Training Strategy for Aircraft Collision Avoidance with Deep Reinforcement Learning

Osmany Corteguera
Dept. of EECS
MIT
osmanyc@mit.edu

Mohammad Islam
Dept. of AeroAstro
MIT
moislam@mit.edu

Alexander Joerger
Dept. of AeroAstro
MIT
ajoerger@mit.edu

***Abstract*—The aircraft collision avoidance problem consists of trying to resolve a conflict between two aircraft in the airspace to avoid a collision. This problem has seen a lot of work in a tabular reinforcement learning approach. In this work we explore deep reinforcement learning to tackle the problem. We build a new Python environment that simulates aircraft encounters in a 2D continuous plane, with an ownship and an intruder aircraft. We test how the geometry of the encounter, the frequency of collisions in the training set, and the structure of the reward function affect the policy learned. We present empirical results showing how the composition of the training data and reward structure affect the performance and training time of the policies learned.**

I. Background

The problem of airborne collision avoidance has been a topic of interest for many years. A typical metric is called a near mid-air collision (NMAC). It captures incidences where two aircrafts come within a set distance to each other. In response to airborne collisions in the US, the Traffic Collision Avoidance System (TCAS) was developed [1]. TCAS works by sensing aircraft around its airspace and issuing alerts to the pilot with suggested maneuvers to help them avoid these nearby aircraft.

TCAS, though, has many limitations. It is built as a rule-based decision system, with these rules being heuristics developed by teams of experts over years of development. The resulting system is therefore a very large base of interrelated pieces, which makes the system very hard to improve and change.

These limitations have become more important to address as the airspace rapidly changes, with new types of aircraft and higher density of aircraft in the airspace. Another system, Airborne Collision Avoidance System X (ACAS X), aims to address the limitations of TCAS and further improve on its safety. ACAS X uses a dynamic programming (DP), optimization-based approach to select the best alert to issue to the aircraft at any given moment.

The decision-making logic of ACAS X can be modeled as a Markov Decision Process (MDP), which is then solved using the Value-Iteration algorithm. The solution computed is in the form of a state-action value function, which is turned into a policy by greedily choosing the best action at each state.

A problem with this approach is that the runtime of DP scales exponentially with the number of variables in the MDP. An airborne encounter has many variables, like the position coordinates and angles of heading for the aircraft in conflict, as well as their first and second derivatives, among others. ACAS X addresses this problem by training independent decision policies for the vertical and horizontal dimensions, and combining variables in each of these policies to decrease the total number of variables present. The fact that different policies are trained means that the decisions learned are often sub-optimal. Another issue is that to use DP, the state space must be discretized, which potentially further degrades performance.

Because the Discrete Value Iteration approach is limited in its input space, it is an open question whether different approaches can perform better without that limitation. An alternative approach to DP is that of Deep Reinforcement Learning (DRL). The salient difference between DP and DRL is that DP uses large tables to store its state value function, while in DRL a neural network acts as the state value function.

The advantage of DRL is that the training time does not scale exponentially in the number of features as it does in DP, so the large number of variables inherent to an airborne encounter can be used to form a richer state representation. It is also easier to work with the continuous input features of the state because neural networks do not require inputs that are discretized. Another advantage is that neural networks have relatively light memory footprints. While logic tables from DP can take upwards to 10GB of storage, neural networks can get comparable performance with orders of magnitudes fewer parameters [2].

The downside of using DRL is that, unlike DP, there are no strong convergence guarantees. The value function learned might converge to a sub-optimal one, or it might not converge at all.

II. Research Questions

The objective of this project is to explore ways to alleviate the downsides of training a policy with DRL. Specifically, we look at how the training data we feed to the algorithm affects the learned policy. While a DP solution usually sweeps the entire state space each iteration to update a value function, we don't have the same luxury in the DRL setting, given that our state space is intractably large to sweep over.

This leaves us with the option of using either episodic data generated from a simulator, or state transitions sampled from a model of the environment. We explore the first option in this paper, since the most well-known results in DRL have come from using off-policy RL algorithms in episodic settings [3].

We seek to answer the following questions relating to training:

1) How do the training encounters affect the learned policies? In particular, are there specific training encounter generation processes that work well for DRL models?
2) How do the specified rewards affect the learned policies?

The first set of questions above are explored though various experiments that use pre-specified training encounters. These training encounters can be generated manually or via Markov chains, random sampling, or single stretches of maneuvers. An important aspect of training with pre-defined training encounters is the ratio of encounters with NMAC and without NMAC contained in the training set. The second question is studied by exploring the effects of different rewards and the effects of reward shaping on the learned policies. Intuitively, we expect the rewards to govern the learned policy. We explore these questions while considering both a discrete and continuous state space. The discrete state space provides a more straightforward implementation of our DRL logic and helped us built intuition on how the learned policies are influenced by different training parameters and strategies. The continuous state space provided a more realistic model of the airspace and served as a natural extension to our environment after our studies in the discrete space.

III. Theory

In order to generate a sequence of appropriate maneuvers to effectively resolve close encounters between aircraft, we represent encounters within a mathematical framework. We can describe an encounter as an MDP [4]. An MDP is defined by the tuple $\{S, A, T, R\}$, where

- S is the set of states s. Each member of S describes a state of the encounter, which includes information like the vertical separation, ownship vertical rate, etc. Essentially, each $s \in S$ is a snapshot of the encounter at one moment in time.
- $A(s) \mapsto A_s$ is the set of actions a that can be issued at every state. These can include climb, descend, turn right, etc.
- $T(s, a) \mapsto s'$ is the stochastic transition function from state s and action a, to state s'. This function is stochastic due in large part to the uncertainty about the accelerations applied by the intruder aircraft.
- $R(s, a) \mapsto r$ is the reward distribution from being at state s and taking action a. We use negative rewards to discourage the aircraft from engaging in NMACs, issuing too many alerts, etc. For example, having an NMAC can result in a reward of -100, while issuing an alert can result in a reward of -1.

The MDP model formalizes the idea of an agent acting in an environment where the state is not completely known. An agent acts in a MDP according to a policy, $\pi(s) \mapsto a$, which outputs an action a at any given state. The goal of the agent is to choose the sequence of actions that maximize the rewards it receives from the

environment.
This is the aim of reinforcement learning algorithms - to learn an optimal policy which will maximize the overall reward across steps for any initial state. Mathematically, this is written as

$$\text{Find } \pi^* = \text{argmax}_\pi \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) | s_0 = s\right].$$

Q-learning is one set of techniques used in reinforcement learning. Here we define a Q-function, which gives the expected reward for starting at a particular state s and taking the action a, then following policy π.

$$Q_\pi(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) | s_0 = s, a_0 = a, \pi\right]$$

The optimal policy $\pi^*(s)$ maximizes the returns, and has a corresponding Q-function denoted by $Q^*(s, a)$. Q-learning is guaranteed to find the optimal Q function in the tabular case, as long as each (s,a) pair is visited infinitely often [4].
In this work, we employ a form of parametric Q-learning known as DQN (deep Q-network). In this setting, the Q-function is modeled as a neural network for which we wish to find parameters θ such that $Q(s, a; \theta) \approx Q^*(s, a)$ (the optimal Q-function is approximated accurately). Fig. 1 shows a typical DQN architecture. One must obtain appropriate parameters

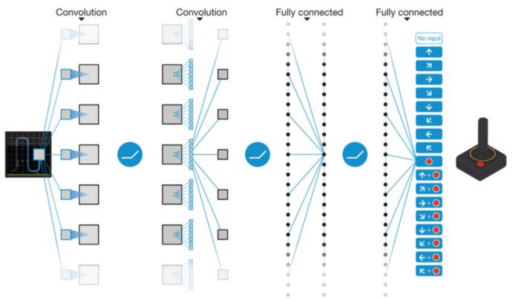

Fig. 1: Example DQN architecture [5]

to approximate the Q-function for the optimal policy correctly. This can be via a gradient descent update. The DQN algorithm can be used to find parameters such that the neural network approximates the optimal Q function, and ultimately so that the optimal policy $\pi^*(s)$ is obeyed. Many techniques are employed in conjunction with DQN to help it succeed in finding an optimal policy (experience replay, separate target networks, reward clipping, etc.).

IV. Model

In this work, we used OpenAI Baselines' implementation of DQN [6]. Our setting involves a continuous state-space and discrete action space simulation environment.
The overall aim of training is to learn policies that enable our aircraft (agent) to reach a goal location (representing an airport for instance) in as few steps as possible without colliding with any objects. Potential objects in the environment are the agent and another aircraft (intruder). The model we develop in this work provides a framework that one can use to perform training of an aircraft in discrete and continuous spaces using reinforcement learning. It provides an avenue through which we can begin to gain insight into the training process in reinforcement learning and understand the essential ingredients for effective training (i.e. good reward structure, useful training sets) so that RL can be used to learn optimal policies.
To this end, our paper focuses on training requirements for our ownship given independently moving intruders. Our metrics for assessing the efficacy of the learned policies are

- Proportion of episodes with collisions (collisions occur when agent comes within ρ_c distance of intruders),
- Time to reach goal state per episode (discrete space only), and
- Number of commands issued per episode (continuous space only).

These metrics assess whether the agent can avoid intruders, while reaching the goal as efficiently as possible. Hence, the metrics assess the learned policies. The metrics are calculated by building a static set of validation episodes, and running the learned policies in these same episodes.

V. Novelty of Work

The problem investigated in this paper is different from other applications of DRL because:

- **Successes in DRL have come from deterministic environments**
  Atari benchmarks, MuJoCo benchmarks, and 2 player perfect information games all have deterministic environments. The sources of stochasticity in these environments is usually due to the agent's own policy. Thus, there's a big weight given to exploration. In our setting, the intruder's behavior is stochastic, and it is not even clear what the patterns in it are.

- **Reward clipping is not an option**
  A strategy used in the DQN paper is to clip all Atari rewards to be either -1 or 1. The agent thus learns a reward that maximizes the frequency of rewards. This is presumably an easier value function to learn than if different reward magnitudes were used. Here, we use the different reward magnitudes to define a trade-off between safety (low collision rate) and operability (low alert rate). This difference makes our environment strictly more difficult than the environment solved by DQN in its Atari benchmarks.
- **No pre-defined simulation environment**
  If we ran this on an already-built simulation environment, collecting data would have been trivial, since all we have to do is run the environment itself. All that is left to modify is the agent's own policy so that a good balance of exploration vs exploitation is reached. In our problem setting, we are building the simulation environment ourselves, and thus defining the the environment's behavior. This adds more degrees of freedom to our learning objective (what intruder behavior are we optimizing for? what intruder behavior do we train on?).

VI. Discrete State Space

A. Approach

As a proof of concept, we use an OpenAI Baselines implementation of DQN in order to learn a policy to reach our goal while avoiding a collision with the intruder aircraft in a two-dimensional discrete grid world environment. Fig. 2 gives a cartoon of a grid world. The state space consists of absolute coordinates.

We report a subset of the results of training

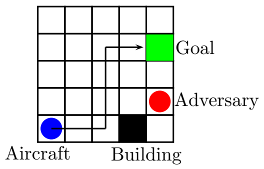

Fig. 2: Cartoon of grid world environment

on a 10 by 10 units grid space. The agent can perform four different actions (movement in the four cardinal directions). In DRL, training data is generated by performing episodes using the environment (there is no data *a priori*). As episodes are performed, the agent learns how to best navigate the environment in order to maximize its own reward.

We treat the number of agent training steps and the reward structure as hyperparameters for the training of our DQN approach. A negative reward is awarded to the agent for each step taken. Collisions with the intruder also entail a penalty. This reward structure is meant to motivate the agent to reach the goal in the shortest possible time while avoiding collisions with the intruder.

The intruder acts independently from the agent (*"at random"*). Two encounter types have been considered for the discrete environment:

1) The motion of the intruder is restricted towards a certain direction, e.g., from top/right to bottom/left of the grid world. However, the specific path is not prescribed, but chosen at random by the intruder. Hence, each episode features a different intruder behavior. Once the intruder reaches the edge of the grid world, it re-spawns at its initial position.
2) The intruder can move freely at random without any restriction of its motion. Here, we designed the experiment to determine how much exploration was needed for the agent to learn an effective policy. To this end, we trained policies where the agent would greedily select an action with respect to its learned state-action function with probability $1 - p$, and with probability p select an action uniformly at random. We refer to this as a p-greedy training policy.

B. Experimental Results

1) Restricted Intruder Motion Training Policy: After training under this encounter type, the agent consistently moves towards the goal while avoiding the intruder. Fig. 3 shows that the learned policies converge towards an optimal policy. As a reference, the reward for each taken step is -1. Hence, the optimal policy has a reward of -19 as it takes 19 steps to reach the goal in the chosen setup (going from top left to bottom right of the grid).

The number of steps is sufficient for all investigated reward training to level out to their long-term value. Hence, we can compare the effects of rewards on the learning rate. The number of performed episodes increases with a larger penalty for NMACs. More episodes correspond to more training epochs. We expect that more performed episodes with the same number of total steps corresponds to more efficient training.

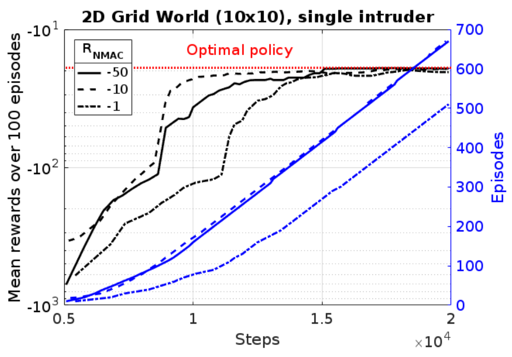

Fig. 3: Learning curve on 2D grid world environment (10x10): Rewards for NMAC with intruder (negative) affect training speed

We can also see that the chosen rewards for NMACs affect the rate of learning. The rewards vs. steps gradient is larger for NMAC-penalties of -10 and -50 than for -1. As a result, a near-optimal policy is learned more quickly.
However, there is an optimal reward. The training efficiency decreases if the negative rewards become too large, i.e., the learning curve for $R_{NMAC} = -50$ in Fig. 3 reaches the near-optimal policy more slowly than the training with $R_{NMAC} = -10$.

2) p-Greedy Training Policy: After training, the agent consistently moves towards the goal while avoiding the intruder. To test the performance of our learned policy, we developed a set of 1000 validation tests. In these tests the motion of the intruder aircraft in our environment is completely specified. This gives us a basis to test the efficacy of different policies learned by the aircraft during our training process. Our main metrics here are the collision rate (proportion of episodes with collisions) and episode length (encapsulating our previous metrics of number of commands issued and time to reach goal state). We explore how these metrics change as our probability p changes.
As we can see in Table I, the value of p did not have an effect on the number of collisions. The policies learned were all able to avoid the intruder when run on our validation set of encounters. The average episode lengths tell a similar story. Most policies were optimal in this metric. The one outlier is most likely due to variance in the policy learned (we only ran one seed per p value), rather than an actual trend.

	Exploration Probability					
	0	0.01	0.05	0.1	0.2	0.3
Collision Rate	0	0	0	0	0	0
Avg. Steps	19	19	135	19	19	19

TABLE I: p-Greedy Training Policy: Performance metrics for each setting of exploration

VII. Lessons Learned from Discrete State Space

Our initial experiments show that DRL can be used to teach the aircraft to achieve our objective. We observe that the reward structure heavily influences the learning of an optimal policy by the aircraft. This learning behavior motivated our research questions regarding training and drove us to explore different reward values for different actions (reaching the goal, taking a step, crashing into an aircraft). If the rewards are ill-conditioned, the agent might ignore the long-term goal of reaching the target location all together and simply learn a policy to stay far away from the intruder without moving towards the goal.
We found that the agent performed well using its learned policy when facing an intruder if the intruder behavior was observed during training. However, when facing an intruder behavior that had not been observed prior to validation (e.g. had not been observed in the training set), the agent begins to prioritize avoiding the aircraft over reaching the goal.

VIII. 2D Continuous State Space

A. Setup

After learning optimal policies for the simple grid world environment, we decided to build an environment that better approximates the dynamics of a real airborne encounter.
Our setup involves the following scenario in which there is a close encounter between the agent and an intruder. Our goal is to avoid the encounter while issuing as few maneuvers as possible. Unlike the grid world setting, there is no longer a distinct goal position. This is similar to an aircraft encounter in a real life, where the principal goal is to clear the conflict. After the conflict is cleared, the aircraft can resume its desired flight path. The updated setup better reflects this real-life scenario.
We assume that there is no coordination between the agent and the intruder. The two aircraft cannot communicate, so the agent

needs to be prepared to avoid any type of behavior by the intruder. There has been some work into ways in which aircraft can coordinate their maneuvers by communicating through messages [7], which could be used to extend the solution we build in this project to cooperative scenarios.
Both the agent and the intruder are advanced at each timestep through linear dynamics, so that turns can occur instantaneously. We limit the rate at which the aircraft can turn to 3 deg/s, so that maneuvers need to be taken far in advance of when a collision is expected in order to avoid it. This lag between command and execution models the system inertia of a real-life scenario.
The transition kernel for an airborne encounter is difficult to estimate. Previous approaches have used sets of aircraft encounters in the US airspace gathered from radar, and used that data to build a directed graphical model that approximates the transition kernel [8]. In this work, we seek to understand whether a variety of other encounter geometries can achieve good performance. The transition kernel in our approach is implicitly defined by the environment and by the intruder behavior. Further work can be done with our environment setup by adding white noise to the state measurements after each transition.

B. *Approach*

The conducted experiments were designed to answer our research questions. How can we best design a training set of data to learn a good policy quickly? Three questions were addressed in detail:

1) **How do different intruder behaviors affect the policy learned?**
Intuitively, we want the intruder behavior to be a superset of the real behavior that we would encounter in the real airspace, so that the agent can effectively deal with any behavior robustly. To this end, we designed an experiment where we trained policies with different sets of encounters:
    - **Sticky policy** where the intruder randomly chooses and sticks to an action for some length of time, then switches to a different action,
    - **Random** where the intruder takes a random action at every timestep,
    - **Single action** where the intruder starts out with an action for a random length of time and transitions to no action for the rest of the environment, and
    - **None** where the intruder takes no action and flies straight for the whole encounter.
2) **What is the best ratio of NMAC to no NMAC to have in the training set?**
If we have too many NMACs, then the agent will learn to always alert, while having too few NMACs will make the agent ignore threats. We want to figure out, given a set reward structure, which ratio is best.
3) **How do the rewards affect the policies learned?**
Ng, Harada, and Russell [9] introduce the concept of shaping rewards that preserve optimal policies. They prove that if rewards are based off the difference between a potential function over states, then the optimal policy can still be learned, but learning can be faster if the reward provides a good heuristic. We evaluate the impact of a shaping reward, using the distance between the aircraft as the potential function. This approach has the effect of penalizing the aircraft for getting close and rewarding them for flying apart. We define $\phi(s) = d$, where s is a state and d is the distance between aircraft. The shaping reward added at each step is then $r'(s_t, s_{t+1}) = \gamma\phi(s_{t+1} - \phi(s_t))$, where γ is the discount factor.

The way we conduct these experiments is by generating actions for the intruder according to a Markov chain. The Markov chain has 3 states, corresponding to the possible actions an intruder can take: (NONE, LEFT, RIGHT) - shown in Fig. 4. We modified the transition probabilities depending on what kind of encounter we wanted (see Appendix A for details).

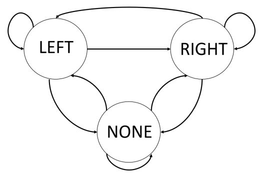

Fig. 4: Markov Chain for possible intruder actions

A time of closest approach t_{TCA} is chosen for when the aircraft will have a collision, and the starting parameters of the encounter are adjusted so that the aircraft collide at t_{TCA}. Then, we randomly perturb the position of the intruder by some a random number of units in a random direction. The size of the perturbation is chosen uniformly at random from the range $(0, d)$. The larger we make d, the fewer NMACS we will have in our set of encounters.

C. *Experiments*

We first created a base set of hyperparameters for the DQN algorithm. The hyperparameters are recorded in Appendix B. This base set policy was tuned by trying different architectures and reward parameters until the learned policy was qualitatively reasonable. The following experiments are aimed at modifying this base setup and measuring what effect this has on the performance of the learned policy. The base policy was training against an intruder that took random actions at each timestep.
We evaluate the performance of a policy by:

1) **P(NMAC)** Proportion of encounters that had an NMAC. We define an NMAC to occur when the aircraft get closer than 500 units of each other. The lower our P(NMAC) the better.
2) **P(Alert)** Proportion of time with alerts per episode. We want to issue as few alerts as possible, while issuing enough alerts to avoid NMACs. This creates a tension between P(NMAC) and P(Alert).
3) **P(Reversal)** Proportion of alerts that are reversals of the previous alert. For example, if the previous alert was a RIGHT, issuing a LEFT next is a reversal. We want to keep this number as low as possible.

We ran the following experiments:

1) Intruders with none, sticky, single action, and random policy compared with the base policy and the control where no action is taken.
2) Training set of encounters with (5, 10, 15, 25, 50) percent of encounters including an NMAC.
3) Weights on the shaping reward of (1e-4, 3e-4, 1e-3, 3e-2, 1e-2).

We made a set of 5000 encounters to evaluate each learned policy. This set of encounters is composed of $1/4$ of each encounter (no action, single action, sticky, and random). Fig. 5-7 illustrates the performance of each of the three experiments with respect to the proportion of encounters that had a NMAC (fewer NMACs are better). The other performance metrics (proportion of alerts and reversals) for the three experiments are included in Appendix C. We also show our learning curves as measured by the reward per episode as a function of the number of training steps taken for the three experiments in Fig. 8-10. The results indicate that ”random” and ”sticky” intruder behaviors perform best for training a policy. Similarly, the values in the mid-range of NMAC proportion and reward shaping weights seemed to work best. We think that putting these improvements together and testing how they interact is a viable path for further work. Furthermore, the learning curves indicate that the rewards are increasing as training is performed, which is a good sign that the agent is learning.

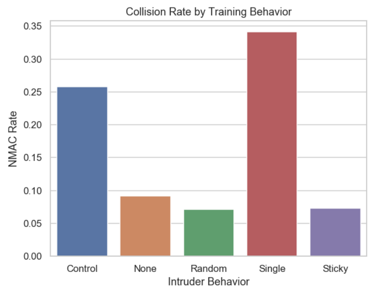

Fig. 5: Percentage of NMACs observed using policies learned from training with different intruder behaviors

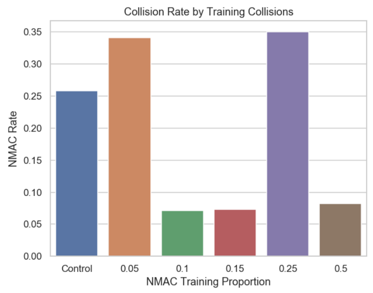

Fig. 6: Percentage of NMACs observed using policies learned from training set containing various proportions of NMACs themselves

IX. 3D Continuous State Space

We limit the dynamics to the horizontal 2D plane for three reasons:

1) The state space is large enough that any approaches are already very computationally taxing. Generally, the larger the state space, the longer it takes the policy to

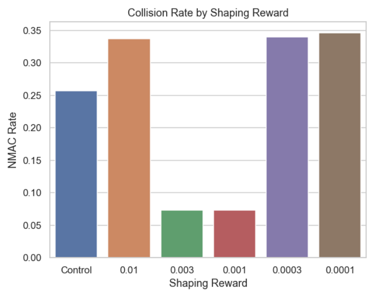

Fig. 7: Percentage of NMACs observed using policies learned from training with different reward shaping weights

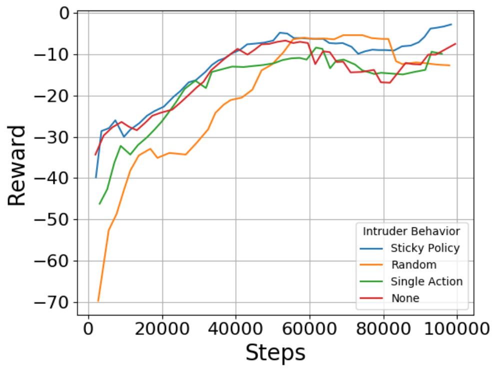

Fig. 8: Learning Curve for different intruder behaviors

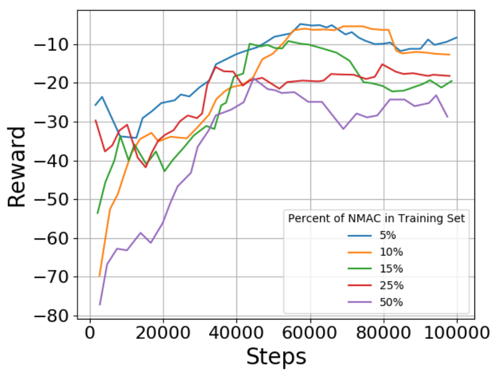

Fig. 9: Learning Curve for different percentages of NMAC encounters in the training set

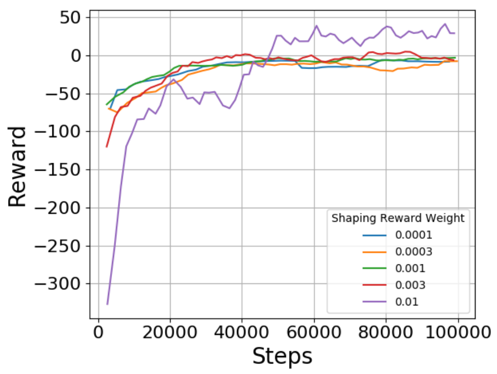

Fig. 10: Learning Curve for different reward shaping weight parameters

converge. We had a lot of experiments to run with very low computational resources, which was a big detractor from using 3D.

2) The 2D state space is very similar to the 3D state space. We just need to add a few more state variables and actions to get a 3D state-space. The lessons we learned from reward shaping, encounter geometries, and NMAC proportions can be carried over directly to the 3D space with minor modifications.
3) There are fewer parameters to adjust than in the 3D state space (fewer actions, smaller value function neural network, etc.). With larger state and action spaces, there is more variance in the policies learned, and more hyperparameters and rewards to tune. We decided that the time spent tuning hyperparameters was not worth the small gains we would get by moving to 3D.

X. Conclusion

In this work, we have presented findings on the training of aircraft avoidance collision system in a discrete and continuous space. We contributed a novel Python environment in which to simulate aircraft, which can be used in future work to research the aircraft collision avoidance problem. We also presented empirical results comparing the effects of different simulated training data on the performance of the collision avoidance policies.

XI. Collaboration

Osmany was the project leader and guided the research and coding. He created the grid world and continuous space simulation environments, and designed and coded the experiments for the continuous state-space. Mohammad worked on the visualization of the grid-world environment, and the creation of validation encounters for the gridworld environment. As our project focus shifted, most of his work did not make it into the report. Alexander focused on the gridworld environment and policy sensitivity to rewards in this environment.

All team members participated in the discussion, supported the model training, and contributed to the milestones, the poster, and the final report.

References

[1] Mykel J. Kochenderfer and James P. Chryssanthacopoulos. "Robust Airborne Collision Avoidance through Dynamic Programming". In: 2011.

[2] Kyle Julian et al. "Policy compression for aircraft collision avoidance systems". In: Sept. 2016, pp. 1–10. DOI: 10.1109/DASC.2016.7778091.

[3] Zhuora Yang, Yuchen Xie, and Zhaoran Wang. "A theoretical analysis of deep Q-learning". In: *arXiv preprint arXiv:1901.00137* (2019).

[4] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: The MIT Press, 2018.

[5] Volodymyr Mnih et al. *Mnih2015*. doi:10.1038/nature14236. 2015.

[6] Prafulla Dhariwal et al. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[7] Rachael E Tompa et al. "Collision avoidance for unmanned aircraft using coordination tables". In: *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE. 2016, pp. 1–9.

[8] Andrew J Weinert et al. *Uncorrelated encounter model of the national airspace system, version 2.0*. Tech. rep. Massachusetts Inst of Tech Lexington Lincoln Lab, 2013.

[9] Andrew Y Ng, Daishi Harada, and Stuart Russell. "Policy invariance under reward transformations: Theory and application to reward shaping". In: *ICML*. Vol. 99. 1999, pp. 278–287.

Appendix

A. Transition Matrices for MDP

The transition probabilities are modified dependent on what kind of encounter is desired. The transition matrices for each of the types of encounters we generated are provided below (Rows numbers to actions are 0: NONE, 1: LEFT, 2: RIGHT):

Random

$$\begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}$$

One Act

$$\begin{bmatrix} 1 & 0 & 0 \\ 1/25 & 24/25 & 0 \\ 1/25 & 0 & 24/25 \end{bmatrix}$$

Sticky

$$\begin{bmatrix} 14/15 & 1/30 & 1/30 \\ 1/30 & 14/15 & 1/30 \\ 1/30 & 1/30 & 14/15 \end{bmatrix}$$

No Act

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

B. Hyperparameters

- Total timesteps: 100,000
- Optimizer: Adam
- Learning Rate: 0.0005
- Experience replay buffer size: 30,000
- Q-network structure: 3 hidden layers of 64 ReLU units each.
- Discount factor γ: 0.99
- Target network update frequency: 500
- Exploration factor ϵ: 1 to 0.01 annealed linearly over the first 30,000 timesteps.
- Batch size: 64
- Training frequency: Every step.

C. Policy Performance

Intruder behavior experiment

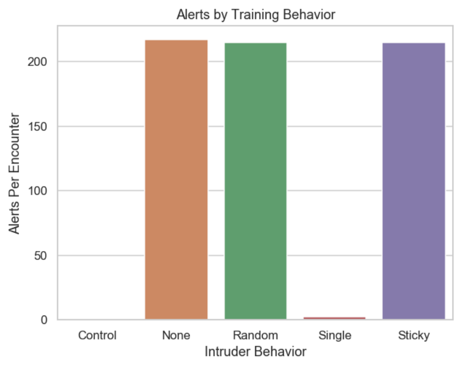

Fig. 11: Percentage of alerts observed using policies learned from training with different intruder behaviors

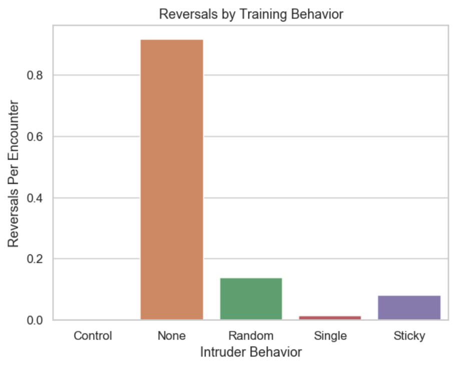

Fig. 12: Percentage of reversals observed using policies learned from training with different intruder behaviors

Percent of NMAC encounters in training set experiment

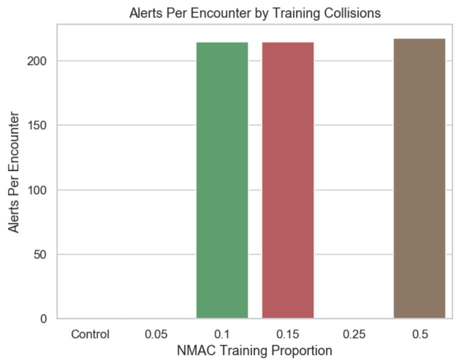

Fig. 13: Percentage of alerts observed using policies learned from training set containing various proportions of NMACs themselves

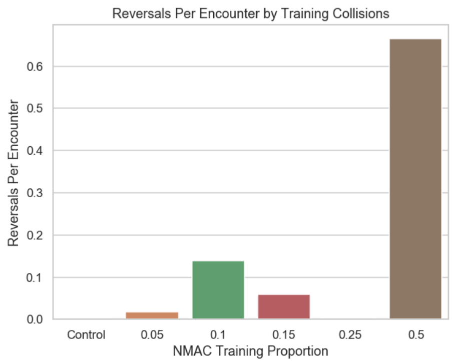

Fig. 14: Percentage of reversals observed using policies learned from training set containing various proportions of NMACs themselves

Reward shaping experiment

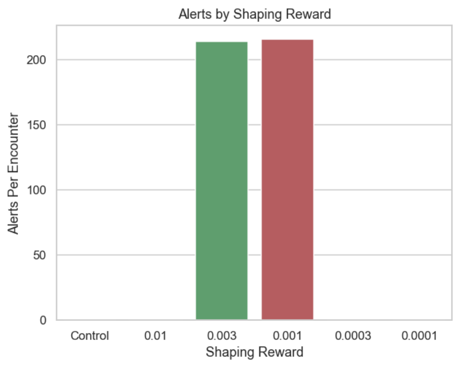

Fig. 15: Percentage of alerts observed using policies learned from training with different reward shaping weights

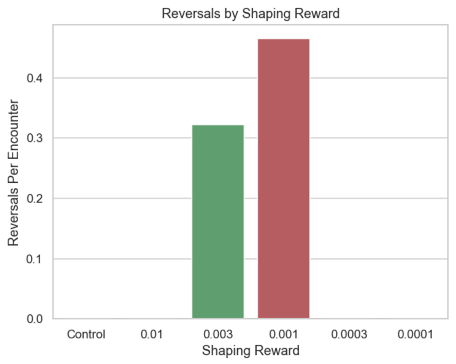

Fig. 16: Percentage of reversals observed using policies learned from training with different reward shaping weights