Gaming from Pixels: Building Control Networks using Deep Reinforcement Learning

Apoorve Dave* A1DAVE@UCSD.EDU
University of California San Diego

Nivedita Prasad* N1PRASAD@UCSD.EDU
University of California San Diego

Saurabh Gupta* SAG043@UCSD.EDU
University of California San Diego

Abstract

In this project, we apply a deep learning model recently developed by (Mnih et al., 2015) to learn optimal control patterns from visual input using reinforcement learning. While this method is highly generalizable, we applied it to learn video game strategy, specifically for Atari Games of Pong, Boxing and KungFu master. Given raw pixel values from the screen, we used a convolutional neural network trained with Q learning to approximate future expected reward for all possible actions, and then selected an action based on the best possible outcome. We find that this method is capable of generalizing to new problems with no adjustment of the model architecture. After sufficient training, our model achieved better than human performance while playing Pong. As with Boxing, it learned surprising strategies like pinning down the opponent to a corner to maximize the score.

1. Introduction

For a long time, reinforcement learning algorithms have been mostly successful in solving problems with state spaces of low dimensionality. When applied to tasks that involve high dimensional inputs (e.g. pixel data), the prevalent approach has been to rely on handcrafted features that condense the relevant information into a low dimensional feature representation. The quality of such a system depends on the quality of the feature representation (whether the features allow for a good reconstruction of the input) and its utility for the original learning problem (whether the features are actually useful for learning a policy).

The most radical breakthrough of the recent years came from using a neural network as a direct function approximator, linking an arbitrarily complex state space to a Q-value function. According to the reported results, the system was able to learn how to play some simple Atari games using the described approach - i.e. by taking as inputs the pixel data, information about the action taken and achieved reward and ultimately estimating the expected value of each action for a given point of the state space.

In other words, using convolutional neural network representation as a function approximator for Q-value function enables us to embed very high-dimensional input image data into the network. The alternative is to use a Q-table to map every possible state in this high-dimensional state space with a Q-value, would have been infeasible and impractical. For e.g. consider an image of size 80×80 binary pixels, the state space will be $2^{80 \times 80}$. Hence, neural network seems to be the obvious choice for this case. Once trained, it learns to predict the action which leads to maximum reward from the given state.

1.1. Problem Description

To train a neural network for reinforcement learning task, we chose to implement Deep Q-Learning Network (Mnih et al., 2015) on traditional Atari Games, specifically Pong, Boxing and KungFu Master (Figure 1). Our task was to train the network to choose an action (out of 18 possible actions) based on current state of the game such that it maximizes the long term discounted reward R_T(equation 1).

1.2. Dataset

Our dataset consists of game states each game is currently in. Each state is defined completely by the image provided

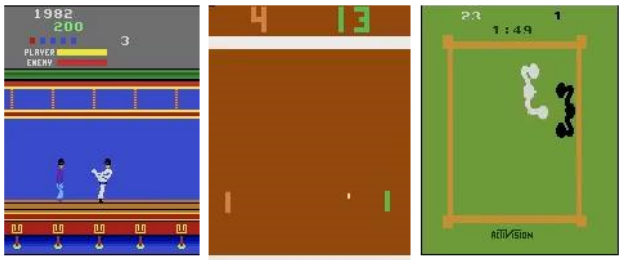

Figure 1. Atari Games: KungFu, Pong and Boxing used in our training

by game Interface provider. We used OpenAI Gym for simulating Atari Games. As such, our data set was generated with time as the game agent played a number of games and explored the state space. Since this dataset was ever increasing, we used a buffer of the last 1000000 game states for training. The buffer keeps on updating with latest states and discards the earliest ones as the game progresses.

2. Background

Reinforcement learning develops control patterns by providing feedback on a models selected actions, which encourages the model to select better actions in the future. At each time step, given some state s, the model will select an action a, and then observe the new state s' and a reward r based on some optimality criterion.

2.1. Q-Learning

As mentioned earlier, Reinforcement learning explores a problem, where an agent tries to maximize a cumulative reward in an a priori unknown environment by relying on interactions with the environment. The environment can be modeled as a Markov Decision Process (MDP), i.e. a set of states S, a set of actions A, a transition function T: $S\times A\times S\rightarrow[0,1]$, which defines a probability distribution over possible next states given the current state and an action, and a reward function $S\times A\times S\rightarrow\mathbb{R}$. The goal of the agent is to find the policy π: $S\rightarrow A$ that maximizes the discounted cumulative reward:

$$R_T=\sum_{t=T}^{\infty}\gamma^t r_t \tag{1}$$

where γ is a discounted factor between 0 and 1, r_t denotes the reward obtained at time-step t and T is the current time-step. We then define the action-value function:

$$Q^*(s,a)=\mathbb{E}[R_t|s_t=s,a_t=a] \tag{2}$$

Note that if the optimal Q function is known for state s', we can write the optimal Q function at preceding state s as the maximum expected value of $r+\gamma Q^*(s',a')$. This identity is known as the *Bellman equation*:

$$Q^*(s,a)=\mathbb{E}[r+\gamma*max_{a'}Q^*(s',a')|s,a] \tag{3}$$

The intuition behind reinforcement learning is to continually update the action-value function based on observations using the Bellman equation. It has been shown by Sutton & Barto that such update algorithms will converge on the optimal action-value function as time approaches infinity. Based on this, we can define Q as the output of a neural network, which has weights θ, and train this network by minimizing the following loss function at each iteration i:

$$L_i(\theta_i)=\mathbb{E}[(y_i-Q(s,a;\theta_i))^2] \tag{4}$$

Where y_i represents the target function we want to approach during each iteration. It is defined as:

$$y_i=\mathbb{E}[r+\gamma*max_{a'}Q(s',a';\theta_{i-a})|s,a] \tag{5}$$

Note that when i is equal to the final iteration of an episode (colloquially the end of a game), the Q function should be 0 since it is impossible to attain additional reward after the game has ended. Therefore, when i equals the terminal frame of an episode, we can simply write:

$$y_i=\mathbb{E}[r|s,a] \tag{6}$$

2.2. Deep-Q learning

Consider our game environment where each game state is represented by an image. If we also want to capture the motion of objects in the image, we can choose to take previous n images and call it the current state(plus the recent past information).

If we apply the same pre-processing to game screens as done by Mnih et al., we would take the four last screen images, resize them to 84×84 and convert to grayscale with 256 gray levels, then we would have $256^{84\times84\times4}\approx10^{67970}$ possible unique game states.

This is the point where deep learning steps in. Neural networks are exceptionally good at coming up with good features for highly structured data. We could represent our Q-function with a neural network, that takes the state (four game screens) and action as input and outputs the corresponding Q-value. Alternatively we could take only game screens as input and output the Q-value for each possible action. This approach has the advantage that if we want to perform a Q-value update or pick the action with the highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions available immediately. This is the approach followed by Mnih et al..

Input to the network are four 84×84 grayscale game screens. Outputs of the network are Q-values for each possible action (18 in Atari). Q-values can be any real values, which makes it a regression task, that can be optimized with simple squared error loss:

$$L = \frac{1}{2}[r + max_{a'}Q(s', a') - Q(s, a)]^2 \tag{7}$$

Given a transition $< s, a, r, s' >$ (i.e. action a taken in state s, leading to reward r and new state s'), the Q-table update rule will be the following:

1. Do a feedforward pass for the current state s to get predicted Q-values for all actions.
2. Do a feedforward pass for the next state s' and calculate maximum overall network outputs $max_{a'}Q(s', a')$.
3. Set Q-value target for action a to $r+\gamma.max_{a'}Q(s', a')$ (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. Update the weights using backpropagation.

2.3. Experience Replay

During the training, there is a lot of similarity between subsequent game states. This makes learning slow as the input states are highly correlated. This may also lead to network getting stuck in a local minima. Experience replay (Lin, 1993) is one of the most important tricks used in Deep Q Learning to counter this issue.

During gameplay all the experiences $< s, a, r, s' >$ are stored in a replay memory. When training the network, random mini-batches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples. Also, experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm. One could actually collect all those experiences from human gameplay and then train network on these.

2.4. Exploration vs Exploitation

Observe that when a Q-network is initialized randomly, then its predictions are initially random as well. If we pick an action with the highest Q-value, the action will be random and the agent performs crude "exploration". As a Q-function converges, it returns more consistent Q-values and the amount of exploration decreases. So one could say, that Q-learning incorporates the exploration as part of the algorithm. But this exploration is "greedy", it settles with the first effective strategy it finds.

A simple and effective fix for the above problem is ϵ-*greedy exploration* – with probability ϵ choose a random action, otherwise go with the "greedy" action with the highest Q-value. Mnih et al. actually decreases ϵ over time from 1 to 0.1 - in the beginning the system makes completely random moves to explore the state space maximally, and then it settles down to a fixed exploration rate.

2.5. Algorithm

Algorithm 1 gives us the final deep Q-learning algorithm with experience replay.

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory D to capacity N
initialize action-value function Q with random weights
observe initial state s
repeat
    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 D
    calculate target for each minibatch transition
    if ss' is a terminal state then
        tt = rr
    else
        tt = rr + γmax_a' Q(ss, aa)
    end if
    train the Q network using (tt − Q(ss, aa))^2 as loss
    Set s = s'
until terminated
```

2.6. OpenAI Gym

We used OpenAI Gym (Brockman et al., 2016) for simulating Atari environment. It is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games

like Pong or Boxing.

There are two basic concepts in reinforcement learning: the environment (namely, the outside world) and the agent (namely, the algorithm you are writing). The agent sends actions to the environment, and the environment replies with observations and rewards (that is, a score). OpenAI gym provides an interface for the environment to train on.

3. Experiments

3.1. Q-learning using simple feed forward network

Our first experiment was based on our home-grown simple feed forward network instead of CNN's Deep Mind network. We trained this network on ATARI game - Pong. On the low level the workflow is as follows: we receive an image frame (a 210×160×3 byte array (integers from 0 to 255 giving pixel values)) from the simulator and we get to decide if we want to move the paddle Up or Down (i.e. a binary choice). After every single choice the game simulator executes the action and gives us a reward: Either a +1 reward if the ball went past the opponent, a -1 reward if we missed the ball, or 0 otherwise. And of course, our goal is to control the paddle so that we get lots of reward.

3.1.1. Preprocessing

A single image frame cannot capture the entire state of the game. So, we need at least two consecutive frames for determining the state. We converted the output frames from the simulator (210×160×3) from RGB representation to gray-scale and downsampled it to 80×80 images. Finally, we found the difference frame by subtracting the current frame from the last frame and fed it to the network.

3.1.2. Model Architecture

We used a 2 layer Feed-forward neural network which takes in the pre-processed image and outputs the Q-value of two actions (going Up and Down). The network architecture is shown in Figure 2. The hidden layer has 200 units with ReLU activation function. The output layer is a fully connected linear layer with a single output for each valid actions. The hyperparameters can be found in Table 1.

3.1.3. Training

We used the RMSProp algorithm with minibatches of 10 episodes (each episode is a few dozen games, because the score goes upto 21 for either player) to train our network. The behavior policy during training was ϵ-greedy with ϵ starting from 1 and gradually decreasing till it reaches 0.1, and fixed at 0.1 thereafter. We used a replay memory of one million most recent frames with mini batch size of 10 randomly chosen samples. Because of the simplicity of the network and limitations of CPU, the training took more than 5 days of training in CPU mode.

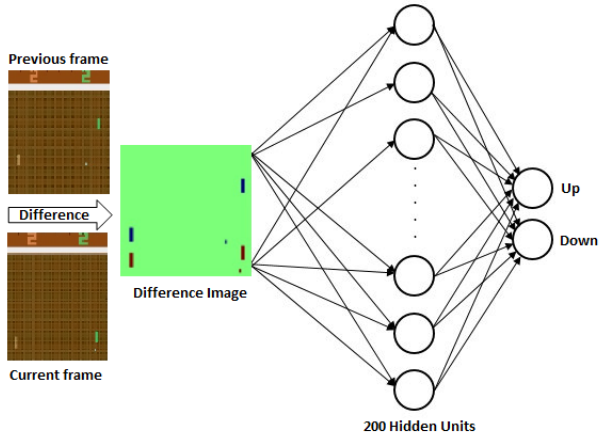

Figure 2. Simple architecture for training Pong. It contains 200 hidden neurons and outputs Q(s,a) value for actions UP and DOWN. A difference image of current and previous frames is fed as input to the network to capture motion information.

Table 1. Hyperparameters for MLP Q-Learner

Number of Outputs	2
Hidden Layer Neurons	200
Batch Size	10
Learning Rate	1e-4
γ(discount factor for reward)	0.99
optimizer	RMSProp
RMSProp Decay Rate	0.99
initial ϵ value	1.0
final ϵ value	0.1

3.1.4. Results

The training process is still going on and showing a slow progress. From the random play score of -20.7, we managed to achieve an average score of 3.8 and a max score of 13. It still performs poorer than a human player, whose average score is 9.3, but has learned to outperform the atari hardcoded agent most of the time by a small margin.

3.2. Q-learning using Convolutional Neural Network

Our next experiment was to train a Deep Convolutional Neural network or DQN for Q-Learning. We performed training on ATARI games - Pong, Boxing and Kungfu. We used the same network architecture, learning algorithm and hyperparameters for training our agent on the three differ-

ent games. To implement the model, we chose to work with Keras (Chollet, 2015).

3.2.1. PREPROCESSING

Since working directly with raw Atari frames, which are 210 $\times$ 160 pixel images with a 128 color palette, is computationally demanding, we apply a preprocessing step aimed at reducing the input dimensionality. The raw frames are preprocessed by first converting their RGB representation to gray-scale and down-sampling it to a 110 $\times$ 84 image. The final input representation is obtained by cropping an 84 $\times$ 84 region of the image that roughly captures the playing area. The final cropping stage is only required because we use the GPU implementation of 2D convolutions, which expects square inputs. We preprocessed the last 4 frames of a history using the above steps and stacked them to produce the input to the Q-function, to preserve motion information in the game.

3.2.2. MODEL ARCHITECTURE

For this experiment, we used the DQN model developed by (Mnih et al., 2015). The input to the neural network is a 84 $\times$ 84 $\times$ 4 image. The first hidden layer convolves 32 8 $\times$ 8 filters with stride 4 with the input image and applies ReLU. The second hidden layer convolves 64 4 $\times$ 4 filters with stride 2, again followed by ReLU. This is followed by a third convolutional layer that convolves 64 filters of 3 $\times$ 3 with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action(18 in total). The network model is shown in Figure 3. The network architecture and hyper parameters can be found in Table 3 and Table 4.

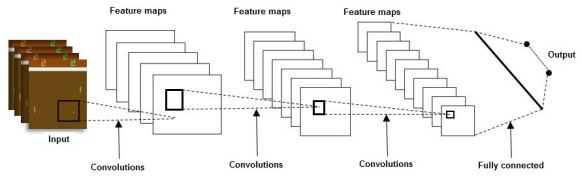

Figure 3. DeepMind DQN architecture.

3.2.3. TRAINING

Similar to the training procedure followed by (Mnih et al., 2015), we trained our model using RMSProp algorithm with mini batch size was 32. Following the ϵ-greedy policy, we annealed the value of ϵ linearly from 1 to 0.1 over the first million frames and fixed it at 0.1. We used a replay memory of one million most recent frames for training throughout the training process. This network being far more complex than our previous model, learned to score much higher scores in just 2 days of training on GPU (80 epochs). The network hyperparameters can be found in Table 4.

3.2.4. RESULTS

We were able to achieve super-human performance for our trained agent for Boxing. Our model was able to achieve an average score of 41, which is around 9.5 times more than average human score of 4.3. After about just 50 epochs, the agent learned to pin the opponent to a corner to maximize the score. Being trapped, the opponent was not able to fight back which made our agent to win games by huge margins. In Figure 4, we can see the comparison of average reward with random play, and its growth with number of iterations. In Figure 5, we have shown a snapshot of a sample game played at different epochs through the training, alongwith the final scores achieved.

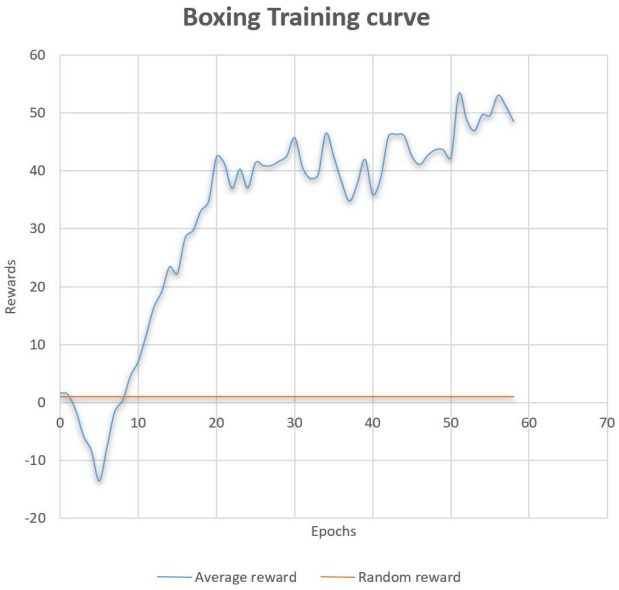

Figure 4. Average Reward vs number of iterations for Boxing. A comparison with random action reward is shown in the figure.

Similarly, with Pong DQN model, we were able to achieve an average of 11.8 which is more than average human score of 9.3. Compared to MLP model of Pong(discussed in previous section), it performed twice as good and learned the model much faster. Since Pong-MLP model is much simpler, it took more time to learn and did not achieve as good results as Pong-DQN model. Figure 6 for average reward growth with iterations.

Table 2. Final Averaged Gameplay Scores

Game	Random Play*	Human*	DQN	Multi-layer Perceptron	Normalized (% Human)
Boxing	0.1	4.3	41.2 ± 11.4		958.14 %
Pong	-20.7	9.3	11.8 ± 2.1	3.8 ± 1.4 (40.86 %)	126.8 %
Kung-Fu Master	258.5	22736	14129 ± 3234		62.14 %

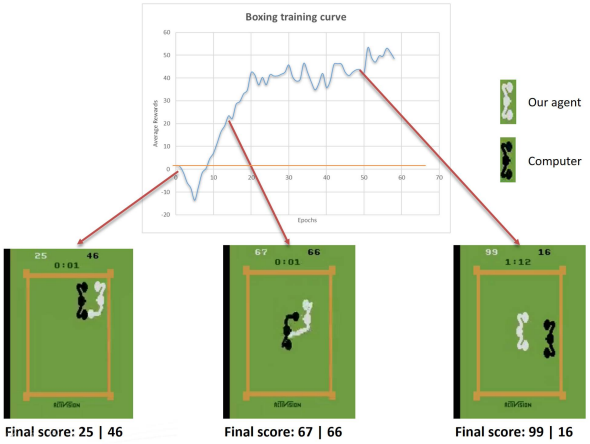

Figure 5. Final scores of three game plays corresponding to epochs 1, 15 and 70. Note that the DQN agent(white) increased its score from 25-46 to 99-16.

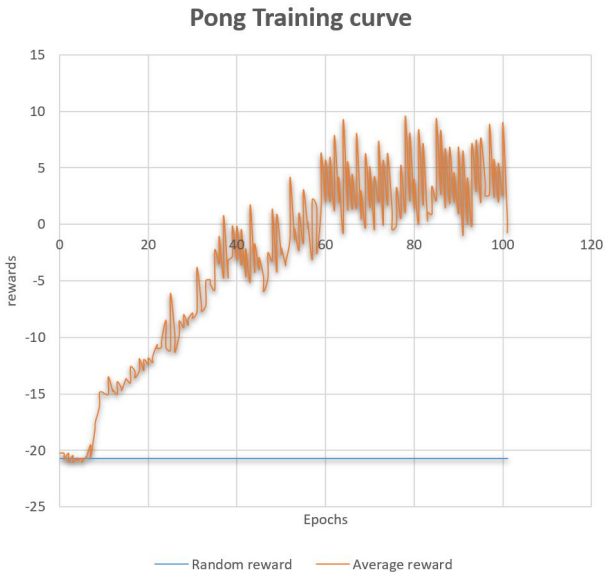

Figure 6. Average Reward vs number of iterations for Pong-DQN. A comparison with random action reward is shown in the figure.

With KungFu Master, our agent learned to play a little and consistently score(average score 14129) more than random agent (score 258.5) but was not able to beat average human score (22736). Our assumption is that since KungFu Master requires a bit more of intelligence (e.g. to keep moving forward for next screen to proceed in the game instead of just continuing hitting thugs in current screen), our agent was not able to learn what comes naturally to humans. Still, it leaned to hit combo punches and kicks when multiple thugs came in a row. Figure 7 for average reward growth with iterations.

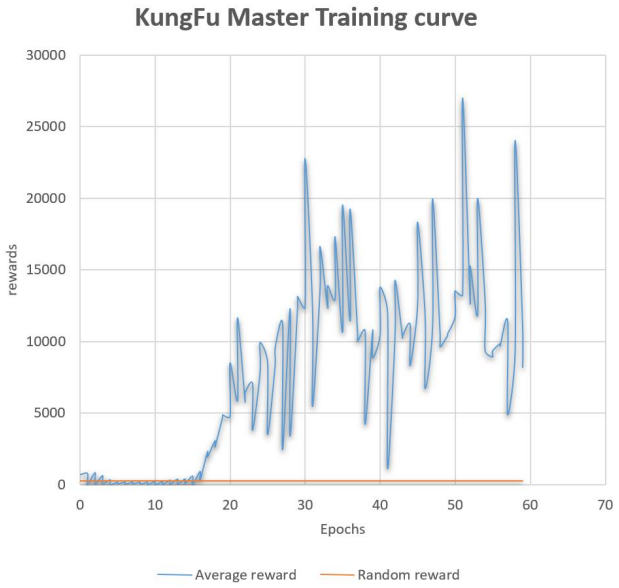

Figure 7. Average Reward vs number of iterations for KungFu. A comparison with random action reward is shown in the figure. This model performs better than random agent but still shows too much variation in average rewards, probably because KungFu requires a higher level of intelligence or more time to train.

Our final results are shown in Table 2.

Conclusion

In this project, we implemented our own 2-layered Feed-Forward neural network for training Pong. We were able to achieve an average score matching that of the Atari's hard-coded computer player. We also implemented the DQN network mentioned in Mnih et al. using Keras and were able to achieve super-human performance for the games of Pong and Boxing. For Kung-Fu Master, the same network learned to score some points by hitting the enemies, but

it was not able to make progress in the game by moving forward on the map. We think this can be due to slightly higher level of intelligence the game is expecting from the agent, which comes intuitively to humans, but is difficult for the agent to learn.

References

Brockman, Greg, Cheung, Vicki, Pettersson, Ludwig, Schneider, Jonas, Schulman, John, Tang, Jie, and Zaremba, Wojciech. Openai gym, 2016.

Chollet, Franois. keras. https://github.com/fchollet/keras, 2015.

Lin, Long-Ji. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A., Veness, Joel, Bellemare, Marc G., Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K., Ostrovski, Georg, Petersen, Stig, Beattie, Charles, Sadik, Amir, Antonoglou, Ioannis, King, Helen, Kumaran, Dharshan, Wierstra, Daan, Legg, Shane, , and Hassabis, Demis. Human-level control through deep reinforcement learning. *Nature*, pp. 529–533, 2015.

Sutton, Richard and Barto, Andrew. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

APPENDIX

Deep-Q Network Architecture

Table 3. Deep-Q Network Architecture

Layer	Input	Filter Size	Stride	Num Filters	Activation	Output
conv1	$84 \times 84 \times 4$	8×8	4	32	ReLU	$20 \times 20 \times 32$
conv2	$20 \times 20 \times 32$	4×4	2	64	ReLU	$9 \times 9 \times 64$
conv3	$9 \times 9 \times 64$	3×3	1	64	ReLU	$7 \times 7 \times 64$
fc4	$7 \times 7 \times 64$			512	ReLU	512
fc5	512			18	Linear	18

Deep-Q Network Hyperparameters

Table 4. Hyperparameters for Deep Q-Network

Hyperparameters	Value	Description
minibatch Size	32	Number of training cases over which each stocastic gradient descent update is computed
replay memory size	1000000	SGD updates are sampled from this number of most recent frames
target network update frequency	10000	The frequency with which the target network is updated
discount factor	0.99	Discount factor gamma used in Q-learning updates
action repeat	4	Repeat each action selected by the agent this many number of times
update frequency	4	The number of actions selected by the agent in between successive SGD updates
learning rate	0.00025	Learning rate used by RMSProp
gradient momentum	0.95	Gradient momentum used by RMSProp
squared gradient momentum	0.95	Squared gradient momentum used by RMSProp
min squared gradient	0.01	Constant added to squared gradient in denominator of RMSProp update
initial exploration	1	Initial ϵ value in ϵ-greedy exploration
final exploration	0.1	Final ϵ value in ϵ-greedy exploration
final exploration frames	1000000	The number of frames over which the ϵ value is annealed to its final value
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory
no-op max	30	Maximum number of 'do nothing' actions to be performed by the agent at the start of an episode