Playing video games with deep networks

Sandeep Konam
The Robotics Institute
Carnegie Mellon University
skonam@andrew.cmu.edu

Utkarsh Sinha
The Robotics Institute
Carnegie Mellon University
usinha@andrew.cmu.edu

Abstract

Combination of modern Reinforcement Learning and Deep Learning approaches holds the promise of making significant progress on challenging applications. In this report, we present results of our implementation of Deep Mind's Deep Q Network, a breakthrough in combining model-free reinforcement learning with deep learning. It is the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future-rewards.

1. Introduction

Deep learning has dramatically advanced the state of the art in vision, speech and many other areas. It is made possible through the large amounts of data and computational power available. Deep learning methods employ several architectures ranging from convolutional neural networks to recurrent neural networks to restricted boltzmann machines. However most of the success has been in the realms of supervised and unsupervised learning, but not in reinforcement learning.

Reinforcement Learning lies in between supervised and unsupervised learning. In supervised learning, every training example has a target label and in unsupervised case, labels do not exist. In reinforcement learning, there are sparse and time-delayed labels, also referred to as rewards. Based on these delayed labels or rewards, agent should learn the task by choosing appropriate action.

Application of deep learning methods to reinforcement learning poses several challenges. Primary challenge as mentioned above is sparse and time delayed label makes it a hard problem unlike supervised learning. The second challenge is that data in reinforcement learning is often highly correlated - breaking the IID assumption. Most deep learning techniques assume data to be independent. The third challenge is that data distribution changes as the network learns new behaviours in reinforcement learning whereas most deep learning methods assume a fixed distribution.

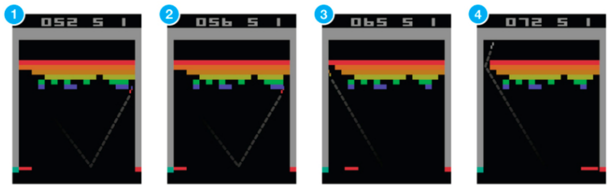

Figure 1. Atari Breakout game. Image credit: DeepMind.

DeepMind's work [5] addressed the above mentioned problems and presented the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. Based only on visual information, the system learns which action to choose as to maximize the score based on the present state inferred through the game screen. It is similar to how a human learns the games which makes it an interesting approach. Our current project is an implementation of DeepMind's approach[5].

2. Background

2.1. Reinforcement learning

Reinforcement learning can be represented as a Markov decision process. It can be explained with an example of breakout game. Consider an agent being situated in an environment which in current case is the Breakout game. Location of the paddle, existencce of the bricks and location of the ball constitute the state of the environment. Agent can perform certain action such as moving the paddle to the left or the right or stay in the same position, resulting in a change in state such as clearing more bricks and change in position of the ball and position of the paddle. It could sometimes result in a reward which is the increase in score in our case. Agent can perform another action in the new state. Rules for which actions should be chosen constitute

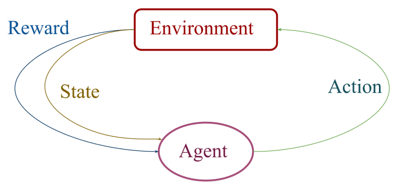

Figure 2. Reinforcement learning problem

policy. Environment in which the agent operates can be considered stochastic.

The set of states and actions, together with rules for transitioning from one state to another, make up a Markov decision process. One episode of this process captured from one game forms a finite sequence of states, actions and rewards:

s_0,a_0,r_1,s_1,a_1,r_2,..............,s_{n-1},a_{n-1},r_{n-1},s_n

Here s_i represents the state, a_i is the action and a_{i+1} is the reward after performing the action. The episode ends with terminal state s_n which can be referred to as "end of game" or "game over" screen. A Markov decision process relies on the Markov assumption, that the probability of the next state s_{i+1} depends only on current state s_i and action a_i, but not on preceding states or actions

3. System overview

In this section, we provide an overview about the task and how it is formulated in terms of a deep neural network.

3.1. Task

Goal is to train the network so that based on a picture of the game screen as shown in Figure 1, system should be able to choose an action. On executing this action, there is a change in state and it eventually receives the reward. The network should choose an optimal action that maximizes this reward. Based on this information and through playing the game numerous times, the system learns the rules of the game and play it. There is no prior information about the rules or objects of interested is provided to the network.

3.2. Approach

The input to the network at any time point consists of the four consecutive game screens, so that system is given not only the last position, but also a bit of information about how the situation in the game has evolved recently. This input is then passed through three successive hidden layers of neurons and finally to the output layer of the deep network. The output layer has one node for each possible action and the activation of those nodes indicates the expected reward from each of them. The action with the highest activation is picked as the best action and is executed.

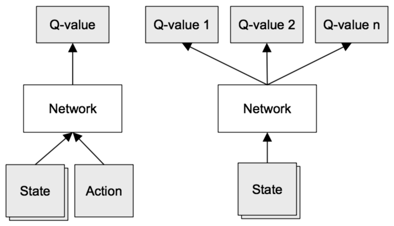

Figure 3. Formulations of Deep-Q-network; *Left*: naive formulation, *Right*: optimized formulation used for simultaenous computation of all possible button combinations.

4. System components

4.1. Deep Q Network

We can represent the Q-function with a neural network, that takes the state (four game screens) and action as input and outputs the corresponding Q-value (Figure 3, left). Alternatively we could take only game screens as input and output the Q-value for each possible action (Figure 3, right). 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 approach was introduced in DeepMinds work [5].

The network used is a classical convolutional neural network [3] with three convolutional layers, followed by two fully connected layers (Table 1). Unlike object recognition networks, the network doesnt have any pooling layers. Pooling layers are used to buy translation invariance the network becomes insensitive to the location of an object in the image. That makes perfectly sense for a classification task like ImageNet, but for games the location of the ball is crucial in determining the potential reward and this information shouldnt be discarded.

From Table 1), it can be noticed that inputs to the network are four 84×84 grayscale game screens. Outputs of the network are Q-values for each possible action (18 in case of Atari). Q-values can be any real values, which makes it a regression task, that can be optimized with simple squared error loss.

Given a transition $\langle s, a, r, s' \rangle$, the Q-values are updated in the following way:

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 to $r + \gamma \; max_{a'} Q(s', a')$

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Table 1. ConvNet architecture used for learning the Q-function

(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.

4.2. Experience Replay

As mentioned in Section 4.1, future reward can be estimated using Deep-Q-network and Q-function could be approximated using a convolutional neural network. But it turns out that approximation of Q-values using non-linear functions is not very stable. [5] introduces tricks for the network to converge, out of which "Experience Replay" is the most important one. During gameplay all the experiences $\langle s, a, r, s' \rangle$are stored in a replay memory. When training the network, random minibatches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. Also experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm.

4.3. Exploration-Exploitation

Q-learning attempts to solve the credit assignment problem it propagates rewards back in time, until it reaches the crucial decision point which was the actual cause for the obtained reward. When a Q-table or Q-network is initialized randomly, then its predictions are initially random as well. If an action with the highest Q-value is picked 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 choose a random action, otherwise go with the greedy action with the highest Q-value. In their system DeepMind 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.

5. Implementation details

Prototype of the system is built using the Torch framework[2]. It was trained on a system with an Intel i7 processor and the nVidia 980 Ti graphics processor.

5.1. Arcade Learning Environment

To simulate the game we use Atari Learning Environment (ALE)[1]. It is a game simulator which allows to programmatically send player commands and receive the game output (image of the game screen, score, state of the game). The team at DeepMind has open-sourced the Lua wrapper for the framework. The framework provides simple commands to load specific ROMs, which are freely available on the internet.

During out experiments, we found that the framework is unable to support any arbitrary Atari 2600 game. This is due to the fact that the framework needs to have a perfect one-to-one correspondence between rewards and in-game actions. We worked around this by using only games that were supported by the framework.

5.2. Preprocessing

The Arcade Learning Environment provides us with a continuous stream of screenshots from games. Feeding these images directly into the neural network is possible but isn't recommended. Instead, we apply some basic preprocessing to help the neural network learn the Q function.

The screenshots are 160×120 is size and contain three channels. For each game, we manually created a crop window for the playing field. This ensures metadata like game score and number of lives do not affect training. We then scale these cropped images into 84×84 windows. This window is then fed into the network. We use the same crop window and scaling technique for both training and testing.

5.3. Training Details

We trained the neural network from scratch on three different games - Defender, Space Invaders and Breakout. The network initialized using gaussian weights and trained over a period of one week each. We used a discount factor $\gamma = 0.99$ to account for future rewards. To run gradient

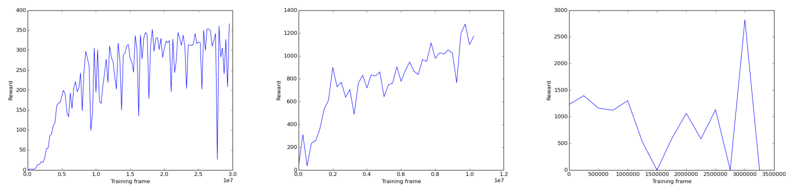

Figure 4. Training rewards vs time for the three games we used: Breakout, Space Invaders and Defender

descent on, we used a replay memory size of 1 million recent states. This amounted to about 8GB of RAM during training since our input image size is fixed for all games.

The network starts by randomly picking a move out of the 18 possible moves at the beginning. The frequency of these random exploratory moves is linearly annealed until 50000 training steps. After this, there is a 10% chance of picking a random move vs using the network to predict the next move. The learning rate for this network was set to 0.00025. We used a batch size of 32 states to compute the gradient and used the standard stochastic gradient descent package in Torch.

6. Results

During training, we monitored the rewards accumulated over time in each episode of the game. We found that the training worked as expected for the games of Breakout[7] and Space Invaders[9]. The positive rewards over time increase over time and plateau after a certain point in training.

However, for Defender[8], we found that the network never converged. This game was not mentioned in the paper and we wanted to test if the network can learn this game or not. Our hypothesis is that the side-scrolling nature of the game caused drastic changes in viewpoint and relatively small sprites in the image contributed to this. In a more recent paper[6] by the same group, the game was learned and played successfully a neural network.

Unlike the original paper, our implementation can only reach very close to winning the game but not actually win the game. We are uncertain why this might be the case. Our hypothesis is that the end game requires very specific moves which the network has not had a chance to learn well. The end of the game happens only at specific time intervals in the replay memory and training specifically on those memories might improve the endgame.

7. Conclusion and Future work

In the current project, we were able to replicate DeepMinds work [5] and achieve promising results. We've found this project to be a good first step towards reinforcement learning in general and hope to work further in this field.

We found that the training with reinforcement learning takes longer than we had originally anticipated. There have been recent publications that help improve the training time. Specifically, asynchronous deep reinforcement learning seems promising[4]. Another recently published paper[6] helps prioritize certain key memories for gradient descent. This would significantly speed up training time and also possibly produce better results.

We wish to further continue working on it by incorporating better memory efficiency in the implementation. Deep reinforcement learning promises general artificial intelligence that can adapt and learn in any environment. We are also interested to see how deep reinforcement learning techniques can be applied to Unmanned Aerial Vehicles for indoor navigation which are 3D environments.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2012.

[2] R. Colloboert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. *NIPS*, 2011.

[3] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time series. 1995.

[4] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop*, 2013.

[6] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *ICLR*, 2016.

[7] U. Sinha and S. Konam. Deep reinforcement playing breakout, https://www.youtube.com/watch?v=yshvrkmrszw. 2016.

[8] U. Sinha and S. Konam. Deep reinforcement playing defender, https://www.youtube.com/watch?v=v_4NXo9SURc. 2016.

[9] U. Sinha and S. Konam. Deep reinforcement playing space invaders, https://www.youtube.com/watch?v=GW_O3miLY8A. 2016.