DEEP REINFORCEMENT LEARNING FOR VIZDOOM

Martin Simon (S163008), Niels Justesen† (S165591), Søren Jesper Bloch* (S123809)*

* Technical University of Denmark, † IT University of Copenhagen

ABSTRACT

(Martin) As Deep Reinforcement Networks have progressed fast, autonomous agents have started to outperform humans in simple games such as the ones made by Atari in the 80s. In our coursework we combined monochrome pixel data with game feature information to train a Deep Q Network to play Doom in its Artificial Intelligence Research Platform: ViZDoom. By training the policy in separate learning phases, first with a navigation reward and then with a shooting reward, the policy succeeded to show significant improvement in only 2x200 epoch leaning. The policy developed a very specific strategy and scored higher than average built-in bots. The strategy was especially successful in the ViZDoom death match scenario.

***Index Terms*—** neural networks, deep reinforcement learning, Q-Learning, first-person perspective games, policy gradient method, reward shaping, Deep Q Networks

1. INTRODUCTION (SØREN)

The goal of this project is to teach an AI how to play the ZDoom using deep reinforcement learning (an open source version of Doom 2 (a 3D shooter from 1995) that contains parts of the original source code) using nothing but the visual output and scoring variables using ViZDoom to determine the action to take. ViZDoom, an open source environment meant for creating and teaching an AI to play ZDoom, is utilized to help achieve this goal. In this report we will go through the various methods and variables and what result they yielded or is likely to yield. Deep learning has had a lot of success in other older games such as the old Atari games[1] and is in most cases better than linear learning, in a great number of cases the AI policy even reaches beyond human levels in terms of ability [2]. In this project we aimed to create an AI policy that goes beyond that of the default build in AI, in this we where successful although one may argue that is just because the policy prefer bigger weapons over smaller ones because of their ammo consumption. The project source code can be found here[1].

[1]https://github.com/njustesen/vizdoom-ai

2. DEEP REINFORCEMENT LEARNING

2.1. Markov Decision Processes (Niels)

This section will go through some of the theory on deep reinforcement learning that was used in our experiments. In reinforcement learning the goal is to find an optimal policy for a Markov Decision Process (MDP). A MDP consists of a set of possible states S, a set of available actions A, a probabilistic state transitioning function $P_a(s, s')$ determining the probability of transitioning from s to s' by taking action a. Finally an MDP also has a reward function $R(s, s')$ that determines the reward of transiting to states. The problem in MDPs is to find an optimal policy π that maximizes the expected future reward. The expected future reward when being in state s using policy π can be described using the Bellman equation:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^{\pi}(s') \qquad (1)$$

where $\gamma \in [0 : 1]$ is a discount factor that gives rewards in the near future higher value than rewards in the far future. Two reinforcement learning methods were implemented in this project and will be described below using the terminology of MDPs and the Bellman equation.

2.2. Deep Q Networks (Niels)

Google's DeepMind has recently achieved human-level control in a range of Atari games using a deep reinforcement learning algorithm called Deep Q Networks (DQN) [1]. DQN implements the traditional Q-learning algorithm using a deep neural network such as a convolutional neural network (ConvNet). Such networks are able to learn complex policies from just raw pixel inputs. In Q-learning a Q-function $Q(s, a)$ is learned that estimates the discounted future reward by taking action a in state s. A policy can then use the Q-function by taking the action that leads to the highest future reward $\max_a Q(s_t, a)$.

Agents observe and act in discrete time intervals in MDPs. For each action made by an agent a so-called trajectory $< s_t, a.s_{t+1}, r_{t+1} >$ is obtained, where action a taken in state s_t results in state s_{t+1} and a reward r_{t+1}. Using this trajectory

the Q-function is incrementally updated using the Q-learning update rule, which is based on the Bellman equation. First the learned value y is calculated from our trajectory:

$$y = r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) \tag{2}$$

Then the error δ is calculated which is the difference between the learned value y and our current estimation of $Q(s_t, a_t)$.

$$\delta = y - Q(s_t, a_t) \tag{3}$$

Finally $Q(s_t, a_t)$ is updated with δ using a learning rate $\alpha \in [0 : 1]$.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \delta \tag{4}$$

In DQN the Q-function is a neural network and is updated using Stochastic Gradient Descent (SGD) with backpropagation. The error backpropagated is δ and SGD uses α as the learning rate.

2.3. Experience Replay (Niels)

A key improvement to DQN is *experience replay* [1]. The naive approach in DQN is simply to collect n consecutive trajectories, where n is the batch size, and backpropagate the error from this mini-batch. The issue with this approach is two-fold. The first issue is that it learns from very correlated trajectories that will produce very inaccurate updates and ultimately result in slow learning ending in local optimum. The second issue is that trajectories are only used for learning once and then deleted, while they just as well could be reused.

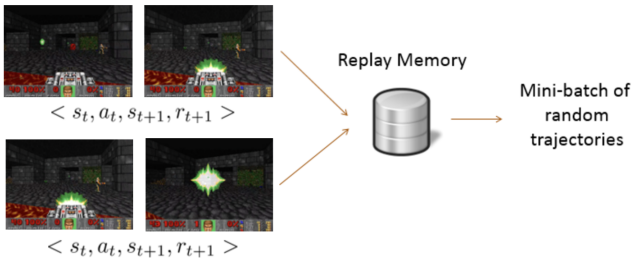

Fig. 1. Trajectories are added to a replay manager as the agent interacts with the environment. At some fixed interval a mini-batch of randomly selected trajectories are pulled.

The experience replay method solves these two issues by storing observed trajectories in a replay memory. The memory will have a fixed size and the oldest trajectories will be overridden when it is full. A mini-batch of trajectories are at some fixed interval randomly pulled from the replay memory and used by the SGD.

2.4. Deep Recurrent Q Networks (Niels)

In some environments it is not possible to capture the entire state from a single time step. It is e.g. not possible to capture the speed and direction of moving objects from a single frame in a video game. A single frame in VizDoom also shows only a very limited part of the level and it is therefore often said that the Q-function estimates observation-action value $Q(o, a)$ instead of the state-action value $Q(s, a)$. For some environments it is necessary to use multiple consecutive observations as input to the DQN. This approach was also used by [1] where four frames were stacked as input to the network.

Another approach that was implemented in our project is the Deep Recurrent Q Network (DRQN) which has been shown to be superior to the stacking DQN approach [3] in some games where observations only show the partial state. In DRQN a recurrent layer is added to the network just before the output layer to implement a many-to-many Recurrent Neural Network (RNN) architecture. DRQN was also used by [4] with Long Short-Term Memory (LSTM) cells to achieve good results in ViZDoom.

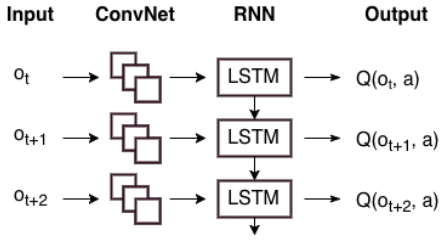

Fig. 2. How the hidden state of the recurrent layer in DRQN is propagated through time following the many-to-many RNN architecture. In our model LSTM gates were used.

Training of the DRQN can be done in several ways. We choose to focus on the approach by [4]. During training sequences of trajectories of length n are sampled from the replay memory. The input shape thus becomes [$bath\_size$, n, $frame\_width$, $frame\_height$, $color\_channels$]. The trajectory sequences are then fed through the network, but only the last part of the outputs are used for backpropagation while the first part is used only to initialize the hidden state. The first part is called *observation history*. After the policy is trained the hidden state of the network is stored after each activation of the network and is feed as an additional input for the following activation.

2.5. Policy Gradient (Martin)

Instead of using value functions to optimise the loss function the Policy Gradient Method was also implemented. This method uses real-time gained experience by evaluating the policy gradient change direction to make available action decisions.

3. APPROACH

3.1. Models (Martin)

The monochrome 48x64 pixel data was imported to the first layer (see Figure 3). Two convolutional layers with respectively 32 and 64 filters were used for training with this input data. The output was projected to a fully connected layer and then translated to four actions (move left, right, forward and shoot).

The DRQN model used in the experiments were identical to the DQN model but with on additional layer added. A layer of 512 LSTM cells were added after the fully connected layer. The number of cells were picked to be identical to the size of the previous layer.

3.2. Root Mean Squared Gradient & RMSprop (Martin)

In optimization functions RMSProp with size 64 mini batches was used for gradient calculations. It is largely recommended as a basic method for learning methods in a large neural network with a large redundant data set. RMSProp is a method of using the sign of the gradient combined with the root of the nearby averaged square gradients to decide on the step size. This makes it more usable for mini-batches unlike RProp, which uses only the sign of the gradient. Although Google DeepMind improved the factors in the regular gradient, there was mild improvement compared to the regular RMSProp when applied to a similar Q-Learning tasks and only showed better performance after millions of epochs as it continued to learn [5] [2].

3.3. Divide and Conquer (Søren)

In order to create a fully functional AI, the AI will need to try a lot of different combinations for different actions, and these actions may not be related to one another, which means that the complexity of the learning process can be reduced by splitting the process into smaller more direct tasks. By dividing up each of the task that is sought to be accomplished, and then conquering each of them individually, a result may be reached faster than without. In our case we utilized this by teaching it to walk and explore and shoot separately. By teaching it to walk first, we were able to have it reach scenarios were is possible to shoot the enemies, therefore it does not need to guess the walking procedure and can focus on learning where to aim and when to pull the trigger.

More specifically, we first trained the policy with one reward function using the ϵ-greedy strategy with ϵ going from 100% to 10%. After this first phase of training the learned policy (or network parameters) was trained further with a new reward function and ϵ going from 50% to 10%, thus utilizing what was learned in the first phase during training in the second phase.

[2] http://maciejjaskowski.github.io/2016/03/09/space-invaders.html

3.4. Navigation Reward (Niels)

A reward function was implemented to make the agent navigate the level in the deathmatch scenario. To access the players position we had to change the ViZDoom source code and recompile it. The naive approach to train an agent to navigate would be to give a reward when the game ends relative to the total distance travelled by the agent. This has two issues; (1) it creates very sparse rewards that will be difficult learn and (2) walking in circles is just as good as exploring all the different areas in the level. Our approach gives a reward at every time step in the game and solves these two issues. It uses the following formula to calculate the reward:

$$R(s_t, s_{t+1}) = \sum_{\tau=0}^{t+1} \beta^{t+1-\tau} \big(|p(s_{t+1}) - p(s_\tau)| - |p(s_t) - p(s_\tau)|\big) \tag{5}$$

where $p(s)$ is the position of the player in state s and $\beta \in [0:1]$ is a constant that determines to what degree movement away from recent positions are rewarded higher than movement away from past positions. This reward function rewards movement away from previous positions by comparing the distance to all previous positions at $t+1$ by the distance to all previous positions at time t. Several experiments were attempted to come up with this solution.

3.5. Shooting Reward (Martin)

As the implementation of the model trained with exploration reward increases the probability of finding bots, a function was applied to reward decrease of ammo when enemies were detected.

$$R(s_t, s_{t+1}) = \big|a(s_t) - a(s_{t+1})| \cdot e(s_{t+1}\big) \tag{6}$$

where $a(s)$ is the amount of ammo available for the player in state s and $e(s)$ the amount of enemies in range.

3.6. Damage Reward (Søren)

Since the damage necessary to kill an opposing player is not necessarily meet with the weapon equipped, the AI will not necessarily learn how to kill enemy players. however if it is rewarded for damaging enemy players, it is more likely to approach a solution were it will kill the enemy player. In order to accomplish this we had to modify the very core of ViZDoom, as it does not normally give this output, so the memory buffer which returns the desired variables to us had to be altered to contain this info, and we had to find one of the areas were damage is given, which we found but it is utilized by everything, causing the code to currently have issues with environmental damage, causing an unhandled exception, we mainly see this when the player tries to go through lava. Our changes to ViZDoom can be found on GitHub[3].

[3] https://github.com/neroes/ViZDoom

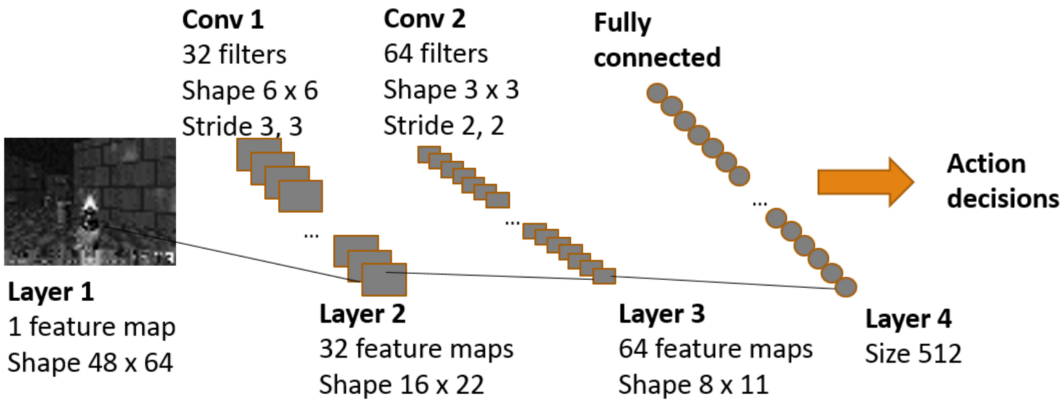

Fig. 3. Illustration of DQN Model layout.

4. RESULTS

4.1. System Specifications (Søren)

In order to test these ideas we used 2 different setups, system 1 was a machine to which Niels was granted access in order to run these simulations from ITU, and system 2 was a private machine built by Martin.

	Test System 1
CPU	Intel(R) Core(TM) i7-5820K CPU
GPU	2 x GeForce GTX TITAN X
RAM	32GB 2666MHz DDR4 Ram

	Test System 2
CPU	Intel(R) Core(TM) i7-3720QM CPU
GPU	GeForce GTX 750 Ti
RAM	24GB 1600Mhz DDR3

4.2. Basic Scenario (Niels)

Our first experiments were made on the basic ViZDoom scenario where the the agent is located in a room with one enemy. If the enemy is hit it is killed and the agent is rewarded with 100 points. For every bullet used the agent is given a negative reward of -5. Our DQN model was able to learn an almost perfect policy for this problem within 30 minutes on Test System 1 using the CPU (see Figure 6). The configurations used were the following:

- Replay memory size = 100,000
- Frame skip = 4
- Update frequency = 4
- Learning steps per epoch = 2000
- Batch size = 64
- Learning rate = 0.00025
- Discount factor = 0.99

The policy gradient method was also tested in this problem several times but always ended in a local optimum such as shooting without moving. Several experiments were made with different hyper-parameters such as the following:

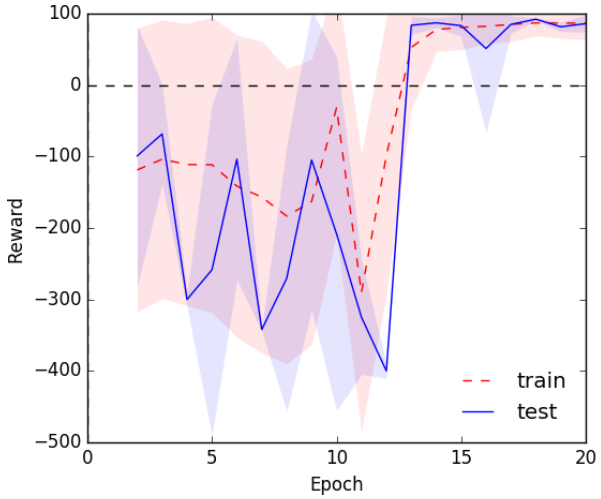

Fig. 4. Rewards per. episode in the basic scenario with DQN. Opaque fill shows standard deviation.

- Frame skip = 4
- Learning rate = 0.1
- Discount factor = 0.99

The DRQN model was also tested in this problem without success. We expected that it would take much longer to train, but 3 hours of training on Test System 1 with GPU showed no sign of progress at all. It is unclear to us why it failed to learn. The configurations were the same as for the DQN but with the following additions:

- Observation history = 4
- History = 10

Based on these results from the basic scenario we decided to focus only on DQN. We expect that with more debugging

and testing the DRQN model should work just as well as the DQN model for this problem.

4.3. Deathmatch Navigation (Niels)

The DQN model was tested in the deathmatch scenario without enemy bots using the navigation reward (see Section 3.4). The same configuration as in Section 4.2 were used. For the navigation reward function the β constant was set to 0.95. The training ran on for $\sim$8 hours on Test System 1 and achieved a satisfying result. The trained policy was able to navigate through more than half of the level in a very human-like manner with some distinct issues. Sometimes, but rarely, it can get stuck by continuously walking into an explosive barrel. Another issue is a corner in the level where there is a mirror and the agent has some difficulties moving away from it. An example of the navigation policy can be seen on YouTube[4].

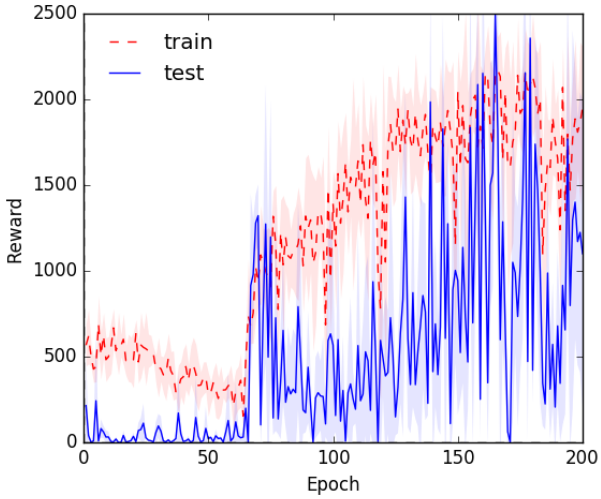

Fig. 5. Exploration rewards per. episode obtained by DQN in the death match scenario. Opaque fill shows standard deviation.

4.4. Deathmatch Shooting (Martin)

The trained policy swiftly learned to move in the center of a room full of enemies to shoot with its most ammo-consuming weapon. This way it managed to spend a large amount of ammo while in presence of many opponents. While the trained policy continued to get stuck sometimes, as exploration was not further improved in this training model, the improved model was now competitive with built-in bots by reaching approximately four kills per death during training. Although this kamikaze-style strategy would not be competitive against human players, it is a clear sign of progress when taken into account only 2x200 epochs of training.

[4]https://www.youtube.com/watch?v=aSoToirhlhc

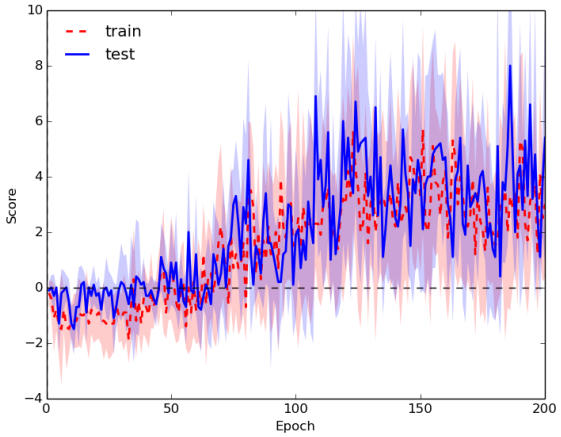

Fig. 6. Score of kills per death during Deathmatch Shooting training.

The learned policy can be seen in this YouTube video[5].

4.5. Final Evaluation (Martin)

During final evaluation the trained model was set against seven built-in bots for 100 2-minute long games. As a result it averaged 3.4 kills per death and ranking on third place in the Hall of Fame.

Fig. 7. Example end-game screen where our agent won against the build-in bots.

[5]https://www.youtube.com/watch?v=oGt34SxL9h8

5. DISCUSSION (SØREN)

While a perfect result where the machine learned the full relationship between killing an enemy and winning has not been found, we have however manage to get a kill/death ratio well above over 1[6]. Meaning that our player is stronger than the average build-in bot, although it's caused by the player trying to shoot as many bullets as possible with as many as possible enemies on screen as possible, causing it to pick the weapon with the largest bullet consumption, these weapons have a large area of effect. The Damage reward would likely yield more direct and more efficient especially in cases of non splash damage weapons, as we currently just shoot in their general direct not caring about the placement of the enemy on screen, we just care that they are there. Additionally the AI policy may be improved by various other measures, such as making it play against an earlier AI policy to train it further, in order to make it more capable at surviving and scoring against harder opponents. Although this would still not teach the AI policy to shoot its enemies rather than just shooting big weapons in their general direction, unless it gets a metric for damaging. Because we have so many ways of evaluating how well the AI policy has faired, we can most likely save on training time by going thought the various ways of scoring and then using the previous as the next ones starting, for instance we could start with the shooting reward, which makes us shoot in the right direction, then move onto damaging which would teach us to damage the enemy and finally move onto killing the enemy which is what we ultimately want the AI policy to do.

6. CONCLUSION (MARTIN)

During this project we successfully trained a policy to outperform built-in bots of a first-person shooter game Doom by implementing the Deep Q Network (DQN) algorithm. Experiments with the Deep Recurrent Q Network architecture as well as the Policy Gradient method did however not show any positive results.

Two strategies for DQN were implemented. First a naive end-to-end training and then a divide-and-conquer strategy. Experimenting with a naive end-to-end training failed and the failure can be attributed to killing a bot being an extremely random event. Using the divide and conquer strategy the policy was first trained to explore the level and thereafter rewarded for ammo decrease while enemies were on the screen. Despite training for a limited amount of epochs due to lack of computational resources, the algorithm developed a clear winning strategy. This illustrates the necessity of having a clear understanding of the task handed to a network.

[6]kill/death ratio shows the relationship between a players kills and death, meaning that if it is above 1, they player generally kills more than he dies and assuming that each opponent is of the same strength the player is better than average

Further improvements can be made by (1) increasing the number of epochs and (2) train the network against itself and (3) reward the agent when it deals damage to opponents.

7. REFERENCES

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] Matthew Hausknecht and Peter Stone, "Deep recurrent q-learning for partially observable mdps," *arXiv preprint arXiv:1507.06527*, 2015.

[4] Guillaume Lample and Devendra Singh Chaplot, "Playing fps games with deep reinforcement learning," *arXiv preprint arXiv:1609.05521*, 2016.

[5] Mateusz Kurek, "Deep reinforcement learning in keepaway soccer," .