UAV Collision Avoidance Policy Optimization with Deep Reinforcement Learning

Blake Wulfe
Department of Computer Science
Stanford University
Stanford, CA 94305
wulfebw@stanford.edu

Abstract

The safe operation of unmanned aerial vehicles demands effective collision avoidance strategies. One method for solving the collision avoidance problem is to formulate it as a Markov decision process and solve for the action to take in any situation so as to maximize safety and efficiency. Solving for such a policy using dynamic programming can be time-consuming and requires discretizing the continuous state space, thereby potentially reducing safety. An alternative solution method is to use reinforcement learning with function approximation to learn a parameterized value function that maps directly from the continuous state space to state-action values, from which an approximately-optimal policy can be derived. We demonstrate that one such method, the Deep Q-Network [12], produces safer and more efficient policies than does value iteration, and does so in less time. We further investigate and present conclusions regarding a number of extensions to the baseline deep reinforcement learning model, including efficient sampling of the state space to speed learning, alternative solution methods such as Double DQN and Asynchronous Advantage Actor-Critic, and transfer learning of different reward structures.

Introduction

As UAVs become more widespread in use, they must be equipped with collision avoidance systems in order to maintain airspace safety. The Traffic Collision Avoidance System (TCAS) and its long-term replacement ACAS X play a central role in ensuring the safety of manned aircraft, and a number of solutions are being developed for the unmanned case [10]. One solution in development, ACAS Xu, takes a similar approach to ACAS X, and derives a collision avoidance strategy by formulating the problem as a Markov decision process (MDP) and solving for a policy using value iteration [9]. One major advantage of this approach is that researchers design the reward structure of the system rather than the full behavior of the policies, thereby producing superior collision avoidance strategies with reduced effort. As a result, the usefulness of these policies depends upon the extent to which the reward structure of the MDP captures the true values of the airspace. In order to assess the fitness of various policies, a large number of potential reward structures must be considered. Solving for this set of policies can be time consuming when using value iteration (requiring approximately 60 hours on a 16 core machine), and is furthermore sub-optimal due to the discretization of the state space.

Deep reinforcement learning (DRL) is an alternative method for learning policies in challenging MDPs that has seen recent success in a variety of control tasks [12, 17, 4]. DRL parameterizes a value function, policy, or learned model with a neural network, thereby improving the scalability of the underlying algorithm. Although the collision avoidance setting differs from previously considered applications of DRL in that value iteration is a feasible solution method, we hypothesize that DRL may yet be well suited to this problem for three reasons. First, DRL benefits from generalization between states - by learning a parameterized function of the state space, these models may need to perform fewer updates in order to arrive at a good policy. Second, DRL allows for efficient state and state-action sampling - by focusing only on the state-action pairs that occur under a reasonable policy, we may save computation. Third, DRL may benefit from superior transfer learning than does value iteration. We present empirical evidence that supports the first and second hypotheses, as well as initial evidence supporting the third.

While our experiments ultimately indicate that DRL approaches outperform dynamic programming, accomplishing this in practice required extensive hyperparameter tuning, investigation into sampling methods, and experimentation with various algorithms. As such, we present experiments and results for each of these topics in addition to experimental results comparing the baseline DRL and dynamic programming methods.

Related Work

Policy Compression

A prerequisite question to answer prior to applying DRL to the collision avoidance problem is, “can neural networks efficiently represent UAV collision avoidance policies?”. Although neural networks are powerful function approximators, it is plausible that the significant variation of the value function across the relatively large state-space considered might pose challenges to learning a value function mapping with reasonably-sized networks. Julian et al. addressed this question, finding that a neural network can be used to com-

State Variable	Min	Max	Units
ρ	0	60760	ft
θ	-pi	pi	rad
ψ	-pi	pi	rad
v_{own}	0	1200	ft/s
v_{int}	0	1200	ft/s
τ	0	100	sec
prev ra	1	5	n/a

Table 1: State variables for the UAV collision avoidance problem: ρ gives the range between the aircraft, θ the relative angle, ψ the relative heading, v_{own} the ownship velocity, v_{int} the intruder velocity, τ the time until loss of vertical separation, and prev ra the previous resolution advisory (i.e., action).

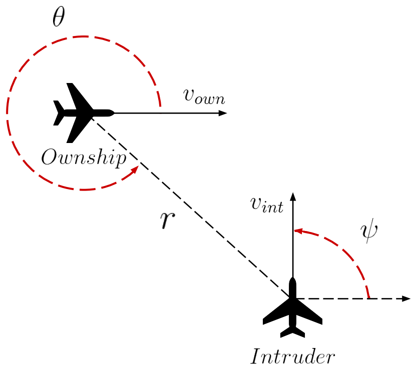

Figure 1: Depiction of the state variables in the UAV collision avoidance problem.

press a 2GB Q-value table to 2MB, and in doing so improve the performance of the policy derived from that value function in terms of safety and efficiency [7].

Deep Reinforcement Learning

DRL has recently seen a great deal of success. For example, it has been used to develop autonomous agents capable of surpassing human-level performance in challenging games such as Go [17], has enabled the development of systems capable of learning to drive simulated automobiles directly from high-dimensional visual input [11], and has been used to learn simulated, robotic control policies [6]. Central to this success is the ability of neural networks to dramatically improve scalability by allowing for generalization of learned behavior to unseen settings and efficient processing of high-dimensional input. Discussion of specific methods for efficient sampling (prioritized replay), alternative solution methods (Double DQN, Asynchronous Advantage Actor-Critic), regularization (dropout), and transfer learning are given in the methods section or in the appendices.

Problem Formulation

We consider a MDP in which the action of one UAV, the “ownship”, is controlled, and that of another, “the intruder”, is not. The state space is seven dimensional, and consists of the values presented in table 1. The previous action taken by the agent, referred to as the previous resolution advisory, is incorporated into the state in order to make it Markovian with respect to rewards dependent on reversals. The agent can take one of five actions, which correspond to standard rate (3°) and half-standard rate (1.5°) turns to the left and right, as well as an action to advise a straight trajectory.

The transition model of the MDP uses Dubin’s kinematic equations [3] to propagate the state forward in one second intervals. The heading of both aircraft is drawn from a Gaussian with mean being the advised action and a standard deviation of 0.0175 °, the velocity of the ownship and intruder are similarly distributed as a Gaussian with standard deviation 1.64 ft/s and 3.64 ft/s respectively. The rewards of the MDP additively comprise a structure selected to produce a policy reflective of the values of airspace stakeholders, and specifically consist of exclusively negative rewards for alerting, being in conflict (within 4000 ft), being in an NMAC (within 1800 ft), continuing to alert, switching to a “clear of conflict” (i.e., straight) action too early, reversing, and strengthening advisories.

Background and Methods

MDP

We model the collision avoidance problem as a MDP, which consists of a set of states S, actions A, transition probability function, P, deterministic reward function, R, start state distribution ρ, and discount factor γ. The maximum expected discounted returns or value of a state satisfies the Bellman optimality equation:

$$V^*(s) = \max_a R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^*(s')$$

Treating this equation as an update rule gives the value iteration algorithm, a dynamic programming method that solves for an optimal set of values that may be used to derive a policy.

Reinforcement learning

In reinforcement learning [20], transition and reward models are assumed to be unknown, and an agent instead uses experience from interacting with the environment to learn a value function or policy.

DQN The temporal difference algorithm Q-learning minimizes the squared error between the current estimate of a state-action value and a bootstrapped estimate of the optimal value of the next state plus reward. Parameterizing the Q-value function with a neural network yields the deep Q-network algorithm [12], which minimizes the same loss in batches of experience:

$$Loss_{DQN} = \sum_{s,a,r,s'} (r + \gamma max_a \bar{Q}(s',a;\theta) - Q(s,a;\theta))^2$$

Two methods are used to stabilize learning. First, a replay memory stores experience tuples (s, a, r, s'), from which batches are sampled to perform updates. This reduces correlations between the samples, and also leverages the parallel computation across batches using GPUs. Second, a target network (denoted by $\bar{Q}$) provides next state values, and is updated periodically with the weights of the prediction network, thereby further stabilizing learning.

Double DQN Q-learning and DQN have been shown to overestimate state-action values [5], which can diminish performance. Intuitively, this results from the fact that the algorithm uses a max operation over a set of approximate values, which biases towards selecting over-estimates of the value function [18]. The method of double estimators can be used to mitigate this issue, and when applied in the context of Q-learning is referred to as Double Q-learning (or Double DQN in the DRL context) [22]. In practice, this amounts to using the best action of the prediction network rather than a max over the values of the target network when computing the loss:

$$Loss_{Double-DQN} =$$
$$\sum_{s,a,r,s'} (r + \gamma\bar{Q}(s', argmax_a Q(s', a; \theta); \theta) - Q(s, a; \theta))^2$$

State space sampling

Reducing the number of samples an algorithm must see to reach convergence can dramatically speed learning. A variety of methods exist for reducing sample complexity, though we focus on the subset inspired by prioritized sweeping [13] and the more recent prioritized replay algorithm [16].

The intuition behind these approaches is that the state-values that should be next updated are those near states that previously gave "surprising" updates. Prioritized sweeping accomplishes this by maintaining a priority queue of states to update, prioritizing by the td-error of the transitioned-to state. Prioritized replay, which does not require a transition model, also prioritizes by td-error, but does so by performing weighted sampling from a replay memory.

Ideally, prioritized replay would sample new experience from areas of the state-space that are surprising so as to avoid overfitting noisy features of surprising transitions. Since a model is available in our setting, we take this approach - i.e., we adapt the initial state distribution ρ during training so as to re-sample experience from regions of the state-space that have previously been surprising, and consider the following metrics:

- td-error: the heuristic used by prioritized replay and prioritized sweeping: $r + \gamma max_a \bar{Q}(s', a; \theta) - Q(s, a; \theta)$
- Optimal action changes: this approach forward propagates states through the network after performing an update, and states for which the optimal action changed are stored, prioritized again by td-error.
- td-error overestimates: our problem as formulated consists exclusively of negative state-values. This method aims to reduce state-values as quickly as possible by only prioritizing states that are assigned values more positive than their estimates.

Transfer learning

Transfer learning with neural networks has seen success in the supervised setting [14]. The intuition behind transfer learning is that initial layers in a neural network can derive abstract features that are useful for a variety of applications. Transfer may be performed in the DRL setting by training a neural network on one set of rewards, the weights of which are used to initialize a second network used in learning the optimal value function for a second set of rewards.

Experiments

In order to compare the effectiveness of the various methods, we solve for value functions in a series of experiments in which only one aspect of training differs within each experiment between training runs. The resulting policies are then evaluated qualitatively through visualization of the learned state-action function, and quantitatively using both convergence metrics as well as through performance in a set of 10,000 simulated encounters.

These encounters were generated by a Bayesian network learned from real-life flight trajectories, and differ significantly from the relatively simple trajectories used during training. By evaluating against encounters generated from a fundamentally different MDP, we avoid over-fitting and are more likely to produce policies that may generalize in real-life application. The primary policy-evaluation metrics are the probability of near mid-air collisions (NMAC), probability of alerting, and the probability of reversing a previous advisory. At the time of evaluation, networks have been trained for approximately 30 hours each. Additional network implementation and training details are provided in the appendices.

Prioritized Sampling

We train three networks with prioritized start state sampling as well as a control network. For each network using priority sampling, 25% of the sampled states were drawn from a priority queue that is periodically updated with start states. The relatively small degree of sampling limits bias in the network towards more frequently sampled areas of the state-space. Figure 2 shows convergence metrics during training, and table 2 gives the final performance metrics for the three methods and control network. The convergence graphs are inconclusive as to the significance of priority sampling in speeding training, though simulation results indicate sampling may improve performance. We expect that a higher priority sampling probability would yield more significant convergence impact.

Regularization

Recent DRL work generally does not report the use of traditional neural network regularization methods such as dropout or L2 weight regularization. When training and evaluation of an agent is performed on an identical task, overfitting to that task may not be an issue. In contrast, in the case of learning UAV collision avoidance policies, not only do we train and evaluate the agent in separate environments, but we also ultimately intend to apply the learned policies in

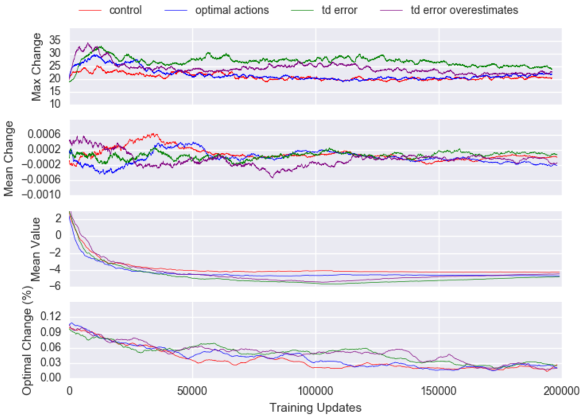

Figure 2: Convergence metrics for prioritized sampling methods considered. From the top: (1) maximum change in Q-value between updates (2) mean change in Q-value between updates (3) mean Q-value during training (4) percent change in optimal action between updates.

Sampling Method	Pr(nmac)	Pr(alert)	Pr(reversal)
no sampling	0.000191	0.481240	0.000826
td-error	0.000197	0.514591	0.010091
optimal-action	0.000026	0.575355	0.010304
td-error$^+$	0.000031	0.518542	0.002060

Table 2: Aggregate simulation metrics for various start-state sampling methods. No sampling refers to the control run, optimal-action to the prioritization by optimal action changes, and td-error$^+$ by td-error overestimates exclusively.

the airspace. For this reason, we experimented with the use of regularization in our models, and found it to be critical to learning smooth policies.

In supervised learning tasks, training is frequently performed with 50% dropout probability for units in hidden layers [19]. We initially experimented with 40%, 20%, and 5% dropout probabilities, but found that these values dramatically over-regularize the network. We then experimented with much smaller dropout probabilities of 1% and .1%, finding that these led to appropriately regularized policies, with 1% dropout performing the best. Figure 3 shows convergence metrics during training, figure 4 shows visualized policies for differing amounts of dropout, and table 3 provides aggregate simulation results for the different policies.

Notably, when training with dropout, the change in optimal actions between updates converges much more slowly, and the max change in Q-value does not converge at all. This is because these metrics are computed by comparing two separate forward propagations of the network in which the first uses dropout while the second does not. This approach is taken in order to limit the computational expense of computing the convergence metrics, and results in different units activating across propagations.

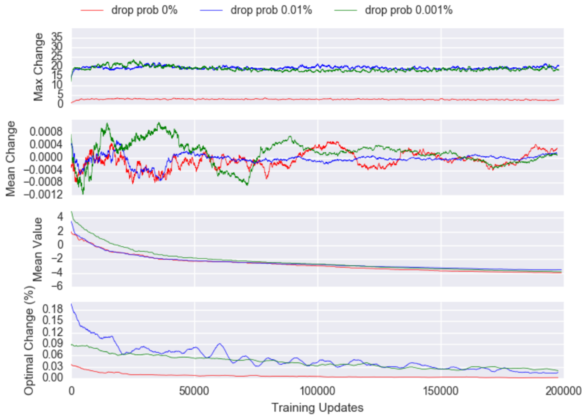

Figure 3: Convergence metrics for dropout probabilities considered. From the top: (1) maximum change in Q-value between updates (2) mean change in Q-value between updates (3) mean Q-value during training (4) percent change in optimal action between updates.

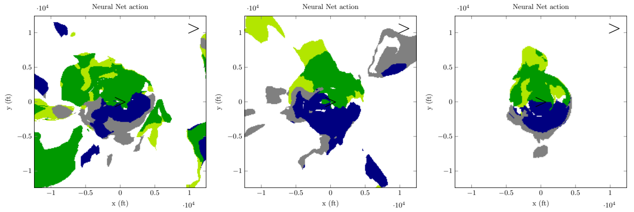

Figure 4: Left to right, policy plots for networks trained with 0, 0.001, and 0.01 probability of dropping units. These policy plots show the optimal action for the ownship (located at the center of the plot and represented by a black arrow) to take for every position of the intruder (the heading of which is indicated by the arrow in the top right of each plot). In the depicted plots, both aircraft are traveling in the same direction, at 200ft/s, equal elevation, and with the ownship not having taken a previous action. Light green, dark green, blue, gray, and white indicate strong left, left, right, strong right, and straight actions respectively.

State-space Discretization

A primary benefit of the DQN over value iteration is its ability to learn a mapping directly from the continuous state-space to state-action values rather than discretizing the state-space; however, since our primary goal is to use a DQN to learn policies rapidly in order to tune system parameters, we must ensure that the policies learned by the DQN correlate well with the policies learned through value iteration. Furthermore, because the reward structure is assumed fixed for this research, and that reward structure has been tuned to optimize the performance of discrete value iteration, we hypothesize that discretizing the dynamics (i.e., next state transitions) while training the DQN will give superior evaluation performance as well as allow for a fairer comparison of the two methods.

To accomplish this, DQN training was performed as

Dropout Probability	Pr(nmac)	Pr(alert)	Pr(reversal)
0.0	0.000195	0.552565	0.002850
0.001	0.000033	0.537368	0.012232
0.01	0.000028	0.487603	0.000290

Table 3: Simulation performance of DQN trained with varying dropout probability. 0.0 indicates the control network trained without dropout. Performance improves with increasing dropout for both pr(nmac) and pr(alert).

usual, except that for each state, a set of next states is returned along with their probability weights as determined by multilinear interpolation. These next states and weights are then stored in a replay memory, and during training the loss of each (s, a, r, s', w) tuple is weighted by its transition probability w. As can be seen from figure 5, the network trained on discretized dynamics converges more slowly than the control network. This is at least partially attributable to the heavy regularization effect of discretizing the dynamics.

Figure 6 shows policy and value plots for the Q-value table solved for using value iteration, and for the network trained with discretized dynamics. These visualizations indicate that the network converged to a policy that resembles that of the table, but that is often more conservative (i.e., advises non-straight actions where the table does not). This may indicate that the network had not yet converged, or that the combination of dropout and discretized dynamics over-regularized the network. Nevertheless, the empirical results given in table 4 show that the network trained on discretized dynamics outperforms the value iteration policy in all evaluation metrics considered.

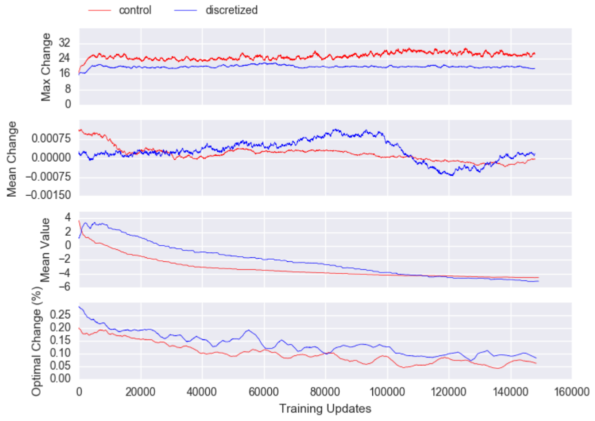

Figure 5: Convergence metrics for training on discretized dynamics. From the top: (1) maximum change in Q-value between updates (2) mean change in Q-value between updates (3) mean Q-value during training (4) percent change in optimal action between updates.

Double DQN

As previously stated, DQN frequently overestimates state-action values, often to the detriment of the policy. In an-

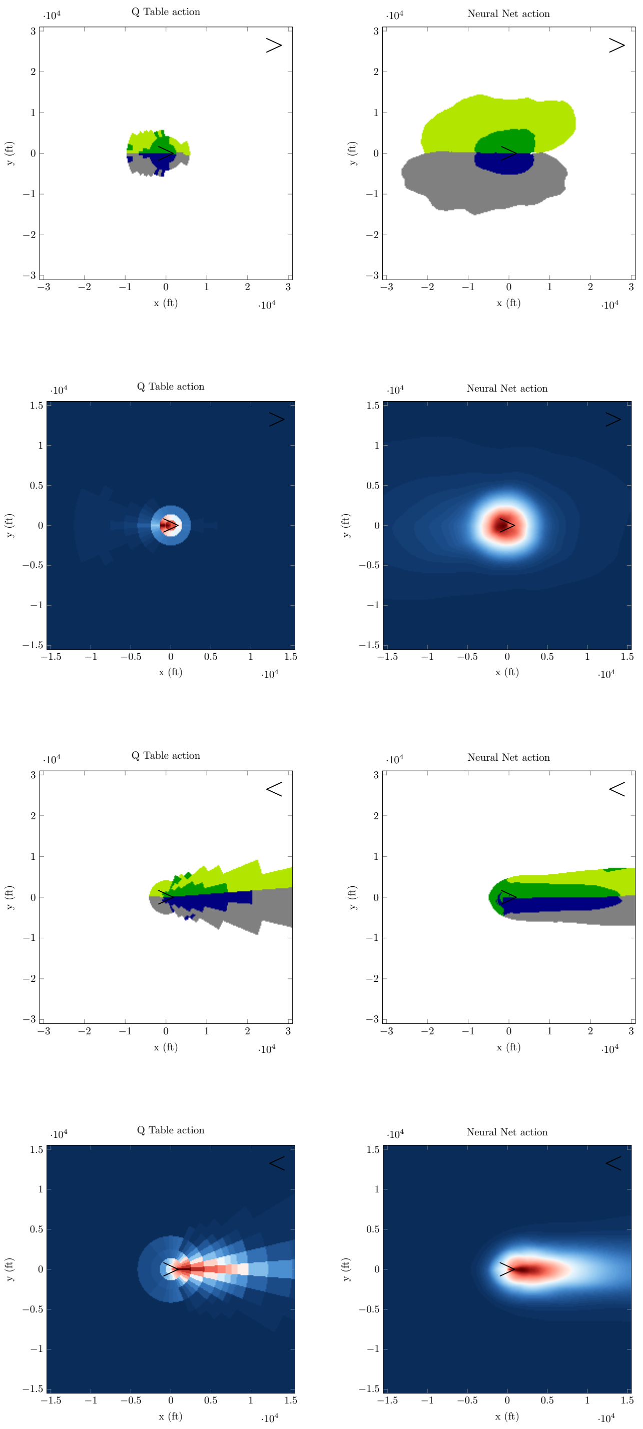

Figure 6: Policy and value plots for the Q-value table solved for with value iteration and the DQN trained on discretized dynamics. Top two plots show the two aircraft in a parallel geometry, and the bottom two at a head-on geometry. The table policy contains sharp edges because nearest neighbor is used to select the value of a state rather than an interpolation method. The value function plots depict the Q-values for the straight action with varying location of the intruder UAV. Light green, dark green, blue, gray, and white indicate strong left, left, right, strong right, and straight actions respectively.

Policy	Pr(collision)	Pr(alert)	Pr(reversal)
Q-table	0.000019	0.551474	0.015865
DQN	0.000017	0.545651	0.011015

Table 4: Results for the Q-table and DQN trained on discretized dynamics. Evaluation was performed against 10,000 encounters.

Policy	Pr(nmac)	Pr(alert)	Pr(reversal)
DQN	0.000194	0.464866	0.013367
Double DQN	0.000036	0.568237	0.000936

Table 5: Results for Double DQN and DQN against 10,000 encounters.

alyzing the evaluation encounters, the DQN policy would frequently alert later than the value iteration policy, often resulting in an NMAC. Although overestimating Q-values does not necessarily cause this issue, we hypothesize that it may be a factor, and as such trained and compared double DQN and DQN policies. Figure 7 shows the mean Q-value during training for both models, and clearly illustrates that Double DQN underestimates rather than overestimates Q-values. Table 5 gives aggregate simulation metrics for Double DQN and DQN, and suggests that underestimating Q-values leads to a higher alert rate as anticipated.

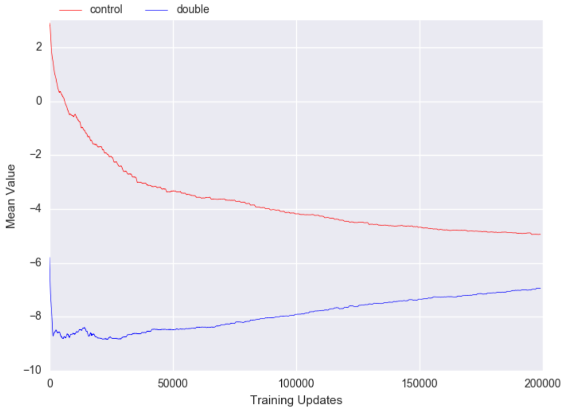

Figure 7: Mean Q-value for DQN and Double DQN.

Transfer

To evaluate the ability of the DQN to adapt pre-trained weights to a new reward structure, we first trained a network to convergence on the original reward structure, and then evaluated three models on a different reward structure. Because the hyperparameters of the DQN are tuned to learn a policy from scratch, we compared a control DQN, a DQN with transferred weights but unchanged hyperparameters, and a DQN with transferred weights and hyperparameters better suited to the transfer task. In this experiment, we consider the simple case of changing a single reward value, specifically increasing the cost of conflict from -1 to -5.

Figure 8 shows convergence metrics during training of the three models. The network using transfer weights and optimized hyperparameters seems to converge more quickly than the network trained from randomly-initialized weights, as can be seen from the combination of its mean change in Q-values being the lowest, while its mean Q-value converges to the same value as that of the control network, albeit more quickly. Unfortunately, since these networks were necessarily trained with different hyperparameters, this comparison is not entirely fair since it is possible that the convergence metrics simply reflect the use of a lower learning rate, larger target update interval, and increased batch size. Nevertheless, given that each of these parameters and others have been optimized for the respective tasks, it seems likely that the transfer network benefits from a significantly improved convergence rate.

The second conclusion that can be drawn from the convergence visualizations is that network transfer can be harmful to convergence if additional hyperparameter tuning is not performed. This can be seen most prominently in the exaggerated increase in the mean Q-value during training of the transfer network without tuning. We believe the reason for this mean Q-value increase is that the combination of a large initial learning rate and large initial loss “kills” a large fraction of the hidden layer ReLU units. This eliminates a balancing effect against the bias terms, which are large positive values (since they are not included in L2 regularization), yielding large, positive output values. Because in this case training begins with a low target update interval, these values are then transferred to the target network resulting in a self-reinforcing cycle.

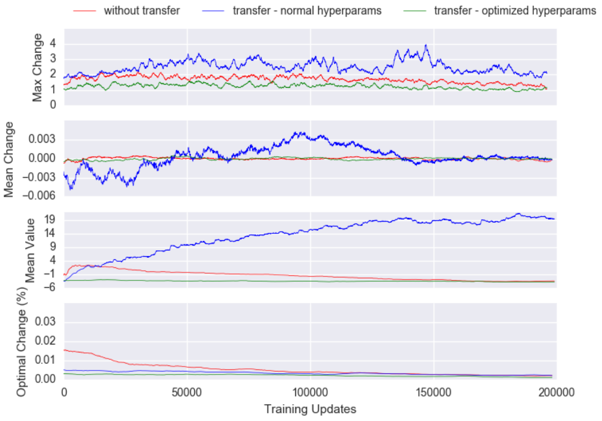

Figure 8: Convergence metrics during training over transfer and control networks. From the top: (1) maximum change in Q-value between updates (2) mean change in Q-value between updates (3) mean Q-value during training (4) percent change in optimal action between updates.

Conclusion

This research compared deep reinforcement learning methods to traditional dynamic programming methods in deriving solutions to a UAV collision avoidance problem. We considered different facets of training such as prioritized sampling, regularization, discretization of dynamics, alternative solution methods, and transfer learning, ultimately concluding that DQN can outperform value iteration both in terms of evaluation performance and solution speed.

This superior performance comes at a cost; namely extensive hyperparameter tuning and specialized hardware. Furthermore, the results presented in this paper potentially suffer from an "overfitting" of the relatively small encounter set used throughout. Future work may address each of these issues. For example, Asynchronous Advantage Actor-Critic (see appendix 2) has been shown to outperform DQN without requiring the use of GPUs. Addressing the latter concern may be accomplished through evaluation of the learned policies on a larger set of encounters.

Additional future work may consider network training across multiple GPUs, further investigation into transfer-learning, and model-based methods that bridge the gap between those considered in this research.

References

[1] Martın Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[2] James Bergstra et al. "Theano: Deep learning on gpus with python". In: *NIPS 2011, BigLearning Workshop, Granada, Spain*. Citeseer. 2011.

[3] Lester E Dubins. "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents". In: *American Journal of mathematics* 79.3 (1957), pp. 497–516.

[4] Alex Graves et al. "Hybrid computing using a neural network with dynamic external memory". In: *Nature* 538.7626 (2016), pp. 471–476.

[5] Hado V Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems*. 2010, pp. 2613–2621.

[6] Nicolas Heess et al. "Learning and Transfer of Modulated Locomotor Controllers". In: *arXiv preprint arXiv:1610.05182* (2016).

[7] Kyle Julian and Mykel Kochenderfer. "Neural Network Guidance for UAVs". In: 2017.

[8] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[9] Mykel J Kochenderfer, Jessica E Holland, and James P Chryssanthacopoulos. "Next generation airborne collision avoidance system". In: *Lincoln Laboratory Journal* 19.1 (2012), pp. 17–33.

[10] JE Kuchar and Ann C Drumm. "The traffic alert and collision avoidance system". In: *Lincoln Laboratory Journal* 16.2 (2007), p. 277.

[11] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *arXiv preprint arXiv:1602.01783* (2016).

[12] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[13] Andrew W Moore and Christopher G Atkeson. "Prioritized sweeping: Reinforcement learning with less data and less time". In: *Machine Learning* 13.1 (1993), pp. 103–130.

[14] Maxime Oquab et al. "Learning and transferring mid-level image representations using convolutional neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 1717–1724.

[15] Benjamin Recht et al. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in Neural Information Processing Systems*. 2011, pp. 693–701.

[16] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[17] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[18] James E Smith and Robert L Winkler. "The optimizer's curse: Skepticism and postdecision surprise in decision analysis". In: *Management Science* 52.3 (2006), pp. 311–322.

[19] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[20] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[21] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural Networks for Machine Learning* 4.2 (2012).

[22] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double Q-learning". In: ().

[23] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[24] Bing Xu et al. "Empirical evaluation of rectified activations in convolutional network". In: *arXiv preprint arXiv:1505.00853* (2015).

hyperparameter	value
initial learning rate	0.001
final learning rate	0.00001
initial target update interval	100
final target update interval	20000
replay memory size	10000000
optimizer	adamax
adamax beta 1	.99
dropout probability	0.01
l2 regularization penalty	1e-6
discount factor	.99
policy	softmax
initial softmax temperature	6
final softmax temperature	3
initial batch size	16000
final batch size	64000
steps per episode	10
steps between training updates	10

Table 6: Final hyperparameter values for DQN training.

Appendix 1: Network Implementation and Training Details

DQN and Double DQN were both implemented using Lasagne and Theano [2]. Hyperparameter settings differed between experiments as improvements were found during the research. Table 6 gives the final hyperparameter values used in training. Network sizes also differed across experiments, with the final architecture being an eight-hidden-layer network with 256 hidden units in each layer, except for two central layers that contained 512 units each. See the section below on network size for discussion. DQN training was performed on an Nvidia DIGITS DevBox with four Titan X GPUs. Results presented are generally from networks trained for 30 hours on a single GPU.

The following sections discuss subtleties of the various hyperparameters with an emphasis on their role specifically within the UAV collision avoidance problem.

Maximum Episode Steps The number of steps in an episode in the UAV collision avoidance problem controls the degree to which samples focus on straight versus turning previous advisories. This is because even though initial states are sampled with random previous advisories, the optimal action will generally be the straight advisory. As such, the longer the episode is run, the greater the bias toward straight previous advisory samples.

The maximum steps per episode also impacts the state-visitation frequencies - running episodes for longer time periods will sample from larger-range states because the UAVs generally avoid each other provided softmax policy sampling is run at a sufficiently low temperature.

Finally, as each simulation is run, the value of τ will tend towards 0. As such, lower max step values with uniform initial distribution over τ yields more even coverage, whereas longer max step values will bias towards τ values of zero.

State Normalization Prior to input to the DQN, the state variables are normalized by subtracting their mean value and dividing by their range. In the case of angle inputs, if the angle is provided to the network directly, then a discontinuity will exist in the policy, which can diminish performance. To avoid this, we insert the normalized sine and cosine of the angle into the network rather than the angle itself.

Baseline Start State Sampling Implicit in any DRL approach is a distribution over the encountered states. In the UAV collision avoidance problem as formulated, sampling states by uniformly sampling variables yields a heavy bias toward low-range states. By sampling the x and y distances between aircraft, and then converting to the polar-coordinate state variables, more equitable state-visitation frequencies can be achieved, which we found empirically improved policies.

Network size Our experiments indicated that larger networks better capture the variability within the collision avoidance policy state-values. We compared three network sizes, each with six layers, and each containing two intermediate layer sizes of 512 hidden units. The remaining layers contained 512, 256, and 128 units each. Td-errors were on average much lower for the larger networks, indicating that this additional capacity enabled the network to remember more of what had previously been learned.

Target Update Interval The update rate for the target network significantly impacts training time because value information can propagate only after a target update is made. We found that using a target update schedule that increased the interval periodically by a constant amount dramatically improved performance.

Optimization We considered three optimization methods: Adamax [8], RMSProp [21], and stochastic gradient descent with Nesterov momentum. Adamax slightly outperformed RMSProp.

Non-linearity We compared three nonlinear functions for use in the network: tanh, ReLU, and leaky ReLU [24]. The leaky relu performs a $\max(x, 0.01)$ operation that is intended to alleviate the "dying relu" issue. We found that ReLU and leaky ReLU performed comparably given an appropriate learning rate selection, and that tanh performed the worst, but gave the smoothest policies.

Reward Baseline and Scaling Reward magnitude and sign can impact DQN performance to varying degrees based upon the nonlinearity used within the neural network. For example, ReLU units can have difficulty fitting negative values, particularly when regularization is used, and tanh units can have difficulty fitting output values with large magnitudes. We considered two methods to address these issues.

First, since all of the rewards in the UAV collision avoidance problem are negative, we can "baseline" them by adding a fixed constant to the reward values. We found that this approach did not help with learning and frequently caused the network to diverge due to the large initial errors. Second, we experimented with reducing the magnitude of the reward values, specifically reducing them by a factor of

10. We found that this approach yielded similar policies to the original reward structure, and did not significantly speed learning.

Gradient and Loss Clipping A popular method for stabilizing learning in DRL agents is to clip (i.e., bound) the norm of the gradient or the loss directly. We found this method to be useful for this purpose, but that its use prevented the DQN from learning appropriate reversal policies. The reason for this is that reversals incur a large negative reward, but do so quite rarely. Because the impact of these rare events is limited by gradient clipping the network would not learn to reflect it in the policy, and as such we did not use gradient or loss clipping during training.

Appendix 2: Asynchronous Advantage Actor-Critic

DQN exclusively parameterizes a value function with a neural network. In this section, we consider a DRL algorithm that parameterizes both its policy and value functions with neural networks. Asynchronous Advantage Actor-Critic (A3C) [11] uses policy and value networks to perform REINFORCE [23] updates with a return baseline:

$$Loss_{\pi} = -\log \pi(a_t|s_t;\theta)(R_t - V(s_t;\theta)) - \beta H(\pi(s_t;\theta))$$

$$Loss_{val} = (R_t - V(s_t;\theta))^2$$

Where R_t is the experienced return from state s_t, which have been bootstrapped using the value network, and β is a scaling factor on the entropy H of the policy that encourages diverse action probabilities. Intuitively, the gradient of this loss updates the policy to encourage the taking of actions that have produced greater than expected returns, while simultaneously learning those expected returns by minimizing the L2 loss for the value function. To overcome oscillation in the policy from correlated updates, A3C executes many agents in parallel, and makes Hogwild!-style [15] gradient descent updates using experience batches from each of these agents.

For this research, we implemented a thread-based version of A3C in TensorFlow [1]. We do not yet have results on the UAV collision avoidance task, and instead we present results on a toy cart pole problem. Figure 9 shows the average steps per episode during training and action probabilities of the agents as a function of the cart x position. A3C effectively solves the task in 1,000 episodes, which takes approximately 60 seconds on a macbook pro. While not shown, DQN empirically takes much longer to solve the task. While this might be due to particular advantages of low-bias methods in learning cart-pole policies, it may also indicate an advantage of A3C in general, and we plan the compare the algorithms in the primary UAV collision avoidance task.

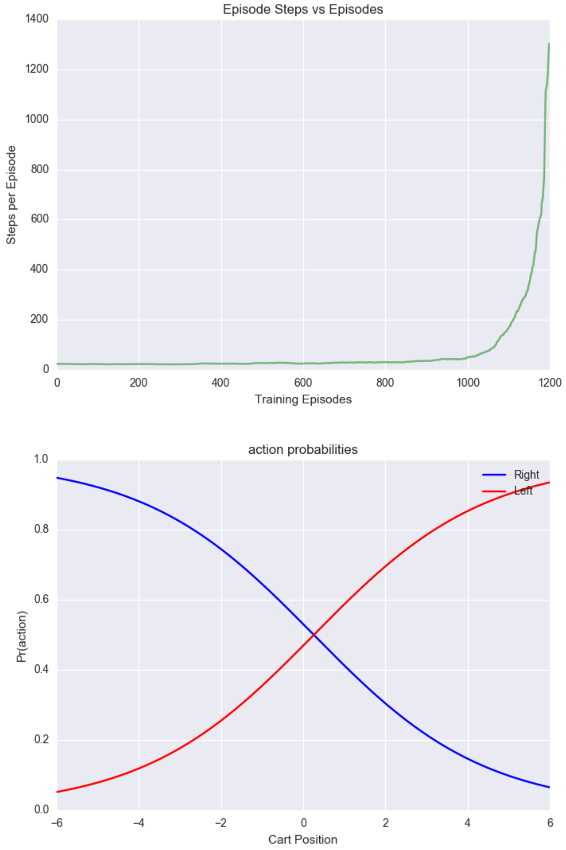

Figure 9: Steps per episode versus training episodes, and action probabilities as a function of cart x position.