Using Human Gameplay to Augment Reinforcement Learning Models for Crypt of the NecroDancer

Buck Bukaty
Stanford University
353 Serra Mall, Stanford CA
bbukaty@stanford.edu

Dillon Kanne
Stanford University
353 Serra Mall, Stanford CA
dkanne@stanford.edu

Abstract

Training reinforcement learning models to play video games is a difficult task, made harder by the fact that they must simultaneously learn important visual features of the input images and attempt to find the best policy based on these features. We present a possible approach to alleviate this problem by pre-training the parameters of reinforcement learning models using behavioral cloning, a type of supervised learning.

First, we develop a behavioral cloning model using a convolutional neural network that can proficiently play the 2-D video game Crypt of the Necrodancer just by studying human gameplay, achieving 80% validation accuracy on labeled data. We use this model's learned parameters to pre-train two reinforcement learning models: a policy gradient model and a Deep Q-Network model. Despite being limited in training time due to difficulties with the infrastructure of our training environment, we report that this method of pre-training is reasonably effective for bootstrapping reinforcement learning. In particular, pre-training allows models to more quickly explore the state space of the game and achieve meaningful reward sooner than non pre-trained models.

1. Introduction

1.1. Problem Statement and Overview

Training a Reinforcement Learning (RL) model to play a moderately complex game is a difficult problem. While algorithms such as deep Q-learning have begun to demonstrate good performance on some games, applying reinforcement learning to games in general remains finicky [14]. One of the reasons for this is that an RL algorithm starts off with no information about what actions are effective in a given game, and, when using an untrained convolutional network as input to the agent, no intuition about what visual signals are important for making decisions, like a human would have. Recent work in understanding human priors in video games (such as the importance of objects and visual consistency) indicates how RL algorithms are at a distinct disadvantage for playing games intended for humans [9]. Might there be a way to give a reinforcement learning agent this information so it doesn't have to learn it all through trial and error?

One way to address this problem involves recorded human play in the form of a dataset of screen captures. Labels for this dataset would be the human player's corresponding control inputs to the game. A convolutional neural network trained on such a dataset could learn visual features important for decision making, and we could use the weights of this network to "pre-train" a reinforcement learning model. This reduces the problem to two, more manageable chunks: a standard image-classification problem on labeled data (the human actions) followed by the RL reward-maximization problem. This method is promising because it prevents the model from needing to learn visual features of the environment solely by random exploration.

Our results indicate that using behavioral cloning to pre-train the weights allows the reinforcement learning agent to better explore the state space of the game during training and achieve higher average reward. Although our model's training time was limited by system issues (described later), this method seems promising for bootstrapping the training of reinforcement learning models. In particular, without pre-training, our reinforcement learning agents spend many iterations taking random actions without achieving any reward. However, our pre-trained reinforcement learning agents are immediately capable of taking non-random actions and achieving reward, which both allows meaningful training to begin earlier and serves as a better starting point for complex decision making.

1.2. Related Work

The field of reinforcement learning has been reinvigorated in the last decade, partially due to the rise of neural networks as general purpose function approximators. In groundbreaking work on applying reinforcement learning to classic Atari games, Mnih et al. demonstrated the efficacy of using a neural network to approximate the Q function for Q-learning [14]. Additionally, they applied the technique of *experience replay*, where a buffer of state, action, reward, new state pairs (s, a, r, s') is used to update the Q-network instead of the most recent observed transitions, helping to stabilize the training process.

Despite recent advances in RL techniques inspired by the success of this paper, many have found that training reinforcement learning agents is a difficult process that often encounters "lack of stability during learning, low levels of robustness in the learnt behaviours, and slowness in convergence" [16]. Different approaches have been proposed to solve some of these problems, including averaging over previous Q-value estimates [4], but the training process for RL algorithms remains challenging. We consider the idea of transfer learning as an aid for this challenge.

The idea of using useful information from one model to boost the performance of another is not new in the field of computer vision. Transfer learning has been used in many domains to assist in the training of neural network models [20], and studies have attempted extending this technique to reinforcement learning. One study successfully pre-trained simpler RL models that receive domain-specific features as input instead of raw pixels [3]. Additionally, "supervised reinforcement learning" approaches have been developed that use prior knowledge to solve problems in mobile robotics [16]. Notably, DeepMind's AlphaGo first learned its policy networks through supervised learning of expert play [18].

These methods precede the training of their RL algorithms with the incorporation of some prior information, so that the agent doesn't have to learn all of the environment dynamics through random exploration. However, they do not operate based solely on visual information, incorporating sensorial data, game features, and expert knowledge. For our project, the most notable related work is Cruz et al.'s exploration of pre-training convolutional neural networks for deep reinforcement learning [7]. In this work, human gameplay is used as prior information for RL algorithms operating in the domain of Atari games, similar to Minh et al. We build off of this work by using a similar technique of pre-training the convolutional portion of a DQN with human demonstration. However, we also explore different RL algorithms, including more traditional policy gradients [19], and different neural network architectures, including ResNet [10].

A final piece of related work worth noting is the recently developed asynchronous advantage actor-critic algorithm (A3C) [13] for reinforcement learning. In this algorithm, a policy and value function are estimated with neural networks instead of a Q-function, and additional parallelism is supported through the training of multiple "workers" that each contribute to model updates. Cruz et al. describes A3C as "a new benchmark for deep RL" and experiments with applying pretraining to A3C as well as DQN. We did not experiment with the A3C algorithm as it is substantially complex to implement, and fell outside the scope of our project.

2. Data Infrastructure

2.1. Game Choice and Data Acquisition

We will explore this problem specifically through the game *Crypt of the Necrodancer*, a 2-D rhythm roguelike. Our dataset consists of gameplay frames (resized to 180x180 or 224x224 px.) recorded by one of the researchers with the associated input on each frame out of 4 categories: up, right, down, and left. All recorded data is from zone 1-1, the first zone of the game, and uses the basic character without picking up any items.

We chose *Crypt of the Necrodancer* for its small number of allowed inputs (arrow keys) and its rhythmic nature. Because everything in the game happens on the beat, the player's actions and the environment's responses are very discrete. This makes the comparison to a Markov Decision Process (MDP) especially clear. To simplify training and testing our model, we disable the rhythm component in the game (by choosing to play as a specific character) and query the model for the next action only once it has arrived on the next square.

We acquired 100 sessions of human gameplay in which a random variant of level 1-1 is beaten, each about 1 minute long in wall-clock time. We captured game frames at the instant the player pressed an arrow key, giving us the game state that the player was responding to by pressing said key. Beating level 1-1 requires roughly 130 moves on average, so each session had about 130 frames of labeled training data. In total, we had 12460 captured frames which were randomly split 90-10 into a training set of size 11214 and a validation set of size 1246.

2.2. Data Processing

Our input data, before processing, are 1024x1024 px. image captures of the game after each action. We then pad these images with black to 1080x1080 px. in order to align the image with the in-game grid of tiles. Once this is done, the image is a 15x15 grid of 72x72px in-game grid tiles. We then downscale these images to 180x180 px. using nearest-neighbor interpolation in order to train our network. We chose nearest-neighbor interpolation over bicubic because games have colorful, well defined shapes, and nearest neighbor interpolation can do a better job preserving the information-dense, rough edges between them. This decision is informed by a past CS231N project dealing with the same issue [6]. See Fig. 1 for details on our image processing pipeline.

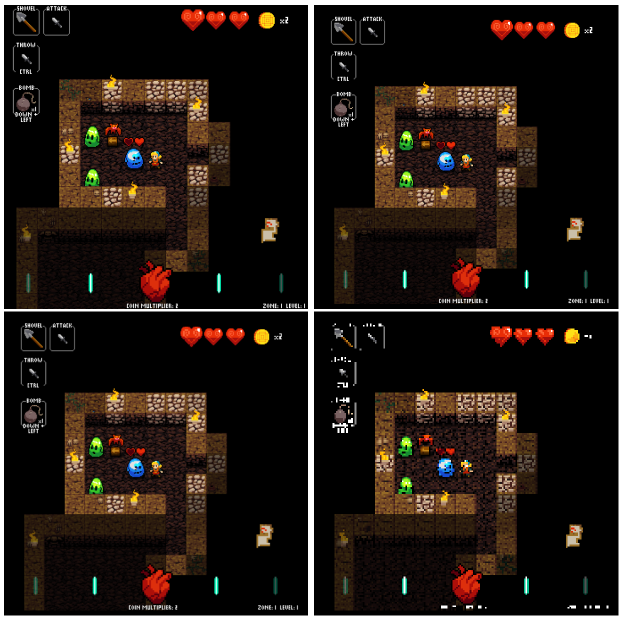


Figure 1. **Top Left.** A direct game capture of the game with size 1024x1024. **Top Right.** The same image as Top Left, but padded to 1080x1080 as mentioned in section 3.1. **Bottom Left.** The 360x360 result from downscaling the 1080x1080 image. Note that in the original game, each game pixel is a 3x3 square of actual pixels, so this downscaling is lossless. **Bottom Right.** The (lossy) 180x180 result from downscaling the Bottom Left image using nearest-neighbor interpolation. This image is then normalized as mentioned in 3.1 before training.

After we prepare the training data as in Fig. 1, we then normalize our data channel-wise by subtracting the mean of each of the three RGB channels over all images in the training set and dividing by the standard deviation similarly. We considered pixel-wise normalization, but encountered problems with it. Namely, some UI elements in the game almost never changed in the human training set. For example, the heart meter almost never reached 1/2 a heart in the training set, because disabling rhythm makes the first level of the game fairly easy for a human player. This caused the standard deviation of those pixels in the training set to be nearly 0. Because of this, if they changed during the agent's play, their normalized values became extremely high. This often caused the agent to disregard other information from the input and make the same erroneous decision over and over again. Generally, we found that channel normalization makes the convolutional network more robust to situations where certain specific pixels take on values never seen in the training set.

Moreover, we also experimented using pre-made architectures, namely ResNet [10], which use input image sizes different than 180x180. For these models, we simply took the 1024x1024 px. game capture and used nearest-neighbor interpolation to downscale to 224x224 px., the size that this architecture expects. This does not use any zero padding to perfectly align the game squares as described earlier.

We do not use data augmentation because our captured images have precisely the same format across the entire training set, so many of the traditional augmentation methods, such as cropping, rescaling, flipping, etc., do not apply in our context. We did consider using tools such as SMOTE [5], which takes convex combinations of pre-existing data to create more training data, but decided that these would create images too different from the original recorded data to be useful due to the game's regular structure.

3. Methods

3.1. Behavioral Cloning (BC)

Using our dataset of human gameplay and associated control inputs, we tested multiple Convolutional Neural Network (CNN) architectures in a supervised learning environment: classifying actions given the current game capture frame. This reduces the problem to simple image classification. Of course, this simplicity means that these CNNs have no notion of reward in the game environment, as their goal is not playing the game well (as defined by a reward function) but just to imitate human play. The parameters and weights of these CNN models were then used to pre-train reinforcement learning algorithms, described later in sections 3.3 and 3.4.

We anticipated that a somewhat non-standard convolutional architecture could prove more effective than an existing architecture such as ResNet. This is because of the grid-aligned nature of our training data; stride lengths and kernel sizes which align with the game grid to preserve spatial information may be more effective than, say, a 3x3 filter with stride 1. We tested using the following two architectures: "Simple" and "Deeper."

Model 1: "Simple" - input 180x180x3 image

1. CONV6: 6×6 size, 32 Filters, stride 3, padding 9.
2. CONV3: 3×3 size, 16 Filters, stride 1, padding 1.
3. CONV3: 3×3 size, 16 Filters, stride 2, padding 0.
4. Fully Connected: 8192×200 size.
5. Fully Connected: 200×60 size.
6. Fully Connected: 60×4 size.

Model 2: "Deeper" - input 180x180x3 image

1. CONV6: 6×6 size, 32 Filters, stride 3, padding 9.
2. Batch Normalization layer.
3. CONV3: 3×3 size, 32 Filters, stride 1, padding 1.
4. Batch Normalization layer.
5. CONV3: 3×3 size, 32 Filters, stride 1, padding 1.
6. Max pooling layer, stride 2.

7. Batch Normalization layer.
8. CONV3: 3×3 size, 32 Filters, stride 1, padding 1.
9. Batch Normalization layer.
10. CONV3: 3×3 size, 32 Filters, stride 1, padding 1.
11. Max pooling layer, stride 2.
12. Batch Normalization layer.
13. CONV3: 3×3 size, 32 Filters, stride 2, padding 0.
14. Fully Connected: 8192×200 size.
15. Dropout, $p = 0.5$.
16. Fully Connected: 200×60 size.
17. Dropout, $p = 0.5$.
18. Fully Connected: 60×4 size.

For both models, immediately succeeding every convolutional and fully connected layer is a ReLU layer, omitted for brevity. Additionally, the large padding on layer 1 resizes the next layer from 180x180 to 64x64 for convenience. We chose these architectures based on the architecture used in the original DQN paper [15], which used three convolutional layers and a single fully connected layer, similar to our "Simple" architecture.

Furthermore, as a baseline, we also tested using the ResNet34 architecture included in PyTorch [17]. For brevity, we do not include details of their architecture here, which can be found in Section 3.3 of [10]. The only modification we made to this architecture was changing the number of output classes in the final Fully Connected layer from 1,000 to 4, because ResNet34 was originally created to classify ImageNet images [8]. Furthermore, because the ResNet34 architecture in PyTorch only accepts images of size 224x224 px. or larger, we downscaled the gameplay images to 224x224 px. as mentioned in section 2.2. Note that even though we used the ResNet34 architecture in our tests, we did not use transfer learning by pre-training this network on something like ImageNet. This is because we are trying to classify actions, not image classes, and the style of example images used in ImageNet are very different than those in *Crypt of the Necrodancer*, which are rougher, more regular, and not photographs.

For all of these models, we used the Adam gradient descent update rule [12] using the cross entropy loss and a learning rate of 1e-3. Other hyperparameters for Adam used the defaults in PyTorch, namely $\beta_1 = 0.9, \beta_2 = 0.99$ and zero weight decay. We had a batch size of 64, which was the largest possible for our GPU capacity, and trained over eight epochs (about one hour). Due to time constraints, we did not perform learning rate annealing, which we suspect could have improved the classifier performance slightly. All of this testing took place using PyTorch in a customized version of the Jupyter notebook from Assignment 2.

3.2. Time Integration

Using the behavioral cloning networks in Section 3.1, we incorporated temporal information in the same way as done in [14]. To do this, for each training, validation, and test image, we "stacked" the last four frames in the history to create a 180x180x12 input (or 224x224x12 in the case of ResNet) instead of the single 180x180x3 image. This allows the network to see backwards in time a handful of frames, which is necessary for some decision making in the game, namely navigation and enemies with sparse movement patterns that only move once every few frames.

To implement this, the only necessary change was to change the first convolutional layer in all three of our models ("Simple," "Deeper," and ResNet34) to expect 12 input channels instead of three. From now on, we call this integration "quads" to differentiate it from classification using only a single frame, which we call "singles."

We also experimented with other ways of incorporating temporal information, including Late Integration, introduced in [11], but in our testing we found that this offered no improvement over single-frame models, so we did not pursue it in this work.

3.3. Policy Gradients

The BC networks from Section 3.1 take a game state as input and output class scores for the four actions. We interpret these networks as stochastic policy networks, in which the softmax of their output is used as probabilities of taking each of the four actions. Doing this, we can use the BC networks as the initial policy network for the policy gradient based REINFORCE algorithm [19]. In this RL algorithm, we attempt to maximize the value, J, of a policy π parameterized by θ,

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t \mid \pi_\theta\right]$$

where γ is a discount factor for future rewards and r_t is the reward at timestep t. Using an approximation of the gradient with respect to θ shown in class, the algorithm is as follows:

1. Sample a trajectory $\tau = (s_0, a_0, r_0, s_1 ...)$ from $\pi_\theta(a_t \mid s_t)$
2. Calculate $\nabla_\theta J(\theta) \approx \sum_{t\geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t)$
3. Update $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

The only hyperparameter of this algorithm is learning rate α. Intuitively, this algorithm samples a trajectory in the environment based on the current policy, then performs a gradient update on the policy which increases the probability of taking actions that led to higher rewards than expected.

3.4. Deep Q-Networks

In a reinforcement learning environment, the Q function gives the expected value of taking an action from a certain state: $Q(s, a)$. When the state space of an environment is very large, this Q function may be intractable to explicitly represent. Therefore, in deep Q-Networks, we train a neural network to approximate the Q function. Having an estimate of the Q function for a given state and its actions is useful because you can use it to define the optimal policy, in which you take the action with the maximum expected value:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s,a)$$

To measure the correctness of our Q function approximator, and therefore train it, we articulate the loss in terms of the Bellman equation, which defines the Q function in terms of immediate reward and expected future reward.

$$Q(s,a) = r + \gamma \max_a Q(s',a)$$

$$\text{Approximation error } \delta = Q(s,a) - (r + \gamma \max_a Q(s',a))$$

To minimize this error, we use the Huber loss, defined as

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,r,s')\ \in\ B} \mathcal{L}(\delta)$$

$$\text{where} \quad \mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \le 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

where B is a batch of (s,a,r,s') tuples. We choose this loss function because it is "more robust to outliers when the estimates of Q are very noisy" due to its linearity when the loss is of high magnitude [2].

Because our BC convolutional network outputs scores for the four actions as opposed to estimates of the Q-values of those actions, directly using its weights to preinitialize a Q-function approximator network does not make sense from a theoretical standpoint. As such, we removed the fully connected layers from the BC network before using it as a starting point for deep Q-learning; see Section 5.3.

4. Experiments

4.1. Behavioral Cloning

For our behavioral cloning models, we primarily used validation accuracy as a performance metric. However, in order to measure the performance of these models in the actual game, we developed a test set of randomly seeded levels in the game. We compared our models' gold score on these test levels against a human "oracle" score and the other models scores'.

4.2. Reinforcement Learning

Next, we tested both policy gradients and DQNs in the *Crypt of the NecroDancer* environment with a variety of different settings. The most important experimental change was of course whether preinitialization was used for the RL model before training. For all reinforcement learning experiments where we pre-trained the model, we pre-trained using the parameters and architecture of the "Deeper" BC model. This is because it performed better than the "Simple" model and the ResNet34 model was too big to fit on our GPU memory during RL training.

Another setting we tested was which of the fully connected layers of the BC model to omit for preinitialization, so that the RL algorithms could relearn decision making weights based on the somewhat lower level features learned by the BC network.

Finally, we experimented with a few different epsilon decay schedules for Deep Q-learning, though we mostly settled on one schedule and focused on the other aforementioned changes.

The reward function we used in these experiments rewarded the agent for picking up gold and for reaching the end of the level. If the agent did not get a reward within the last 80 actions, we terminated the episode to encourage the agent to continue to actively pursue rewards. We also tested a variant of this reward function in which the agent was punished for taking damage from enemies for both policy gradients and deep Q-learning.

Because we did not have access to a simulator for the game, many of these experiments were limited to the low number of 100 episodes, each representing 1 playthrough of zone 1-1. The game screen was captured for input to our models and manually parsed for reward indicators. The models took actions in the environment by programatically sending keyboard inputs.

Additionally, we quickly found that reinforcement learning algorithms had a difficult time learning a good policy when we used a new randomly seeded level for each training episode. To simplify our results, we therefore performed all our reinforcement learning experiments on the same seeded level across all training episodes.

5. Results

5.1. Behavioral Cloning Performance

The results of our tests with the various behavioral cloning architectures can be seen in Table 1. We saw a substantial improvement from our naive model ("Simple") to the slightly deeper model ("Deeper") with better normalization and regularization. We also saw more subtle improvements when using image quads instead of single images. Qualitatively, the use of image quads had a noticeable effect on in-game performance. We observed that BC models trained and tested using image quads were better at navigating levels and avoiding getting stuck in loops.

Compared with a ResNet34, our best architecture was much less deep with comparable performance. This was useful for intensive reinforcement learning training, when the necessary information for gradient updates for the ResNet34 overflowed GPU memory.

Model	Val. Acc.	1	2	3	4	5	Mean
Simple, singles	73%	16	14	0	20	0	10
Deeper, singles	77%	3	0	16	35	34	17.6
ResNet, singles	78%	41	31	23	18	2	23
Simple, quads	74%	8	16	9	24	2	11.8
Deeper, quads	79%	9	37	13	43	0	20.4
ResNet, quads	80%	9	34	39	12	46	28
Oracle		154	185	144	200	169	170.4

Table 1. Various model performances, measured by validation accuracy and gold accrued on seeded levels. Each of the numbers, 1 through 5, indicate the gold accrued on each of the levels with the mean over all five levels in the rightmost column.

5.2. Policy Gradient Performance

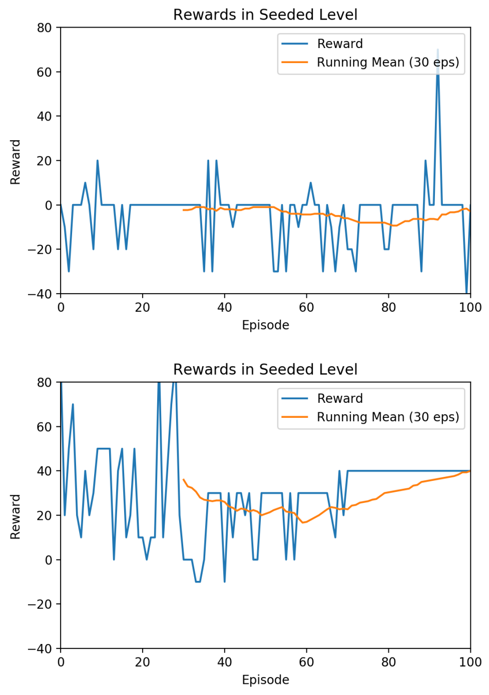


Figure 2. REINFORCE algorithm performance over training epochs. *Above*: no preinitialization. *Below*: with preinitialization from "deeper" BC network.

Figure 2 illustrates the performance of the REINFORCE algorithm over the course of 100 training epochs with and without preinitialization with the weights from our "Deeper" BC network. The reward function being used for this experiment included punishments for taking damage. Additionally, none of the fully connected layers of the behavioral cloning network were reset prior to its use as the policy network for REINFORCE.

When initialized with the BC weights, the REINFORCE algorithm was immediately able to navigate the level, with reward fluctuating around 30 for the first 20 epochs of training. Around the halfway point of training, the variance of the reward gained by the agent decreases, and towards the end of the set of training epochs the algorithm converges to a policy which gains a consistent 40 reward. By the end of training, the model network was functionally a fixed policy, and it would take the same trajectory each episode with no exploration. This learned fixed policy can be seen here: `https://youtu.be/emKc8WQkaQo`.

In comparison, the model without preinitialization qualitatively did a poor job exploring the level, often getting stuck in corners. In the instances where it navigated towards enemies, it often took damage and got a negative reward, having the unfortunate consequence of reinforcing the agent not to go near the enemies again.

Other experiments with REINFORCE included:

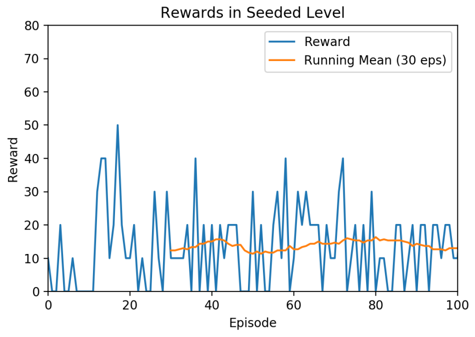
- Not using the last fully connected layer from the BC network.
- Not using the last two fully connected layers from the BC network.
- Using a reward function which didn't punish the agent for taking damage.

In these experiments, neither the preinitialized network nor the non-preinitialized network were able to make significant progress in learning a good policy for the environment. We believe this is mostly due to the significant stochasticity present in the training process, especially considering how our lack of a simulator for the game severely limited the number of epochs, and therefore number of experiments, we were able to perform.

5.3. DQN Performance

Figure 3 illustrates the performance of the Deep Q-learning algorithm over the course of 100 training epochs with and without preinitialization with the weights from our BC network. The reward function being used for this experiment did not include punishments for taking damage. In contrast to our experiments with REINFORCE, DQN did seem to benefit from removing the last fully connected layer from the BC network before using it as our Q-function approximator. This makes sense, as the behavioral cloning network was not trained with the environment's reward values in mind. Instead, it simply outputs scores for the purpose of supervised classification. Because the network would need to update those fully connected weights to accommodate for this, it makes more sense to just reinitialize them before use in deep Q-learning.

With preinitialization, the algorithm had a higher average reward throughout training and especially in the last 10



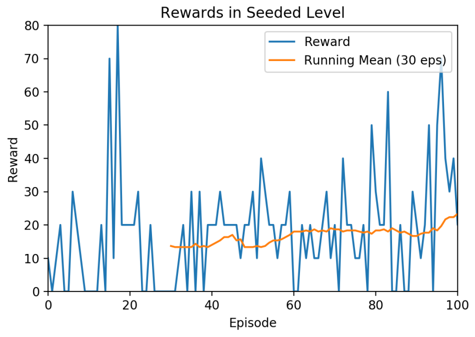


Figure 3. Deep Q-learning algorithm performance over training epochs. *Above*: no preinitialization. *Below*: with preinitialization from “deeper” BC network.

episodes. Additionally, the higher variance in rewards suggests that the agent was doing a better job exploring the environment; qualitative observation of the agent during training supports this idea.

5.4. Behavioral Cloning CNN Visualization

After quantifying the performance of our models, we performed tests to investigate and understand the decision-making of one of our behavioral cloning CNNs, namely the best performing ResNet34 CNN using image quads. Figure 4 shows an example training set image and its corresponding occlusion map, created using the same method as described in [21]. We also created other saliency and occlusion maps, but they presented largely similar qualitative results, so we relegated them to our supplementary material.

Despite Figure 4 only showing example occlusion maps from our test using ResNet34, the different models (“Simple” and “Deeper”) show similar patterns. This figure indicates that the model understands that the most important image in each classification is the most recent image, despite being given no information concerning which of the four images were most important or even that they were correlated. This seems to indicate that our behavioral cloning model had some notion of how to use the time information given despite never being explicitly told that the twelve channels it was given were related temporally.

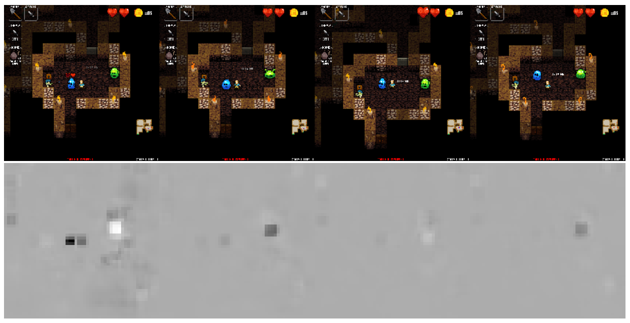


Figure 4. An occlusion map of a given training set image quad during a fight with slimes where the correct action is to move left (attack the blue slime). Black regions correspond to areas of the image important for making correct classifications, and white regions correspond to areas that, when removed, make correct classification more likely. The three most important areas for classification are the two slimes and the character itself. Note that the leftmost image is the most recent and the rightmost is the furthest back in time.

Furthermore, to investigate features the network learned to classify the best, we sorted all training images by classification score of the correct class and qualitatively analyzed what types of situations the model is most confident about. These findings can be seen in Figure 5. These categories agree with our own experience watching the model play the game: it is adept at killing slimes, hopping on gold, and navigating from room to room, in general. We discuss reasons why the model may have learned these situations particularly well in Section 6.1.

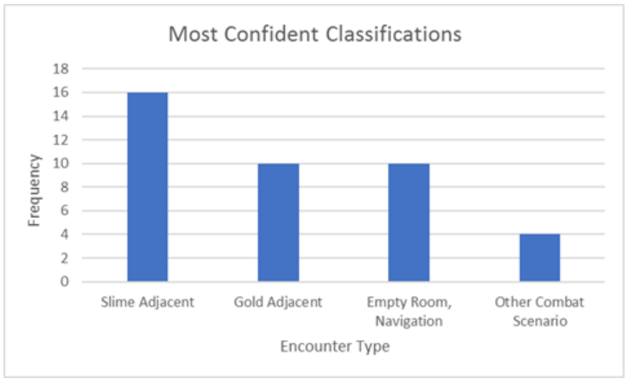


Figure 5. Categories the top forty most confident classifications roughly fall into. Note that the vast majority of frames do not have slimes or gold adjacent to the character, indicating that our CNN model has learned these behaviors (hitting adjacent slimes, hopping on gold, and moving from room to room) significantly more strongly than others.

6. Discussion

6.1. Effectiveness of Behavioral Cloning

Behavioral cloning, although conceptually and theoretically simpler than RL, is surprisingly effective for playing games even with limited, non expert training data. Our ResNet using quads achieved a validation accuracy of 80%, which is similar to previous CS 231N projects using behavioral cloning in video games [6]. Unfortunately, mistakes in *Crypt of the Necrodancer* are more punishing than many other games, and a single mistake in a combat encounter often means death.

Our models had mixed results in encounters with difficult-to-predict enemies (e.g. bats) and enemies that move directly at the player (e.g. skeletons and ghosts), which explains the significantly lower gold accrued than the oracle in Table 1. We believe that additional training data could help solve these problems, but primarily being able to run reinforcement learning for a longer period than 100 sessions would also help. Nevertheless, in many situations, the model effectively navigates and kills all monsters in randomly-generated rooms never seen before in the training data, often beating the entire zone. Considering that this didn't require any of the complicated reinforcement learning infrastructure that we later developed, behavioral cloning has a lot of appeal as a simple, approachable strategy for game AI. Our best behavioral cloning model, utilizing image quads, can be seen in action here: `https://www.youtube.com/watch?v=8Quvjy1_GfY`.

6.2. Impact of Pre-Training on RL

After our experiences training reinforcement learning models for *Crypt of the NecroDancer*, we have a much better understanding for why such models can be so difficult to train. Models without pre-training, for lack of a better word, flail around aimlessly in the game with no conception of the game's rules or objectives other than the sparse rewards it receives. When preinitialized with weights from the behavioral cloning model, however, even given the stochastic nature of RL training, it was clear that the agent's explorations of the state space were more useful. For example, without pre-training, the model often hit a wall with the shovel 80 times before end of the episode, resulting in zero reward. However, the pre-trained model would often run through a fair portion of the level, killing one enemy but missing its dropped gold, then dying to a spike trap in a complicated combat encounter. Both of these resulted in zero total reward, but the pre-trained model is clearly a better starting point for RL training.

Looking at figures 2 and 3, it is hard to rigorously and quantitatively justify the increase in performance caused by pre-training due to the small (but noticeable) difference in the rewards over time for non pre-trained and pre-trained models. This is due to the high variance present during the training process of RL algorithms and the low number of experiments we were able to run without a simulator. Nonetheless, our qualitative results and our observations of the agent's play give us confidence that pre-training RL models is a useful strategy, backing up Cruz et al.'s similar results [7]. At the bare minimum, we believe that our work indicates that using behavioral cloning as a starting point for training RL algorithms is certainly better than randomness.

6.3. Conclusion and Future Work

In this project, we explored neural network models for game playing in the domain of *Crypt of the NecroDancer*. Simple behavioral cloning models that mimicked human gameplay exceeded expectations in this domain. Attempts to use these models as starting points for more intelligent reinforcement learning algorithms were not as successful as expected, but nonetheless validated the intuition of the project that such a strategy is promising.

Possible Alternative Architectures. We believe that possible alternative methods to play *Crypt of the Necrodancer* could involve using a Recurrent Neural Network (RNN). The fact that our models only had access to the last four frames of the game limited their ability for long-term planning. Planning in *Crypt of the Necrodancer* is important for navigating through levels, so using one of these architectures with longer-term memory may increase in-game performance.

Importance of a Simulator. We learned late into the project that our reinforcement learning algorithm is rate-limited not by our compute, but by the ability to run the game fast enough to play a significant number of sessions. In fact, *Crypt of the Necrodancer* runs using the OS clock, so we couldn't even use utilities such as Cheat Engine to forcibly speed up the game in order to train our RL models faster. We believe that creating a simulator to run the game fast enough to perform the millions of gameplay steps (as seen in other reinforcement learning studies [7]) would allow our model to perform much better after RL training and clarify the advantages and disadvantages of pretraining.

Generalizing Behavioral Cloning. Given our success with BC and the fact that it did not take up nearly as many of our person-hours on this project to implement as reinforcement learning, we are interested in exploring how general these models can be while retaining good performance on simple games. We envision a unified system in which a user can record gameplay of themself playing any keyboard input based game, then receive as output a solidly performing CNN model for the game.

7. Contributions and Acknowledgements

7.1. Code Used and Problems Encountered

We used and adapted starter code from the Assignment 2 notebook to train our BC neural networks. However, we programmed the game capture, input recording, and game-playing (using the model to play the game and record rewards) infrastructure entirely ourselves without starter code. In this process, we encountered many problems that needed fixing, including (but not limited to): Windows taskbar captured by recording software, parsing individual pixels on screen to measure reward attained, Steam notifications being interpreted as beating the level, etc. We spent a significant portion of our time creating, tuning, and bug-fixing this code in order to create the training data and test our models.

We applied the REINFORCE algorithm using PyTorch's implementation as a starting point [1]. We modified the implementation to work with our RL environment wrapper around the game, and changed the training loop to work with our reward function's termination conditions. We also substituted in our own network architecture, and modified training code to work with stacks of four images, as described in 3.2.

Similarly, we used PyTorch's official Deep Q-Learning tutorial [2] as a base for our Q-Learning experiments. Similar changes were made to make things work with our game environment wrapper, as well as work with our own DQN network architecture with four frame stacked input. On top of this, changes were made to the way replay memory was stored, shuttling batches of replay memory transitions to and from the GPU when needed to prevent GPU memory overflow issues present in the original implementation. We also used Adam for optimization instead of their choice of RMSProp.

7.2. Author Contributions

We report no collaborators outside of the two authors on this assignment. Buck Bukaty worked on most of the programming for capturing game images, training the reinforcement learning models, and creating the wrapper around the game to allow our models to play live. Dillon Kanne worked on implementing and evaluating the behavioral cloning models, adapting single-image models to work with quads, and network visualization. Both authors worked equally on the presentation and writing of this project.

References

[1] Pytorch reinforce implementation. `https://github.com/pytorch/examples/blob/master/reinforcement_learning/reinforce.py`.

[2] Reinforcement learning (dqn) tutorial. `https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html`.

[3] C. W. Anderson, M. Lee, and D. L. Elliott. Faster reinforcement learning after pretraining deep networks to predict state dynamics. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–7. IEEE, 2015.

[4] O. Anschel, N. Baram, and N. Shimkin. Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. *arXiv preprint arXiv:1611.01929*, 2016.

[5] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[6] Z. Chen and D. Yi. The game imitation: Deep supervised convolutional networks for quick video game ai. *arXiv preprint arXiv:1702.05663*, 2017.

[7] G. V. Cruz Jr, Y. Du, and M. E. Taylor. Pre-training neural networks with human demonstrations for deep reinforcement learning. *arXiv preprint arXiv:1709.04083*, 2017.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[9] R. Dubey, P. Agrawal, D. Pathak, T. L. Griffiths, and A. A. Efros. Investigating human priors for playing video games. *arXiv preprint arXiv:1802.10217*, 2018.

[10] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[11] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[12] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[15] 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*, 518(7540):529, 2015.

[16] D. L. Moreno, C. V. Regueiro, R. Iglesias, and S. Barro. Using prior knowledge to improve reinforcement learning in mobile robotics. *Proc. Towards Autonomous Robotics Systems. Univ. of Essex, UK*, 2004.

[17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[18] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[19] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.

[20] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[21] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.