Pixel to Pinball: Using Deep Q Learning to Play Atari

Adam Rosenberg
School of Engineering and
Applied Science
University of Virginia
Charlottesville, Virginia 22904
Email: ahr7ee@virginia.edu

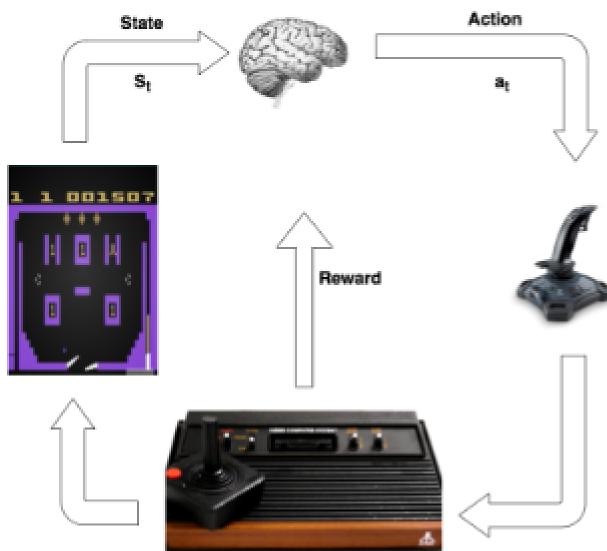
Gautam Somappa
School of Engineering and
Applied Science
University of Virginia
Charlottesville, Virginia 22904
Email: gs9ed@virginia.edu

***Abstract*—In this paper, we investigate Reinforcement Learning methods to learn how to play an Atari game, Video Pinball. More specifically, we use the Deep Q Learning algorithm, a recent extension of the more traditional Q Learning algorithm, which is a tabular method. To implement this algorithm, we used the Keras package in Python, which is a widely used deep learning library. The environment of the game is obtained through OpenAI Gym, a website hosting environments of various Atari games. After 8 days of training, our results were a modest improvement in performance over the random strategy.**

***Index Terms*—Atari, Reinforcement Learning, Deep Q Learning, Neural Networks, Pinball.**

I. Introduction

The field of Artificial Intelligence intends to simulate how a human being learns to complete an arbitrarily difficult task. Similar to how humans learn to ride a bicycle, an AI is trained to learn by trial and error. This model requires an environment, which can be represented by a game; a "brain", which serves as the AI agent; feedback, to be received by the agent after each action, and consisting of the reward and next state of the environment, wherein the reward is generally defined by the game designer.



There are many algorithms that define the behavior of the AI agent, including SARSA, Temporal Differencing [4], and Value Iteration. The most relevant traditional Reinforcement Learning algorithm for the purpose of this paper is Q-learning, which seeks to compute the value of each state-action pair. Its extension, Deep Q Learning, is the ideal algorithm to complete the training of the Atari learning due to the excessively large state space. Keras is the Deep Learning library that we used to conduct the training of the model.

A. OpenAI Gym

OpenAI [5] Gym is a toolkit written in python that provides the environments for games allowing for the easy implementation of reinforcement learning algorithms. This website is extremely useful for reinforcement learning research because previously, the researchers needed to write the code to simulate the environment of the game that they were trying to develop and algorithm for. Now that this time-consuming step has been removed, they can focus exclusively on training the best possible model without the underlying implementation details. Another useful feature included on the website is the availability of code and results for previous attempts to "solve" the game. It is therefore possible for a user to upload their solution and see how it compares to previous results. Currently, the most prominent applications available on the website are games from the Atari system, but there are other developing categories, such as Board Games, Parameter Estimation, and even Doom.

Fig. 1. OpenAI Code Snippet

```
import gym
env = gym.make('CartPole-v0')
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
```

B. Arcade Pinball

Arcade Pinball is an Atari game released in 1980. The goal of the game is achieve the highest score possible by hitting the ball with the bumpers. If the ball falls into the pit enough times, then the game ends. Bonus points can be acquired when the ball moves through the tunnels on the top right and top left of the screen, as well as when it moves through special symbols that appear occasionally. There is also a tilt function which moves the ball if it is stuck.

In terms of the environment representation, we chose to use the video input, which is a 210 by 160 by 3 array indicating the RGB configuration at each pixel on the screen. This would be what the human player sees. However, OpenAI Gym also had a version of the game where the environment was a 128 byte RAM. This representation would be what the Atari's processor sees. Both methods are valid ways to represent the game state on the Atari system.

We ended up choosing the pixel representation because it seemed more intuitive and it adds a computer vision aspect to the problem, justifying the use of convolutional neural networks. One potential issue is that the pixel representation has a much larger state space, since there are theoretically $256^{210*160*3}$ different state configurations, versus only $2^(8 * 128)$ for RAM. This discrepancy means that the training time could be signficiantly longer for the pixels. However, the interpretation of our pixel-based results should make more sense. In addition, the current results on OpenAI for this problem are less robust than they are for RAM, meaning that this problem is unsolved and there is more potential for discovery.

Fig. 2. OpenAI Code Snippet

II. Related Work

Mnih, V., et al. proposed the novel technique of Deep Q Learning to learn how to play Atari games [1]. Their technique was the first deep learning model that was capable of learning control policies from the high-dimensional sensory input, through the use of reinforcement learning. Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [6, 7, 8] and speech recognition [9, 10]. These methods utilise a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines and recurrent neural networks, and have exploited both supervised and unsupervised learning. Their overarching model was a convolutional neural network, but it was trained with an offshoot of Q-learning, where the input was represented by the individual pixels and the output was the value function predicting the future rewards. Their method was applied to seven Atari games found from an online source, and they did not adjust the neural network architecture or learning algorithm at all. This lack of adjustment showed that their model was capable of learning how to play each game without any human intervention. In the end, their model outperformed the state-of-the-art for six of the games, and beat the best human players for three of them.

The idea of Q Learning, which served as the foundation for further research in Deep Q Learning, was proposed in 1989 by Watkins [2]. The details of exactly how Q-Learning works will be explained further, but the essential aspect of this paper is the proof that it will eventually converge. The property of convergence is necessary for reinforcement learning algorithms because without it, there is no way to know whether the algorithm will actually terminate. If this is not the case, then the training process could take infinitely long, without any noticeable progress in the learning. The key to this convergence proof was the application of the action-replay process, a construction from the episode sequence as well as the learning rate sequence.

LeCun, Y., et al. [3] laid the groundwork for the modern day implementation of neural networks. His research focused on the application of the backpropagation algorithm to update the weights of the neural network, reducing the empirical error rate. The specific problem he focused on was the automatic classification of handwritten ZIP code digits available from the U.S. Post Office's MNIST database. The main challenge with this dataset was the high level of noise due to the vastly different styles of handwriting that different people have, as well as the similarity that several pairs of digits have with each other, such as 4 and 9. Previous attempts to classify these digits used feature vectors as the input, which produced middling results, but this time he fed the images directly into the neural networks, which demonstrated the impressive ability of neural networks to process vast amounts of low-level information using this backpropagation approach. In the end, the results were a 5% error rate on the test set, which is certainly not ideal, but is an important first step in prving that neural networks are applicable to real machine learning problems.

III. Methodology

We will now discuss the algorithm that we used to learn pinball, Deep Q Learning, as well as its tabular predecessor,

Q Learning; in addition, we cover the deep learning library used to train the network, Keras.

A. Q Learning

The goal of Q-Learning is to learn the action-values $Q(s, a)$ off-policy. We adopt a greedy policy such that $\pi(S_{t+1}) = argmax_{a'} Q(S_{t+1}, a')$, but with the epsilon-greedy rule: with probability ϵ, choose a random action instead of the action currently with the highest value. The benefit of the epsilon-greedy strategy is that it introduces the possibility of exploring other actions instead of staying with the current best actions, which avoids hill-climbing, i.e. only caring about local maxima. This distinction is an area of active research in reinforcement learning, called the Bandit Problem. One of the biggest challenges is choosing an optimal epsilon value such that the algorithm will converge in a reasonable amount of time, but also find a satisfactory solution. For convenience, the full Q-Learning algorithm is listed below:

Fig. 3. Q-Learning with epsilon-greedy strategy

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$;
  until S is terminal

B. Deep Q Learning

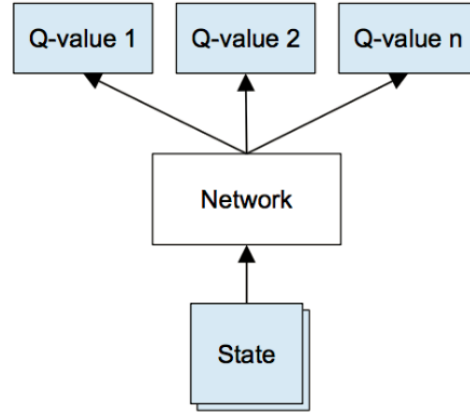
The key motivation for using deep learning in this problem is that the state space is exponentially large. Using Q-Learning in its base form is unrealistic because there are $256^{84*84} * 4$ state-action pairs. Another issue with tabular methods is that the entries in the Q matrix are extremely sparse in a state space of this size, since it is extremely rare for the exact same state configuration to repeat during training. Neural networks avoid this problem because the state-action pair values do not need to be stored in memory; rather, the network's weights are adjusted so that the value can be obtained by feeding the corresponding inputs.

One important efficiency improvement for Deep Q Learning is that the only input to the neural network is the game's current state, as opposed to a state-action pair. Instead, the output of the neural network is a series of Q-values, one for each action. In ordinary supervised learning we would feed an image to the network and get some probabilities.In an implementation we would enter gradient of 1.0 on the log probability of one class and run backpropogation to compute the gradient vector

$$\nabla(_w \log p\ (y \ = \ class1 \mid x))$$

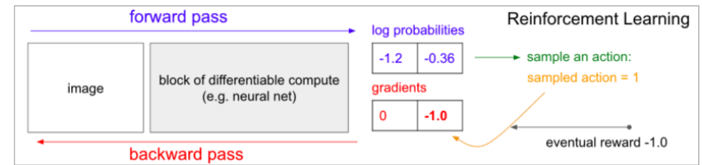
. This gradient would tell us how we should change every one of our million parameters to make the network slightly more likely to predict one class. This way, if we do a parameter update, our network would now be slightly more likely to predict that class when it sees a very similar image in the future.

Fig. 4. Deep Q Learning Architecture



But the same cannot be said about reinforcement learning, because we do not have the correct label. So in our case, we sample an action from the distribution and execute it in the game. We also update the gradient vector so that it performs the same action in the future.

Fig. 5. Weight Updates in



The loss function, used to compute the error in the back-propogation step of the neural network algorithm, is defined as

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

. The algorithm will be briefly summarized for a given transition (s, a, r, s'). First, we feed s into the neural network, where a Q value is predicted for every action. Next, we feed s' into the neural network, and calculate the action a' with the best value Q'. Then we set the target Q value for the current action to be $r + \gamma Q'$, and set all of the remaining target Q values to be the same as their predictions, ensuring that the other errors are 0. If this were not the case, then backpropagation would unfairly penalize other actions that are not currently under consideration. Finally, the weights in the neural network are updated according to the normal backpropagation rules.

C. Network Architecture

As explained in the previous section, the state spaces for video pinball is exponentially large and it is nearly impossible for the system to explore each state space and learn from it. Hence, to overcome this difficulty, we are using an approximation that gives results almost as good as exploring all the states.

Fig. 6. Deep Q Learning Algorithm

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

A policy network is a network of convolutional networks that implement our player. This neural network will take the state of the game and decide what action should be taken. The action in the case of pinball can take 9 values. The paddles can move up and down, which equates to 4. The players have to option to tilt the game to control the ball direction, this is another 4 and the action to hit the ball when starting. For our network, we have use a five layer network where the first three layers are convolutional layers and the last two layers are fully connected layers. We have used stochastic policy, meaning that we only produce a probability of taking one action. After every iteration, we will sample from the distribution to get the actual move.

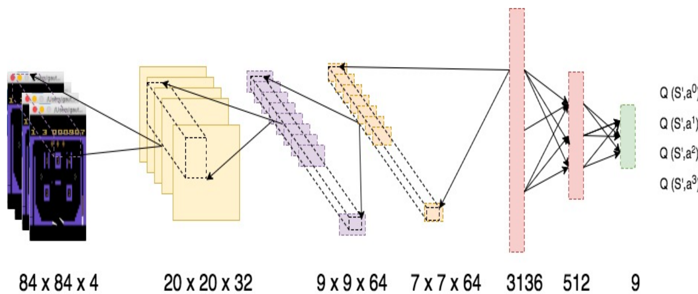


Fig. 7. CNN Architecture

The weights of these neural networks are initially randomized,as the game progresses. The network learns to stabilize its weights which in turn determines the action taken by the agent.

For training images, we are feeding 4 images in sequence, this is done so that the system is able to estimate the velocity of the ball in the game. The input image for the game is of size (210 x 180 x 3), we re-size this to 84 x 84 x 1. The last dimension corresponds to the channels of the image we are dealing with. We have converted RGB to grey scale because we do not need to work with colours as we are only concerned with the location of the ball. The image size is reduced to 84 x 84 as this reduces the number of states and makes computation easier.

The first layer in our network is a convolutional layer. Convolutional Neural Networks are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

Our model Parameters:

- INPUT [84 x 84 x 1] will hold the raw pixel values of the image, in this case an image of width 84, height 84, and with one channel.

- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. In our model, we have three convolutional layers.
  - The first layer has 32 filters and a stride of (4,4) with a ReLU activation
  - The second layer has 64 filters with a stride of (2 , 2) and ReLU activation
  - The third layer has 64 filters and ReLU activation.

  We use ReLU activation because it converges faster and it is typically used for large and complex networks. ReLU essentially gives $\max(0, x)$. Stride controls how the filter convolves around the input volume,The amount by which the filter shifts is the stride.

- Fully Connected layer will compute the class scores, resulting in volume of size [1x1x9], where each of the 9 numbers correspond to a class score, such as among the 9 actions of the game. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume. In our model, we have used two FC layers. The first layer is of dimension 3136 and the second one has a dimension of 512.

- Softmax is done on the last layer of Fully connected layer to get the log probability of actions

ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other dont. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons).

D. Exploration-Exploitation

One issue that needs to be addressed by the policy network is exploration versus exploitation. The former property is

desirable early in the network's life cycle in order to find new actions that could result in a greater reward sequence than what has already been found. Exploitation, on the other hand, is useful after a long period of training because it ensures that the agent is maximizing the value from each state by taking the established optimal action.

For the purposes of the Q-network, we need to keep in mind that the Q-values are initially randomized. That means that the network initially produces random predictions. This is why the Q-network performs exploration more early on, because the Q-values have not converged yet. However, the exploration stage without any modifications will be greedy, which results in a hill-climbing approach. As discussed earlier, we implement the ϵ-greedy strategy, which results in a random action taken with probability ϵ. A common approach is to start with a high value of ϵ, and shrink it down as time passes, so that the exploration phase gradually shifts to exploitation. It is evident that the proper choice for the ϵ parameter is instrumental in achieving the optimal balance between exploration and exploitation.

E. Experience Replay

The computation of Q-value through the means of non-linear functions can be highly unstable, so some extra measures need to be taken to ensure that these approximations will actually converge in a reasonable amount of time, i.e. a week on one GPU. Besides the ϵ-greedy strategy, the next modification is experience replay. As the game progresses, every experience (s, a, r, s') will be stored in memory, to be replayed later. As the network is training, we use a random minibatch of recent experiences as opposed to only the last transition. The effect of this step is to decorrelate the sequence of training samples, with the intention of avoiding a local minimum in error. Additionally, experience replay makes the algorithm more similar to supervised learning, so it becomes easier to debug and test.

IV. Experimental Results

We ran experiments for 100 epochs. Every epoch was recorded in a graph plotted against rewards. Training was done on CPU and it took about seven days for the system to complete training. In the end of the training, we saw substantial improvement in the behavior of the model. We also saw increase in average reward through subsequent training. This shows that the system improved on time.

From Fig 8. above, we can see a sudden increase in the reward after which is reduces and settles on zero. The initial spike in reward is because the system is using random actions before it starts learning. This random action is enough to keep the ball moving in the game, hence the system seems to be performing well. But, after the 20th epoch, it settles to zero. At this point, the system is sampling from previous experiences and trying to fit the model. Most of the predictions are incorrect as can be seen from the graph.

After 7 days of training, you can see that the average reward is increasing over time. There are some spikes in the graph, this is because we introduce some form of randomness in the model to make it not biased. But overall, the average reward is better than it was in the first 200 epochs.

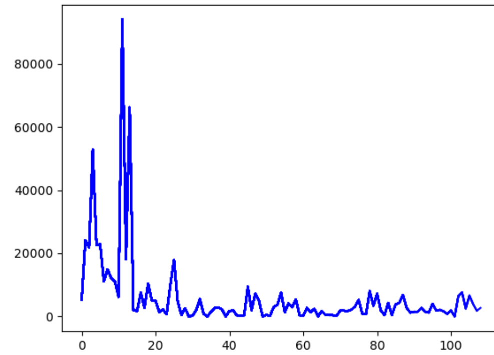


Fig. 8. Day 1 of Training

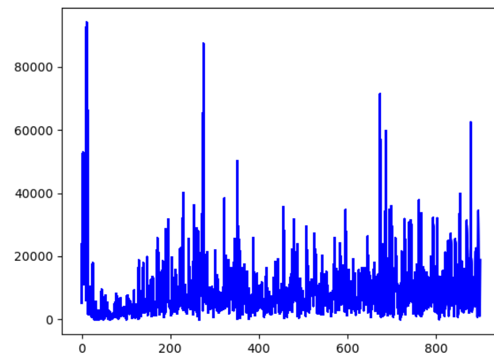


Fig. 9. Day 7 of Training

V. Conclusion

In this report, we address the issue of training an AI agent to play video pinball. We have implemented a deep q network for the model to train and we have used OpenAI's gym to load our environment. The model was trained for seven days and finally it was able to play pinball. The performance of the model was not comparable to a human, because it made some mistakes while predicting the location of the ball. But we think that over time, it can surpass the high score in pinball given enough training time.

This is just an example of what a machine can do. We can import the same model to work on other real world applications such as training an AI agent to deliver pizzas, develop an All-terrain robot and automate machines to deliver packages.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (Dec 2013). Playing Atari with deep reinforcement learning. Technical Report arXiv:1312.5602 [cs.LG], Deepmind Technologies.

[2] Watkins, Christopher JCH, and Peter Dayan. ”Q-learning.” Machine learning 8.3-4 (1992): 279-292.
[3] LeCun, Yann, et al. ”Backpropagation applied to handwritten zip code recognition.” Neural computation 1.4 (1989): 541-551.
[4] A. J. Lipton and H. Fujiyoshi and R. S. Patil. Applications of Computer Vision, 1998. WACV '98. Proceedings., Fourth IEEE Workshop on, ”Moving target classification and tracking from real-time video”
[5] https://gym.openai.com/
[6] K. Simonyan, A. Zisserman. ”Very Deep Convolutional Networks for Large-Scale Image Recognition”
[7] C. M. Bishop. ”Neural networks for pattern recognition”. Oxford university press, 1995
[8] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In CVPR, 2014.
[9] Oord, Aaron van den, Dieleman, Sander, Zen, Heiga, Simonyan, Karen, Vinyals, Oriol, Graves, Alex, Kalchbrenner, Nal, Senior, Andrew, and Kavukcuoglu, Koray. ”Wavenet: A generative model for raw audio”. arXiv preprint arXiv:1609.03499, 2016.
[10] Agiomyrgiannakis, Yannis. Vocaine the vocoder and applications is speech synthesis. In ICASSP, pp. 42304234, 2015.