Deep Reinforcement learning on Atari Games

Ammar Haydari, Muhammed Sarmad Saeed
STA 208 Final Project
Department of Electrical and Computer Engineering
UC Davis

1 Introduction

In machine learning, supervised learning makes decision based on the output labels provided in training. Unsupervised learning works based on pattern discovery without having the pre-knowledge of output labels. The third machine learning paradigm is reinforcement learning (RL), which takes sequential actions rooted in Markov Decision Process (MDP) with a rewarding or penalizing criterion. To tackle with high dimensional input states an output actions, a new approach is proposed by Mnih et al. (1) called Deep Q Networks. Deep reinforcement learning is considered as the real Artificial Intelligence and the closest Machine Learning paradigm to human learning, in which Deep Neural Networks and reinforcement learning models are combined for more efficient and stabilized approximations of Q-functions, especially for high-dimensional states problems. Hence, Deep RL becomes more appealing to decision-making problems including control-based research areas such as Atari games, autonomous vehicles, robotics and many other control environments. In this project, we explored the performance of RL agent in two Atari games where deep learning models including NN and CNN are involved

An RL agent follows these steps for taking a an action: first, the control unit collects the state information, which can be in different formats such as raw RGB input or specific information about the environment like position of controller, speed of controller etc., and then control unit takes an action based on the current policy of proposed deep RL method. Finally, agent (control unit) gets a reward with respect to the taken action. By following these steps agent tries to find an optimal policy in order to maximize the score in ATARI game.

2 Deep Reinforcement Learning

2.1 Q learning

In a general RL model, an agent controlled with an algorithm, observes the system state s_t at each time step t and receives a reward r_t from its environment/system after taking the action a_t and the system transitions to the next state s_{t+1}. After every interaction, RL agent updates its knowledge about the environment (2). For value-based RL, value function determines how good a state is for the agent by estimating the value of being in a given state s under a policy π. Adding the effect of action, state-action value function named as quality function (Q-function) is commonly used to reflect the expected return in a state-action pair:

$$Q^{\pi}(s,a) = E[R_t|s|a,\pi] \tag{1}$$

Optimum action value function (Q-function) is calculated similarly to the optimum state value function by maximizing its expected return over all states. Relation between the optimum state and action value functions is given by

$$V^*(s) = \max_a Q^*(s,a), \forall s \in S \tag{2}$$

Q-function Q^*(s,a) provides the optimum policy π_* by selecting the action a that maximizes the Q-value for the state s:

$$\pi^*(s) = \underset{a}{argmax}\, Q^*(s,a), \forall s \in S \tag{3}$$

The values of state-action pairs (Q-value) are stored in a Q-table, and are learned via the recursive nature of Bellman equations utilizing the Markov property:

$$Q^\pi(s_t, a_t) = E_\pi[r_t + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1})] \tag{4}$$

Q^πestimates are updated with a learning rate α to improve the estimation as follows

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha(y_t - Q^\pi(s_t, a_t)) \tag{5}$$

where y_t is the temporal difference (TD) target for $Q^\pi(s_t,a_t)$. The TD step size is a user-defined parameter and determines how many experience steps (i.e., actions) to consider in computing y_t, the new instantaneous estimate for$Q^\pi(s_t,a_t)$. The rewards $R_t^{(n)} = \sum_{i=0}^{n-1} \gamma^i\, r_{t+i}$ in the predefined number of n TD steps, together with the Q-value $Q^\pi(s_{t+n},a_{t+n})$ after n steps give y_t. Q-learning is an off-policy model, in which actions of the agent are updated by maximizing Q-values over the action:

$$y_t^{Q-learning} = R_t^{(n)} + \gamma^n \underset{a_{t+n}}{max}\, Q^\pi(s_{t+n}, a_{t+n}) \tag{6}$$

In the ϵ-greedy policy, a random action is taken with probability ϵ, and the best action with respect to the current policy defined by Q(s,a) is taken with probability 1-ϵ. Fig 1 depicts the schematic of the RL process.

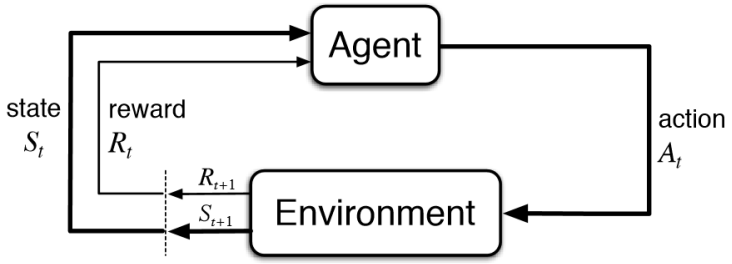


Figure 1: General RL structure (2)

2.2 Deep Q network

The leading approach to solve a large state space and continuous action problems on value-based RL algorithms is Deep Q-Network (DQN) (1). Original DQN receives raw input image as state, and estimates Q-values from them using CNNs. Denoting the neural network parameters with θ the Q-function approximation is written as Q(s,a;θ). The output of neural network is the best action selected using a discrete set of approximate action values.

One of the main parts of DQN that stabilize learning is the target network. DQN has two separate networks denoted as the main network that approximates the Q-function, and the target network that gives the TD target for updating the main network. In the training phase, while the main network parameters θ are updated after every action, target network parameters θ^- are updated after a certain period of time.

DQN also introduces another promising feature called experience replay which stores recent experiences (s_t,a_t,r_t,s_{t+1}) in replay memory, and samples batches uniformly from the replay memory for training neural network. it can prevents the agent from getting stuck into the recent trajectories by doing random sampling since RL agents are prone to temporal correlations in the consecutive samples. Besides, DQN agent can learn over mini-batches that increases the efficiency of the training. DQN can also use experience replay technique to sample experiences uniformly from the memory by changing the sampling distribution of DQN algorithm. It samples with higher TD error to receive higher ranking in terms of probability than the other samples by applying a stochastic sampling with proportional prioritization or rank-based prioritization.

2.3 Settings for Deep Neural Nets and RL agent

Nonlinear function approximators is very efficient for RL based controllers. Today deep learning is a breakthrough for AI and learning model. Applying deep learning techniques to RL provides very good results for complicated controllers. Designing a neural network structure for better performance is another key point for deep RL applications. Multi-layer perceptron (MP), i.e., the standard fully connected neural network model, is a useful tool for classic data classification.

In this project we implemented two games as mentioned earlier. For CartPole our model was purely based on neural nets, as the input was scalar in this case. For BreakOut we had a combination of convolution layers and deep layers, because input in this case were raw images of the game. The details are as follows:

2.3.1 CartPole

CartPole's prediction model is based on non-linear neural net architecture. The model consists of two deep layers. Input to the model are four variables. The position of the Cart, that is where the cart is currently in the frame with respect to both the ends. Second input is the cart speed, that is the speed with which the cart moves once the command is given to it. Third is the position of the pole, this tells where is the pole currently. Last and fourth input is the pole angle, this is the angle the pole forms with the Cart. The position and the angle of the pole play a vital role in deciding when the game ends. If the cart reaches either of the ends, the game ends. The second case when the game ends is when the pole has an angle less than 75 degrees with the cart. So for the network to correctly play the game, it has to keep the pole balanced and remain within both the ends.

The two deep layers each have Relu as the activation function. Both the layers have 24 neurons each. The output of these deep layers is a vector of size two. The output layer has a linear activation function to decide which action to take.

The output of the neural net tells us what actions to take. We are allowed to take only two actions that is whether to move to the left or right such that the pole remains balanced on the cart. The output of the network is a vector of size two and we use the maximum value of the two to decide our action.

For optimization we use for learning ins CartPole is ADAM with a learning rate of 0.001. The loss we use for our gradient descent is Mean Squared Error. And this concludes our model for training to play the CartPole game.

Our RL model utilizes reward as the learning mechanisms to decide what kind of decisions to make. If the reward for a certain action is good or positive this means that the action that the network took led to increase in reward thus this is the right action to take and it learns this. If for a particular action the network is rewarded a negative reward that means that the network should unlearn or never utilize that action under those circumstances. In CartPole game the reward for the game is positive one for an action that leads to increase in score and on the contrary it is awarded a reward of negative 10 for a wrong action which leads to termination of the game. The negative reward is higher because in this helps network learn faster.

Besides Neural Nets we also used other non linear functions to play the cartpole game which are mentioned in the next section.

2.3.2 BreakOut

In Breakout game we implemented two implementations. One of them being the novel implementation mentioned in (1) and the other one where we employ Recurrent Neural Net and made the necessary modifications to the network. And another is a modification to the first one where we add pooling layers and change the optimizer.

The overall implementation for the BreakOut game is similar to that of the cartpole implementation with the major difference being in the input. For cartpole we have a vector as an input but in this case our input is raw image of the game. The image is scaled to 84x84 and it is converted from RGB to Grey scale after pre-processing and before being fed to the network.

The input to all the networks is 84x84x4 image produced after pre-processing where 4 is the number of images that we look at at a time. The first implementation of BreakOut consists of three convolutional layers. The first layer consists of 32 8x8 convolution filter with a stride of of 4. The activation

function used is Relu. The second layer is a 64 4x4 filter with a stride of 2 and Relu as the activation function and in the final convolutional layer we use 64 3x3 filters with a stride of 1 and Relu activation function. The output from the convolutional layers are then flattened and fed to the the deep layer which is the final layer in this network. The deep layer consists of 512 neurons and Relu as the linear function. The output of the neural net is four in our case as we can move the bottom bar only left and right, make it stay and to launch the ball when the game begins. We pick the action with the maximum value/probability after output from linear output neural net layer. The optimizer used for learning was RMSProp and the loss function is the same as the one used in (1) which consists of a quadratic term the mean squared error and a linear term which is variance. The loss is based of bellman equations as an iterative update. This loss is then used in our gradient descent and optimization.

The reward in our case is similar as previously with each wrong action being penalized and each correct action given a positive reward. All the rewards are clipped to -1 and +1 and zeros are left as it is.

Our second implementation is a modification of the above network where we add pooling layers to our network. Our pooling layers are based of max pooling where a maximum value is selected from the pool. The pooling layer is of size 2x2 which helps reduce the dimensions by two. We add two pooling layers, after the first and the second convolutional layers in the previous network. As a result we also modify our last convolutional layer where we change the size of convolution filter from 3x3 to 2x2 where as everything else remains the same. In addition to making changes to the model we also change the optimizer to Adam with default learning rate of 0.001. One of the reason for choosing Adam was that it is one of the best performing optimizer and how would like to know why the authors of (1) chose RMSprop over Adam. The reason for choosing a pooling layer of 2 is not to lose a lot of information. This would also help us learn why the authors did not choose pooling layers which is a common practice when dealing with images as it helps reducing variables and makes computation faster.

Last model that we experimented with breakout was working with RNN, referenced to this paper (3), our new model now consisted of a combination of CNN and RNN layers. The model builds on the first model using the already created layers and adds an additional layer to the model. Doing so we made all the necessary changes to the original code to make it run witht the RNN model. The RNN model we chose was based of LSTM (Long short term memory). The reason behind choosing LSTM was that LSTM helps resolve problem of vanishing gradients which is something that these attari are prone to facing the problem with. The LSTM layer is added before the deep layer and after flattening from convolutional layer. The output of LSTM is then fed to the deep layers.The figure 2 from (3) gives the pictorial depiction of RNN model, with the left layer being at time t-1 and the right one at time t, the only difference being our model has a deep layer for output after LSTM.

2.4 Other Function Approximators on CartPole

We also tested different function approximator models on Cartpole game along with DQN model. The reason we employed this game with different models is that the state definition is not a high dimensional data and different regression models can perform good output. Although neural network based learning models are effective for low dimensional states, we tried two tree based XGBoost (4) and LightGBM (5) gradient boosting models with Ridge regression and K nearest neighbors models. These linear learners and neural network models are compared with each other. We also tried some other linear approximation models like Lasso, Bayesian regression etc. none of these models produced higher scores.

Gradient boosted decision tree structures are highly capable machine learning models.It is known that these three methods generally performs better with structured datasets but we would like to test them on not-structures models as well like RL. LightGBM is the improved version of XGBoost and it is generated by Microsoft in 2017. LightGBM mostly ourperforms the XGBoost tree model. The other highly capable tree structure is catboost but it requires categorical dataset and since our model RL is not categorical, it is not suitable to our application.

For all these function approximator models including linear regression models and decision tree models, we used SKLearn's MultiOutputRegressor API to produce multi label output where we did individual regression for each output class (action).

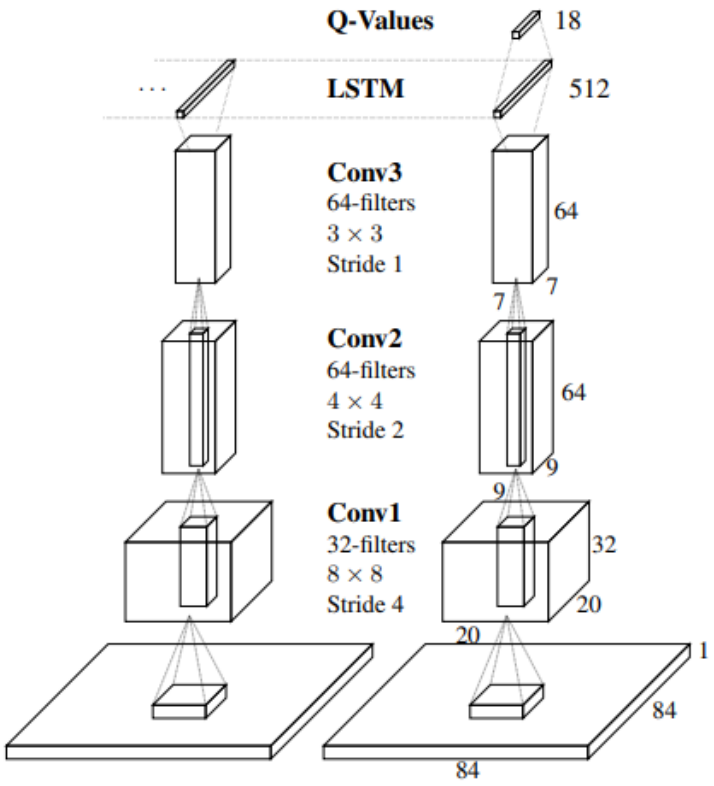


Figure 2: Shows how the layers are stacked together for the RNN implementation.

XGBoost model worked best with defult hyper parameters however in LGBosst model we tried a different number of estimator as 100 rather than default value. For KNN model performs best with 10 number of neighbors. Decreasing the number of neighbors components results in poor performance.

3 Implementation and Experimental Setup

3.1 Implementation

Our implementation consists of a main class Agent encompassing all the functions for the the Deep RL model. Which are briefly described here. The Build Model function consists of layers that we discussed above to build our model for training. The act function is responsible for getting us the action to be carried out and sent to the environment. The remember function saves the states in the memory from which later on we take samples and use them to decide the next state. In the replay function we get the samples from the memory and then get our predictions that we use to compute the loss. The load and save functions are used to load and save our models when training and when testing or to resume.

For the Breakout game we had two extra functions, the building training op function here is a separate function where we apply the gradient update. Unlike in cartpole we update our network and its weights in act next function instead of the main function. We used (6) to get our baseline structure for the code.

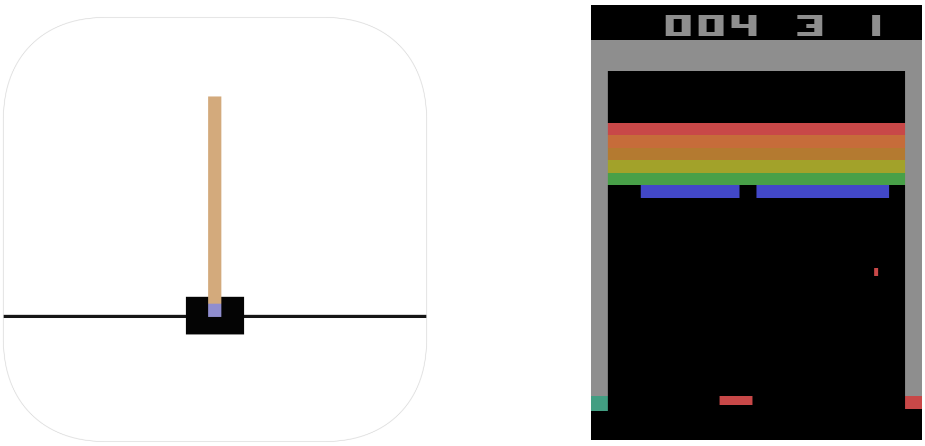
3.2 Experimental Setup

We evaluated the performance of deep RL based TSCs using Atari games on GYM Python API and Tensorflow (7) for deep RL agent. We used Cartpole and Breakout atari game environments with DQN Rl model. We compared our results with linear function approximators on Cartpole and CNN and RNN nonlinear function approximators on Breakout.

We performed episodic RL approach for experiments where we run each game learning agent updates it during the episode. Cumulative total reward as score is the output of each episode which are discussed in the next section with more detail.

Trade-off between exploration and exploitation plays an important role in RL. While agent learns how to behave more it is expected to lead agent to exploit its knowledge instead of exploring the environment more. We used exploration decay trick in our learning model that decreased the epsilon rate gradually while playing the game. We started our epsilon rate as 1 and decreased it to 0.05 as min epsilon rate.

Storing experiences in memory for training the RL agent is a critical point. In standard DQN, this replay memory used for training the neural networks with bathc sampling, however the same concept in linear learner models does not work. In order to improve the learning performance we used all the samples in the replay memory for training linear learning models. We stored recent 1000 samples in replay memory and trained our learners with these samples. Increasing the replay memory do not impacted the performance of learners which only costs us more computational time. This approach gives promising results in such learners and agents gradually learns while playing the game.



(a) Cartpole

(b) Breakout

Figure 3: Atari games for Q learning

4 Experimental Results & Discussions

We will discuss our numerical results in this section. Training RL agents always requires high computational power. Since we are doing a class project, in this report, we presented only the agents that improves its actions. Due to the time constraints, we only trained agents with enough number of episodes. If we keep agent to pay more, the agents would hit higher scores but that training takes a long time.

Cartpole game agents are trained with varying number of episodes. Neural network model, KNN model and Ridge regression models are trained using 100 episodes however, due to requiring high training time, we only trained tree based XGBoost and LightGBM models with 50 episodes. All these training are enough to show learning curve of agents. On the other hand, training raw RGB images requires thousands of episodes and we trained breakout game agents with 13k episodes with different network structures.

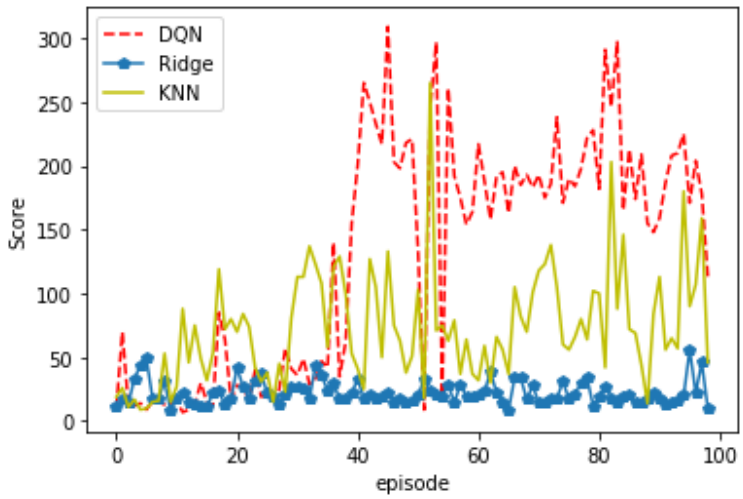


Figure 4: Simulation results for Cartpole game comparing linear approximators with neural networks with 100 episodes

The first figure we would like to discuss compares the performance of DQN model with KNN model and ridge regression model (see Fig 4). We used several other linear regression models but since none of them worked we only showed ridge regression results as an example. It is clearly seen that DQN outperforms the KNN and ridge regression model. After around 50 episodes, DQN results sharply increases and fluctuates around 200-300.

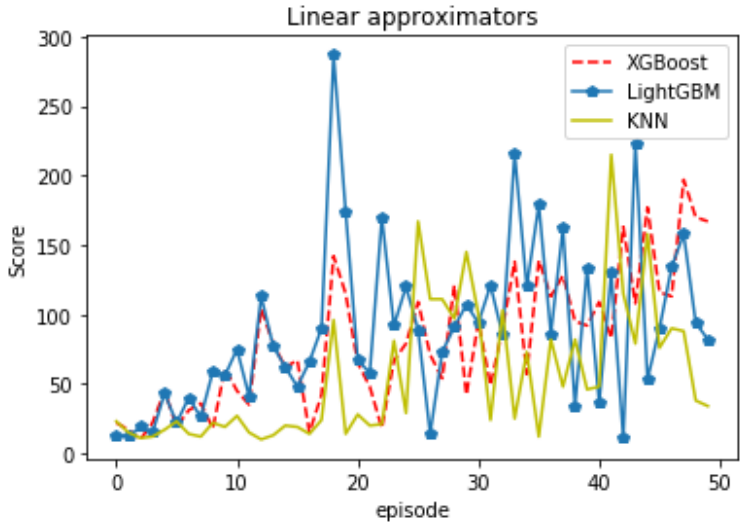


Figure 5: Simulation results for Cartpole game comparing linear approximators with 50 episodes where only KNN and tree based learner models are shown

In Fig 5, we compared the performance of tree structures with KNN, since we know that DQN performs better than KNN, we just used the results of KNN. While, all three models have similar results in this figure, LightGBM is the improved version of XGboost and it performs better than XGboost. We also compared the maximum scores of all models in Table 1 with 50 episodes. As it is seen that, DQN always performs the best performance and LightGBM follows it as having the second best score.

In this project, we wanted to experiment learning performance of breakout game with different neural network structures. 6a shows the results of three different settings of DQN agents with 13k episodes.

Table 1: Maximum scores reached by different approximators

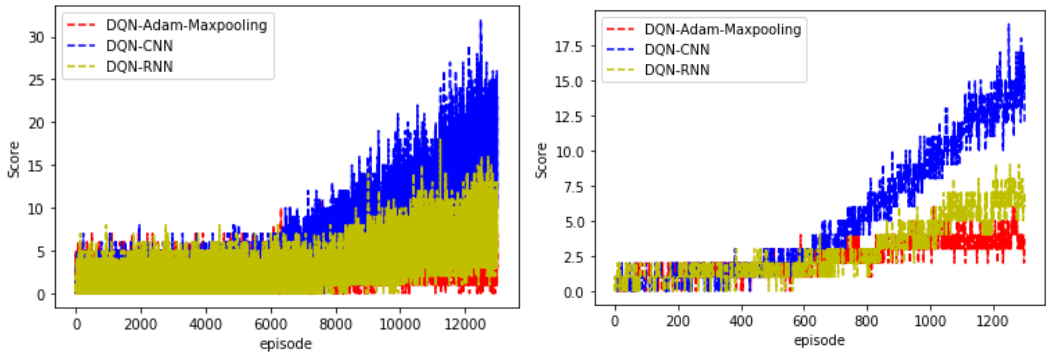
Learning Model	Maximum score
Neural Network	309
KNN	265
Ridge	56
XGBoost	197
LightGBM	287

Since the performance of agent fluctuates a lot, it is not easy to see the difference of the learning performance, in order to show it we averaged every 10 episode and that result is presented in Fig 6b. It is clearly seen that the original DQN settings performs better than DQN with different network structure and RNN models. Although RNN performs better than our proposed network model, still using CNNs with the structure proposed in (1) achieves the best results.

We observe that the original DQN model performs the best under given circumstances but we see that all the three models begin to converge at the same time but it is the original DQN model which then takes the lead. The RNN implementation in addition to CNN not performing better than DQN also took the longest time, which makes sense since it builds on the original DQN model. The possible reason for this worse performance can be in the way we designed our LSTM layer, since we only got to play with one dimensionality of neurons it could have been possible that for some other value of neurons the network performed better. Authors in (3) also concludes that RNN DQN do not always performs better than CNN model.

Our expectation with implementation of our model with max pooling was to replicate the results of the original model while also improving the training time by decreasing the number of variables involved. Our results show that even though the DQN with max pooling is the fastest in terms of training time, it is unable to perform as good as the original model, and instead it performs worst than the RNN model. One of the possible reasons for that can be that with pooling too much of information is lost to make the correct decision. Which ultimately results in slower training. The other reason could be that the Adam optimizer did not perform as well as the original RMSProp optimizer.

Under our experimentation it is clear that authors decision in (1) to stick with a three layered CNN network was better than adding pooling layers to it or using the RNN implementation in it.



(a) Raw simulation results with 13k episodes

(b) Average of last 10 episodes

Figure 6: Simulation results for breakout game comparing CNN and RNN neural network performance

5 Conclusion

We studied Q learning based RL algorithms on two atari games namely Cartpole and Breakout. In this project, we employed regression based function approximators (K nearest neighbors, Ridge, XGBoost and Light GBM based gradient boosted tree model) and nonlinear neural network approximators on

Cartpole game and image formatted state input structure with CNN and RNN neural networks on breakout game. In the cartpole game, we see that for linear function approximators, batch replay from experience replay memory does not perform well, therefore we change the setting as training the learners with whole memory. However we used batch sampling method on neural network models and this gives us very good results. The results indicates that, although LightGBM may reach the closest performance to the neural network based model, it is highly time consumed model, Hence, the optimum learning model even in simpler problems is neural network base RL. In the breakout game, we compared two configurations with the baseline models (1) CNN with max pooling and Adam optimizer and (2) LSTM model. Our results shows that our first configuration performs the worst while take shortest training time. The second model which uses LSTM also does not overer come the baseline CNN model. Our LSTM model also takes more training time than CNN.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[3] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in *2015 AAAI Fall Symposium Series*, 2015.

[4] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[5] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Advances in neural information processing systems*, 2017, pp. 3146–3154.

[6] Tatsuya, "Dqn in keras + tensorflow + openai gym," 2016. [Online]. Available: https://github.com/tokb23/dqn

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/