DuckQ: Playing Duck Game with Deep Reinforcement Learning

December 18, 2023

Tiger Wang
tiger.wang@yale.edu

Garrek Chan
garrek.chan@yale.edu

Contents

1. Introduction 1
2. General Approach 1
3. Data 2
4. Methodology 2
5. Implementation 4
6. Results 5
7. Appendix A: Source Code 7
8. Appendix B: How to Run 7
References 8

1. Introduction

Duck Game is a multiplayer 2D platform action game, in which players play as ducks with the aim of eliminating other players using a combination of platforming skill and the help of various gadgets placed around the map. Unlike most other platform action game franchises in this genre, Duck Game does not include a bot or computer player that allow for a single-player mode.

While Duck Game is best enjoyed with friends, ideally the game should be equipped with a single-player mode that allows a player to compete against bots, for either leisure or training purposes. The goal of this project is to develop a bot or computer agent that can play Duck Game with some degree of competence. While there is at least one existing community project that attempts to create a bot using manually hard-coded heuristics, there has been no previous attempt to apply the tools of machine learning to create a Duck Game bot to our knowledge.

2. General Approach

We will take a computer vision approach based on deep learning. The computer agent we develop will take as its input display buffer data (i.e. pixels) from the game. Rather than dealing with a video stream, the input will consist of a single still frame (i.e. a screenshot) from the game. Using frames as input, the agent will compute the optimal next step to take. Possible actions include movement (e.g., moving to the right), picking up or dropping a gadget, or applying a gadget (e.g., equipping a shield or shooting a weapon at another player). The agent will repeatedly go through this cycle of [input frame, optimal action, output frame] in order to play the game in a continuous manner.

How will the computer agent compute the optimal action given an input frame? This is where deep reinforcement learning comes into play. In reinforcement learning, the agent learns through trial-and-error. Upon taking an action, the agent is rewarded based on its performance. If the outcome is positive (e.g., an opponent is eliminated), then the agent is rewarded with some positive value. Otherwise, the agent

Figure 1: Screenshot of *Duck Game* gameplay

gains zero reward. This action-reward data will be continuously channeled into a deep neural network that will attempt to learn a policy that determines the optimal reward given a particular frame input. In other words, the network will be trained in real-time and the agent's performance, theoretically, should improve over time.

3. Data

As discussed in the previous section, deep reinforcement learning trains a model in real-time based on feedback data from the agent's interactions with the environment. The data required is the agent's repeated interactions with the game environment, which will be generated in real-time and does not need to be prepared beforehand. However, modifications to the game environment will be necessary to ensure that frame data can be correctly captured, and that the agent can autonomously take actions in the game without manual intervention.

4. Methodology

Our general methodology is inspired by [1] and [2]. Our project are similar to these previous works in that we all attempt to apply deep reinforcement learning methods to existing games. Our project is distinctive in applying those methods to a novel game, Duck Game, that has not previously received reinforcement learning attention. The project consists of two parts. Firstly, we must modify the game code of Duck Game in order for it to be interoperable with deep learning frameworks like Gym and PyTorch. Secondly, we must train a neural network to play the modified game.

In some respects, the most challenging aspect of this project is setting up the game environment (akin to data cleaning in a non-RL machine learning project). The game must provide clean interfaces for a remote agent (the neural network) to take actions, and the game must be able to return data in the form of pixel output and metadata. The game does not natively support any of this functionality, so we must

deep dive into the game code and make necessary additions and modifications. This turns out to be the most difficult part of the project to implement. For the second part (learning), we implement a neural network using PyTorch based on standard reinforcement learning algorithms.

We specify a simplified API for interacting with Duck Game. The action space of a duck agent in a round of Duck Game consists of five possible actions: do nothing, move left, move right, grab, and shoot. This simplified action space does not represent all game actions (such as quack or ragdoll), but enables the agent to interact with its environment and offers sufficient complexity. After each step, the game returns an 84x84 RGB pixel rendering of the game state, as well as a boolean flag indicating whether or not the agent (DuckQ) succeeded in shooting the target. The modified game environment also provides easy-to-use interfaces to reset the game (initiating a new episode). For each episode, we cap interactions at 300 steps. After 300 steps have elapsed and DuckQ has not succeeded in shooting its target, we will truncate the interaction and start a new episode.

We design a custom game level (Figure 2) as the target of this reinforcement learning project. The custom game level consists of two players, the agent (DuckQ) and the target. The target does not take any actions and remains stationery. The map provides two weapons with which DuckQ may terminate the round by shooting the target. However, DuckQ must first navigate to the item, manually pick up the item, turn in the target's direction, then press shoot. So this seemingly simple task for experienced players is not as straight-forward as it might first appear.

We use a convolutional neural network architecture based on an official PyTorch tutorial for reinforcement learning [3]. The network has five layers. The first three layers are 2D-convolution layers. The first layer has 32 output channels, a kernel size of 8, and a stride size of 4 (such that each input channel image is downscaled by 4x after the first layer). The second layer has 64 output channels, a kernel size of 4, and a stride size of 2. The third layer has 64 output channels, a kernel size of 3, and a stride size of 1 (no

Figure 2: Target game level. DuckQ can be identified by its egg hat

downsampling). The fourth layer is a linear convolution layer that expects flattened inputs and has 512 outputs. Finally, the fifth layer is the output layer, which has an output size that corresponds to the size of the action space. Each layer except the output layer is followed by rectified linear unit activation (ReLU).

The convolutional neural network expects monochrome pixel data as input channels. Each monochrome image is square in the shape of 84x84 pixels. We have four input channels, each corresponding to a single monochrome image. The four channels represent stacked pixel data, so that the network has visibility into not only the immediate pixel data on the game screen, but also three frames prior to the current frame. This allows the network to detect movement (is the duck agent moving left or right?) that it can otherwise not detect using a single frame of pixels. We additionally implement frame skipping. Consecutive game frames do not offer much additional information to the neural network, so we only record every fourth frame of the game.

The output space of our convolutional neural network corresponds to the value of each action in the action space in contributing to the optimal reward value. Given input frames, the neural network is trained to evaluate the optimality of each of the five actions in the game action space. The DuckQ agent will take the action with the highest predicted optimality.

We use the DDQN (double Q-learning) learning algorithm to train the convolutional neural network, following the official PyTorch tutorial [3]. The learning algorithm initializes two instances of the neural network, `online` and `target`. The `target` model's weights are periodically synced with the `online` model's weights. The `online` network is responsible for determining which action is optimal (so that its output values are only used as cardinal rankings). The `target` network is responsible to determine the actual optimality value of the proposed action [4], [5].

On each step, the agent may either explore with probability ε or exploit with probability $1 - \varepsilon$. As a hyperparameter, ε is initialized to 1 and decreases by a factor of 0.9999972797 on each step. We determined the ε such that, after 10000 training episodes each averaging 75 steps, the exploration rate remains between 10% and 20%. The agent explores by taking a random action in the action space of five actions. The agent exploits by running inference on the neural network.

After each action, the agent observes the 84x84 pixel output, as well as whether or not it has succeeded in shooting its target. We assign a reward value of 1 if the agent successfully shoots its target, and a reward value of 0 otherwise. The agent records each interaction (including the input, output, and reward value) into a replay buffer. The neural network updates itself in real-time; DuckQ will periodically sample from its replay buffer and backpropagate to update values in the `online` model. We use a batch size of 32 when sampling from the replay buffer, following the PyTorch tutorial [3].

5. Implementation

To modify Duck Game with the requisite changes, we use an open-source, decompiled version of the game, DuckGameRebuilt [6]. The decompiled version of the game gives us access into the C# game source code, which we subsequently modify to provide the necessary features to enable reinforcement learning.

The normal game environment runs at a predefined speed of 60 frames per second. This offers a consistent experience for human players, but is not useful for reinforcement learning. We first modified game logic to uncap the frame speed. This ensures that each iteration of the game's main update loop results in exactly one frame re-render. This also ensures that the game speed is accelerated for training purposes. We also resize the game window to 252x252 to alleviate graphics processing requirements.

In order to allow inter-process communication (IPC) between Duck Game, written in C# .NET, and our reinforcement learning algorithm, written in Python, we use the TCP messaging library ZeroMQ [7], [8]. We embed into Duck Game source code a ZeroMQ message server, which allows it to receive and send messages with our Python client. Messages are routed through the network layer (TCP), but the latency remains minimal because messages are merely routed from one `localhost` port to another.

We inject the core game update loop with custom logic. On each update operation, the game will poll ZeroMQ for a message. It will block game execution until a message is received. The message is the Python client's command to tell the game what to do in the next step. The message may be `reset`, `step`, or `pass`. When Duck Game receives a `reset` message, it will invoke our implemented logic to start a new round. The `step` message is followed by `left`, `right`, `grab`, `shoot`, or `none`, corresponding to each of the five actions in the action space. We modify the game to inject the appropriate controller input. After rendering the output on screen, it will return the resulting metadata and pixel data. The `pass` message is a technical detail that allows the game to pass through a large number of frames without transmitting pixel data. It is useful to allow the game a predefined number of delay frames when waiting for the round to initialize.

We capture game frames and return pixel data from Duck Game using common .NET APIs. We initialize a new empty frame with size 84x84. We then invoke a Duck Game API to render the game onto the empty frame in addition to the game window. We then serialize the RGB pixel data with string operations and return it to the Python client as a ZeroMQ message.

In order to allow a consistent Python game interface for PyTorch to interact with, we implement a OpenAI Gym custom environment that wraps the modified C# Duck Game. The wrapper environment implements standard methods such as `step` and `reset`. The wrapper is merely responsible for connecting to the Duck Game ZeroMQ TCP messaging server, sending appropriate messages, and parsing server responses into pre-defined formats expected by reinforcement learning pipelines. We used [9], [10] as references for understanding how `Gym.env` classes are implemented.

To implement the convolutional neural network, we use Python 3.10 with PyTorch and TorchVision as reinforcement learning libraries. Our architecture and hyperparameters have already been described in Section 4. Our implementation of the neural network and the training/evaluation loop is adapted from the official PyTorch tutorial on reinforcement learning [3].

6. Results

We train our convolutional neural network for approximately 2,660 episodes and 200,000 cumulative steps. Each episode corresponds to a round of Duck Game and each step corresponds to DuckQ taking an action based on pixel inputs. The average reward, loss, step length, and Q values as a function of training episodes is plotted in Figure 3.

The resulting model can be found in the `DuckQ` source code repository (see Appendix A). It is also submitted as part of the Canvas submission (see `README.md` for the project index). We recorded a video demo of the training and evaluation loop. It can be accessed at this link.

We qualitatively evaluate the performance of the resulting model. The trained DuckQ agent is unable to find the correct sequence of moves to shoot the target duck. However, the model meets the qualitative evaluation criteria we outlined in the project prospectus. In order to shoot the target duck, the agent needs to move to the left, pick up the weapon, move to the right again to face the target, and shoot the weapon. Our trained agent exhibits up to 3 of the 4 required steps. Our trained agent consistently moves

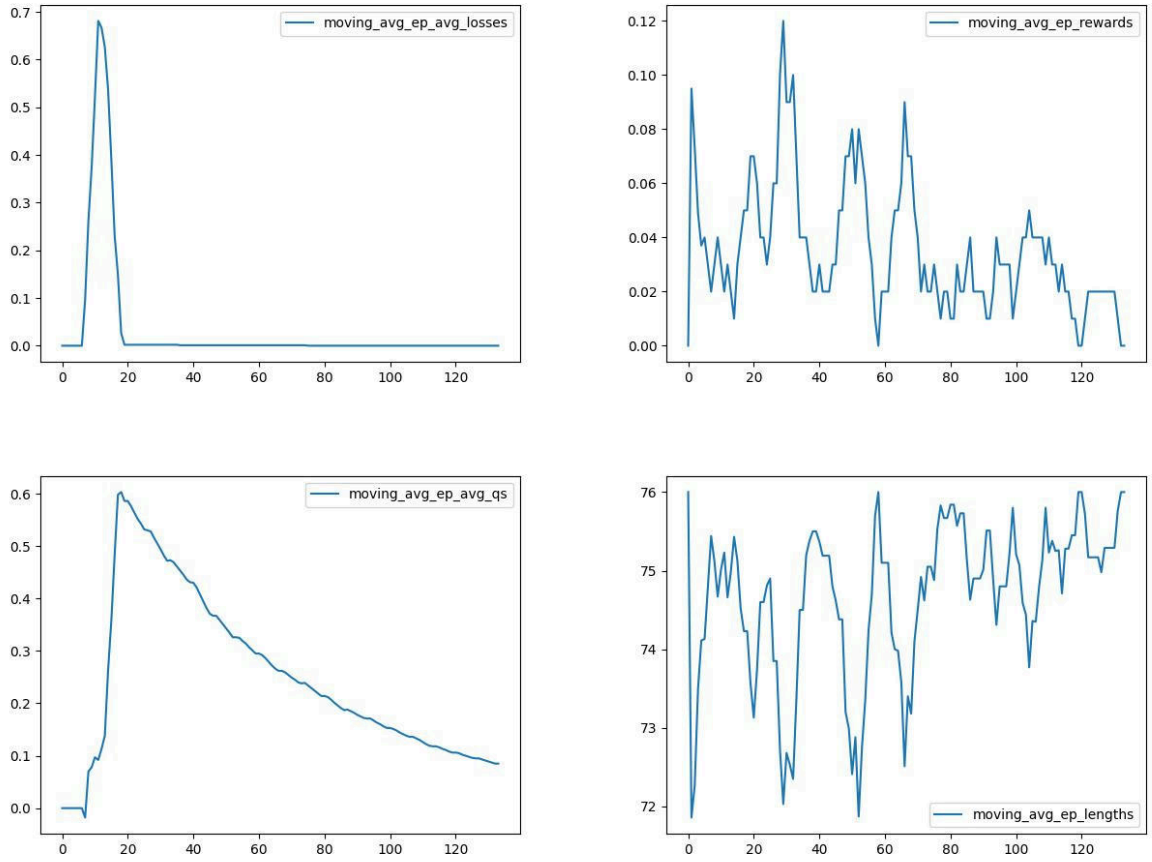

Figure 3: Average loss (top left), average reward (top right), average Q value (bottom left), average step length (bottom right)

to the left of the screen to the area of the weapon. It consistently picks up the weapon, and is sometimes able to turn right and shoot the weapon. It proves difficult, however, for the trained agent to master the exact set of controller operations to execute the kill. The timing must be very precise such that the agent shoots immediately after turning right to face the target, so it is understandably difficult for the trained agent to achieve.

Importantly, the trained agent achieves our qualitative aims of understanding basic movement. It in fact achieves the core steps required to pass the level (moving in the appropriate direction, picking up the weapon, shooting the weapon), though it finds it difficult to combine the operations in a time-sensitive manner. Nonetheless, this demonstrates that our neural network has done positive work in understanding the game environment and the actions it must take to maximize reward values.

One possible way to improve our result is to train the agent for more iterations. Due to computation constraints (we only had access to laptop computers without GPUs), we were only able to train up to approximately 2,600 episodes, which required 5-6 hours of training time using a CPU. If we had access to more compute power and GPUs, we could have possibly improved our result by training for more iterations. This also highlights the computational intensity required for reinforcement learning, which requires us to run the training code as well as game rendering simultaneously. More computer power than expected was therefore required.

We believe that one of our most important results is the modified game environment itself. There is no previous public work of a reinforcement learning agent for Duck Game to our knowledge. DuckQ, therefore, may be the first reinforcement learning agent for Duck Game. Not only can future work build on top of our neural network architecture and improve performance, they can leverage our modified game environment to build entirely new neural networks or AIs.

To conclude, in this project, we have applied the tools of deep reinforcement learning and double Q-learning to a novel game. We built a functional game environment that can support computationally-intense reinforcement learning tasks. We trained a sample model using that modified game environment, which passes the desired qualitative evaluation criteria but could be improved by future work with more dedicated compute power and GPUs.

7. Appendix A: Source Code

Project source code can be found on our personal GitHub repositories (public):

- https://github.com/thenewpotato/DuckGameRebuilt - Modified C# Duck Game implementation
- https://github.com/thenewpotato/DuckQ - Gym custom environment and PyTorch neural network training and evaluation

8. Appendix B: How to Run

1. Set up a Windows 10 or Windows 11 environment. Because of Duck Game requirements, our project can only be run on Windows. Ensure there is at least 4GB of free disk space after installing all dependencies.
   - We recommend finding a Windows machine to run this project on. If you are using a Mac, you can install Windows on a dual-boot partition using Bootcamp (this is what we use for development). Alternatively, you may set up a Windows virtual machine (we did not test this option and you may experience performance issues).
   - Note: The training and evaluation scripts **require** a running Duck Game instance. Proper environment setup as described is required to run **any** part of this project.
2. Install Visual Studio Community 2019 (Important: VS2019 is required, the project may not function on higher or lower versions). Link to install (Scroll to 2019, click download, sign in to a Microsoft account, and download the VS2019 Community Installer).
3. Launch the Visual Studio installer. Select ".NET Desktop Environment". On the right hand side panel, additionally select ".NET Framework 4.8 Development Tools". Also select "Desktop Development with C++". Wait for the installation to finish.
   - Note: Installing VS2019 is required even if you are not building the game. This step will populate necessary shared libraries used by the Duck Game executable and the Python training script.
4. Install Python 3.10 (Important: PyTorch on Windows will only work on Python versions 3.8-3.10)
5. Install Python dependencies. In the Windows command line, run

   ```
   py -m pip install gym numpy pyzmq torch torchvision torchrl tensordict jupyter
   ```
6. Install Git for Windows.
7. Install Steam for Windows. Steam must be running (in the background) for Duck Game to launch.
8. Clone our GitHub repository containing the modified Duck Game game code. Launch Git Bash and run

   ```
   git clone https://github.com/thenewpotato/DuckGameRebuilt.git
   ```

- At this point, you can launch Visual Studio Code 2019 and open `DuckGame.sln` in the cloned repository. This allows you to build/debug the game yourself to verify our implementation (and make any changes, if necessary).

9. Build Duck Game. Launch VS2019, open `DuckGame.sln` in the project repository. Wait for the project to initialize. On the top bar in Visual Studio, change "Debug" to "Release" and ensure that architecture is set to "Any CPU" and the target is set to "DuckGame". Then navigate to menu "Build" - "Build Solution" to initiate the build. The build should take about a minute to complete. The built executable is stored in `bin\DuckGame.exe` in the project repository.
10. Launch `bin\DuckGame.exe` (found under the Duck Game game code repository). Wait until the game finishes loading and you're on the title screen (it reads "Duck Game Rebuilt").
11. Focus on the Duck Game window. Type *exactly* the following sequence:
    - Click once the "~" key to open the game command line
    - Type "level next", followed by enter
    - At this point, the game should hang (it waits for the neural network agent). If the game does not hang, then you probably mistyped one of the above commands (we have modified the game to run with unlimited FPS, so a single key press may register as more if you press it down for too long). If the game does *not* hang as expected, force close the executable and restart this step.
12. Clone our GitHub repository containing our Python training and evaluation code. Launch Git Bash and run

    ```
    git clone https://github.com/thenewpotato/DuckQ.git
    ```
13. Launch Jupyter Lab by going to the Windows command line and running

    ```
    py -m jupyter lab
    ```
14. In Jupyter Lab, navigate to the cloned Python DuckQ repository. Open the `DuckQ` notebook.
15. To train, execute up to the "Evaluation" section. This will periodically print reward/loss values and save model weights into the "checkpoints" directory at the end. The training logic will automatically connect to the running Duck Game instance and control it remotely.
16. To evaluate, restart Duck Game and the Python kernel as specified in step 17. Execute the notebook up to the "Training" section. Skip the "Training" section and execute the block in the "Evaluation" section.
17. Note: Before starting a new training or evaluation session, force close Duck Game and reopen it. Rerun the initialization sequence as specified in step 11. Then in Jupyter notebook, restart the kernel and run train or evaluate steps as specified in steps 15 or 16.

References

[1] V. Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning". 2013.

[2] T. Li and S. Rafferty, *Playing Geometry Dash with Convolutional Neural Networks*. 2017. [Online]. Available: http://cs231n.stanford.edu/reports/2017/pdfs/605.pdf

[3] PyTorch, "Train a Mario-playing RL Agent — PyTorch Tutorials 2.2.0+cu121 documentation". [Online]. Available: https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html

[4] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning". 2015.

[5] Kari, “Q-Learning: Target Network vs Double DQN”. [Online]. Available: https://datascience.stackexchange.com/a/32348

[6] TheFlyingFoool, “GitHub - TheFlyingFoool/DuckGameRebuilt: Duck Game decompiled & rebuilt with some added features,”. [Online]. Available: https://github.com/TheFlyingFoool/DuckGameRebuilt

[7] ZeroMQ, “C# Guide”. [Online]. Available: https://zeromq.org/languages/csharp/

[8] ZeroMQ, “Python Guide”. [Online]. Available: https://zeromq.org/languages/python/

[9] yfeng997, “GitHub - yfeng997/MadMario: Interactive tutorial to build a learning Mario, for first-time RL learners”. [Online]. Available: https://github.com/yfeng997/MadMario/tree/master

[10] PyTorch, “Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 2.2.0+cu121 documentation”. [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html