FoxNet: A Deep-Learning Agent for Nintendo's Star Fox 64

Kevin Looby
Stanford University
Stanford, CA
klooby@stanford.edu

Joshua Grinberg
Stanford University
Stanford, CA
jgrinber@stanford.edu

Kat Gregory
Stanford University
Stanford, CA
katg@stanford.edu

Abstract

In this paper, we explore a number of models and learning strategies for autonomous aircraft navigation and combat through the simplified environment of the Nintendo video game, Star Fox 64. Our primary model builds off DeepMind's 2015 DQN model. We first compare the performance of this model and three other neural architectures in an on-policy classification task, achieving up to 71% validation accuracy with the DeepMind DQN model. We then improve these models with off-policy Deep Q-Learning. After 80,000 training iterations on Level 1, our best agent achieves an end-of-game score of 61 points, surpassing a random baseline's score of 12 but falling short of a human player's score of 115. We also demonstrate our agents' abilities to generalize game strategy to unseen levels. After 140,000 training iterations in "Train Mode", our best agent achieves an end-of-game score of 63 after only 10,000 additional training iterations on Level 1.

Figure 1. Sample training image from Star Fox 64. We extract the score (in this case, 10) and the health (max of 10) from the pixels.

1. Introduction

Autonomous vehicle navigation is a common problem in the field of artificial intelligence control. Video games provide an excellent framework for testing and evaluating control models, as they represent an encompassing simulation environment that approximates realistic vehicle navigation in repeatable training scenarios. We present a deep learning model that pilots a video game aircraft through a hostile environment, seeking both to eliminate adversarial agents and navigate past obstacles.

We believe that developing such an agent is valuable beyond the scope of the video game itself because of potential applications in fields that include autonomous airborne vehicles. For example, training a military drone in a simulated environment using the methods described in this paper could serve as an effective warm-start before training the drone in the real world.

1.1. Game Description

We chose to focus on Star Fox 64, a popular 3D scrolling shooter game originally released on the Nintendo 64 console in 1997. The player controls a one-person aerial combat vehicle called Arwing. Each level, or "mission," begins with a phase in which the player must destroy or navigate past a series of enemies and obstacles. The second phase of each level consists of a "boss fight" in which the player and its allies must destroy a single, powerful enemy agent. We focus only on the first level.

The player's movement is restricted to navigation within a 2D plane (i.e., the player can move up, down, left, and right), which travels through space at a fixed speed Each mission is populated with numerous enemy crafts that attack or obstruct the progress of the player. In addition to the four navigational commands, the player can fire its semi-automatic primary weapon at these enemy agents. Each eliminated enemy rewards the player with points that contribute to an accumulated score across all completed missions in a single game. If the player itself comes under enemy fire or fails to avoid obstacles, its health decreases until it loses a life. After three lives, the game ends.

1.2. Objective

We aim to develop an agent that can outperform an experienced human player, as measured by the final score. At every time step in the game, given an input frame from the video game as represented by a 2D pixel array, our model

uses a CNN to output one of six actions for the agent to take. We consider the game to be over if the agent loses all three lives or completes the first level.

2. Related Work

Hubel and Wiesel's 1959 revelations on the layered architecture of neurons in cats' primary visual cortex [6] inspired the use of convolutional neural networks (CNNs) to perform similar pattern recognition for computer vision. CNNs have since become one of the dominant tools in the field. Our work is inspired by recent advancements in a number of different research areas related to computer vision. Below, we present relevant research in each.

2.1. Imitation Learning

Imitation learning involves training a model to imitate a set of "correct" decisions in a classification setting. For example, in "Deep Neural Network for Real-Time Autonomous Indoor Navigation" [7], Kim and Chen train a drone equipped with a single camera to navigate through hallways in order to find a target object, predicting from each frame which action to take. Training data was obtained primarily by recording the actions of an expert human drone pilot. This imitation learning approach performs well and is dramatically more efficient than other approaches to locating points in 3D spaces, such as SLAM and Stereo Vision. The first approach to our own project similarly frames the game as a classification task.

2.2. Reinforcement Learning

While classification tasks are interesting, recent research suggests that reinforcement learning has the potential to develop even more powerful systems. As Sutton and Barto describe in their overview of the field, reinforcement learning, agents learn a policy for how to operate in an environment, optimizing not for consistency with an existing policy (as in a classification task) but instead for maximizing reward [16]. Chris Watkins introduced the Q-learning algorithm in 1989 as a method to find the optimal policy for action selection in a controlled Markovian environment [19].

Bellemare et al. created the Arcade Learning Environment, a framework to interact with Atari games, in 2013 as a way to evaluate the performance of different reinforcement learning agents [1]. Mnih et al.'s 2015 paper "Playing Atari with Deep Reinforcement Learning" [11], in presenting an agent for this environment, was the first to combine Q-learning with convolutional neural networks. Although the same authors later demonstrated in their paper "Asynchronous Methods for Deep Reinforcement Learning" [10] that Policy Gradients can actually outperform Q-learning, Deep Q-learning continues to be a popular approach. DeepMind Technology's paper "Human-level control through deep reinforcement learning" [12], took this research a step further, presenting a model trained with Deep Q-learning that matches the performance of a professional game player across 49 different Atari games. A number of recent papers have noted performance boosts from a variety of DQN variations: Stadie et al. proposed new exploration strategies [15], van Hasselt et al. pioneered Double Q-learning [17], Schaul et al. explored experience replay buffers [13], and Wang et al. originated a dueling network architecture [18]. These models for Deep Q-learning formed the primary inspiration for the second part of our project.

2.3. Navigational agents

While the performance and the adaptability of these model is impressive, we were interested in whether models could adapt to more varied and realistic visual environments than that of Atari games. We thus looked into several navigational agents.

Given the raw pixels of the scene ahead of a real-life car, Bojarski et al's 2016 paper "End to End Learning for Self-Driving Cars" [2] uses a CNN without any reinforcement learning component to steer the car on local roads.

Dewing and Tong's project "Now this is podracing - driving with neural networks" [4], uses a deep convolutional network and reinforcement learning to train an autonomous driving agent to play the Star Wars Episode I Podracer racing game, which involves a significantly more complex visual environment than the Atari games.

2.4. Visualizing Networks

In "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps" [14], Simonyan et al. present a method for visualizing image classification models by producing a class saliency map for a specific output class and input image. Taking the gradient of the loss for a given class with respect to the pixels of an image helps highlight the most significant regions of the image. We employ this technique in order to get a sense of how well our models learn to detect game-specific components, such as enemy ships and obstacles that must be avoided.

3. Dataset

Below we explain how we interface with the video game and describe the data we collected and processed.

3.1. Emulator

We use an emulator, software that simulates a video game console's hardware, to run Star Fox 64 on a computer rather than on the original Nintendo 64 console. We chose to use the mupen64plus emulator, which is free, open source, and allows us to interface with the game via TCP sockets. At each time step, the emulator sends the current video frame to our model and waits for our trained agent to

give it a command to perform at that time step. The framework for enabling this connectivity was graciously provided by Alexander Dewing and Xiaonan Tong [4].

3.2. Data Collection

We acquired training data by recording human players over the course of many games. Twenty times per second, we save a snapshot of the video game screen as a single image frame of size 640×480 pixels. We label the sample with the alphanumeric key corresponding to the action the human took at that timestep. The human player's action space is limited to six primary actions: do nothing ("noop"); move up, left, down, or right; and fire weapon. In addition, there is a seventh action used for selecting options in menus and skipping cut-scenes.

The final dataset we used to train our model consists of 16,828 samples. This represents nearly 40 minutes of game play encompassing 5 successful passes through the level, with no lives lost. Figure 2 overviews the distribution of samples.

Action	Label	Count
up	`w`	2292
left	`a`	3989
down	`s`	2109
right	`d`	4946
fire	`j`	1990
do nothing	`l`	1502

Figure 2. Available in-game actions, corresponding labels, and counts of each action in our final dataset.

Note that this dataset is only used in the classification part of our project, as online Q-learning does not require an existing dataset.

3.3. Data Processing

3.3.1 State Representation

Our default state is defined as the pixels in one frame. Since each frame is a 480×640 pixel image with 3 color channels, this is a $480 \times 640 \times 3$ tensor.

We also explored the use of a MultiFrame state consisting of a stack of three consecutive frames. This more complex representation allows our model learn dynamic features such as the velocity and acceleration of objects moving across the screen. A MultiFrame state is a $3 \times 480 \times 640 \times 3 \times 3$ tensor.

3.3.2 Subsampling

One episode of the game, lasting 5-6 minutes, produces 6,000 to 7,200 frames. Close to 70% of these frames have "no-op" (no operation) actions. In order to increase our data efficiency, we subsample the recorded frames by discarding each "no-op" frame with a 95% chance. The result is roughly 3,000 to 4,000 frames for 5 to 6 minutes of play.

3.3.3 Image Compression

Figure 3. An original 480x640 frame (left) compared to a compressed 48x64 frame (right).

Since the number of parameters in our neural network models increases dramatically with the number of input pixels, we compress each frame to a 48×64 pixel image using bilinear interpolation using the `misc.imresize` function from the `scipy` library. Although the compressed frames have only 1% of the original number of pixels, they retain enough resolution to identify important features of the game, such as enemy ships and incoming lasers (see Figure 3).

3.4. Reward Inference

Our second approach, Q-learning, requires that we need to know the reward associated with being in a particular state. We quantify the reward of a given state as a summation of the current score (>0) and the agent's health (0-10): `reward = game_score + agent_health`. Below we describe how we calculate these components.

3.4.1 Score Extraction

Each frame of the game displays the agent's score in the top left corner. In order to extract this value, we implemented a simple optical character reader (OCR). Since the score is always displayed in three digits at the same pixel coordinates on the screen, the reward extractor crops each digit to a 31×26 pixel image, then classifies each digit independently. Classification is performed using a nearest-neighbor model with only 10 training data points (i.e. one template image per digit). Each pixel is treated as its own feature. We tested the model using 1,340 frames. When using all three color channels and inner-product as the similarity metric, accuracy fell short of 90%. However, by using only the blue color channel and Pearson correlation as the similarity metric, we achieved an accuracy of 100%. Perfect performance is not surprising because the only variability in the test data comes from discoloration in the background and small amounts of occlusion in the foreground.

One of the challenges involved with reward extraction was that the transition between score values when the agent

kills an opponent is not reflected instantaneously; instead, the number appears to flip around sideways in 3D space. Since the digits appear squeezed during this animation, our extractor performed poorly on these frames, achieving an accuracy below 25%. In order to handle these frames, if the maximum Pearson correlation classification value for a digit is less than 0.9, we return the score extracted from the previous frame. This method achieves an accuracy of 100% on transition frames.

3.4.2 Health Extraction

Similarly, we extract the health by calculating the fraction of the health bar, located in the top right corner, that is filled in. We represent the agent's health as a float between 0 and 1, with 1 being peak health, and scale it by a hyperparameter `health_weight`.

Another challenge is that health and score are sometimes obscured by explosions on screen. When health or score are occluded, we simply return the previous value.

3.4.3 Train-Test-Validation Split

In total, we have 5 episodes consisting of a total of 16,828 frames. We randomly selected 10% of the frames from each episode to use as validation data.

3.5. Approach

We framed this problem in two ways - as a classification problem and as a Q-learning challenge - and will describe both approaches.

4. Classification Task to Compare Models

We developed and evaluated four different models with on-policy classification, optimizing for consistency with a human players actions. Below is a brief description of each of the four models we evaluated.

4.1. Models

4.1.1 Linear

Our baseline was a fully connected, two-layer neural network with 1024 nodes in the hidden affine layer and ReLU activations. It outputs a probability distribution over the 7 possible actions.

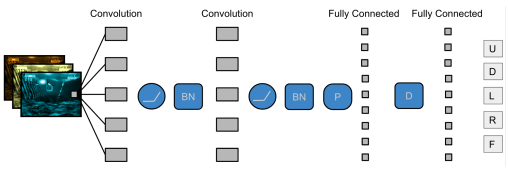

Figure 4. Fox CNN model architecture.

4.1.2 Fox CNN

Our first CNN model, illustrated in Figure 4 has two convolutional layers, each followed by ReLU activation and spatial batch normalization. Both layers have 32 filters: the first layer has a kernel size of 7, while the second has a kernel size of 9. The output of the convolutional layers undergoes max pooling with a kernel of size 2 and a stride of 2. It is then flattened and fed into a two-layer neural network with 50% dropout after a 1024 node hidden layer.

4.1.3 DeepMind DQN

We also implemented a model based on the DQN published by Google DeepMind [12]. This model has three convolutional layers with ReLU activations but no batch normalizations. The three layers have 32, 64, and 32 filters each, with kernel sizes 8, 4, and 3, respectively. The output is flattened and fed into a two-layer neural network with a 512 node hidden layer, and it is not subject to dropout.

4.1.4 DeepMind DQN MultiFrame

Our final model is a version of the DeepMind DQN model that uses 3D convolutions over MultiFrame states (composed of the frame in question and its two predecessors) to incorporate temporal information.

4.2. Methods

```
model.train(states_train, actions_train)
actions = model.predict(states_eval)
accuracy = sum(actions==actions_eval)/len(actions)
```

Figure 5. Pseudocode for computation of classification accuracy.

The goal of the classification task is for the model's predicted action for a given state to correspond to the action taken by a human player in the same state. To accomplish this, we perform stochastic gradient descent with Adam optimization to minimize the softmax cross entropy loss between the model's probability distribution over different actions and the human player's actual action.

4.3. Experiments

For each of our experiments, we iterate over the training dataset in minibatches of size 100 for up to 20 epochs. After each epoch (which represents a full pass through the training data), we perform incremental validation using the validation dataset.

Our experiments use an initial learning rate of $4e{-}6$ combined with Adam optimization to compute an adaptive learning rate for each parameter that anneals over time. To reduce overfitting, we incorporate L2 regularization into our loss with a regularization lambda of 0.001. Although we do

Figure 6. Training and validation accuracy of Fox CNN with dropout (right) and without (left).

Model	Train	Validation
Linear	64%	41%
Fox CNN	84%	35%
DQN	93%	**71%**
DQN MultiFrame	**100%**	48%

Figure 7. Accuracies of four models after 20 epochs of offline training.

not perform dropout on the DQN or DQN MultiFrame models, our 50% dropout when training the Fox CNN model has an important role in reducing the gap between training and validation accuracies, as demonstrated in Figure 6.

4.4. Results

Figure 7 compares the training and validation accuracies of the four models trained offline to perform classification.

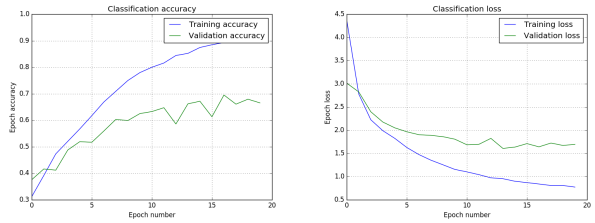

Figure 8. Accuracy and loss of DeepMind's DQN Model for offline classification.

The performance of the DeepMind DQN model is impressive: it achieves a validation accuracy over five times what we would expect from a random player, and its accuracy and loss graphs shown in 8 demonstrate a relatively smooth increase in accuracy and decrease in loss over epochs, as well as a reasonable gap between training and validation accuracy.

In contrast, the lower training and validation accuracies of the Linear model and Fox CNN suggest that both models lack the complexity to handle the task. The DQN model has more convolutional layers, which create more nonlinearities, than the Linear and Fox CNN models. Theoretically, it should be able to perform at least as well as shallower models.

The DeepMind DQN MultiFrame model seems to have the opposite problem. It is powerful enough to completely memorize the training set, but overfits and thus does not perform as well as the single frame version on the validation set. We believe that the model shows promise but acknowledge that it would require hyperparameter optimization to reduce overfitting, as well as more training time and data to achieve this potential.

4.5. Confusion Matrix

The confusion matrix in Figure 9 visualizes the performance of the DQN model against the validation set. This model achieves 71% accuracy, which is reflected in the strong diagonal of correctly predicted actions. It has the highest recall - over 81% - for the action 'do nothing', closely followed by the action 'right', and is particularly good at identifying directional actions, like 'right' and 'left' and, to a lesser extent, 'up' and 'down'. In contrast, it struggles most with the action 'fire', achieving under 30% recall and predicting 'left' nearly as often as 'fire' in situations where it ought to do the latter. The may reflect how objects that signal dodging are larger and more persistent than the small enemy planes that signal firing.

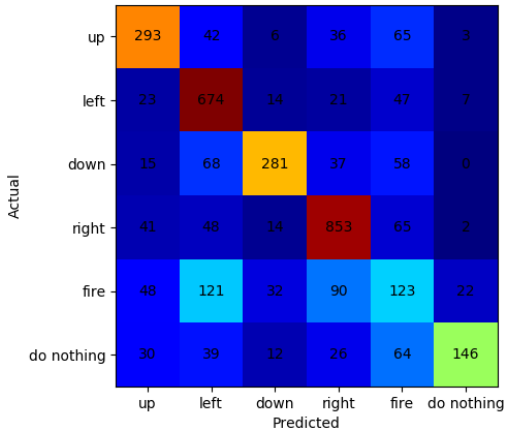

Figure 9. Noramlized confusion matrix of our best performing DQN model evaluated on the validation set.

4.6. Saliency Maps

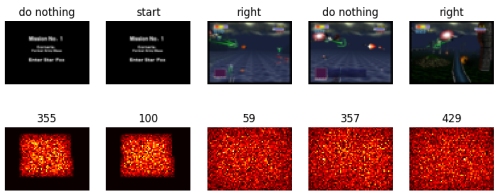

Figure 10. Saliency maps.

The saliency maps in Figure 10 indicate that though our classification models do prioritize the center of the screen, they have yet to identify the most relevant features of a given frame, such as the enemy planes or the upcoming obstacles. However, that they focus on a different area when in a menu state than when in game play suggests that the models are learning and that more training time may improve their ability to identify salient features from their surroundings.

Further, the saliency maps demonstrate the difficulty of the classification task: The image pairs 355 & 100 and 59 & 357, for example, look nearly identical but are labeled with different actions. We sample frames from the game at 20 frames per second. As the player transitions from one action to the next, this high frame rate may capture consecutive images that are visually nearly identical but are labeled with different actions. This increases the difficulty of the task of identifying the correct action.

4.7. Weights Visualization

In Figure 11, we show a visualization of the 32 8×8 filter weights in the first convolutional layer of the DQN model trained for classification. Some interesting features can be seen in many of the weights, including small regions of similar shapes (2) and lines (13). We expect that more training iterations would result in more clearly defined features.

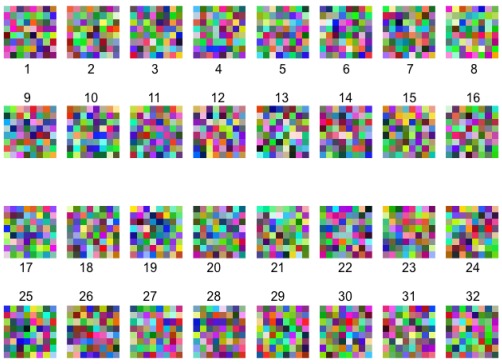

Figure 11. Visualization of the weights of the first convolutional layer of the DQN model.

5. Deep Q-Learning for Online Learning

The best model trained through our classification task above can only hope to match the performance of the expert human player it was trained to emulate. However, our stated objective was to create an agent that can *outperform* a human player. To pursue that goal, we turned from on-policy learning to off-policy learning.

5.1. Methods

With off-policy Deep Q-learning, we evaluate performance based on the agents max score for the game, regardless of the intermediate actions it takes to arrive at that score. Conceptually, this makes sense: an agent can perform well even if it does not follow the same policy our human player used; indeed, it might discover an even more successful policy. The goal of Q-learning is therefore to use numeric reward signals to improve a game-playing policy over time, optimizing, in our case, for the maximum end-of-game score.

```
init state, Q
while True:
    Pick action a for state using e-greedy
    Observe reward r, new_state s'
    target = r + max_a' Q(s',a')
    loss = 0.99(target - Q(s,a))^2
```

Figure 12. Pseudocode for Q-learning.

To do this, we aim to learn an optimal Q-value function $Q*(s,a)$, which gives us the maximum expected cumulative reward possible when taking action a from state s. Figure 12 outlines the basic algorithm. The extraction of reward r from an image frame is described in Section 3.4. We made a number of modifications to the vanilla Q-learning algorithm:

5.1.1 Experience Replay Buffer

Since consecutive frames from online playing are often correlated, the agent can get stuck in local minima when trained on frames in the order in which they appear in a given episode. In order to mitigate this issue, we make use of Experience Replay: we maintain a buffer queue of the past 1000 frames of game play. When returning a batch of frames, we sample randomly from the buffer. This allows the agent to train on past data in a randomized order.

5.1.2 Target Network

$$\theta = \theta + \alpha \left(r + \gamma \max_{a' \in A} Q_{\theta^-}(s', a') - Q_\theta(s, a) \right) \nabla_\theta Q_\theta(s, a)$$

Figure 13. Update rule for Q-learning with a Target Network, where Q_θ uses the parameters of our primary network while Q_{θ^-} uses the parameters of our target network.

In vanilla Q-learning, we use our estimated Q-value function to compute both the target and estimated Q-values. This means that each update shifts our network values towards a set of values that is also shifting, which can lead to feedback loops. Instead, we maintain a second target network, whose values are only occasionally updated to the primary network's Q-values. In our simulations, we experimented with several different frequencies with which to up-

Model	Maximum Score
Random	12
Linear	14
Fox CNN	18
DQN	**44**

Figure 14. Maximum end of game score via different models after 10,000 training frames.

date the primary network. Our update rule is described in Figure 13.

5.1.3 Warm Start

We can run Q-learning offline on the dataset from our classification task to initialize a preliminary Q-value function before we begin online training.

5.2. Experiments

While our Classification task happened offline on a prerecorded and labeled dataset, Q-learning happens online, in real-time. Every 5 frames, the emulator sends our model the current frame, which represents state s. The model uses its current Q-value function to return the argmax action a to $Q(s, a)$.

We compared the results of deep Q-learning using different models, then ran several additional experiments with the DQN model. We tested various target network update frequencies and replay buffer sizes to better understand how they affect the agent's learning. In addition, in order to help our models learn to avoid obstacles, we introduced a new hyperparameter, `health_weight`: When returning the reward for a particular state and action, rather than extracting just the score, we also extract the health and return a weighted sum computed as `score + health_weight * health`. Since health ranges from 0 to 1, the `health_weight` hyperparameter corresponds to the number of score points that having full health is worth. Without any additional modifications to the reward, increasing the value of `health_weight` resulted in the agent colliding more often because the agent learned that once it dies, it begins a new life with full health. In order to mitigate this issue, we artificially inflict a score of `-health_weight` each time the agent loses a life.

With regard to other experimental parameters, we use a batch size of 10 and an E-greedy exploration rate epsilon of 0.05. Like in classification, we add L2 regularization to the loss with a regularization lambda of 0.01.

5.3. Results

Figure 14 compares the maximum score achieved using different models. Results were obtained without using a target network, a replay buffer of size 1000, and

DQN Configuration	Maximum Score
Update T.N. every batch	**61**
Update T.N. every 10 batches	52
Update T.N. every 100 batches	44
Update T.N. every 1000 batches	17
With replay buffer	44
Without replay buffer	**52**
`health_reward` = 0	20
`health_reward` = 10	44
`health_reward` = 100	**53**

Figure 15. Maximum end of game score via different versions of DQN after 10,000 training iterations. Default parameters were updating the target network (T.N.) every 100 batches, replay buffer of size 1000, and health reward of 10.

`health_reward` value of 10. We selected the DQN model for the remaining experiments because it outperformed the other models and runs in close to real-time.

Figure 16 compares the maximum score achieved using different configurations of DQN. The DQN model with the default parameters utilizes a target network with an update frequency of 100 batches, a replay buffer of 1000 frames, and a `health_weight` of 10. Each of the runs involved removing one of these optimizations and training for 10,000 iterations (1,000 `batches` * `batch_size` 10). The results of the experiments suggest that removing the target network, removing the replay buffer, and setting `health_reward` to 100 produces the best results. However, the authors of this paper might argue that 10,000 iterations may not be enough for the models, particularly the model updates the target network every 1000 batches, to converge.

The experimental results above suggest that updating the target network more frequently improves results. However, we suspect that if the experiment were rerun with much longer training periods, the opposite may be true. Target networks are known to help stabilize feedback loops, and with sufficient training time, may converge to better optima.

5.3.1 Loss

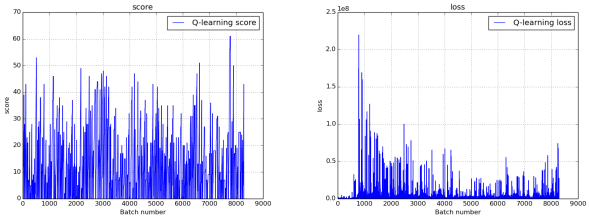

Figure 16. Score (left) and loss (right) plotted over 8,000 training batches using online q-learning.

Figure 16 shows the maximum score achieved at the end of games and the corresponding Q-loss as the DQN model is trained over time. Over the course of training, the loss on the whole decreases but retains smaller spikes that correspond to points at which the agent scores points by shooting an enemy ship or dies and is given (artificial) negative reward for having lost all its health. The more our model trains, the stronger it plays; we believe that it has yet to converge and will continue to benefit from more online learning. Nevertheless, these initial results are promising.

5.3.2 Strategy

Qualitatively, the difference between untrained and well trained models is striking. While new models seem to operate randomly, running into objects and enemy fire indiscriminately and rarely shooting, let along aiming for opponents, trained models quickly adopt what an experienced player describes as the "simplest effective strategy" for the game. The agent stays high on the screen, where it can avoid most obstacles. It learns to charge its weapon to fire homing shots, whose homing mechanism eliminates the need for accurate aim and whose wider blast range often takes out other nearby enemies. However, we notice that the agent seems less inclined to fire while flying over open water at the beginning of the level despite the enemies present at that stage. This suggests it may associate shooting more with the ground and buildings than with enemy aircrafts. We hope that with more training time, it will learn to recognize enemy crafts.

5.3.3 Generalization

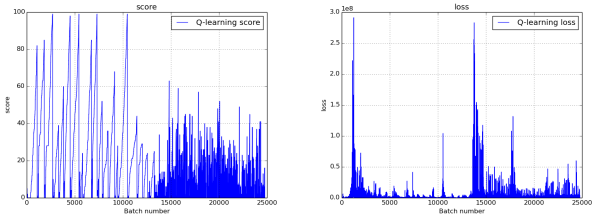

Figure 17. Score (left) and loss (right) recorded when the model played 140,000 iterations in Training Mode before being entering Level 1.

Finally, we tested our model's ability to generalize by spending 140,000 training iterations in Star Fox's Training Mode, which represents a less hostile environment with an infinite supply of enemies to shoot and very few obstacles to avoid. Correspondingly, during this time, the agent survived for a long time and achieved high scores (note the widely spaced peaks in the score graph). We then set the model loose on Level 1. After an initial (expected) drop in score, marked by corresponding spikes in the loss graph, the model quickly recovered and learned to apply what it had learned in Training Mode to the new, more challenging environment in less than 10,000 additional training iterations. Its best score of 63 is still substantially less than that of an experienced human player (115), yet its ability to generalize to unseen environments is exciting.

6. Conclusion and Future Work

Although more training time is required to explore the full extent of its potential, our initial results suggest that online Deep Q-Learning with DeepMind's DQN model yields promising results for autonomous navigation and combat.

In the future, with more time and compute power, there are a number of directions we would like to explore. First, we would optimize our hyperparameters using random grid search. We are particularly interested in experimenting with the learning rate and regularization lambda, though we would also like to explore changes to the counts and kernel sizes of the filters in our convolutional layers. For Q-learning, we would like to modify the batch size and decrease exploration over time by decaying the value of epsilon in e-greedy.

Second, we would increase training data and time. While each experiment ran at a minimum for several hours, our online models would benefit from days to weeks of learning. In particular, we believe that the DeepMind DQN Multi-Frame model has the potential to outperform the standard version, so we would like to experiment with using it for online Q-learning. We would recompute saliency maps and convolutional filter weights for these models, anticipating more refined and recognizable features in both.

Finally, after the success of our first generalization experiment, we would like to test how a model trained on Level 1 performs on Level 2 of the game. This level takes places on an asteroid field, where the obstacles themselves have velocities and, for a human, are more difficult to discern from the background. Success is this task would be a testament to the learning capacity of our model.

We are optimistic that agents trained in realistic virtual environments, where they can develop advanced policies online, could provide a warm start to physical autonomous navigation and combat systems.

7. Code

We publish our code at

`https://github.com/katgregory/fox-net`

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012.

[2] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[3] E. Brunskill. Cs234 assignment 2 replay buffer starter code, 2017.

[4] A. Dewing and X. Tong. Now this is podracing - driving with neural networks. 2016.

[5] J. J. Fei-Fei Li and S. Young.

[6] D. H. Hubel and T. N. Wiesel. Shape and arrangement of columns in cat's striate cortex. *The Journal of Physiology*, 165(3):559-568.2, 1963.

[7] D. K. Kim and T. Chen. Deep neural network for real-time autonomous indoor navigation. *CoRR*, abs/1511.04668, 2015.

[8] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman. Building machines that learn and think like people. *CoRR*, abs/1604.00289, 2016.

[9] C. Man, K. Xu, and K. Gregory. Nli-calypso. `https://github.com/katgregory/nli-calypso`, 2017.

[10] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wiestra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[13] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[14] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013.

[15] B. C. Stadie, S. Levine, and P. Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *CoRR*, abs/1507.00814, 2015.

[16] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[17] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[18] Z. Wang, N. de Freitas, and M. Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[19] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.