Playing Doom with Deep Reinforcement Learning

Manan Kalra
SoCS, UPES Dehradun
India

J. C. Patni
SoCS, UPES Dehradun
India

ABSTRACT
In this work, we present a deep learning model based on reinforcement learning that is tied to an AI agent. The agent successfully learns policies to control itself in a virtual game environment directly from high-dimensional sensory inputs. The model is a convolutional neural network, trained with a variant of the Q-learning algorithm, whose input is raw pixels and whose output is a Q-value directly associated with the best possible future action. We apply our method to a first-person shooting game - Doom. We find that it outperforms all previous approaches and also surpasses a human expert.

Keywords
Machine Learning, Reinforcement Learning, Q-Learning, DQN, CNN

1. INTRODUCTION
Reinforcement Learning can hone the power of machine Intelligence and is the closest technique via which humans have come to building a real AI. Game-playing agents make use of the same technique to mimic the learning process of a human brain. They can be regarded as learning machines which can learn from experiences to grow , i.e. by building itself dynamically and adapting to its knowledge base. Deep Neural Networks help those agents to achieve exceptional performance benchmarks which are even impossible for humans. By implementing state-of-the-art reinforcement learning techniques and combining its power with deep neural networks, we plan to develop an understanding of how gaming agents can outsmart mimic the learning process of a human brain, and finally outsmart human intelligence [1]. We initially develop this for Doom, which is a 1993 video-game. However, the same implementation can be trained on various other gaming environments by tweaking the input action space.

1.1 Bellman Equation
Bellman Equation is a mathematical formulation to help an AI agent to choose the best action in the current state in order to maximize future rewards. It takes into account a discount factor (gamma) which helps find optimal values for each state of the environment. The equation is: $V(s) = \max_a (R(s, a) + \gamma V(s'))$. Here, s - State, a - Action, R - Reward, γ - Discount Factor.

1.2 Markov Decision Processes
Markov Decision Processes take care of scenarios where the outcomes are partly random and partly under the control of an agent [2]. Concisely, they provide a mathematical framework for modeling decision making in such situations. Thus, the Bellman equation can be modified as follows after taking the stochastic probabilities of events under consideration: $V(s) = \max_a (R(s, a) + \gamma \sum P(s, a, s') V(s'))$.

1.3 Policy VS. Plan
An AI agent after a certain amount of training starts to devise certain policies in the environment that lead to achieving the goal or the maximum possible reward. This is where AI agents are capable of outsmarting a human and can even take combination of steps or make a strategy to win the game in a way that humans can't even think of [3]. On the other hand, a Plan is a thought-out fixed path that the agent can follow to reach the goal. However, this approach doesn't involve the other factors that can have a significant impact on the final reward, and can sometimes also result in the agent not reaching the goal at all and get stuck in a local minimum [4].

1.4 Living Penalty
A Living Penalty refers to a certain negative reward that an agent gets on each progressing to every state that is not included in its policy. For example: to explore more, the agent is allowed to take random steps even after a policy that leads to the final goal is decided. This is necessary to acknowledge the possibility of finding a better policy or path than the previous one. In case, that random behavior leads the agent further away from the goal, it gets a negative reward. Higher the living penalty, less amount of time will be spent by the agent on exploring other parts of the environment again. So, the policies depend on the magnitude of living penalty provided.

1.5 Deep Convolutional Q-Learning
Q-Learning is a reinforcement learning procedure which attempts to learn the value of being in a given state. It tells us the utility of each state in the environment and also decides on the action that an AI agent shall take to maximize final reward by moving to a state with higher utility.

A temporal difference is the difference between the new and the old Q-value that is calculated after the agent changes a state and receives a certain reward. The method of temporal differences is used to update Q-values while the AI agent is exploring the environment and receiving continuous rewards based on its performance [5]. A learning rate parameter decides how fast the AI learns, i.e. upto what extent the Q-values are updated. However, in complex environment such as Doom, this method is very slow and we need to combine the typical Q-learning method with a neural network, and this is where Deep Q-Learning comes in. This is needed because there are endless combination of possible states in such environments and it becomes difficult to deal with them with such an intuitive approach.

As when we humans play a game, we are capable of using our sense of sight to get an insight of the current state of the environment. Providing numerical values as a vector for each change in the current state to an ANN proves to be an added advantage to a computer system against us. In order to make our AI capable of thinking as close as possible like a human brain, we have made use of Convolutional Neural Networks instead. With this approach, the images of each frame are fed as an input to feature detectors. The detected features are then max pooled and finally passed on as a flattened vector after applying an activation function [6]. Therefore, our AI also looks at the environment from a human perspective by looking at the environment. To sum it up, Deep Convolutional

Q-Learning involves leveraging the power of convolutional neural networks to update Q-values.

1.6 Experience Replay and Eligibility Trace

After running the training algorithm, a random selection from all the experiences is gathered and an average update for neural network weights is created which shall maximize Q-values or rewards for all actions taken during those experiences [7]. Since very early experiences are not so important, only a fixed number of past experiences are tracked and rest are forgotten. This process is termed as experience replay. Eligibility Trace is when the agent makes an arrangement of steps, rather than just 1 step at a time before calculating the reward it has achieved. The agent keeps a trace of eligibility during the process. For example: if there is a step that provides a negative reward it monitors that progression and tries to avoid it.

2. MOTIVATION

Tuning algorithms on self-learning procedures can significantly help to solve a wide variety of business problems. Virtual worlds such as gaming environments are a fertile training ground for AIs to learn before being released into the real world. The strategies used to conquer games may also allow us to conquer unrelated domains, such as, cancer diagnosis, climate change mitigation, financial investment decisions etc. Also, by building machines capable of thinking and acting like a human, we may move closer to the ultimate benchmark of machine intelligence: a machine that passes the Turing Test.

3. PROPOSED METHOD

OpenAI's Gym provides environment bundles for various gaming scenarios. We used the DoomCorridor-v0 environment. In this environment, there are 6 enemies (3 groups of 2), and there is a vest at the end of the corridor. Our goal is to reach the vest as soon as possible, without being killed by the enemies.

The action space consists of the following: ATTACK, MOVE_FORWARD, MOVE_LEFT, MOVE_RIGHT, TURN_LEFT, and TURN_RIGHT. Our agent receives a continuous positive reward when it gets closer to the vest, a continuous negative reward when it gets further away from the vest, a penalty of 100 points if it is killed by the enemies, and a reward of 1000 points on reaching the vest (or atleast get past all the enemies). The game-playing session ends when our agent reaches the vest, it dies, and when there is a timeout (2,100 frames).

The dependencies of our implementation include PyTorch - for implementing a Deep Convolutional Q-Network, Gym (also, ppaquette-gym-doom) - for testing our agent, and ffmpeg - for recording the game-playing sessions.

3.1 Building the Brain

The Brain makes the neural framework of our agent's AI. The framework consists of three convolutional layers and two linear fully-connected layers. The repeated convolution operations are used to detect more detailed features. Input of the successive operation is the output of the previous operation.

The first convolutional layer has only 1 in_channel because the AI deals with black & white images. The first layer outputs 32 features by using a feature detector of size equal to 5 pixels. The second convolutional layer takes those 32 detected features and outputs 32 new features for each input. Similarly, the third convolutional layer takes the output from the previous convolution operation and outputs 64 features for each input. We keep on decreasing the size of the feature detector in subsequent layers in order to detect minute details.

Another function of the Brain is to propagate the whole network in the forward direction. After each convolution operation, max pooling is applied followed by a rectifier activation function. Finally, the output from the third convolution operation is flattened into a vector. This flattened vector is fed to the hidden layer to complete the first full connection. Number of input features for the first full connection is equal to the total number of neurons in the flattened layer. The neurons in this hidden layer are activated using the same rectifier function and fed to the output layer to complete the second full connection. This output layer contains the Q-values corresponding to the number of possible actions that the agent is capable of taking, which is 6 in this case.

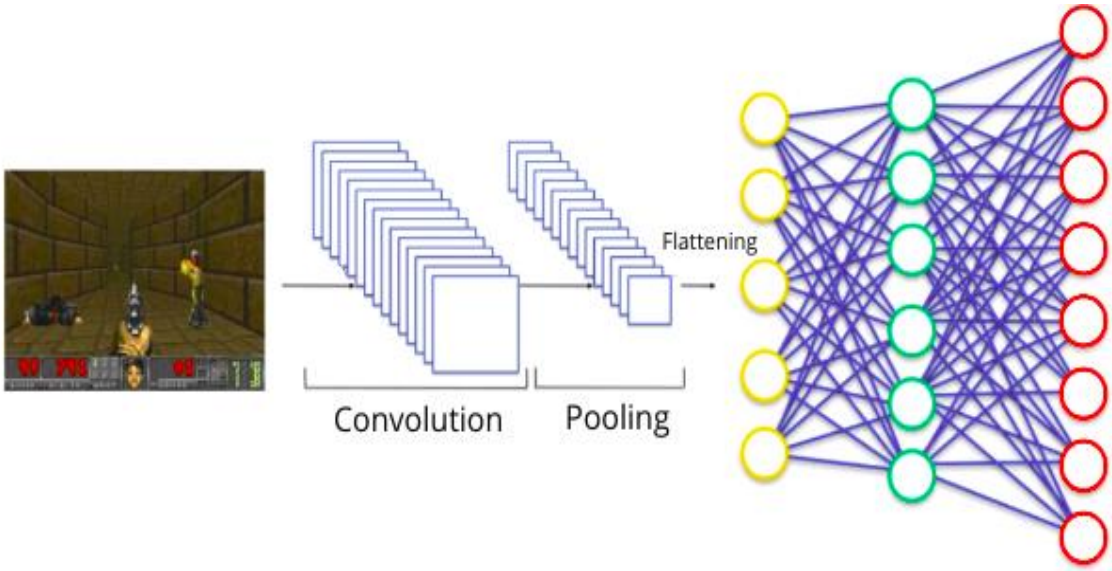

Fig 1: Neural Architecture of the Agent's AI

3.2 Building the Body

The Body is responsible for taking an appropriate action according to the received Q-values from the Brain. Its initialization includes a temperature parameter which is directly responsible for the confidence level of the agent to take an action.

The Softmax function is used to determine a probability distribution over the possible Q-values which is used to sample the best action to take in the current state.

3.3 Assembling the AI

The Brain and the Body are assembled to function as a complete AI unit that implements a convolutional neural network and also finalises the action to be played. The initialization involves an object of the Brain type and an object of the Body type. The Body also takes care of converting the input images to suitable format that is acceptable by the CNN. The input images are converted into Tensor data type of PyTorch and the actions are returned as a numpy array.

3.4 Preprocessing Images

Gym provides the functionality to read inputs from the chosen environment and alter them according to our suitability. In this scenario, we pre-process each frame by reducing the number of channels to 1 to convert it to grayscale and then reduce its size to (80 x 80).

3.5 Implementing Experience Replay and Eligibility Trace

Rather than backpropagating losses according to the rewards received from the next possible future state, the agent learns from the experience it gains from the subsequent 10 steps which are randomly sampled from its memory capacity of the last 10000 states it has been in. We iterate the agent for those next 10 steps according to the following algorithm:

I. Initialize the doom environment, AI, a list of rewards and number of steps.

II. Define *__iter__* as follows*:*

A. *State* ← Reset the environment.

B. *history* ← an empty double-ended queue

C. *reward* ← 0

D. repeat while True ←

1. *action* ← output from AI

2. *next_state*, *r*, *done*, _ ← take the next step according to *action*

3. *reward* ← *reward* + *r*

4. repeat while length(*history*) > number of steps + 1 ←

a) Pop one element from the left of *history.*

5. if length(*history*) == number of steps + 1:

a) return *history* as a generator

6. *state* ← *next_state*

7. if *done:*

a) if length(*history*) > number of steps + 1:

(1) Pop one element from the left of *history.*

b) repeat while length(*history*) > 1 ←

(1) return *history* as a generator

(2) Pop one element from the left of *history.*

c) Append reward to the initialized empty list.

d) reward ← 0

e) Reset the environment.

f) Clear the *history*

This stepwise progress can return rewards collected from those 10 subsequent steps when needed. As stated earlier, these steps are randomly sampled from the past 10000 steps, if they exist.

For implementing eligibility trace, we need a decay factor. It is used to make decision-making better by respecting the true nature of a Markov Decision Process. In this case, we have chosen it to be 0.99. We apply eligibility trace on a batch. For each series of steps in a batch, the final goal is to get the inputs and the associated targets ready to minimize the squared difference between the two for training.

3.6 Training the AI

Finally, we train the neural network to output the right predictions of the actions that the agent is supposed to take at each state. Mean Squared Error is used to calculate the loss and for optimization, we have used the Adam optimizer with a learning rate of 0.001.

We trained the AI for 100 epochs. For each epoch, 200 successive runs of 10 steps are made. From these runs, some randomly sampled batches of a fixed size are obtained. The fixed size is taken to be 128. This means that every 128 steps, our memory will give us a batch of size 128 which will contain the last 128 steps that we have just run. The learning happens on these batches. Inside these batches, we have eligibility trace running in order to learn every 10 steps.

The inputs and targets corresponding to a single batch are obtained at each batch in a single epoch. Taking those inputs, the AI returns certain predictions which are further used to calculate the overall loss for that batch. The calculated loss is backpropagated to update the weights of the network.

We also keep track of the average reward for each epoch. After many training cycles, we see that the agent surely reaches the vest and clears the level on an aggregate score of 1500. So, we stop training cycles after this score is achieved.

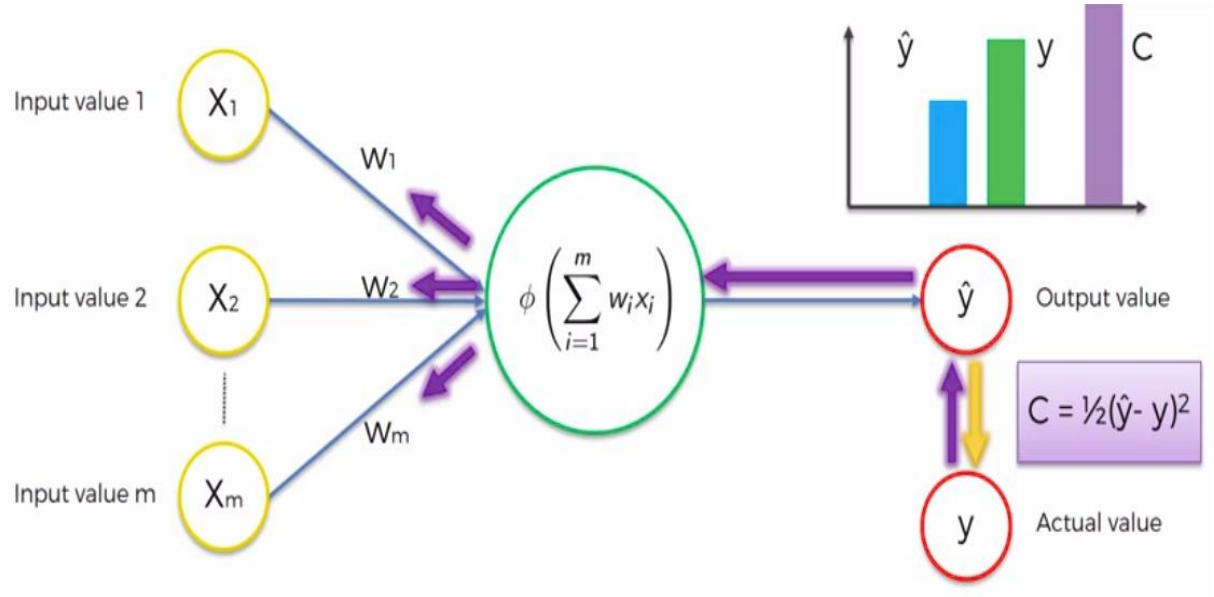

Fig 2: Simple Overview of Backpropagation

4. EXPERIMENTAL RESULTS

We achieved quite rewarding results and our agent was able to reach the desired score after only 25 epochs. This number varies significantly everytime we train the AI again because exploration is a stochastic process.

```
Terminal
(/home/manankalra/anaconda2/envs/main) manankalra@manankalra:~/folder/Python/doom_ai$ python brain.py
[2018-04-30 14:44:01,155] Making new env: ppaquette/DoomCorridor-v0
[2018-04-30 14:44:03,432] Starting new video recorder writing to /home/manankalra/folder/Python/doom_ai/videos/openaigym.video.0.6437.vid
eo000000.mp4
brain.py:82: UserWarning: Implicit dimension choice for softmax has been deprecated. Change the call to include dim=X as an argument.
  probs = F.softmax(brain_outputs*self.T)
[2018-04-30 14:44:08,638] Starting new video recorder writing to /home/manankalra/folder/Python/doom_ai/videos/openaigym.video.0.6437.vid
eo000001.mp4
Epoch: 0, Average Reward: 37.043121337890625
Epoch: 1, Average Reward: 27.14600372314453
Epoch: 2, Average Reward: 68.446291242327
[2018-04-30 14:44:40,552] Starting new video recorder writing to /home/manankalra/folder/Python/doom_ai/videos/openaigym.video.0.6437.vid
eo000008.mp4
Epoch: 3, Average Reward: 90.31376075744629
Epoch: 4, Average Reward: 90.31376075744629
Epoch: 5, Average Reward: 44.940028797496446
Epoch: 6, Average Reward: 21.389574323381698
Epoch: 7, Average Reward: 547.0705148797286
Epoch: 8, Average Reward: 916.9173858642578
```

Fig 3: Training the AI:Average Reward After Each Epoch - I

```
[2018-04-30 14:47:04,695] Starting new video recorder writing to /home/manankalra/folder/Python/doom_ai/videos/openaigym.video.0.6437.vid
eo000027.mp4
Epoch: 9, Average Reward: 842.1489552089146
Epoch: 10, Average Reward: 1009.159013227983
Epoch: 11, Average Reward: 981.844851175944
Epoch: 12, Average Reward: 1023.7386127471924
Epoch: 13, Average Reward: 999.8979559610057
Epoch: 14, Average Reward: 1063.1795592016103
Epoch: 15, Average Reward: 1152.4668429339374
Epoch: 16, Average Reward: 1107.7224263158337
Epoch: 17, Average Reward: 1176.7899073040674
[2018-04-30 14:54:06,463] Starting new video recorder writing to /home/manankalra/folder/Python/doom_ai/videos/openaigym.video.0.6437.vid
eo000064.mp4
Epoch: 18, Average Reward: 1122.226123639007
Epoch: 19, Average Reward: 1170.4565797570633
Epoch: 20, Average Reward: 1239.108026625235
Epoch: 21, Average Reward: 1312.6105475930606
Epoch: 22, Average Reward: 1347.7064546019167
Epoch: 23, Average Reward: 1396.2997868855794
Epoch: 24, Average Reward: 1463.0281442260741
Epoch: 25, Average Reward: 1569.7484532165527
CONGRATULATIONS!
Your agent has cleared "DoomCorridor-v0"!
```

Fig 4: Training the AI: Average Reward After Each Epoch - II

During these 25 epochs, multiple video sessions of our agent playing Doom under the DoomCorridor-v0 environment were recorded. The following stills from those videos depict the progress of our agent.

Fig 5: Video Stills Showing Agent's Progress

Fig 6: Agent Reaches the Vest

5. CONCLUSION

In this paper, we presented a Deep Convolutional Q-Network that learns to play a first-person shooting game - Doom, solely by exploring the environment and collecting rewards on subsequent changes that occur in its state. It attempts to learn from relevant experiences that it collects on following different sequences of steps during the exploration phase. We also implement Experience Replay and Eligibility Trace in order to make it function as close as possible like an actual human brain learns. We make use of an environment bundle from OpenAI which makes it easy for us to compare different approaches of our reinforcement learning algorithm. We find that our implementation of this gaming agent learns from random reward providing experiences directory from high-dimensional sensory input, that is, images. After a few training cycles, we achieved significant results on clearing the game by surpassing all previous approaches and reaching an all-time high score.

5.1 Future Work

As this model is based on a reinforcement learning approach, which doesn't require any training data like conventional machine learning techniques, we plan to test our agent on various other game-playing environments, such as Breakout. We also plan on modifying our neural architecture to adopt asynchronous behavior which will involve multiple agents interacting with it's own copy of the environment to learn more efficiently, and from each-other as well.

6. REFERENCES

[1] Sutton, R. S. & Bartro, A. G. (1998). Introduction to Reinforcement Learning. Cambridge, MA: MIT Press.

[2] Bellman, R.E (1957). Dynamic Programming., Princeton, New Jersey: Princeton University Press.

[3] Bellemare M.G., Naddaf, Y., Veness, J. & Bowling M. (2013). The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research, 47:253–279.

[4] Bellemare, M. G., Veness, J. & Bowling, M. (2012). Investigating contingency awareness using atari 2600 games. In AAAI.

[5] Silver, D. (2016). Deep Reinforcement Learning. DeepMind Technologies.

[6] Krizhevsky, A., Sutskever, I. & Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25, pages 1106–1114.

[7] White, D. J. (1993, November). A Survey of Applications of Markov Decision Processes. The Journal of the Operational Research Society, Vol. 44, No. 11, pp. 1073-1096.