Reinforcement Learning

Playing Doom with a Deep Q Network

Sailesh Rajagopalan Robotics Engineering Worcester Polytechnic Institute Worcester, MA srajagopalan@wpi.edu	Gokul Srinivasan Robotics Engineering Worcester Polytechnic Institute Worcester, USA gsrinivasan@wpi.edu	Rohith Venkataramanan Robotics Engineering Worcester Polytechnic Institute Worcester, USA rvenkataramanan@wpi.edu

Introduction

Deep reinforcement learning has shown to be a huge success in game play. One of the most notable examples is the application of Q-learning-based approaches in training deep reinforcement learning (DRL) players, such as the use of Deep Q-Networks (DQN) to play the Atari game and AlphaGo's triumph over a human champion in Go. In this research, we wanted to see how DQN may be used to train an AI agent in the 3D game Doom, a classic First-Person Shooting (FPS) game released in 1993. There are some key distinctions between Doom training and that of other games like Atari Breakout or Pong. The intricacy of Doom AI is one big difference that makes it more difficult. Specifically, when compared to other games,

The partially observable Markov Decision Process problem (POMDP). When we just have a restricted set of information about the environment, we call it a partially observable MDP (Markov Decision Processes). So far, we've seen a fully observable MDP where we know all possible actions and states—even though the agent might be unaware of transition and reward probabilities, it had complete knowledge of the environment, such as a frozen lake environment, where we clearly know all the states and actions of the environment and can easily model it as a fully observable MDP. However, most real-world environments are only partially visible; we are unable to perceive all the states. Consider the agent who is learning to walk in a real-world setting; obviously, the agent is learning to walk in a real-world setting. The agent will not have a thorough understanding of the surroundings and will have no access to information outside of its field of view. Although states in POMDP only provide partial information, preserving information about previous states in memory may aid the agent in better understanding the nature of the environment and improving policy. As a result, in POMDP, we must keep track of past states in order to execute the best possible action.

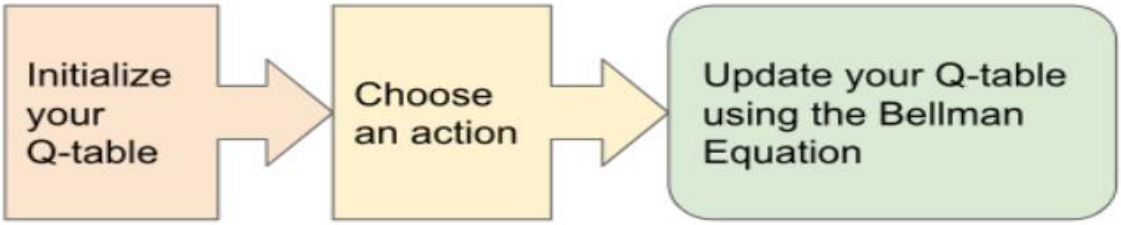
Related Work

Many of the DRL AIs that have been trained so far have been on completely viewable games like Atari Breakout or Space Invaders. The majority of game AIs taught using Q-learning and its variant algorithms, such as DRQN, prioritized (weighted) replay, and dueling, outperformed benchmarks utilizing other well-performing reinforcement learning models and received the highest rewards. The innovative A3C model, which has been introduced to train AIs for Atari games and TORCS car racing, draws a link to the dueling architecture. The use of an asynchronous framework in conjunction with a classical actor-critic paradigm was able to solve some of the issues associated with single-threaded Q learning. In training, this model displayed a high level of resilience and stability, as well as significant training gains through multi-threading.

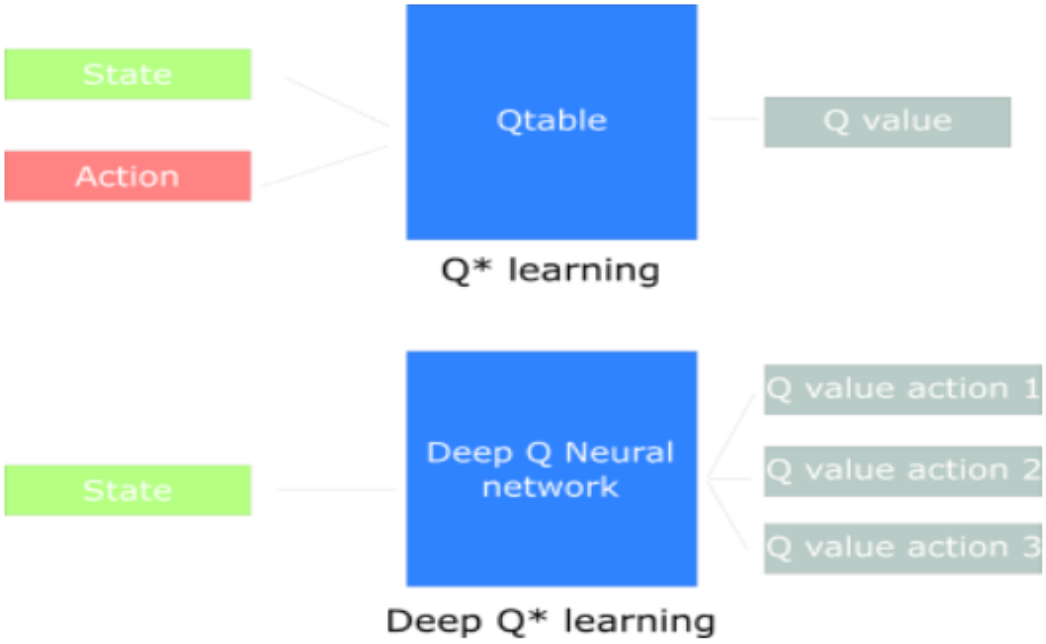
Prior work on Doom AI training has primarily focused on extending the existing deep recurrent Q-learning network (DRQN) model with high-level game data. Hafner and Kempka, for example, have experimented with replay memory (batch size 64) for stabilization on DQN. Obtaining the internal game state of enemies as an added layer of supervision during their training phase. Their model essentially trained two different networks: one for the action network that used DRQN enriched with game features, and another for the navigation network that used a simpler DQN. Their methods have resulted in enhancements to the conventional DRQN network, and they could easily be integrated with other common Q-learning techniques like prioritized experience replay. These previous studies in relevant fields have all been quite useful in assisting us in the development of this project.

Methodology

Reinforcement Learning is a prominent and fascinating topic of Machine Learning that is gaining a lot of attention. Breakthroughs in Reinforcement Learning have allowed computer algorithms like Alpha Go and OpenAI Five to attain human-level performance on games like Go and Dota 2. The Deep Q-Learning algorithm is a key idea in Reinforcement Learning. Naturally, many of us are curious about the algorithms behind these remarkable achievements. We'll share a basic Deep Q-Network implementation (minDQN) in this tutorial, which will serve as a practical guide to assist new learners code their own Deep Q-Networks. There are two types of learning algorithms: model free and model-based Reinforcement Learning algorithms.



Model-free Reinforcement Learning algorithms provide predictions of future states and rewards without learning a model of their environment's transition function. Deep Learning, Q-Learning Policy and Q-Networks Because they do not generate a model of the environment's transition function, gradient techniques are model-free algorithms.



The vanilla Q learning algorithm follows a methodical process to inhibit, initializing the Q table and choosing an action using the epsilon greedy exploration strategy and updating the Q table using the bellman equation. The implementation of the Q-table is a key distinction between Deep Q-Learning and Vanilla Q-Learning. Deep Q-Learning, crucially, substitutes the traditional Q-table with a neural network. A neural network maps input states to (action, Q-value) pairings rather than mapping a state-action pair to a q-value. The learning procedure in Deep Q-Learning employs two neural networks. The design of these networks is the same, but the weights are different. The weights from the main network are replicated to the target network every N step. When both networks are used, the learning process becomes more stable, and

the algorithm learns more successfully. Every 100 steps, the main network weights replace the target network weights in our implementation.

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha\left(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)\right)$$

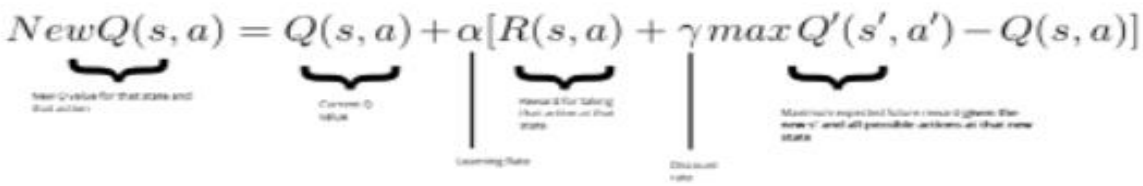
The current reward plus the maximum future reward for the following state is the Q value, often known as the maximum future reward for a state and action. And it all makes sense when you think about it. Gamma (γ) is a number between [0,1] that is used to discount the reward as time passes, assuming that early actions are more significant than later actions (an assumption that is confirmed by many real-life use cases).

Experience Replay

The notion of experience replay is to assist the agent in remembering and not forgetting its earlier acts by replaying them. We feed the network a sample of prior events (which are kept in a buffer) every now and then. As a result, the agent is able to revisit its history and improve its memory. Another rationale for this assignment is to drive the agent to break free from oscillation, which occurs when there is a significant degree of correlation between certain states, resulting in the same behaviors being repeated over and over. The act of storing and repeating game states (the state, action, reward, and next state) that the Reinforcement Learning algorithm may learn from is known as experience replay. Off-Policy algorithms can employ Experience Replay to learn in an offline mode. Off-policy approaches can use saved and stored information from prior operations to adjust the algorithm's parameters. To prevent skewing the dataset distribution of distinct states, actions, rewards, and next states that the neural network will observe, Deep Q-Learning leverages Experience Replay to learn in tiny batches. The agent does not need to train after each step, which is important. We utilize Experience Replay to train in tiny groups once every four steps rather than every single step in our approach.

Deep Q Learning Algorithm

In this project using the Q value update for a given state and action using the Bellman equation, updating the neural net weights which reduce the error value that will be present.



The difference between our Q target (highest feasible value from the next state) and Q value is used to determine the error (or TD error) (our current prediction of the Q-value).

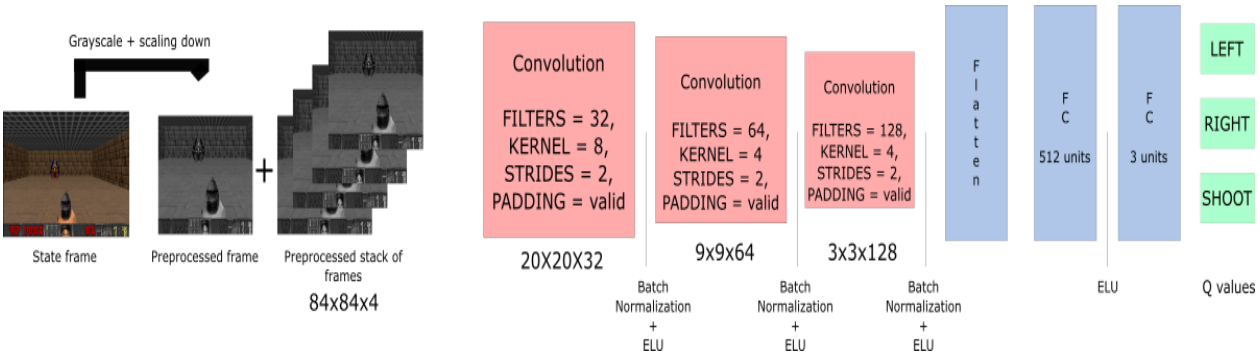
$$\Delta w = \alpha[(R + \gamma \, max_a \, \hat{Q}(s', a, w)) - \hat{Q}(s, a, w)] \, \nabla_w \hat{Q}(s, a, w)$$

Change in weights | learning rate | Maximum possible Qvalue for the next_state (= Q_target) | Current predicted Q-val | TD Error | Gradient of our current predicted Q-value

In this approach, two steps occur: we sample the environment in which we execute actions and store the observed experience tuples in a replay memory; and we sample the environment in which we conduct actions and store the observed experience tuples in a replay memory. Choose a small batch of tuple randomly and use a gradient descent update step to learn from it.

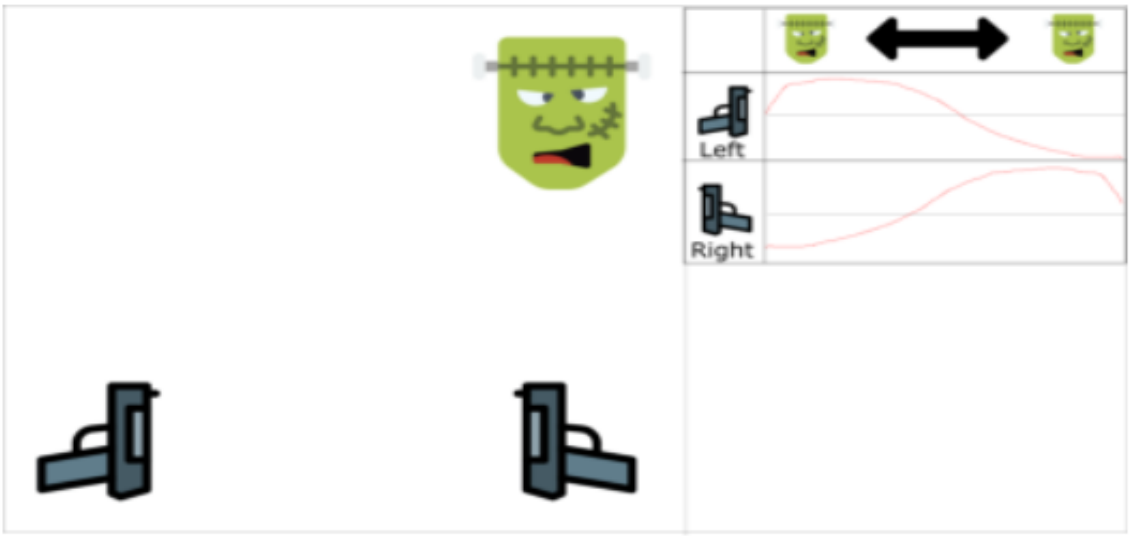
Architecture

A stack of four frames is fed into our Deep Q Neural Network. These are sent through its network, which generates a vector of Q-values for each conceivable action in the current state. To choose our optimum action, we must take the vector's highest Q-value. Initially, the agent performs dreadfully. However, it learns to correlate frames (states) with the optimum actions to do over time.



Preprocessing is a step through which reduction of complexity of the present states is possible reducing the computation need required for training enabling it for low cost for derivation. The process includes grayscale of every state by reducing the color channels present to 1 and alongside reducing the size of the frame and stacking four subframes together is an important first step.

Three convolution layers are applied to the frames. These layers allow the usage of spatial connections in photos to your advantage. By taking advantage of some spatial features across frames because they are stacked together. The convolutions use ELU activation function, using a fully connected layer with ELU activation function and one output later, a fully connected layer with a linear activation function which produces the Q value estimation in every section.



Q value approximations

Every action present is dependent on the state and affects the next state. A sequence of experience tuples is highly correlated, reducing the correlation between experiences, This project trains the network in a sequential order. We can disrupt this association by sampling at random from the replay buffer. This prevents action values from wildly fluctuating or diverging. To enable rational behavior of the agent with the process of how the agent shoots with the gun, First and foremost, we must cease learning while engaging with our environment. To explore the state space, we should attempt new things and play a bit haphazardly. These encounters can be saved in the replay buffer. Then we may look back on these events and draw lessons from them. Returning to playing with the updated value function after that. As a result, we will have a more comprehensive set of instances. We'll be able to generalize patterns from these instances and recall them in whatever order we want. This keeps you from becoming stuck in one part of the state space. This keeps the same action from being reinforced over and over.

Result:

Architecture:

The input consists of an 84x84x4 image directly fed from the game environment, on which 2D batch normalization is performed. The model has 2 layers of convolution neural networks, and

each of them has an ELU activation function to act as rectifier layer, through which the image is passed.

The hyperparameters used are as follows:

A discount factor gamma of 0.95 was used, the closer it is to 1, the better the performance of the model. A learning rate of 2e-4 was used since the model seemed to perform better for lower value of alpha. The epsilon value was taken as 1 and a decay rate of 1e-4 was used.

```
### MODEL HYPERPARAMETERS
state_size = [84,84,4]      # Our input is a stack of 4 frames hence 84x84x4 (Width, height, channels)
action_size = game.get_available_buttons_size()              # 3 possible actions: left, right, shoot
learning_rate =  0.0002      # Alpha (aka learning rate)

### TRAINING HYPERPARAMETERS
total_episodes = 500        # Total episodes for training
max_steps = 100              # Max possible steps in an episode
batch_size = 64

# Exploration parameters for epsilon greedy strategy
explore_start = 1.0            # exploration probability at start
explore_stop = 0.01            # minimum exploration probability
decay_rate = 0.0001            # exponential decay rate for exploration prob

# Q learning hyperparameters
gamma = 0.95               # Discounting rate

### MEMORY HYPERPARAMETERS
pretrain_length = batch_size   # Number of experiences stored in the Memory when initialized for the first time
memory_size = 1000000          # Number of experiences the Memory can keep

### MODIFY THIS TO FALSE IF YOU JUST WANT TO SEE THE TRAINED AGENT
training = False

## TURN THIS TO TRUE IF YOU WANT TO RENDER THE ENVIRONMENT
episode_render = True
```

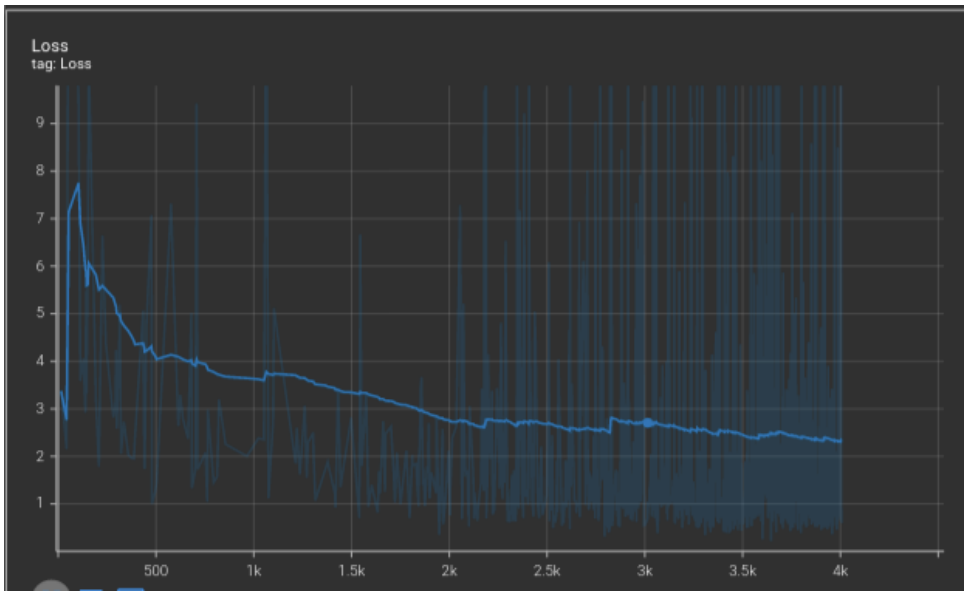
Loss Function and Optimizer:

Mean Squared error loss function and RMSprop optimizer was used for this project.

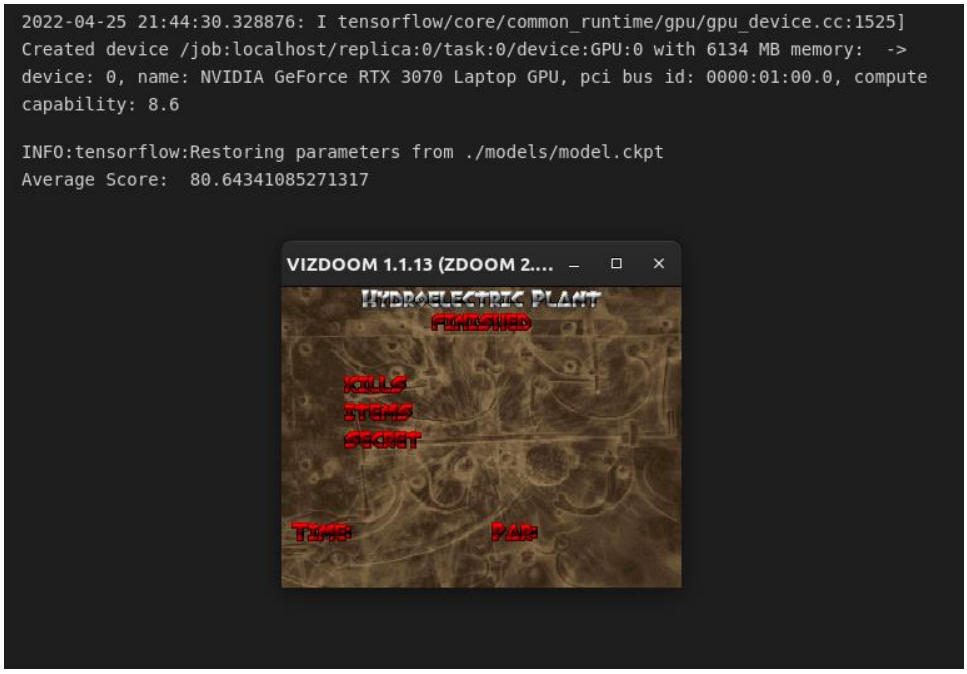
Visualization:

Loss curve:

The training was performed for 4000 games and the training loss per game is shown below. As the number of games increased, the loss also reduced well. As such it took 5 hours for training 4000 games, of which the last few thousands took the major portion of the time consumed. As the number of games increases, the model starts getting better at the game, which results in slow training speeds.



On testing the model, the average score is obtained.



References:

[1] S. Ravichandiran, *Hands-on reinforcement learning with python: Master reinforcement and deep reinforcement learning using openai gym and tensorflow*. Packt Publishing, 2018.

[2] G. Lample and D. S. Chaplot, “Playing FPS Games with Deep Reinforcement Learning.” arXiv, 2016. doi: 10.48550/ARXIV.1609.05521.

[3] Micha l Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jáskowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In IEEE Conference on Computational Intelligence

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013

[5] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In AAAI, volume 2, page 5. Phoenix, AZ, 2016.

[6] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581, 2015.

[7] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. CoRR, abs/1507.06527, 7(1), 2015.

[8] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In International conference on machine learning, pages 1928–1937, 2016.

[9] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In International Conference on Machine Learning (ICML), volume 2017, 2017.