Udacity Capstone Project

Playing Doom with deep reinforcement learning

1. Project Definition

The goal of the Project is to train an agent to play a basic scenario of the video game Doom. Doom is a first-person shooter developed by id Software in 1993. “It is considered one of the most significant and influential titles in video game history”[1]. To be able to effectively train an agent on a 3D video game, I will use reinforcement learning with a deep neural network. This is approach is known as a deep Q-network and was pioneered by DeepMind.[2] DeepMind published a famous Article in the Journal Nature with the title “Human-level control through deep reinforcement learning”[3]. In this article they describe their novel approach in training an agent to play a wide variety of Atari Games.

The goal of this Project is to determine if their approach is also able to learn the game Doom. For this purpose, I use the ViZDoom environment[4]. This environment was developed as an AI research platform. For this purpose, it allows direct access to the video screen buffer and offers API’s for the programming languages C++, Python and Java.

To be able to determine if a deep Q-network is able to learn to play Doom, I will use the most basic scenario offered in the ViZDoom environment. To measure performance of the learning agent I will test the trained agent against an agent which will only use random chosen actions. As a metric to measure performance I will use the average earned reward after 20 episodes for the random agent against my trained agent. I expect the trained agent to clearly outperform the random agent.

Another goal of the implementation is to create a fast learning agent. DeepMind trained their agent over 100 epochs and “one epoch corresponds to … roughly 30 minutes of

1 https://en.wikipedia.org/wiki/Doom_(1993_video_game)

2 https://deepmind.com/

3 http://home.uchicago.edu/~arij/journalclub/papers/2015_Mnih_et_al.pdf

4 https://github.com/Marqt/ViZDoom

training time”[5]. I hope to create a simple agent which can learn faster, so it can be used as a baseline implementation for iterative purposes.

2. Analysis

Environment

In the ViZDoom environment I used the basic scenario, which looks like the following:

On Startup the player, our agent starts in a center position of a room. Only the hand and the gun of the agent is visible. The target is on the other side of the room, which is not moving in this basic scenario. The goal of the agent is to shoot the target; the target will be killed with one hit. The agent only has one weapon with unlimited ammunition.

The agent is able to freely move inside the environment room and to shoot at any given time. Each move is called an action, and can contain a move to the left, to the right, forward,

5 https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

attack or a combination. I restricted the agent only to use a move left, move right or attack without a combination. The reason for this is discussed under 3. Methodology.

The ViZDoom environment has no restriction on how fast the agent has to shoot the target. But to be able to learn a reasonably policy I set a timeout after 300 actions. This means the agent has to kill a target within 300 action after this 300 actions the environment will restart. The environment will also restart when the agent successfully shoots the target. After the restart the agent will respawn in the center of the room and the target on a random position on the other side. The point between start and end of the environment is defined as one episode.

The goal is to let the agent learn an optimal policy only from observing the raw pixels. The only other input the agent is feed with is the reward from the environment. The reward is structured the following way:

- For moving left or write the reward is -1
- For shooting the reward and missing -6
- For killing the target +100

For every new frame the agent is receiving it has to choose an action and it is not allowed to do nothing.

Based on this reward structure we can conclude; an optimal episode can be rewarded with 100 points. This could only be the case if the target spawns directly in front of our agent. A reasonable average for an optimal policy should be around +70.

An agent that only uses random chosen actions, is able to archive an average earned reward from -70 after 20 played episodes. That means we should see that an agent while training will start at an average earned reward around -70 and will learn to archive an average earned reward at around +70.

Algorithms

For training the agent I used reinforcement learning with a Deep Q Network. To understand how this works, one must think of Doom as a Markov decision process (MDP). In a MDP we have an Agent, which is our character in Doom, which is suited in our ViZDoom environment. The environment is in a certain state (e.g. positon of the agent, position of the enemy,

existence of the walls and so on). The agent can perform the actions move right, left or shoot in the environment. These actions result in a reward. Actions transform the environment and lead to a new state, where the agent can perform another action. The rules for choosing actions are called policy.

The goal of the agent is to maximize its total reward. To perform well in the long-term, we need to take into account not only the immediate rewards, but also the future rewards we are going to get.

To archive this, we use the Q-learning technique. In Q-learning "the action that is optimal for each state is the action that has the highest long-term reward. This reward is a weighted sum of the expected values of the rewards of all future steps starting from the current state "[6]. It also applies a discount factor which trades off the importance of sooner versus later rewards and a learning rate that determine to what extent the newly acquired information will override the old information. The algorithm is a value iteration update.

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

To implement this algorithm for the Doom environment there is the problem of how to represent the states. The most general idea one can come up with is to use all the pixel of the screen. With this approach there is no need to manual handcraft states for our environment. It is also closest to how a human would play the game. A human sees a screen image which are all the visible pixels and based on this information he or she chooses the next action. The problem on using only one screen is that it does not contain information of direction or speed of movements. To solve this DeepMind uses four consecutive screens.

There is one big problem in this approaches. DeepMind resizes the screen to 84 x 84 pixel converted to grayscale with 256 gray levels. That would yield to $256^{84x84x4} \approx 10^{67970}$ possible game states, which is more than the number of atoms in the universe. So it is impossible to calculate a Q-Value for every possible state. To solve this problem DeepMind uses a deep

6 https://en.wikipedia.org/wiki/Q-learning

convoluting neural network (CNN) which is exceptional good in helping with this kind of problem.

CNN are often used for image classification problems. For solving the kind of Problems CNN are made up of artificial neural that have learnable weights. Each neuron receives an input, performs a dot product and uses an activation function (eg. sigmoid function[7]). In a CNN there are many so called hidden layers of neurons. This hidden layers are created by striding a filter through the input image which create an activation map of neurons. Through this layer one can stride again a filter to get an additional activation map. The size of the filter, number of hidden layer, kind of activation function and so on are all hyper parameter. The created CNN is a single differentiable score function with has a loss function on the last layer, so the network can be trained using backpropagation[8].

With this architecture DeepMind designed a CNN which takes in four screen images and outputs the Q-values for each possible action. Q-values can be any real values, which makes it a regression task so the loss can be defined as a squared error loss, between the real Q-value and die prediction from the CNN.

$$L = \frac{1}{2}[\underbrace{r + max_{a'}Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

One import trick DeepMind uses is experience replay. During playing all the MDP property's <*s, a, r, s*> are stored in a replay memory. The CNN is than trained on a random mini batch from the replay memory after every new frame. This technique is used for avoiding to drive the CNN into a local minimum. The hole deep Q-learning Algorithms is then:

7 https://en.wikipedia.org/wiki/Sigmoid_function

8 https://en.wikipedia.org/wiki/Backpropagation

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

Methodology

To implement the CNN, I used Googles open software library TensorFlow. I used the GPU version of TensorFlow. Training was done on a PC with a Core i7 4770k with 16 GB RAM and a NVIDIA-GeForce GTX 970.

The goal of the implementation was to learn the basic scenario in the fastest training time possible. I choose this approach to establish a baseline learning agent on which one can build up a more complex agent.

For the learning agent I set the render resolution in the ViZDoom environment to 640x480 pixel and to grayscale images. Also the rendering of the environment is set to not be displayed on the screen, this has a profound performance impact on the learning agent. Without displaying the screen there can be rendered more frames per second, therefore training takes less time. For the trained agent displaying is activated.

For my first Iteration of the learning agent, I implemented it as closely as possibly to the approached used by DeepMind. I used three convolutional layer and two fully connected layer at the end. I choose to allow all possible action combination, which are left, right and shoot so this yield $2^3 = 8$ possibility's. Which are the output of the last fully connected layer

in the CNN. For calculating gradient decent I used the Adam optimizer[9] with a learning rate from $1e^{-6}$. Input to the CNN were four consecutive 84x84 pixel images. With this architecture training took roughly 24 Hours. Which was too much time for my goal, for a fast agent for iterating purposes.

To reduce training time, I first choosed a network where I only used two convolutional layer instead of three. Then I resized the screen from 84x84 to 40x40. To node here, the ViZDoom environment can only display aspect ratio like 4:3 or 16:9. I reshape the image to a 1:1 aspect ratio, because the CNN GPU implementation in TensorFlow seems to be faster with an even aspect ratio. In the optimizing process, I found out that the agent is also able to learn with only one screen. This is because the agent only has to learn to get in front of the enemy and then to shoot, there is no speed or movement information required. Additionally, I restricted the agent to three action possibility's move left, move right or attack, to make the last fully connected layer smaller and those significantly reduce training time. I also downgraded the first fully connected layer from 512 filter to 256 filter. For calculating gradient decent I use the Adam optimizer with a reduced learning rate from $1e^{-5}$. My final Network architecture is as follows:

Layer	Input	Filter size	Stride	Num filters	Activation	Output
Conv1	40x40x1	8x8	4	32	ReLu	9x9x32
Conv2	9x9x32	4x4	2	64	ReLu	3x3x64
Fc3	3x3x64			512	Relu	256
Fc4	256			3	Linear	3

I also implemented a technique used in the DeepMind paper called frame-skipping. Where the agent sees and selects actions on every k^{th} frame instead of every frame, and its last action is repeated on skipped frames. This technique allows the agent to play roughly k times more games without significantly increasing the runtime. I used k = 7, which is the number of frames rendered between the agent fires a shoot and hitting the target.

I used an ε-greedy exploration with probability ε choose a random action, otherwise go with the "greedy" action with the highest Q-value. The first 25.000 frames the agent only makes random moves to fill the replay memory which I assigned a size of 50.000 MDP property's. After this first 25.000 frames the agent starts training on a random 64 MDP mini-batch from

9 https://arxiv.org/pdf/1412.6980.pdf

the replay memory. I was able to double the mini-batch size compared to DeepMind, because of my smaller input to the network. One mini-batch has to fit on the GPU memory, which is the limiting factor of CNN learning.

In the learning state I decreases ε over 1.000.000 frames from 1.0 to 0.05. I defined one epoch as 10.000 frames and totally trained the agent over 150 epochs.

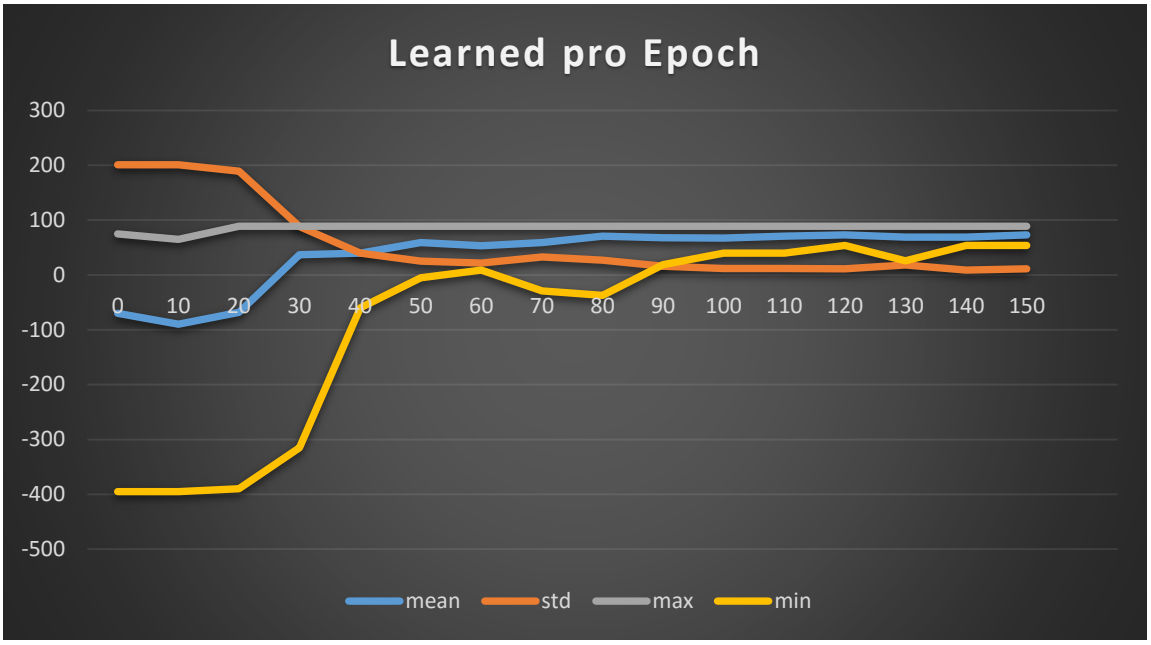
I implemented the network saving function from TensorFlow to save the network weights every 10 epochs up to 150. Then I developed a playing Agent that only chooses actions based on the Q-Values output from the learned CNN. To measure performance of the playing agent I let him play for 20 episodes for every saved weights. For the 20 episodes I calculate the gained rewards in form of the average earned reward, the minimal and maximal reward achieved and the standard deviation of the reward.

3. Results

I tested with one agent that chooses action only randomly to establish a baseline. This random agent is listed as epoch 0 in the following table.

Epoch	avg	Std.	Max.	Min.
0	-70	201	75	-395
10	-90	201	65	-395
20	-68	189	89	-390
30	37	88	89	-315
40	40	40	89	-60
50	59	25	89	-5
60	53	22	89	9
70	59	33	89	-29
80	71	27	89	-37
90	68	16	89	19
100	67	12	89	40
110	71	12	89	40
120	73	11	89	54
130	69	18	89	26
140	69	9	89	54
150	73	11	89	54

The random agent slightly outperforms the trained agent after 10 epochs, but even after 20 epochs of training the trained agent gets the upper hand. At around 110 epochs the agents have learned a nearly optimal policy. I did play this basic scenario myself and was on pair with the result achieved from the trained agent. For visualization the results as a graph:



I was able to run one epoch of training mini-batch updates of my hardware in around 1:03 minutes. So training for 150 epochs took 157 minutes. And as shown in the result, after only 30 minutes of training, one can validate if the agent is able to learn.

The faster learning time compared to DeepMinds approach is based on two factors. At first, through the smaller CNN there are not so many training examples required. The second factor is that the Atari environment DeepMind uses is limited to displaying 60 frames per second. There is no such limitation in the ViZDoom environment, it can display as many frames as the CPU is able to calculate per second.

4. Conclusion

My approach shows that the technique established by DeepMind can also be used for other Games. My agent is able to learn an optimal policy for the video game Doom. I was also able to establish a fast learning agent on which one is able to add additional features.

One particular aspect where I was struggling with, during the development of my agent was frame-skipping. I had the problem that when this parameter was not correctly set the agent would not learn any policy. I figured out that the problem was, that the agent was shooting at the target than it would do one or two additional other actions and only then get the

reward. To found out this Problem I had to save many frames and view than directly to find this problem.

One of this feature could be the double Q-Learning approach which was published by DeepMind in December 2015. That “shows that the recent DQN algorithm, which combines Q-learning with a deep neural network, suffers from substantial overestimations … We then show that the idea behind the Double Q-learning algorithm, which was introduced in a tabular setting, can be generalized to work with large-scale function approximation.”[10]

Another interesting addition to the Q-Learning algorithm is the prioritization of experience replay. Published in January 2016 “Experience replay lets online reinforcement learning agents remember and reuse experiences from the past. In prior work, experience transitions were uniformly sampled from a replay memory. However, this approach simply replays transitions at the same frequency that they were originally experienced, regardless of their significance. In this paper we develop a framework for prioritizing experience, so as to replay important transitions more frequently, and therefore learn more efficiently.”[11]

One approach I find particular intersecting to improve the algorithm was presented at ICML 2016 in the article “Dueling Network Architectures for Deep Reinforcement Learning”.[12] They authors present a network architecture, which separates the representation of state values and (state-dependent) action advantages. These two independently learned functions, uses the same convolutional layer and outputs together the Q-values for each action. With this approached they were able to improve the score on 75.4% (43 out of 57) of the tested Atari Games.

With this added algorithms one can try to learn more complex scenarios in the ViZDoom environment or try to compete in the OpenAi Gym.[13]

10 http://arxiv.org/pdf/1509.06461v3.pdf

11 http://arxiv.org/pdf/1511.05952v3.pdf

12 http://arxiv.org/pdf/1511.06581v3.pdf

13 https://gym.openai.com/