*Proceedings of the 2020 Winter Simulation Conference*
*K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.*

# DEEP REINFORCEMENT LEARNING IN LINEAR DISCRETE ACTION SPACES

Wouter van Heeswijk
Han La Poutré

Department of Intelligent and Autonomous Systems
Centrum Wiskunde & Informatica
Science Park 123
Amsterdam, 1098 XG, THE NETHERLANDS

## ABSTRACT

Problems in operations research are typically combinatorial and high-dimensional. To a degree, linear programs may efficiently solve such large decision problems. For stochastic multi-period problems, decomposition into a sequence of one-stage decisions with approximated downstream effects is often necessary, e.g., by deploying reinforcement learning to obtain value function approximations (VFAs). When embedding such VFAs into one-stage linear programs, VFA design is restricted by linearity. This paper presents an integrated simulation approach for such complex optimization problems, developing a deep reinforcement learning algorithm that combines linear programming and neural network VFAs. Our proposed method embeds neural network VFAs into one-stage linear decision problems, combining the nonlinear expressive power of neural networks with the efficiency of solving linear programs. As a proof of concept, we perform numerical experiments on a transportation problem. The neural network VFAs consistently outperform polynomial VFAs as well as other benchmarks, with limited design and tuning effort.

## 1 INTRODUCTION

Problems in operations research (OR) are generally concerned with allocating resources in a way that maximizes some reward function. Applications of OR are found in domains such as transportation, energy, and manufacturing. Resources are typically allocated over time (i.e., multi-period planning), with future problem states being subject to uncertainty. Markov Decision Process (MPD) models are useful to describe such problems. However, as problem states and actions are typically represented by fairly large vectors – in the magnitude of hundreds or thousands of variables – these MDPs cannot be solved exactly and must be solved using simulation instead. Reinforcement Learning (RL) finds approximate solutions to such dynamic planning problems. One solution class within RL focuses on value function approximations (VFAs) to estimate downstream values corresponding to each state-action pair, as such creating a decision policy. This typically requires designing a set of features (i.e., explanatory variables), organizing them as a polynomial function (i.e., a linear approximation) that represents the value of a given state-action pair, and performing linear regression after sampling value observations to learn weights associated to the features.

Although polynomial VFAs often yield satisfactory results, designing appropriate features is challenging. Particularly higher-level interactions are difficult to grasp for human designers. For instance, many allocation problems have both spatial and temporal components, yet the relationships between these components are often difficult to formalize. As the VFA is a linear combination of features, all these relations must be explicitly defined in separate features. In recent years, the application of neural networks to design VFAs has become increasingly popular. Neural networks are able to learn complex nonlinear functions. Because neural networks are not restricted by linearity, they may identify nonlinear structures between lower-level features, without explicitly defining these structures as separate features.

The problem of defining appropriate polynomial VFAs is often amplified when action spaces are too large for enumeration. In OR, linear programming is a popular approach to deal with large action spaces. Many

OR problems may be formulated as mathematical programs – preferably linear programs (LPs) – that solve to optimality within reasonable time. Although linear programming allows to deal with vastly larger action spaces than enumeration, polynomial VFAs are difficult to embed, as the variables representing the features in the objective function must also be linear. Although nonlinear features might be designed – only the combination of features needs to be linear – all features must be expressed as linear systems, often requiring complicated constructions of artificial variables. Complex polynomial VFAs are therefore difficult to embed in LPs. Neural network VFAs may circumvent this problem, by automatically identifying relations between simple linear features.

We design a hybrid approach to address RL problems, integrating neural networks in a decision-making LPs to obtain VFAs. The planning problem tested is a dynamic transportation problem. To preserve focus on the methodological aspect of the paper, we will not discuss design choices in detail. The problem serves as a test case that is sufficiently rich and challenging to adequately test the solution method. This paper contributes to the state of the art in the following ways. First, we design a hybrid approach for integrating neural network VFAs and LPs to tackle RL problems with large action spaces in the OR domain. Second, we provide insights into the performance of various neural network structures based on simulation experiments in a transportation context, focusing on solution quality and computational effort.

## 2 LITERATURE REVIEW

This paper integrates linear programming and neural networks in the context of value function approximation, aimed at solving multi-period optimization problems with large discrete action spaces. We reflect on related works in these areas.

Typical RL algorithms require at least several thousands of simulated observations to learn a policy, so it is essential that decision problems are solvable in limited time (at most a few seconds). There are several approaches to deal with large action spaces, see, e.g., Powell and Ma (2011), Busoniu et al. (2017). We present a coarse categorization here. The first one – which is utilized in this paper – is mathematical programming. Expressing the decision problem as a mathematical program retains optimality. Although it vastly improves scalability compared to enumeration, it may require action space reduction (e.g., by deploying meta-heuristics) when scaling to very large action spaces. Second, factorization methods (Pazis and Parr 2011; Dulac-Arnold et al. 2012) divide the action space into linear subproblems, exponentially reducing the computational effort. However, the size of the state space is a limiting influence in this approach, requiring a value to be stored for each unique state. Third, a subset of (promising) actions may be sampled (Dulac-Arnold et al. 2015), rather than evaluating the full action space. Sampling methods are scalable to very large action spaces, but might result in selecting suboptimal actions. Furthermore, effective action sampling requires a fixed action space, whereas in OR settings we often work with state-dependent action spaces.

We proceed to discuss the application of neural networks in RL, which are considered as potentially powerful tools in learning algorithms. In OR contexts, neural network VFAs have mainly been applied on classical engineering problems that typically have low-dimensional action spaces (Powell and Ma 2011). Bertsekas and Tsitsiklis (1995) discuss the use of feature vectors as input to neural networks as an alternative to polynomial VFAs. Alternatively, as a preliminary step, neural networks may automatically derive the feature vector from the state description (Keller et al. 2006). Bertsekas (2012) addresses neural networks in RL from a helicopter perspective. The author broadly defines neural networks as essentially nonlinear VFAs, using either the full state description or a smaller feature vector as input. Again, neural networks may also be used as a pre-processing step to extract feature vectors from the state. In the RL community, the integration of neural networks is known as deep reinforcement learning (LeCun et al. 2015; Schmidhuber 2015). The basic application is Deep Q-learning (DQN), where the full state description is input for the neural network, aiming to replicate a lookup table that returns accurate values corresponding to each state (Lillicrap et al. 2015). As with regular Q-learning, this approach requires the state space to be relatively small. Say et al. (2017) use neural networks to capture transition functions rather than VFAs, also applying linear programming in this context. Schmidhuber (2015) provides a survey of deep learning studies, including the use of neural networks in reinforcement learning. The neural networks are generally used to learn values associated to state-action pairs, i.e., as nonlinear VFAs.

Finally, we discuss linear programming in RL. Two approaches may be distinguished, namely (i) solving the full RL model directly by linear programming (also known as approximate linear programming) and (ii) using linear programming to formulate and solve the decision problem encountered in value iteration (the use in this paper). The latter implies solving a linear program for each state that is encountered. De Farias and Van Roy (2003) study the approximate linear programming approach for RL. They use linearly defined VFAs to deal with large state spaces and constraint sampling to deal with large action spaces action. Powell (2016) discusses the use of linear programming in the context of value iteration, solving a linear program at each decision epoch. Decision problems with tens of thousands dimensions can generally be handled with modern commercial solvers. However, when instances become too vast, also linear programming will result in unsatisfactory computational times, requiring to reduce the action space. For very large or complicated combinatorial problems we often resort to meta-heuristics (Powell 2009), which typically embed forms of mathematical programming as well.

## 3 SOLUTION METHOD

We briefly introduce the notation for Markov decision problems (MDPs) as used in this paper. MDP models are useful to mathematically model decision problems with stochastic and dynamic properties. In OR, many decision problems are combinatorial optimization problems. An MDP is a stochastic control process for which the objective is to maximize rewards (or minimize costs) over a discrete time horizon $\mathscr{T}$, with decision epochs $t \in \mathscr{T}$ separated by equidistant time intervals. A discounted MDP can be described by the tuple $(\mathscr{S}, \mathscr{X}(S), \mathbb{P}(S' \mid S,x), R(S,x), \rho)$, with $\mathscr{S}$ being the set of problem states, $\mathscr{X}(S)$ being the set of feasible actions when in state $S \in \mathscr{S}$, $\mathbb{P}(S' \mid S,x)$ being the transition probability of moving from state $S$ to $S' \in \mathscr{S}' \subseteq \mathscr{S}$ after taking action $x \in \mathscr{X}(S)$, $R(S,x)$ being the direct reward when taking action $x$ in state $S$, and $\rho \in [0,1)$ being the discount rate applied to future rewards. The Bellman equation (Bellman 1952) yields the maximum value corresponding to each state:

$$V(S) = \max_{x \in \mathscr{X}(S)} \left( R(S,x) + \rho \sum_{S' \in \mathscr{S}'} \mathbb{P}\big(S' \mid S,x\big) V(S') \right) .$$

Solving the Bellman equation for all states yields the optimal policy, yet for many realistic problems exact solutions are computationally intractable. The next section addresses this issue.

### 3.1 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning that aims to learn policies for MDPs that are too large to solve exactly within reasonable time. This section provides a short and high-level overview. We refer to Powell and Ma (2011) for an extensive discussion on the topic. At its core, RL uses Monte Carlo simulation to sample rewards and estimate the downstream values of state-action pairs, enabling to learn good policies without exhaustively exploring the MDP. From a computational perspective, problems may arise in three areas of MDPs, namely the sizes of the state space (number of states), action space (number of actions per state) and outcome space (number of possible outcomes per action). Multiple solution approaches exist for each of these areas; we restrict ourselves to the ones related to the VFA method used in this paper.

We start with the outcome space $\mathscr{S}' \subseteq \mathscr{S}$. To identify the best action in any state, the Bellman equation requires computing $V(S')$ for each $S' \in \mathscr{S}'$, where $\mathscr{S}'$ might be unique for each state-action pair. RL circumvents this task by instead attaching a single value to a state-action pair, replacing the stochastic expression $\sum_{S' \in \mathscr{S}'} \mathbb{P}\big(S' \mid S,x\big) V(S')$ with a deterministic value function $V(S,x)$. For each state-action pair, we now only evaluate one downstream value rather than $|\mathscr{S}'|$ outcomes. This downstream value is estimated by repeated Monte Carlo sampling, randomly drawing outcome states $S'$ and observing their corresponding values.

Next, we discuss the state space $\mathscr{S}$. In many optimization problems the state is a high-dimensional vector with numerous possible realizations. Computing the value for each individual state may therefore be intractable. Therefore, we replace the true value function with a value function approximation (VFA) $\bar{V}(S,x)$. The VFA is a function that returns an expected value given a set of features (explanatory variables) that capture the essential information in state-action pairs needed to estimate their downstream value. The VFA design step is further discussed in Section 3.2.

Finally, we address the action space $\mathscr{X}$. In combinatorial problems, this space quickly grows beyond the limits of enumeration. As we need thousands of observations to learn a good policy, each decision problem should typically be solvable within a few seconds. To avoid enumerating the full action space, the decision problem may be expressed as a mathematical program. In particular LPs are well-studied; modern solvers often handle such problems highly efficiently. Mathematical programs can be solved to optimality, while significantly upscaling the action space sizes that can be handled.

We now present the outline of the RL algorithm, based on Powell (2016). We perform $N$ iterations to learn the VFA; each iteration represents a discrete time step. At every iteration $n$, the action maximizes expected value given the prevailing VFA $\bar{V}_{n-1}(\cdot)$, resulting in the following observed value (obtained by solving the LP):

$$\hat{v}_n = \max_{x_n \in \mathscr{X}(S_n)} \left( R(S_n, x_n) + \rho \bar{V}_{n-1}(S_n, x_n) \right) .$$

The difference between expected value for the preceding state-action pair at $n-1$, described by $\bar{V}_{n-1}(S_{n-1}, x_{n-1})$, and the observation at $n$ (i.e., $\hat{v}^n$) is utilized to update the VFA, using an updating function $\bar{V}_n(\cdot) \leftarrow U\left(\bar{V}_{n-1}(\cdot), S_{n-1}, x_{n-1}, \hat{v}_n\right)$. Algorithm 1 shows the outline of the RL algorithm to learn the VFA.

**Algorithm 1** Basic RL simulation algorithm to learn the VFA.

```
1:  initialize $\bar{V}_0(\cdot)$
2:  $n \leftarrow 1$
3:  $S_1 \overset{\mathbb{P}}{\leftarrow} \mathscr{S}$
4:  while $n \leq N$ do
5:      $x_n \leftarrow \operatorname{argmax}_{x_n \in \mathscr{X}(S_n)} \left( R(S_n, x_n) + \rho \bar{V}_{n-1}(S_n, x_n) \right)$
6:      $\hat{v}_n \leftarrow \left( R(S_n, x_n) + \rho \bar{V}_{n-1}(S_n, x_n) \right)$
7:      $\bar{V}_n(\cdot) \leftarrow U\left(\bar{V}_{n-1}(\cdot), S_{n-1}, x_{n-1}, \hat{v}_n\right)$
8:      $\mathscr{S}' \leftarrow (S_n, x_n)$
9:      $S_{n+1} \overset{\mathbb{P}}{\leftarrow} \mathscr{S}'$
10:     $n \leftarrow n+1$
11: end while
12: return $\bar{V}_N(\cdot)$
```

## 3.2 Polynomial VFA (PL-VFA)

This section addresses the VFA in more detail. As mentioned earlier, the algorithm operates on features that are extracted from state-action pairs. Let $\mathscr{F}$ be the set of indicators describing the features, with each indicator $f \in \mathscr{F}$ referring to some representative feature of a state-action pair. We define a mapping $\phi$ that extracts features for any given state-action pair, i.e., $\phi : (S, x) \mapsto \mathbb{R}^{|\mathscr{F}|}$, the corresponding vector of features is $[\phi_f]_{\forall f \in \mathscr{F}}$. Formally, the VFA is described by $(\bar{V} \circ \phi) : \mathscr{S} \times \mathscr{X} \mapsto \mathbb{R}$; we may compute a value for each state-action pair.

VFAs are commonly arranged in polynomial form (PL-VFA), also known as linear function approximation. Let $w_f \in \mathbb{R}$ be a weight associated to feature $\phi_f \in \mathbb{R}$. Then, the polynomial VFA may be described by $\bar{V}(S, x) = \sum_{f \in \mathscr{F}} w_f \phi_f(S, x)$. PL-VFAs are popular for several reasons. Polynomials are able to approximate most functions, an appropriate polynomial in theory may approach the true value function arbitrarily close. Furthermore, although the features might be nonlinear, the expression itself is linear. It can therefore be incorporated into linear programming formulations. Techniques such as temporal-difference learning are applicable to update the feature weights (Sutton and Barto 2018).

Although polynomials might theoretically approximate the true value function, randomly defining a polynomial will likely not lead to good performance (Powell 2016). A properly designed PL-VFA is aligned with the structure of the value function. This manual design of VFAs is a key challenge for successful implementations, requiring careful modeling and testing of individual value functions. This is where the linear formulation becomes restrictive, as features representing higher-order effects must be explicitly modeled. Additional problems arise when we resort to linear programming to handle large action spaces. It then becomes challenging to express non-linear features in linear form. Such conversions often require complicated structures involving many artificial variables.

To overcome the limitations of polynomial VFAs, the VFA may be expressed by neural networks. The nonlinear architecture of such networks allows to unravel complex structures, even when inputs are linear operands of state-action pairs. We further discuss neural network VFAs in the next section.

## 3.3 Neural Network VFA (NN-VFA)

This section addresses VFA design using neural networks; for a general introduction to neural networks we refer to Gurney (2018). In neural network VFAs (NN-VFAs), the feature vector $[\phi_f]_{\forall f \in \mathscr{F}}$ is transformed by a weighted set of nonlinear activation functions (neurons), resulting into a single output value $\bar{V}(S,x)$. Compared to the PL-VFA, the main advantage is that the NN-VFA may learn higher-order effects that are not explicitly defined in the feature vector. We emphasize that the input quality remains crucial for the NN-VFA performance, yet feature design is comparatively easier than for PL-VFAs.

The NN-VFA is composed of an input layer (the feature vector), at least one hidden layer containing neurons (having multiple hidden layers constitutes a 'deep' neural network), and an output layer with a single node that returns the expected value for the given state-action pair (Van Heeswijk and La Poutré 2018). In a fully connected network, every neuron in the network connects to all neurons in the preceding layer. Each neuron receives the inner product of all neurons in the preceding layer and their corresponding output weights as input and transforms it into a single neuron value.

The NN-VFA contains $K \geq 1$ hidden layers; we use $\mathscr{K} \triangleq \{1,...,K\}$ to denote the set of hidden layers. The indicator $k=0$ refers to the input layer that contains the features; layer $K+1$ is the output layer. Furthermore, the index $d_k \in \mathbb{N}$ refers to a specific neuron in layer $k \in \mathscr{K}$, with $\mathscr{D}_k$ denoting the set of neurons in layer $k$. Each neuron represents a nonlinear activation function $\sigma_{d_k}$, corresponding to neuron $d$ in layer $k$. For layers $k>0$, an input weight $w_{d_k,d_{k-1}} \in \mathbb{R}$ describes the weight of a neuron as input for $d_k$; the vector $\vec{w}_{d_k} = [w_{d_k,d_{k-1}}]_{\forall d_{k-1} \in \mathscr{D}_{k-1}}$ denotes all inbound weights for neuron $d_k$.

Neuron values are denoted as follows. The value of neuron $d$ in layer $k$ is described by $y_{d_k}$; the value vector for layer $k$ is given by $\vec{y}_k = [y_{d_k}]_{\forall d_k \in \mathscr{D}_k}$. The input layer equals the features, i.e., $\vec{y}_0 = [y_{d_0}]_{\forall d_0 \in \mathscr{D}_0} \triangleq [\phi_f]_{\forall f \in \mathscr{F}}$, with $|\mathscr{D}_0| = |\mathscr{F}|$. The values of the neurons are expressed by $y_{d_k} \triangleq \sigma_{d_k}(\langle \vec{y}_{k-1}, \vec{w}_{d_k} \rangle)$. Finally, the output value of the network is given by $\bar{V}(S_{n-1}, x_{n-1}) \triangleq y_{d_{K+1}} = \langle \vec{y}_K, \vec{w}_{d_{K+1}} \rangle$.

Activation functions in neural networks are by definition nonlinear. Therefore, they cannot be directly computed when embedded in linear programs. However, most common activation functions in contemporary neural networks can be modeled by simple piecewise linear functions. Integration of the NN-VFA in linear programs for decision-making is discussed in the next section.

## 3.4 Integrating the NN-VFA in LPs

Nowadays, many neural networks use (variants of) rectified linear units (ReLUs) as activation functions (Wilmanski et al. 2016). A ReLU returns either 0 or its input value, whichever is larger. They can be represented by a piecewise linear function with two components, allowing to incorporate them in the LP designed to solve the decision problem. Each state-action pair has a unique expected downstream value. To evaluate actions, the neural network must therefore be expressed as a set of linear equations. We follow an implementation comparable to that of Bunel et al. (2018), using binary variables and big M constraints to correctly compute the ReLU values. Additional artificial variables are required to compute the basis functions corresponding to actions. To preserve linearity of the action problem, the features should be linear expressions that can be derived from $[S,x]$.

The updating function $U(\cdot)$ is represented by a stochastic gradient descent (SGD) algorithm to update the weights, meaning that the network weights are adjusted after each iteration and corresponding observation $\hat{v}_n$

(Haykin 2009). At $n=0$, we use He initialization to generate starting values for the weights (He et al. 2015). The learning rate $\eta \in (0,1]$ determines how responsive the weights are to observations deviating from the estimate.

The following constraints (valid $\forall [k,d_k] \in \mathscr{K} \times \mathscr{D}_k$ and $M^+$ being a sufficiently large number) describe the ReLU values within the decision-making LP:

$$
\begin{aligned}
y_{d_k} &\geq 0 \quad , \\
y_{d_k} &\geq \sum_{d_{k-1} \in \mathscr{D}_{k-1}} w_{d_{k-1},d_k} \cdot y_{d_{k-1}} \quad , \\
y_{d_k} &\leq z_{d_k} \cdot M^+ \quad , \\
y_{d_k} &\leq (1 - z_{d_k}) \cdot M^+ + \sum_{d_{k-1} \in \mathscr{D}_{k-1}} w_{d_{k-1},d_k} \cdot y_{d_{k-1}} \quad , \\
z_{d_k} &\geq \frac{\sum_{d_{k-1} \in \mathscr{D}_{k-1}} w_{d_{k-1},d_k} \cdot y_{d_{k-1}}}{M^+} \quad , \\
z_{d_k} &\leq 1 + \frac{\sum_{d_{k-1} \in \mathscr{D}_{k-1}} w_{d_{k-1},d_k} \cdot y_{d_{k-1}}}{M^+} \quad , \\
y_{d_k} &\in \mathbb{R}, z_{d_k} \in \{0,1\} \quad .
\end{aligned}
$$

To summarize: in order to embed the NN-VFA into the problem-specific LP, we (i) add linear expressions that compute features based on the state-action pair, (ii) express the ReLU values for all hidden nodes, and (iii) compute $\bar{V}(S_{n-1}, x_{n-1})$ as output of the neural network.

## 4 EXPERIMENTAL DESIGN

To validate the behavior and performance of the NN-VFA, we run a number of simulation experiments that compare it to various benchmarks. We evaluate both the behavior of the VFAs $\bar{V}_N(\cdot)$ under varying circumstances and the performance $R(\cdot)$ of the resulting policies.

The first benchmark is linear programming (LP) without lookahead, i.e., only maximizing the current reward. This may be viewed as the classical deterministic OR approach, in which probabilistic future information is ignored. For the second benchmark, we perform deep reinforcement learning (DQN), using the full state description as input layer for the network (rather than manually defined features). The third benchmark is the PL-VFA, using the same features as the NN-VFAs, but organized in a linear fashion.

For a clear comparison that distills the essential insights, we keep the applications basic. For the weight updating function $U(\cdot)$, we use TD(0) for the PL-VFA and SGD for DQN and the NN-VFAs. To allow for a fair comparison between methods, we utilize the same learning rate $\eta$; however, various learning rates are tested for the initial calibration. In all cases, we use He initialization to set the weights at $n=0$. We use pure exploration to acquire value observations, i.e., each decision maximizes the expected value given the prevailing policy. Furthermore, we deliberately do not put excessive effort into designing and fine-tuning features; after all, the main goal of the NN-VFA is to reduce the manual design effort compared to the PL-VFA.

The experiments compare two neural network VFAs: the NN(1,20)-VFA (1 layer, 20 neurons) and the NN(3,20)-VFA (3 layers, 20 neurons per layer) to the single-period LP (no lookahead), two DQNs (DQN(1,20) and DQN(3,20), note the first is technically not 'deep'), and a PL-VFA. Although a single-layer network theoretically suffices to learn a function, deep neural networks may model the same function with significantly fewer neurons (Delalleau and Bengio 2011). In fact, for many common functions, the required number of neurons decreases exponentially with the number of layers (Lin et al. 2017). Rolnick and Tegmark (2018) suggest that, for many functions encountered in practical settings, relatively small networks suffice to accurately describe functions. Downsides of deeper networks are the longer training time and potential loss of information (Huang et al. 2016).

The experimental design is as follows. First, we compare convergence properties of the algorithms, i.e., the number of iterations required for convergence to a stable policy. Second, we perform experiments on neural network configurations. We test various numbers of hidden layers and hidden nodes, as well as several learning rates. These experiments provide insight into the behavior and robustness of the NN-VFA under varying conditions. Third, we report the computational times corresponding to multiple NN-VFAs and benchmarks policies, providing insights into the impact of adding nodes and layers. Fourth, we evaluate the performance (i.e., by measuring the direct rewards) of the tested solution methods. We measure offline performances by fixing the policy after every 10,000 training iterations and subsequently perform 10,000 performance iterations. These experiments show how policy quality changes when performing additional training iterations, which is valuable when computational budgets are limited. In total we perform $N = 100{,}000$ training iterations, resulting in 10 offline policies per solution method. All procedures are coded in C++ and CPLEX 12.8 is used to solve the linear decision problems. The experiments run on a 64-bit Linux machine with a 4x1.60GHz CPU and 8GB RAM.

## 4.1 Problem Definition

This section outlines the transportation problem inspired by the nomadic trucker problem (Powell et al. 2007). Our main deviation is that the trucker may load and unload jobs at any location, being rewarded also when bringing a job closer to its destination. At every vertex, the truck decides which adjacent vertex to visit (or stay). Both deterministically known jobs and probabilistic arrivals influence this decision. A simplified representation of the problem is sketched in Figure 1. The problem is characterized by a large discrete action space and a complex optimal policy. Let a strongly connected graph $\{\mathscr{V},\mathscr{E}\}$ represent a transport network. Vertex set $\mathscr{V}$ represents the potential origins and destinations of transport jobs. Edge set $\mathscr{E}$ specifies the undirected connections between vertices; each edge has travel time 1. Edge lengths are $L^2$ distances between vertex pairs and are used to compute travel costs. A capacitated agent roams the graph, traveling between directly connected vertices. At each decision epoch $t \in \mathscr{T}$, the agent decides (i) which jobs to load, (ii) which jobs to unload, (iii) which vertex to visit next (including the option to remain at the current vertex).

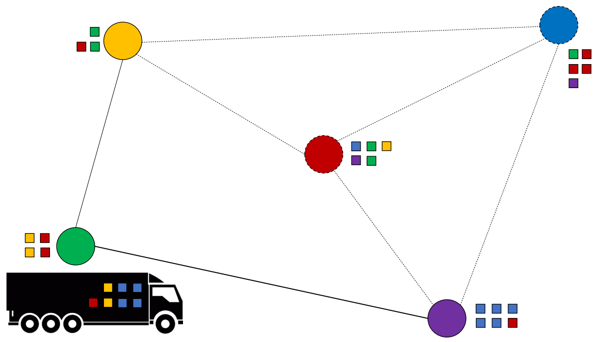


**Figure 1:** Simplified problem representation.

We sketch the corresponding MDP. The problem state $S$ contains the information necessary for decision-making, including the relevant properties of all transport jobs in the graph and the current location of the agent $v^{loc} \in \mathscr{V}$. Each job is defined by four properties, namely (i) the vertex $v \in \mathscr{V}$ at which the job is currently located, (ii) the destination vertex $v^+ \in \mathscr{V}$, (iii) the time remaining until the due date $t^+ \in \mathscr{T}^+$, and (iv) the assignment status $a \in \{0,1\}$ ($a = 1$ means the job is currently carried by the agent). Each unique combination of properties constitutes a job type $j \triangleq [v,v^+,t^+,a]$; the number of jobs per type is denoted by $I_j$. For the full system we define the vector $I = [I_j]_{\forall j}$. The problem state is given by $S \triangleq [I,v^{loc}]$; the set containing all possible states is denoted by $\mathscr{S}$.

We proceed to describe the action $x$, which is a vector describing the number of each job type (un)loaded and the next vertex to visit. Let $\mathscr{V}^{adj}_{v^{loc}} \subseteq \mathscr{V}$ be the set containing both $v^{loc}$ and the vertices adjacent to it. The variable $v^{nxt} \in \mathscr{V}^{adj}_{v^{loc}}$ describes the next destination of the agent. Furthermore, $\gamma = 0$ indicates that a job is unloaded and

$\gamma = 1$ that it is loaded. The action is defined by $x(S) = [x_{j,v^{nxt}}]_{\forall [j,v^{nxt}]}$. The action space $\mathscr{X}(S)$ is bound to various constraints; due to space limitations the full LP model for decision-making is omitted. The key constraints are straightforward though: (i) the agent may only (un)load at its current location, (ii) jobs are always unloaded when at their destination vertex, and (iii) the agent's transport capacity may not be exceeded.

Next, we describe the reward function $R(S,x)$. The rewards consist of: (i) a fixed reward for each successful delivery and (ii) a reward for bringing a job closer to its destination, proportional to the reduction in shortest path distance (an increase in distance yields a negative 'reward'). We proceed to discuss the costs: (i) a fixed cost per distance unit covered (i.e., a cost associated with each edge), independent of the number of jobs carried, (ii) a fixed cost associated with each job that is (un)loaded, and (iii) a penalty for violating due dates. Jobs may be voluntarily unloaded by the agent or forced to be unloaded when $t^+ = 0$, i.e., when the due date has been reached. The reward components are linear with respect to jobs and distances, as to not give NN-VFAs an unfair advantage by introducing obvious nonlinearities.

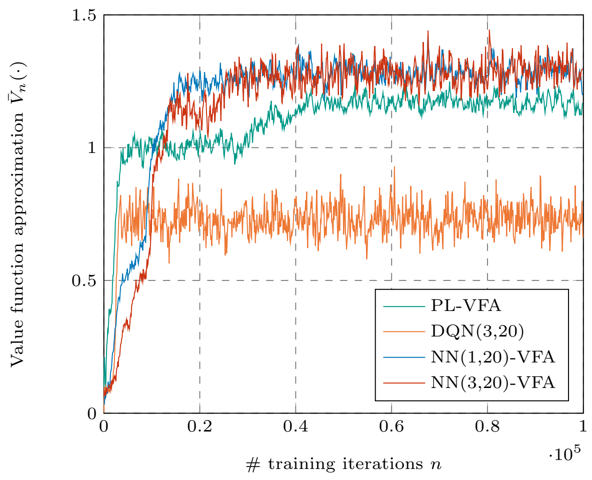
Feature design reflects the components of the reward function. The features are low-level and expressed by linear equations based on $[S,x]$. We define the following features: a bias scalar, the number of jobs carried by the agent, the location of the agent, the number of jobs per vertex in $\mathscr{V}^{adj}$, the total time slack per neighboring vertex, and the most likely vertex to visit after visiting the neighboring vertex (given the weighted shortest path of each job). The total number of features is $2+3\cdot|\mathscr{V}|$.

For the experiments, we design an instance with $|\mathscr{V}| = 5$, a maximum degree of 3, and up to 5 new jobs generated per vertex at each epoch, with accumulation possible up to 45 jobs. The agent may carry up to 20 jobs. The action space grows exponentially with the number of jobs, rendering enumeration infeasible even for this modestly-sized instance: an indicative upper bound for the size of $|\mathscr{X}|$ is given by $2^{20}\cdot 2^{45}\cdot(3+1)$, representing the maximum number of combinations of drop-offs, pick-ups and destinations that may be conceived.

## 5 NUMERICAL RESULTS

This section discusses the experimental results. We start with the convergence results. Preliminary experiments on simplified problem settings with trivial policies indicate that all VFAs work correctly, converging to the true optimal value function, i.e., $\bar{V}_N(\cdot) \approx V(\cdot)$. Figure 2 shows a convergence example for the real problem instance. The PL-VFA converges fastest, but to considerably lower values than the NN-VFAs. Similarly, the NN(1,20)-VFA converges faster than the NN(3,20)-VFA, but to somewhat lower values. The DQNs converge to the same value as the LP with lookahead (to aid the representation, only DQN(3,20) is shown in Figure 2), implying that they are not suitable to learn future values. Most likely, this is due to the high dimension of the problem state.

We proceed to address learning rates, testing for $\eta = \{0.001, 0.01, 0.1\}$. Figure 3 illustrates the convergence speeds per learning rate for the NN(1,20)-VFA; to aid the visual representation, we omit the other VFAs (which display comparable behavior). For completeness, we mention that the NN(3,20)-VFA with $\eta = 0.1$ does not converge to a stable policy. In general, we find that deeper neural networks are less robust with respect to larger learning rates. Errors may be magnified when passing through multiple layers, returning extreme values. Furthermore, NN-VFAs with $\eta = 0.001$ do not converge within 100,000 iterations. We therefore use $\eta = 0.01$ onwards.



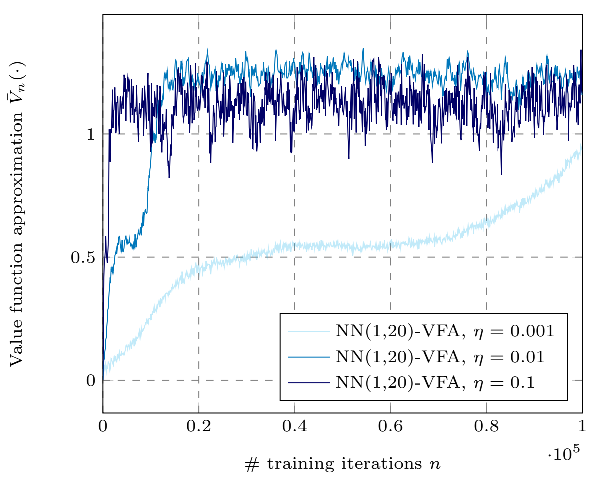


**Figure 2:** Example of $\bar{V}_n(\cdot)$ convergence for various VFAs. **Figure 3:** $\bar{V}_n(\cdot)$ for NN(1,20), using various learning rates $\eta$.

Next, we look at the effects of altering neural network configurations, varying the number of neurons per layer. The results are shown in Table 1. The performance differences between the configurations are relatively small, with the exception of the NN(1,10)-VFA, which performs notably worse than other configurations. Balancing performance and speed, we use 20 neurons per layer for the remainder of the experiments.

**Table 1:** Policy performance $R(\cdot)$ for the NN(1,·)-VFA and NN(3,·)-VFAs, normalized w.r.t. best VFAs.

| | **# neurons per layer** | | | | |
|---|---|---|---|---|---|
| **Solution method** | **10** | **15** | **20** | **25** | **30** |
| NN(1,·)-VFA | 0.66 | 0.91 | 1.00 | 0.92 | 0.98 |
| NN(3,·)-VFA | 0.92 | 0.99 | 0.99 | 0.98 | 1.00 |

We assess the computational time per iteration; roughly 99% of the computational budget is allocated to solving the LPs. On average, the polynomial VFA is solved in 0.02*s* per iteration, the NN(1,20)-VFA takes 0.16*s*, and the NN(3,20)-VFA takes 0.39*s*. Due to the additional sets of variables and constraints, NN-VFAs are inherently slower to compute than the PL-VFA. Table 2 shows the times for other network configurations also; both adding layers and neurons considerably increases the computational effort. It is important to note that – although we can perform many more iterations for a PL-VFA than for an NN-VFA with the same computational budget – the VFAs converge to stable policies; even increasing the number of iterations considerably beyond 100,000 will not improve performance.

**Table 2:** Computational time (in *s*) per iteration for various VFAs.

| **Solution** | | **# neurons per layer** | | | | |
|---|---|---|---|---|---|---|
| **method** | | **10** | **15** | **20** | **25** | **30** |
| LP | 0.02 | - | - | - | - | - |
| DQN(1,·) | - | - | - | 0.18 | - | - |
| DQN(3,·) | - | - | - | 0.41 | - | - |
| PL-VFA | 0.02 | - | - | - | - | - |
| NN(1,·)-VFA | - | 0.07 | 0.15 | 0.16 | 0.17 | 0.19 |
| NN(3,·)-VFA | - | 0.15 | 0.33 | 0.39 | 0.66 | 0.71 |

To conclude the experiments, we reflect on the qualities of NN-VFA policies. An example of offline performances – measured by testing the policies learned after each 10,000 training iterations – is shown in Figure 4. Both LP without lookahead and DQN (equivalent in terms of performance) are ineffective in this problem setting, decidedly being outperformed by the VFA methods. From 20,000 iterations onwards, the PL-VFA is considerably outperformed. In general, we note that the PL-VFA needs relatively few iterations to converge to a stable – but often clearly suboptimal – policy. Furthermore, the NN(3,20)-VFA performs better than the NN(1,20)-VFA and is slightly more stable over time.

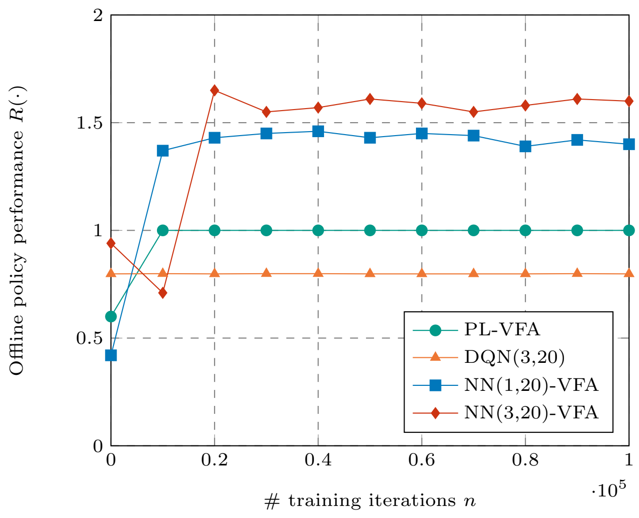


**Figure 4:** Example of offline policy performance $R(\cdot)$ for various $\bar{V}_n, n \in \{0, N\}$

Table 3 shows the average policy performance of repeated replications, measured after completing $N = 100{,}000$ training iterations. The NN(1,20)-VFA outperforms the PL-VFA by 10.1% and the NN(3,20)-VFA does so by 19.6%. Although both NN-VFAs achieve comparable policy performance at times, the NN(1,20)-VFA is more prone to fluctuating performances, which is also indicated by its higher standard deviation. Furthermore, the NN(3,20)-VFA has more expressive power, enabling more accurate approximations. The results demonstrate that – given sufficient training iterations – NN-VFAs significantly outperform the PL-VFA for our transportation problem. The DQN networks and the LP approach have equal performance; they do not anticipate downstream costs and therefore have no standard deviation.

**Table 3:** Average policy performance $R(\cdot)$ after $N$ iterations, normalized w.r.t. PL-VFA.

| **Solution method** | **Mean** | **St. dev.** |
|---|---|---|
| LP | 0.798 | 0.00% |
| DQN(1,20) | 0.798 | 0.00% |
| DQN(3,20) | 0.798 | 0.00% |
| PL-VFA | 1.000 | 1.25% |
| NN(1,20)-VFA | 1.101 | 1.65% |
| NN(3,20)-VFA | 1.196 | 0.72% |

## 6 CONCLUSIONS

This paper proposes the integration of linear programs for decision-making and value function approximations in the form of neural networks, geared towards solving high-dimensional and combinatorial problems in operations research. Our hybrid method is rooted in the paradigm of reinforcement learning with value function approximations. Traditionally, large action spaces in OR problems are handled by formulating the decision problem as a linear program, yet it is difficult to properly define polynomial VFAs in this context.

The main contribution of the neural network VFA is the reduced effort of manual feature design, which is a crucial and precarious step in all solutions relying on VFAs. Unlike PL-VFAs, NN-VFAs are able to learn higher-order effects of simple input features without explicitly designing them, reducing the effort for manual feature design. This is particularly relevant when embedding VFAs in linear programs, in which the design of nonlinear features may be a cumbersome task.

We test our solution method on a representative transportation problem with a large discrete action space, a complex optimal policy, and a multi-component reward function. We compare NN-VFAs to an LP without lookahead, two DQN architectures and a PL-VFA, keeping all other factors equal. For our test problem, we observe significant improvements in performance, with the best NN-VFA outperforming the PL-VFA by 19.6%. The findings are also robust across neural network configurations; with various settings for training iterations, learning rates, neurons, and layers, the benchmarks are consistently outperformed. NN-VFAs with multiple hidden layers yield the best and most stable policies, but also require more iterations to converge and more computational effort per iteration. This paper presents an exploration with respect to the integration of LPs and NN-VFAs; additional research on different problem classes is needed to draw more general conclusions about the approach. Furthermore, advanced techniques in neural network configuration and weight updates may reduce training time and/or improve performance. In our opinion, the obtained results warrant such further studies.

## ACKNOWLEDGMENTS

This work is part of the research program Scalable Interoperability in Information Systems for Agile Supply Chains (SIISASC) with project number 438-13-603, which is partially funded by the Netherlands Organization for Scientific Research (NWO). The truck icon in Figure 1 is obtained from http://www.flaticon.com and freely available under the Flaticon Basic License.

## REFERENCES

Bellman, R. E. 1952. "On the Theory of Dynamic Programming". *Proceedings of the National Academy of Sciences of the United States of America* 38(8):716–719.

Bertsekas, D. P. 2012. *Dynamic Programming and Optimal Control*. 4th ed, Volume II. Belmont, Massachusetts: Athena Scientific.

Bertsekas, D. P., and J. Tsitsiklis. 1995. "Neuro-Dynamic Programming: An Overview". In *Proceedings of the 34th IEEE Conference on Decision and Control*. December 13$^{th}$-15$^{th}$, Louisiana, New Orleans, 560-564.

Bunel, R., I. Turkaslan, P. H. S. Torr, P. Kohli, and M. P. Kumar. 2018. "A Unified View of Piecewise Linear Neural Network Verification". In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, 4795–4804. Red Hook, New York: Curran Associates Inc.

Busoniu, L., R. Babuska, B. De Schutter, and D. Ernst. 2017. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Boca Raton, Florida: CRC Press.

De Farias, D. P., and B. Van Roy. 2003. "The Linear Programming Approach to Approximate Dynamic Programming". *Operations Research* 51(6):850–865.

Delalleau, O., and Y. Bengio. 2011. "Shallow vs. Deep Sum-Product Networks". In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, edited by J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, 666–674. Red Hook, New York: Curran Associates Inc.

Dulac-Arnold, G., L. Denoyer, P. Preux, and P. Gallinari. 2012. "Fast Reinforcement Learning with Large Action Sets Using Error-Correcting Output Codes for MDP Factorization". In *Machine Learning and Knowledge Discovery in Databases*, edited by P. A. Flach, T. De Bie, and N. Cristianini, 180–194. Berlin Heidelberg, Germany: Springer-Verlag.

Dulac-Arnold, G., R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin. 2015. "Deep Reinforcement Learning in Large Discrete Action Spaces". *arXiv preprint arXiv:1512.07679*.

Gurney, K. N. 2018. *An Introduction to Neural Networks*. 1st ed. London, United Kingdom: CRC Press.

Haykin, S. 2009. *Neural Networks and Learning Machines*. 3rd ed. Upper Saddle River, New Jersey: Pearson.

He, K., X. Zhang, S. Ren, and J. Sun. 2015. "Delving Deep Into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In *Proceedings of the 2015 IEEE International Conference on Computer Vision*. December 11th-18th, Santiago, Chile, 1026-1034.

Huang, G., Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger. 2016. "Deep Networks with Stochastic Depth". In *Proceedings of the 2016 European Conference on Computer Vision*, edited by B. Leibe, J. Matas, N. Sebe, and M. Welling, 646–661. Cham, Switzerland: Springer International Publishing.

Keller, P. W., S. Mannor, and D. Precup. 2006. "Automatic Basis Function Construction for Approximate Dynamic Programming and Reinforcement Learning". In *Proceedings of the 23rd International Conference on Machine Learning*, edited by W. W. Cohen and A. Moore, 449–456. Madison, Wisconsin: Omnipress.

LeCun, Y., Y. Bengio, and G. Hinton. 2015. "Deep Learning". *Nature* 521:436–444.

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. 2015. "Continuous Control with Deep Reinforcement Learning". *arXiv preprint arXiv:1509.02971*.

Lin, H., M. Tegmark, and D. Rolnick. 2017. "Why Does Deep and Cheap Learning Work So Well?". *Journal of Statistical Physics* 168(6):1223–1247.

Pazis, J., and R. Parr. 2011. "Generalized Value Functions for Large Action Sets". In *Proceedings of the 28th International Conference on Machine Learning*, edited by J. Dy and A. Krause, 1185–1192. Madison, Wisconsin: Omnipress.

Powell, W. B. 2009. "What You Should Know About Approximate Dynamic Programming". *Naval Research Logistics* 56(3):239–249.

Powell, W. B. 2016. "Perspectives of Approximate Dynamic Programming". *Annals of Operations Research* 241(1-2):319–356.

Powell, W. B., B. Bouzaiene-Ayari, and H. P. Simao. 2007. "Dynamic Models for Freight Transportation". *Handbooks in Operations Research and Management Science* 14:285–365.

Powell, W. B., and J. Ma. 2011. "A Review of Stochastic Algorithms with Continuous Value Function Approximation and Some New Approximate Policy Iteration Algorithms for Multidimensional Continuous Applications". *Journal of Control Theory and Applications* 9(3):336–352.

Rolnick, D., and M. Tegmark. 2018. "The Power of Deeper Networks for Expressing Natural Functions". In *Proceedings of the 2018 International Conference on Learning Representations*. April 30th - May 3rd, Vancouver, Canada, 1-14.

Say, B., G. Wu, Y. Q. Zhou, and S. Sanner. 2017. "Nonlinear Hybrid Planning with Deep Net Learned Transition Models and Mixed-Integer Linear Programming". In *Proceedings of the 2017 International Joint Conferences on Artificial Intelligence*, edited by C. Sierra, 750–756. Marina Del Rey, California: International Joint Conferences on Artificial Intelligence.

Schmidhuber, J. 2015. "Deep Learning in Neural Networks: An Overview". *Neural Networks* 61:85–117.

Sutton, R. S., and A. G. Barto. 2018. *Reinforcement Learning: An Introduction*. 2nd ed. Cambridge, Massachusetts: The MIT Press.

Van Heeswijk, W. J. A., and H. La Poutré. 2018. "Scalability and Performance of Decentralized Planning in Flexible Transport Networks". In *Proceedings of the 2018 IEEE International Conference on Systems, Man, and Cybernetics*. October 7th-10th, Miyazaki, Japan, 292-297.

Wilmanski, M., C. Kreucher, and J. Lauer. 2016. "Modern Approaches in Deep Learning for SAR ATR". In *Algorithms for Synthetic Aperture Radar Imagery XXIII*, edited by E. Zelnio and F. D. Garber, Volume 9843, 195 – 204. Bellingham, Washington: International Society for Optics and Photonics.

## AUTHOR BIOGRAPHIES

**WOUTER VAN HEESWIJK** is an assistant professor in the Department of Industrial Engineering & Business Information Systems at the University of Twente, previously working as a postdoctoral researcher at Centrum Wiskunde & Informatica. He holds a master's degree in Financial Engineering and a PhD in Operations Research. His research interests include multi-agent simulation and stochastic optimization, focusing primarily on reinforcement learning. His email address is w.j.a.vanheeswijk@utwente.nl.

**HAN LA POUTRÉ** is a senior researcher at Centrum Wiskunde & Informatica and full professor at the Technical University of Delft. He performs research on multi-agent simulation and computational intelligence techniques. His research is multidisciplinary, in cooperation with electrical engineering and economics. His early research concerned algorithms. He was member of the Scientific Directory of Schloss Dagstuhl and the editorial board of ACM TOIT. His email address is han.la.poutre@cwi.nl.