Using Reinforcement Learning to Train a Computer to Play Pitfall on Atari 2600

Game Background

Pitfall is a side scrolling action game where you have to avoid hitting logs, snakes, fire, alligators. You must also avoid falling into any pits by either jumping onto floating obstacles or swinging on vines. Hitting a log causes you to lose points while falling in pits and getting hit by monsters kills you. The goal is to collect all 34 treasures within the 20 minute time limit.

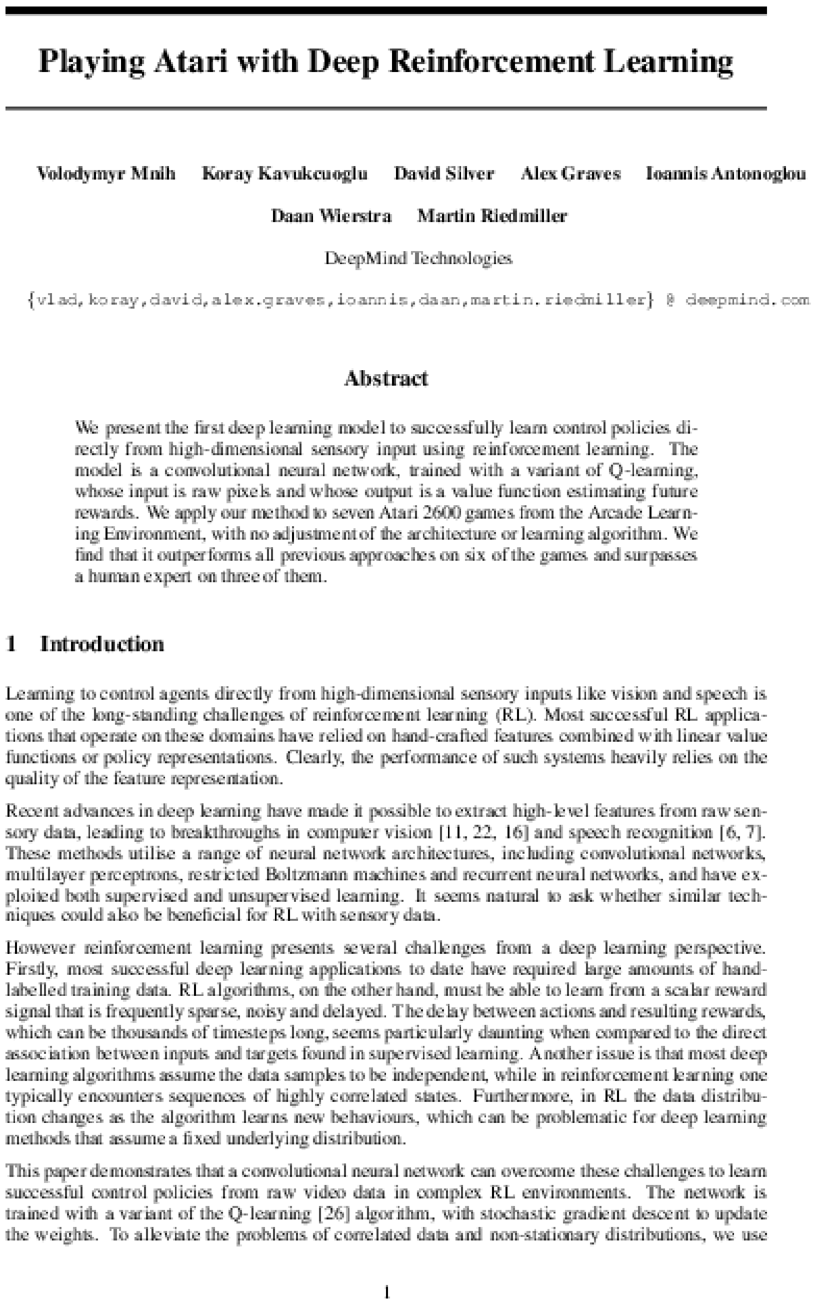
Process:

I started off attempting to train the CNN with no replay buffer. Unfortunatly, the rewards in Pitfall are a long way off. Reading through some walkthroughs shows that the first reward is at least 12 screens away if you utalize the underground shortcuts. Without the replay buffer there was no way for my model to get far enough along to find a reward. Therefore, my first attempt resulted in my model learning that not moving at all was the best option because you didn't lose any points.

After doing some further research I found a white paper that generalized exactly what I wanted to do. Specifically the algorithm outlined on page 5.

```
In [12]: from wand.image import Image as WImage
         img = WImage(filename='dqn.pdf[0]')
         img
```

Out[12]:



Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou
Daan Wierstra Martin Riedmiller
DeepMind Technologies
{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

1 Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [11, 22, 16] and speech recognition [6, 7]. These methods utilise a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines and recurrent neural networks, and have exploited both supervised and unsupervised learning. It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data.

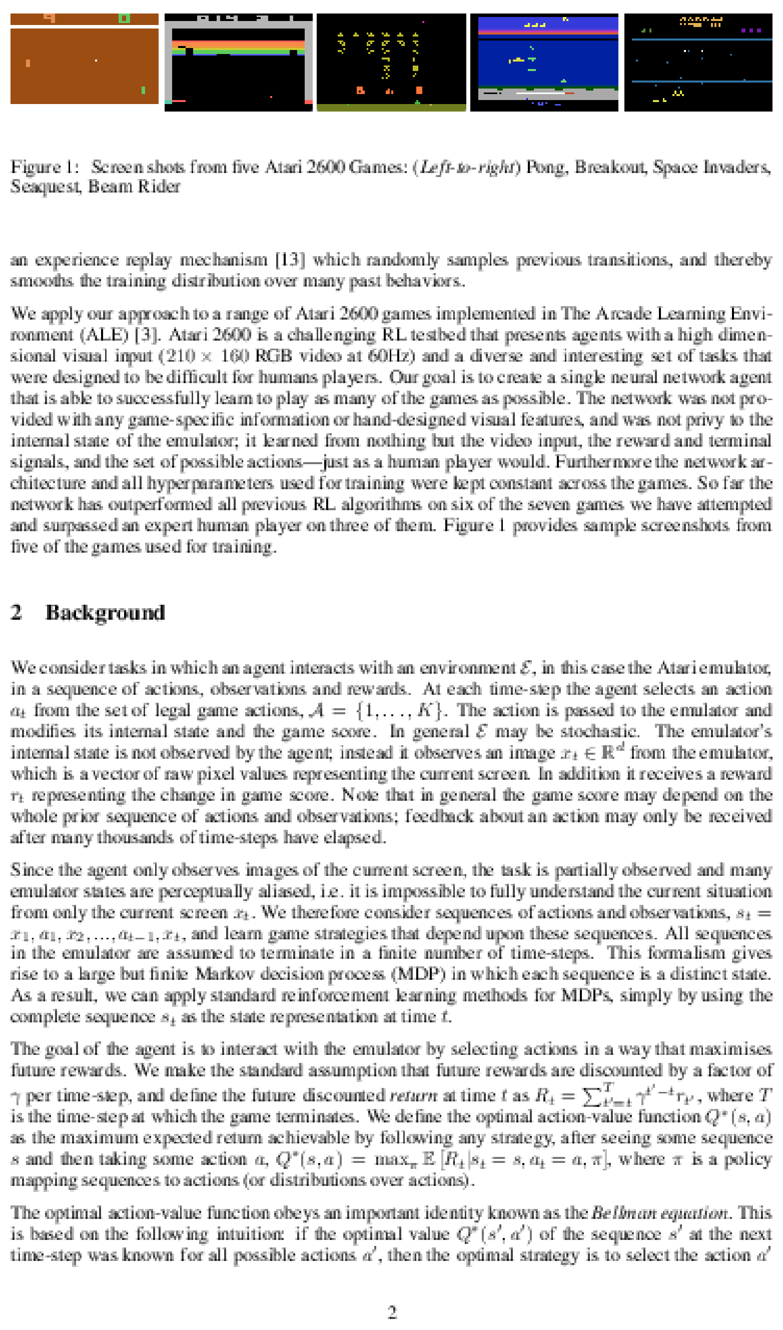
However reinforcement learning presents several challenges from a deep learning perspective. Firstly, most successful deep learning applications to date have required large amounts of hand-labelled training data. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting rewards, which can be thousands of timesteps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning. Another issue is that most deep learning algorithms assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviours, which can be problematic for deep learning methods that assume a fixed underlying distribution.

This paper demonstrates that a convolutional neural network can overcome these challenges to learn successful control policies from raw video data in complex RL environments. The network is trained with a variant of the Q-learning [26] algorithm, with stochastic gradient descent to update the weights. To alleviate the problems of correlated data and non-stationary distributions, we use

1

In [13]:
```
img = WImage(filename='dqn.pdf[1]')
img
```

Out[13]:



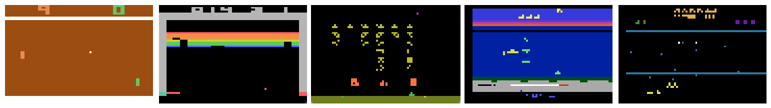


Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

an experience replay mechanism [13] which randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors.

We apply our approach to a range of Atari 2600 games implemented in The Arcade Learning Environment (ALE) [3]. Atari 2600 is a challenging RL testbed that presents agents with a high dimensional visual input (210×160 RGB video at 60Hz) and a diverse and interesting set of tasks that were designed to be difficult for humans players. Our goal is to create a single neural network agent that is able to successfully learn to play as many of the games as possible. The network was not provided with any game-specific information or hand-designed visual features, and was not privy to the internal state of the emulator; it learned from nothing but the video input, the reward and terminal signals, and the set of possible actions—just as a human player would. Furthermore the network architecture and all hyperparameters used for training were kept constant across the games. So far the network has outperformed all previous RL algorithms on six of the seven games we have attempted and surpassed an expert human player on three of them. Figure 1 provides sample screenshots from five of the games used for training.

2 Background

We consider tasks in which an agent interacts with an environment $\mathcal{E}$, in this case the Atari emulator, in a sequence of actions, observations and rewards. At each time-step the agent selects an action a_t from the set of legal game actions, $\mathcal{A} = \{1, \ldots, K\}$. The action is passed to the emulator and modifies its internal state and the game score. In general $\mathcal{E}$ may be stochastic. The emulator's internal state is not observed by the agent; instead it observes an image $x_t \in \mathbb{R}^d$ from the emulator, which is a vector of raw pixel values representing the current screen. In addition it receives a reward r_t representing the change in game score. Note that in general the game score may depend on the whole prior sequence of actions and observations; feedback about an action may only be received after many thousands of time-steps have elapsed.

Since the agent only observes images of the current screen, the task is partially observed and many emulator states are perceptually aliased, i.e. it is impossible to fully understand the current situation from only the current screen x_t. We therefore consider sequences of actions and observations, $s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$, and learn game strategies that depend upon these sequences. All sequences in the emulator are assumed to terminate in a finite number of time-steps. This formalism gives rise to a large but finite Markov decision process (MDP) in which each sequence is a distinct state. As a result, we can apply standard reinforcement learning methods for MDPs, simply by using the complete sequence s_t as the state representation at time t.

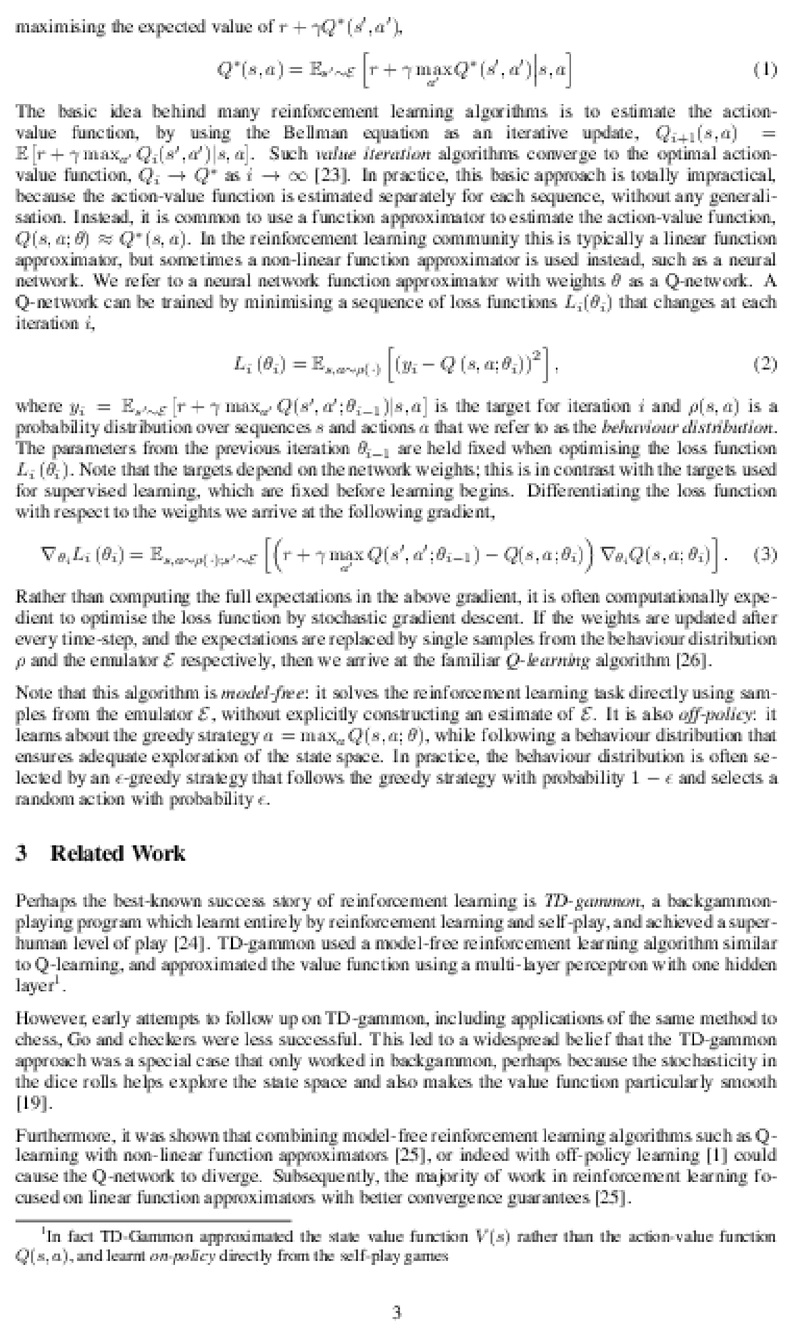
The goal of the agent is to interact with the emulator by selecting actions in a way that maximises future rewards. We make the standard assumption that future rewards are discounted by a factor of γ per time-step, and define the future discounted *return* at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time-step at which the game terminates. We define the optimal action-value function $Q^*(s,a)$ as the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a, $Q^*(s,a) = \max_\pi \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$, where π is a policy mapping sequences to actions (or distributions over actions).

The optimal action-value function obeys an important identity known as the *Bellman equation*. This is based on the following intuition: if the optimal value $Q^*(s', a')$ of the sequence s' at the next time-step was known for all possible actions a', then the optimal strategy is to select the action a'

2

In [14]:
```
img = WImage(filename='dqn.pdf[2]')
img
```

Out[14]:



maximising the expected value of $r + \gamma Q^*(s', a')$,

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') \middle| s, a \right] \tag{1}$$

The basic idea behind many reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation as an iterative update, $Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s', a') | s, a\right]$. Such *value iteration* algorithms converge to the optimal action-value function, $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$ [23]. In practice, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalisation. Instead, it is common to use a function approximator to estimate the action-value function, $Q(s, a; \theta) \approx Q^*(s, a)$. In the reinforcement learning community this is typically a linear function approximator, but sometimes a non-linear function approximator is used instead, such as a neural network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i\left(\theta_i\right) = \mathbb{E}_{s,a \sim \rho(\cdot)}\left[\left(y_i - Q\left(s, a; \theta_i\right)\right)^2 \right], \tag{2}$$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a\right]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a that we refer to as the *behaviour distribution*. The parameters from the previous iteration θ_{i-1} are held fixed when optimising the loss function $L_i\left(\theta_i\right)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. Differentiating the loss function with respect to the weights we arrive at the following gradient,

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right]. \tag{3}$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimise the loss function by stochastic gradient descent. If the weights are updated after every time-step, and the expectations are replaced by single samples from the behaviour distribution ρ and the emulator $\mathcal{E}$ respectively, then we arrive at the familiar *Q-learning* algorithm [26].

Note that this algorithm is *model-free*: it solves the reinforcement learning task directly using samples from the emulator $\mathcal{E}$, without explicitly constructing an estimate of $\mathcal{E}$. It is also *off-policy*: it learns about the greedy strategy $a = \max_a Q(s, a; \theta)$, while following a behaviour distribution that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an ϵ-greedy strategy that follows the greedy strategy with probability $1 - \epsilon$ and selects a random action with probability ϵ.

3 Related Work

Perhaps the best-known success story of reinforcement learning is *TD-gammon*, a backgammon-playing program which learnt entirely by reinforcement learning and self-play, and achieved a super-human level of play [24]. TD-gammon used a model-free reinforcement learning algorithm similar to Q-learning, and approximated the value function using a multi-layer perceptron with one hidden layer[1].

However, early attempts to follow up on TD-gammon, including applications of the same method to chess, Go and checkers were less successful. This led to a widespread belief that the TD-gammon approach was a special case that only worked in backgammon, perhaps because the stochasticity in the dice rolls helps explore the state space and also makes the value function particularly smooth [19].

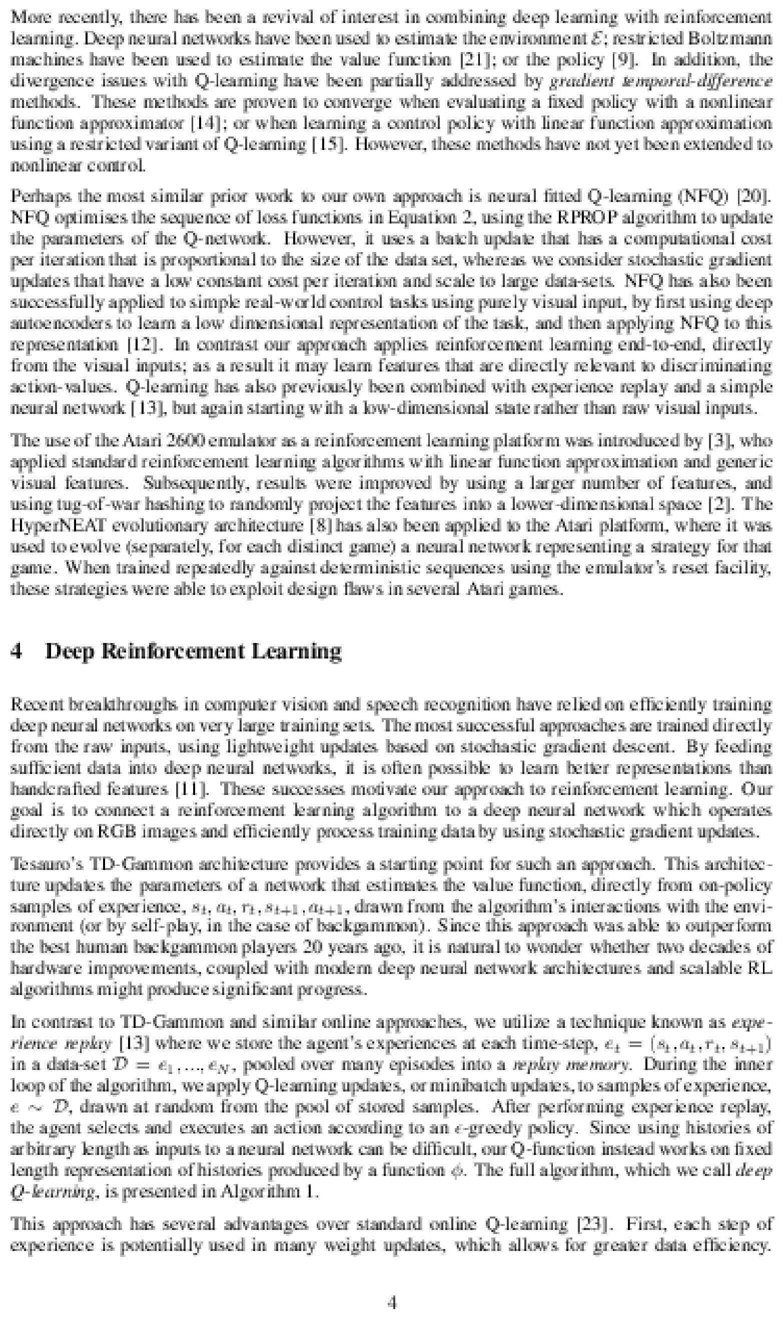
Furthermore, it was shown that combining model-free reinforcement learning algorithms such as Q-learning with non-linear function approximators [25], or indeed with off-policy learning [1] could cause the Q-network to diverge. Subsequently, the majority of work in reinforcement learning focused on linear function approximators with better convergence guarantees [25].

[1]In fact TD-Gammon approximated the state value function $V(s)$ rather than the action-value function $Q(s, a)$, and learnt *on-policy* directly from the self-play games

3

In [15]:
```
img = WImage(filename='dqn.pdf[3]')
img
```

Out[15]:



More recently, there has been a revival of interest in combining deep learning with reinforcement learning. Deep neural networks have been used to estimate the environment $\mathcal{E}$; restricted Boltzmann machines have been used to estimate the value function [21]; or the policy [9]. In addition, the divergence issues with Q-learning have been partially addressed by *gradient temporal-difference* methods. These methods are proven to converge when evaluating a fixed policy with a nonlinear function approximator [14]; or when learning a control policy with linear function approximation using a restricted variant of Q-learning [15]. However, these methods have not yet been extended to nonlinear control.

Perhaps the most similar prior work to our own approach is neural fitted Q-learning (NFQ) [20]. NFQ optimises the sequence of loss functions in Equation 2, using the RPROP algorithm to update the parameters of the Q-network. However, it uses a batch update that has a computational cost per iteration that is proportional to the size of the data set, whereas we consider stochastic gradient updates that have a low constant cost per iteration and scale to large data-sets. NFQ has also been successfully applied to simple real-world control tasks using purely visual input, by first using deep autoencoders to learn a low dimensional representation of the task, and then applying NFQ to this representation [12]. In contrast our approach applies reinforcement learning end-to-end, directly from the visual inputs; as a result it may learn features that are directly relevant to discriminating action-values. Q-learning has also previously been combined with experience replay and a simple neural network [13], but again starting with a low-dimensional state rather than raw visual inputs.

The use of the Atari 2600 emulator as a reinforcement learning platform was introduced by [3], who applied standard reinforcement learning algorithms with linear function approximation and generic visual features. Subsequently, results were improved by using a larger number of features, and using tug-of-war hashing to randomly project the features into a lower-dimensional space [2]. The HyperNEAT evolutionary architecture [8] has also been applied to the Atari platform, where it was used to evolve (separately, for each distinct game) a neural network representing a strategy for that game. When trained repeatedly against deterministic sequences using the emulator's reset facility, these strategies were able to exploit design flaws in several Atari games.

4 Deep Reinforcement Learning

Recent breakthroughs in computer vision and speech recognition have relied on efficiently training deep neural networks on very large training sets. The most successful approaches are trained directly from the raw inputs, using lightweight updates based on stochastic gradient descent. By feeding sufficient data into deep neural networks, it is often possible to learn better representations than handcrafted features [11]. These successes motivate our approach to reinforcement learning. Our goal is to connect a reinforcement learning algorithm to a deep neural network which operates directly on RGB images and efficiently process training data by using stochastic gradient updates.

Tesauro's TD-Gammon architecture provides a starting point for such an approach. This architecture updates the parameters of a network that estimates the value function, directly from on-policy samples of experience, $s_t, a_t, r_t, s_{t+1}, a_{t+1}$, drawn from the algorithm's interactions with the environment (or by self-play, in the case of backgammon). Since this approach was able to outperform the best human backgammon players 20 years ago, it is natural to wonder whether two decades of hardware improvements, coupled with modern deep neural network architectures and scalable RL algorithms might produce significant progress.

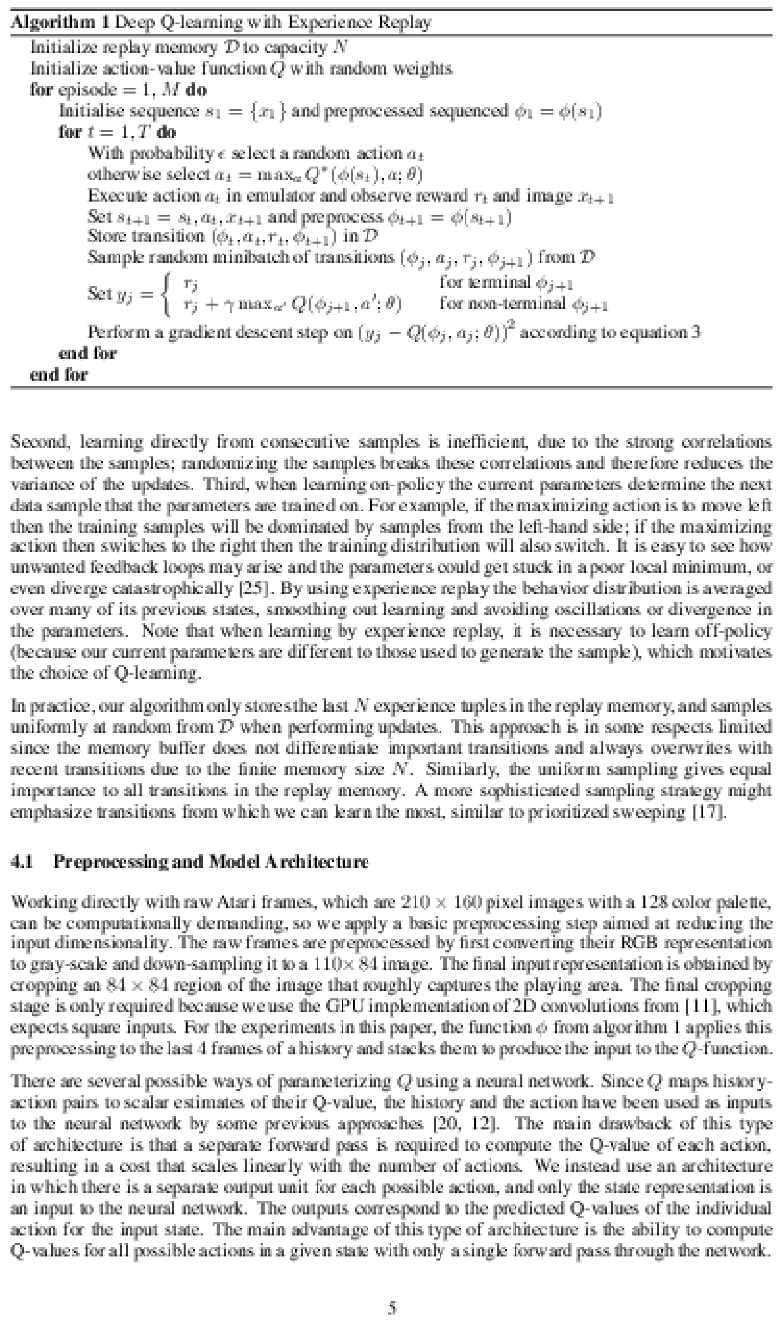
In contrast to TD-Gammon and similar online approaches, we utilize a technique known as *experience replay* [13] where we store the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data-set $\mathcal{D} = e_1, ..., e_N$, pooled over many episodes into a *replay memory*. During the inner loop of the algorithm, we apply Q-learning updates, or minibatch updates, to samples of experience, $e \sim \mathcal{D}$, drawn at random from the pool of stored samples. After performing experience replay, the agent selects and executes an action according to an ϵ-greedy policy. Since using histories of arbitrary length as inputs to a neural network can be difficult, our Q-function instead works on fixed length representation of histories produced by a function ϕ. The full algorithm, which we call *deep Q-learning*, is presented in Algorithm 1.

This approach has several advantages over standard online Q-learning [23]. First, each step of experience is potentially used in many weight updates, which allows for greater data efficiency.

4

In [16]:
```
img = WImage(filename='dqn.pdf[4]')
img
```

Out[16]:



Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Second, learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates. Third, when learning on-policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically [25]. By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. Note that when learning by experience replay, it is necessary to learn off-policy (because our current parameters are different to those used to generate the sample), which motivates the choice of Q-learning.

In practice, our algorithm only stores the last N experience tuples in the replay memory, and samples uniformly at random from $\mathcal{D}$ when performing updates. This approach is in some respects limited since the memory buffer does not differentiate important transitions and always overwrites with recent transitions due to the finite memory size N. Similarly, the uniform sampling gives equal importance to all transitions in the replay memory. A more sophisticated sampling strategy might emphasize transitions from which we can learn the most, similar to prioritized sweeping [17].

4.1 Preprocessing and Model Architecture

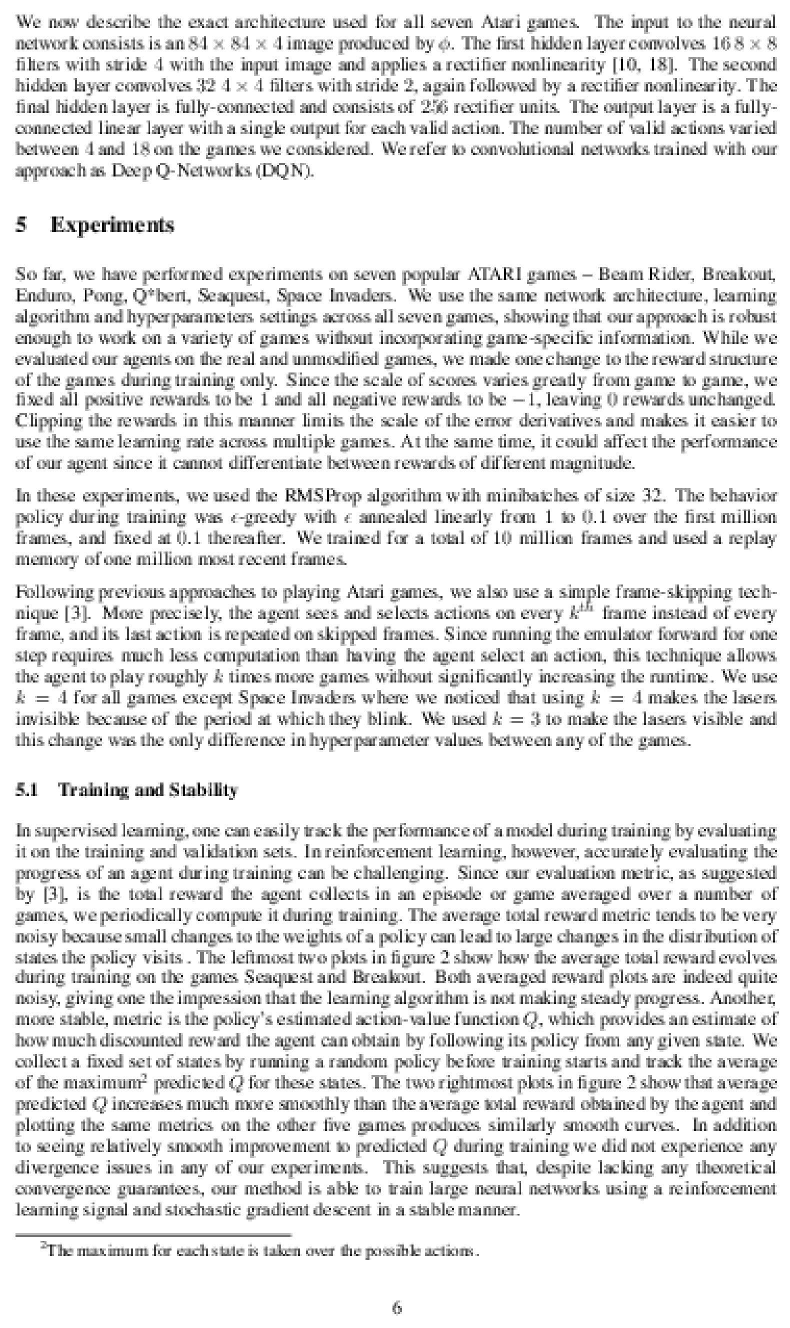
Working directly with raw Atari frames, which are 210×160 pixel images with a 128 color palette, can be computationally demanding, so we apply a basic preprocessing step aimed at reducing the input dimensionality. The raw frames are preprocessed by first converting their RGB representation to gray-scale and down-sampling it to a 110×84 image. The final input representation is obtained by cropping an 84×84 region of the image that roughly captures the playing area. The final cropping stage is only required because we use the GPU implementation of 2D convolutions from [11], which expects square inputs. For the experiments in this paper, the function ϕ from algorithm 1 applies this preprocessing to the last 4 frames of a history and stacks them to produce the input to the Q-function.

There are several possible ways of parameterizing Q using a neural network. Since Q maps history-action pairs to scalar estimates of their Q-value, the history and the action have been used as inputs to the neural network by some previous approaches [20, 12]. The main drawback of this type of architecture is that a separate forward pass is required to compute the Q-value of each action, resulting in a cost that scales linearly with the number of actions. We instead use an architecture in which there is a separate output unit for each possible action, and only the state representation is an input to the neural network. The outputs correspond to the predicted Q-values of the individual action for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network.

5

In [17]:
```
img = WImage(filename='dqn.pdf[5]')
img
```

Out[17]:



We now describe the exact architecture used for all seven Atari games. The input to the neural network consists is an $84 \times 84 \times 4$ image produced by ϕ. The first hidden layer convolves 16 8×8 filters with stride 4 with the input image and applies a rectifier nonlinearity [10, 18]. The second hidden layer convolves 32 4×4 filters with stride 2, again followed by a rectifier nonlinearity. The final hidden layer is fully-connected and consists of 256 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. The number of valid actions varied between 4 and 18 on the games we considered. We refer to convolutional networks trained with our approach as Deep Q-Networks (DQN).

5 Experiments

So far, we have performed experiments on seven popular ATARI games – Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, Space Invaders. We use the same network architecture, learning algorithm and hyperparameters settings across all seven games, showing that our approach is robust enough to work on a variety of games without incorporating game-specific information. While we evaluated our agents on the real and unmodified games, we made one change to the reward structure of the games during training only. Since the scale of scores varies greatly from game to game, we fixed all positive rewards to be 1 and all negative rewards to be -1, leaving 0 rewards unchanged. Clipping the rewards in this manner limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. At the same time, it could affect the performance of our agent since it cannot differentiate between rewards of different magnitude.

In these experiments, we used the RMSProp algorithm with minibatches of size 32. The behavior policy during training was ϵ-greedy with ϵ annealed linearly from 1 to 0.1 over the first million frames, and fixed at 0.1 thereafter. We trained for a total of 10 million frames and used a replay memory of one million most recent frames.

Following previous approaches to playing Atari games, we also use a simple frame-skipping technique [3]. More precisely, the agent sees and selects actions on every k^{th} frame instead of every frame, and its last action is repeated on skipped frames. Since running the emulator forward for one step requires much less computation than having the agent select an action, this technique allows the agent to play roughly k times more games without significantly increasing the runtime. We use $k = 4$ for all games except Space Invaders where we noticed that using $k = 4$ makes the lasers invisible because of the period at which they blink. We used $k = 3$ to make the lasers visible and this change was the only difference in hyperparameter values between any of the games.

5.1 Training and Stability

In supervised learning, one can easily track the performance of a model during training by evaluating it on the training and validation sets. In reinforcement learning, however, accurately evaluating the progress of an agent during training can be challenging. Since our evaluation metric, as suggested by [3], is the total reward the agent collects in an episode or game averaged over a number of games, we periodically compute it during training. The average total reward metric tends to be very noisy because small changes to the weights of a policy can lead to large changes in the distribution of states the policy visits . The leftmost two plots in figure 2 show how the average total reward evolves during training on the games Seaquest and Breakout. Both averaged reward plots are indeed quite noisy, giving one the impression that the learning algorithm is not making steady progress. Another, more stable, metric is the policy's estimated action-value function Q, which provides an estimate of how much discounted reward the agent can obtain by following its policy from any given state. We collect a fixed set of states by running a random policy before training starts and track the average of the maximum[2] predicted Q for these states. The two rightmost plots in figure 2 show that average predicted Q increases much more smoothly than the average total reward obtained by the agent and plotting the same metrics on the other five games produces similarly smooth curves. In addition to seeing relatively smooth improvement to predicted Q during training we did not experience any divergence issues in any of our experiments. This suggests that, despite lacking any theoretical convergence guarantees, our method is able to train large neural networks using a reinforcement learning signal and stochastic gradient descent in a stable manner.

[2]The maximum for each state is taken over the possible actions.

6

```
In [18]: img = WImage(filename='dqn.pdf[6]')
         img
```

Out[18]:

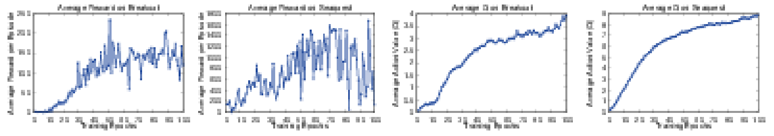


Figure 2: The two plots on the left show average reward per episode on Breakout and Seaquest respectively during training. The statistics were computed by running an ϵ-greedy policy with $\epsilon = 0.05$ for 10000 steps. The two plots on the right show the average maximum predicted action-value of a held out set of states on Breakout and Seaquest respectively. One epoch corresponds to 50000 minibatch weight updates or roughly 30 minutes of training time.

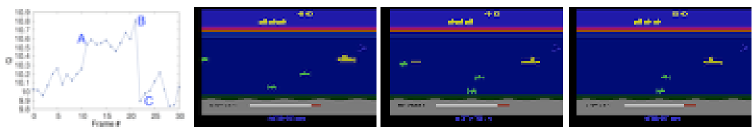


Figure 3: The leftmost plot shows the predicted value function for a 30 frame segment of the game Seaquest. The three screenshots correspond to the frames labeled by A, B, and C respectively.

5.2 Visualizing the Value Function

Figure 3 shows a visualization of the learned value function on the game Seaquest. The figure shows that the predicted value jumps after an enemy appears on the left of the screen (point A). The agent then fires a torpedo at the enemy and the predicted value peaks as the torpedo is about to hit the enemy (point B). Finally, the value falls to roughly its original value after the enemy disappears (point C). Figure 3 demonstrates that our method is able to learn how the value function evolves for a reasonably complex sequence of events.

5.3 Main Evaluation

We compare our results with the best performing methods from the RL literature [3, 4]. The method labeled **Sarsa** used the Sarsa algorithm to learn linear policies on several different feature sets hand-engineered for the Atari task and we report the score for the best performing feature set [3]. **Contingency** used the same basic approach as **Sarsa** but augmented the feature sets with a learned representation of the parts of the screen that are under the agent's control [4]. Note that both of these methods incorporate significant prior knowledge about the visual problem by using background subtraction and treating each of the 128 colors as a separate channel. Since many of the Atari games use one distinct color for each type of object, treating each color as a separate channel can be similar to producing a separate binary map encoding the presence of each object type. In contrast, our agents only receive the raw RGB screenshots as input and must *learn* to detect objects on their own.

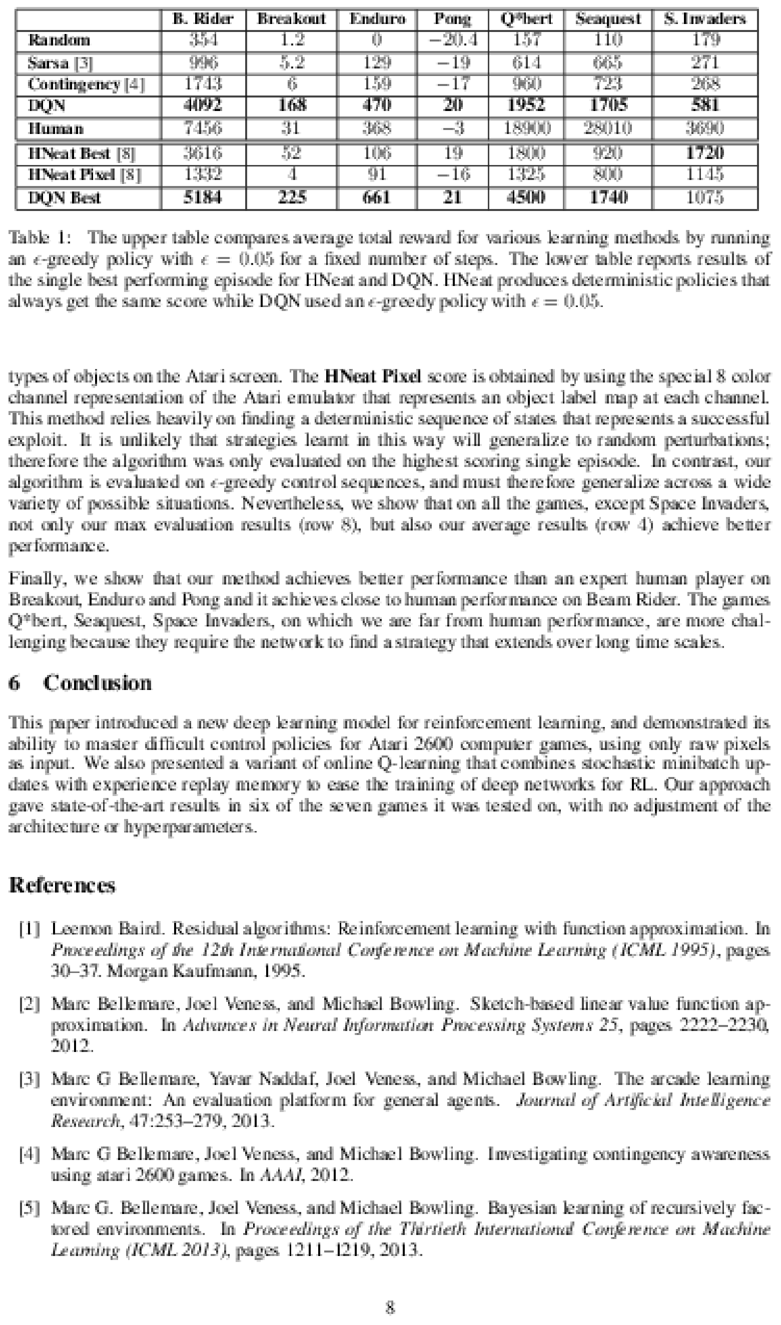
In addition to the learned agents, we also report scores for an expert human game player and a policy that selects actions uniformly at random. The human performance is the median reward achieved after around two hours of playing each game. Note that our reported human scores are much higher than the ones in Bellemare et al. [3]. For the learned methods, we follow the evaluation strategy used in Bellemare et al. [3, 5] and report the average score obtained by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The first five rows of table 1 show the per-game average scores on all games. Our approach (labeled DQN) outperforms the other learning methods by a substantial margin on all seven games despite incorporating almost no prior knowledge about the inputs.

We also include a comparison to the evolutionary policy search approach from [8] in the last three rows of table 1. We report two sets of results for this method. The **HNeat Best** score reflects the results obtained by using a hand-engineered object detector algorithm that outputs the locations and

7

In [19]:
```
img = WImage(filename='dqn.pdf[7]')
img
```

Out[19]:



	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

types of objects on the Atari screen. The **HNeat Pixel** score is obtained by using the special 8 color channel representation of the Atari emulator that represents an object label map at each channel. This method relies heavily on finding a deterministic sequence of states that represents a successful exploit. It is unlikely that strategies learnt in this way will generalize to random perturbations; therefore the algorithm was only evaluated on the highest scoring single episode. In contrast, our algorithm is evaluated on ϵ-greedy control sequences, and must therefore generalize across a wide variety of possible situations. Nevertheless, we show that on all the games, except Space Invaders, not only our max evaluation results (row 8), but also our average results (row 4) achieve better performance.

Finally, we show that our method achieves better performance than an expert human player on Breakout, Enduro and Pong and it achieves close to human performance on Beam Rider. The games Q*bert, Seaquest, Space Invaders, on which we are far from human performance, are more challenging because they require the network to find a strategy that extends over long time scales.

6 Conclusion

This paper introduced a new deep learning model for reinforcement learning, and demonstrated its ability to master difficult control policies for Atari 2600 computer games, using only raw pixels as input. We also presented a variant of online Q-learning that combines stochastic minibatch updates with experience replay memory to ease the training of deep networks for RL. Our approach gave state-of-the-art results in six of the seven games it was tested on, with no adjustment of the architecture or hyperparameters.

References

[1] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Machine Learning (ICML 1995)*, pages 30–37. Morgan Kaufmann, 1995.

[2] Marc Bellemare, Joel Veness, and Michael Bowling. Sketch-based linear value function approximation. In *Advances in Neural Information Processing Systems 25*, pages 2222–2230, 2012.

[3] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

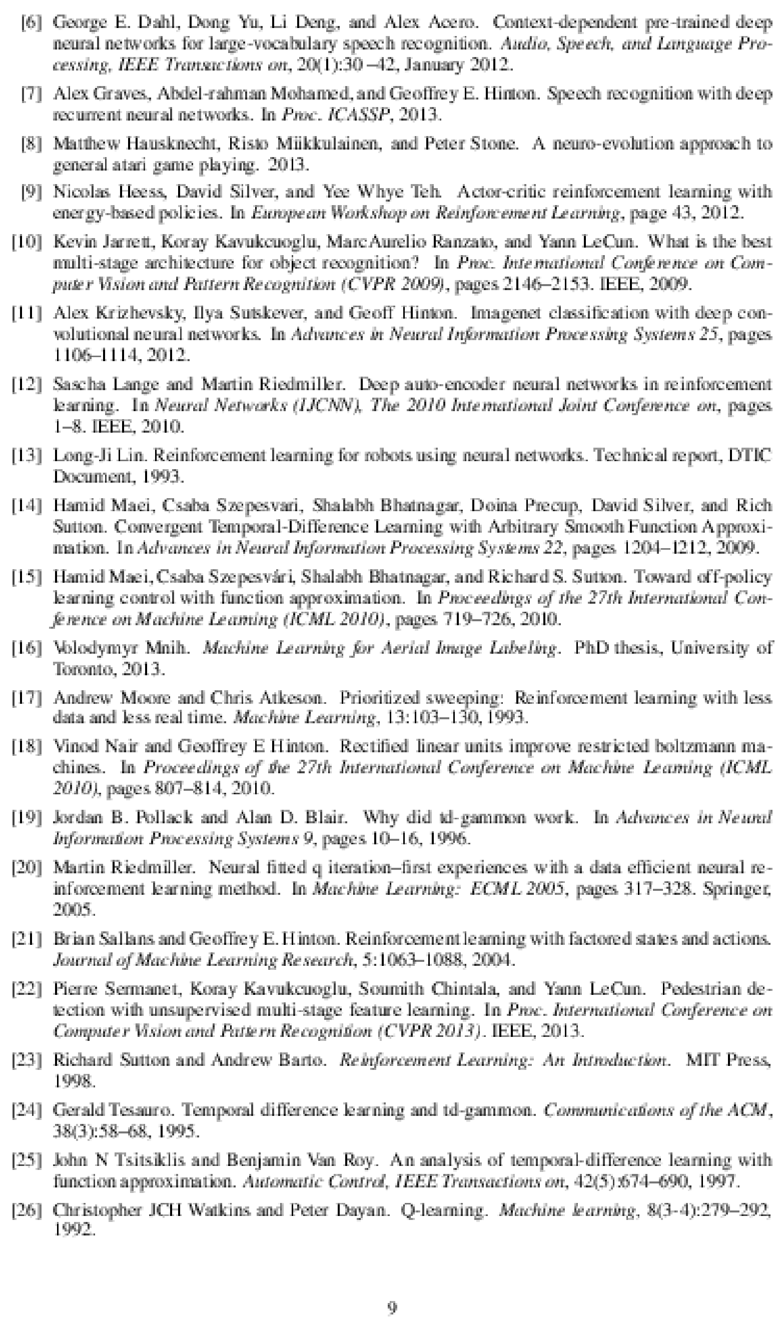
[4] Marc G Bellemare, Joel Veness, and Michael Bowling. Investigating contingency awareness using atari 2600 games. In *AAAI*, 2012.

[5] Marc G. Bellemare, Joel Veness, and Michael Bowling. Bayesian learning of recursively factored environments. In *Proceedings of the Thirtieth International Conference on Machine Learning (ICML 2013)*, pages 1211–1219, 2013.

8

In [20]:
```
img = WImage(filename='dqn.pdf[8]')
img
```

Out[20]:



[6] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30 –42, January 2012.

[7] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. In *Proc. ICASSP*, 2013.

[8] Matthew Hausknecht, Risto Miikkulainen, and Peter Stone. A neuro-evolution approach to general atari game playing. 2013.

[9] Nicolas Heess, David Silver, and Yee Whye Teh. Actor-critic reinforcement learning with energy-based policies. In *European Workshop on Reinforcement Learning*, page 43, 2012.

[10] Kevin Jarrett, Koray Kavukcuoglu, MarcAurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR 2009)*, pages 2146–2153. IEEE, 2009.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114, 2012.

[12] Sascha Lange and Martin Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8. IEEE, 2010.

[13] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[14] Hamid Maei, Csaba Szepesvari, Shalabh Bhatnagar, Doina Precup, David Silver, and Rich Sutton. Convergent Temporal-Difference Learning with Arbitrary Smooth Function Approximation. In *Advances in Neural Information Processing Systems 22*, pages 1204–1212, 2009.

[15] Hamid Maei, Csaba Szepesvári, Shalabh Bhatnagar, and Richard S. Sutton. Toward off-policy learning control with function approximation. In *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*, pages 719–726, 2010.

[16] Volodymyr Mnih. *Machine Learning for Aerial Image Labeling*. PhD thesis, University of Toronto, 2013.

[17] Andrew Moore and Chris Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.

[18] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*, pages 807–814, 2010.

[19] Jordan B. Pollack and Alan D. Blair. Why did td-gammon work. In *Advances in Neural Information Processing Systems 9*, pages 10–16, 1996.

[20] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005*, pages 317–328. Springer, 2005.

[21] Brian Sallans and Geoffrey E. Hinton. Reinforcement learning with factored states and actions. *Journal of Machine Learning Research*, 5:1063–1088, 2004.

[22] Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala, and Yann LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR 2013)*. IEEE, 2013.

[23] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[24] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[25] John N Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *Automatic Control, IEEE Transactions on*, 42(5):674–690, 1997.

[26] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

9

I wanted to take the code that was provided for us and make it far more robust. I did this by seperating the files into a main driver file, files with classes for the model and some seperate utility function files.

# %load atari/atari.py# %load atari/DQNetwork.py# %load atari/DQAgent.py# %load atari/evaluation.py

The output was getting unmanagable so I got lucky and stumbled upon some code from github that I could repurpose to help divide up the output and make it more readable.

# %load atari/Logger.py# %load atari/utils.py# %load atari/plots.py

After doing multiple troubleshooting tests to try and get a conv3d model to train I came to the conclusion that even a very simple model was just to large to train. Even using my VM with large amounts of memory I was not able to complete a training session with the following conv3d model.

Layer (type) Output Shape Param

conv3d_1 (Conv3D) (None, 32, 4, 84, 84) 8224

---

flatten_1 (Flatten) (None, 903168) 0

---

dense_1 (Dense) (None, 512) 462422528

---

dense_2 (Dense) (None, 18) 9234

Total params: 462,439,986 Trainable params: 462,439,986 Non-trainable params: 0

Because of this inability to to get a conv3d model to train I started looking for clever solutions and came across a paper by Google's DeepMind published in Nature that had a link to the code they used. They had written it in Lua so I had to translate there ideas to into Python.

lua_atari (Human_Level_Control_through_Deep_Reinforcement_Learning/dqn)

The basic idea is that you can still create a type of conv3d model passing multiple frames into a conv2d model. It allows you to pass the same amount of data and come up with a less memory hungry model in the end.

Just for fun I also followed the outline of Google's code and made my program so it could potentially load any of the openai gym atari games.

Smaller 2d CNN with Roughly 20,000 Sample Replay Buffer

Initially I ran my code on my laptop but was only able to fill the replay buffer with about 16,000-20,000 samples before I ran out of memory. I ran multiple training sessions doing my best to gradually decrease epsilon from 1 to 0.1. My model was a conv2d model with 3 hidden layers followed by one dense layer.

Layer (type) Output Shape Param

conv2d_1 (Conv2D) (None, 32, 26, 20) 8224

---

conv2d_2 (Conv2D) (None, 64, 12, 9) 32832

---

conv2d_3 (Conv2D) (None, 64, 10, 7) 36928

---

flatten_1 (Flatten) (None, 4480) 0

---

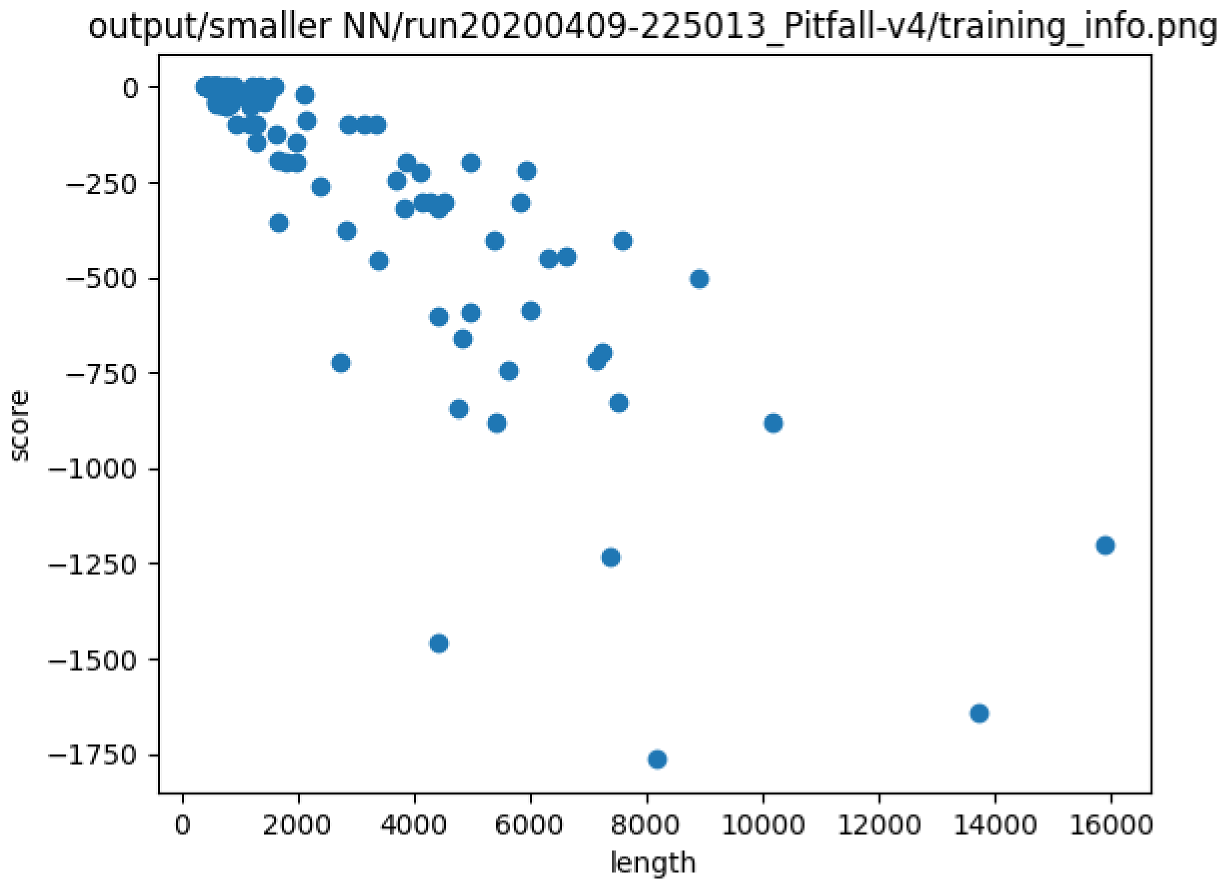
dense_1 (Dense) (None, 512) 2294272

---

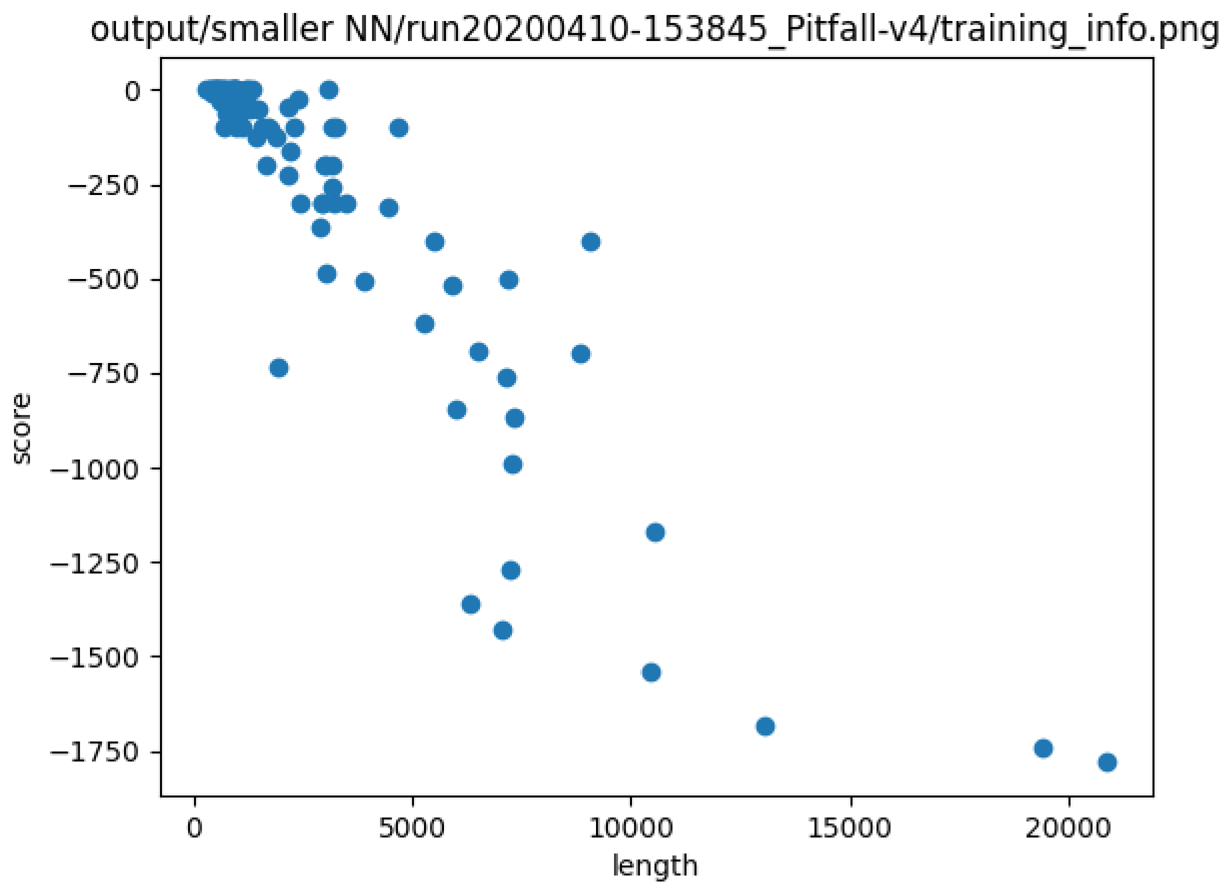
dense_2 (Dense) (None, 18) 9234

Total params: 2,381,490 Trainable params: 2,381,490 Non-trainable params: 0

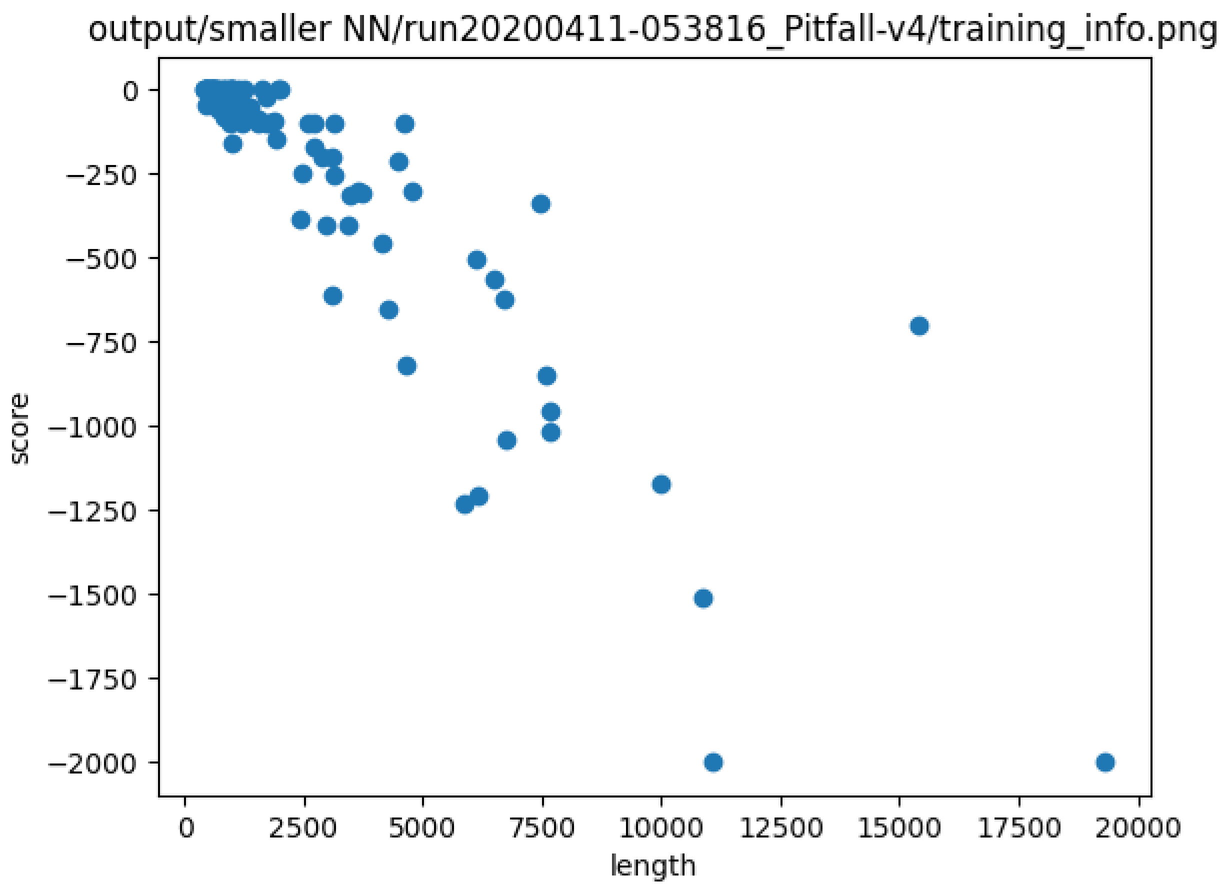
The output below plots the average score of a batch of samples with the number of moves made.



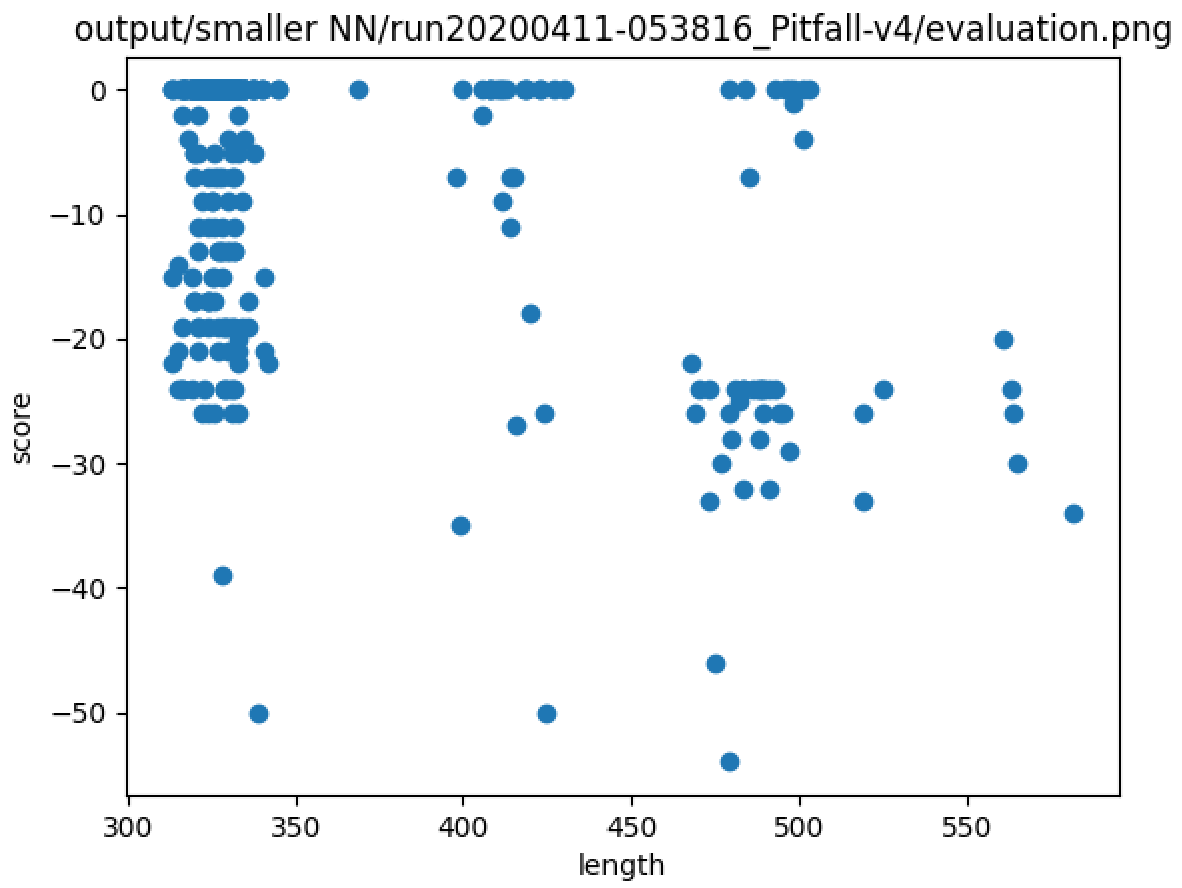
avg_score	avg_Q
-328.9562328111802	4.253121242812553



avg_score	avg_Q
-390.7353559290824	3.7946990955465365



After doing this training 5 times I ran my evaluation function which drops epsilon to 0.05 and got the following data:



avg_score	avg_Q
-412.5182987616657	5.083157772519628

0:00 / 2:08

Larger 2d CNN with 2,000 Sample Replay Buffer

I was still unsure about the cause of my errors at this point so I decreased the training buffer to 2,000 and increased the size of the hidden layers.

Layer (type) Output Shape Param

conv2d_1 (Conv2D) (None, 32, 26, 20) 8224

---

conv2d_2 (Conv2D) (None, 128, 12, 9) 65664

---

conv2d_3 (Conv2D) (None, 256, 10, 7) 295168

---

flatten_1 (Flatten) (None, 17920) 0

---

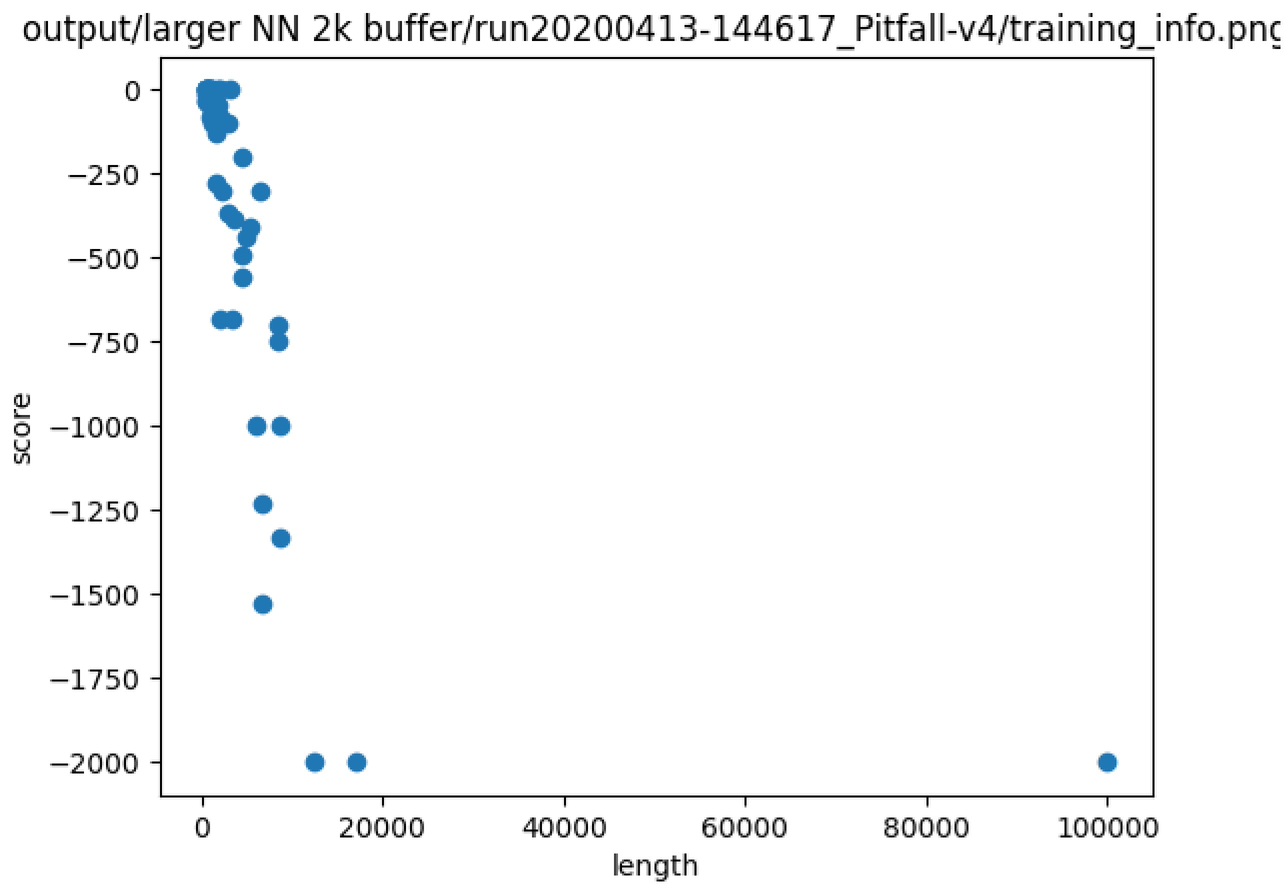
dense_1 (Dense) (None, 512) 9175552

---

dense_2 (Dense) (None, 18) 9234

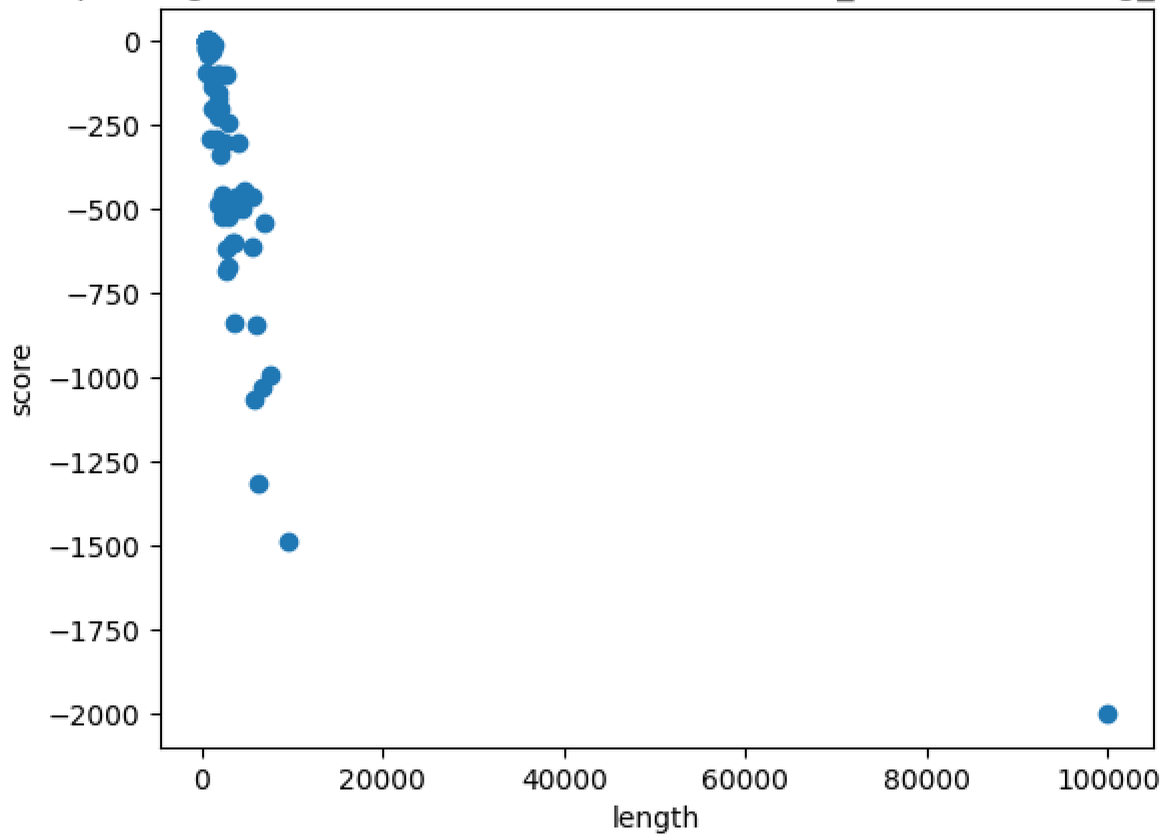
Total params: 9,553,842 Trainable params: 9,553,842 Non-trainable params: 0

I was able to train this model longer in between crashes with the following results:

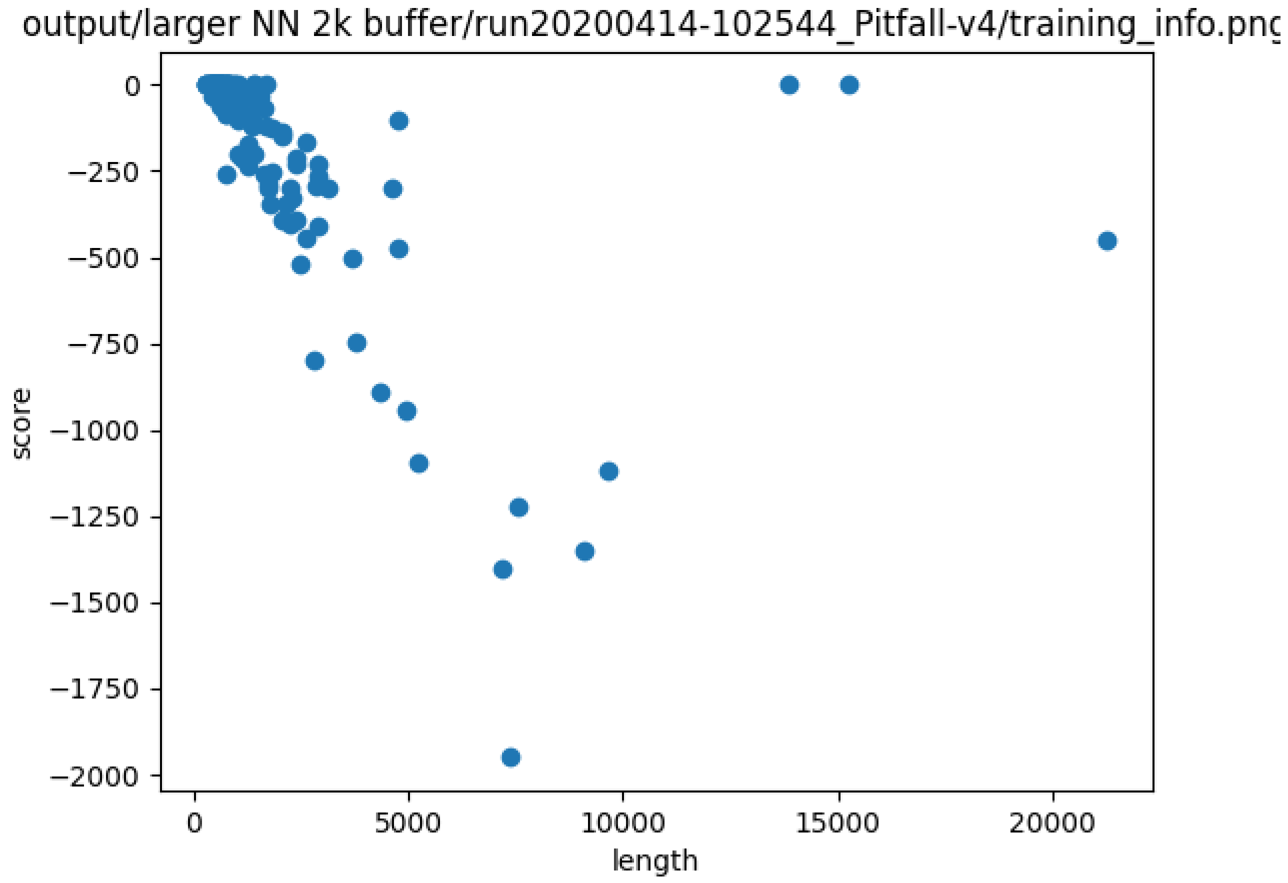


avg_score	avg_Q
-1018.7747097734613	4.732479361001398

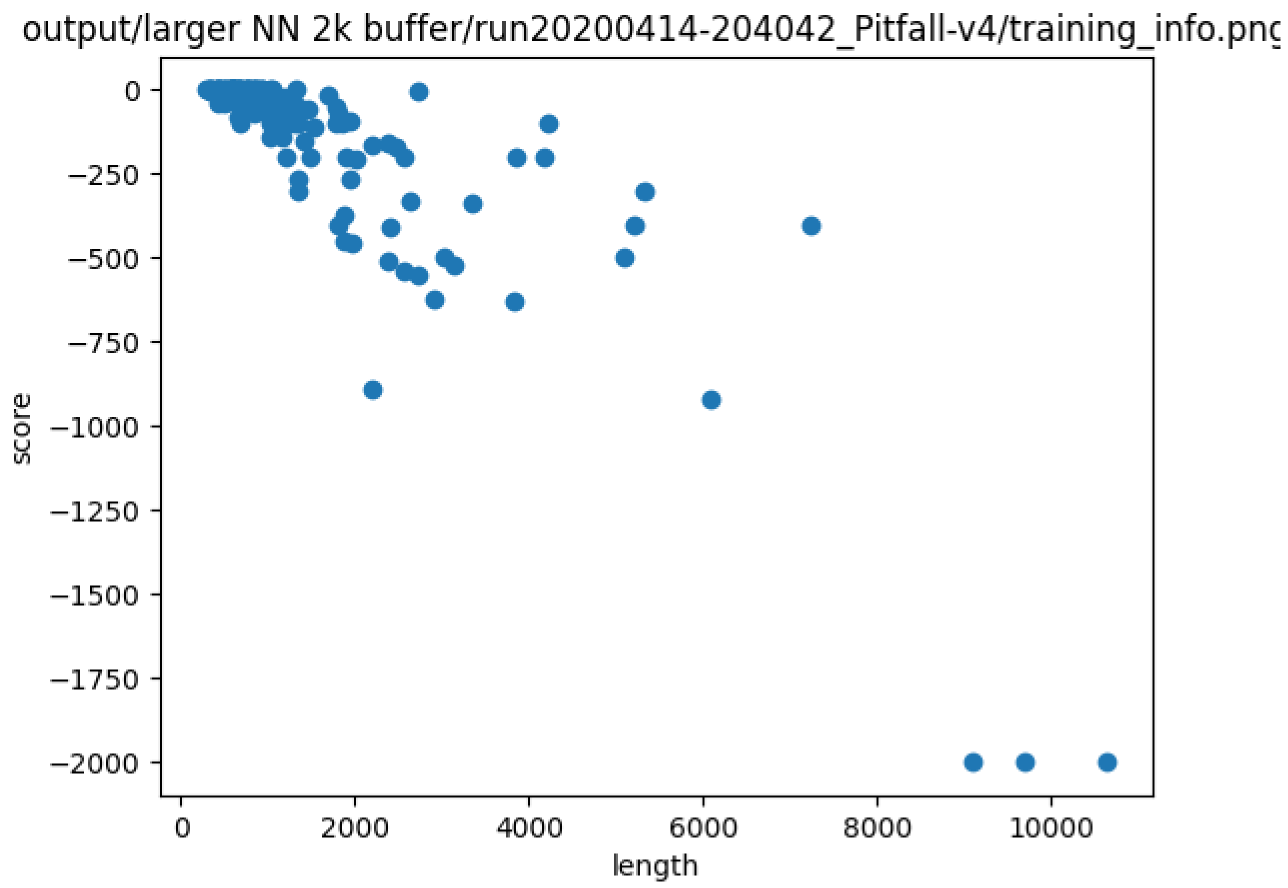
output/larger NN 2k buffer/run20200414-001658_Pitfall-v4/training_info.pnc



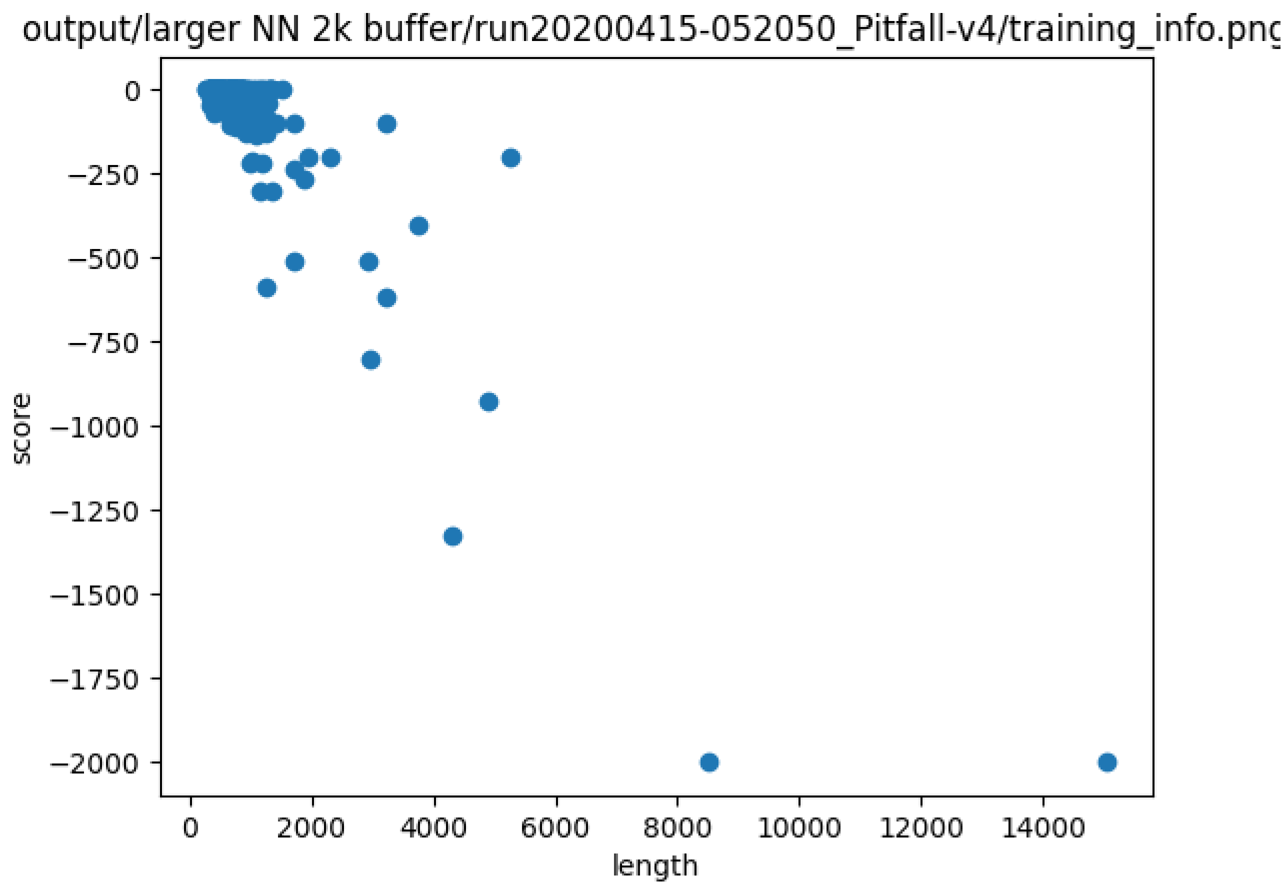
avg_score	avg_Q
-992.7576462300214	4.111783611505421



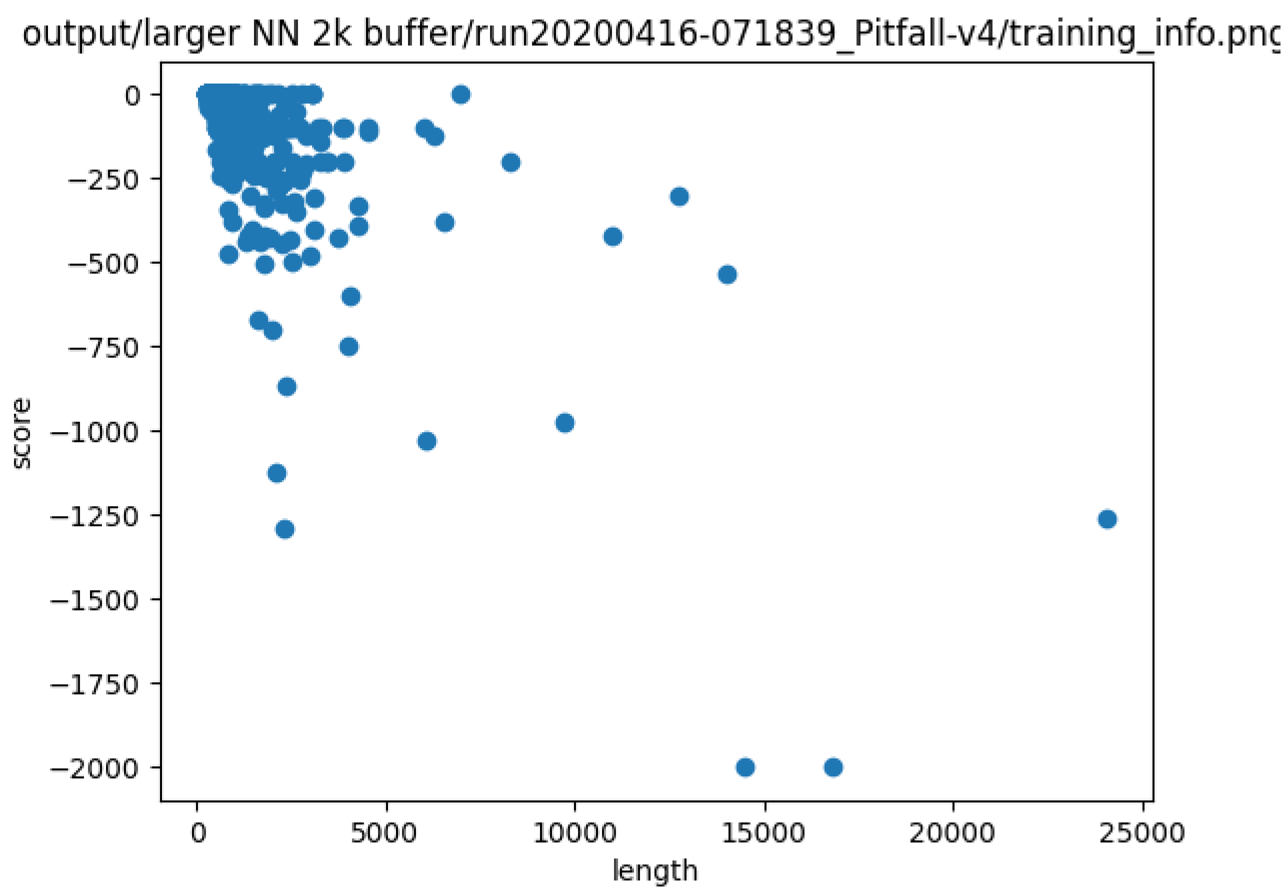
avg_score	avg_Q
-251.22486242720856	4.396723217694758

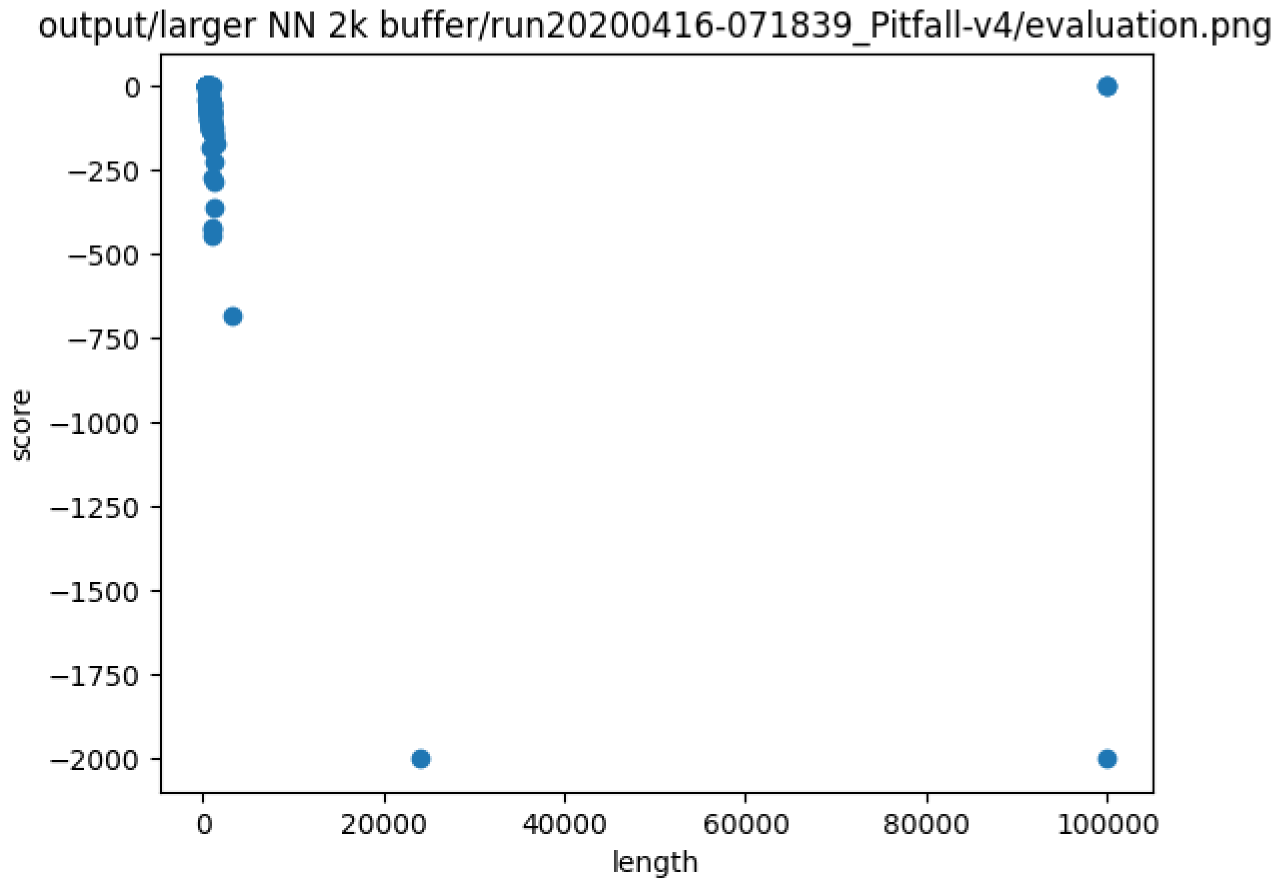


Data failed to save for this training session

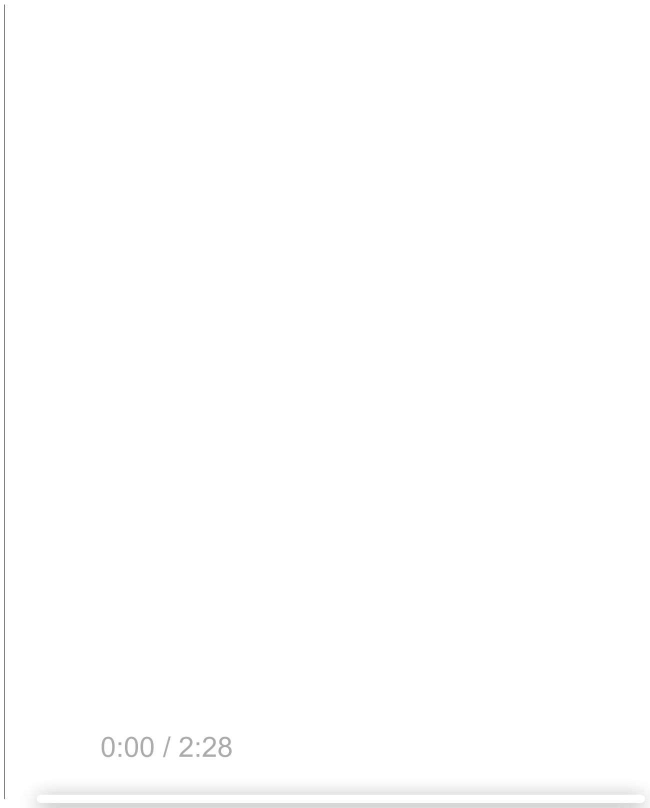


Data failed to save for this training session


output/larger NN 2k buffer/run20200416-071839_Pitfall-v4/training_info.png
score
length
0
−250
−500
−750
−1000
−1250
−1500
−1750
−2000
0
5000
10000
15000
20000
25000


output/larger NN 2k buffer/run20200416-071839_Pitfall-v4/evaluation.png
score
length
0
−250
−500
−750
−1000
−1250
−1500
−1750
−2000
0
20000
40000
60000
80000
100000

avg_score	avg_Q
-317.63224937571897	4.001707762392947
-46.76040052748487	3.5257336499018925
-38.541254108850026	5.154112006786231
-125.14807483173884	5.221036148929534



New Strategy

At this point I decided that I needed more memory in order to give this a really good chance at succeeding. I purchased 32 GB to put in my desktop but quickly found that without a more powerfull GPU I would be unable to complete even 1 training set. Using the more powerfull GPU in my laptop I was able to train 1 epoch about every .5 seconds. Using my desktop with it's less powerfull GPU it took about 1.5 seconds. My solution to this was to use the $300 google credit and create a vm that could handle it. In order to train an entire session with a 1 million sample replay buffer I had to spin up an 8 core 100 GB server with an Nvidia Tesla T4 GPU. Even with those specs it would still take over 5 days for me to fully train a model with 1 million samples.

```
In [28]: # Time it takes to train 1 million epochs
         ! cat time.log
         print("Hours: ", 7586 / 60)
```

```
aurvand+  5049  122 71.2 102943528 76440084 ?  Sl   Apr24 7586:10 pyt
hon3 ./atari.py -t -e Pitfall-v4
Hours:  126.43333333333334
```

I went back and retrained my original sized model using the larger 1 million sample replay buffer.

Layer (type) Output Shape Param
conv2d_1 (Conv2D) (None, 32, 26, 20) 8224

---

conv2d_2 (Conv2D) (None, 64, 12, 9) 65664

---

conv2d_3 (Conv2D) (None, 128, 10, 7) 295168

---

flatten_1 (Flatten) (None, 17920) 0

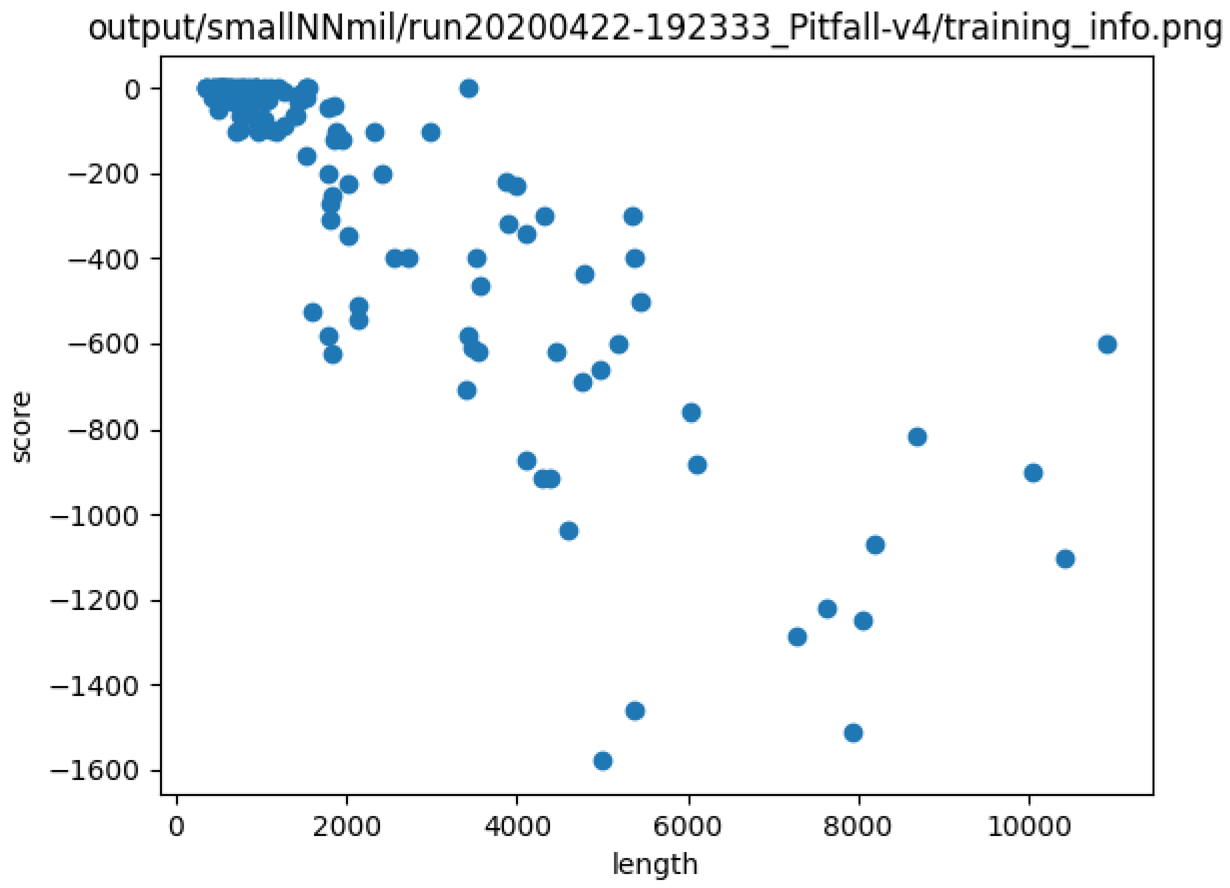
---

dense_1 (Dense) (None, 512) 9175552

---

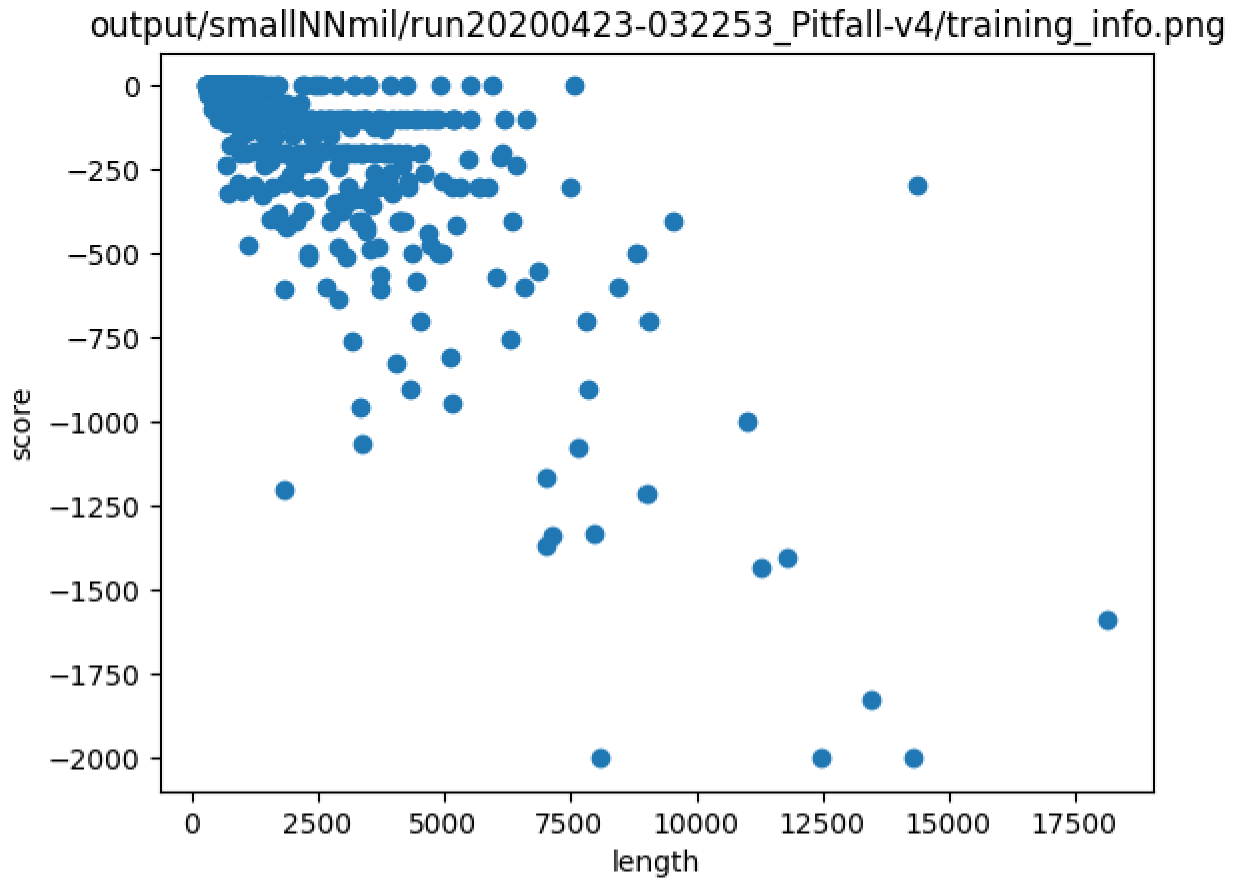
dense_2 (Dense) (None, 18) 9234

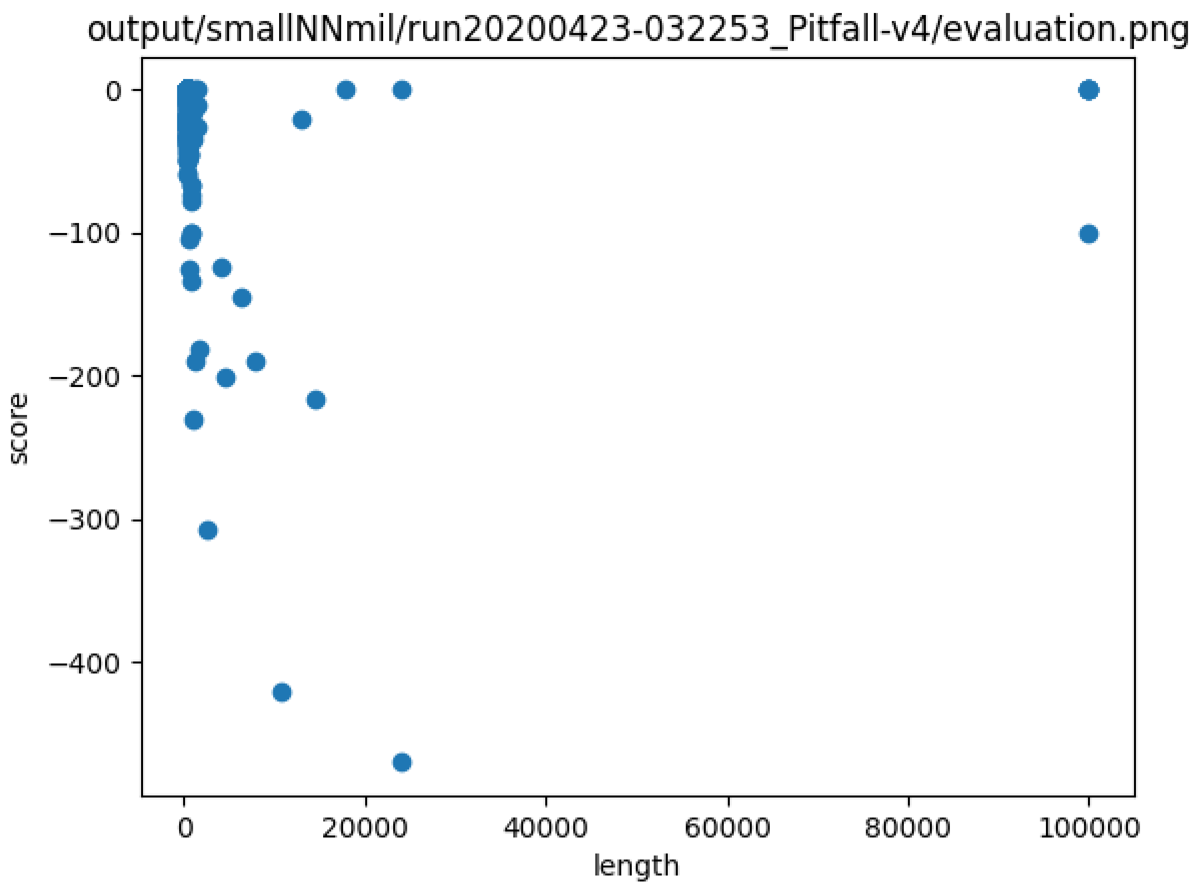
Total params: 9,553,842 Trainable params: 9,553,842 Non-trainable params: 0



avg_score	avg_Q

avg_score	avg_Q
-320.0560353629386	3.4940004595508096





avg_score	avg_Q
-378.4650823898666	3.6717535136462383
-319.22153977073634	4.340920725500992
-188.83790915782464	4.165799738619692
-114.38929478159412	4.39711780013387
-138.25828726515243	4.389452723363912
-115.3563883866638	4.519118165078188

0:00 / 1:39

I then trained another model that was larger than the last with 1 million samples.

Layer (type) Output Shape Param #

conv2d_1 (Conv2D) (None, 32, 26, 20) 8224

---

conv2d_2 (Conv2D) (None, 128, 12, 9) 65664

---

conv2d_3 (Conv2D) (None, 256, 10, 7) 295168

---

flatten_1 (Flatten) (None, 17920) 0

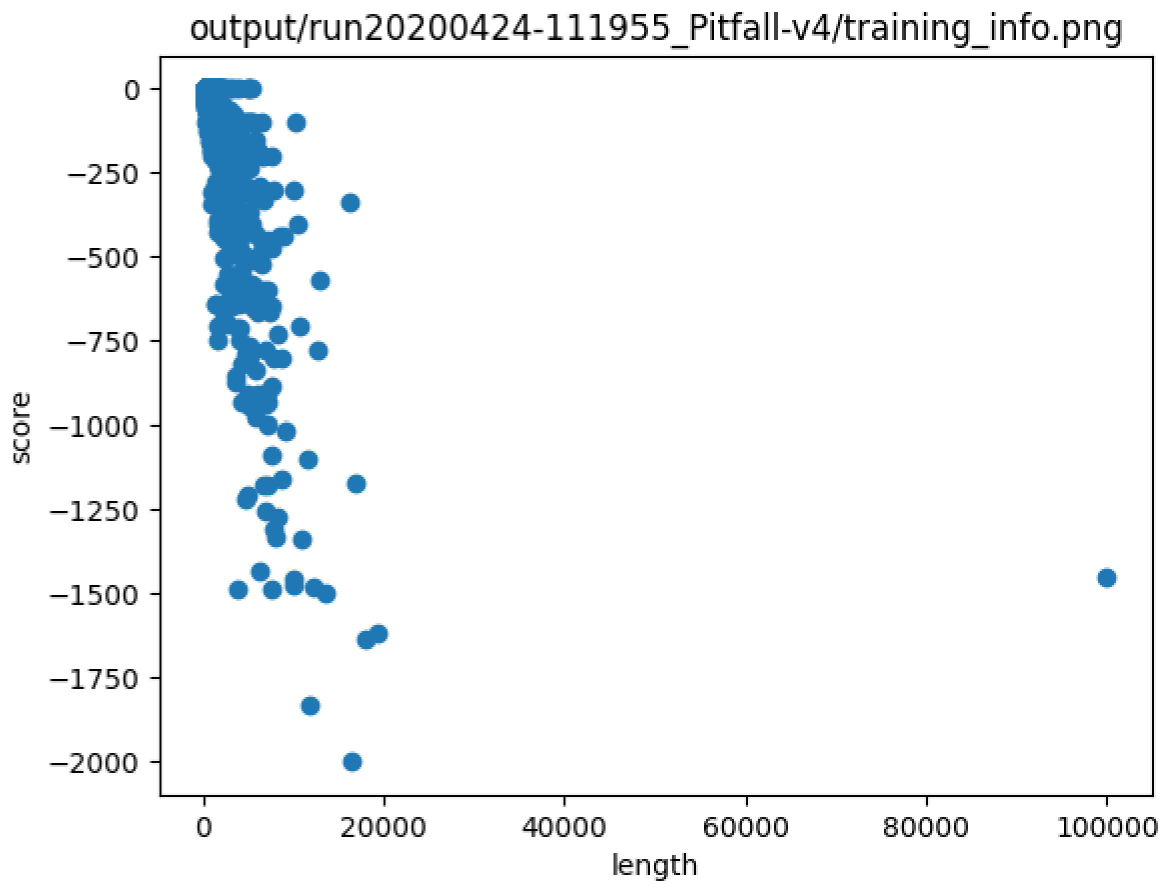
---

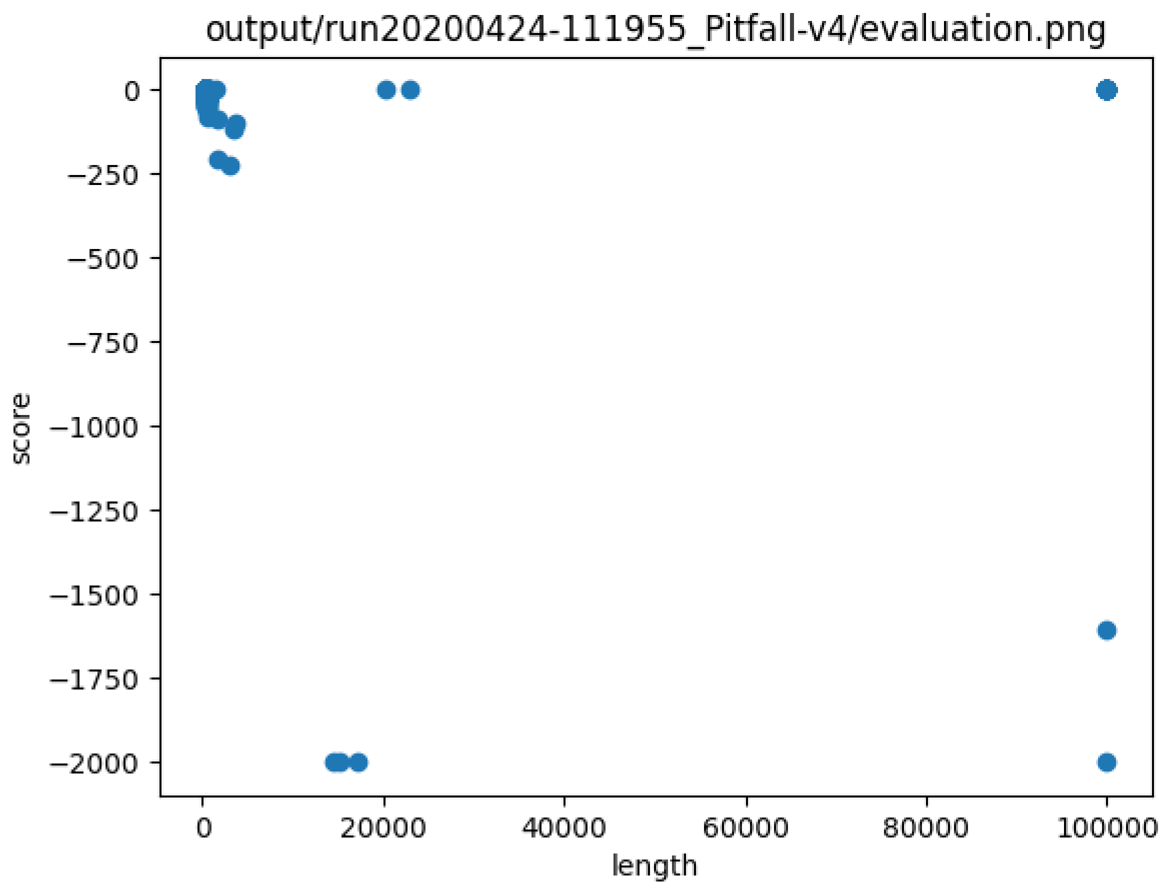
dense_1 (Dense) (None, 512) 9175552

---

dense_2 (Dense) (None, 18) 9234

Total params: 9,553,842 Trainable params: 9,553,842 Non-trainable params: 0





avg_score	avg_Q
-373.40908308342773	5.0460114708370885
-401.4379504684872	4.382644886165483
-234.6242172130532	4.420151734835599
-170.76304136569922	4.611473512657193
-149.20268753632482	4.1421130755162086
-153.6132927636196	4.558212537264004
-35.726420877044895	5.800859385230968
-43.177003438906766	4.980112685403583
-171.86959287531806	4.490612483769497
-345.6061252064255	4.395581889767593
-765.9674276224965	4.388082309987472
-175.58773940989778	4.282803310132605
-171.02147517261753	4.237126226864874
-155.628070649996	4.221333059280689
-253.2433757260164	4.3082387923738406

avg_score	avg_Q
-241.77820283004698	4.287316184724803
-137.1431312240768	4.3214661528507525
-389.77924561359595	4.360648439335574
-156.5556423663281	4.336246556380365
-219.60423769970612	4.321512595731582

0:00 / 2:28

Conclusion

I learned far more about "real life" machine learning doing this I would have learned from a normal final. It was fun to be able to develop solutions to a lot of the problems I found. I learned that finding the correct tools for the job is extremelly important but that finding and implementing those tools creates it's own challange.

There are a couple of things I would do different if I were continuing on with this exploration. For starters, I don't think a one million move replay buffer is large enough for Pitfall. I think given more resources I would have like to try a larger buffer, maybe two million. Another thing I thought to change after I ran out of time was to log if my agent ever recieved a positive reward. I'm pretty sure that my agents with the larger replay buffer did at some point find a piece of treasure because the closest treasure is found by going left and my two large replay buffer agents constantly choose to start out going left.