ENPM 667: Control of Robotic Systems

Playing Atari with Deep Reinforcement Learning

Nov 24, 2021

Students:

Paras Savnani

Instructor:
Dr. Wasim Malik

Semester:
Fall 2021

Course code: ENPM667

Contents

1. Abstract

This paper tries to learn the control policies from high dimensional sensory input space (image-3D) using reinforcement learning. It combines the state-of-the-art Convolutional Neural Network of 2013 with Q learning [4] with raw pixels as inputs and estimated future states/actions as outputs. This method is validated on the Atari environment in OpenAI [1] Gym with PyTorch [2] as the deep learning framework. Also, it was a groundbreaking work by Deep Mind in the field of RL because it outperformed human experts and gave us new insights on how AI agents can discover novel ways of playing these games.

2. Introduction

Traditional control policies like PID, MPC etc. have been around for a while, but they are system specific and do not generalize well with multiple different systems. Till this paper most policies relied on hand designed features with some add-ons. But the bottleneck with these policies was that their performance relied heavily on the feature representation.

Year 2012 saw one of the biggest breakthroughs in Deep Learning with the Alexnet [3] paper, which shook the DL community with off the chart's performance on the image classification task. This architecture presented a hierarchical structure for feature extraction in which the initial layers learned the low-level features like edges and corners and the final layers learned high level features like small image patches, finally all this was fed into fully connected layers and the output was a classification index. Furthermore, leveraging the power of deep learning for RL seemed natural after this.

Now, the classification problem is posed as a supervised learning problem, and it requires a large amount of labelled training data. But RL algorithms learn based on a scalar reward function which is sparse and noisy. Also, the images seldom have correlation between them in classification and in RL one encounters high correlated states. Also, the data distribution changes in RL as the agent becomes more intelligent, but deep learning-based models assume a fixed data distribution.

This paper shows how to combine traditional Q-learning[4] with CNN to learn complex control policies from raw video feed. Also, it uses an experience replay mechanism to randomly sample previous data and smoothen out the training process.

This algorithm can be applied to a multitude of problems, but we use the Atari 2600 Breakout-v1 environment from OpenAI Gym [2016] [1] which has 4 set of actions (None, Fire, Left, Right) to test and validate the performance of the algorithm and it outperforms the human expert.

3. Background

The problem setup revolves around 4 main components i.e., Environment, State/Reward, Action, Agent. The agent interacts with the environment *epsilon, Atari emulator* for this use case. Agent has to select an action from available actions A = {1, …, K}. This action modifies the env. and the game score. And we do not provide the internal state information to the agent. Also, in addition it receives a reward that represents the change in game score. Furthermore, it is impossible to understand the game situation based on the current screen and we consider sequences of actions and observations and make the agent learn those sequences. This line of thought gives rise to the **Markov Decision Process (MDP)** [5] in which each sequence is a distinct state.

Markov Decision Process: A stochastic process has a Markov Property if the conditional Probability distribution of future states of the process (conditional on both past and present states) depends only upon the present state and not on the sequence of events that preceded it. Also, MDP [5] provides a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the decision maker's control.

The goal in a Markov decision process is to find a good "policy" for the decision maker: a function that specifies the action (s) that the decision maker will choose in state s.

$$E * \sum_{t=0}^{\infty} \gamma^t R_{at}(s_t, s_{t+1}) \tag{1}$$

The main aim of the agent is to maximize future rewards by selecting favorable actions. These rewards are discounted by a factor of at each time step. The future discounted return is:

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2}$$

The optimal action-value function obeys an important identity known as the Bellman equation [4]. This is based on the following intuition: if the optimal value $Q^*(s', a')$ of the sequence s' at the next time-step was known for all possible actions a', then the optimal strategy is to select the action a' maximising the expected value of $r + \gamma Q^*(s', a')$.

Proof for Bellman Equation:

Equation (2) is defined with a constant discount factor 0≤γ≤1 and we can have T=∝ or $\Upsilon = 1$, but not both. Since the rewards Rk are random variables, so is Gt as it a mere linear combination of variables.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} \mid S_t = s] + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_t = s] \end{aligned} \tag{3}$$

Last line follow shows the linearity of expectation values. Rt+1 is the reward the agent gains after taking action at time step t. For simplicity let's assume it can take finite values in Real space.

We need to compute the expectation values of Rt+, given that we know the current state s. The formula is:

$$\mathbb{E}_\pi[R_{t+1} \mid S_t = s] = \sum_{r \in \mathcal{R}} r p(r \mid s) \tag{4}$$

The probability of appearance of reward r is conditioned on state s, i.e., different states may have different rewards. This p(r|s) distribution is a marginal distribution of a distribution that also contained the variables aa and s', the action taken at time t and the state at time t+1 after the action, respectively:

$$p(r \mid s) = \sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} p(s', a, r \mid s) = \sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} \pi(a \mid s) p(s', r \mid a, s) \tag{5}$$

Here we have used π(a|s) ≐p(a|s), also we suppress the s (the probability now looks like a joint probability), use the law of multiplication, and finally reintroduce the condition on ss in *all* the new terms. It is now easy to see that the first term is:

$$\mathbb{E}_\pi[R_{t+1} \mid S_t = s] = \sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} r \pi(a \mid s) p(s', r \mid a, s) \tag{6}$$

The second term, where I assume that Gt+1 is a random variable that takes on a finite number of values g∈Γ. Just like the first term:

$$\mathbb{E}_\pi[G_{t+1} \mid S_t = s] = \sum_{g\in\Gamma} gp(g \mid s) \tag{7}$$

Let's un-marginalize the probability distribution by writing (law of multiplication again):

$$\begin{aligned} p(g \mid s) &= \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} p(s', r, a, g \mid s) = \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} p(g \mid s', r, a, s)p(s', r, a \mid s) \\ &= \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} p(g \mid s', r, a, s)p(s', r \mid a, s)\pi(a \mid s) \\ &= \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} p(g \mid s', r, a, s)p(s', r \mid a, s)\pi(a \mid s) \\ &= \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} p(g \mid s')p(s', r \mid a, s)\pi(a \mid s) \end{aligned} \tag{8}$$

The last line in there follows from the Markovian property. Remember that Gt+1 is the sum of all the future (discounted) rewards that the agent receives *after* state s'. The Markovian property is that the process is memory-less with regards to previous states, actions, and rewards. Future actions (and the rewards they reap) depend only on the state in which the action is taken, so *p(g|s',r,a,s)=p(g|s')*, by assumption. Ok, so the second term in the proof is now:

$$\begin{aligned} \gamma\mathbb{E}_\pi[G_{t+1} \mid S_t = s] &= \gamma \sum_{g\in\Gamma} \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} gp(g \mid s')p(s', r \mid a, s)\pi(a \mid s) \\ &= \gamma \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']p(s', r \mid a, s)\pi(a \mid s) \\ &= \gamma \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} v_\pi(s')p(s', r \mid a, s)\pi(a \mid s) \end{aligned} \tag{9}$$

Now, Combining the two terms completes the proof:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \sum_{a\in\mathcal{A}} \pi(a \mid s) \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} p(s', r \mid a, s)[r + \gamma v_\pi(s')] \end{aligned} \tag{10}$$

We have written the above equation in terms of value function, writing that in terms of Q function we get:

$$Q^*(s, a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a\right] \tag{11}$$

We have to estimate the value function by using Bellman Equation as an iterative update. Such functions converge to an optimal action value function. In practice, this basic approach is totally impractical because the action-value function is estimated separately for each sequence, without any generalization. Instead, it is common to use a function approximator to estimate the action-value function, $Q(s, a; \theta) \approx Q^*(s, a)$. In the reinforcement learning community this is typically a linear function approximator, but sometimes a non-linear function approximator is used instead, such as a neural network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q(s, a; \theta_i)\right)^2\right] \tag{12}$$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) \mid s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a that we refer to as the behaviour distribution. The parameters from the previous iteration θ_{i-1} are held fixed when optimizing the loss function $L_i(\theta_i)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. Differentiating the loss function with respect to the weights we arrive at the following gradient,

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right] \quad (13)$$

We optimize the above loss function using stochastic gradient descent, which updates the weights batch-wise. This algorithm is model-free, and it solves the reinforcement learning tasks using samples from the environment. It is also off-policy, i.e., it learns about the greedy strategy while following a distribution that ensures adequate exploration of the state space.

4. Methodology

Deep Reinforcement Learning: The goal here is to combine the reinforcement learning algorithm with a deep neural network that operates directly on RGB images and processes the training data by using some optimization technique like Adam or SGD/SGD with momentum. Here, this paper uses a technique called experience replay where the agent's experience is stored at each time step $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset $\mathcal{D} = e_1, \ldots, e_N$, pooled over many episodes into a replay memory. And we are applying the Q-learning updates to the sampled experience.

Algorithm:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Using experience replay has several advantages over online learning, so primarily each step of experience is potentially used in many weight updates, and it allows for good data efficiency, also learning directly from the consecutive examples is not efficient due to high correlation with temporally preceding data. This also prevents the algorithm to diverge and smoothens out the learning process.

So, while using this algorithm they store N experience samples in the memory and sample D and perform updates on them. This approach doesn't differentiate important transitions and tend to overwrite with recent transitions due to finite buffer size N.

Convolution Neural Network Architecture and Data Preprocessing: As a data preprocessing step this algorithm tries to reduce the input dimensionality. The raw frames are preprocessed by first converting the RGB to grayscale and down-sampling it. The final input representation is a cropped 84*84 region of the image that captures the playing area. Also, the input is produced by stacking 4 frames from history and feeding them to the Q-function.

For this problem we use a Convolutional neural network with an input dimension of 84*84*4 image. The first hidden layer convolves 16 8*8 filters with stride 4 with the input image and applies a Relu non-linearity. The second hidden layer convolves 32 4*4 filters with stride 2 and followed by a Relu non-linearity. The final hidden layer is fully connected and consists of 256 rectifier units. The output layer is a fully connected linear layer with a single output for each valid action. The number of valid actions is 4 for breakout-v1.

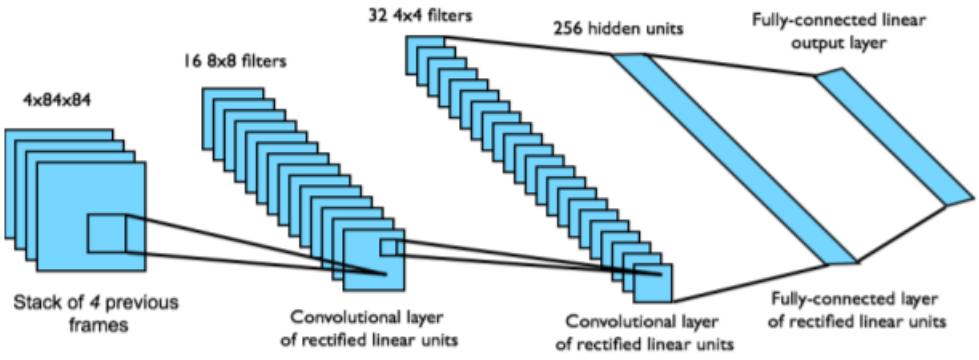

Fig.1: CNN architecture [4]

5. Simulation

This section gives a brief overview of each class and its methods and attributes in the simulation code. The code structure has following three files:

a) **models.py:** This file defines the Convolution model class which inherits from the nn.Module parent class. The constructor initializes observation shape and number of actions variables, defines the Sequential relation between Convolution layers and fully connected layers. Finally, it defines the optimizer as 'Adam' with an input learning rate. Furthermore, it defines the forward propagation for the model by combining the conv and fc layers. This is a standalone file i.e., the model is tested in the file itself and it can be run independently.

b) **utils.py:** This file contains the input data preprocessing class i.e., FrameStackingandResizingEnv. This acts as a wrapper class to concatenate n frames together and resize them before passing them to the conv model. The constructor initializes variables like buffer array, width, height of the input, env, frame etc.

It also contains the preprocess_frame method which resizes and converts the image from RGB to gray. Furthermore, it has a step function which preprocesses the frame and performs a step in the environment and fills up the buffer array. It also contains property functions which return action space and observation space.

The reset method and the render methods are the overloaded functions which works with our input data. This class is tested by running utils.py.

c) **new_agent.py:** This file contains the main training script of our agent. It contains a dataclass which provides data instances (state, action, reward, next_state and done) to insert to the buffer array. The ReplayBuffer class is used to implement the technique called experience replay and it stores N previous examples in the array and sample D at the training time and optimizes them. The insert function inserts the dataclass instance to the buffer array and increases the index. The sample function samples D examples using the random module's sample function. The update_tgt_function copies the state dictionary from our model to the target model.

The train step function extracts the individual data elements from the sampled examples and calculates the loss using the bellman equation and backpropogates it through the network. The run test episode function is for logging the episodes in the weights and biases api. This api is used for visualizing the loss, average reward, epsilon etc. over the training period.

The main function is responsible for integrating all the classes and train the agent and log the training progress. It takes in arguments like mode, checkpoint, device etc. Here, we initialize the necessary variables and start the breakout-v0 environment, instantiate the model and the ReplyBuffer class. In the main training loop, we use epsilon decay policy to choose the action randomly or from the last_observation. Next, we input this action in the environment and observe the new state and reward and if the game is not done, we log the loss, epsilon, and average reward. We also update the tgt_model after a certain number of epochs.

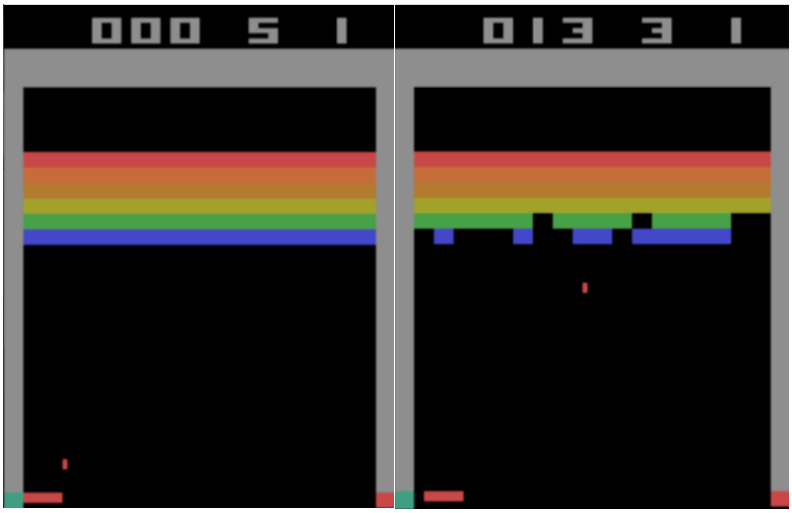

Fig.2: Breakout Simulation

6. Results

During the training of the agent, the performance of the model can be tracked on the training and test data. We compute the total and average rewards during the training and plot them to see how the network progresses. We also decay epsilon at each iteration to balance the exploration vs exploitation aspect. The average total and test rewards tend to be very noisy due to the small changes in the weights of the policy which results I large changes in the distribution of states the policy visits.

We use a learning rate of 0.0001 with epsilon decaying from 1 to 0.1 in 14 million steps. We also use a batch size of 32 with replay buffer size as 50,000. Also, we are able to get an average reward of 45 at around 9M steps. Following plots show the evolution of the agent during the 14M steps, we also save the checkpoints between training to recover best weights if the average reward starts to go down due to overfitting. Please refer to this [link] for the video simulation and this [link] for the GitHub repository.

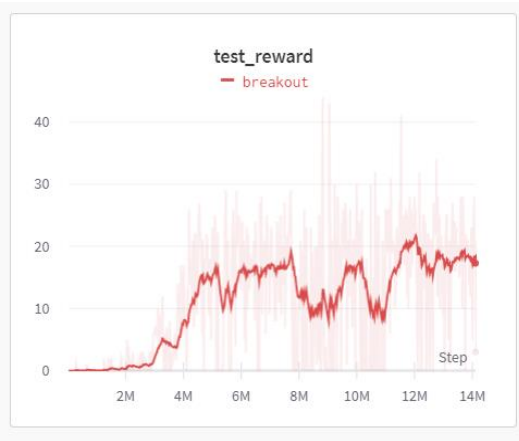

Fig.3: test reward

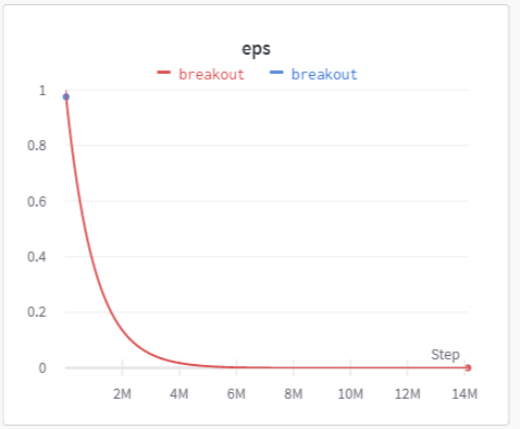

Fig.4: epsilon

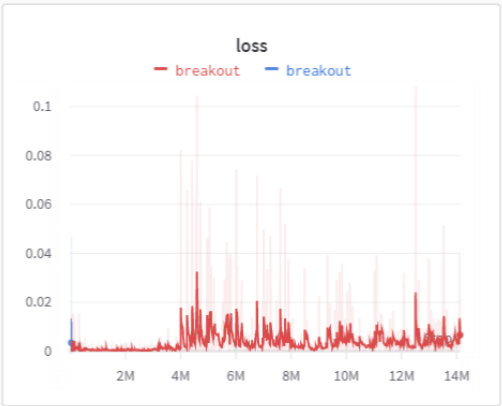

Fig.5: loss

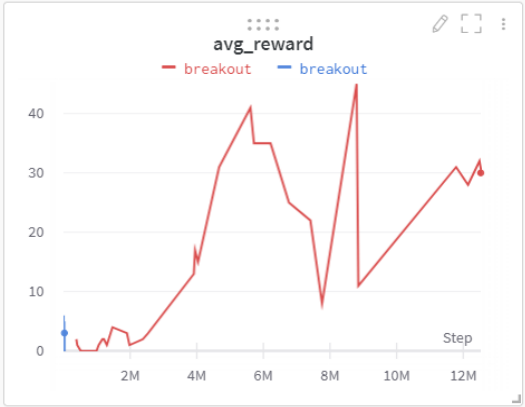

Fig.6: Average Reward

7. Future Research

Future Research for deep reinforcement learning diverges in two different directions. To improve the performance of the current system we can either improve the RL algorithm and use methods like Double DQN, Dueling DQN, Actor Critic, Deep Deterministic Policy Gradient

etc. or we can use more complex architectures like Resnet, InceptionNet, VGGNet etc. with more parameters to learn complex policies on other environment with much more actions, states, and observation space. Generally, the community use both the techniques to achieve a significant performance boost. Also, the hyperparameters are not tuned for the convolution layer, that can also help to improve the performance of the system.

8. Conclusion

This paper presented a novel way to combine Reinforcement learning with deep learning to predict the control policies for the Atari 2600 Breakout game using only image pixels as the input. The training process involves using a variant of Q-learning with mini-batch Adam (RMSprop with momentum) optimization with a technique called experience replay memory to ease the training of deep networks. This approach is groundbreaking for RL, and it easily outperforms human experts on the Atari 2600 games.

9. Bibliography

[1] *Open AI gym Environment Documentation [link]*

[2] *PyTorch API Tutorials [link]*

[3] *ImageNet Classification with Deep Convolutional Neural Networks – Alex Krizhevsky [link]*

[4] *Reinforcement Learning: An Introduction – Richard S. Sutton and Andrew G. Barto [link]*

[5] *An Introduction to Markov Decision Processes – Bob Givan and Ron Parr [link]*

[6] *Deep Reinforcement Learning – David Silver – Google Deepmind [link]*

[7] *Residual algorithms: Reinforcement learning with function approximation [link]*

[8] *A neuro-evolution approach to general Atari game playing [link]*

[9] *Actor-critic reinforcement learning with energy-based policies [link]*

[10] *Deep Reinforcement Learning in Action – GitHub Repository [link]*

[11] *Bayesian Learning of recursively factored environments – Marc Bellamare, Joel Veness, [link]*

[12] *Machine Learning for Aerial Image Labeling - Volodymyr Mnih's thesis [link]*