Playing DOOM using Deep Reinforcement Learning

Tushar Dhoot (tdhoot), Daniyal Khan (danikhan), Ben Konyi (bkonyi)

Introduction

For our CS229 final project, our group implemented the Deep Q-learning algorithm, (Minh et. al, 2014) to teach an agent to perform well in various scenarios in the video game classic, DOOM. Unlike the AI prepackaged with DOOM, our agent learns to maximize rewards in our training scenarios using only the state that would be available to a human player, such as visual output and basic game state. Specifically, our agent was trained and tested on the "seek-and-destroy" and "health gathering" scenarios. The agent was able to achieve near optimal performance in both scenarios. In developing our agent, we experimented with many network and training configurations, the results of which are included below.

Related Work

Mnih et. al (2013) first presented deep reinforcement learning as a method for training an agent to play Atari 2600 games. Our work implements several concepts in their paper, such as using visual data as input and experience replay. In contrast, our efforts were focused on DOOM, which has a larger and more complex state space than Atari games.

The foundational work in DOOM playing agents is a paper published in 2016 by the creators of VizDoom (Kempka, Wydmuch, Runc, Toczek, & Jaśkowski, 2016). In the paper, the authors demonstrate the capability of a basic DQN with experience replay to learn to play DOOM. In our work, we first implemented such a basic network, but have since supplemented with more powerful features (as mentioned below). Despite later research into more complex approaches, (such as an attempt in using neuroevolution by Alvarez and Togelius (2017)), DQNs remain the state-of-the-art in learning to play DOOM and continually outperform other methods.

Many groups have chronicled their attempts at DQN-based agents for the VizDoom competition. One major influence in the field is Lample and Chaplot (2016). Similar to their findings, we significantly improved performance by augmenting our visual input with basic game information. However, we did not follow their approach of adding a boolean to indicate the presence of an enemy. While such an indicator would likely increase performance, we believe it betrays the goal of training an agent that plays with the same information as a human player.

In addition to the approaches taken by previous groups, we have incorporated recent advances in reinforcement learning such as double DQNs and dueling DQNs (as documented below).

Dataset and Features

The data was sampled from the environment by taking different actions, producing a 160x120 pixel image buffer, with each pixel being an individual feature input into the DQN. The sampled image buffer was downscaled to 40x30 through 2 max-pooling layers. For some experiments, we preprocessed the buffer, which the environment returns in RGB, to having a single grayscale channel. Other experiments took advantage of consecutive image buffers in order to capture time-based state. Specifically, if the agent transitioned from state S_1 to state S_2 using action A, then the frames associated with S_1 and S_2 would be $[F_{n,}, ..., F_{n+k}]$, $[F_{n+1,}, ..., F_{n+k+1}]$ respectively, where F_j is a 160x120 image buffer, and F_{n+k+1} was the next frame resulting from action A. Thus the input to the DQN's first convolutional layer was a 4-dimensional tensor: **[buffer width, buffer height, # of channels, # of frames]**. To make training/validation run faster, we set # of channels = # of frames = 1. In code, this data was collected as a tuple $[F_n, A, F_{n+1}, R]$, where F_n is the current frame, and action A produced frame F_{n+1} with a reward R. Lastly, we fed in cues available on the image itself. Specifically the remaining ammo and health available. In theory, a well trained agent should react more conservatively in the environment if their ammo or health is low (taking less shots at the enemy, actively avoiding combat).

Methods

The basic architecture of our DQN consists of 2 convolutional layers, with 7x7 and 5x5 kernels and ReLU activations, used to detect features in our frames such as edges and shapes, and two fully-connected hidden layers (Figure 1). These feature detectors are used to provide spatial information to deeper layers, which eventually determine an optimal action to take based on the image buffer content. With k kernels, we produce k output buffers, each of which get downscaled by a factor of 2 through each max pooling layer. After 2 layers, the k image buffers, each scaled by 25% in each dimension, flatten to a vector which propagates to two fully connected layers, and eventually a vector of size $|A|$, where A is the action space size. An input image of original size 160x120 flattens to a vector of size 38,400 before propagating through the hidden units. The goal is to have enough neurons to model the complex behavior that is embedded in the input image. The following diagram is shows an image buffer as it forward propagates through the DQN.

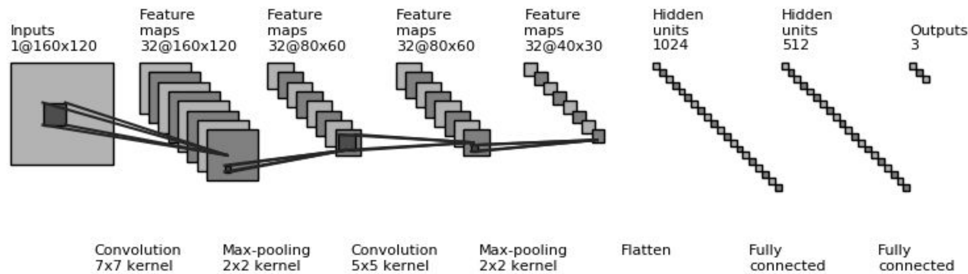

Figure 1: Neural Network Diagram showing Forward Propagation of a 40x30 image buffer

For training, we used the Adam optimization algorithm, which is a stochastic optimization algorithm with adaptive learning rates (Kingma & Ba, 2014), and the standard temporal difference loss function described in (Mnih et. al, 2013). The Adam algorithm was chosen over standard stochastic gradient descent to remove the need to tune yet another hyperparameter for learning-rate decay.

Exploration and exploitation was achieved by using simulated annealing (Carr, 2017) during training. This is done by taking a random action instead of the network's predicted action with probability ε, where ε decays from 1 to 0.1 over some number of training iterations. This allows for the agent to initially explore the space to get an idea of values for specific states before gradually converging towards an optimal policy. After each action, the resulting state tuple is placed into replay pool (Mnih et. al, 2013), from which tuples are randomly sampled when creating training batches in order to reduce correlation between training samples.

In both standard and deep Q-learning algorithms, action selection and evaluation are done using the same value function, which can lead to large overestimations of the value function. To combat this, action selection and network evaluation for our target network can be split in such a way that our target network is evaluated based on the action deemed optimal by our online network, as proposed by Van Hasselt, Guez, & Silver (2016). The resulting training target is the following:

$$Y_t^{Double\ DQN} = R_{t+1} + \gamma Q(s_{t+1} \arg\max_a Q(S_{t+1,} a;\ \theta_t),\ \theta_t^{-})$$

After a number of iterations (fixed at 1024 for our experiments), the online network is copied into the target network. Consequently, the target network takes advantage of learning done by the online network and also maintains its ability to reasonably estimate the online network.

We further augmented the DQN to output two separate streams. This new dueling network (Figure 2), returns a scalar for one stream, $V(s)$, that measures how good it is to be in state s. The other stream outputs a vector $A(s,a)$ ($|A| = \# \, of \, actions$) that measures the comparative *advantage* that each action has, given s. The final output of the network is then calculated as $Q(s,a) = V(s) + A(s,a) + f(s,a)$. The dueling network captures how advantageous a certain action might be. Some state might be indifferent to all actions, and the value is from being in the state itself rather than any particular action in that state. The presence of function f removes the ambiguity where a constant increase in V might be offset by the same decrease in A (Wang et. al, 2015). For our implementation, we chose $f = -\sum_{a'} A(s,a') / |A|$. Using the mean of $A(s,a)$ preserves the rank of $A(s,a)$. It is evident that the new gradient, $\nabla Q = \nabla V + \nabla A + \nabla f$, will attempt to optimize the state and advantage function separately. Optimizing V over the state space leads to faster convergence, especially over large action spaces (Wang et. al, 2015).

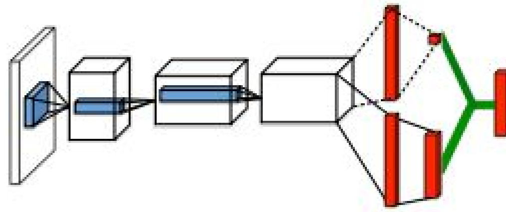

Figure 2: A Dueling Neural Network, outputting a scalar $V(s)$ and vector $A(s,a)$ (Wang et. al, 2015)

The final technique we explored in our experiments was reward shaping, a common method of guiding agents to adapt certain behaviors by adding additional rewards to encourage these behaviors (Ng et al, 1999). In our Health Gathering trials, we added a positive shaping reward based on the distance the agent had moved since the last state in order to dissuade our agent from standing in one place and hopefully reduce convergence time.

Experiments

For all of our experiments, we used the following hyperparameters:

- Learning rate: 1e-5
- Discount rate: 0.99
- Epsilon decay period: 100,000
- Shaping reward: 1e-5 x distance
- Starting epsilon: 1.0
- Ending epsilon: 0.1
- Replay pool size: 16,384
- Frame-skip rate: 4
- Batch size: 64
- Copy Steps (for target network): 1024
- Episodes per epoch: 25
- Validation episodes per epoch: 15

These hyperparameters were not chosen for any particular reason, with a few exceptions, and are comparable to those used in prior works. However, the learning rate was tuned over a few trial runs to ensure that the DQN converged at a reasonable rate. Finally, the batch size of 64 was chosen in order to strike a balance between time training and time our agent actually spent playing.

In order to compare performance of various modifications to the DQN architecture and learning algorithm, we ran the following experiments in the ViZDoom environment:

Seek and Destroy

- Standard
- Double DQN (DDQN)
- Dueling DQN (with and without DDQN)
- Dueling DQN with DDQN and RGB*
- Dueling DQN with DDQN and multiple frames*

Health Gathering

- Double DQN (DDQN)
- Dueling DQN (with and without DDQN)
- Double DQN (DDQN) with reward shaping
- Dueling DQN (with and without DDQN) with reward shaping

* RGB and multiple frames were only run on these examples

Figure 3: Seek and Destroy

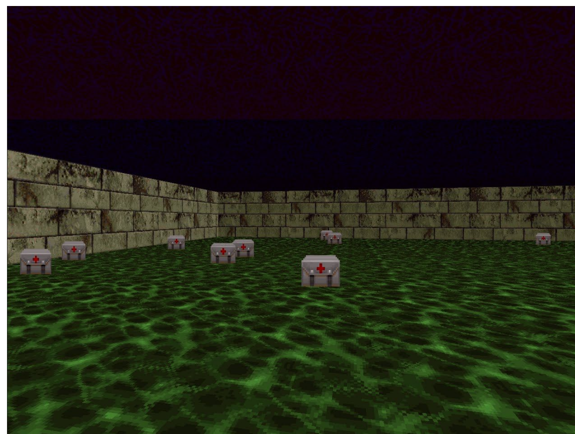

Figure 4: Health Gathering

In Seek and Destroy (Figure 3), our agent is tasked with moving towards the monster and shooting it as quickly as possible. The rewards for the scenario included a -1 reward per time tick to reward completing the scenario quickly, a -5 reward for a missed shot, and a reward of 100 for a successful hit. In the Health Gathering scenario (Figure 4), the agent is hurt each game tick and needs to collect health kits which randomly spawn in order to survive. The agent is rewarded with 1 point per game tick in order to encourage the agent to stay alive as long as possible.

Results

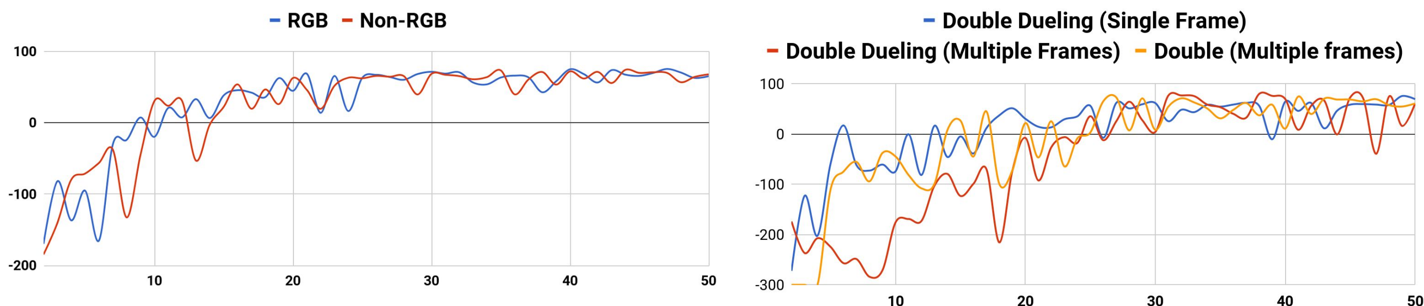

Figure 5: Average Validation Reward by Epochs for Seek and Destroy

Due to the relative simplicity of Seek and Destroy, we experimented with many variations to our DQN inputs in addition to the various modifications to the network architecture and training algorithm (Figure 5). In our first round of experiments, we sought to determine whether or not a more feature rich state space would improve performance of our agent.

First, we trained on Seek and Destroy with both RGB (3 channels) and grayscale (1 channel) to see if a 3x increase in our state space size would lead to better performance. As seen above, our DQN using RGB did not perform any better than our grayscale DQN and actually took longer to train to get the same results. Based on these results, we chose to only run our health gathering experiments using only grayscale in order to improve our ability to iterate on our experiments.

Our second set of experiments for the Seek and Destroy scenario were used to compare the performance of our various network configurations while also exploring whether adding a time dimension would lead to any improvements for this scenario. Although all of our configurations converged to an optimal policy within 50 epochs, the dueling DDQNs

converged quicker than DDQN. This was expected since the dueling architecture is designed to achieve faster convergence by training a state value function and advantage function separately. The single frame dueling DDQN converged faster than the multi-frame dueling DDQN, which is unsurprising because of the increased size of the state space when using multiple frames.

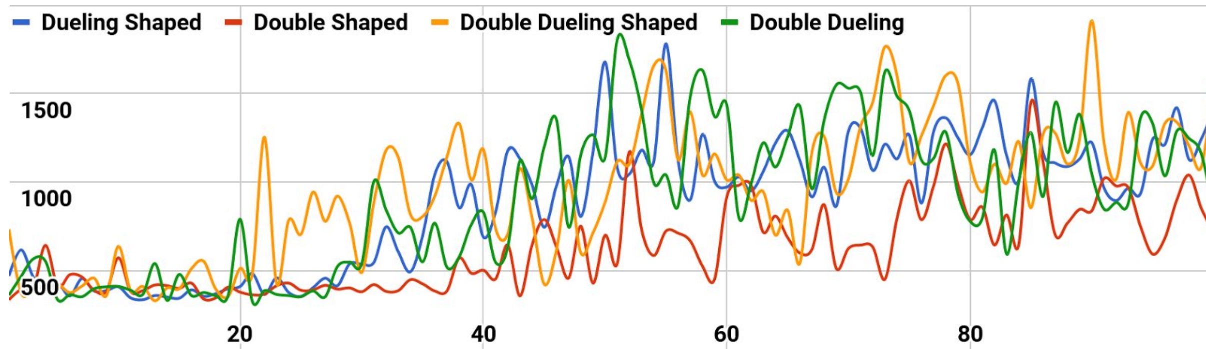

Figure 6: Average Validation Reward by Epoch for Health Gathering

Finally, we ran four different network configurations on Health Gathering (Figure 6) to compare the effect of reward shaping on different network configurations. For trials that used reward shaping, a small reward was added based on distance moved during training, which appears to have improved validation performance as well as reduced the number of epochs needed for convergence. This is clear when comparing the dueling DDQN with reward shaping to the dueling DDQN without reward shaping, as the average reward for the DDQN with shaping climbs more quickly than the standard DDQN, implying that continuous movement is important in the Health Gathering scenario.

Conclusions and Future Work

Through our experiments, we have shown that it is possible to train an agent to perform well in the non-trivial state space of the video game DOOM, using only information that would be available to a human player. Our agents were able to achieve optimal performance in both the Search and Destroy and Health Gathering scenarios using various training and network configurations. Through our experiments, we were able to determine that grayscale images contained enough information to successfully learn and performed no worse than trials using RGB frames, at least in simple scenarios. In addition, we were able to replicate the performance benefits of using DDQNs and dueling DDQN compared to standard DQN architectures in the DOOM environment. Finally, we explored the effect of reward shaping on agent performance and convergence time and showed that rewarding movement in the Health Gathering scenario reduces convergence time compared to training without reward shaping.

Potential future works that could be explored would include training on more complex scenarios to further refine our hyperparameters and neural network architecture. If successful, we could then train our agent to perform well in a general deathmatch scenario and potentially compete in the annual ViZDoom competition in Fall 2018. Other avenues that may be worth exploring include experimenting with novel DQN architectures that exploit the properties of MDPs or POMDPs in order to improve average rewards or reduce the number of training iterations required for convergence. These novel architectures could also be inspired by algorithms used to find optimal policies for a given system, such as point based value iteration for POMDPs.

Contributions

- Ben led the design and architecture of the neural network, experimenting with the number of convolutional and fully-connected layers, as well as hyperparameter tuning. As Ben was the most familiar with Tensorflow and ViZDoom coming into the project, he provided a tutorial on Tensorflow and ViZDoom to the rest of the team and served as the lead on any infrastructure-related issues. Ben programmed the base training and deep learning algorithm and reviewed code from other team members. Ben implemented reward shaping and the ability for the network to turn off/on RGB frames. Ben helped to implement multiple frames per training example, as well as saving the validation data into logs. Ben also ran training locally on his GPU for faster iteration and experimentation. Ben presented at the poster presentation.

- Daniyal contributed to the architecture of the network in collaboration with Ben. Daniyal also implemented the baseline models described above, and shared context regarding the performance of the model with the rest of the group. In addition, Daniyal programmed the model save/restoration logic using Tensorflow, reviewed code from other team members, and refactored code to improve development velocity. Daniyal and Tushar jointly implemented the model in Tensorflow. Daniyal further implemented the dueling network, code utilities (parsing command line arguments, saving different model weights), and also helped to implement multiple frames per training example. Daniyal put together and printed the poster, and presented at the poster presentation.

- Tushar contributed to the architecture of the network in collaboration with Ben. Tushar implemented Tensorboard to log debugging/performance data as well as visualize the model and convolutional filters. He also helped program the training algorithm (specifically, the target network) and reviewed code from other team members. Daniyal and Tushar jointly implemented the model in Tensorflow. Tushar led editing and citations for the production of the milestone. Tushar implemented the Double DQN. As he was the most familiar with cloud-based processing, Tushar led the deployment of training and validation runs for the final data onto cloud GPUs. Tushar led the writing of the report

- All group members contributed to the writing and editing of the final report.

References

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *NIPS Deep Learning Workshop*. n.d.Retrieved November 20, 2017, from https://arxiv.org/abs/1312.5602

Kempka, M., Wydmuch, M., Runc, G., Toczek, J., & Jaśkowski, W. (2016, September). Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on* (pp. 1-8). IEEE.

Alvernaz, S., & Togelius, J. (2017, August). Autoencoder-augmented neuroevolution for visual doom playing. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on* (pp. 1-8). IEEE.

Lample, G., & Chaplot, D. S. (2017). Playing FPS Games with Deep Reinforcement Learning. In *AAAI* (pp. 2140-2146).

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*, 529-533. (2015, February 26). Retrieved November 20, 2017, from https://www.nature.com/articles/nature14236

Ng, A., Harada, D., & Russell, S. (1999). Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. *International Conference on Machine Learning*, 278-287. n.d.Retrieved November 20, 2017, from https://dl.acm.org/citation.cfm?id=657613

Pineau, J., Gordon, G., & Thrun, S. (2003). Point-based value iteration: An anytime algorithm for POMDPs. *International Joint Conference on Artificial Intelligence*, 1025-1030. (2003, August 9). Retrieved November 20, 2017, from https://dl.acm.org/citation.cfm?id=1630806

Kingma, D. P. (2015). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*. (2014, December 22). Retrieved November 20, 2017, from https://arxiv.org/abs/1412.6980v8

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*.

Van Hasselt, H., Guez, A., & Silver, D. (2016, February). Deep Reinforcement Learning with Double Q-Learning. In *AAAI* (pp. 2094-2100).

Carr, Roger. "Simulated Annealing." From MathWorld--A Wolfram Web Resource, created by Eric W. Weisstein. http://mathworld.wolfram.com/SimulatedAnnealing.html

Ng, A., Harada, D. and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. [ebook] Berkely: University of California, Berkely. Available at: https://people.eecs.berkeley.edu/~pabbeel/cs287-fa09/readings/NgHaradaRussell-shaping-ICML1999.pdf [Accessed 15 Dec. 2017].