture2vec network(Dai, Dai, and Song 2016) to represent information in a policy. Then a DQN (Mnih et al. 2015) is trained to provide a greedy policy on the representation. This method is able to achieve close to optimal solutions, with the ability to generalize to problems with sizes over 1000.

To solve routing related problems, an attention model (AM) based method was proposed in (Kool, van Hoof, and Welling 2018). The attention model contains an encoder that produces embedding of the context of the problem, and a decoder that produces the solution sequentially. For TSP, the attention based method is able to achieve solutions that are closer to optimum compared with S2V-QDN and the method in (Bello et al. 2016).

In (Yang et al. 2018), the authors proposed a method called neural network dynamic programming (NNDP) to boost the performance of DP with NNs. For each instance of TSP, their method trains a new set of parameters with a solution reconstruction process that samples solutions. The neural network is trained to estimate quality of each solution. The main difference between our approach and NNDP is that our approach does not require training on test samples, and instead of training one NN, we train a series of NNs for each problem size. Note that NNDP still needs to run the NN multiple times for a multi-step DP problem. With the training time on each instance, NNDP may be unsuitable for time-critical tasks.

A classifier based approach was proposed in (Lee et al. 2018) to solve LSAP. The LSAP requires a solution to assign n jobs to n people that maximize reward or minimizes cost. In their approach for each person a single classifier is trained to get a suitable job assignment for that person. Inevitably there may be jobs that are assigned to more the one person. A greedy heuristic based approach was proposed to resolve assignment collisions.

To overcome the curse of dimensionality, neuro-dynamic programming was proposed for space and computation time reduction for the original DP (Bertsekas and Tsitsiklis 1996). However, the previous works on neuro-dynamic programming mainly focus on using simple approximation functions such as linear functions or polynomial regression to approximate value functions (Powell 2007). Most of the works on neuro-dynamic programming also focus on stochastic control problems (Bertsekas and Tsitsiklis 1996; Van Roy et al. 1997; Lam, Lee, and Tang 2007).

Different from above mentioned previous works, our method is proposed for general combinatorial optimization problems that can be solved by traditional DP. Unlike the method in (Bello et al. 2016), NDP does not require sophisticated network structures like the pointer network or the attention model in (Kool, van Hoof, and Welling 2018). NDP does not require a graph structure in the problem like S2V-DQN(Khalil et al. 2017), which may not exist in many problems that can be solved by DP. Unlike the method proposed in (Yang et al. 2018), which requires training on each of the testing samples for at least 1 second, in this paper we assume that only the distribution of the problem instances is known. The training set and testing set are generated according to the known distribution with different random seeds. NDP is trained on a training set and does not require training (Yang et al. 2018) or searching (Bello et al. 2016) for optimization on new instances of the problem. The method proposed in (Lee et al. 2018) requires training of n problem-size specific classifiers for a size n problem. While in NDP each DNN is responsible for solving a sub-problem with different size, DNNs for smaller problem sizes are used for training and testing on problems with larger sizes. In addition, certain amount of the optimal solutions are needed for the training of the classifier based approach, this may be time consuming or even infeasible for hard problems. Unlike previously proposed neuro-dynamic programming methods (Bertsekas and Tsitsiklis 1996), in this paper we focus on using DNNs for value or policy function approximation. We apply NDP on general combinatorial optimization problems instead of stochastic control problems.

There may be concern that with multiple NNs, NDP may be more time consuming than other methods. However, except for the classifier based approach mentioned in (Lee et al. 2018), the other methods all work in a step by step way. For problems with n steps the NNs are used n times. So the main extra cost introduced by NDP is the space for storage of NNs in memory, which is usually abundant in modern computers.

We emphasize that in this paper, the main contribution is not to gain performance improvement on previously well studied problems. We focus on developing a simple and general framework to speedup DP for combinatorial optimization problems. In addition, the flexibility of this framework allows it to be combined with more powerful network architectures or training procedures for better performance, or simpler network structures for less complexity. For general problems, especially problems without existing high quality

solver or heuristic based methods such as talent scheduling, NDP is a simple and efficient approach for obtaining a close to optimal solution with computation time reduction.

Dynamic Programming and Approximation Methods

Dynamic programming was developed for a class optimization problems that can be converted into a process of making a decision in several steps. The optimal solution of the overall problem should be obtainable by making optimal choices at each step; this is called the principle of optimality (Bellman and others 1954). By dividing the problem into smaller subproblems, DP can effectively reduce the search space of combinatorial optimization problems. For a given state s, which corresponds to a subproblem, and an action from the feasible set of actions $a \in \mathcal{A}(s)$, the Bellman equation for a reward maximization problem can be written as:

$$V(s) = \max_{x \in \mathcal{A}(s)} (R(s,a) + V(s')), \tag{1}$$

where s' is the next state followed after choosing action a, and $R(s,a)$ is the obtained reward by choosing action x, $V(s)$ is the value function for state s. The Bellman equation can be solved by backward induction, by computing the value functions for smaller problems and obtaining the final value function step by step. However, for problems with a large state space, backward induction may be time consuming or even infeasible. Several techniques have been proposed to address the curse of dimensionality (Bertsekas and Tsitsiklis 1995; 1996; Powell 2007; Buşoniu, De Schutter, and Babuška 2010; Mes and Rivera 2017; Yang et al. 2018). Many approaches have been proposed to approximate the value function, including using basis functions (Buşoniu, De Schutter, and Babuška 2010), linear models, polynomial regression (Powell 2007) and DNNs as proposed in (Yang et al. 2018; van Heeswijk and La Poutré 2019). As far as we know, apart from (Yang et al. 2018; van Heeswijk and La Poutré 2019), this is the only other work that uses DNNs for function approximation in DP. As previously stated, different from (Yang et al. 2018), our approach requires no training on testing samples, and uses different DNNs for different decision steps instead of one DNN for all the steps. (van Heeswijk and La Poutré 2019) focuses more on value function approximation. They only studied the nomadic trucker problem, under a Markov decision process setting, with smaller problem sizes, which is more like a reinforcement learning approach. They used a single NN for a fixed problem size, which is different from this paper.

Dynamic programming can be used to solve optimization problems in two ways. For the value function based variation, the value function is solved for the states, then policy is chosen based on maximization of the value function. The other approach is to derive optimal policy for each state, and perform optimal actions at each state.

Training Procedure

We propose a two phase training procedure for NDP. In the first phase, the DNNs are pre-trained for only a few iterations with data generated from given distributions. In the second phase, all the DNNs are fine-tuned together. DNNs for smaller problem sizes use data generated by the policies of DNNs from earlier steps. The pre-training procedure helps the networks to converge faster to suitable policies. The fine-tuning procedure helps to further improve performance of the trained policy.

Training for Value Approximation

For value function approximation, training starts from the DNN for the smallest problem size, or the states in the last decision step. In the case of value based NDP, for the smallest subproblem P_1 with size N_1 and A_1 possible actions, a DNN denoted by G_1 is trained to minimize the mean squared error (MSE) of estimation results of the value function:

$$MSE = \frac{1}{T} \sum_{s_1 \in \mathcal{S}_1} \sum_{a \in \mathcal{A}(s_1)} (G_1(s_1, \theta_1, a) - R(s_1, a))^2, \tag{2}$$

where T is the total number of the combination of states and actions, $\mathcal{S}_1$ is the set of all the possible states for the subproblem P_1, θ_1 is the coefficient for G_1, $R(s_1, a)$ is the instantaneous reward obtained by choosing action a at state s_1. Since P_1 is the final decision step, there is no state transition so value can be directly calculated from the action in a given state. Then for a following DNN corresponding to the subproblem with size N_n and A_n feasible actions, DNN G_n is trained to minimize

$$MSE = \frac{1}{T} \sum_{s_n \in \mathcal{S}_n} \sum_{a \in \mathcal{A}(s_n)} (G_n(s_n, \theta_n, a) - R(s_n, a) - \tag{3}$$

$$V^G_{n-1}(s_{n-1}))^2, \tag{4}$$

where $\mathcal{S}_n$ is the set of possible states, $V^G_{n-1}(s_{n-1})$ is the value function obtained by following policy generated by previous trained DNNs

$$V^G_n(s_n) = \sum_{i=1}^{n} R(s_i, a^G_i), \tag{5}$$

where

$$a^G_i = \arg\max_a G_i(s_i, \theta_i, a). \tag{6}$$

s_i is the state for subproblem i caused by following the policy generated by the DNNs. Since the number of possible states can be infinite, it is infeasible to calculate MSE on all the states. We follow the common procedure of performing gradient descent on mini-batches of data to train the NNs. The pre-training procedure is shown in algorithm 1, where M is the number of steps in the problem, E is the number of epochs to train a DNN.

In the second phase, the DNNs are fine-tuned together. The intuition behind this phase is that the distribution of the states lead by a given policy may be different from the states used in the pre-training phase, and it is hard to estimate the distribution since it also changes with the updates of policy. To help the DNNs better approximate the value functions, in the fine-tuning phase the states for subproblem k are obtained by following policies generated by G_{k+1},...G_M, for solving the overall problem of M steps. The fine-tuning procedure is shown in algorithm 2.

Algorithm 1 Pre-training Process of NDP

for $i = 1; i < M; i++$ **do**
  for $j = 1; j < E; j++$ **do**
    Generate B batches of states for problem i from a given distribution.
    Calculate $V_{i-1}^{G}(s_{n-1})$.
    for $k = 1; k < B; k++$ **do**
      Update θ_i with data batch k in s_i.
    end for
  end for
end for

Algorithm 2 Fine Tuning Process of NDP

for $i = 1; i < E; i++$ **do**
  Generate B batches of states for problem M.
  for $j = 1; j < M; j++$ **do**
    Obtain B batches of data for s_j for problem j, following policy given by previous DNNs.
    Calculate $V_{j-1}^{G}(s_{n-1})$.
    for $k = 1; k < B; k++$ **do**
      Update θ_j with data batch k in s_j.
    end for
  end for
end for

Training for Policy Approximation

Similar to traditional DP, DNNs in NDP can also be trained to directly provide policies at each decision step. Instead of training the DNNs to estimate value functions for each actions at given state, the DNNs are trained to directly estimate the best action to be taken at a given state. In this case, the DNNs are used for classification of the best action. Since this can be seen as a classification task, cross entropy loss is used for training of the DNNs.

The Two Phase Training Procedure

The two phase training procedure is adopted mainly for three reasons. To stabilize training performance, pre-training is used to help the DNNs to get close to a good local optimum. On the other hand, if there is need to solve problems of various sizes, pre-training the DNNs for different problems sizes and saving the models for fine-tuning can help save overall training time. Finally, the fine-tuning phase is adopted to train the DNNs with more accurate distribution of data. Ideally, if the DNNs can achieve optimal solutions at each step, the pre-training procedure should be sufficient for obtaining the optimal policy. However, the DNNs' limited capacity and sensitivity to data distribution requires further training with data closer to the real distribution.

The two phase training procedure is partially motivated by (Hinton and Salakhutdinov 2006), however in their case the term pre-training and fine-tuning are used for layers of a NN, in this paper we are using the terms for a series of DNNs that are trained and used sequentially. In experiments, we find that the pre-training process helps the DNNs converge to efficient solutions in less training steps. On the other hand, the training process of NDP is also similar to multi-agent reinforcement learning (Buşoniu, Babuška, and De Schutter 2010). But with clearly defined problem structure, at each training step, the agents are able to obtain feedback for all possible actions, instead of performing a single action in the reinforcement learning context. Similar to fixing the policy by using a target network in the training process of DQN (Mnih et al. 2015), the coefficients for each agent are also updated consecutively in several batches, with the coefficients of other DNNs fixed, which helps stabilize training results with gradient descent. This is also confirmed by our experiments, updating each DNN for several batches, while keeping other DNNs the same, achieves more robust testing performance compared with updating the DNNs simultaneously.

Solving Optimization Problems with Neural Network Approximated Dynamic Programming

In this section, we describe how NDP is applied to the LSAP, TSP and the talent scheduling problem.

The Linear Sum Assignment Problem

We first start with the LSAP, this problem is can be solved optimally with the Hungarian algorithm with complexity of $\mathcal{O}(n^3)$ (Kuhn 1955). Meanwhile, it is also reducible to subproblems and thus can be solved by DP. In LSAP, n jobs have to be assigned to n people in an optimal way. The reward of assigning job i to person j is c_{ij}. The objective is to maximize the sum of the rewards, with the constraint that each job should be assigned to one and only one person.

We formulate the DP solution to this problem as a multi-stage decision process. For a subproblem with size n, a decision of assigning the first job to a person has to be made. After assigning the job to person k, by removing rewards $c_{1,1\ldots n}$ and $c_{1\ldots n,k}$, the problem is transitioned into a subproblem with size $n-1$.

The value function for subproblem with size 2 can be directly written in matrix multiplication form, so training of DNNs start from subproblems with size 3.

The Traveling Salesman Problem

The traveling salesman problem (TSP) is one of the well investigated NP-hard problems. Various heuristic methods have also been proposed to solve TSP (Applegate et al. 2006). In TSP an agent has to find a route to visit all given cities exactly once and return to the starting city. The objective is to minimize the total distance traveled in the route.

We formulate the subproblems as finding the shortest route from a given starting city to a given ending city which visits all the cities exactly once. d_{ij} denotes the distance from city i to city j. The state of each subproblem can be represented by a matrix, and for convenience we assume the starting city is the city with index 1, while the ending city has index n. State transitions involve removing $d_{1,1\ldots n}$, $d_{1,\ldots n,1}$ and assigning the chosen city with index one. By assigning the same city to index one and index n, the returning requirement can be enforced at the first decision step.

The Talent Scheduling Problem

The talent scheduling problem is also an NP-hard problem that can be solved by DP (Garcia de la Banda, Stuckey, and Chu 2011; Qin et al. 2016). In the problem a suitable schedule of shooting a number of movie scenes has to be found. Each scene may involve a number of actors and last for a number of days. Each actor incurs a certain amount of cost per day. The actor has to be paid for the duration from the first involved scene to the last involved scene, including the time that scenes without the actor is scheduled.

The subproblems are formulated as follows; given a number of scenes, actors that are currently on hold, and costs for each actor, find the next best scene to be scheduled. State transitions involves removing the scheduled scene and updating the list of actors that are on hold. At the first decision stage there is no actor on hold.

Experimental Setup

LSAP. For LSAP we test the performance of NDP with problem size up to 20. The performance of NDP is compared with the Hungarian method implementation in SciPy (Jones et al. 2016). A simple greedy heuristic, in which at each step the assignment with highest reward among all jobs and people is chosen is used as a baseline method for comparison. For uniformly generated rewards with 20 jobs, compared with the Hungarian method, the greedy method achieves a performance gap of 5.02% and NDP-policy achieves a performance gap of 3.14%. So instead of using uniformly generated rewards, for which the greedy baseline can easily achieve close to the optimal solution, we focus on a scenario where there is no obvious simple heuristic. The rewards are generated from a Beta distribution, with $\alpha = 0.07$ and $\beta = 0.17$ for evaluation purposes.

TSP. For TSP we follow the same practice as mentioned in (Kool, van Hoof, and Welling 2018). The cities are generated uniformly from the unit square. Euclidean distance is used as cost. The same test dataset and baseline methods as in (Kool, van Hoof, and Welling 2018) are used for evaluation.

The talent scheduling problem. As mentioned in (Cheng, Diamond, and Lin 1993), the talent scheduling problem with each actor required for two scenes and a universal daily wage of one is already NP-hard. However we still assume the actors can have random uniformly distributed daily wages. Since the duration of most of the test cases are all ones in (Garcia de la Banda, Stuckey, and Chu 2011), we focus on the scenario where all scenes have the same duration. Instead of integer wages used in previous works (Qin et al. 2016; Garcia de la Banda, Stuckey, and Chu 2011), we use floating point wage value generated from a uniform distribution to train the DNNs, which is beneficial for training. For testing we still use integer wages, so that the previous methods can be applied directly to get the optimal solution. When generating scenes, for each actor we randomly choose the number of scenes from 2 to the maximum number the actor is in, and randomly allocate the scenes. Currently there is no well-known heuristic baseline for the talent scheduling method. We propose a heuristic similar to the principle of selecting equivalent scenes first mentioned in (Garcia de la Banda, Stuckey, and Chu 2011); we define waiting cost as the cost of actors waiting on the site, at each step, the scenes with the least amount of waiting cost are selected. We denote this method as least waiting cost (LWC). For problem size, we select 20 actors and 20, 25 and 30 scenes. The cost for actors are random integers from 1 to 100 for evaluation, and floating numbers from 1 to 100 for training.

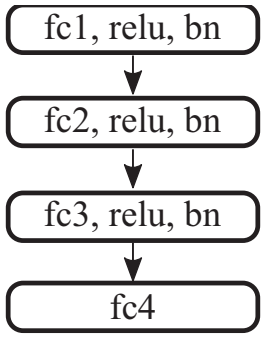

Figure 1: Network Architecture for TSP

Comparison of computation time. Since our method can be run in parallel for batches of instances, we evaluate the computation time of the methods in the same way as in (Kool, van Hoof, and Welling 2018). For all the experiments we include the computation time of 10,000 problems for TSP and LSAP, and 1,000 problems for talent scheduling. Experiments are conducted on a server with one P100 GPU and two Xeon Silver 4114 CPUs. For baseline methods with only CPU implementation, we test on the same server using all the cores of the CPU in parallel. We use a batch size of 10,000 for both the attention model (AM) in (Kool, van Hoof, and Welling 2018) and NDP.

Network architectures. For all the problems, simple fully connected DNNs are used in NDP. Figure 1 shows the network architecture used for TSP. Each fully connected layer is followed by a relu activation function and a batch normalization (BN) function (Ioffe and Szegedy 2015). Using more advanced network architectures such as Resnet (He et al. 2016) can further improve performance, however in this paper we do not focus on finding the best parameters for the NNs. Relu is used as the activation function for all the DNNs.

For LSAP with subproblem of n jobs, DNNs with one hidden layer of size $8n$ are used. Batch normalization is used for the policy-NDP. Using a larger hidden layer size $16n$ brings less than 0.5 percent performance gap reduction but introduces longer computation time.

For TSP with 20 cities, DNNs with three hidden layers of size $2n^2$, $4n^2$ and $16n$ is used for each subproblem. For TSP with 50 cities, smaller DNNs with hidden layers $8n$, $4n$ and $2n$ are used for each subproblem to reduce training and testing time. Batch normalization is used for both value based NDP and policy based NDP.

For talent scheduling problem with 20 actors and n scenes, DNNs with two hidden layers with size 1200 and

$n * 4$ are used. The cost for each actor can be represented as a vector $\boldsymbol{c}$, with c_i for cost of actor i. Instead of representing the scenes in a binary matrix $\boldsymbol{O}$ with $o_{ij} = 1$ indicating actor i is in scene j, and zero otherwise, we set $o_{ij} = c_i$ if actor i is in scene j and concatenate the matrix with $\boldsymbol{c}$ and a binary vector indicating the actors waiting on site. Batch normalization is only used for policy based NDP.

For DNNs used in NDP, in general it is beneficial to select a relatively large first hidden layer. However to achieve a balance between computation time and the performance gap, for large problems such as TSP with 50 cities we are using smaller DNNs. While performance varies with the choice of parameters, in our experiment the performance of the DNNs always improves with training time until they converge.

Training settings. All the models of NDP are implemented with Pytorch (Paszke et al. 2017) and the DNNs are trained with the Adam optimizer (Kingma and Ba 2014). A learning rate of 0.001 and batch size of 100 is used for both pre-training and fine-tuning.

For pre-training, all the DNNs for LSAP and TSP are trained with 3000,000 samples of data generated on the fly, each sample is used only once. For talent scheduling, since making sure the scenes in each problem are all different is time consuming, 2000,0000 samples are generated and each sample is used for three times for pre-training.

For LSAP with 20 jobs, results from 1000 epochs of fine-tuning are selected for comparison. For fine-tuning each epoch consists of training of all the DNNs each with 100,000 samples of data. It takes approximately one hour to pre-train the DNNs from problem size 3 to 20. While for the fine-tuning phase, with NDP-policy, each epoch takes about 100s. Fine-tuning all the DNNs for 1000 epochs take about 28 hours on one GPU. For LSAP with 50 jobs, results after 65 epochs of fine-tuning are used for comparison. In this case fine-tuning for 65 epochs takes around 30 hours on a single GPU.

For TSP with 20 cities, we show results for NDP-policy and NDP-value after 1500 epochs of fine-tuning, which takes about 70 hours on one GPU. For TSP with 50 cities, the smaller DNNs converge after 100 epochs of fine-tuning, which takes about 75 hours for NDP-policy and 55 hours for NDP-value on a single GPU.

For the talent scheduling problem, since the DNNs are larger, fine-tuning takes longer time. For problems with 20 actors and 20 scenes, each epoch of fine-tuning takes around 400s. For problems with 20 actors and 25 scenes, each epoch takes about 800s. For problems with 20 actors and 30 scenes, each epoch takes about 1200s. We fine-tune the models with different number of epochs. With fixed dataset for training, the DNNs converge within less number of epochs but possibly to policies further from the optimal ones. For the 20 scene case, we show results of NDP-policy fine-tuned for 600 epochs, NDP-value fine-tuned for 400 epochs. For the 25 scene case we fine-tune the DNNs for 150 epochs. For the 30 scene case the DNNs for both NDP-value and NDP-policy are fine-tuned for 65 epochs.

Figure 2 shows the change of validation performance gap during the fine-tuning phase of NDP. We select the results of first 600 epochs of fine-tuning for all three problems for plotting. All curves are for results of NDP-policy. For LSAP and TSP the problem size is 20, for talent scheduling the problem setting is 20 actors with 20 scenes. It can be seen from the figure that in the fine-tuning phase, the performance gaps are consistently reduced during the training process. For TSP the DNNs converge after about 1500 epochs of fine-tuning. While for talent scheduling training for another 300 epochs only reduces the performance gap by 0.2%. The relatively high initial performance gap of the talent scheduling problem may be caused by the difference in the distribution of data. For pre-training phase, the actors on site are generated randomly, while in the fine-tuning phase, distribution of on site actors may be different due to previous policy of the DNNs.

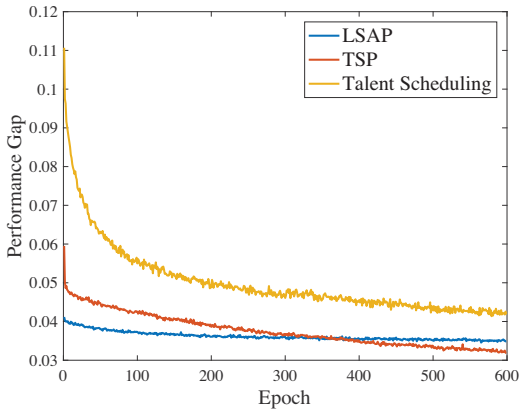

Figure 2: Validation Accuracy in Fine-tuning Phase

Performance Evaluation

We evaluate the performance of NDP in terms of the solution's performance gaps from the optimal or best ones, and computation time.

For LSAP, the Hungarian method is used as the benchmark method. The comparison results are shown in table 1. The simple greedy heuristic, in which at each step the assignment with highest reward among all jobs and people is chosen as a baseline method for comparison. The NDP based methods are able to achieve considerable computation time reduction, due to the parallel operation on GPU. Both NDP methods achieve about three percent performance gap from the optimal solution in the 20 job case. For problems with 50 jobs, due to diversity gain, making a sub-optimal decision in one step has less impact on the overall performance. So the non-optimal methods achieve closer to optimal solutions. NDP methods still achieve better performance than the greedy method.

For TSP we compare the performance on problem size 20 and 50. Results are shown in table 2. We include results of Gurobi since according to (Kool, van Hoof, and Welling 2018), it achieves the best solution in the least amount of time. We also include results of AM from (Kool, van Hoof, and Welling 2018), as far as we know it is the NN based method with closest to optimum solutions for TSP. Random

Table 1: Performance Comparison on LSAP

20 Jobs			
Method	Reward	Gap	Time
Hungarian	19.77	0.00%	1.47s
Greedy	17.49	11.55%	0.28s
NDP-Policy	19.07	3.47%	0.01s
NDP-Value	19.16	3.07%	0.01s
50 Jobs			
Hungarian	49.99	0.00%	7.19s
Greedy	47.25	5.49%	1.27s
NDP-Policy	49.12	1.77%	0.10s
NDP-Value	49.27	1.45%	0.10s

Table 2: Performance Comparison on TSP

20 Cities			
Method	Cost	Gap	Time
Gurobi	3.84	0.00%	5.42s
Random Insertion	4.00	4.36%	0.37s
Farthest Insertion	3.93	2.36%	0.58s
AM	3.84	0.08%	0.31s
NDP-Policy	3.93	2.54%	0.06s
NDP-Value	3.98	3.84%	0.06s
50 Cities			
Gurobi	5.70	0.00%	74.84s
Random Insertion	6.13	7.65%	1.23s
Farthest Insertion	6.01	5.53%	1.82s
AM	5.80	1.76%	1.38s
NDP-Policy	6.38	12.02%	0.14s
NDP-Value	7.10	24.69%	0.14s

insertion (RI) and farthest insertion (FI) are also included for comparison. Due to the simple architectures of DNNs in NDP, both policy based NDP and value based NDP are able to obtain solutions faster than AM. However the NDP based methods perform worse in terms of performance gap from the best solution. In terms of solution quality, policy based NDP performs better than random insertion, but worse than farthest insertion in the 20 city case. For the problem size of 50, NDP methods suffer a larger performance gap but run with less computation time.

For the talent scheduling problem, according to (Qin et al. 2016) the enhanced branch and bound (EBB) method is so far the fastest method that achieves optimal solution, we include results generated by the C++ implementation provided by the authors. Even though EBB is run to solve the problems in parallel on all cores of the CPU, the computation time is still high for problems with 30 scenes. Note that when EBB is used to solve each problem sequentially, on average it takes 72ms, 102ms and 172ms to solve one problem with 20, 25 and 30 scenes. For the talent scheduling problem, the policy based NDP achieves much better performance than the value based NDP. This may be because due to the limited capacity of the DNNs, the DNNs were not able to sufficiently approximate the value functions. Overall,

Table 3: Performance Comparison on Talent Scheduling

20 Scenes			
Method	Cost	Gap	Time
EBB	13660.32	0.00%	8.00s
LWC	16420.23	20.2%	0.08s
NDP-Policy	14231.52	4.18%	0.08s
NDP-Value	15818.16	15.80%	0.07s
25 Scenes			
EBB	17278.15	0.00%	49.28s
LWC	21142.65	22.37%	0.08s
NDP-Policy	18599.89	7.65%	0.13s
NDP-Value	20675.90	19.67%	0.08s
30 Scenes			
Method	Cost	Gap	Time
EBB	20639.77	0.00%	1030.80s
LWC	25677.81	24.41%	0.10s
NDP-Policy	23659.64	14.63%	0.15s
NDP-Value	25132.72	21.77%	0.10s

the performance gap of NDP methods are higher for talent scheduling problems. This may be because the dataset used for training consists of fixed samples, while for other problems each update of the DNNs are performed with new randomly generated data. The experiment results are shown in table 3.

Conclusions and Future Work

In this paper, we proposed a deep neural network based dynamic programming approximation method to solve combinatorial problems. Experiment results show that the proposed method is able to achieve considerable computation time reduction, with less than 5% loss on TSP with 20 cities and LSAP with 20 jobs. When applied to the talent scheduling problem, tested with 1000 problems with 25 scenes and 20 actors, NDP-policy is able to achieve solutions within 10% gap from the optimal cost, within 200ms, while current best solver takes almost 50s to obtain the solutions. NDP can be a promising method to reduce computation time of traditional DP for time critical applications. With NDP's unsupervised training procedures, it can also be an alternative to solve relative large problems that DP can not solve in feasible time.

Meanwhile, there are still several open research directions for NDP. The training of a series of DNNs may be time consuming, especially in the fine-tuning stage. Simply increasing the learning rate does not lead to faster performance and may cause instability during training. Hopefully with the development of hardware and faster DNN training techniques, this problem can be mitigated. Alternatively a more efficient fine-tuning procedure for NDP can be found. For policy based NDP, we found that using more advanced network architectures such as Resnet (He et al. 2016) can further reduce the performance gap from the optimal value. However, training such networks is more time consuming. One possible solution is to share some of the parameters such as the convolution kernels in Resnet. We leave the work of finding

more suitable network architectures and parameter sharing methods for future research.

References

Applegate, D. L.; Bixby, R. E.; Chvatal, V.; and Cook, W. J. 2006. *The traveling salesman problem: a computational study*. Princeton University Press.

Bellman, R., et al. 1954. The theory of dynamic programming. *Bulletin of the American Mathematical Society* 60(6):503–515.

Bellman, R. 1962. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)* 9(1):61–63.

Bellman, R. 1966. Dynamic programming. *Science* 153(3731):34–37.

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

Bertsekas, D. P., and Tsitsiklis, J. N. 1995. Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE Conference on Decision and Control*, volume 1, 560–564. IEEE.

Bertsekas, D. P., and Tsitsiklis, J. N. 1996. *Neuro-dynamic programming*. Athena Scientific Belmont, MA.

Buşoniu, L.; Babuška, R.; and De Schutter, B. 2010. Multi-agent reinforcement learning: An overview. In *Innovations in multi-agent systems and applications-1*. Springer. 183–221.

Buşoniu, L.; De Schutter, B.; and Babuška, R. 2010. Approximate dynamic programming and reinforcement learning. In *Interactive collaborative information systems*. Springer. 3–44.

Cheng, T.; Diamond, J.; and Lin, B. 1993. Optimal scheduling in film production to minimize talent hold cost. *Journal of Optimization Theory and Applications* 79(3):479–492.

Dai, H.; Dai, B.; and Song, L. 2016. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, 2702–2711.

Garcia de la Banda, M.; Stuckey, P. J.; and Chu, G. 2011. Solving talent scheduling with dynamic programming. *INFORMS Journal on Computing* 23(1):120–137.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Held, M., and Karp, R. M. 1962. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics* 10(1):196–210.

Hinton, G. E., and Salakhutdinov, R. R. 2006. Reducing the dimensionality of data with neural networks. *Science* 313(5786):504–507.

Ioffe, S., and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Jones, E.; Oliphant, T.; Peterson, P.; et al. 2016. Scipy: Open source scientific tools for python, 2001.

Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, 6348–6358.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kool, W.; van Hoof, H.; and Welling, M. 2018. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*.

Kuhn, H. W. 1955. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2(1-2):83–97.

Lam, S.-W.; Lee, L.-H.; and Tang, L.-C. 2007. An approximate dynamic programming approach for the empty container allocation problem. *Transportation Research Part C: Emerging Technologies* 15(4):265–277.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *Nature* 521(7553):436.

Lee, M.; Xiong, Y.; Yu, G.; and Li, G. Y. 2018. Deep neural networks for linear sum assignment problems. *IEEE Wireless Communications Letters* 7(6):962–965.

Looi, C.-K. 1992. Neural network methods in combinatorial optimization. *Computers & Operations Research* 19(3-4):191–208.

Mes, M. R., and Rivera, A. P. 2017. Approximate dynamic programming by practical examples. In *Markov Decision Processes in Practice*. Springer. 63–101.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529.

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A. 2017. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*.

Powell, W. B. 2007. *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. John Wiley & Sons.

Qin, H.; Zhang, Z.; Lim, A.; and Liang, X. 2016. An enhanced branch-and-bound algorithm for the talent scheduling problem. *European Journal of Operational Research* 250(2):412–426.

Smith, K. A. 1999. Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS Journal on Computing* 11(1):15–34.

van Heeswijk, W., and La Poutré, H. 2019. Approximate dynamic programming with neural networks in linear discrete action spaces. *arXiv preprint arXiv:1902.09855*.

Van Roy, B.; Bertsekas, D. P.; Lee, Y.; and Tsitsiklis, J. N. 1997. A neuro-dynamic programming approach to retailer inventory management. In *Proceedings of the 36th IEEE Conference on Decision and Control*, volume 4, 4052–4057. IEEE.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer networks. In *Advances in Neural Information Processing Systems*, 2692–2700.

Yang, F.; Jin, T.; Liu, T.-Y.; Sun, X.; and Zhang, J. 2018. Boosting dynamic programming with neural networks for solving np-hard problems. In *Asian Conference on Machine Learning*, 726–739.