1234

Missile Defense and Interceptor Allocation by Neuro-Dynamic Programming

Dimitri P. Bertsekas, Mark L. Homer,
David A. Logan, Stephen D. Patek, and Nils R. Sandell

Abstract— **The purpose of this paper is to propose a solution methodology for a missile defense problem involving the sequential allocation of defensive resources over a series of engagements. The problem is cast as a dynamic programming/Markovian decision problem, which is computationally intractable by exact methods because of its large number of states and its complex modeling issues. We have employed a Neuro-Dynamic Programming (NDP) framework, whereby the cost-to-go function is approximated using neural network architectures that are trained on simulated data. We report on the performance obtained using several different training methods, and we compare this performance with the optimal.**

Keywords— **Theater Missile Defense, Dynamic Programming, Neuro-Dynamic Programming, Reinforcement Learning.**

I. Introduction

In this paper we consider a complex dynamic interceptor allocation problem that is typical of Theater Missile Defense (TMD). We assume that the defense allocates interceptors to counter an opponent, who has a limited inventory of ballistic missiles that can be used to attack assets (cities, air fields, etc.). Due to the limited number of missile launchers, and also due to the offense's desire to conserve its missile inventory for future contingencies, the offense typically does not launch all its missiles simultaneously but rather in discrete attack waves. These waves may be spaced hours apart, and there may be multiple waves per day over a period of many days. Thus the TMD interceptor allocation problem is a dynamic decision problem, since a series of decisions must be made over an extended period of time, with the consequences of a given decision affecting the subsequent decisions.

The defense has an inventory of interceptors and a set of launchers. The decisions to be made concern the utilization of interceptors. Given the current attack wave, the defense must decide on how many interceptors to use against the current wave and how many to withhold for use against future waves. The number of interceptors fired per wave is constrained by the number of launchers. Furthermore, the defense must assign individual interceptors to specific attacking missiles.

The problem is complicated by the presence of multiple types of assets with different values and probabilities of destruction when attacked by different types of missiles, and also by the presence of multiple types of interceptors with different effectiveness against different types of missiles. As a result the state space and the control space of the problem are very large (Bellman's "curse of dimensionality"), and while there is some favorable structure that can be exploited, the exact solution by dynamic programming (DP) is impractical for realistically sized problem.

In this paper, we describe a mathematical optimization model of TMD, and a *Neuro-Dynamic Programming* (NDP) approach that we have used for its solution. NDP is a class of reinforcement learning methods that deals with the curse of dimensionality by using neural network-based approximations of the optimal cost-to-go function. NDP has the further advantage that it does not require an explicit system model; it uses a simulator, as a model substitute, in order to train the neural network architectures and to obtain suboptimal policies. We refer to the textbook [1], the research monographs [2] and [3], and the survey [4] for descriptions of the NDP methodology and detailed references.

We present the results of our computational experimentation with what seem to be the most promising methods for our problem, and we evaluate the effectiveness of the NDP formulation for TMD. Our aim is not only to delineate the best methods for the TMD model discussed here, but also to develop reliable guidelines for the methods that are likely to be effective when applied to other more complex TMD problems. The results of the present paper are also relevant to a number of logistics and resource allocation problems that have a structure similar to TMD; for example in Section 2.4 of [2], a machine maintenance problem is described that has a structure almost identical to TMD (missiles can be identified with machine breakdowns, interceptors can be identified with spare parts that can be used to repair the breakdowns, and launchers can be identified with repairmen).

II. Problem Formulation

The basic elements of the problem are the assets of different types, the missiles available to the attacker, and the interceptors available to the defender. The interceptors are used to destroy the attacker's missiles. We denote by $p_{D,t}(m,n)$ the probability that an asset of type t will be de-

Dimitri Bertsekas is with Massachusetts Institute of Technology, Laboratory for Information and Decision Systems, Room 35-210, Cambridge, MA 02139 (dimitrib@mit.edu)

David Logan and Nils Sandell are with ALPHATECH, Inc., 50 Mall Road, Burlington, MA 01803 (david.logan@alphatech.com, nils.sandell@alphatech.com). Mark Homer was with ALPHATECH, Inc. at the time this research was conducted.

Stephen Patek is with the University of Virginia, Department of Systems Engineering, 101C Olsson Hall, Charlottesville, VA 22903-2442 (patek@virginia.edu).

This research was supported by a Small Business Innovative Research (SBIR) Grant at ALPHATECH, Inc.

stroyed when it is attacked by m missiles and it is defended by n interceptors. This probability may not be known explicitly, but is at least built into a battle simulator, which may be parameterized by more fundamental quantities. We assume that the action takes place in discrete time periods. There is a maximum number L_M of missiles, and a maximum number L_I of interceptors that can be launched at each time period, corresponding to the respective numbers of launchers.

We assume that attacks in different time periods are independent, and that the number of missiles launched and the assets targeted for attack are selected by a probabilistic mechanism. Without loss of generality, we assume that at least one missile will be launched at each time period. This guarantees that the battle will terminate in finite time (due to either exhaustion of the missiles or destruction of all the assets). We assume that each asset of type t has value V_t, and the objective is to maximize the expected value of the assets that are surviving at the end of the battle.

We formulate the problem as a stochastic shortest path problem of the type considered in [5], [6], [1], or [2]. The state has two components. The first component is

$$i = (A_1, \ldots, A_n, I, M),$$

where A_t is the number of surviving assets of type t, n is the number of asset types, I is the number of interceptors, and M is the number of missiles. This is the major component of the state, and it is also referred to as the *reduced state*. However, there is a second component of the state, which is the *current attack vector*, given by

$$a = (a_1, \ldots, a_A),$$

where $A = A_1 + \cdots + A_n$ is the total number of surviving assets, and a_j is the number of missiles attacking the jth asset. The control to be selected at a given state is the choice of interceptors to counter each attacking missile, and is modeled as the vector

$$d = (d_1, \ldots, d_A),$$

where d_j is the number of interceptors defending the jth asset. We refer to d as the *defense vector*. We use $p_{(i,a)(i',a')}(d)$ to denote the probability that the new state will be (i', a') given that (1) the current assets, missiles, and interceptors are $i = (A_1, \ldots, A_n, I, M)$, (2) the attack a occurs; and (3) the defense d is chosen. These are the transition probabilities that are specified by the problem data. [Although the NDP methodology does not require that the probabilities $p_{(i,a)(i',a')}(d)$ be known explicitly (as long as they are built into a simulation of the process being optimized), it is useful to have a notation for them.] We assume that we have perfect state information, including the number of remaining missiles of the attacker.

Because the conditional probability of the next attack a' given i' is independent of d, we have

$$p_{(i,a)(i',a')}(d) = P\{i' \mid (i,a), d\} P\{a' \mid i'\},$$

where
1. $P\{i' \mid (i,a), d\}$ is the probability that i' is the next profile of assets, missiles, and interceptors given that i is the current profile and that a is the current attack and
2. $P\{a' \mid i'\}$ is the conditional probability that the next attack will be a' given the next profile of surviving assets, missiles, and interceptors i'.

The above structure of the transition probabilities simplifies Bellman's equation for the stochastic shortest path problem. In particular, let $J^*(i,a)$ denote the optimal (minimal) expected long-term cost (due to destroyed assets) starting at state (i,a), and let

$$\hat{J}(i) = \sum_a P\{a \mid i\} J^*(i,a). \tag{1}$$

We refer to $\hat{J}$ as the *reduced optimal cost* at the reduced state i. Then $\hat{J}$ can be shown to satisfy the following reduced form of Bellman's equation ([1] or [2], Section 2.4)

$$\hat{J}(i) = \sum_a P(a \mid i) \min_d E_{i'} \{ g(i,i') + \hat{J}(i') \mid i, a, d \}, \tag{2}$$

where

$$g(i,i') = \sum_t V_t \big(A_t(i) - A_t(i') \big) \tag{3}$$

is the one time period cost (the value of assets destroyed during the period), and $A_t(i)$ [or $A_t(i')$] is the number of assets of type t corresponding to state i (or i', respectively).

Since termination is guaranteed under all policies, by using the available theory for stochastic shortest path problems ([1]), we have that Bellman's Eq. (2) has a unique solution, which is the reduced cost function $\hat{J}(i)$. Furthermore, a stationary policy $\mu(i,a)$, which assigns defense vectors d to states (i,a), is optimal if and only if $\mu(i,a)$ minimizes in the right-hand side of Eq. (2) for every state (i,a).

A. Solution via DP

The problem can in principle be solved by classical methods. In the *value iteration method*, we start with an estimate J_0 of the reduced optimal cost function $\hat{J}$, and we iterate according to

$$J_{k+1}(i) = \sum_a P(a \mid i) \min_d E_{i'} \{ g(i,i') + J_k(i') \mid i, a, d \}. \tag{4}$$

It can be shown that the generated sequences $J_k(i)$ will converge to the (reduced) optimal cost $\hat{J}(i)$ for all states i.

In the classical value iteration method, the estimate of the cost function is iterated for all states simultaneously. An alternative is to iterate one state at a time, while incorporating into the computation the interim results. This method is known as the *Gauss-Seidel method*, and it is valid, in the sense that it converges to $\hat{J}$ under the same conditions that the ordinary method converges. In fact the same result may be shown for a much more general version of the Gauss-Seidel method, called the *asynchronous Gauss-Seidel method*. In this version, it is not necessary to maintain a fixed order for iterating on the cost estimates

$J(i)$ of the different states; an arbitrary order can be used, as long as the cost $J(i)$ of each state i is iterated infinitely often.

In general, value iteration requires an infinite number of iterations to obtain the optimal cost function. However, in our problem there is special structure that can be exploited to obtain a finitely terminating value iteration method. In particular, we observe that the transition probability graph of our problem is acyclic because with each transition the inventory of attacking missiles is strictly reduced. This acyclic structure implies a partial order on the set of states, and if we use a Gauss-Seidel method that uses a state iteration order that is consistent with this partial order, the problem can be solved exactly using only one value iteration per state (see also the discussion of [2], Section 2.2).

An alternative to value iteration is *policy iteration*, which terminates finitely for problems with finite state and action spaces. This algorithm operates as follows: we start with a stationary policy μ_0, and we generate a sequence of new stationary policies $\mu_1, \mu_2, \ldots$ Given the policy μ_k, we perform a *policy evaluation step*, that computes $J_{\mu_k}(i)$, $i = 1, \ldots, n$, as the solution of the (linear) system of equations

$$J(i) = \sum_a P(a \mid i) E_{i'} \big\{ g(i,i') + J(i') \mid i, a, \mu_k(i,a) \big\} \quad (5)$$

in the unknowns $J(1), \ldots, J(n)$. We then perform a *policy improvement step*, which computes a new policy μ_{k+1} as

$$\mu_{k+1}(i,a) = \arg\min_d E_{i'} \big\{ g(i,i') + J_{\mu_k}(i') \mid i, a, d \big\}.$$

The process is repeated with μ_{k+1} used in place of μ_k, unless we have $J_{\mu_{k+1}}(i) = J_{\mu_k}(i)$ for all i, in which case the algorithm terminates with the policy μ_k. It can be shown that the policy iteration algorithm generates an improving sequence of policies [that is, $J_{\mu_{k+1}}(i) \leq J_{\mu_k}(i)$ for all i and k] and terminates with an optimal policy.

When the number of states is large, solving the linear system (5) in the policy evaluation step by direct methods such as Gaussian elimination is time-consuming. One way to get around this difficulty is to solve the linear system iteratively by using value iteration. In fact, we may consider solving the system approximately by executing a limited number of value iterations. This is known as the *modified policy iteration algorithm*, and it is analyzed in several sources, see e.g., [1], Section 2.3, or [7]. It is also possible to use more general, asynchronous versions of policy iteration, where value iterations and policy evaluations are intermingled in a fairly uncoordinated manner (see [8] or [2], Section 2.2). This indicates that the methods of value and policy iteration have a considerable degree of robustness, which is particularly helpful within the simulation-driven approximation context of NDP.

The classical methods are applicable to problems with relatively small numbers of states (say a few thousand at most). Otherwise they are prohibitively time-consuming. In our problem, however, the number of states can easily be very large and far beyond the range of applicability of the classical methods. We are thus forced to consider methods that can produce a suboptimal policy with reasonable amount of computation, as described in the next section.

III. Neuro-Dynamic Programming Framework

The suboptimal solution methods that we have employed center around the evaluation and approximation of the reduced optimal cost function $\hat{J}$, through the use of neural networks and simulation. In particular, we replace the optimal cost $\hat{J}(i)$ with a suitable approximation $\tilde{J}(i,r)$, where r is a vector of parameters, and we use at state i the control $\tilde{\mu}(i,a)$ that attains the minimum in the (approximate) right-hand side of Bellman's equation

$$\tilde{\mu}(i,a) = \arg\min_d E_{i'} \big\{ g(i,i') + \tilde{J}(i',r) \mid i, a, d \big\}. \quad (6)$$

The function $\tilde{J}$ will be called the *scoring function*, and the value $\tilde{J}(i,r)$ will be called the *score* of state i. Scoring functions of this type are known as *compact representations* of the optimal cost function. The methods that we have used also employ compact representations $\tilde{J}(i,r)$ of the reduced cost functions $\hat{J}_\mu(i)$ of stationary policies.

An important issue is the *selection of architecture*, that is, the choice of a parametric class of functions $\tilde{J}(\cdot,r)$. We have experimented with two architectures:

(a) A *neural network/multilayer perceptron* architecture. Here $\tilde{J}(i,r)$ is implemented by a standard multilayer perceptron structure with sigmoidal nonlinearities and a single hidden layer. The state i is the input to this network, the approximation $\tilde{J}(i,r)$ is the output, and r is the vector of weights that are determined by a training algorithm as described below.

(b) A *feature extraction mapping*, that maps the state i into some vector

$$f(i) = \big(f_1(i), \ldots, f_q(i) \big),$$

called the *feature vector* associated with the state i, followed by a *linear* mapping that produces a cost approximation of the form

$$\tilde{J}(i,r) = r_0 + \sum_{j=1}^{q} r_j f_j(i),$$

where $r = (r_0, r_1, \ldots, r_q)$ is the vector of linear weights to be obtained by a training algorithm. Feature vectors summarize, in a heuristic sense, what are considered to be important characteristics of the state, and they are very useful in incorporating prior knowledge or intuition about the problem. For example, an important feature of the state $i = (A_1, \ldots, A_n, I, M)$ is the *expected leakage*, that is, the expected number of missiles that will leak through the defense if each missile is intercepted by the "typical" number $\overline{k}$ of interceptors used to defend against a single missile (for example $\overline{k} = 1$ if only one interceptor is typically used to defend against a single missile).

Some of the most successful applications of neural networks are in the areas of pattern recognition, nonlinear regression, and nonlinear system identification. In these applications the neural network is used as a universal approximator: the input-output mapping of the neural network is matched to an unknown nonlinear mapping F of interest using a least-squares optimization. This optimization is known as *training the network*. To perform training, one must have some training data, that is, a set of pairs $\big(i, F(i)\big)$, which is representative of the mapping F that is approximated.

It is important to note that in contrast with these neural network applications, in our stochastic shortest path context there is no readily available training set of input-output pairs $\big(i, \hat{J}(i)\big)$, which can be used to approximate $\hat{J}$ with a least squares fit. The only possibility is to evaluate (exactly or approximately) by simulation the cost functions of given (suboptimal) policies, and to try to iteratively improve these policies based on the simulation outcomes. This creates analytical and computational difficulties that do not arise in classical neural network training contexts. Indeed the use of simulation to evaluate approximately the optimal cost function is a key new idea, that distinguishes the NDP methodology from earlier approximation methods in DP.

IV. Neuro-Dynamic Programming Methods

Most of the methods that we have concentrated on are approximate versions of policy iteration, whereby a sequence of policies $\{\mu_k\}$ is generated and the corresponding cost functions $\hat{J}_{\mu_k}$ are evaluated approximately using compact representations $\tilde{J}(\cdot, r_k)$. Our approximations have been based on the two architectures described in the preceding section. The training to obtain the parameter vector r_k was performed by using forms of Monte Carlo simulation and least squares fit, as well as the TD(λ) algorithm of Sutton [9], where λ was chosen from the range $[0,1]$.

Let us provide a more detailed description of the training methods and also describe their theoretical convergence properties.

A. Approximate Policy Iteration Using Monte Carlo Simulation

One of the principal methods that we tried is an approximate form of the policy iteration method that uses approximations $\tilde{J}(i,r)$ to the reduced cost $\hat{J}_\mu$ of stationary policies μ. The algorithm alternates between approximate policy evaluation steps and policy improvement steps. The parameter vector r corresponding to the current policy determines the next policy $\overline{\mu}$ via the equation

$$\overline{\mu}(i,a) = \arg\min_d E_{i'}\big\{g(i,i') + \tilde{J}(i',r) \mid i,a,d\big\}, \quad \text{for all } i. \tag{7}$$

The policy $\overline{\mu}$ thus defined can be used to generate by simulation sample trajectories and corresponding sample costs starting from various initial states. The parameter vector r (which induces the policy μ) remains unchanged as the sample trajectories of policy $\overline{\mu}$ are generated. The corresponding sample costs are used in an approximate evaluation of the cost function of $\overline{\mu}$ using a least squares scheme. In particular, suppose that we have a subset of "representative" reduced states $\tilde{S}$, and that for each $i \in \tilde{S}$, we have $M(i)$ samples of the cost $\hat{J}_{\overline{\mu}}(i)$. The mth such sample is denoted by $c(i,m)$. We evaluate the cost of the improved policy $\overline{\mu}$ by solving the least squares problem

$$\min_{\overline{r}} \sum_{i\in\tilde{S}} \sum_{k=1}^{M(i)} \big|\tilde{J}(i,\overline{r}) - c(i,k)\big|^2. \tag{8}$$

The least squares problem (8) that is used to approximate $\hat{J}_{\overline{\mu}}$ can be solved by a gradient-like method. The method that we used operates as follows: Given a sample state trajectory $(i_1, i_2, \ldots, i_N, T)$ of reduced states generated using the policy $\overline{\mu}$ (state T is the termination state), the parameter vector $\overline{r}$ associated with $\overline{\mu}$ is updated by

$$\overline{r} := \overline{r} - \gamma \sum_{k=1}^{N} \nabla\tilde{J}(i_k,\overline{r}) \left(\tilde{J}(i_k,\overline{r}) - \sum_{m=k}^{N} g\big(i_m, i_{m+1}\big)\right), \tag{9}$$

where ∇ denotes gradient, and γ is a stepsize, which, for convergence, should diminish as the number of trajectories used increases. A popular choice is to set $\gamma = c/m$ during the mth trajectory, where c is a constant. The summation in the right-hand side above is a sample gradient corresponding to a term in the least squares summation of problem (8). The iteration (9) can be used to make several passes through the data in order to improve the accuracy of the approximation through convergence of the vector $\overline{r}$ to the least squares solution of problem (8).

B. Approximate Policy Iteration Using TD(1)

There is a temporal differences implementation of the gradient iteration (9). The temporal differences d_k are given by

$$d_k = g\big(i_k, i_{k+1}\big) + \tilde{J}(i_{k+1},\overline{r}) - \tilde{J}(i_k,\overline{r}), \qquad k = 1,\ldots,N, \tag{10}$$

and the iteration (9) can be alternatively written [except for terms that are of order $O(\gamma^2)$], as follows [just add the equations below using the temporal difference expression (10) to obtain the iteration (9)]:

Following the state transition (i_1, i_2), we set

$$\overline{r} := \overline{r} + \gamma d_1 \nabla\tilde{J}(i_1,\overline{r}). \tag{11}$$

Following the state transition (i_2, i_3), we set

$$\overline{r} := \overline{r} + \gamma d_2 \big(\nabla\tilde{J}(i_1,\overline{r}) + \nabla\tilde{J}(i_2,\overline{r})\big), \tag{12}$$

and so on, until following the state transition (i_N, T), we set

$$\overline{r} := \overline{r} + \gamma d_N \big(\nabla\tilde{J}(i_1,\overline{r}) + \nabla\tilde{J}(i_2,\overline{r}) + \cdots + \nabla\tilde{J}(i_N,\overline{r})\big). \tag{13}$$

The vector $\overline{r}$ is updated at each transition and the gradients $\nabla\tilde{J}(i_k,\overline{r})$ are evaluated for the value of $\overline{r}$ that prevails at the time i_k is generated. This is the cause of a slight difference between the "batch" update $\overline{r}$ given by Eq. (9) and the update process (10)-(13) that is based on temporal differences. In particular, $\overline{r}$ in the batch method changes only at the end of trajectories, while $\overline{r}$ in the temporal differences method changes at the end of each state transition. The difference between the updates produced by the two methods is proportional to γ^2 and is negligible when γ is small.

Note that for both the Monte Carlo/batch and the TD(1) implementations, the convergence of the training process to a weight vector $\overline{r}$ that solves the corresponding least squares problem follows from known results on incremental gradient methods (see e.g., [10], Section 1.5 or [2], Section 3.2, and the references quoted there). (Some assumptions on the frequency of sampling of various initial states are required to state a precise convergence result.)

On the other hand, the sequences $\{\mu_k\}$ and $\{\tilde{J}(\cdot, r_k)\}$ produced by the approximate policy iteration method [with both the Monte Carlo/batch implementation and the TD(1) implementation] generally do not converge. A typical behavior is that $\hat{J}_{\mu_k}$ improves in the initial iterations and once it reaches the vicinity of the optimal cost function $\hat{J}(\cdot)$, it tends to oscillate. Error bounds for the sup-norm of the error function $\tilde{J}(\cdot, r_k) - \hat{J}(\cdot)$ have been obtained (see [1] or [2]).

C. Approximate Policy Iteration Using TD(λ)

A variant of TD(1), known as TD(λ) (Sutton [9]), provides an alternative method for policy evaluation. TD(λ) uses a parameter $\lambda \in [0, 1]$ in the formulas (10)-(13). It has the following form:

For $k = 1, \ldots, N$, following the state transition (i_k, i_{k+1}), set

$$\overline{r} := \overline{r} + \gamma d_k \sum_{m=1}^{k} \lambda^{k-m} \nabla \tilde{J}(i_m, \overline{r}). \tag{14}$$

The convergence properties of TD(λ) have been investigated in [11], where it is shown that the method typically converges (with an appropriate choice of stepsize) in the case of a linear architecture. An example of divergence was also given for the case of a nonlinear architecture. Furthermore, the limit of the method (when it converges) is typically not the least squares optimal solution of problem (8), unless $\lambda = 1$. In particular, the limit to which TD(λ) converges depends on λ, and there are simple examples where the approximation error $\tilde{J}(i, r) - \hat{J}_\mu(i)$ corresponding to TD(λ) in the limit progressively becomes worse as λ approaches 0 (see [12]). However, there have been reports of computational studies that have found the use of $\lambda < 1$ preferable to the use of $\lambda = 1$. Basically, as λ becomes smaller, the variance of the "simulation noise" in iteration (14) also becomes smaller (see [2] for a broader discussion).

D. Optimistic Policy Iteration

In the approximate policy iteration approach discussed so far, the least squares problem that evaluates the cost of the improved policy $\overline{\mu}$ must be solved completely for the vector $\overline{r}$. An alternative is to solve this problem approximately and (optimistically) replace the policy μ with the policy $\overline{\mu}$ after a single or a few simulation runs and corresponding updates of the new weight $\overline{r}$. An extreme possibility is to replace μ with $\overline{\mu}$ at the end of each state transition, as in the next algorithm:

Following the state transition (i_k, i_{k+1}), set

$$\overline{r} := \overline{r} + \gamma d_k \sum_{m=1}^{k} \lambda^{k-m} \nabla \tilde{J}(i_m, \overline{r}), \tag{15}$$

and generate the next transition (i_{k+1}, i_{k+2}) by simulation using the control

$$\overline{\mu}(i_{k+1}, a) = \arg\min_{d} E_{i'} \{ g(i_{k+1}, i') + \tilde{J}(i', \overline{r}) \mid i_{k+1}, a, d \}. \tag{16}$$

One can view this iteration as the approximate policy iteration described earlier that uses only a *single* iteration of $TD(1)$ to evaluate the current policy, rather than a complete policy evaluation. In this sense, the method is reminiscent of the value iteration method. Similarly, the variants that replace the policy μ with the policy $\overline{\mu}$ after multiple state transitions and corresponding updates of the new weight $\overline{r}$ are reminiscent of the modified policy iteration.

The convergence properties of the optimistic policy iteration method are quite complex and are not fully understood (see [2], Section 6.4 for an extensive discussion.) Optimistic policy iteration in conjunction with TD(λ) is reputed to be one of the most effective NDP methods, and has been used with success for solving some challenging problems, e.g. the backgammon work of Tesauro [13].

E. Comparison of the Methods

There are several important questions regarding the performance of the NDP methodology on a given type of problem. In particular, for the approximate policy iteration algorithms (regular and optimistic), it is interesting to:
(a) Know whether the methods will converge to some optimal cost approximation and policy, or whether they will oscillate.
(b) Provide estimates of the difference between the cost function of the final policy obtained from the algorithm and the optimal cost function. These estimates should involve the "power" of the architecture (a measure of richness and approximation capability of the class of functions that can be represented compactly with the given architecture). The estimates may also involve λ, if a TD(λ) algorithm is used.

Regarding question (a), computational experimentation has demonstrated that the regular approximate policy iteration algorithm need not converge to a policy. Typically, the method makes progress up to a point and then the iterates $\tilde{J}(\cdot, r_k)$ oscillate within a neighborhood of $\hat{J}(\cdot)$. For the optimistic policy iteration, examples and analysis indicate that the method exhibits the same type of oscillatory behavior as with approximate policy iteration. In particular, the sequence of policies that is generated is oscillatory in nature. However, the nature and amplitude of the oscillations may be different in the two methods and may depend on λ, so one may want to try both types of methods with different values of λ. A peculiar phenomenon here is that the sequence r_k produced by the optimistic policy iteration method typically converges to some $\hat{r}$, even though the generated sequence of policies may oscillate. This phenomenon is known as *chattering*, and is explained in Section 6.4 of [2]. Generally, the optimistic policy iteration seems to require substantially less computation time to reach a comparable stage of oscillatory behavior to the one of the regular version. On the other hand, it may be difficult to select a final policy without a full evaluation of the several policies involved in the chattering, and this may negate the faster convergence advantage of optimistic policy iteration (see the following discussion).

It should be noted here, that even though the NDP policy iteration methods need not converge to a policy, they can still be useful algorithms. Once the policies generated start oscillating, one can terminate the iterations and extract a policy with relatively good performance (as determined by simulation), out of the sequence of policies produced thus far by the method. This process of final policy selection, which we refer to as *screening*, is much more time-consuming for optimistic versions of policy iteration, because of the large number of policies that are generated during training, and also because the number of trajectories per policy that become available while training is limited. Typically, in order to reliably screen policies in optimistic policy iteration, one has to generate a large number of additional trajectories (after training has been completed), and use them for a more reliable evaluation of some of the more promising policies that have been generated while training.

Regarding question (b), there is a theoretical result that estimates the performance of the regular approximate policy iteration method. In particular, let us denote by $\mathcal{R}$ the *range* of the architecture, that is, the class of functions of the form $\tilde{J}(\cdot, r)$ as the parameter vector r ranges over all possible values. Let us assume that, for some $\epsilon > 0$, the architecture of the compact representation is such that the cost function $J_{\mu_k}(\cdot)$ of every generated policy μ_k can be approximated within ϵ by a function $\tilde{J}(\cdot, r_k)$ from the range of the architecture (in the maximum norm sense). Then a result of Bertsekas and Tsitsiklis for discounted problems with discount factor α ([1], p. 42, or [2], Section 6.2) shows that, in the limit of the iterations, the supremum of the differences $\tilde{J}(\cdot, r_k) - \hat{J}(\cdot)$ is bounded (in the maximum norm sense) by

$$\frac{2\alpha\epsilon}{(1-\alpha)^2}. \tag{17}$$

There is a qualitatively similar result for stochastic shortest path problems, which applies to our model of the TMD problem (see [1], Ch. 2, or [2]). Whether a similar result holds for the optimistic policy iteration is presently unknown, but considerable insight into the practical behavior of optimistic policy iteration has been developed. This insight suggests that the error bounds associated with nonoptimistic and optimistic variants are roughly comparable (see [2], Section 6.4).

V. Computational Experimentation

We experimented with several variations of the methods described in the preceding sections, and in this section we provide a comparative evaluation of some of these methods using 24 test cases. All of these cases involved three asset types, one missile type, and one interceptor type. Our experimentation was performed using the two approximation architectures discussed in Section 3, that is, the neural network/multilayer perceptron architecture, and the linear architecture that uses feature extraction. After experimentation with several different sets of features, we settled on the following four:

(1) Missile leakage, defined as $\max\{0, M - pI\}$, where M is the number of missiles, I is the number of interceptors, and p is the probability of destroying a missile that is intercepted by a single interceptor.
(2) One-by-one surviving asset value, defined as the expected total value of the surviving assets, assuming the defender defends against every missile launched (one at a time) by the attacker using a random attack policy.
(3) Number of assets.
(4) Number of interceptors.

We will present results obtained using three different types of methods:

(a) Approximate policy iteration using Monte-Carlo simulation [cf. Eqs. (7)-(9)] and the neural network architecture (referred to as API-NN).
(b) Approximate policy iteration using Monte-Carlo simulation [cf. Eqs. (7)-(9)] and the linear architecture (referred to as API-FAS).
(c) Optimistic policy iteration [cf. Eqs. (15)-(16)] and the neural network architecture (referred to as OPI-NN).

We have also performed some experimentation using TD(λ) [cf. Eq. (14)], in place of Monte-Carlo simulation, for different values of λ. The results and the performance (expected cost from the given initial state) of the final policy obtained were comparable to the corresponding results and performance obtained using Monte-Carlo simulation. In particular, the value of λ did not seem to have a significant qualitative effect on the computation.

Associated with each algorithm are several parameter settings, such as stepsize parameters, scaling factors, numbers of hidden units in the multilayer perceptron, etc. Because each algorithm has many parameter settings, each of which may take on a variety of values, the total number of possible setting combinations for each algorithm is quite large. In addition, we found during preliminary testing that algorithm performance is not a well-behaved function of the parameter settings. To find the best parameter settings for each of the methods in each of the 24 test cases would require an exhaustive search well exceeding the time and computational resources available. We therefore turned to a mixture of insight and preliminary experimentation to arrive at a combination of settings for each algorithm that would work robustly in all 24 test cases. In the case of API-NN, for example, there was little information on how stepsize would affect performance. Several combinations were tried until we arrived at stepsize parameters that yielded good results for most cases. Some of the parameter settings for each algorithm are listed below. When simulating for training purposes the initial states of the generated trajectories were chosen randomly from some set that was "centered" around a fixed nominal initial state.

A. Algorithms Tested

A.1 API-NN

The neural network has 5 inputs, 8 hidden sigmoidal units, and 1 output. During each test, a total of 50 policy iterations were performed. The first iteration used a heuristic defense policy (described below). All iterations involved 20 trajectories to assess policy performance and another 100 devoted toward generating sample data with which to train the neural net. Within each stage of each trajectory, five Monte-Carlo simulations per assignment possibility were performed to detect the optimal control. Regarding training methods, samples were placed in a buffer and cycled through randomly ten times during the first iteration. All remaining iterations set the number of cycles to five.

A.2 OPI-NN

The neural network has 5 inputs, 16 hidden sigmoidal units, and 1 output node. During each test, a total of 500 policy iterations were performed. In the first iteration, which employed a heuristic defense policy (described below), 20 trajectories were performed to assess the policy' s performance and another 100 were devoted toward generating sample data with which to train the neural network. The latter value was reduced to 10 during the rest of the iterations. Within each stage of each trajectory, five Monte-Carlo simulations per assignment possibility were performed to detect the optimal control. Regarding training methods, samples were placed in a buffer and cycled through randomly ten times during the first iteration. All remaining iterations set the number of cycles to five.

A.3 API-FAS

During each test, a total of 50 policy iterations were performed. The first iteration used a heuristic defense policy (described below). All iterations involved 20 trajectories to assess policy performance and another 100 devoted toward generating sample data with which to train the neural net. Within each stage of each trajectory, five Monte-Carlo simulations per assignment possibility were performed to detect the optimal control. Regarding training methods, samples were placed in a buffer and cycled through randomly ten times during the first iteration. All remaining iterations set the number of cycles to five.

B. Computational Results

We have compared the performance obtained using the NDP methodology with:

(a) The optimal performance, which was calculated using exact DP and the Gauss-Seidel method discussed in Section 2.

TABLE I
Test cases conducted on each algorithm examined.

	Initial Inventory		Pkill		Launchers	
Case	Int.	TBM	Int.	TBM	Int.	TBM
1	60	40	0.9	1.0	6	4
2	60	40	0.9	1.0	6	6
3	60	40	0.9	1.0	4	6
4	60	40	0.8	1.0	6	4
5	60	40	0.8	1.0	6	6
6	60	40	0.8	1.0	4	6
7	40	40	1.0	1.0	6	4
8	40	40	1.0	1.0	6	6
9	40	40	1.0	1.0	4	6
10	40	40	0.9	1.0	6	4
11	40	40	0.9	1.0	6	6
12	40	40	0.9	1.0	4	6
13	40	40	0.8	1.0	6	4
14	40	40	0.8	1.0	6	6
15	40	40	0.8	1.0	4	6
16	40	60	1.0	1.0	6	4
17	40	60	1.0	1.0	6	6
18	40	60	1.0	1.0	4	6
19	40	60	0.9	1.0	6	4
20	40	60	0.9	1.0	6	6
21	40	60	0.9	1.0	4	6
22	40	60	0.8	1.0	6	4
23	40	60	0.8	1.0	6	6
24	40	60	0.8	1.0	4	6

(b) A heuristic defense policy that operates as follows:
(1) Set a parameter called the maximum limit to infinity.
(2) Set the asset category of current interest to the one that has some assets remaining and has the highest value per asset.
(3) Assign one interceptor to each missile attacking the asset category of current interest subject to the physical constraints of the problem and the maximum limit.
(4) If it exists, set the asset category of current interest to the next most valuable one remaining, otherwise stop.
(5) Set the maximum limit to the surplus of interceptors over missiles minus the total number of assets in all asset categories more valuable than the one of current interest excluding the most valuable remaining asset category.
(6) Loop back to step 3.

We present results involving a collection of 24 problems. In all cases, three asset categories are present with values of 1, 2, and 3 and initial inventories of 10, 10, and 10. What differs among the cases is the capabilities of the attacker and defender as is listed in Table 1.

Table 2 gives the performance of the final policy obtained by the three NDP methods, the optimal policy, and the heuristic policy, starting from the initial state indicated in Table 1.

Figures 1-3 give more detailed results for test case 21. In particular, Fig. 1 shows the sequence of performances

TABLE II
Empirical performance, in terms of expected value of remaining assets.

Case	API NN	OPI NN	API FAS	Heur.	Opt.
1	49.45	50.44	49.65	52.00	56.70
2	49.83	50.38	49.92	51.93	55.47
3	44.50	44.51	44.78	46.30	47.92
4	38.96	39.97	40.14	44.28	52.36
5	39.86	40.63	40.33	44.08	50.20
6	35.95	35.81	35.90	38.83	41.88
7	60.00	60.00	60.00	48.88	60.00
8	60.00	60.00	60.00	48.40	60.00
9	53.68	53.59	53.68	45.28	53.62
10	48.45	49.53	51.02	42.05	52.00
11	48.69	49.44	50.68	41.59	52.00
12	44.36	44.56	44.53	38.75	47.13
13	39.26	39.50	41.40	34.96	44.00
14	39.58	39.82	41.55	34.89	44.00
15	35.55	35.78	35.85	31.89	44.00
16	28.43	28.67	27.57	30.00	30.00
17	28.35	29.15	27.62	30.00	30.00
18	28.73	28.43	27.15	19.39	29.86
19	16.54	16.68	12.66	18.00	18.01
20	16.96	16.89	13.02	18.07	18.01
21	16.60	16.51	13.82	10.64	17.97
22	0.09	4.91	3.35	6.85	6.97
23	4.27	4.92	3.64	6.89	6.97
24	5.49	5.19	4.02	3.58	6.96

(expected costs from the given initial state, as evaluated by averaging 20 cost samples) of the policies generated by API-NN. Figure 2 gives the analogous results for the policies generated by API-FAS. Figure 3 shows the sequence of performances of the policies generated by OPI-NN; however, the performance shown for each policy is very "noisy" because it has been evaluated by averaging very few cost samples. Generally, as mentioned earlier, the optimistic policy iteration method has the drawback of requiring a substantial post-training phase, whereby the policies obtained during training must be further evaluated using additional simulation, in order to delineate or "screen" the one(s) that are best in some sense. By contrast, nonoptimistic policy iteration requires a much less time consuming post-training phase, because the number of generated policies is relatively small.

The performance measurements shown for each policy in Figures 1-3 are very "noisy" because they represent the average of very few cost samples. Therefore, the true score of the top policy within a given run cannot be found simply by inspection. A process we call screening selectively resamples promising policies uncovered during a particular test run to get finer estimates of their true expected performance. Through screening, the uncertainty in estimating the best score in a run can be reduced to acceptable levels. Our screening procedure adopts a multi-level approach. At

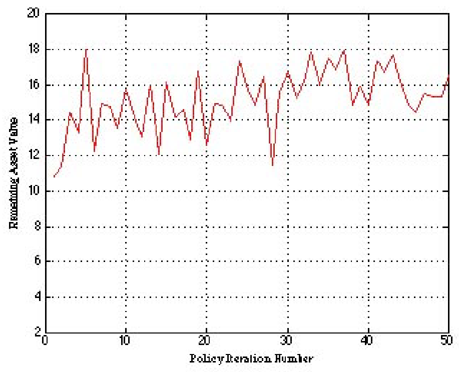

Fig. 1. Performance Sequence from API-NN Applied to Case 21.

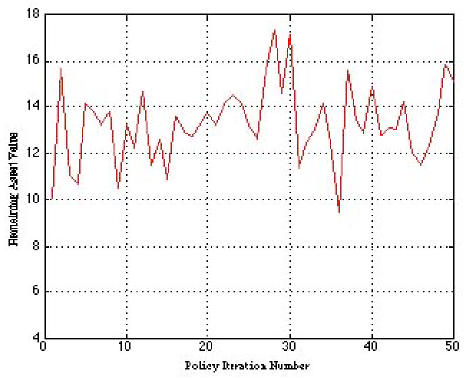

Fig. 2. Performance Sequence from API-FAS Applied to Case 21.

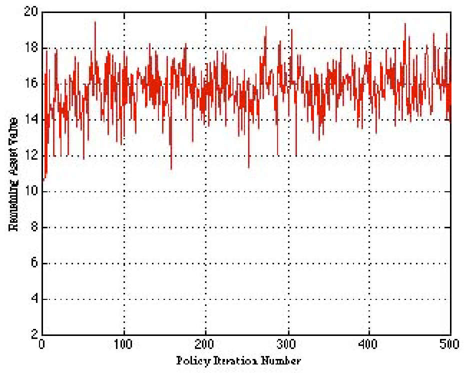

Fig. 3. Performance Sequence from OPI-NN Applied to Case 21.

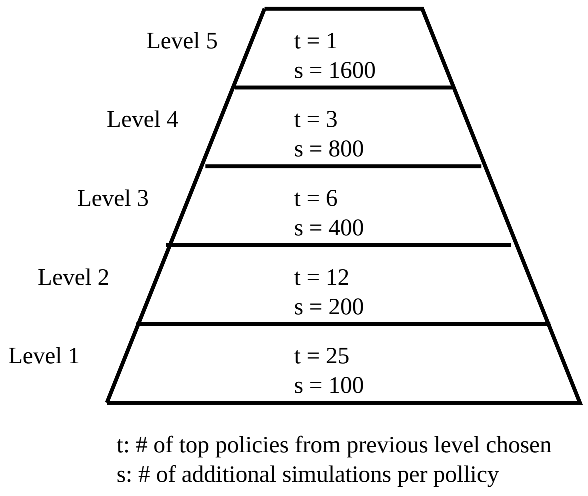

Fig. 4. Screening Parameters for API-NN and API-FAS.

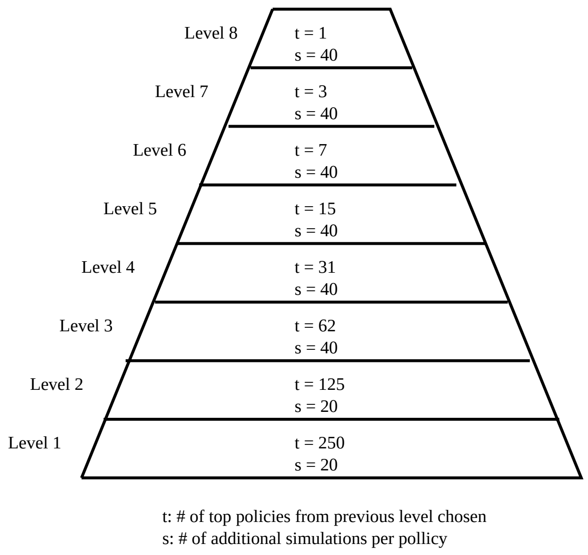

Fig. 5. Screening Parameters for OPI-NN.

each level, three actions occur. First, the top t performing policies from the previous level are chosen for further examination, where t is a user specified parameter. s sample trajectories are then generated from each of the t policies, where s is another user specified parameter. The samples scores from these additional simulations are combined with the older sample data to arrive at new sample score averages. In the next level, this new average is used to select the top policies from the current level' s group. The number of top policies examined (t) along with the number of simulation runs per policy (s) are listed below in Figures 4 and 5 for each level in the screening processes used on our test runs. Note that schedules for the API methods are different from the one for OPI-NN due to the smaller number of policies that need to be screened. While screening usually does not locate the policy with the best performance, it does lead to policies whose scores are quite close to the best. The technique effectively allows us to estimate the true best performance to within a tight margin of error.

C. Discussion

The objective of this research was to develop a new solution methodology for theater missile defence based on NDP. The case study of this paper is one aspect of the research where we have sought to understand NDP from the perspective of limiting performance. No real attempt was made to optimize our prototype code. Run-time comparisons across algorithms aren't particularly meaningful. (Generally, the NDP runs were set to take approximately between 10 minutes and 2 hours each, with more simulation being required for the nondeterministic cases. On the other hand, exact computation of the optimal policy could take as long as 36 hours.)

The 24 case problems in Table 1 were designed to cover a spectrum of interceptor allocation problems ranging from "overwhelmed defender" to "overwhelmed attacker," with varying degrees of interceptor effectiveness. The problems are sized so that exact solution by stochastic dynamic programming is feasible, providing a useful means of comparison for algorithm performance. The case problems are not strictly ranked in order of difficulty, either in terms of optimal expected value or algorithm performance, although generally the situation is more grim for the defender in higher case numbers (due to a diminishing ratio of interceptors to missiles). Case problem 21 is one that we identified as being a rich problem, offering a challenging resource allocation problem to the defender. It's also one where there is a substantial gap between the performance of the optimal policy and the handcrafted heuristic, leaving room for the NDP methods to make an improvement.

In some cases, the heuristic turned out to be optimal, or very nearly so. Naturally, the simulation-based methods don't stand much chance of beating the heuristic when this situation prevails. Of course, in more realistic, larger-scale TMD scenaria, it would impossible to determine in advance the gap between heuristic and optimal performance. On the other hand, it is clear from Table 2 that even when the heuristic/optimal performance gap is nontrivial, the heuristic policy sometimes beats the policies produced by NDP. Perhaps one reason for this is that the training parameters (stepsizes, numbers of simulation runs, etc.) for the experiment were tuned for case problem 21, which is qualitatively very different from the earlier cases where the heuristic tended to win. *These parameters were not optimized for each case individually.* Our experience is that NDP algorithm performance is highly dependent on the tuning of these parameters, even to the extent that it is possible to "unlearn" the positive aspects of the heuristic (which served as the initial policy for policy iteration).

VI. Conclusions

Our experimental results suggest several conclusions and point to some further questions and extensions of our methodology:

(a) None of the different architectures we tried seems to be uniformly superior for approximation of the type of cost functions that arise in our problem. It would appear therefore that the linear, feature-based architecture (FAS),

which is easier to train, holds an advantage for our problem. It is plausible that the performance of this architecture may be improved by introducing some more effective features.
(b) While optimistic and nonoptimistic policy iteration methods produced comparably performing policies, the time required for optimistic methods is substantially larger than for the nonoptimistic methods because of the time-consuming screening process. On the other hand, the training time for optimistic policy was generally smaller than for its nonoptimistic counterpart. Thus, it is plausible that with a more effective screening process, some significant improvement in computation time may be obtained.
(c) Our experience suggests that the NDP methodology scales well with the size of the problem. Thus, experimentation with more complex types of problems is worthwhile. As the size and complexity of the problem increases, it may be worth considering techniques for problem decomposition (see [2] for a discussion of such techniques). In particular, one may split the TMD problem into *opening*, *middlegame*, and *endgame* phases, corresponding to different stages of the battle. One may solve the endgame phase first, and then use this solution as a terminal condition for the middlegame stage. Similarly, after solving the middlegame, one can use this solution as a terminal condition for the opening stage.
(d) For large problems, the minimization in the right-hand side of Bellman's equation over all defense vectors [cf. Eq. (2)] may be very time consuming. It is thus worth considering approximate ways of doing this minimization. In particular, Section 6.1 of [2] describes ways to trade-off a reduced control space complexity with an increased state space complexity. These ideas appear to be well-suited for the TMD context.
(e) Finally, there are several effective parallelization possibilities in NDP, which are relevant to our context. These include the generation of trajectories by simulation to use either for training architectures or for screening of policies, and the parallelization of various training algorithms. These possibilities may hold the key to addressing very large problems.

References

[1] D. P. Bertsekas, *Dynamic Programming and Optimal Control: Vol. 2*, Athena Scientific, Belmont, MA, 1995.

[2] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, Belmont, MA, 1996.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998.

[4] A. G. Barto, S. J. Bradtke, and S. P. Singh, "Learning to Act using Real-Time Dynamic Programming," *J. Artificial Intelligence*, 1995.

[5] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[6] D. P. Bertsekas and J. N. Tsitsiklis, "Analysis of Stochastic Shortest Path Problems," *Mathematics of Operations Research*, vol. 16, no. 3, pp. 580–595, 1991.

[7] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, John Wiley & Sons, Inc., New York, 1994.

[8] R. J. Williams and L. C. Baird, "Analysis of Some Incremental Variants of Policy Iteration: First Steps Toward Understanding Actor-Critic Learning Systems," Report NU-CCS-93-11, College of Computer Science, Northeastern University, Boston, MA, 1993.

[9] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3, pp. 9–44, 1994.

[10] D. P. Bertsekas, *Nonlinear Programming*, Athena Scientific, Belmont, MA, 1995.

[11] J. N. Tsitsiklis and B. V. Roy, "Analysis of temporal-difference learning with function approximation," Tech. Rep. LIDS-P-2322, Lab. for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA, 1996.

[12] D. P. Bertsekas, "A Counterexample to Temporal Differences Learning," *Neural Computation*, vol. 7, pp. 270–279, 1995.

[13] G. J. Tesauro, "Practical Issues in Temporal Differences Learning," *Machine Learning*, vol. 8, pp. 257–277, 1992.

Dimitri P. Bertsekas Blah, blah, ...

Mark L. Homer Blah, blah, ...

David A. Logan Blah, blah, ...

Stephen D. Patek Blah, blah, ...

Nils R. Sandell Blah, blah, ...