Deep Q-Learning for Atari River Raid

3Yes1No - *Yash Mehta, Yuetian Li, Yingfei Hong, Nange Li*

Introduction

In the past couple years, through deep reinforcement learning, artificial agents have achieved a human-level of performance and generality in playing games and solving tasks. With enormous amounts of training, they can even outperform humans. In the current project, we focused on training deep reinforcement learning models to play Atari 2000 River Raid. Specifically, we used Deep Q-Learning Networks (DQN). In 2013, the paper by Deepmind Technologies explored the method of using Deep-Q learning on seven Atari 2600 games and showed with DQN, the agent can learn control policies directly from high-dimensional sensory input [Ref 1].

In this project, we first started with a vanilla DQN in combination with a convolutional neural network. To improve the training accuracy and stability, we applied two strategies: fixed Q-targets and replay memory mechanism. Fixed Q-targets handle the oscillation in training due to shifting Q target values. Replay memory is to break the correlation between consecutive samples of experience in the environment that could lead to inefficient learning. River Raid is a complex game with 18 actions. Our goal was to train the agent to outperform the random model and to achieve human-level performance. With the optimization techniques and given enough training time, we hope the agent can outperform our team members.

Methodology

Preprocessing

There is no specific dataset to work on. Instead, each action step the gym emulator provides an observation image of size 210 pixels by 160 pixels with RGB channels, which is a fairly huge state space (an upper bound of $255^{210\times160\times3}$). For the input frames, we are

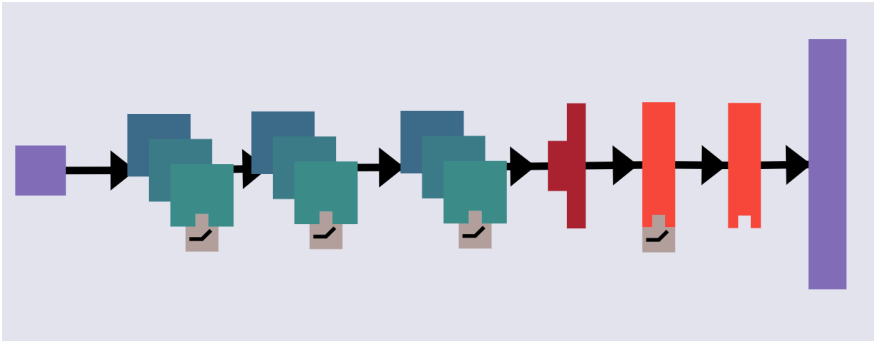
concerned about the layout of the track, the states (positions, moving trends) of the enemies and bullets, and the position of the fuel depots. As object attributes are not changing as the game goes on, converting the RGB image to grayscale would not cause a loss of information. We also resize and crop the frame to 84×84 pixels. Another critical step is k-skipping, which is suggested by the DeepMind paper. We concatenate k=4 consecutive frames for the model to predict one action, and repeat this action four times to get the next series of four observations. The intuition of this method is to capture the motion of objects, as well as speed up the training process.

Models

Vanilla DQN

In our Q network, with images as input, we add 3 convolutional layers with ReLu activation and one dense layer with ReLu activation sequentially before the final dense layer that has the output size same as the number of actions. The model architecture is described in the following table.

	Channels	Filter	Stride	Activation	#parameters
Convolutional layer 1	32	12	4	ReLU	18464
Convolutional layer 2	64	6	2	ReLU	73792
Convolutional layer 3	64	4	1	ReLU	65600
	Hidden size				
Dense layer1	640			ReLU	656000
Dense layer2 (output)	18 (Number of actions)			None	11538



Fixed Q Target

For a vanilla DQN, we estimate the target Q value by summing the reward of taking the action at the current state and the discounted highest Q value for the next state.

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

where s is the current state, s' is the next state after taking the action a, r is the reward for taking the action a, γ represents the discounted factor, and Q is the Q network. In this case, we use the same parameters to calculate the Q value and the target.

The fixed Q target estimates the target value by a separate neural network that has the same architecture as the Q Network. And every T step, we will update the target network with the parameters of the Q Network.

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

where $\widehat{Q}$ is the target network and every T steps: w' = w. Introducing the separate neural network will cause more stable learning because our target value will stay the same for T time steps.

Memory mechanism

The memory mechanism is the technique that uses past experiences to update the Q network in addition to the update in each time step. Each experience is a tuple that contains the current state, the action based on the max Q value, the reward, and the next state after taking the action. Given a fixed number of past experiences, we will randomly sample a small batch of experiences from the memory. For each experience in each batch, we will first feed the current state into the Q network and find the Q value for the certain action in the current state. And the target Q value will be estimated by the sum of the reward for taking the certain action in the current state and the discounted maximum Q value of the next state by the target network. Therefore, we will be able to calculate the loss between the target Q value and the current Q value and then train the Q network. The memory mechanism helps the Q network learn from the most recent experiences as well as the past experiences and makes the DQN train more stable and efficient.

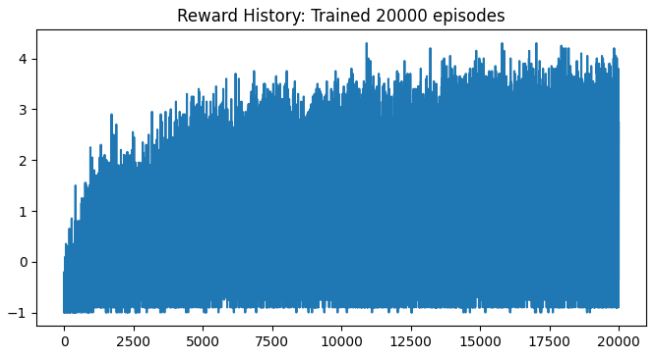
$$error \;= r \;+\; \gamma max_{a} \widehat{Q}(s',a) \;-\; Q(s,a)$$

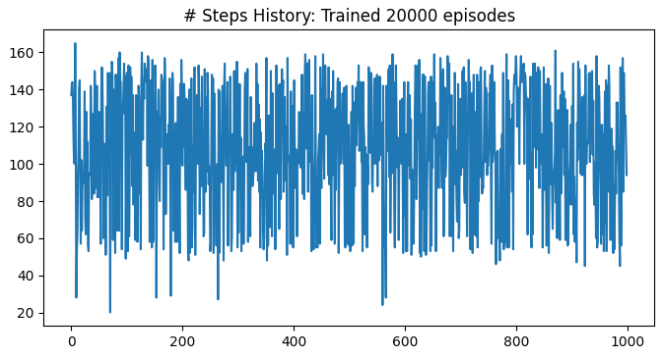
Results

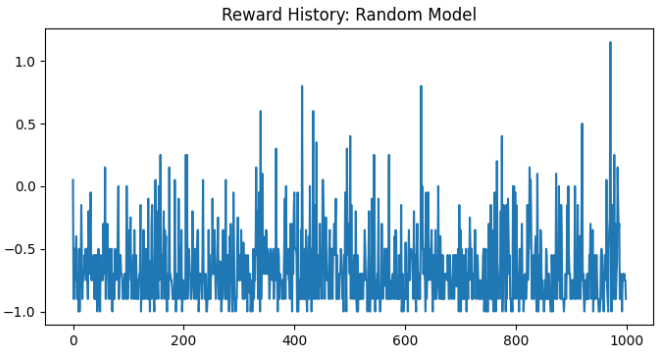
To compare the trained agent with the random model, we used average rewards, rewards for the last 50 episodes and steps for the last 50 episodes as evaluation metrics. The following table shows that the trained agent based on the DQN model with fixed Q-target and the memory mechanism after 20000 episodes outperformed the random model greatly.

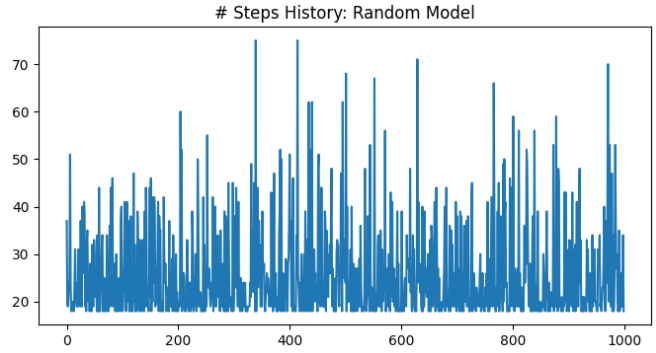
	Random Model	Trained Agent
Average rewards	-0.660	0.932
last 50 episodes' reward	-0.615	1.094
last 50 episode' #steps	26.6	105.6

The following images show that with an increase in the training episodes, our trained agent earned more rewards and made further in the game. However, the figures also showed that the training process of our agent is not stable with large fluctuations. The multi-histograms further confirmed that our trained agent showed distinguishably better performance compared to the random model.

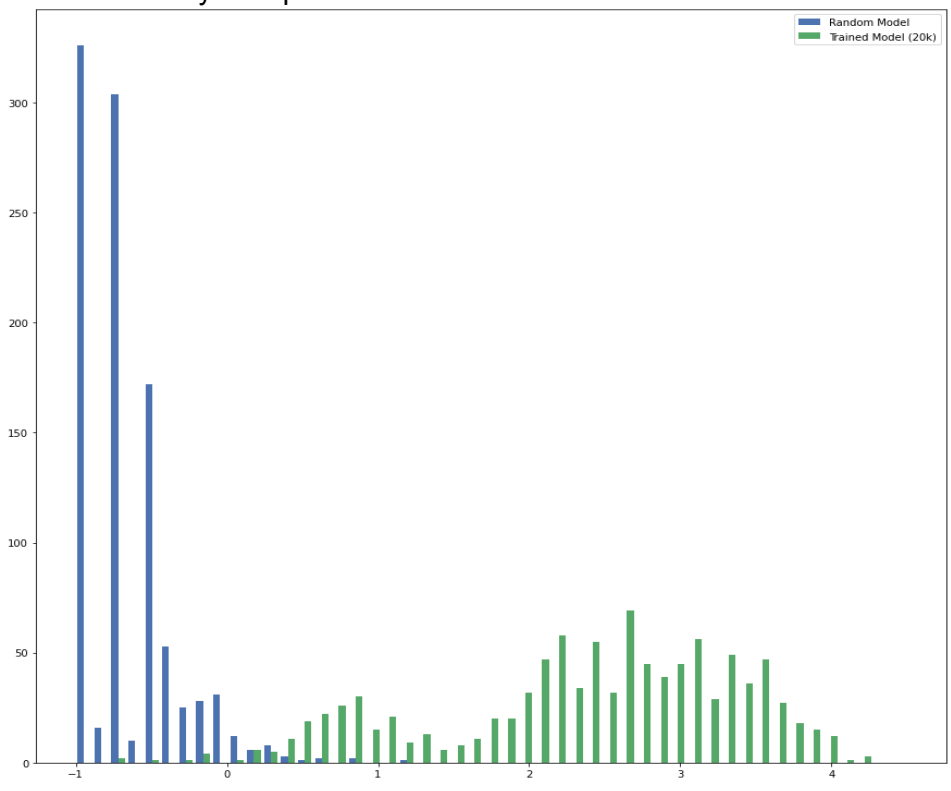




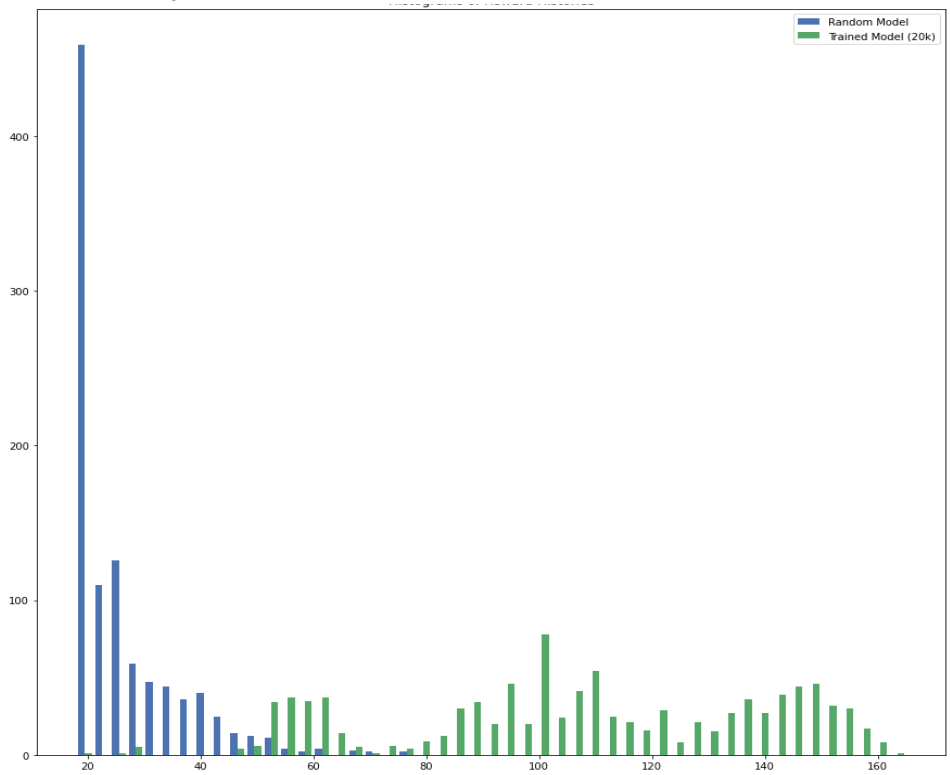




Rewards History Comparison between Trained Model and Random Model



Step History Comparison between Trained Model and Random Model



Challenges

Environment setup

This being a reinforcement learning problem, we were aware that we would need to run several iterations and that it would take significant time and resources to get a signal that a given model was actually learning the Q-function. With this in mind, we made a decision early on to set up a Deep Learning Virtual Machine on GCP. Setting this up was a somewhat elaborate process, requiring the creation of a project, network, VM, etc.

Setting up the Gym environment in a manner that allows us to replicate it on our local computers in a reliable way is also quite challenging. We used the Arcade Learning Environment, which is the officially supported version, and figured out Python and package versions that were consistent with what was available on the GCP VM.

Difficulty in wrapping

Another challenge was modifying the out-of-the-box environment in a way that would make it easy to train models. For example, as an observation of the current state, we needed to consider not just the current observation, which is essentially a screenshot of the game, but also the past few frames, which would allow us to infer things like velocity of objects in the game. For this reason, we decided to build our own individual wrappers on the Gym environment to do the following:

- Preprocess observations: Concatenate observations, convert screenshots to grayscale, and normalize; Crop and downsample to 84*84 for each screenshot;
- Fire Reset environment: starting the River Raid game requires pressing the FIRE button;
- Custom Reward: The default rewards of River Raid are all positive and very large numbers. Because of the absence of negative rewards, the agent did not receive feedback for a long time and did not understand that it needed to stay alive to play the game successfully. We added a large negative death reward that gave it quicker feedback to avoid death. We also normalized the rewards so that gradients were more consistent and allowed the agent to learn in a more stable way.

Unstable training (Finicky with hyperparameters)

When we first had the proto model ready for training, we found it tricky to initialize the hyperparameters. For a long period of time, our running results showed that the model was not learning anything suggested by that the rewards curve decreased, and the DQN even had a worse performance than a random model. It could be due to improper hyperparameters, or problems with model architectures. Since River Raid is quite complicated compared to the rest of atari games, we decided to introduce more CNN channels and layers to capture game states.

We also recognized a problem at the hyperparameters for epsilon-greedy. If the Q-network was not trained properly, as epsilon value decayed exponentially over episodes, actions were chosen greedily from the useless Q-network in most time. The decreasing manner for epsilon value should also be adjusted when the model is trained more episodes on GCP, in case of decaying too fast before the model tries different actions. In the end, we chose this updating strategy:

$$\epsilon = max(0.1,\ 0.9999\epsilon)$$

Additionally, we spent a lot of time working on the steep vacillation of the rewards curve to overcome during the model setup and training. As local training can only support around 1000 rounds of games, which may be inadequate for the model to start to learn something useful. But we were not certain about the cause or what to adjust without a full run for over 10k episodes. The decreasing trend still arises even when the model works in the long term.

Reflection & Takeaway

The final model performs a little better than a human beginner in the game. One of our team members played online RiverRaid on this website a few times, and the best score with 1 life (4 lives for each game, but we adopt only one life for model rewards) is around 1500. A human expert on YouTube can get a score of 10k - 100k. Given these human records, we believe we already meet the base and target goals, and partially reach our stretch goal of achieving human level performance. However, as we evaluated the training process, the jet agent failed to distinguish the fuel depots from other enemies (helicopters,

tanks), but just shooting them with 80 points each. The agent should learn to refuel instead of firing for points all the time, otherwise it would run out of fuel. However, our model has not reached that far. To improve, we should extract the information from the energy bar and add a term to the reward function.

More experiments can be extended. For instance, we can tune k values in the k-skipping method. A larger k value provides the model with more information to predict an action, but as the action is repeated k times and it might not be the optimal strategy. So there is a tradeoff between faster training of large k and fine-grained action decision of small k. Tests on how the training rewards interact with different k values may provide more insights.

Given that we were building a model for a specific game, I think we should have spent some more time upfront in understanding the game mechanics, what it takes to get a high score in the game, what the elements of the game screen are etc. As things stood, when we implemented our models, we were taking a more trial-and-error approach to making model improvements.

RiverRaid is a fairly complex game, and I think we could have implemented a more complex model to play the game. However, we didn't have the time to work on it. It may also have been interesting to consider different approaches, like supplying the states to the model. Eg. instead of passing the last 4 screenshots, we could have passed the most recent screenshots, and the differences between the previous screenshots, i.e. $state_t$ = (screenshot_t, $screenshot_t$ - $screenshot_{t-1}$, $screenshot_{t-1}$ - $screenshot_{t-2,}$ $screenshot_{t-2}$ - $screenshot_{t-3}$) However, this would not be equivalent to how a human user sees the game.

Recalled from the Vanilla DQN, we use Q(s,a) to represent how good it is to be at state s and taking the action a. So the Q(s,a) is actually the combination of the value of being at state s and the advantage of taking action at the state. And this is the idea of DDQN. The DDQN uses the Q network to estimate the advantage of taking action at the state and another V network to estimate the state value. And the ultimate Q value will be the sum of the state value V(s) and the centered advantage (Q(s,a) - the average advantages of all possible actions). This is also a good way to reduce the instability of the DQN training by calculating the state value without learning the effect of actions at a given state. And it is really useful in our case because for most of the actions, like left or right or

accelerate, they do not matter unless the plane hits the wall or hits the moving enemy. And in most states, the choice of actions has no effect on what happens. However, due to the time limit, although we have implemented the dual DQN model, we are not able to train this model and also the combination of this model with other mechanisms like memory mechanism.

In summary, the current project is successful in training the agent to learn control policies based on high-dimensional sensory input through deep Q-learning networks. The additional techniques including k-skipping method, fixed Q-targets, and replay memory mechanism were incorporated into input preprocessing and model building to achieve a more stable and efficient training. We have achieved our base goal and target goal to achieve similar levels or even outperform our team members. Future work could implement additional techniques such as Dual DQN and tuning k values in k-skipping method to improve agent learning.

References

[1] Play Atari with Reinforcement Learning https://arxiv.org/abs/1312.5602

[2] Dueling DQN & Fixed Q-target
https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/

[3] Memory mechanism
https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c