Deep Reinforcement Learning to play Space Invaders

Nihit Desai[*]
Stanford University

Abhimanyu Banerjee[*]
Stanford University

Abstract

In this project, we explore algorithms that use reinforcement learning to play the game space invaders. The Q-Learning algorithm for reinforcement learning is modified to work on states that are extremely high dimensional(images) using a convolutional neural network and is called the Deep-Q learning algorithm. We also experiment with training on states represented by the RAM representation of the state. We also look at an extension of Q-learning known as Double Q learning, explore optimal architectures for learning .

1 Introduction

Video games provide an ideal testbed for artificial intelligence methods and algorithms. In particular, programming intelligent agents that learn how to play a game with human-level skills is a difficult and challenging task. Reinforcement learning (Sutton and Barto 1998) has been widely used to solve this problem traditionally. The goal of reinforcement learning is to learn good policies for sequential decision problems by maximizing a cumulative future reward. The reinforcement learning agent must learn an optimal policy for the problem, without being explicitly told if its actions are good or bad.

Reinforcement learning approaches rely on features created manually by using domain knowledge (e.g. the researcher might study game playing patterns of an expert gamer and see which actions lead to the player winning the game, and then construct features from these insights). However, the current deep learning revolution has made it possible to learn feature representations from high-dimensional raw input data such as images and videos, leading to breakthroughs in computer vision tasks such as recognition and segmentation (Kaiming He et al. 2015 LeCun, Bengio, and Geoffrey Hinton 2015). Mnih et al. 2013 apply these advances in deep learning to the domain of reinforcement learning. They present a convolutional neural network (CNN) architecture that can successfully learn policies from raw image frame data in high dimensional reinforcement learning environments. They train the CNN using a variant of the Q-learning (Watkins and Dayan 1992) and call this network a Deep Q-Networks (DQN).

In this project, we train an agent to play Space Invaders, a popular Atari game using Deep Reinforcement Learning. Our project can be summarized as follows:

- Analyze how different representations of the input (image frame pixels v/s emulator RAM state) impact training time and performance.
- improvements to CNN architectures and to the Fully connected networks for the RAM states
- Experiments on applying dropout and Early stopping and learning how they affect the learned policy

2 Related Work

Reinforcement Learning: Reinforcement learning is an area of machine learning concerned with how an agent should act in an environment so as to maximize some cumulative reward. A software agent that learned to successfully play TD-gammon (Tesauro 1995) was an early example of research in this area. Q-learning is a model-free technique for reinforcement learning which can be used to learn an optimal action-selection policy for any finite MDP. Q-learning was first introduced by Watkins and Dayan 1992. However, it has been shown that Q-learning with

[*]equal contribution

non-linear function approximation is not guaranteed to converge (Tsitsiklis and Van Roy 1997)

Convolutional Neural Networks: Recent advances in deep learning, especially the use of convolutional neural networks (CNNs), have made it possible to automatically learn features representation from high-dimensional raw input data such as images and videos. Now, CNN-based models produce state-of-the-art performance in various computer vision tasks, including image classification (Krizhevsky, Sutskever, and G. Hinton 2012, K. He et al. 2015) and object detection (K. He et al. 2014). There has been considerable work in enhancing CNN components, including pooling layers (Zeiler and Fergus 2013) and activation units (Goodfellow et al. 2013) as well as training, including Batch Normalization (Ioffe and Szegedy 2015) and Dropout (Srivastava et al. 2014).

Deep Q-Learning: An early inspiration for using neural network architectures for reinforcement learning was presented by Riedmiller 2005. This work uses neural fitted Q-learning (NFQ) and updates parameters of the Q-network using the RPROP algorithm. Mnih et al. 2013 present a convolutional neural network (CNN) architecture that can successfully learn policies from raw image frame data in high dimensional reinforcement learning environments. They train the CNN using a variant of the Q-learning, hence the name Deep Q-Networks (DQN). Van Hasselt, Guez, and Silver 2015 recently extended this work by incorporating Double Q-Learning which addresses the overestimation problem (Q-learning algorithm is known to overestimate action values since the same set of parameter weights are used to select an optimal action as well as evaluate the Q-value resulting from this selection). More recently, DeepMind has achieved a breakthrough in using deep reinforcement learning, combined with tree search, to master the game of Go (Silver et al. 2016).

3 Task definition

We consider the task of learning to play Space Invaders, a popular Atari console game. Formally, an agent interacts with an environment E in a sequence of actions, observations and rewards. In our case the environment is the Atari emulator for Space Invaders. The game play is discretized into time-steps and at each time step, the agent chooses an action a_t from the set of possible actions for each state $A = \{1, 2, ...L\}$. The emulator applies the action to the current state, and brings the game to a new state, as shown in figure 1. The game score is updated the a reward r_t is returned to the agent. We formalize this problem as follows:

- State s: A sequence of observations, where an observation, depending on the state space we are operating in, is a matrix representing the image frame or a vector representing the emulator's RAM state.
- Action a: An integer in the range of $[1, L]$. In case of Space Invaders $L = 6$ and the actions are $\{$FIRE (shoot without moving), RIGHT (move right), LEFT (move left), RIGHTFIRE (shoot and move right), LEFTFIRE (shoot and move left), NOOP (no operation)$\}$.
- Reward r: Reward returned by the environment, clipped to be in the range $[-1, 1]$.

Our task is to learn a policy for the agent to enable the agent to make 'good' choice of action for each state.

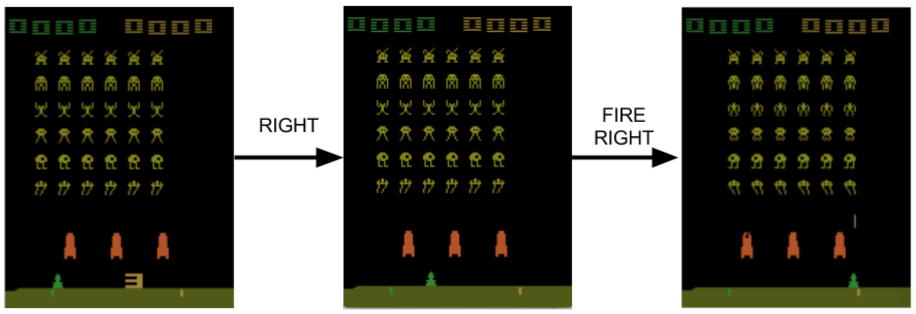


Figure 1: An example of state, action sequence when playing SpaceInvaders

4 Approach

In this project, we use the Atari emulator open sourced by OpenAI (Brockman et al. 2016), the OpenAI gym. This emulator provides with two possible environments to play space invaders . The First environment represents each state by a raw RGB image of the screen which has a size $(210, 160, 3)$ (SpaceInvaders-v0), the second environment

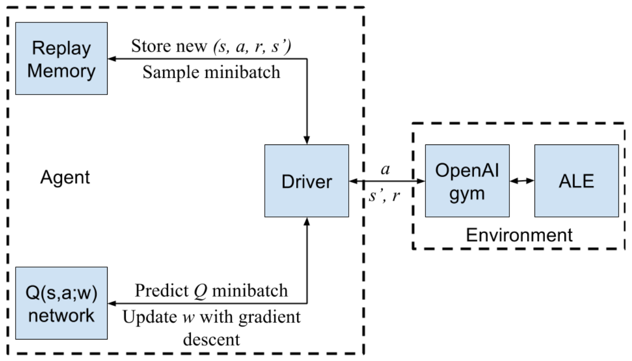


Figure 2: An overview of the Environment and Agent for this Reinforcement Learning task

represents the state by a 128 byte RAM state of the game (SpaceInvaders-ram-v0). As mentioned in the previous section, our task is to learn a policy for the agent so that the agent can play the game. In order to accomplish this task, we use reinforcement learning. However, learning from high dimensional representation of states such as images is difficult for traditional reinforcement learning techniques with handcrafted features. Instead, we use an approach similar to that of Mnih et al. 2013 which uses DQNs and extend their approach in a few ways listed in Section 1. In this section, we introduce the concept of Deep Q-Networks, experience replay and double Q-learning all of which are important foundational blocks of our approach. An overview of our approach is shown in Figure 2.

4.1 Q-Learning

Given a sequence of state, actions, rewards $s_1, a_1, r_1, s_2, a_2, r_2 ..$, we want to learn the optimal strategy to play the game. The goal of the agent is to learn a strategy to maximize future rewards. We assume that the rewards are discounted by a factor of γ at every time step and define the future discounted return as $R_T = \Sigma_{t=0}^{t=T} \gamma^t r_t$ We also define the optimal action value function $Q_{opt}(s, a)$ to be the reward we get on following the optimal policy and starting in state s and playing action a Given a Markov Decision Process with transition probabilities $t(s, a, s^{'})$ and rewards $r(s, a, s^{'})$, the Q-function satisfies a recurrence known as the **Bellman equation**:

$$Q_{opt}(s, a) = \sum_{s'} T(s, a, s^{'})(r(s, a, s^{'}) + \gamma V_{opt}(s^{'})) \tag{1}$$

However in the setting of reinforcement learning, we do not know the transition probabilities and rewards for each transition. We try to learn the optimal strategy by minimizing the squared loss function $\sum_{s,a,r,s'} (Q_{opt}(s, a) - (r + \gamma V_{opt}(s^{'})))^2$. In practice, this equation does not generalize to unseen states and actions, so we use a function approximation to estimate the Q-function. It is common to use a linear approximator in traditional RL approaches where we estimate $Q_{opt}(s, a, w) = w.\phi(s, a)$. We still minimize the same loss function but now use gradient descent to learn the weights-w The update equations are:

$$w \leftarrow w - \eta(Q_{opt}(s, a) - (r + \gamma V_{opt}(s^{'})))\phi(s, a) \tag{2}$$

4.2 Deep Q-Networks

If we have very high dimensional data such as an image, it is hard to extract features manually because it is not obvious what the features are that are relevant to extract from the image. To handle high dimensional inputs where the agent learns from raw inputs such as image frames, Mnih et al. 2013 introduced Deep Q-Networks which uses a convolutional neural network to approximate the Q-function. If the state is n dimensional and the number of actions is m then the CNN is a mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$. We refer to a neural network function approximator with weights θ_i as a Q-network. The objective is to now learn the weights θ_i Q-network.

$$\theta_i \leftarrow \theta_i - \eta(Q_{opt}(s, a, \theta_i) - (r + \gamma V_{opt}(s^{'})))\nabla_{\theta_i} Q_{opt}(s, a, \theta_i) \tag{3}$$

We can also use the same approach of using a neural network as a function approximator for the Q function when we use the 128 byte representation of the state. While it is not evident a-priori to us what each byte represents, we can nontheless use a fully connected neural network to represent the Q-function. Since the state is not an image this time, we do not use a CNN for the function approximator. We feel it is necessary to use a fully connected neural network to approximate the Q-function as there needs to be connections between every two bytes representing the state. Moreover, we can consider the 128 byte representation as a feature vector, which allows us to drastically simplify our network architecture as will be discussed in Section 5.

4.3 Double Q Learning:

The usual setup of Q-learning looks similar to gradient descent and the updates try to bring the current value of $Q(s, a)$ closer towards a target value $Y_t^Q = r + \gamma V_{opt}(s^{'})$ as mentioned in Eq.3. The quantity $V_{opt}(s^{'}) = \max_{a^{'}} Q(s^{'}, a^{'})$, where the Max operator in Q-learning and DQN is used to select the action as well as evaluate the Q-function at the same weights (θ). In a recent work, Van Hasselt, Guez, and Silver 2015 show that this makes it more likely to overestimate the values, resulting in overoptimistic value estimates. The Double Q learning algorithm gets around this by keeping two sets of weights θ and $\theta^{'}$ For each update, one set of weights is used to determine the greedy policy and the other to determine its value. For a clear comparison we first rewrite the target in Q-learning as :

$$Y_t^Q = r + \gamma Q(s^{'}, \mathrm{argmax}_{a^{'}} Q(s^{'}, a^{'}, \theta), \theta) \tag{4}$$

The Double Q learning target can be written as :

$$Y_t^Q = r + \gamma Q(s^{'}, \mathrm{argmax}_{a^{'}} Q(s^{'}, a^{'}, \theta), \theta^{'}) \tag{5}$$

Notice that the argmax is over the set of weights θ while the evaluation of the value of the policy is done with another set of weights $\theta^{'}$ The second set of weights can be updated periodically by evaluating and setting them equal to the weights θ. We experimented training our DQNs with double Q-learning and without it, and found that double Q-learning indeed improves performance.

4.3.1 Experience Replay

A technique called experience replay is used in Deep Q learning to help with convergence. At every time step we store the agents observations $(s, a, r, s^{'})$ into an replay memory $\mathcal{D}$. We then randomly sample a minibatch of observations from the replay memory $\mathcal{D}$ and use this minibatch to train the network (update weights with backpropagation). Learning directly from consecutive samples is inefficient as there could be strong correlations between samples. Randomizing the training by choosing random samples will help with this problem by smoothing the training distribution over many past behaviors.

4.3.2 Frame Skipping

Following the approach of Mnih et al. 2013, we also use a simple frame-skipping technique. During training, the agent sees and chooses optimal actions actions on every kth frame instead of every frame, and its last action is repeated on skipped frames. This works because the game state does not change much in every frame. Skipping frames allows the agent to play roughly k times more episodes for the same time/computational resources. We experimented with $k = 3$ and $k = 4$ in our experiments.

4.4 Exploration Policy

We want to choose an exploration policy to ensure that the model learns from a wide range of states and actions in the domain space. A common policy that balances exploration and exploitation is the ϵ greedy policy. We choose the optimal policy with a probability $1 - \epsilon$ and the random policy with probability ϵ The value of ϵ is set to change linearly from 1.0 to 0.1 over the course of training, ie. We initially only explore while during the later stages of training we choose the optimal policy calculated more often. During the testing phase we use a value of $\epsilon = .05$.

5 Data and Experiments

We use the Atari emulator open sourced by OpenAI (Brockman et al. 2016), the OpenAI gym, to run our reinforcement learning experiments. The agent interacts with the environment in a sequence of (actions, observations, rewards) tuples. In our case the environment is the Atari emulator for Space Invaders. In this project, we have run two different kinds of experiments, corresponding to the two state spaces - the pixel(image) representation (where a game state is a sequence of image frames), and the RAM state representation (where a game state is a sequence of emulator's RAM states). Across both these types of experiments, we use the same learning algorithm, policy and training pipeline and only vary the network architecture, to allow a fair comparison of results.

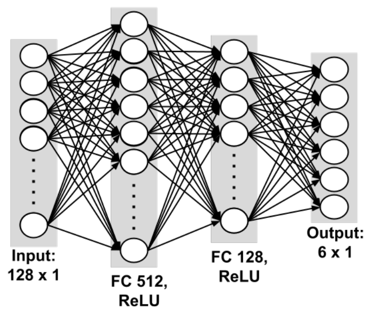
5.1 Preprocessing and Training Details:

Preprocessing: When working directly with raw pixels, the image frames are 210x160, with a 128 color palette. We preprocess the image frames from the emulator to convert them to grayscale images, normalize the intensity values to be between 0 and 1, and downsample them to a size of 84x84 pixels. When working with emulator RAM states, no such preprocessing is necessary. Each such (optionally preprocessed) output of the Atari emulator is an observation. As in Mnih et al. 2013, we stack the last 4 observations of a history to form a game state. We clip rewards to be in the range $[-1, 1]$. This is done to make sure that the magnitude of gradients during backpropagation is bounded

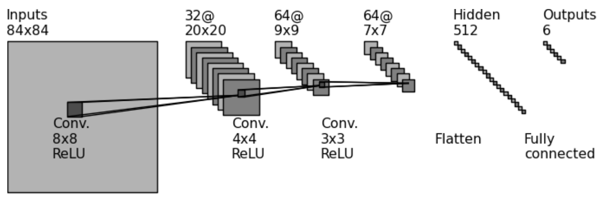
Hyperparameters: In order to train the networks, we tried the Adam algorithm (Kingma and Ba 2014) and the RMSProp algorithm (Tieleman and G. Hinton 2012) both with minibatches of size 32 and learning rate of 0.0002. The policy behavior during training was ϵ - greedy with ϵ annealed linearly from 1 to 0.1 over the duration of the training. We experimented with replay memory of varying sizes, from 100K to 1M but saw little difference in running time or performance of the model. Finally, we decided to go with a replay memory size of 1M . We use a discounting γ value of 0.99.

Training: As mentioned above, we use Adam and RMSProp, both of which are variations of stochastic gradient descent but usually allow the models to learn much faster in practice. As discussed in Mnih et al. 2013, we also use a simple frame-skipping technique where the agent sees and selects actions on every 4th frame instead of every frame, and its last action is repeated on skipped frames. We train our models for the RAM state space for 1M steps and our models for the image frame state space for 2M steps. For both experiment types, we checkpoint the model every 100K steps as a way to implement early stopping.

Infrastructure: Our models are implemented using Keras, with Tensorflow (Martín Abadi et al. 2015) backend and trained on a g2.2xlarge GPU instance (NVIDIA Tesla K40 GPU) on Amazon Web Services.



(a) 3-layer fully connected neural network we used for the RAM state space



(b) 5-layer convolutional neural network we used for the pixel (imageframe) state space

Figure 3: Network architecture of some of the models we used for our experiments

5.2 Experiments on the RAM state space

Atari 2600 uses a 128 byte RAM to for its internal representation of the game state. This compact representation is supposed to contains all information describing the game current game state. This relatively low dimensional representation is attractive for reinforcement learning. When working in this state space, we experimented with four different model architectures:

- **Linear model**: A simple linear model, which approximates the Q-function as a linear combination of the 128 features and has 774 parameters.
- **2-layer fully connected network**: A fully-connected neural network with 1 hidden layer of 512 units. This model has about 69K parameters.
- **3-layer fully connected network**: Two variations of a 3-layer neural network, one with hidden layers of 256 and 128 dimensions and another with hidden layers of 512 ans 128 dimensions.

Performance of these models is shown in Table 1 and the network architecture is shown in 3. We see that relatively simple models do pretty well in the low dimensional state space, with the linear model scoring an average of 323 points per episode. This is already well above the baseline, and the FC-3 models do better still. The RAM state is lower dimensional and has a more compact representation of the game state which makes it easier to learn from.

Model $\downarrow$ Metrics $\rightarrow$	Num. parameters	Best episode	Avg. of test episodes
Linear	774	840	323
FC-2 [512]	69K	835	389
FC-3 [256, 128]	69K	745	357
FC-3 [512, 128]	132K	830	394

Table 1: The experiment results on states represented RAM state of the game. Test time mean reward per episode, and reward of best episode by running the ϵ - greedy policy with $\epsilon = 0.05$. All models use double Q-learning unless specified otherwise.

5.3 Experiments with pixel state space

. As mentioned in Section 5.1, after processing the dimensionality of our states is 84x84x4. In this state space, we have experimented with 4 types of network architectures so far:

- **5 layer Convolutional Neural Network**: 3 convolutional layers and 2 fully-connected layers. Architecture of this network is shown in Figure 3. The number of filters in the three convolutional layers are 32 (8x8, with stride 4), 64 (4x4 with stride 2) and 64 (3x3 with stride 1) and use a ReLU non-linearity. The hidden layer has 512 units. Total number of network parameters are 1.6M.
- **6 layer Convolutional Neural Network**: 4 convolutional layers and 2 fully-connected layers. Architecture of this network is similar to the one shown in Figure 3. The number of filters in the four convolutional layers are 32 (4x4, with stride 2), 32 (4x4 with stride 2), 64 filters (4x4 with stride 1) and 64 filters (3x3 with stride 1) and use a ReLU non-linearity. The hidden layer has 256 units. Number of network parameters is 3.3M.
- **Dropout**: Versions of the above architectures with dropout.

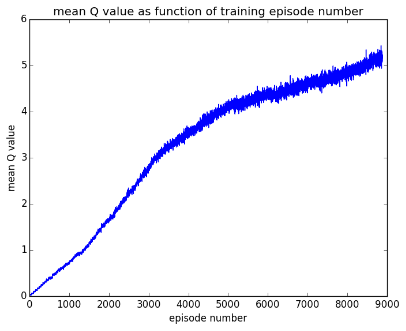
Performance of these models is shown in Table 2. We observe that while the models are much more complex compared the ones used with RAM state space (they have roughly 100x more parameters), the performance of the models quite comparable, even slightly better. The need for more complex models is obvious given that the state space is much larger, and we are working with raw inputs (pixels or image frames). This means the model needs to learn to extract good features from the image. The best performing model gives an average score of 420 after 2M training steps. One point to note is that the performance improvement observed in both the network architectures while going from 1M steps to 2M steps - for example, the average score for CNN-5 went up from 350 to 414. One results which initially surprised us is that adding dropout significantly degraded performance (for example, the performance of CNN-5 at 1M steps with dropout was less than 200). As a result, we omit these results from Table 2 and discuss these results in Section 6.

Model $\downarrow$ Metrics $\rightarrow$	Num. parameters	Best episode	Avg. of test episodes
CNN-5, 1M steps	1.6M	820	350
CNN-5, 2M steps	1.6M	810	414
CNN-6, 1M steps	3.3M	730	310
CNN-6, 2M steps	3.3M	850	420

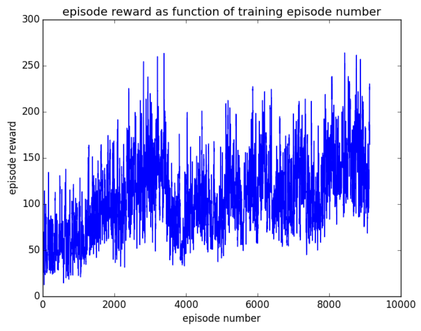
Table 2: The experiment results on states represented as pixels. Test time mean reward per episode, and reward of best episode by running the ϵ - greedy policy with $\epsilon = 0.05$.

6 Discussion and Analysis

We have performed a number of experiments which can broadly be classified as experiments on using a pixel representation of the state and experiments that use a RAM representation of the state. In this section we share some insights we gained while doing these experiments.



(a) Mean Q value as function of training episode (CNN-5 2M steps)

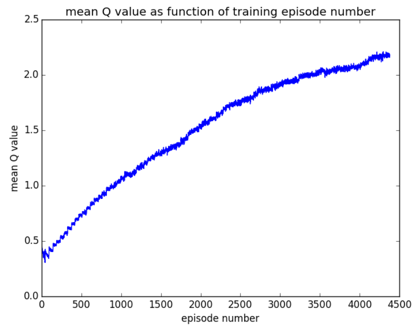


(b) Episode reward as function of training episode (CNN-5 2M steps)

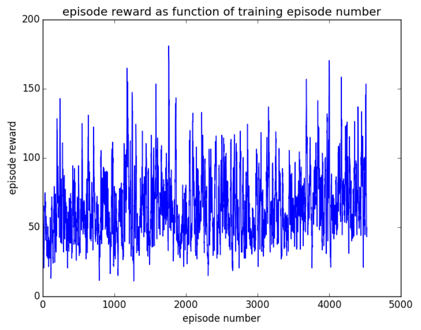
Figure 4: Mean Q value and episode reward as function of episode during training, for the state space of image frames

6.1 Evaluting Training progress:

In reinforcement learning, however, accurately evaluating the progress during training can be challenging. Our evaluation metric is the total reward that the agent collects in an episode (in our case, the final score). We see that the average total reward metric is fairly noisy for both types of experiments because small changes to the weights of a policy can lead to large changes in the states that the policy visits. Therefore, in parallel, we also track the Q-value of the best action at each step during training. This provides an estimate of how much discounted reward the agent can obtain by following its policy from any given state. These tend to be stable and show relatively smooth improvement as the training proceeds. These quantities are plotted as a function of the episode number in Figures 4 and 5.



(a) Mean Q value as function of training episode (FC-3 1M steps)



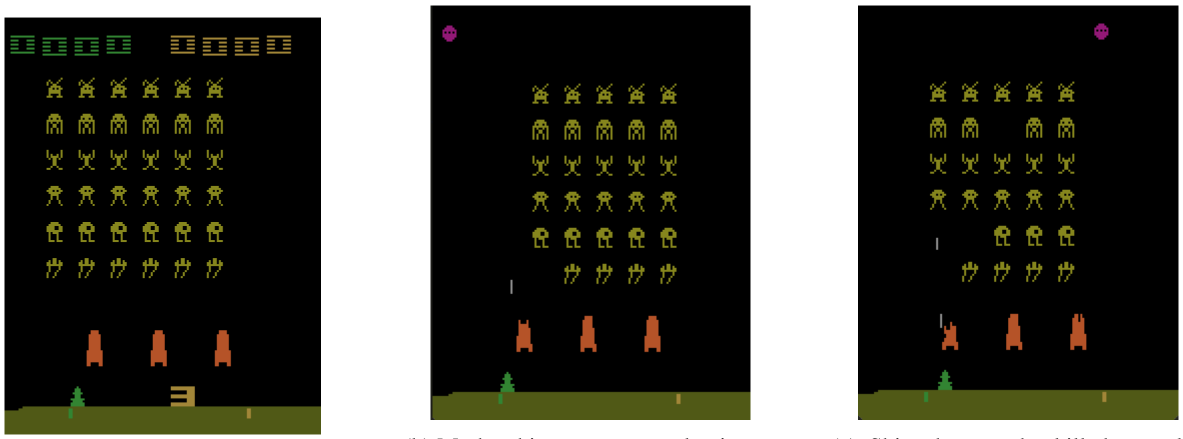
(b) Episode reward as function of training episode (FC-3 1M steps)

Figure 5: Mean Q value and episode reward as function of episode during training, for the RAM state space

6.2 The effect of dimensionality of the state

We have explained how the RAM state is a much lower dimensional representation of the game state than the pixel representation.The models we used for the RAM state have 69K free parameters , while the models used with the pixel representation have 2-3M free parameters This definitely impacts the training a lot. We have observed that the models with the RAM state train very quickly and start showing good performance after just about 200,000 training steps of gradient descent(with Adam optimizer). The models with pixel state representation typically take 2M steps to train.

Dimensionality, and the difference in state extracted features also leads to fairly different policies learned and this is evident when we observe the two agents play a game. For example, all the models with the RAM state space **learned to target the mothership** as it was flying by to get a very high bonus. The pixel represented state models did not learn this policy as we observed from gameplay. This is possibly because this information needs to be teased out



(a) Start of the game: q-value is high, since potential future reward is large)

(b) Mothership appears: q-value jumps by 82% because shooting the mothership can give a huge boost to the score)

(c) Ship about to be killed: q-value drops to nearly 0, or slightly negative when player's ship is about to be killed)

Figure 6: cap cap

of the raw pixel input with a specific set of features which may not be learned by the neural network. The simpler representation of the state enabled us the RAM state models to learn this policy.

6.3 Early stopping

An interesting observation with the RAM state models was related to early stopping. Firstly, we observe that lower dimensionality and relatively few parameters (on the order of 100s or 1000s) means the models train much quicker compared to pixels state - we see good performance after 200K to 250K training steps. However, we also observed that when we train these models for north of 1M steps, we see a degradation in performance of the final model weights. This is possibly because of overfitting, which we address by early stopping.

The pixel represented states take longer to train and take about 1M steps before they learn a reasonable policy. We saw that early stopping is useful for training on these states too, although the extent of degradation is much smaller. For example, in case of CNN-5, the model weights after training for 1.75M steps give a better performance during testing compared to the weights after training for 2M steps.

6.4 The effect of adding dropout

We mentioned in the previous section that early stopping is necessary to get a well trained policy. We initially guessed that maybe we are overfitting because of insufficient regularization and perhaps adding dropout might help in training the models. However we saw that dropout strictly worsened gameplay in both kinds of states. The average score after 10 games(during the test phase) after dropout was lesser , in the range of 100-150 for the models that were pixel based.(scores are 300+ without dropout)

The case for dropout adversely impacting performance in RAM state space is more clear - the 128 dimensional vector is already a compact low dimensional feature representation, and dropping random units from this means that useful information about the state is not conveyed to the model. Even for pixel based models, our CNNs are relatively simple in architecture (with just 5 or 6 layers), which means dropout is less useful as a means to prevent overfitting.

6.5 Interpreting the Q values

: We observe that frequently, big jumps and drops in predicted value of a state, action pair intuitively correspond to events in the game that we would think of as good or bad respectively. For example, predicted value jumps after the mothership appears on the left of the screen, and just before the mothership is about to be hit. Similarly, the value drops just before the player is about to be hit . Some examples of this are shown in Figure 6

7 Conclusion and Future work

We have done different experiments that show that the Deep-Q learning algorithm can learn from gameplay even with very high dimensional state representations. We experimented with architecture and found that our larger models have learned better policies in general(for the pixel states the CNN-6 learned the best policy and for RAM states the FC-3 learned the best policy) More training does help in learning better policies but one must keep checkpoints of the weights stored every 100K or so steps because early stopping seems to be crucial in learning a good policy. Dropout was found to hinder performance in learning a good policy rather than help.
It seems that training a Deep-Q learning algorithm well is quite difficult and seems to depend a lot on choosing a good explore vs exploit policy. In the future we might want to sample states from the memory which would be more beneficial ie. have a large difference between the current and target Q-values. Another aspect which we wanted to try out but did not get the time to is that of policy gradients and compare it to the DQN algorithm.

References

[Bro+16] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[Goo+13] I. Goodfellow et al. "Maxout networks". In: *ICML* (2013).

[He+14] K. He et al. "Spatial pyramid pooling in deep convolutional networks for visual recognition". In: *ECCV* (2014).

[He+15a] Kaiming He et al. "Deep residual learning for image recognition". In: *arXiv preprint arXiv:1512.03385* (2015).

[He+15b] K. He et al. "Deep Residual Learning for Image Recognition". In: *in ArXiv technical report arXiv:1512.03385* (2015).

[IS15] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *JLMR* (2015).

[KB14] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[KSH12] A. Krizhevsky, I. Sutskever, and G. Hinton. "ImageNet classification with deep convolutional neural networks". In: *in NIPS* (2012).

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `http://tensorflow.org/`.

[Mni+13] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[Rie05] Martin Riedmiller. "Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method". In: *European Conference on Machine Learning*. Springer. 2005, pp. 317–328.

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[Sil+16] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[Sri+14] N. Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *JLMR* (2014).

[Tes95] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[TH12] T. Tieleman and G. Hinton. "RMSProp: Lecture 6.5 - rmsprop, COURSERA Neural Networks for Machine Learning". In: (2012).

[TV97] John N Tsitsiklis and Benjamin Van Roy. "An analysis of temporal-difference learning with function approximation". In: *IEEE transactions on automatic control* 42.5 (1997), pp. 674–690.

[VGS15] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double Q-learning". In: *CoRR, abs/1509.06461* (2015).

[WD92] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[ZF13] M. D. Zeiler and R. Fergus. "Stochastic pooling for regularization of deep convolutional neural networks". In: *ICLR* (2013).