Deep Reinforcement Learning for Weapon-Target Assignment

Kevin Huang
Computing + Mathematical Sciences
California Institute of Technology
khuang@caltech.edu

Yongkyun Lee
Computing + Mathematical Sciences
California Institute of Technology
ylee2@caltech.edu

Alexander Pan
Computing + Mathematical Sciences
California Institute of Technology
aypan@caltech.edu

Emily Park
Computing + Mathematical Sciences
California Institute of Technology
eppark@caltech.edu

Megan Tjandrasuwita
Computing + Mathematical Sciences
California Institute of Technology
mtjandra@caltech.edu

Abstract

Weapon-target assignment (WTA) is a NP-complete problem consisting of assigning weapons of m types to n targets, for which several heuristic methods have been developed. In this paper, we propose formulating WTA as a reinforcement learning problem. We embed n, m, and probabilities p_{ij} of weapon i killing target j in the environment and vary target values over different problem instances. We attempt two methods of constructing a solution that maximizes over a learned reward function. The first is local update Q-learning, which modifies a complete weapon-target assignment by changing a single weapon's assignment to a different target. The second is greedy Q-learning, which incrementally builds up a solution by assigning a currently unassigned weapon to a target. In testing both approaches, we only consider cases where there is one weapon per type and $m = n$. We found that substituting the mutation step of a traditional genetic algorithm for a local update Deep Q-Network (DQN) converges to an optimal solution with fewer iterations and more consistently compared to random mutation. Training a greedy DQN is potentially better suited to efficiently finding viable solutions to large WTA problems.

1 Introduction

The weapon-target assignment (WTA) problem is a combinatorial optimization problem where we seek to maximize the total expected damage to a set of enemy targets using a fixed number of weapons. The problem is a generalization of problems that occur in combat operations.

Aside from the direct connection to military planning, there are other applications of the WTA problem. For example, a search-and-rescue operation can be modeled as a WTA, where the weapons are various resources available (aircraft, boats, divers), the targets are locations to search, and the target values are the probabilities that survivors will be found.

Preprint. Under review.

1.1 Integer Program Formulation

Suppose there are n targets numbered $1, \ldots, n$ and m weapon types numbered $1, \ldots, m$. Each target j has value V_j and each weapon type i has W_i available weapons of that type. For a given target j, p_{ij} is the probability of destroying j using a weapon of type i. If we assign x_{ij} weapons of type i to target j, then j has a $(1 - p_{ij})^{x_{ij}}$ chance of survival. Our goal is to determine a weapon assignment minimizing the total expected survival value of the targets.

We may express the WTA problem as the following nonlinear integer program:

$$\begin{aligned}
\text{minimize:} \quad & \sum_{j=1}^{n} V_j \left(\prod_{i=1}^{m} (1 - p_{ij})^{x_{ij}} \right) \\
\text{subject to:} \quad & \sum_{j=1}^{n} x_{ij} \leq W_i, && i = 1, \ldots, m \\
& x_{ij} \geq 0 \text{ and integer}, && i = 1, \ldots, m, \quad j = 1, \ldots, n.
\end{aligned}$$

The WTA problem is NP-complete [1], so it is likely difficult to find exact solutions practical for large instances of the problem.

There also exists a *dynamic* WTA problem where the assignment is done over several rounds. After each round, the attacker is allowed to examine which targets were destroyed and update their assignment. The program described in 1.1 is the *static* WTA problem. The dynamic WTA problem poses additional challenges, as it cannot be easily decomposed into static WTA sub-problems because of the non-linearity [2].

Developing algorithms for WTA is important not only because of its real-world applications, but because of its connections to combinatorial optimization. As mentioned, several known algorithms for WTA are variants of branch and bound [3, 4], so improving WTA algorithms will improve branch and bound algorithms, especially for nonlinear integer programs.

Even though exact algorithms are known, oftentimes one can get by with an approximate solution. The WTA is a combinatorial optimization problem, so it may be viewed as a series of discrete choices that combine to form a solution. Thus heuristic algorithms, which are used to inform the choices, are a good candidate for the WTA. In this report, we focus on learning a heuristic and compare our models to vanilla genetic algorithms [5].

1.2 Technical challenges for the WTA

Two of the key technical challenges of the WTA involve the high dimensionality of the solution space and its constraints on the solution space. Any choice made by the model when creating a solution involves choosing a target, a weapon type, and the number of weapons of that particular type to assign. Therefore, even after the weapon type and target has been selected, there still may be thousands of choices. The complexity of the solution space increases training time and decreases the probability of the model converging. Moreover, choosing the number of weapons to assign is dependent on the number of weapons previously assigned. Applying these constraints in a manner that allows the model to generalize to future constraints is a nontrivial task [6].

To mitigate some of the challenges described, we stipulate that the number of available weapons of each type is exactly 1 and the number of weapon types is equal to the number of targets. That is,

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \text{and} \quad m = n.$$

In all of our work, we choose to solve this simplified version of the static WTA problem. Although our work cannot be directly applied to realistic WTA problems, it serves as a proof of concept and suggests that a learned heuristic algorithm is a viable approach to the WTA problem. We propose two new heuristic algorithms for the static WTA problem that learn a heuristic across various problem instances using reinforcement learning.

1.3 Overview of report

Section 2 gives relevant background for this report and further describes some of the previous work for the WTA problem. In Section 3, we cover an algorithm that uses Q-learning to learn a local update policy for a genetic algorithm (which replaces the vanilla genetic algorithm's random update policy). In Section 4, we cover an algorithm that uses Q-learning to learn a greedy policy for weapon assignment. Section 5 details our experimental methods and highlights some of our collected data. Finally, Section 6 offers analysis of the results and provides directions for future work.

2 Background and Previous Work

We highlight two major directions of previous algorithms for the WTA in Sections 2.1 and 2.2. And we cover the necessary background for Q-learning in Section 2.3.

2.1 Branch and Bound

The branch and bound algorithm is widely used to find exact solutions to discrete combinatorial optimizations. The algorithm consists of branching, lower bounding, and searching strategies.

Branching splits the problem space into smaller spaces, and metrics such as maximal-marginal return are used to find the weapon-target assignment that leads to greatest improvement. The algorithm then branches from the new optimal solution.

Lower bounding is used to prune the search space; it can be implemented if the problem is formulated as a network flow or a linear program [3]. For moderately-sized problem instances, there are a variety of lower bounding schemes that can be used. For instance, the LP-based lower-bounding scheme treats the problem as a linear program, the MIP-based treats it as an integer program, the minimum-cost-flow-based treats it as a minimum-survival/maximum-expected-damage problem, and the maximum-marginal-return-based treats it as a knapsack problem. Among all the above schemes, the MIP-based lower-bounding is able to solve larger problems and give the most consistent results [3].

Search strategies fall into two categories: breadth-first and depth-first. Algorithms that implemented a breadth-first search saw better results for smaller size problems, but those with depth-first search had superior performance for the larger ones. Although both of these strategies yield good results in reasonable amounts of time for moderately-sized problem instances, branch and bound algorithms that solve large-scale instances efficiently and effectively have yet to be developed. [3].

2.2 Heuristic approaches and genetic algorithms

Although branch and bound algorithms output exact solutions, they scale inefficiently when compared to heuristic algorithms. To solve harder problem instances, several heuristic algorithms have been suggested, such as neighborhood search. It first estimates a feasible solution of WTA and iterates over the solution's neighborhood until it reaches a locally optimal solution. The final solution depends on the initial feasible solution and the definition of neighborhood of a solution. One implementation of neighborhood search is to generate an improvement graph from an initial solution and identify a profitable multi-exchange algorithm [3].

Several of the learned heuristic algorithms draw inspiration from biology. In particular, algorithms modeled after genetic programming with eugenics [5], discrete particle swarm optimization [7], and immunity-based ant colony optimization [8] have all been suggested. The hope is that natural processes, tuned by evolution, are well-suited to efficiently determining the optimal assignment.

In particular, we choose to focus on improving a genetic algorithm with greedy eugenics. "Eugenics" refers to the process of gene reformation by choosing candidates of solutions nearby the incumbent solution, i.e., a local update procedure.

For more detail, let $EDV(j)$ denote the expected damage value of the jth target. A synonymous expression of the cost function to minimize for the WTA problem is:

$$\text{minimize: } \sum_{j=1}^{n} EDV(j)^* \left[1 - \left(\prod_{i=1}^{m} (1 - p_{ij})^{x_{ij}}\right)\right]$$

which describes the overall damage to all targets. We seek to assign targets to weapons with the highest $EDV(j) * p_{ij}$, where p_{ij} is defined in 1.1. This motivates the following greedy update procedure, which was first described in [5].

1. Consider the current chromosome. The first phase involves ordering targets by $EDV_{new}(j) * K_{ij}$ in descending order, where $EDV_{new}(i) \leftarrow EDV(j)$ initially.

2. Weapon j corresponding to the highest $EDV_{new}(j) * K_{ij}$ is then assigned to the ith target. We must then update $EDV_{new}(i) \leftarrow EDV_{new}(j) * (1 - K_{ij})$. The assignment process continues until all current targets have been assigned a weapon.

3. In the second phase, unassigned weapons greedily select targets with the highest $EDV_{new}(j) * K_{ij}$.

2.3 Q-learning

Our goal was to design an data-driven algorithm that could learn, therefore scaling and generalizing to larger problem instances.

Vanilla genetic algorithms lack adaptability, as their human-designed update heuristics cannot learn from experience. Additionally, these heuristics oftentimes are computationally expensive and take many iterations to converge.

To address these issues, we use reinforcement learning to learn an update heuristic for the genetic algorithm covered in Section 2.2. Using Q-learning, a type of reinforcement learning first described in [9], we train a neural network that predicts a distribution of improved neighboring solutions rather than naively or randomly choosing a neighboring solution. We also train a neural network that uses Q-learning to learn a greedy policy for building a weapon-target assignment up from partial solutions.

As with all reinforcement learning models, Q-learning guides agents that attempt to select actions in an environment to maximize their total reward. The model learns 'quality' value to an action-state pair. Because Q-learning is an off-policy learner, the model does not base its Q-value estimates on the agent's current action, and Q-values are calculated assuming a greedy choice of the next action.

Deep Q-learning uses a neural network to represent the reward function. Additionally, given a state, the neural network outputs all possible actions and their corresponding Q-values. In contrast, Q-learning typically outputs the Q-value of a single state-action pair. The neural network has the advantage of learning over multiple training instances [9].

To aid the deep Q-learning's performance and model convergence, we use several standard modifications: double DQN, experience replay, and dueling networks [10].

A double DQN, instead of having one target network, has two target networks. One target network is 'primary', i.e., it is updated every iteration and used to calculate the rewards. The other target network is 'fixed', i.e., it is the network that is used to calculate target values. The fixed network is updated every T iterations with the parameters from the primary network. From now on, DQN will refer to a double DQN.

Experience replay refers to a buffer that stores action-state pairs along with their rewards. By using an experience replay buffer, the model is able to learn from experiences (state-action pairs) that occur in a non-sequential order. This discourages the model from overfitting to the order that the experiences were acquired, which is usually irrelevant.

Finally, dueling networks involve training two separate models to measure the Q-value of a state-action pair [11]. The vanilla DQN model uses a single network to compute the Q-value of the state-action pair. However, a Q-value may be thought of as the sum of the current value of the state and the marginal reward (advantage) received by a particular action. By using separate networks to measure the value and advantage for each state-action pair, the model will be able to better discern between state-action pairs, leading to faster training and higher performance.

3 Local Update Q-learning

3.1 Reinforcement learning formulation

Our first approach assumes that the number of weapons m, the number of targets n, and the probabilities p_{ij} are fixed constants and not included in a problem instance. Thus, in our case, a single problem instance consists of only the target values, V_j; we only learn across a distribution of target values, and not across a distribution of problem sizes or probabilities. We later discuss how our approach can easily extend to such other cases.

We define an *assignment* as a vector $z \in \mathbb{R}^m$ such that $z_i = j$ if weapon i is assigned to target j. The corresponding state of the problem is then $s = (z, V)$: the current assignment and the value of the targets. Thus, the expected value of state s is

$$\mathbb{E}(s = (z, V)) = \sum_{j=1}^{n} V_j \left(1 - \prod_{i=1}^{m} (1 - p_{ij})^{\mathbb{1}[z_i = j]}\right) \tag{1}$$

We define an action a as a pair (i, j), which corresponds to changing the assignment of weapon i to target j. Thus, if we apply action a to state $s_t = (z^t, V)$, the new state is $s_{t+1} := (z^{t+1}, V)$ with $z_i^{t+1} = j$. An action is best interpreted as "swapping" the target that a weapon is assigned to in a given assignment.

We define our reward function $r(s, a)$ as the change in the expected value of state s when applying action a; that is,

$$r(s_t, a) = \mathbb{E}(s_{t+1}(z^{t+1}, V)) - \mathbb{E}(s_t(z^t, V)) \tag{2}$$

where $a = (i, j)$ is chosen, while we perform our local update in our genetic algorithm, as follows: given an assignment z, apply the action $\underset{a}{\operatorname{argmax}}\, Q((z, V), a)$.

The network then learns a function $Q(s, a)$, where $Q(s, a)$ represents the improvement of a state s when applying an action a.

3.2 Network description and learning algorithm

We use a deep Q-network (DQN) to approximate the function Q. Given an input state (z, V), the network outputs the approximated value of $Q((z, V), a)$ for every action a in the action space. The input state assignment z is first embedded so that each target is represented as a $\frac{n}{2}$ dimensional vector, and then flattened and concatenated with V to form the input to the rest of the network. We train two different versions of this network. The first is a standard DQN that consists of two fully connected layers followed by ReLU activations and dropout. The second is a dueling DQN, that splits the input into a "value" and "advantage" branch, which each consist of a fully connected layer followed by ReLU activation. The two branches are then combined with the output equal to $value + (advantage - advantage.mean())$.

The networks are trained using the standard Q-learning algorithm. The network parameters are optimized to minimize the loss function $[(\gamma \max_a Q'(s_{t+1}, a) + r((z^t, V), a)) - Q((s_t, V), a)]^2$, where Q' is taken from a separate target network, which mirrors the main network, except its parameters are only updated every T iterations. The network is updated by copying the parameters from the main network. Training examples $(s_t, a, s_{t+1}, r(z^t, a))$ are taken from an experience replay memory.

4 Greedy Q-learning

4.1 Reinforcement learning formulation

For our second approach, we build up partial solutions based on a greedy heuristic Q, learned through Q-learning. We have the same assumptions as for the local update method, where only the values V are included in the problem instance and other parameters are considered fixed constants.

We define a partial solution y^t as a vector $y^t \in \mathbb{R}^t$, where $y_i^t = j$ if weapon i is assigned to target j. y^t represents the assignment for only t weapons, where $0 \leq t \leq n$. A state is then $s_t = (y^t, V)$. An action a consists of (i, j), which corresponds to choosing weapon i as the $t + 1$'th weapon, and assigning it to target j. Thus, when applying $a = (i, j)$ to state s_t, the next state would be $s_{t+1} := (y^{t+1}, V)$, where y^{t+1} has $y_i^{t+1} = j$, and the first t weapon assignments of y^{t+1} are the same as y^t.

We build up a solution using a greedy policy as follows: given a partial assignment y^t, we choose a over $\underset{a}{\operatorname{argmax}}\, Q((y^t, V), a)$. Finally Q is approximated using the reward function r, as defined in (2), except choosing a as just described.

4.2 Network description and learning algorithm

The build-up DQN approximates Q given an input (y^t, V), where each $y_i^t = j$ is one-hot encoded, i.e. represented by a vector of length n of all zeroes, except for the index j element, which is 1. The one-hot encoding is then flattened and concatenated with V to form the complete input to the network. For convenience, we also append to the input a masking vector, $mask$, of length m, where the element at index i is 1 if weapon i has already been assigned in the partial solution and 0 if not. We transpose $mask$ and repeat it such that it has the same length as the output assignment ($output$), $m \times n$, after being flattened. After multiplying the mask with a large integer value, we return $output - mask$. We thus set the probability of assigning a weapon that has already been assigned to a large negative value, effectively preventing invalid assignments from occurring. The network architecture is the same as that of the standard DQN for local updating, consisting of two fully connected layers followed by ReLU activations and dropout. The network is then trained with the standard Q-learning algorithm as described in Section 3.

5 Experiments and Results[1]

We constructed 5 problem instances to test our results on, WTA1-WTA5, with sizes 5, 10, 20, 30, and 40 for examples 1-5 respectively. The size refers to both the number of targets n and the number of weapons m; thus, our experiments have that $n = m$. The probabilities and values were randomly generated from a range of 0-100, uniformly.

5.1 Genetic Algorithm with Random, DQN, and Dueling-DQN Mutation

For the local update networks, training instances were randomly generated, with target values taken from a uniform distribution between 0 and 100. The target values were reset at the beginning of each episode. Actions were selected with an episilon greedy strategy, with a start rate of 0.9 and a decay rate of 200. The discount factor γ was set to 0.999. The number of iterations T before the target network is updated was 500. The RMSprop optimizer was used. A batch size of 128 was used. Training was done for 100 episodes with 500 iterations per episode.

We use the genetic algorithm described by [5] to solve the WTA examples. We compare the effects of using our learned local update policy as a part of the mutation step, compared to random mutation. We compare the reward at each iteration for random mutation, DQN mutation, and Dueling-DQN mutation. Note that reward in this case refers to the reward of the genetic algorithm, which is defined as the expected value of a particular weapon assignment (given by equation (1)). It is not the reward function r used in Q-learning.

In Figure 1, it can be observed that the reward increases fastest per iteration for DQN mutation, followed by Dueling-DQN and random mutation in order. Yet, as the iteration number increases, the reward of random mutation overtakes the reward of DQN and Dueling-DQN mutation. Similar results are observed for different runs and for different problem instances (WTA1 to WTA5). Thus, although GA with DQN mutation reaches sub-optimal rewards in less number of iterations than GA with random mutation, for optimal rewards, the DQN mutation approach takes more iterations and has a lower success rate.

[1] Our code may be found at: `https://github.com/KevinHuang8/CS159-WTA`

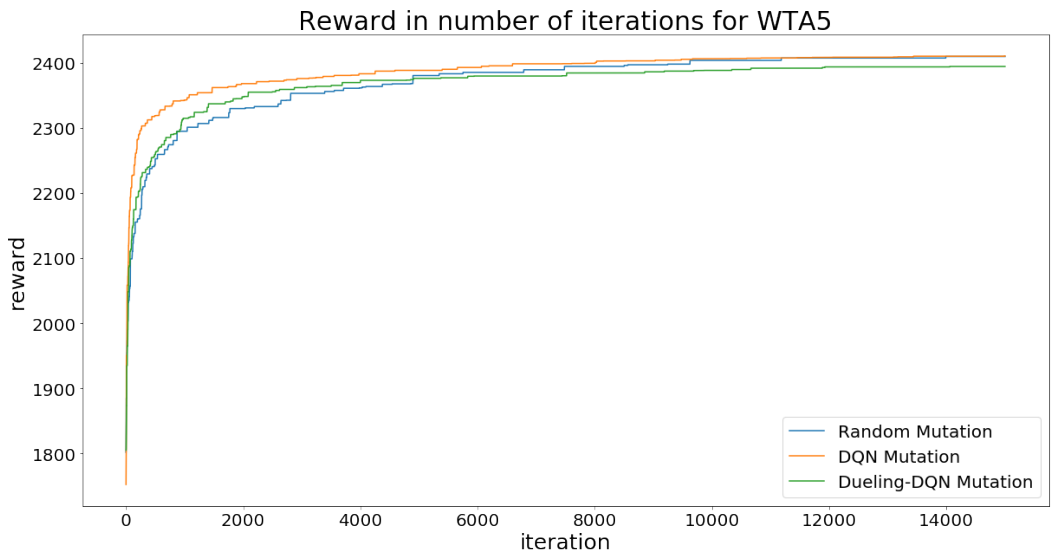


Figure 1: Reward per iteration for WTA5 with genetic algorithm using random mutation, DQN mutation, and Dueling-DQN mutation

We conducted further experiments by obtaining the reward for the GA with three different mutation methods after sufficient iterations. The maximum number of iterations were set to 1000, 3000, 5000, 10000, and 15000 for WTA1, WTA2, WTA3, WTA4, and WTA5 respectively. The number of mutations and crossovers in each iteration were both set to 1, the size of population was set to 15 times the number of weapons, and the number of offsprings in crossover was set to 10 times the number of weapons. According to preliminary experiments, the effect of the population size and the number of offsprings on the final reward was negligible.

Example	Random Mutation	DQN Mutation	Dueling-DQN Mutation
1	325.51 (9.023)	326.61 (4.725)	324.67 (7.212)
2	616.73 (17.377)	593.18 (17.585)	588.57 (24.399)
3	924.47 (58.729)	913.26 (28.217)	907.78 (36.415)
4	1806.07 (9.368)	1746.86 (74.093)	1728.08 (68.477)
5	2407.71 (5.542)	2369.04 (23.914)	2361.94 (9.209)

Table 1: Reward (standard deviation) for different methods of mutation. For WTA1, WTA2, and WTA3, the results were obtained for 15 runs, and for WTA4 and WTA5, the results were obtained with 8 runs.

Table 1 demonstrates that when the maximum number iterations (i.e. the number of iterations for the program to halt) is sufficiently large, random mutation has the highest reward, followed by DQN mutation, and Dueling-DQN mutation. As observed in Figure 1, random mutation GA achieves the highest reward among these three mutations, but it takes the longest time whereas the DQN-based mutation GA fails to exceed suboptimal results even after subsequent iterations. One possible cause for this outcome is that DQN and Dueling-DQN may end up in a local minimum since the mutation is decided repeatedly from the same neural network.

5.2 DQN and Dueling-DQN Mutation with Randomness

To address the problem of reaching local minimum, random mutation and DQN or Dueling-DQN mutation were mixed to add randomness and possibly improve the rewards. For each configurations, experiments were run 8 times. Other settings were the same as the previous experiments.

In Table 2 and Table 3, random mutation 0% implies that only DQN or Dueling-DQN mutation was used, and random mutation 100% implies that only random mutation was used. Also note that

Random Mutation Ratio	DQN Mutation	Dueling-DQN Mutation
0%	947.93 (4.575)	953.45 (1.444)
10%	945.51 (4.191)	938.32 (27.143)
30%	945.74 (5.207)	936.91 (29.640)
50%	948.78 (4.263)	944.25 (27.230)
70%	947.51 (5.649)	952.94 (1.506)
100%	948.63 (4.947)	948.63 (4.947)

Table 2: Reward (standard deviation) when DQN and Dueling-DQN mutation is mixed with random mutation for WTA3.

Random Mutation Ratio	DQN Mutation	Dueling-DQN Mutation
0%	1785.28 (5.971)	1697.71 (57.234)
10%	1788.66 (4.549)	1720.38 (71.329)
30%	1783.15 (7.701)	1696.41 (54.041)
50%	1787.32 (6.681)	1674.36 (0.038)
70%	1788.93 (6.352)	1674.35 (0.000)
100%	1810.64 (6.574)	1810.64 (6.574))

Table 3: Reward (standard deviation) when DQN and Dueling-DQN mutation is mixed with random mutation for WTA4.

because of the small number of runs due to the limitations in time, the standard deviation may be vary widely.

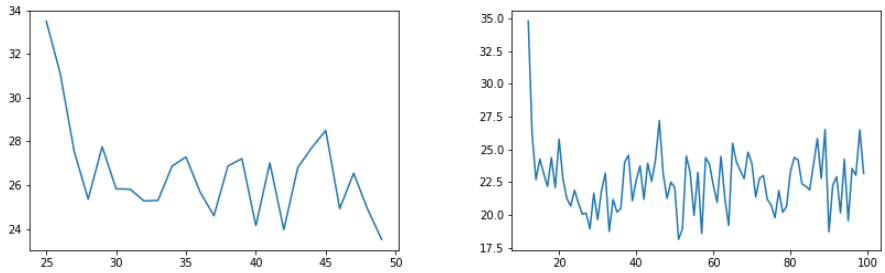
Mixing DQN-based mutation with random mutation has a marginal impact, but it is not significant considering the standard deviation of the experiment. The fact that the random mutation demonstrates the best performance in a sufficiently large number of iterations demonstrates that in long term, random mutation performs better than trained mutation.

5.3 Greedy build-up of partial solutions

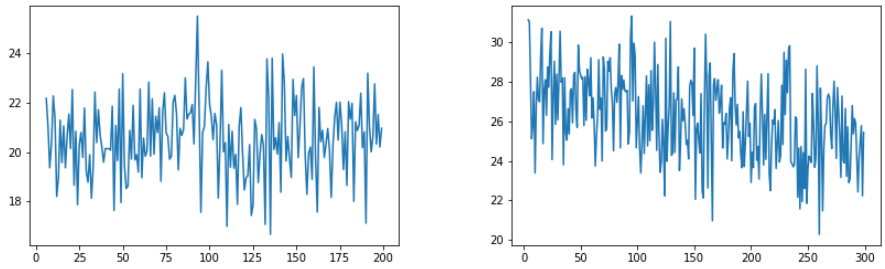
The build-up DQN has the same hyperparameters as the local update networks, except that γ was set to 1. Using the RMSprop optimizer, the DQN was trained for a number of episodes that scaled up with the number of targets of the problem instance (number of episodes = $10 \times$ number of targets, with the exception of example 12, which was run using 200 episodes). Each episode involves drawing a problem instance from a distribution, which we formulate as sampling each target value from a uniform distribution with a lower and upper bound equivalent to the minimum and maximum target value of a test example. After training, the build-up DQN constructs a full assignment, defined as all weapons being assigned to a target, for the test instance.

For each episode, we began with a solution with no weapon-target assignments, and built up on the partial solution at each step. The smooth L1 loss with respect to the number of episodes run are shown in Figure 2. Training the build-up DQN on examples 1, 2, and 4 result in an overall downward trend as the number of episodes increases, whereas the level of loss in examples 3 and 5 seems to remain constant when factoring out the variability between individual episodes. Notably, in examples 1 and 2, much of the decrease in loss occurs within the first 25-30 episodes, as training for more episodes past this number does not decrease the overall level of loss.

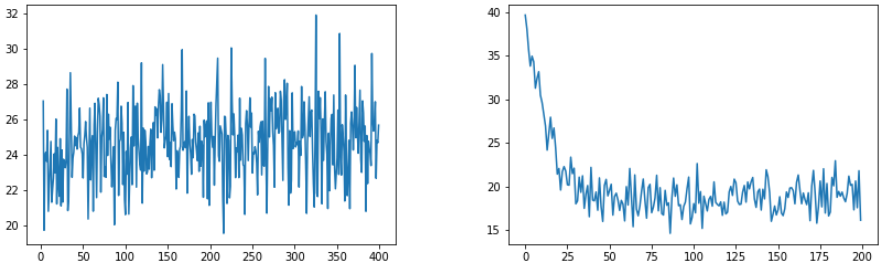
The rewards in Table 4 are calculated using the objective function defined in the introduction. The instance sizes of examples 6-11 are 50 to 100, increasing in increments of 10, and example 12 is 200.



Example 1 **Example 2**



Example 3 **Example 4**



Example 5 **Example 12**

Figure 2: Loss vs. Number of episodes for the build-up DQN models trained for examples 1-5 and for example 12.

Example	Build-up DQN
1	271.6
2	498.6
3	703.0
4	1522.8
5	1741.8
6	2258.7
7	2712.9
8	3281.5
9	3066.0
10	4042.1
11	4640.0
12	8129.9

Table 4: Rewards for each WTA example of weapon-target assignments generated by the build-up DQN.

In the case of the build-up DQN, alternative models were trained using 50 episodes regardless of instance size. These models produced assignments with better rewards for examples 2 (551.9), 3 (726.3), 5 (1745.2), 8 (3429.0), and 11 (4691.8). The only marked improvement that training with a greater number of episodes offered is for example 12, as the model with 50 episodes produced an assignment with a reward of 7452.6. This potentially suggests that training a model for the build-up style of producing an assignment may be preferred for greater instance sizes, those past 100 targets and weapons.

6 Discussion and Conclusion

One explanation for the non-decreasing loss while training the build-up DQN is the relative lack of exploration in later episodes. ε, or probability of choosing a random sample instead of using the policy neural network, exponentially decays from 0.9 to 0.05 with respect to the total number of training steps, where a step refers to assigning one additional weapon to the current solution. While ε is not allowed to decay below a minimum value of 0.05, the factor of exponential decay, $\frac{1}{200}$, may limit the amount of information learned from later episodes.

Another potential cause is that unlike the swapping DQN, the build-up DQN does not occasionally update the target network with the weights and biases of the policy network. While this may be acceptable when training with a fewer number of episodes, the performance of the trained network may suffer because of the Q-function approximation becoming outdated in later stages of training. Implementing the infrequent updating of the target network currently causes loss to diverge in subsequent episodes; however, there is a possibility that using n-step Q-learning, which involves waiting n-steps before optimizing the target network, stabilizes the updating.

Future work could involve generalizing both the GA with DQN and the build-up DQN to WTA problems where the number of weapons $\neq$ number of targets. Furthermore, we currently train DQN's over distribution of problem instances that have variable target values, but fixed probabilities p_{ij} of weapon type i killing target j. Sampling problems with varying p_{ij} would significantly increase the difficulty of training a model that generalizes to the problem instance distribution, which is much broader. A method that potentially addresses the issue is finding a better embedding of the inputs to the DQN, of both the partial solution and the problem instance. If such a concise embedding were found, the number of episodes required for loss to converge may decrease to the point that deep Q-learning still produces high-reward solutions. Furthermore, other reinforcement learning techniques such as actor-critic can be applied, which could improve both performance and speed for larger problem instances and thus larger action spaces.

A direct next step from our work would involve testing how much a DQN trained over a specific problem distribution, defined by n, m, and potentially p_{ij}, can help with tackling a different distribution, where n and m are not exactly the same. For instance, if we have a model that generates assignments for $n, m = 200$, to what extent can we reuse its weights and biases in models for $n, m = 200 \pm 10$? Such work would increase the practicality of our proof of concept, as the computational overhead of training over a significant number of episodes to achieve better generalization would be reduced to a one-time cost over a reasonable deviation from the original parameters.

References

[1] Stuart P Lloyd and Hans S Witsenhausen. Weapons allocation is np-complete. In *1986 Summer Computer Simulation Conference*, pages 1054–1058, 1986.

[2] Robert A. Murphey. *An Approximate Algorithm For A Weapon Target Assignment Stochastic Program*, pages 406–421. Springer US, Boston, MA, 2000.

[3] Ravindra K. Ahuja, Arvind Kumar, Krishna C. Jha, and James B. Orlin. Exact and heuristic algorithms for the weapon-target assignment problem. *Operations Research*, 55(6):1136–1146, 2007.

[4] Jay M Rosenberger, Hee S Hwang, Ratna P Pallerla, Adnan Yucel, Ron L Wilson, and Ed G Brungardt. The generalized weapon target assignment problem. Technical report, TEXAS UNIV AT ARLINGTON, 2005.

[5] Zne-Jung Lee, Shun-Feng Su, and Chou-Yuan Lee. Efficiently solving general weapon-target assignment problem by genetic algorithms with greedy eugenics. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 33(1):113–121, 2003.

[6] Brandon Amos and J. Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 136–145. JMLR.org, 2017.

[7] Xiangping Zeng, Yunlong Zhu, Lin Nan, Kunyuan Hu, Ben Niu, and Xiaoxian He. Solving weapon-target assignment problem using discrete particle swarm optimization. In *2006 6th World Congress on Intelligent Control and Automation*, volume 1, pages 3562–3565, 2006.

[8] Zne-Jung Lee, Chou-Yuan Lee, and Shun-Feng Su. An immunity-based ant colony optimization algorithm for solving weapon–target assignment problem. *Applied Soft Computing*, 2(1):39 – 47, 2002.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015.

[10] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[11] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR.