ReDER: Related Domain Experience Replay in Deep Reinforcement Learning

Richard Tang (rtang26), Kenta Yoshii (kyoshii), Raymond Dai (rdai4), & Akash Singirikonda (asingir5)
Brown University
Providence, RI 02912, USA
{richard_tang, kenta_yoshii, raymond_dai, akash_singirikonda}@brown.edu

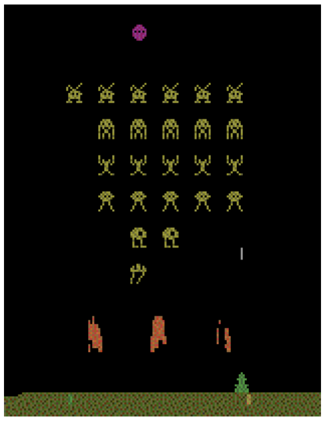
Figure 1: Space Invaders, a classic Atari game

1 Introduction

Recent research in deep reinforcement learning has yielded great advancements, with successes from simple games like Backgammon (https://dl.acm.org/doi/10.1145/203330.203343) and Atari (https://arxiv.org/pdf/1312.5602.pdf) to incredibly complex tasks such as beating Lee Sedol, a 9-dan professional at Go (https://www.nature.com/articles/nature16961) and multi-step robotics tasks (https://arxiv.org/pdf/2104.07749.pdf). However, deep learning models already take significant resources to train, and reinforcement learning agents worsen this issue by an order of magnitude. For instance, AlphaGo trained over 72 million matches in 72 hours, using over 6000 TPUs for self-play simulation, and 64 GPUs and 19 CPUs for parameter updates; in total, training AlphaGo cost $35 million. In a model-free environment, DeepMind's Agent57 (https://arxiv.org/abs/2003.13350) required 10 billion episodic steps to train.

Clearly, deep reinforcement learning agents are incredibly computationally expensive to train. In this project, we attempt to alleviate this cost. From experience, humans are relatively adept at picking up new skills; for instance, proficiency in one first-person shooter game (e.g. Valorant) would likely transfer to other games with similar play styles and mechanics, such as CS:GO or Apex. Intuitively, this aligns with our notion of *shared experience*, or experience transfer; skillsets applicable to one area should theoretically transfer to another similar area, greatly lessening the initial learning and startup overhead required.

To that end, we first aim to develop a deep reinforcement learning agent—based on current reinforcement learning literature—capable of playing Atari games, then attempt to employ a *shared experience buffer* while training to simultaneously learn Atari games with similar styles; here we choose top-down shooter games, specifically Space Invaders and Demon Attack.

2 BACKGROUND

In classic reinforcement learning (RL)[1], an **agent** interacts with an **environment**, and attempts to maximize cumulative **reward**. Specifically, given a state space S of all valid states in an environment, and an action space A of all valid actions, an agent adopts a **policy**

$$\pi : S \longrightarrow A, \pi(s_t) = a_t$$

that picks an available action $a \in A$ from a current state $s \in S$. In general, state and action spaces may be either discrete or continuous spaces; moreover, taking an action a_t in a state s_t can either deterministically or stochastically **transition** into a state s_{t+1}. With Atari games, both the state and action spaces are discrete, and the environment transitions is deterministic. A **reward** value is assigned to these transitions; that is, a function

$$R : S \times A \times S \longrightarrow \mathbb{R},\ R(s_t, a_t, s_{t+1}) = r_t$$

determines the reward of a certain transition from one state into another.

The central goal of reinforcement learning is to maximize cumulative reward over an entire episode in the environment. Because RL agents do not have access to unlimited future rewards, to update and train our model we turn to the **finite discounted future reward** function over a time interval T:

$$R_T(\tau) = \sum_{t'=t}^{T} \gamma^{t-t'} r_t,$$

where $\tau = (s_{t'}, a_{t'}, s_{t'+1}, a_{t'} + 1, \ldots)$ is a trajectory of state-action pairs, and $\gamma \in (0, 1)$ is a discount factor on future rewards. Intuitively, we wish to prioritize current rewards over future rewards; mathematically, we need reward values to converge. Reward factors into the **value** of a state, computed by a function

$$V^\pi : S \longrightarrow \mathbb{R}, V^\pi(s_t) = v_t.$$

Two functions—the value function and the Q function—are particularly important in optimizing future reward, and play a central role in reinforcement learning algorithms. They are given by

$$\begin{aligned} V^\pi(s) &= E_{\tau\sim\pi}[R(\tau) \mid s_0 = s] \\ Q^\pi(s, a) &= E_{\tau\sim\pi}[R(\tau) \mid s_0 = s, a_0 = a]. \end{aligned}$$

The value function determines the expected reward of starting in a state s following a certain policy π, while the Q function determines the expected reward of starting in a state s, taking an action a, then following a certain policy π. Their respective **Bellman equations** provide more insight, and format in a more accessible manner for code implementation:

$$\begin{aligned} V^\pi(s) &= E_{a\sim\pi, P(s'|s,a)}[R(s, \cdot, \cdot) + \gamma V^\pi(\cdot)] \\ &= \max_a \sum_{s' \in S} P(s' \mid s, a) E[R(s, a, s') + \gamma V^\pi(s')] \\ Q^\pi(s) &= E_{P(s'|s,a)}[R(s, a, \cdot) + \gamma V^\pi(\cdot)] \\ &= \sum_{s' \in S} P(s' \mid s, a)[R(s, a, s') + \gamma V^\pi(s')]. \end{aligned}$$

Determining these functions greatly facilitate RL agents in solving an environment; given an optimal Q function Q^*, an optimal policy π^* would simply select the action with the highest Q value in a state s; that is,

$$\pi^*(s) = \max_a Q^*(s, a).$$

Unfortunately, with many scenarios our environments become too unwieldy to provide exact value and Q functions; thus, modern reinforcement learning attempts to use function approximators to estimate these values. As deep learning models become increasingly sophisticated, applications to reinforcement learning become promising. With the Q function, a neural network (a Q-network)

[1]OpenAI's "Spinning Up" Reinforcement tutorial provides a great introduction to RL; we adopted parts of their tutorial in this section. Check it out at `https://spinningup.openai.com/en/latest/spinningup/rl_intro.html`

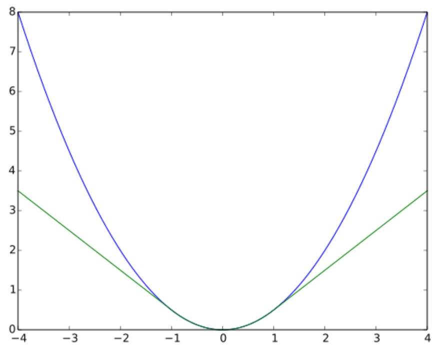

Figure 2: Huber Loss vs. MSE, from Wikipedia (Huber Loss)

may be used as a non-linear approximator, i.e. $Q(s, a; \theta) \approx Q^*(s, a)$. To train a Q-network, we want to minimize distance from the target Q-function:

$$L_t(\theta_t) = E[(y_t - Q(s, a; \theta_t))^2],$$

where $y_t = E[r_t + \gamma \max_{a'} Q(s', a'; \theta_{t+1})]$ is the target Q function. This is the **temporal-difference loss**; that is, the loss between our current Q network and the Q network of the next step, which functions as our target Q network. Because we do not have a ground truth optimal Q function, we instead train by constantly learning and updating our Q-net with "future" Q-values. From this, one can see how training RL models has extremely high variance, and is susceptible to model collapse.

To rememdy this, a few modifications are made. To clip error values and reduce variance, we employ the Huber loss function instead of MSE loss. That is,

$$L_\delta(y_{true}, y_{pred}) = \begin{cases} \frac{1}{2}(y_{true} - y_{pred})^2, & \text{for } |y_{true} - y_{pred}| \le \delta \\ \delta(|y_{true} - y_{pred}| - \frac{1}{2}\delta), & \text{otherwise} \end{cases}.$$

Intuitively, with larger loss values, we inherently clip them such that no drastic changes are made to the model (see Figure 2 for a visualization of its functionality).

Additionally, Minh et. al. 2015 (`https://www.nature.com/articles/nature14236.pdf`) provide a method of stabilizing variance in the training; because a naive DQN trains solely on the next step's Q-network, variance is extremely high, and the model is susceptible to local minima traps. To reduce this, a stabilizing $\hat{Q}$ network is introduced to prevent the model from over-deviating from previous training weights. This $\hat{Q}$ network will share the weights as the original Q network, except it is not trained; $\hat{Q}$ will only update its values to the original Q network periodically instead of every step. Thus, its function as an anchor prevents the training from fluctuating too far from previous steps. Our loss function then becomes

$$L_t(\theta_t) = E[(y_t - Q(s, a; \theta_t))^2],$$

where $y_t = E[r_t + \gamma \hat{Q}(s', \max_{a'}(s', a'; \theta_{t+1}); \theta_{\hat{Q}})]$. This architecture of a second stabilizing $\hat{Q}$ target network is called a **double deep Q-network**, or **DDQN**.

Finally, Wang et. al. 2016 (`https://arxiv.org/pdf/1511.06581.pdf`) provide an improvement to the naive Q-network. Instead of learning directly for a Q-function approximator, the Q-network is split up into approximating functions $V(s)$ and $A(s)$, or the value function and **advantage** function respectively. The advantage function determines the *relative* value of taking a certain action in a state, rather than the absolute value:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s).$$

This reduces variance by comparatively determining the value of a state, rather than absolutely predicting it; thus, in extreme states a large Q value is avoided. The "dueling" architectures of the value and advantage function can theoretically learn valuable (and not valuable) states without having to learn the effect of each action in each state, as the naive Q-network is forced to; this comes as a result of the model learning each separate network, and indirectly predicting the Q-values. The Q-function becomes

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a' \in A} A(s,a') \right).$$

This architecture of splitting up the Q-network into value V and advantage A streams is called a **dueling deep Q-network**, or **Dueling DQN**. Combined with the DDQN, the final model is a **Dueling DDQN** (Figure 3).

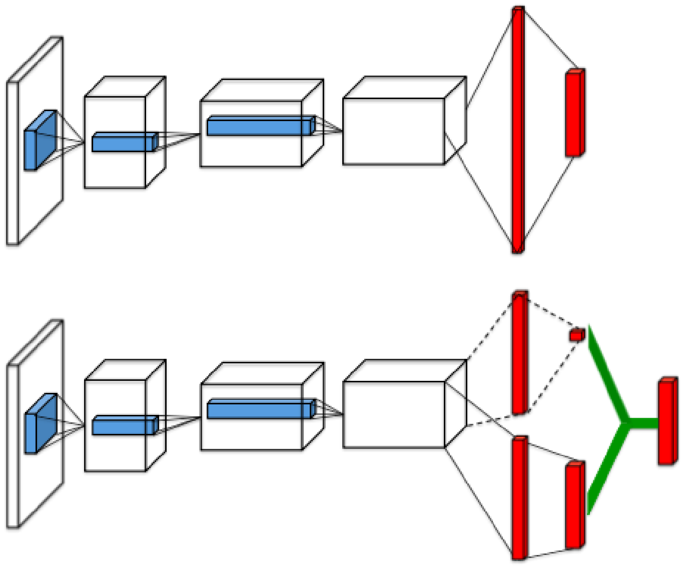

Figure 3: Dueling vs. Naive DQN Architecture

3 Methodology

Our baseline Dueling DDQN agent follows a similar structure to the Mnih et al. 2015 network. An input image is passed through four convolutional layers: the first has 32 8×8 filters with stride 4, the second has 64 4×4 filters with stride 2, and the third has 64 3×3 filters with stride 1. After some experimentation, we decided to add a fourth convolutional layer with 1024 7×7 filters with stride 1; this provided more input weights for the value and advantage streams to work with. The output of the convolutional layers are then split up into 2, one for each stream, and passed through a fully-connected dense layer; the value network outputs a single value, while the advantage network outputs an advantage value for each action in the action space.

Taking inspiration from stochastic gradient descent, the temporal-difference loss function is applied in batches, rather than for each individual step. To ensure a more independent and identically distributed batch of past experiences, we maintain an **experience buffer** of previous experiences, from which we sample a batch. We then feed the batch into the loss function described above, then apply the gradients on our Q-network.

This provides the baseline RL agent to train on Atari games; while some modifications were made, this is mostly in line with methods given in classic Deep RL papers. State representation of shared experiences is far trickier. At the heart of our learning process is one fundamental question: how do we convert an experience from one game to another? We intuitively understand that Space Invaders and Demon Attack have similar mechanics, that learning to dodge bullets in one game will help you dodge bullets in another, and hitting enemies in both boosts your scores. However, machines have no such understanding of states, and any architecture used to interpret the screen in one game will probably fail in the other because of the stark visual differences, animations that play, etc. As such, we want to develop a state representation such that we can somehow map an experience in one game to an experience in another. We have attempted several different methods to encourage shared learning between the two games.

3.1 Direct Learning

By far the simplest way to attempt to learn from either experience is to ignore state representations entirely and train the DQN on experiences as if they came from the same game. This would involve running two games at once and having a shared experience buffer such that the model has a chance of learning from either game. This theoretically encourages the state interpreters—our Dueling DDQN—to focus on the similarities of the two games (movement, shooting, enemy targeting, etc) and forces it to learn "core mechanics" first. In theory, this would make early training more variant in the beginning but converge faster in both games as neither games get caught up in small details in each game and masters core mechanics. However, this naive approach still leaves a lot to be desired: with larger training sets and across more models, more variance is introduced into the state representations, and the model may struggle to find an underlying similarity with such differences. RL agents are already highly susceptible to local minima, and increasing variance only increases the possibility of catastrophic interference and model collapse. Moreover, although we preprocess the inputs (described in the Experiments section), no flexibility is provided in general; altering state shapes would completely negate the Dueling DDQN's convolutional architecture.

3.2 State Translation

Another issue with direct mapping results from fundamental differences in games; learning on one game (e.g. Demon Attack), with different animations and environment structures, may interfere with another game (e.g. Space Invaders) and the model's interpretation of potent features. As such, another intuition is to somehow map one game screen to another. We assume that taking the same series of actions in one game (ie moving left, right, shooting, and pausing) has a direct "translation" in the other game where the same series of actions were taken (Figure 4).

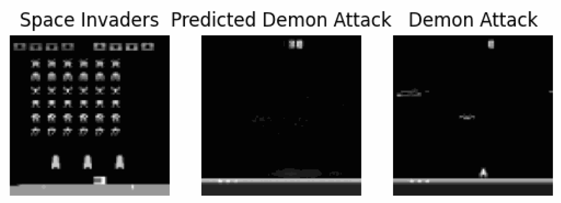

Figure 4: State Translation Mapping

However, in practice, machines are not very good at learning this weak translation. A network trained to map images from one game to the other tends to focus on static, unchanging details (ie the floors, scores, decorative borders, etc). Any attempt for the network to learn something more interesting, like the ships' movements, will immediately be squashed by another frame where the ship is in another region. As such, this model converges to a local minimum where it is very good at capturing the static, unmoving objects but cannot map enemies, ships, or moving objects. Most of either screen is taken up by negative space, and since any loss function would have to give some form of scalar value to describe differences between two images, a model focusing just on static objects can have reasonably low loss for every frame. This is a lot easier to converge to than the complex model that tries to learn the nuances of ship movement. Simply put, one state does not directly map to another, even if the steps leading up to those two states were identical.

3.3 Latent Spaces

Another core idea is to develop a state latent space to learn from rather than learning from screens. Ideally, some model will encode information about both games to a vector of a common dimension. This would remove our screen size problem, and games would be training on a much more simple latent vector rather than complicated screens. More importantly, training a strong latent space allows our model to learn only from the principal components of the game, which lessens the likelihood that the model gets distracted by minor details and should stabilize training. As such, our novel idea was to learn a "double autoencoder" - a model capable of autoencoding both space invaders and demon attack. The model would alternate training between frames of demon attack and space invaders, tugging the gradient long until the model converges to some latent space capable of modelling both games. Using these latent spaces allows both models to learn on the same "domain", making it easier to map states to each other. However, we have so far been preoccupied with states. Even if our autoencoder is perfect, who is to say that the latent space for space invaders matches that of demon attack? Maybe one has the first weight devoted to detecting ship movement while the other uses it to look at bullets. Learning from the autoencoder here could mean that a useful experience for one game could be mapping two completely irrelevant frames for another. We realize that it's not necessarily realistic states we are pursuing but realistic experiences.

3.4 Delta Mapping

In order to focus on the changes in experiences, we apply the state "translation" model to focus on changes between states rather than states themselves. For every action, we calculate a delta state function that subtracts the resulting state from the previous state, which should highlight the changes in states rather than focusing ons tactic objects. Our results, however, were once again less than promising. Our distribution is once again relatively static, this time having streaks where ships frequently appear and hover around (Figure 5).

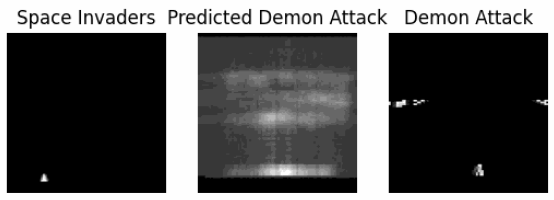

Figure 5: Delta Mapping of States

This is a result of something we are yet to mention about these models: overfitting for early episodes. Most of the initial training for these methods come before we have a trained Q network. As such, we are forced to train on random walks, which rarely get far and have very low step counts. This means our models are optimized to early states, like having the ship in the middle and paying attention to the first few enemies. Still, while this network also yielded negative results, it did tell us a lot about the early stages of the game, and provide us a way to interpret the model during training: even if we add a time value to encourage movement, we see that the local minima model converges to maps where enemies and the ships typically are.

3.5 Combination Methods

Our final method attempts to combine the lessons we learned about state representations. Instead of learning solely states, we want a method that is able to transform experiences from one game to experiences in another. Having some form of shared "experience latent space" could give us the ideal situation: a space that is able to codify experiences such that any model can learn from and apply them. As such, we propose the following architecture (Figure 6).

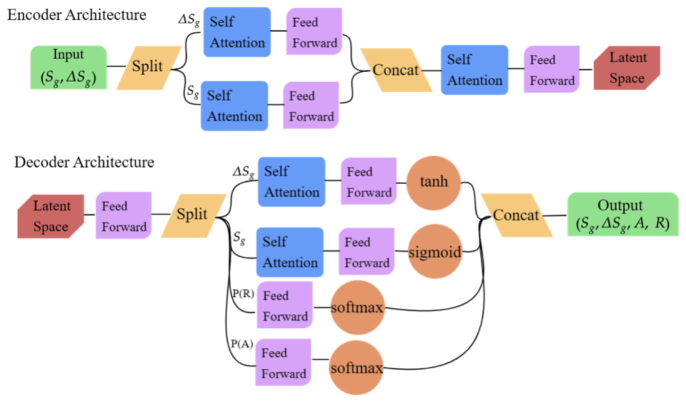

Figure 6: Combination of our State Representation Models

There are a few conditions that we want this function to satisfy. First, we want the latent space to be usable enough to extract experiences from it, so we would want any encoded experience to be returned by the decoder. Second, we wish to glean the most important information about these experiences: the rewards and actions. We therefore note that any encoded experience must return the corresponding action and reward, no matter what decoder is used. This provides the "common thread" between the translations that was lacking in the other latent space implementation: an encoded experience should be decoded to a similar valid experience. Finally, we want visual similarity: that is, decoded outputs should look like actual states and changes in states. We achieve this through the use of a discriminator, which tries to determine which experiences are real and fake (Figure 7).

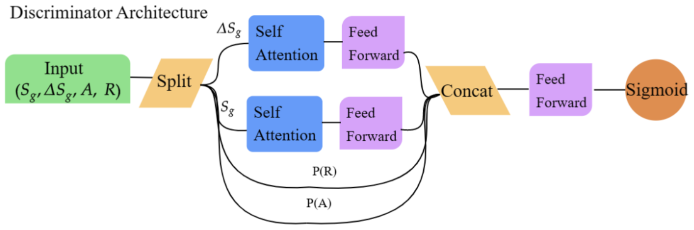

Figure 7: Discriminator aspect of the architecture

As such, we note that our final structure has something similar both to autoencoders and generative adversarial networks (Figure 8):

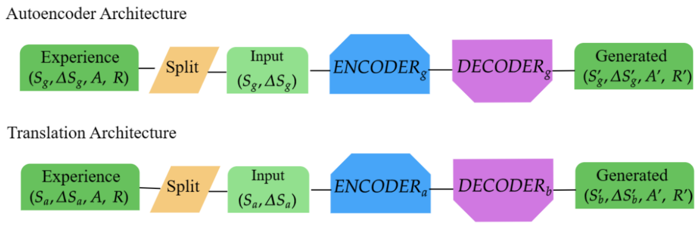
Autoencoder Architecture
Experience $(S_g, \Delta S_g, A, R)$
Split
Input $(S_g, \Delta S_g)$
$ENCODER_g$
$DECODER_g$
Generated $(S_g', \Delta S_g', A', R')$
Translation Architecture
Experience $(S_a, \Delta S_a, A, R)$
Split
Input $(S_a, \Delta S_a)$
$ENCODER_a$
$DECODER_b$
Generated $(S_b', \Delta S_b', A', R')$

Figure 8: Variational Auto-encoder aspect

Since the latent spaces are common to all games, it provides a proper replacement to raw input screens as our state representation. Unfortunately, the model is a bit too involved to train on our current hardware; it takes a minute to train on a singular step of the model given all the gradients that need to be optimized. However, we have the full model coded, although it is questionable whether the model will see any gains over just normal training. The entire translation process might lose some of the "core gameplay" advantage that our first method promised, and it may end up generating the same experiences over and over, which will not improve training.

4 Experiments

Before training our models, we first apply some important preprocessing steps on the Atari environments[2]. To facilitate training, we skip every fourth frame; previous RL literature (e.g. Mnih et. al. 2015) found no detrimental effects on training, while speeding up the process. Then, to reduce the dimensionality of the image and lessen redundancy in weights, each environment image is first converted from RGB to grayscale, then downsampled from 216×160 to 84×84. Finally, an optional state-stacking mechanism is provided, where four consecutive frames are stacked on top of each other. This provides the model with a way to learn the current trajectory of the environment's state (Figure 9).

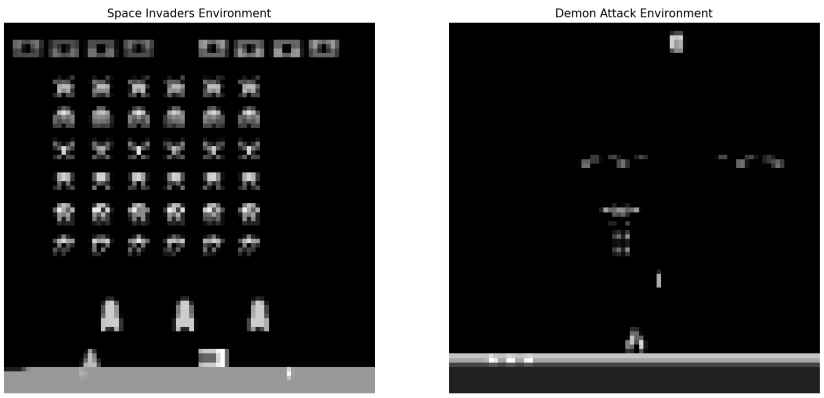

Figure 9: Post-preprocessing Atari Images

Following pre-processing, we train the model. To ensure a proper exploration-exploitation trade-off and lower the possibility of model collapse through optimizing for a local minima, we employ an ε-greedy policy, where we explore a random action with probability ε, and follow the max Q-value output from our model with probability $1 - \varepsilon$. The ε values are linearly annealed, with the different values listed below. When calculating the loss function and applying gradients, we deviate slightly from the strategy proposed in Mnih et. al. 2015; instead of backpropagating on every step, we instead learn every few (listed below) steps; we found that this lowers potential for model collapse in the beginning, when our model may be optimizing for a random (and wrong) target $\hat{Q}$ network. Additionally, while the Mnih architecture updates the target network every 10000 steps, we choose a lower number (listed below) to avoid local minima. We choose an Adam optimizer over RMSProp optimizer; at the time of the original Mnih architecture, the Adam optimizer had not yet been released, and generally for deep learning models the Adam optimizer provides a more robust optimization function. The complete hyperparameters of our models are listed below (see Table 1).

The experience buffers are initialized by randomly sampling `BUFFER_SIZE` states from the environments; for the shared model, the buffer size is doubled, and states are sampled from both environments. Due to computational limits, we only trained our models over 500000 steps, with evaluation benchmarking every 10000 steps. With the single baseline models, we train using one OpenAI gym environment, either Space Invaders or Demon Attack. With the shared model, we train using two

[2]Many of these techniques were first introduced and provided by OpenAI and DeepMind. The helper functionality was primarily adopted from OpenAI's baseline preprocessing methods; see `https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py`

Table 1: Hyperparameters

	Base-s1	Base-s2 / Shared
Learning rate	0.00025	0.00035
ε-start	0.95	0.99
ε-end	0.05	0.02
ε-steps	500000	500000
γ	0.99	0.995
Num episodes	1000000	1000000
Buffer size	10000	20000
Batch size	16	16
Learning frequency	20	10
Target update	2000	2500
Stacked frames	1	4

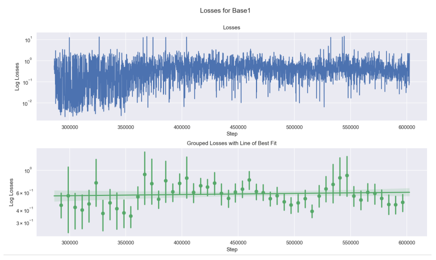

Figure 10: `base-s1` Losses

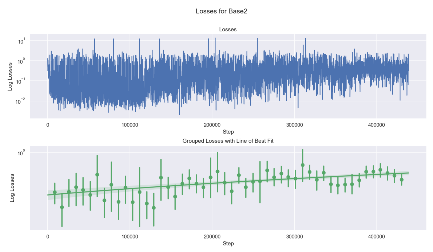

Figure 11: `base-s2` Losses

OpenAI gym environments, Space Invaders and Demon Attack; to facilitate the double environment nature, the model architecture alternates between the two environments, sampling stacked frames one at a time. When training, we lowered the batch size from the recommended 128 to 16, due to more computational limits imposed during training.

5 Results

Overall, due to the limits imposed by time constraints and computational power, we were only able to train our model for 600000 steps, significantly lower than the recommended 4000000 steps (a common benchmark across RL literature to successfully beat one level of Space Invaders); as a result, our graphs are less than optimal. However, fascinating (and promising!) results still appeared.

5.1 Dueling DDQN Metrics

We start by analyzing the Dueling DDQN model's performance on Space Invaders. The key difference between `base-s1` and `base-s2` lie in frame stacking; `base-s1` does not stack frames, while `base-s2` stacks four frames on top of each other. Intuitively, we would suspect that `base-s2` performs better, as the convolutional model learns consecutive state transitions. Indeed, this is somewhat corroborated by the loss, Q-values, and rewards.

For both loss values (Figures 10 and 11), they appear relatively stagnant, with `base-s2` showing slight increases. While this doesn't fit our usual intuition of losses, the graphs do appear sensible given the consideration of RL and our lower training steps. With temporal-difference loss, we compute not the difference between a ground-truth value and our predicted value, but rather the difference between our current predicted value and the predicted value of the next state. As a result, until our model is well trained, the values of loss should remain relatively stagnant; especially given the number of steps needed (4000000) versus the number of steps taken (600000), as well as the high complexity of Space Invaders as a whole, it would make sense that the model still has a lot to learn (Note: the losses for the `base-s1` model start at 300000 rather than 0, since we unfortunately did not set up metrics logging until halfway through training).

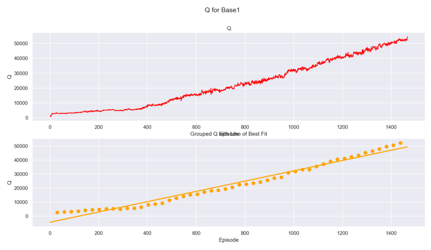

Figure 12: `base-s1` Q values

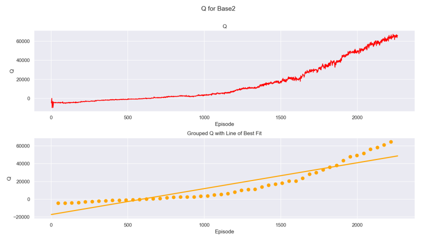

Figure 13: `base-s2` Q values

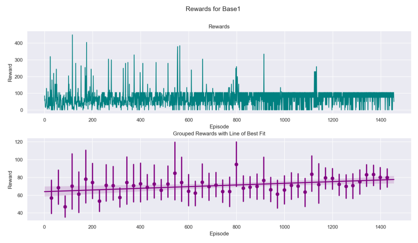

Figure 14: `base-s1` Rewards

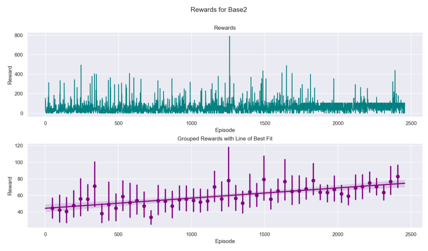

Figure 15: `base-s2` Rewards

To verify that learning is actually occurring, we inspect the Q-values of the models over time (Figures 12 and 13).

Here we see rather promising results. We evaluate the sum of the Q-values over a set collection of states over time, and see how the model values each state. As such, our intuition suggests that over time, as the model learns the benefits of entering each state, the Q-values in the states should increase. Our models demonstrate this feature; over time, we see a steady increase in the sum of the Q-values. Interestingly, potentially due to the robustness modifications provided by the Dueling Double DQN structure and the modified loss function, our models appear to successfully avoid becoming stuck in local minima during training. In `base-s2`'s Q-values, even though they initially decrease as the model explores detrimental states (note the small dip at the beginning), the model recovers and manages to properly evaluate Q-values.

Finally, we see the results over time of each model (Figures 14 and 15).

Here, although progress is slow, we can see a clear trend. Over time, the models gradually learn trends within the games, and how to utilize certain aspects. A qualitative analysis here provides better insights into the model's progress. At the start of training, inspection of the model's behavior is similar to what we expect (see GIFs in action here: `https://imgur.com/a/m6fDYD0`). The model generally gravitates towards the left shield of the environment state, as that has provided the best temporary rewards as the model gradually explores the state space. However, the model has no insights on the importance of prioritizing the Mothership, the benefits of exploration away from the left side, and general dodging of the shots; it is truly just a random model.

However, over time the model gradually grows its knowledge of the state space, and learns certain important features (see GIFs in action here: `https://imgur.com/a/gzdgcw3`). First, the model starts to explore elsewhere; once the first shield is gone, there is no point in hovering around there, so the model grows the ability to explore other aspects of the field. Indeed, in the last example, the model moves all the way to the right-most side. More importantly, the model learns the importance of dodging shots; although not overly effective and sometimes sporadically getting hit, the model generally senses the negative effects of the shots, and makes an effort to dodge them. Perhaps most impressively, the model learns the importance of prioritizing the Mothership. Whenever a Mothership appears, the model makes a concerted effort to move toward the Mothership; while not always successful (sometimes, the model runs directly into shots), the model understands the reward benefits, and sometimes impressively is able to hit it (not once, but twice in the last example).

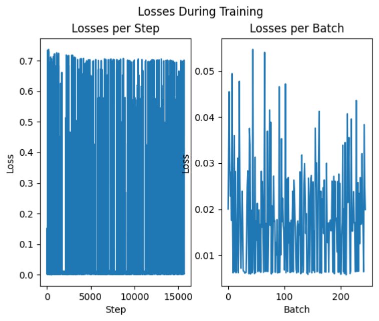

Figure 16: State-to-State Translation Loss

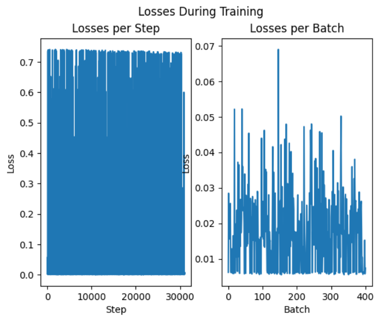

Figure 17: Delta Mappings Loss

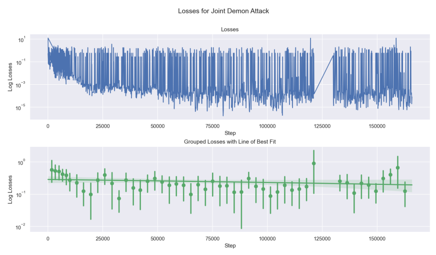

Figure 18: Joint Model (DA) Loss

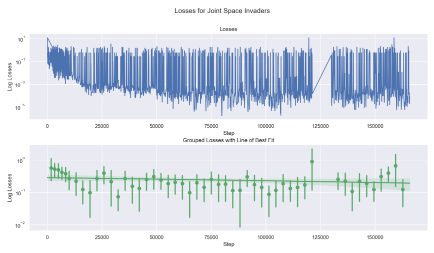

Figure 19: Joint Model (SI) Loss

Over our 600000 training steps, we achieved a maximum reward of 790 (bottom GIF), one that we're very pleasantly surprised with, given the warning of 4000000 steps needed to pass the level. Perhaps more impressively, the model managed this in not three lives, but one single life. Given more time, we would love the explore the capabilities of the model after millions of training steps.

5.2 Shared Model

Here we first inspect the ability of the state representation model to properly map values. Unfortunately, our results here were far from stellar. As it seems, the loss values for both the State-to-State Translations and Delta Mappings did not provide promising results (Figures 16 and 17).

We note that the state-to-state model converges on a local minima; it finds it far more rewarding to model just the static elements of each game than to actively try to represent changes in the model. Any attempt at modeling moving ships are likely undone by future layers, making it extremely difficult to converge further. We see that loss no longer decreases and stays relatively constant.

Like the state to state, the delta mapping model converges to a local minima and does not decreases its losses very much. The difference is that it converges on a static image of what appears to be probability distributions of the ship and enemies; clear white streaks can be seen where enemies typically appear and where the ship moves around. Again, it remains static with no decreasing loss.

Due to computational constraints, we were unable to provide meaningful results for either the latent spaces or combination methods. Due to the sheer intensity of especially the combination methods, our model trained a measly 10 steps over the course of an entire hour; in such a small time frame, no remotely reasonable results were possible.

Personal time constraints also limited our ability to integrate our state mappings with the rest of the model; as a result, we could only perform training on a naive, direct learning state representation (i.e. no states were translated between games). Despite this, our results are rather optimistic:

We note that the loss functions (Figures 18 and 19) behave rather similarly across all the models we created; it seems to decrease a bit towards the beginning but flatten out in the middle of training. There also seems to be a high variation in loss, as shown by how many spikes are in the graphs.

However, what is more interesting about the joint models is their Q functions (Figures 20 and 21):

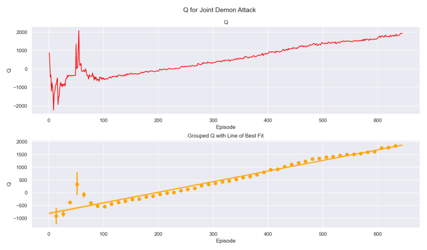

Figure 20: Joint Model (DA) Q Values

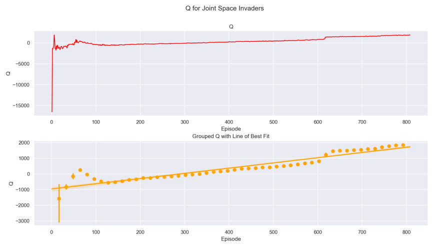

Figure 21: Joint Model (SI) Q Values

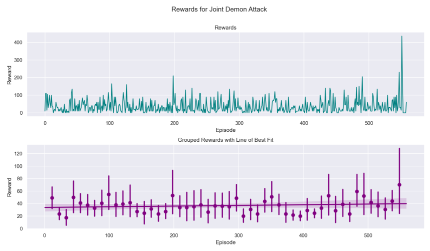

Figure 22: Joint Model (DA) Rewards

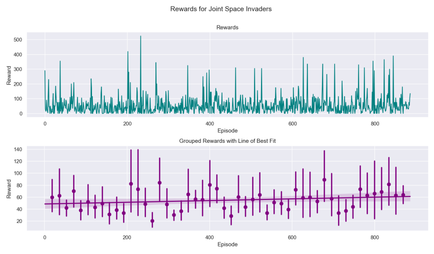

Figure 23: Joint Model (SI) Rewards

In terms of Q functions, both joint models seem to spike in the beginning, jumping from a negative to positive value; this exhibits the same resilience with Q-values as the individual model. Afterwards, they see steady linear growth, which is not uncommon among our models; perhaps what is surprising is that even with the alternating environments, the model is capable of generally increasing its Q-value estimations.

The reward function trends provide the most notable results (Figures 22 and 23):

The Joint Space-Invaders model hits about 60 reward value with 800 episodes, whereas `base2` only hit that mark at about 1250 episodes (a bit of data was lost for `base1`, making it seem like it converges faster than it does). On the other hand, Demon Attack averaged at about 40 reward value at 400 episodes, whereas base Demon Attack was around 50. We see slightly faster and slower convergences respectively, though it is unclear what would happen if the model trained even longer or, if it is within a margin of error. Either way, we remain cautiously optimistic about the potential results of our model; we see definitive strategy changes between the two models, and believe that shared experience probably trains just as well, if not better, than standard baseline models. More tests are necessary to test the method's efficacy, but visually we can see some shared knowledge informing decisions on both games.

6 Challenges

Overall, our project presented significantly more challenges than expected.

Perhaps the first we encountered was the sheer complexity of reinforcement learning research, combined with our naivete and lack of appreciation of its scope. As a result, the focus of our project shifted significantly over time. Our original idea came from the Actionable Models paper (linked in the introduction); with a combination of hindsight experience replay and actionable models on offline reinforcement learning datasets collected from robotic arms, we wished to re-implement key aspects of the paper and hopefully modify the architecture to apply to board games, such as Chess and Shogi.

Unfortunately, as we quickly realized, our knowledge of reinforcement learning, robotics, and board games were all severely lacking. As we continued research on the topic, comprehending the novel architectures in the papers themselves proved extremely difficult, let alone implementing them. After a fruitless week of bashing our heads against robotics and board games research papers, we decided to shift our idea toward an Atari-focused RL model.

This proved a little easier to digest, but still presented major problems. At this point, we began the RL section of the course, which aided in our understanding of the papers; we understood key concepts from Deep Q-Learning and Policy Gradient Learning methods, and better understood how the models trained. However, the sheer complexity of reinforcement learning papers still existed. Moreover, while most RL papers provided pseudocode for the implementations of their algorithms, many aspects were greatly unspecified, and converting the ideas into tangible code proved a great challenge. Even though some online resources were provided, we found it extremely difficult to parse, due to either poor/missing documentation, or significant optimizations that resulted in almost unparseable code (i.e. OpenAI's baseline implementations); moreover, the vast majority of models existed in PyTorch, while our familiarity lied completely in the domain of Tensorflow.

Here too emerged the second major problem: computational resources. Although we began implementing separate baseline RL models in an attempt to form a baseline, it proved exceedingly difficult to verify the results of our implementations. Because the models don't provide that many meaningful metrics, and training steps often vary greatly in their success, we were unable to detect bugs in our implementation (either the loss function or train function, usually), wasting significant hours on revising our models.

Even after we formulated a proper model, we lacked the computational resources to train each to their full extent; despite multiple attempts to set up a Google Cloud Compute engine to train our agents, we were unable to optimize the benefits of a GPU, and were forced to train completely locally. As a result, the maximum amount of time-steps that we could feasibly devote to each model happened to be around $600K$ steps, which fell significantly below the suggested $4M$ steps. The baseline and joint models thus did not fully train, yielding subpar visualizations.

Moreover, our formulated state representations ended up being significantly more complex than expected; although we could train the naive state representations (i.e. direct learning, state-to-state translations), they didn't end up yielding positive results, so we had to revamp our model architecture with additional steps. The final combination method we formulated ended up suffering the fate of lacking computational resources; when attempting to train our behemoth on the environment, we only trained on 10 steps in an entire hour. Without significantly more compute, training and testing the model proved impossible. Thus, we were forced to default to training the joint model on only the naive direct mapping on shared states. Although our results were slightly promising, the sheer complexity and requisite compute meant that we were unable to acquire substantive evaluations on the efficacy of our models.

Finally, while it did not prove an insurmountable bottleneck, our relative inexperience with building end-to-end deep and reinforcement learning systems in general proved a major hurdle. While we had general experience with implementing certain Tensorflow features from the homework assignments, we did not know how to construct support code and integration architecture from scratch. For instance, we focused primarily on getting a model up and running to train; however, we had not set up appropriate checkpoint, visualization, logging, and metrics architectures to properly evaluate our functionality. Especially in Deep Q-Networks, where temporal-difference loss doesn't provide meaningful results until very late into the training process (or even ever), logging the loss values is not particularly useful. More importantly, we did not thoroughly investigate image/GIF storage nor saving model checkpoints at first, which resulted in losing the progress on hundreds of thousands of steps over a few models.

7 Reflection

Although we did not manage to completely implement the shared state representation originally proposed, we are still generally optimistic on the results of our experiments. Once we recognized the sheer complexity and instability of reinforcement learning, our baseline and target goals reflected the desire to formulate a working Deep Q-Network to adequately play the Atari games (here, specifically Space Invaders). To that end, we met our base and target goals: we originally set out simply to replicate results from the seminal Mnih et al. 2013 Deep Q-Network, and hopefully extend some basic state representations. As we researched more, we discovered the Dueling Double Deep Q-Network, and set that as a target goal to implement after recognizing the relative increase in complexity compared to the naive DQN.

Our baseline DQN models also performed significantly better than we had expected; after seeing the suggested $4M$ steps to train the model to success on Space Invaders, we were not optimistic at all about the ability of our model to achieve moderate success on the Atari game. Despite that, after just around $350K$ steps, our model demonstrated a great performance on a single episodic life, reaching 790 points! More importantly though, our model appeared to learn key mechanics of the game. After around $200K$ steps, our model began to learn to explore beyond the left-most shield. Moreover, the model's dodging capabilities proved significantly more non-random than we had expected; and finally, the prioritization of the Mothership was a pleasant surprise.

Given more time, we would have liked to explore more reinforcement learning agents and attempted different models over a more reasonable number of iterations (e.g. $1M - 2M$). For instance, we did not have the time or compute to explore shared experiences with policy-gradient methods; especially given their current prominence in RL literature, a more thorough investigation and comparison of Deep Q-Learning and Policy Gradient methods would have been a worthwhile experience. As mentioned before, further explorations into more complicated shared domain experience buffers during training would be insightful as well.

On a personal level, this final project has been immensely beneficial to our growth as both deep learning engineers/programmers and researchers. After undergoing many iterations and setbacks throughout the process, we've learned to appreciate the complexity of many recent advancements in DL/RL research and the difficulty in turning theoretical pseudocode implementations into concrete models. Moreover, we've developed important skills in setting up and executing a deep learning system from scratch, and cultivated proper deep learning (and general) programming practices, from frequent checkpoints and logging to proper visualization and model presentation techniques.

After struggling to no avail with many research papers, we also feel more confident in assessing the state-of-the-art literature in deep/reinforcement learning, understanding foundational models and architectures, and most importantly modifying and tinkering with current implementations. Novel papers no longer seem impossibly daunting; after some effort, we now feel capable of demystifying key concepts and turning them into tangible implementations.

In all, the entire process has been greatly rewarding and fascinating, and after the final project we would love to continue exploring various RL architectures, playing with them, and (hopefully!) contributing meaningful results to the scientific community.

8 CONCLUSION

Recent advancements in reinforcement learning give promising signals for the growth of the field; many efficient and robust models have been developed to tackle various challenges, from sandbox (yet still fundamentally significant) environments like Atari and board games, to real and meaningful technological developments, such as robotics and self-driving cars. In each of these instances, however, exorbitant amounts of training and evaluation were required, resulting in prohibitively long training times for each individual model.

In this project, we sought first to re-implement—with minor adjustments—the foundational reinforcement learning agent of Dueling Double Deep Q-Networks as an educational experience in state-of-the-art reinforcement learning models. Then, we researched potential models to alleviate the training time through shared experiences, and explored architectures to generalize state representations into a common latent space. We ran initial training steps on a joint experience buffer between two similar environments—Atari's Space Invaders and Demon Attack games—to test the validity of a naive shared experience; while incomplete, we show cautiously optimistic learning results, providing a proof-of-concept possibility with even a naive implementation. With further formulation and benchmarking, we hope that shared experiences may be used in different environments such as robotics to alleviate training time and aggregate learning of similar tasks.