Playing Kung Fu Master with Reinforcement Learning

Frank Ibem and Ruraj Joshi,Texas Tech University

Abstract—Kung Fu Master (KFM) is a classic Atari 2600 game released in the 1980s. Many players still try to beat the world high score in the game to this day. The recorded high score is 95540 as of 2016. Recently, with advances in reinforcement learning (RL), we have progressed from humans playing these classic games for high scores to a more automated approach - let the computer play by itself. This has been facilitated by platforms like OpenAI Gym which provide a collection of environments for training RL agents. These platforms provide simple abstractions over the dynamics of RL environments - states, actions and rewads. We combine the use of a double deep Q-network with prioritized experience replay to train an agent to play Kung Fu Master. To demonstrate that the agent indeed learned, we compare its performance to that of a baseline random agent which takes random actions at each without any other considerations. While the random agent scored an average of 582 over 100 episodes, our agent obtained an average score of 5526.

Index Terms—Atari, Deep Q-Network, CNTK, Prioritized Experience Replay

1 Introduction

KUNG Fu Master is a classic Atari 2600 game where the player takes control of a protagonist Thomas who is an expert kung fu master. The goal is to save his girlfriend Sylvia from the evil crime lord Mr. X. It is a challenging problem to tackle using reinforcement learning with many different types of enemies and actions to learn. Section 2.1 briefly describes reinforcement learning and its important components. Section 2.2 covers the Q-learning method of training RL agents and several pitfalls. Section 2.3 introduces the deep Q-Network and the use of a buffer for experience replay. Sections 2.4 and 2.5 cover the Double Deep Q-Network and prioritized experience replay respectively which have been found to improve learning. In section 3 we show how we have combined these methods to train our agent and the steps taken to preprocess our data. Section 4 describes how we have evaluated our agent and we conclude in section 5.

2 Background

2.1 Reinforcement Learning

Reinforcement learning is a machine learning area inspired by behaviorist psychology. It is based on the idea that agents ought to take actions to maximize some notion of cumulative reward. In typical RL problems, we have an agent interacting with an environment by taking actions which influence said environment. For a given state, the agent takes an action which sends it to the next state and gives it a reward. In KFM, states are pixel images and rewards are points for getting past an enemy. The agent's actions are kicking, punching and so on. RL tasks can be episodic or continuous. Episodic tasks end in a special terminal state after which they are reset to some starting state whereas continuous tasks cannot be easily broken up into episodes but go on continually without limit[4]. KFM is an episodic task since it has a defined start state and terminates when a player loses all her lives. An agent also tries to select actions such that the sum of the discounted rewards it receives over the future is maximized[4]. It chooses actions to maximize the expected discounted return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where R_t is the reward received at time t and $0 \leq \gamma \leq 1$ is the discount-factor. If $\gamma = 0$ then the agent only considers immediate rewards and is short sighted whereas as γ approaches 1, the agent considers future rewards more strongly, becoming farsighted. The goal of RL methods is to learn a policy (denoted by π) – a function from states to actions which determines what action an agent should take in each given state.

2.2 Q-Learning

Although several methods exist for obtaining optimal solutions to RL problems (such as dynamic programming, Monte-Carlo methods, temporal difference methods and so no), we have chosen to use Q-learning, a temporal difference method. Q-learning is a model-free method because the agent is not required to have any prior knowledge of the environment. It is also online because the agent learns and improves as it interacts directly with the environment. Q-learning learns an action-value function:

$$Q(S_t, A_t) = R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$$

The action-value function gives the value of taking an action, A_t, in a state, S_t, and calculates this value as the reward received for taking that action plus the expected discounted future reward while following the current policy. To maximize the expected return, we can then opt to always select the action with the highest reward in every state. Starting out, every action has the same value. If we always select the greedy action, we are said to be exploiting our current knowledge. However, doing so all the time could prevent us from learning potentially better actions. As a

result, we use ϵ-greedy action selection instead where an agent selects a random action with probability ϵ and selects a greedy action with probability $1-\epsilon$. When a random action is selected, the agent is said to be exploring. ϵ can start at 1 and decay with time to allow the agent to explore a lot at the beginning but less and less as its knowledge improves with time. After every step, the target, y_t, is calculated and used to update the action-value function (the difference term is called the temporal difference):

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S',A)]$$

For simple problems, the action-value function can be represented using a table. However, this representation would be impractical for problems with high dimensions like KFM or those with continuous actions. An alternative would be to use a function approximator like a neural network. Such a representation could learn to associate groups of states with a specific value thereby conserving memory. However, with a function approximator, learning becomes unstable or may diverge for several reasons. Consecutive samples are highly correlated and updates to Q could significantly change the policy and hence the data distribution. Q-learning also tends to be optimistic since it uses the max operator in its update which could lead to over estimation of action values[2].

2.3 Deep Q-Network

To solve some of the problems introduced when function approximators are used with Q-learning, Mnih et al. introduced the Deep Q-Network (DQN). The DQN introduced the use of a replay buffer for experience replay and a target network for improved stability. The replay buffer has a fixed capacity and is used to store transitions (S_t, a, r, S_{t+1}) of the current state, the action taken, the reward received and the next state. When it is time to learn, transitions are sampled uniformly at random. This improves data efficiency as transitions are used in multiple updates. Correlations between observations are also broken by using uniform random sampling[2]. The target network is initialized with the same weights as the online network but is periodically updated by copying over the weights of the online network. The online network is used to select actions and the target network is used to compute the targets for updates. The target network stabilizes learning by reducing correlations between action values and targets. The target is calculated as

$$y_t^{DQN} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-)$$

and the loss as

$$J = (y_j - Q(S,A;\theta_t))^2$$

These functions are parameterized by θ and θ^- the weights of the online and target networks respectively.

2.4 Double Deep Q-Network

Although the DQN reduces the correlation between observations and stabilizes learning, it still overestimates action-values. The question of whether this optimism is bad is still open for research. The double deep Q-Network (DDQN) reduces overestimations by extending the double Q-learning technique to the DQN. Double Q-learning uncouples action selection from evaluation. It learns two value functions by randomly assigning each experience to one of the two value functions such that that there are two sets of weights. For updates, one set of weights is used to determine the greedy policy (action selection) and the other to determine its value (action evaluation)[1]. In DDQN, the target network from the DQN is used as the second value function. Hado et al. have proposed the use of the online network to select actions and the target network to estimate the values of the selected actions. The target for updates is then calculated as

$$y_t^{DDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t); \theta_t^-)$$

2.5 Prioritized Experience Replay

In the original DQN algorithm, transitions are sampled uniformly at random from the replay buffer. It is natural to ask if perhaps some other means of sampling will yield better results. The key idea behind prioritized experience replay is that an RL agent can learn more effectively from some transitions than from others[3]. The temporal difference error is used to prioritize transitions because it gives a measure of how surprising or unexpected a transition is[3]. Rank-based prioritization prioritizes transitions by their rank when sorted using a priority queue, with transitions of higher priority being at the front of the queue. The counterpart to rank-based prioritization is proportional prioritization where the buffer is divided into segments and then a sample is chosen at random from each segment. Proportional prioritization can be implemented using the sum tree data structure. To give an analogy, imagine you get into fights often and feel pain when punched. You may then learn to avoid being punched as much as possible. Now suppose you get into a fight and are punched but feel no pain. This will surprise you (and potentially your attacker) because clearly this experience differs from the norm. Later, you sit and ponder over the situation to make sense of it (adjust your model to account for the unexpected observation). One conclusion could be that something inside you is broken and you need to see a doctor. Another is that you have gained inhuman abilities that make you immune to pain. In any case, it is a situation worth careful consideration.

3 DATA PREPROCESSING AND TRAINING

We have used OpenAI Gym to simulate KFM. OpenAI Gym is a platform for testing RL algorithms and provides simple abstractions for observations, actions, and rewards - the core components of RL tasks. We also used the computational network toolkit (CNTK) for learning. CNTK is a high performant open source toolkit from Microsoft for solving several machine intelligence tasks. It allowed us to express our network structure and operations concisely with little difficulty. For KFM, OpenAI Gym provides the following abstractions:

- States: 210x160 pixel RGB image
- Actions: 14 (punch, jump, crouch, jump kick . . .)
- Reward: 200 for every enemy killed

3.1 Preprocessing

Every state (screen image) is preprocessed before being used as input to our neural network in three steps: converting to grayscale, cropping and downsampling. Figure 1 gives an illustration of the process. The RGB image is first converted to grayscale by taking a mean across its channels. We then crop off the top and bottom portions of the image which we consider to not be influential to learning leaving us with a 60x160 grayscale image. Finally, the image is then scaled down to 40x100. We are left with an input that is 4% of the original size, a significant reduction. We found that preprocessing each image in such a manner largely reduced the training time and improved the performance of training.

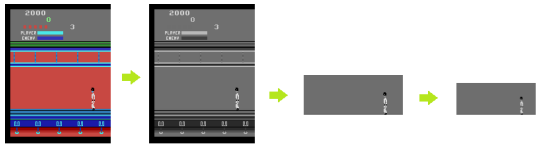

Fig. 1. Preprocessing an input image

3.2 Training

The RL agent was trained with a neural network with the following network structure.

- Input layer: 4 x 40 x 100 (see section 3.2.2)
- 3 convolutional layers with no pooling
  - Eight 8x8 kernels with 4x4 stride
  - Sixteen 4x4 kernels with 2x2 stride
  - Sixteen 4x4 kernels with 1x1 stride
  - All with ReLu activation
- 1 dense/fully connected layer with 256 nodes and ReLu activation
- Output layer with 14 nodes and no activation

The input to the network is 4 stacked frames and each node in the output represents the value of the corresponding action – we select actions with the highest value.

3.2.1 Training Algorithm

The algorithm we have used for training is a combination of all the techniques described beforehand. It is a modification of the algorithm in [3] but without importance sampling. We use a replay buffer of a fixed capacity and a sumtree data structure to prioritize transitions proportionally based on their priority. At each step, we select an action based on the current policy and then store the observed transition in the replay buffer. Every replay period, K, we pick a transition from each of the k segments to compute the TD-errors for the next update. The TD-error is computed using the DDQN update formula and then used to update the priorities of the sampled transitions. We then update the network weights using stochastic gradient descent. Several techniques were applied to improve the training as explained in following sections.

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch size k, step-size η, constant α and replay period K
Initialize replay memory H using random transitions, $\Delta = 0$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t+1}, A_{t+1}, R_t, \gamma_t, S_t)$ in H with maximal priority
  if $t \equiv 0$ mod K **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute TD-error:
      $\delta_j = R_t + \gamma_j Q_{target}(S_j, argmax_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change:
      $\Delta \leftarrow \Delta + \gamma_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

3.2.2 Stacking Frames

A single frame is not sufficient to provide enough context regarding the motion of objects in the game. For example, with a single frame, it is difficult if not impossible to tell if an enemy is moving toward the player or moving in the opposite direction. To solve this problem, we stack the last 4 frames together so that the agent can learn to discern the motion of different objects on the screen. Figure 2 shows what the stacked frames look like.

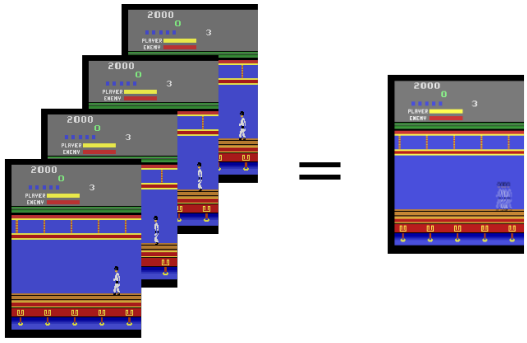

Fig. 2. Stacking frames to allow motion inference

3.2.3 Repeating Actions

When a human player pushes a button on a controller, although it seems to have lasted a mere instant, the button is actually held down across several frames. To simulate the same effect, OpenAI gym stochastically repeats the last selected action for the next n frames, where $n \in \{2, 3, 4\}$. We found that this simplifies learning as the agent can learn to associate values with groups of states as opposed to single states. It also sped up training since we have less states to evaluate.

4 Performance Evaluation

Training was done on an Ubuntu 16.04 virtual machine with 7GB RAM and 2 cores running on Microsoft Azure. Figure 3 show the how the agent's performance increased with time. In the early stages, exploration is high, allowing the agent to learn a lot. As time progresses and the exploration decreases, the agents average performance stabilizes at about 3000 averaged over the last 10 episodes. Similarly, better performance by the agent directly translates into longer episodes as seen in figure 4. The loss with time is also shown.

To establish a baseline against which we could compare our agent, we ran a random agent (takes a random action at each step) for 100 episodes and obtained an average score of 582 rewards. When the same task was performed by our agent, we obtained an average reward of 5526 rewards.

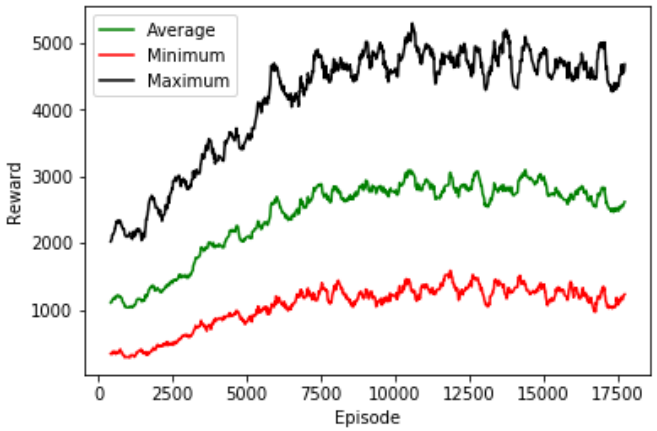

Fig. 3. Progression of reward obtained with time

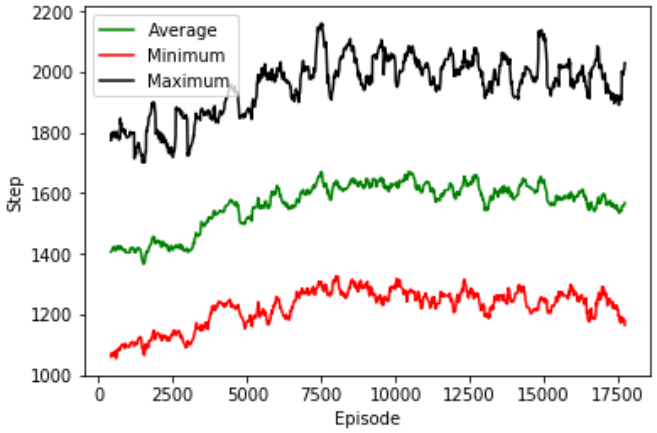

Fig. 4. Steps per episode

5 Conclusion

Since our trained agent significantly outperformed the random agent, we conclude that learning took place. The use of a DQN allowed us to train the agent without any prior knowledge of the dynamics of the environment and the use of DDQN and prioritized experience replay led to substantial improvements. While our score is no where close to best records, we consider the experience a valuable one as it has exposed us to the world of RL. If we allowed external knowledge, then filling the replay buffer at the initial stages with transitions where the agent only punched/kicked left may have sped up learning as those actions would have allowed the agent to kill a lot of enemies. We also hypothesize that we would have achieved even better performance as in the original papers if we used larger networks with more nodes and a replay buffer with a capacity of 1 million as opposed to 50000 like in the references. There is also room for optimizing the various parameters we used.

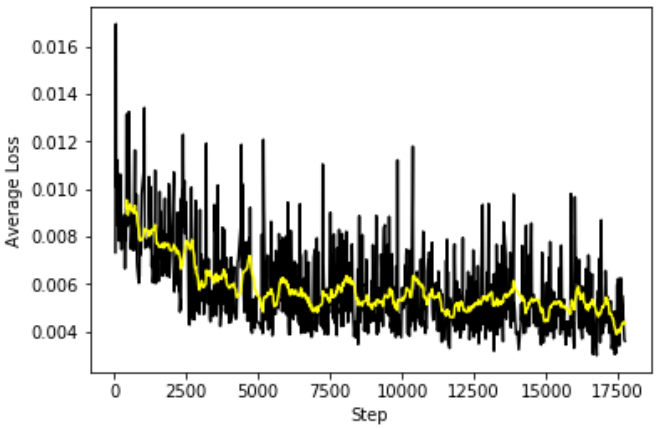

Fig. 5. Average loss per episode

References

[1] van Hasselt, H. et al. 2015. Deep Reinforcement Learning with Double Q-learning. arXiv:1509.06461 [cs]. 2 (2015), 15.
[2] Mnih, V. et al. 2013. Playing Atari with Deep Reinforcement Learning. arXiv: 1312.5602. (2013), 19.
[3] Schaul, T. et al. 2015. Prioritized Experience Replay. arXiv:1511.05952. (2015), 123.
[4] Sutton, R.S. and Barto, A.G. 2013. Reinforcement learning: an introduction. Neural Networks IEEE Transactions on. 9, 5 (2013), 1054.

APPENDIX A
LIST OF HYPERPARAMETERS AND THEIR VALUES

Hyperparameter	Value	Description
minibatch size	64	Number of transitions over which each stochastic gradient descent (SGD) update is computed
replay memory size	50000	SGD updates are sample from this number of most recent transitions
agent history length	4	The number of most recent frames that are stacked to form the input to the network
target network update frequency	50	The frequency with which the target network is updated using the online network's weights
discount factor	0.99	The discount factor used in the Q-learning update
action repeat	{2,3,4}	The last selected action is repeated this many times
replay frequency	16	The number of actions selected by the agent between successive SGD updates
learning rate	0.00025	The learning rate for SGD
initial exploration	1	Initial value of ϵ in ϵ-greedy exploration
final exploration	0.1	Final value of ϵ in ϵ-greedy exploration
ϵ decay rate	0.0003838	Rate at which ϵ is exponentially decayed to its final value