# Airborne Collision Avoidance Through Deep Reinforcement Learning

Kyle Julian
Aeronautics and Astronautics
Stanford University
Palo Alto, California, 94304
Email: kjulian3@stanford.edu

***Abstract*—Unmanned aircraft have a wide range of potential applications, including package delivery, emergency medical assistance, and wildfire surveillance. In order to ensure the safe use of unmanned aircraft technology, a robust system of collision avoidance must be implemented. One approach to solving this problem generates advisories for aircraft to follow in order to avoid collision. This approach has been formulated as a partially observable Markov decision process and solved with standard algorithms like QMDP. These solutions generate large score tables with millions of state-action values, which can become intractable to store in memory in small aircraft. In addition, these algorithms can take a long time to converge, making trade studies on the reward and transition models very time consuming. This paper presents an alternative approach using deep reinforcement learning, a new technique capable of tackling complex sequential decision making processes. This model-free approach trains a neural network representation of Q values while exploring the state space. The trained neural network parameters occupy much less space in memory, allowing unmanned aircraft to incorporate collision avoidance systems within flight software.**

## I. Introduction

Unmanned aerial systems (UAS) have the potential to impact many aspects of modern society by delivering packages, inspecting roadways, delivering defibrillators, and fighting forest fires. With the drop in prices of UAS, many groups already want to use the technology to improve their businesses. However, without supervision, UAS could collide with commercial aircraft, buildings, or one another. Currently, NASA is creating a UAS Traffic Management (UTM) system to supervise the operation of UAS [1]. This system is currently in development, but as the number of UAS grows, it will need to incorporate some sort of automatic collision avoidance system.

Already, the collision avoidance system ACAS X has proven to be a safe and reliable method of ensuring collision avoidance in commercial aircraft [2]. This approach frames the problem as a Markov decision process (MDP) and determines the optimal advisories through dynamic programming. In simulation, this approach is shown to be robust to sensor noise and to diverse pilot responses, and ACAS X has been shown to out-performs legacy TCAS systems.

A variant of ACAS X is already under developement for use with UAS, known as ACAS Xu [3]. In 2015, the system passed an initial flight test, showing that the MDP formulation can also be applied to UAS [4]. Since UAS could one day operate in a tighter airspace with heavier traffic, it is important to ensure that coordinated maneuvers can be calculated for conflicts involving more than two aircraft. This problem can be solved by reducing the problem into separate pairwise encounters and then building advisories based upon the sum of the Q values for each of the pairwise encounters [5].

These problem formulations have all been solved with different variants of MDP solvers using dynamic programming [2]. The resulting solution is table composed of relative scores for each discretized state-action pair. In order to render an accurate solution, these discretizations have to be very refined, resulting in very large score tables. This poses two problems. First the memory required to store these score tables can be on the order of 5GB, which is too large for small drones with limited memory to store onboard. Second, as the discretization becomes more refined, the time required to compute the score tables takes much longer, which makes tuning parameters in the algorithm very time consuming.

However, recent developments in deep reinforcement learning offer a solution. In 1995, Garld Tesauro's TD-Gammon algorithm demonstrated that neural networks can be used in reinforcement learning to solve complex problems, like learning a policy for the game backgammon [6]. Google's DeepMind extended this idea using a type of algorithm known as deep reinforcement learning in order to create a single agent that can learn to play a wide variety of Atari games using only pixel data [7]. The DeepMind group went on to create the first Go playing agent to beat a professional human player using a combination of deep reinforcement learning and Monte Carlo tress search [8].

This paper incorporates the advancements in deep reinforcement learning into the problem of aircraft collision avoidance for UAS. Because the network is a global non-linear approximater, it can learn a good approximation for the score table. After training, the neural network parameters are all that is needed to represent the table. Since there are only a few thousand parameters, a score table with millions of values can be compressed greatly. In addition, the neural network is able to abstract information at a global level, making it possible to learn more quickly than by iterating through every possible discretized state-action pair.

## II. Problem Formulation

This approach uses a formulation very similar to Ong and Kochenderfer, which will then serve as a comparison to the work produced here [5].The problem is formulated as a Markov decision process, which is composed of state space $S$ and action space $A$. When an agent is in state $s \in S$ and chooses an action $a \in A$, it will receive reward $r$ and move to state $s'$ with some transition probability $T(s' \mid a, s)$. A policy $\pi$ describes the action to take in any given state. The optimal policy seeks to maximize the expected utility, which is the sum of immediate reward and discounted utility of future states. The Bellman equation gives a recursive formula that solves for the optimal utility of each state-action pair, $Q^*(s, a)$.

$$Q^*(s,a) = R(s,a) + \sum_{s' \in S} T(s' \mid s,a) \max_{a' \in A} Q^*(s',a') \quad (1)$$

However, this formulation assumes we have knowledge of the transition function. Instead, the optimal Q values can be solved for iteratively and model-free by using samples of *(s,a,r,s')* using learning rate $\alpha$. This approach is known as Q-learning

$$Q(s,a) = Q(s,a) + \alpha * \left[ R + \gamma \max_{a' \in A} Q(s',a') - Q(s,a) \right] \quad (2)$$

In this problem formulation, there are two aircraft, an ownship that is controlled by policy $\pi$ and an intruder flying nearby at the same altitude. The Q values correspond to the expected utility of being in state $s$ and taking action $a$ and then acting optimally from there. The goal of this problem is to calculate the optimal policy the ownship should take given the state of the ownship and intruder.

### A. States

This formulation uses five continuous states. Because neural networks are global approximaters, they do not need any discretization like Ong and Kochenderfer [5].

- $R$: Range from ownship to intruder. [0,3000m]
- $\theta$: Angle from ownship's heading to intruder. $[-\pi, \pi]$
- $\psi$: Relative heading of the intruder from the ownship's heading. $[-\pi, \pi]$
- $V_{own}$: Speed of ownship. $[10m/s, 20m/s]$
- $V_{int}$: Speed of intruder. $[10m/s, 20m/s]$

These states are the same as used by Ong and Kochenderfer, except $x$ and $y$ have been converted into polar coordinates $R$ and $\theta$ [5].

### B. Actions

The actions used are the same as used in Ong and Kochenderfer [5]. These actions represent bank angle advisories. Therefore $A = \{-20°, -10°, 0°, 10°, 20°, COC\}$, where COC represents a clear-of-conflict advisory, allowing the ownship to move freely. The negative bank angles represent right turns while the positive bank angles represent left turns.

### C. Rewards

The reward system used here differs from that used by Ong and Kochenderfer slightly. Neural networks do not respond well to large discontinuities in the reward function, because this can produce large loss terms that make learning chaotic and suboptimal. So instead of issuing a large penalty when the intruder is within 500 meters of the ownship and no large penalty when the intruder is farther away, the reward function is smoothed with a sigmoid function. In addition, the scale of the rewards was adjusted to make the largest reward issued to be approximately $-1$. Finally, the magnitudes of the components of the reward were brought closer together in order to allow the neural network to approximate all components of the reward function better.

There are three components that make up the reward overall function, with $\phi$ representing the bank angle:

- $R_1 = -\left(1 + e^{(r_{sep} - r_{min})/C}\right)^{-1}$
- $R_2 = -0.0002\phi^2$ with $\phi$ in degrees
- $R_3 = -0.03$ if $\phi \neq COC$

The overall reward is the sum of the three individual components. $R_1$ represents the smoothed step function that penalizes the ownship for being too close to the intruder. The $r_{sep}$ used here represents the minimum separation of the ownship and intruder during the maneuver. The parameter $r_{min}$ is the minimum allowable separation distance, which is set to 500 meters. The parameter $C$ represents the amount of smoothing applied to the step function. After trying different values, as shown in Figure 1, 100 was used as the final smoothing parameter. $R_2$ gives a penalty for banking unnecessarily, and $R_3$ gives a penalty for issuing an advisory when one is not needed.

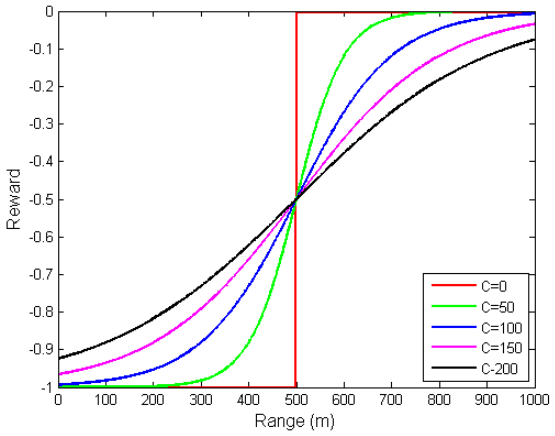


Fig. 1. $R_1$ rewards for different values of smoothing parameter

### D. Dynamics

When modeling how the ownship follows a given advisory, one cannot expect the aircraft to follow the bank angle precisely. Instead, the ownship's true bank angle is sampled from a normal distribution with the advisory as the mean

and a standard deviation of 4°. When there is no advisory (COC), then the ownship's true bank angle is sampled from a normal distribution with mean 0° and standard deviation of 10°. This represents the idea that an aircraft in a conflict situation may adhere more closely to advisories than trying to maintain a straight path when there is no conflict. This distribution follows Ong and Kochenderfer, but the samples are allowed to be continuous rather than discrete values with discrete probabilities [5].

When modeling the dynamics, advisories are given every 5 seconds, and true bank angles are held for the total period of five seconds. It is assumed that the time to change bank angles is very small, so the change in bank angle is modeled as an instant change. Furthermore, the speeds of both the ownship and the intruder are allowed to change. The velocity of the next state is sampled from a normal distribution with mean equal to the velocity at the current state and a standard deviation of 2 m/s. Therefore, the transition from one state to the next is stochastic.

A simple dynamic model was used to calculate the new state variables given the true bank angle, as shown by the equations below. The coordinate frame is chosen so that the ownship is always moving in the positive x-direction. Therefore, after calculating how much the ownship has turned due to its bank angle, the coordinates are rotated to have the ownship traveling along the x-axis once more.

$$x_{i+1} = r_i \cos\theta + (V_{int}\cos\psi - V_{own}) * dt \tag{3}$$

$$y_{i+1} = r_i \sin\theta + V_{int}\sin\psi * dt \tag{4}$$

$$\Delta\psi = -g\frac{\tan\phi}{V_{own}} * dt \tag{5}$$

$$r_{i+1} = \sqrt{x_{i+1}^2 + y_{i+1}^2} \tag{6}$$

$$\theta_{i+1} = \arctan\left(\frac{y_{i+1}}{x_{i+1}}\right) + \Delta\psi \tag{7}$$

$$\psi_{i+1} = \psi_i + \Delta\psi \tag{8}$$

## III. Deep Q Networks

Up until this points, the approach has been very similar to Ong and Kochenderfer, but the actual solution technique differs greatly. While Ong and Kochednerfer used a POMDP algorithm known as QMDP to generate a solution, I propose a deep reinforcement learning algorithm. At the core of this is approach is a deep fully connected neural network with seven hidden layers and rectified linear unit activations between the hidden layers. The five state variables are first normalized to have a mean of 0 and range of 1. These variables are then passed into the network, which performs feedforward calculations to produce six outputs. Each output represents an estimate of the Q value for taking one of the actions for the given state. The loss function used to train the neural network parameters, $\theta$, comes from the Bellman residual term in Q-learning, seen in Equation 2. Given a sample $(s, a, r, s')$, the loss function is the square of the Bellman residual.

$$L(\theta) = \left[r + \gamma \max_{a' \in A} Q(s', a', \theta) - Q(s, a, \theta)\right]^2 \tag{9}$$

### A. Target Network

Because neural networks are global approximaters, and because usually $s$ is very similar to $s'$, modifying the network parameters to move $Q(s, a)$ closer to $r + \gamma \max_{a' \in A} Q(s', a', \theta)$ will also move $Q(s', a', \theta)$ too. This can lead to instabilities in the network by over-estimating Q values. To combat this problem, a second neural network with fixed parameters is used to evaluate the current version of the Q network. This second network, called the target network, acts an evaluator of the current Q network's ability to reduce the Bellman residual. Every so often, the target network is updated to the parameters of the current Q network. Between updates of the target network, the learning process becomes supervised learning, making the training slower but more stable.

### B. Expected Loss

For a given state and action pair, the resulting reward and next state could vary greatly due to the stochastic transitions. For a neural network, this can make training difficult since the target reward and next state can change. Therefore, the neural network loss uses the expectation of the reward and next state given a state and action. This expectation can be estimated by simulating $K$ trajectories and averaging the rewards and max Q values of the next states. The final loss function becomes:

$$L(\theta) = \left[\mathbb{E}\left(r + \gamma \max_{a' \in A} Q(s', a', \theta^-)\right) - Q(s, a, \theta)\right]^2 \tag{10}$$

### C. Experience Replay

Consecutive samples of $(s, a, r, s')$ are often highly linked together. If the Q-network were trained on consecutive samples, one area of the state space would be over estimated while other areas would be left unexplored. This results in poor performance for the global approximater. This solution is to incorporate experience replay, where samples of $(s, a, r, s')$ are stored in memory, and random samples are drawn from memory to update the network parameters [7]. In this way, successive updates will occur all over the state space, ensuring that the state space is well explored. First, the algorithm generates 200,000 samples where actions are chosen randomly. Then this set is used to draw random samples to update the parameters. Batch updates are performed on the network, with a batch size of 512 samples. This technique helps to speed up the learning process and makes estimations of the gradient more reliable [9].

Between batch updates of the network, new samples must be added to the experience replay memory. Because we update 512 random samples at a time, 20 new samples are added between updates. This ensures that existing samples are not overplayed while not wasting un-played samples. The

experience replay memory has a limited space for 1.5 million samples, and the oldest samples are thrown out first. This helps to ensure that new samples based on choosing better actions have a chance to be played.

When generating samples, actions are chosen $\epsilon$-greedily, where $\epsilon$ represents the probability of choosing a random action rather than choosing the action with the maximum Q value. Choosing random actions helps to explore the full state-action space. However, as the network parameters learn a better and better representation of Q, it is inefficient to explore actions that are shown to be suboptimal. For this reason, $\epsilon$ is annealed from 1 to 0.1 over the course of 7.5 million trainings of the neural network parameters.

## IV. Algorithm

The algorithm implemented contains three central classes. The first class, Q, contains the two neural networks as well as the functions used to select actions and train. It contains internal counts so that $\epsilon$ can be annealed, and so that the target network can be updated periodically.

The second class, D, stores all of the samples of $(s, a, r, s')$ and contains methods to draw from the memory at random. Storing new data will result in deleting the oldest data if the maximum storage size is reached.

The third main class, G, is a generator of states as well as next states. This class contains functions for propagating dynamics, and it will calculate N different trajectories and return the next states and rewards. It also contains methods to select a random state if the intruder ever moves farther than 3000 meters away.

A central function ties all of these components together into one algorithm. This main function is set to loop until one is satisfied that the resulting plot has converged satisfactorily.

**Algorithm 1** ACAS Xu DQN Algorithm

**Input:**
  $Q$, class containing deep Q networks
  $D$, class for experience replay
  $G$, class for dynamcis and state generator
**function** MAIN(Q,D,G)
  $s \leftarrow G.randomState()$
  **repeat**
    **for** $i \leftarrow 1$ **to** $k$
      $a \leftarrow Q.getAction(s)$
      $s', r \leftarrow G.nextStates(s, a, N)$
      $D.store(s, a, r, s')$
      $s \leftarrow s'[0]$
      **if** $s.Range > 3000$
        $s \leftarrow G.randomState()$
    **if** $D.canTrain()$
      $batch \leftarrow D.sampleBatch()$
      $Q.train(batch)$
  **until** converged

**Algorithm 2** Q Functions

**function** GETACTION(s)
  selft.count++
  **if** $self.count < initial\_size$
    **return** $A.randomAction$
  $\epsilon \leftarrow getEps(self.count)$
  **if** $random(0, 1) < \epsilon$
    **return** $A.randomAction$
  **else**
    **return** $A\left[argmax\left(self.network.evaluate\left(s\right)\right)\right]$
**function** TRAIN(batch)
  $target \leftarrow network$, every C calls
  $qNetwork \leftarrow network.predict(batch.states, actions)$
  $qTarget \leftarrow \max(target.predict(batch.nextStates))$
  $loss \leftarrow (qNetwork - batch.rewards - \gamma * qTarget)^2$
  $network.train(loss)$

**Algorithm 3** D Functions

**function** SAMPLEBATCH( )
  **for** i $\leftarrow$ 1 **to** batch_size
    $sample[i] = D[random(0, D.size)]$
  **return** $sample$

**Algorithm 4** G Functions

**function** NEXTSTATES(s,a,N)
  **for** i $\leftarrow$ 1 **to** N
    $bank \leftarrow getBank(a)$
    $s'[i], r[i] \leftarrow nextState(s, bank)$
    $s'[i].vown \leftarrow changeSpeed(s.vown)$
    $s'[i].vint \leftarrow changeSpeed(s.vint)$
  **return** $s', r$

## V. Implementation

The algorithms above were written in python using the Keras neural network library. Keras was used because it contains many built in functions that make prototyping networks with different architectures, loss functions, and optimizers very easy. Both networks were comprised of 6 hidden layer with five inputs and six outputs. From input to output, the sizes of these hidden layers are 128, 512, 512, 128, 128, and 128 respectively. After testing a few different optimizers, AdaMax was chosen. This flavor of gradient optimization adaptively estimates lower-order moments, which allows the optimizer to be very efficient without need for parameter tuning [10]. The hyper-parameters used in the algorithm were tuned over a few iterations of the algorithm. Table 1 below shows these parameters.

Table 1: DQN parameters

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| D_Initial_Size | 200,000 | Train_Freq | 20 |
| D_Max_Size | 1.5M | Target_Freq | 15000 |
| Final_Epsilon | 0.1 | Batch_Size | 512 |
| Final_Exploration | 7.5M | Num_Traj | 16 |
| $\gamma$ | 0.97 | $\Delta t$ | 5 sec |

## VI. Results

The DQN algorithm was run on an Nvidia DIGITS DevBox, allowing the neural networks to be trained on four Titan X GPUs. The time required to run the algorithm between target updates is approximately 25 minutes. Target updates are similar to value iterations in that each new target update spreads the rewards out to neighboring states. After just 10 target updates, the policy of the neural network algorithm begins to take shape, and running the algorithm for longer produces small improvements to the policy. The discretized problem was solved with QMDP in one hour by Ong and Kochenderfer, so this current solution is not yet as fast as desired [5]. However, this algorithm does not slow down as the state space becomes discretized further, and the algorithm should scale well if more inputs are given to expand the problem to three dimensions. Therefore, it is hopeful that after optimizing the hyper-parameters more and expanding the problem to be more complex, the DQN approach should present a relatively quick solution.

A policy plot was created to compare the policies created by the DQN and by the QMDP algorithm presented in Ong and Kochenderfer [5]. In Figure 2, the ownship and intruder are moving at in a 180° relative heading collision path with speeds of 10 m/s. The figure shows an overhead view of the conflict between the ownship and the intruder. The arrow in the upper right corner shows that the intruder is heading towards the left side of the plot, while the ownship is heading to the right side of the plot.

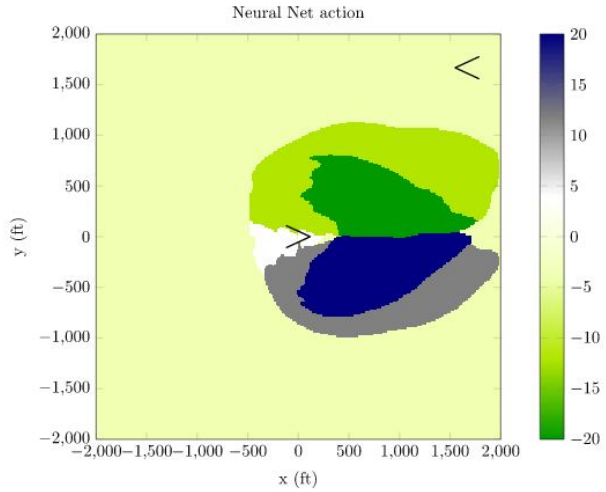


Fig. 2. Optimal policy for 180° conflict.

The colors on the map correspond to the maximum Q value if the intruder was in that location relative to the ownship at the center of the plot. If the intruder is in a green area, then the best action for the intruder is to bank to the right. If the intruder is in a yellow area, then best action is to issue a COC, white represents a straight advisory, and blue and grey represent left bank advisories.

Figure 2 shows the policy computed after 28 target updates. At this point, the policy has mostly converged and is fluctuating very little. When compared to Ong and Kochenderfer plots, there are many similarities. First, both plots follow natural intuition by telling the ownship to bank left if the if the intruder is in front and to the right, and bank left if the intruder is in front and to the right. In addition, the advisories become stronger if the intruder is closer to the ownship, which one might expect. Both plots also show a notch directly in front of the ownship. This notch tells the ownship to fly straight if the intruder is directly in front and close to the ownship. This seems like a very bad policy, but the ownship would normally have turned before the intruder made it to that location, so that characteristic isn't problematic in practice.

However, there are also some differences in the two plots. First, the DQN policy is much more eager to issue a COC. This could be a side-effect of reconstructing the rewards in order to help the DQN converge in a more stable manner.

Second, the actions extend much farther down rage, even as far as 2000 meters, while the QMDP solution only extends to 1500 meters. Likely, this due to smoothing the reward function. For the DQN, sizeable penalties are given when the intruder is as far as 1000 meters away, while the QMDP approach only issued large penalties when the intruder was within 500 meters. This will make the penalties spread out farther away from the ownship.

Lastly, the QMDP policy shows a few "bubbles" of different actions in places they do not seem to belong. The DQN approach does not contain these bubbles. In fact, the DQN policy shows generally smooth shapes. Deep Q networks trained on random samples converge to global approximations of the best fit curves, while QMDP may be over-fitting in some areas. In this way, the DQN approach may be able to create better policies, even though they are only estimates of the true Q values.

The neural network parameters occupy less than 5 MB in memory. With 14 million discretized state-action pairs, it would take about 53MB to specify each individual Q value. Therefore, the DQN approach requires a factor of 10 less memory to store the representation of Q. Thus, the goal of compressing the policy has been achieved.

There are still some limitations of this approach. First, the neural network's loss converges to a finite value. This is mainly a result of the stochastic nature of the transitions and rewards. The DQN will never be able to predict exactly the rewards and Q values of the next states when they are subject to some random noise. This results in policies that do not converge completely either. Places where Q values are very close between different actions can oscillate to make different

actions the optimal action. This effect only occurs near the boundaries of two optimal action areas, so it is not a major limitation.

Figure 3 shows the training loss for the neural network. The loss is very low in the beginning, since the Q values have yet to incorporate future discounted rewards, making the Q function very easy to model. The periodic spikes at the beginning of the training occur when the target network parameters are updated. Theses spikes become less pronounced over time as the Q values begin to converge.

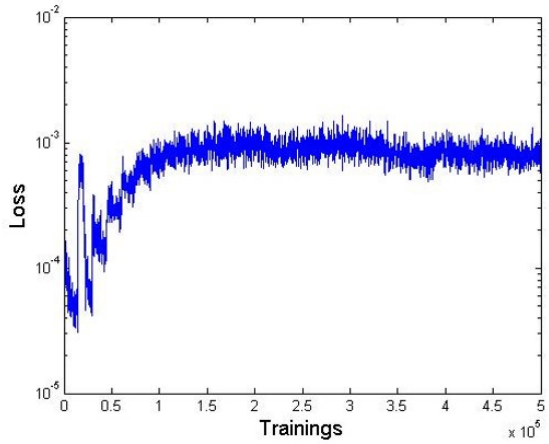


Fig. 3. Loss of the DQN during training

Another limitation can be seen by viewing a conflict scenario in which an intruder is approaching from the right at a relative heading of 90°, as seen in Figure 4.

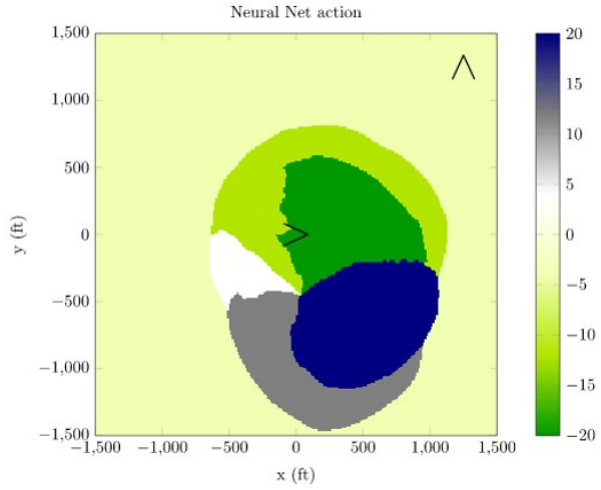


Fig. 4. Optimal policy for a 90° conflict scenario

When the intruder is in front and to the right of the ownship, within 500 meters' range, the ownship's policy is to bank right and move towards the intruder. This policy exacerbates the conflict rather than resolving it. However, this policy generates separation from the intruder as fast as possible, which will minimize its accumulated penalty. In order to deter dangerous policies such as these, the reward function may need to move away from a step or sigmoid shape in favor of a more pointed shape. This would deter the ownship from ever trying to fly through the intruder.

## VII. Conclusion and Future Work

This paper demonstrates a novel approach to generating advisory policies for airborne collision avoidance systems. This method uses deep reinforcement learning to approximate the Q values any state in the state space. This approach has the advantage that the representation of Q requires much less space in memory. In addition, continues states can be used when training the DQN. Finally, the policy produced by the DQN is generally smoother and more sensible than the policy produced by QMDP, which has pockets of very different optimal actions in areas that seem strange.

Further refinement to the DQN parameters should allow the algorithm to train more quickly. An improved reward function will discourage dangerous actions currently present in the DQN policy. Simulating different conflict scenarios in which the ownship follows the DQN policy will demonstrate the safety and performance of this policy.

After the algorithm becomes finalized, the state space can be expanded to allow for three dimensions and more realistic dynamics. The final version of this algorithm will be able to efficiently determine the optimal policy of real UAS conflict scenarios with a compact representation of Q values. This will allow UAS to incorporate automatic collision avoidance systems, unlocking the vast potential of the technology to improve society as a whole.

## REFERENCES


[1] NASA. (2015) Unmanned aircraft system (uas) traffic management (utm). http://utm.arc.nasa.gov/index.shtml. Accessed: 2016-03-10.

[2] M. J. Kochenderfer and J. Chryssanthacopoulos, "Robust airborne collision avoidance through dynamic programming," *Massachusetts Institute of Technology, Lincoln Laboratory, Project Report ATC-371*, 2011.

[3] P. Merlin. (2015) Nasa, faa, industry conduct initial sense-and-avoid test. www.nasa.gov/centers/armstrong/Features/acas_xu_paves_the_way.html. Accessed: 2016-03-10.

[4] Tech. Rep.

[5] H. Y. Ong and M. J. Kochenderfer, "Short-term conflict resolution for unmanned aircraft traffic management," in *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*. IEEE, 2015, pp. 5A4–1.

[6] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[9] S. Lange, T. Gabel, and M. Riedmiller, "Batch reinforcement learning," in *Reinforcement Learning*. Springer, 2012, pp. 45–73.

[10] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.