Training Game-playing Agents through Deep Q Learning

Valliappa Chockalingam, Hongyuan Ji, Nan Wu
Department of Electrical Engineering and Computer Science
University of Michigan
valli,hongyji,wunan@umich.edu

Abstract

Using reinforcement learning successfully in domains with high dimensional sensory input in a model-free manner poses a difficult task; agents must be able to both derive efficient representations of the input and generalize from past experience to new experiences. In this paper, we replicate Deep Q-Learning networks, proposed by Google DeepMind [1, 3], which allow for the creation of artificial agents capable of learning successful policies for playing Atari games with near or better than human-level control through spatio-temporal visual input. The problem of exploration vs. exploitation is then discussed. In particular, we look at autoencoders for learning good representations of state and thus providing reward bonuses for visiting unexplored parts of the state space, as done previously by Stadie et al [2]. We finally evaluate how the use of denoising autoencoders can affect exploration.

1 Introduction

Learning from high-dimensional raw sensory data to control agents has been a long standing challenge of reinforcement learning (RL). One way to handle the large state spaces that are common to computer vision and speech recognition type tasks is to use hand labeled features. However, hand-labeled features can be highly domain dependent and there is a question of how to get such features.

Artificial Neural Networks (ANNs) are a commonly known method for function approximation and hence machine learning. However, for many decades, their use was negligible in complex reinforcement learning domains because computational costs of using ANNs with many thousands or millions of hidden units were prohibitive. With Moore's law meaning that "harder" problems are easier to solve, we have been seeing a major interest in deep learning in the past few years.

Recently, Deep Q-Learning Networks has been shown to successfully "learn human-level control policies from high-dimensional sensory input using reinforcement learning" [1]. These methods have in particular been used to play video games, particularly Atari 2600 games. The main idea is to use a convolutional neural network that takes in preprocessed frames from the game and train the network with a variant of the Q learning algorithm. In addition, a separate target Q-network is used to delay the Q-value function updates and limit divergence and an experience replay mechanism is implemented to randomly sample previous transitions and avoid the correlation of using similar frames from same parts of the state space.

What follows is a description of how to replicate DQNs and arrive at the results previously published [1, 3]. We then discuss the common RL problem of exploration vs. exploitation particularly through the use of exploration bonuses using autoencoders [2]. Finally, we evaluate how modifying the autoencoder to a denoising one affects performance.

2 Preliminaries

The task of learning policies for playing Atari games can be modeled as an infinite horizon Markov Decision Process (MDP) [2], defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. Here, $\mathcal{S}$ is a finite set of states. $\mathcal{A}$ is a finite set of actions. $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ gives the probability of transitions to different state given that a particular action is taken at a given state. $\mathcal{R} : \mathcal{S} \to \mathbb{R}$. Finally, $\gamma \in (0, 1)$ is the discount factor that represents how much future rewards are weighted in comparison to immediate rewards. The aim is to find a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ that maximizes expected reward. This maximization can be done using different reinforcement learning algorithms.

2.1 Q-Learning

Q-Learning is a model-free reinforcement learning technique that can find an optimal action-selection policy for any given (finite) Markov decision process (MDP). Learning the policy is done through estimation of an action-value function, Q, which is the the expected utility of taking a given action in a given state and following the optimal policy thereafter.

The core of Q-Learning algorithms is a value iteration update of the following form

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t) \cdot \Big(R_{t+1} + \gamma \max_a Q_t(s_{t+1}, a_t) - Q_t(s_t, a_t)\Big)$$

where s_t and s_{t+1} are the states at time t and time $t + 1$ respectively, $\alpha_t(s_t, a_t)$ is a possibly time, state and action dependent learning rate and R_{t+1} is the immediate reward received upon performing action a_t in state s_t.

2.2 Convolutional Neural Nets

A Convolutional Neural Net (CNN) is a type of feed-forward ANN whose architechture is particularly suited to capturing features in visual input that are modeled after visual mechanisms in living organisms particularly that the cells in eyes are responsible for detecting light in small, overlapping sub-regions of the visual field, called receptive fields. More about such architectures can be found in [8].

3 Related Work

Previous work has shown that model-free reinforcement learning algorithms like Q-learning, which represent the action-value function with a non-linear function approximator, such as a neural network, become unstable and even diverge [5]. Therefore, a majority of efforts in reinforcement learning focused on adopting linear function approximators, on purpose of better convergence.

However, some stable method exist for training neural networks under the framework of reinforcement learning, in particular, neural fitted Q-learning (NFQ) [6]. NFQ involves the repeated training of networks de novo on hundreds of iterations. Thus, NFQ can be inefficient for use with large networks. On the other hand, Deep Q-Networks, introduced by Mnih et al [1], applies reinforcement learning end-to-end, capturing features from raw visual inputs that may be directly relevant to discriminating action-values. In addition, the DQN method uses a stochastic gradient descent based optimization that only has a low constant cost per iteration, making such an approach scalable to a large state spaces with possibly many millions of iterations.

Since the paper that proposed the idea of DQNs, many other papers have looked at how to improve the performance of DQNs on the Atari games benchmark [1, 3]. One in particular that we will be focusing on is the paper titled "Incentivizing exploration in Reinforcement Learning With Deep Predictive Models" by Sadie et al [2].

4 Methodology

4.1 Deep Q-Network

Deep Q-Networks are networks that, as the name suggests, perform Q-Learning using neural networks. In the Atari games domain, a CNN architecture can be used to approximate a Q-value function for the game being played using representations of the frames as state. The architecture used in the paper that introduced DQNs is shown below in the output of the program used for training DQN agents.

```
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> (7) ->
            (8) -> (9) -> (10) -> (11) -> output]
  (1): nn.Reshape(4x84x84)
  (2): nn.SpatialConvolution(4 -> 32, 8x8, 4,4, 1,1)
  (3): nn.Rectifier
  (4): nn.SpatialConvolution(32 -> 64, 4x4, 2,2)
  (5): nn.Rectifier
  (6): nn.SpatialConvolution(64 -> 64, 3x3)
  (7): nn.Rectifier
  (8): nn.Reshape(3136)
  (9): nn.Linear(3136 -> 512)
  (10): nn.Rectifier
  (11): nn.Linear(512 -> 6)
}
```

A description of DQNs in relation to the above architecture now follows. Evaluation of DQNs and questions related to model selection will follow later. First, a few steps of preprocessing are done to reduce the input dimensionality and ensure that the frames provide a good representation of state. In certain games, some objects appear only in every other frame as the Atari 2600 had a limit on the number of sprites that can be displayed at a time. To account for this, at time t, each pixel color value is taken to be the maximum of the pixel color value at time $t-1$ and the pixel color at time t. These "corrected" frames are then downsampled from the Atari 2600's screen resolution of 210×160 to 84×84. A conversion from RGB colorspace to YUV followed by removal of the U and V channels is also performed to extract the luminance of the frames. This completes preprocessing and thus, we now have the 4 frames that are passed into the CNN as input.

The CNN consists of three convolutional layers. The first convolutional layer consists of 32 filters with a 8×8 kernel and a stride of 4. The second convolutional layer consists of 64 filters with a 4×4 kernel and a stride of 2. The third convolutional layer consists of 64 filters with a 3×3 kernel and a stride of 1. To allow for the capturing of non-linearities, after each convolutional layer, we use a rectified linear unit (ReLU) which performs $f(x) = \max(0, x)$. The last convolutional layers output (after passing through a rectifier) are reshaped into a set of linear neurons and a linear layer to 512 neurons follows. Finally, a linear layer to the output follows. The size of the output layer is game dependent, but it is equal to the number of actions available in the game. Running the emulator forward for one frame step requires much less computation than having the agent select an action by a multiple forward passes on the neural network when the output is of size 1, such technique allows the agent to play approximately k times more frames of games without significantly increasing the total runtime.

There are two remaining parts to the DQN architecture. The first is the use of experience replay. At each time step, an experience tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ is stored into a "replay memory" dataset D of size 10^6. During learning, we apply Q-learning updates, on samples or minibatches of stored experience $(s, a, r, s') \in D$, drawn uniformly at random from the pool of stored samples.

The second is the use of a separate target Q-network. Every 50000 steps the network Q is cloned to obtain a target network $\hat{Q}$ and $\hat{Q}$ is then used for generating Q-learning targets for Q in the next 50000 steps. In the Q-learning updates, a mini-batch size of 32 is used with the following sequence of loss functions:

$$L(\theta_i) = \mathbb{E}_{s,a,r}\left[\left(\mathbb{E}_{s'}\left[y|s,a\right] - Q(s,a;\theta_i)\right)^2\right]$$

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s',a',\theta_i^- - Q(s,a;\theta_i\right) \nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

In addition, the rewards returned by Atari emulator is clipped for generalization. As the scale of scores varies vastly between different games, all positive rewards are clipped at 1 and all negative rewards are clipped at -1, leaving 0 rewards unchanged. Such modification limits the scale of the error derivatives and also makes a single learning rate more robust across multiple games. Meanwhile, a drawback of this modification is that the agent cannot differentiate rewards of different magnitude, potentially influencing performance.

Following the same procedure as proposed by [7], a simple frame-skipping technique is also used. Specifically, the agent perceives and selects actions on every k^{th} frame instead of every frame, with the last action repeated on skipped frames.

To conclude, we discuss a few other key points and hyperparamters of the architecture, a full list of hyperparameters can be found in [1]. For updating the weights of the networks, RMSProp (proposed by Geoffrey Hinton but unpublished) is used. Action selection is done using an ϵ-greedy policy. This means that with probability ϵ, a random action is chosen and with probability $1 - \epsilon$, the action that corresponds to the highest Q-value is chosen. $epsilon$ is linearly interpolated from 1 to 0.1 over the first 10^6 steps (actions). Finally, the model is trained for 5×10^6 iterations. Note that this is less than the 5×10^7 steps used in [1].

4.2 DQN with Reward Bonuses

To incentivize exploration, the agent can be rewarded bonuses for visiting novel parts of the state space. Using the frames that the CNN gets as input in a novelty function is however problematic as the state space induced by the frames themselves is very large. Thus, the use of encodings has been suggested [2]. An autoencoder is a type of neural network whose aim is to reconstruct it's inputs. The architecture proposed in [2] to produce encodings is as follows

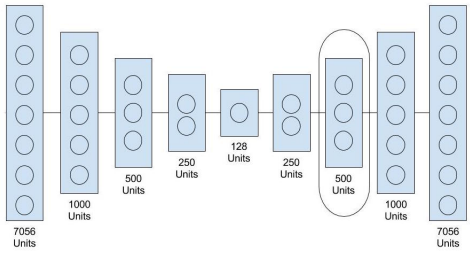

Figure 1: Autoencoder network

As can be seen above, the input to the autoencoder takes as input a flattened frame (note that $84 \times 84 = 7056$). The frame passes through multiple fully-connected linear layers until it reaches a bottleneck at 128 units. Then, the frame passes through larger and larger hidden layers until the output layer which has the same size as the input layer. Thus, the inner layers capture encodings of the frames, and hence the state.

To be able to use the above described autoencoder for incentivizing exploration, we can come up with a model learning architecture that takes in the encoding of a state and the action performed in that state and gives a prediction for the encoding of the next state. The error in this prediction is presumably low when if we are in part of the state space we have visited often and high if we are in part of the state space which we have not visited or not visited often. Hence, we can come up with an error measure for this prediction and a method for awarding reward bonuses for exploration.

Let $\sigma(s)$ be the autoencoder's output at the sixth layer (the one circled in the diagram) given that s is passed as input. In other words, we are using the 500 units of the sixth layer as the encoding of state.

Let $\mathcal{M}_\phi$ be the output of the model learning architecture which takes in the encoding of a state and the action taken in that state. The model learning architecture is given by a very simple feed-forward network. It takes in a vector of size 518 which comes from the vector that represents the encoding

of state (of size 500) concatenated with a one-hot action vector (only one entry is a 1 and all others are 0s). We use 18 since there are a maximum of 18 available actions in any Atari 2600 game.

With the above explanation, we then have an error function

$$e(s_t, s_{t+1}) = ||\sigma(s_{t+1} - \mathcal{M}_\phi(\sigma(s_t), a_t)||_2^2$$

The above error function can be used to come up with a novelty function

$$\mathcal{N}(s_t, a_t) = \left\| \frac{\overline{e}_t(s_t, a_t)}{t \cdot C} \right\|_2^2$$

where $\overline{e}_t(s_t, a_t)$ is the normalized error $\overline{e}_T = \frac{e_T}{\max_{t \leq T} e_t}$ and $C > 0$ is a decay constant.

Finally, introducing a multiplier to weight how much exploration is "valued," we get

$$\mathcal{R}_{\text{Bonus}}(s, a) = \mathcal{R}(s, a) + \beta \mathcal{N}(s_t, a_t)$$

Using the above functions for calculating reward bonuses, we can train an autoencoder prior to beginning the training of DQN. Once the DQN training begin, we update $\mathcal{M}_phi$ during the evaluation by sampling frames from experience replay.

4.3 Denoising Autoencoders and reward bonus modification

Since autoencoders are trained to reconstruct their inputs it is quite possible that without any constraints they learn the identity function and in that process, fail to capture useful representations of state. The idea behind denoising autoencoders is that in order to force hidden layers to discover more robust features and prevent them from simply learning the identity, we can train autoencoders to reconstruct the input from a corrupted version of it. Corrupting the input can be done in different ways. The one we used was to use dropout [7] whereby with probability p, which we set to 0.5, some inputs are set to 0 before each layer.

5 Results

5.1 DQN Replication

The performance of the DQN replication can be seen in the following table. Since our replication was run for 5×10^6 steps for each game where 1 epoch is 50,000 time steps, the training was run for 100 epochs. Thus, we should compare our DQN replication results with DQN results achieved after 100 epochs of training. The original paper by Google DeepMind [1] gives results for 1000 epochs of training. However, the paper by Stadie et. al [2] which talks about incentivizing exploration quotes results after 100 training epochs. Thus, we compare column 2 with columns 3 and 4. The third column titled "DQN Replication (Average)," as the name states, gives the average score over the 100 training epochs plus/minus the standard deviation while the fourth column titled "DQN Replication (Best)," as the name states, gives the highest score achieved during the 100 training epochs.

Game ↓ Agent →	DQN [1]	DQN Replication (Average)	DQN Replication (Best)	DQN (1000 epochs) [1]	Random Player
Bowling	30.5	22.29 (± 12.47)	53.25	42.4 (± 88.0)	23.1
Breakout	169.9	46.40 (± 54.26)	205.80	401.2 (± 26.9)	1.7
Enduro	321.0	248.70 (± 247.62)	777.00	301.8 (± 24.6)	0
Freeway	6.0	13.38 (± 11.93)	29.25	30.3 (± 0.7)	0
Q*Bert	5524	749.90 (± 551.50)	3027.50	10596.0 (± 3294.0)	163.9
Seaquest	2104	637.31 (± 447.16)	1665.00	5286.0 (± 1310)	68.4

Table 1: DQN Replication Results

As seen from the table above, our replication's score is better than the results quoted (for 100 epochs in [2]) on Freeway only. However, our best scores are better than the quoted results in Bowling, Breakout and Freeway. Furthermore, a random player (one who takes random action at each time step) clearly does worse in all the games except possibly Bowling where the average score received is lower than the random player's. The training curves for the DQN replication appear on the following page.

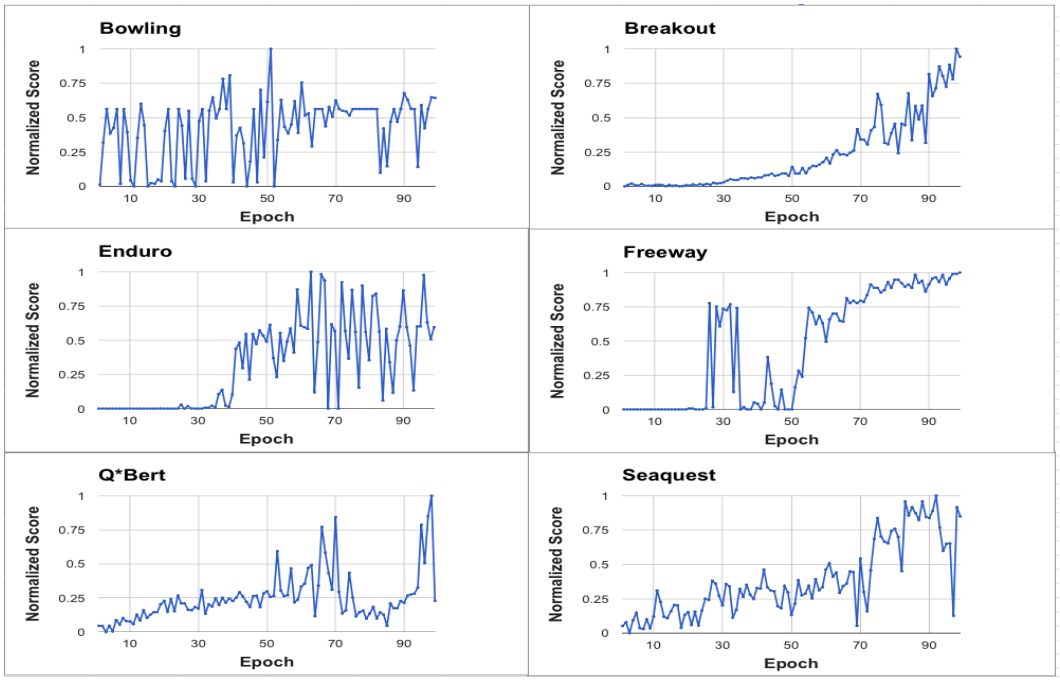

Figure 2: Training Curves for DQN Replication with different games

In the above plots, we plot the normalized score (score after each evaluation divided by the maximum score over the entire training). Some trends are evident. Firstly, as the plots indicate, there is a lot of noise in the performance measure (the game score). This is likely a simple artifact of the presence of numerous local minima as is generally the case with neural networks. This stability however does vary among games. Breakout, for example, looks very stable and this is likely because breakout has a very small number of actions available (move left, move right and fire) and that the game is fully observable and deterministic (the screen contains the entire state and there are no stochastic elements). Consequently, other games like Seaquest which have Stochastic elements like enemies (fishes) are less stable. Now, we can look further into the interpretation of a learned policy.

5.2 Interpretation of a learned policy

To interpret a learned policy, we can observe the gameplay as the DQN agent plays after training. We can also look at the training curves and the learned filters of the first convolutional layer (later convolutional layers will only capture a small portion of the image and are generally hard to interpret). To illustrate this, we focus on the game Seaquest.

The first convolutional layer filter responses after forwarding 4 frames are shown below

The activation of each pixel can be seen using the luminance of the pixel (brighter means higher activation). From the frames above, it appears that there is a focus on the submarine and the enemies (fish) surrounding the submarine as well as the score and oxygen bar indicator at the bottom. Thus, the filters give us an idea of what the agent is focusing on.

How the agent acts requires a different kind of analysis. On looking at the game play after training, we found that the agent never goes to the top (out of the water) to retrieve oxygen. Notably, the game ends when the oxygen bar reaches zero. It is possible that even though the agent recognizes

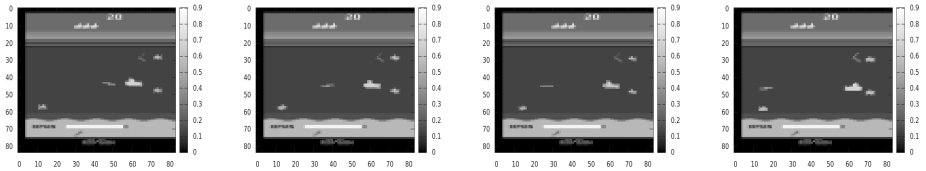

Figure 3: Filter Visualization

that the oxygen bar somehow seems to affect the score received, it still has not explored enough of the state space to understand in what way.

5.3 Exploration Bonus affects

On adding the idea of reward bonuses for exploration, we get the results shown in the following table.

Game ↓ Agent →	DQN w. AE [2]	DQN w. AE Replication (Avg.)	DQN w. AE Replication (Best)	DQN w. DAE (Avg.)	DQN w. DAE (Best)
Bowling	130	100.00 (± 15.67)	130	120 (± 20.54)	160.00
Breakout	162	165.70 (± 30.26)	210	160.80 (± 41.55)	215.00
Seaquest	2636	2750 (± 450.15)	3055	3380 (± 490.54)	3980.70

Table 2: DQN with AE replication and DAE results

As can be seen above, on adding reward bonuses for exploration, the performance of the agents seem to generally get better. The average score for the AE replication is quite similar to the scores quoted in the paper by Stadie et al [2]. In particular, the replication seems to do better on Breakout and Seaquest but worse on Bowling. The denoising autoencoder seems to help in Seaquest only based on the average score, however; the best scores for the DAE seem to be higher for all the games we tried. The best improvement comes from Seaquest.

DAEs can be helpful as they increase the robustness of the encodings of state. Since nearly half of the inputs are corrupted during dropout in the first layer itself, it is reasonable to assume that better features are captured. How these features help through reward bonuses is another question.

To analyze how the use of reward bonuses help, we visualized the convolutional filters, however, the attentions to different parts of the frame look about the same. In the case of Seaquest, the attention to the oxygen bar is still present (as the case with the simple DQN without any reward bonuses). Therefore, the convolutional filters alone cannot tell us how the use of autoencoders and exploration bonuses help. Looking at the gameplay however, it then became evident that the agent had learned to surface for oxygen.

Continuing our analysis, we questioned why a DAE does much better on Seaquest and Bowling but not on Breakout. Since denoising corrupts the inputs, it is possible that the DAE will help more in games with stochastic elements like Seaquest where the enemies (fish) spawn at random times. These stochastic visual elements can be viewed as corruptions in the input.

It is also possible that the DAE helps in games with stochastic elements because the reward bonuses assigned are not affected by the stochastic changes in the game. In other words, suppose that the model architecture predicts the encoding of the next frame as one without any fish. If a single fish appears in the next screen by chance a simple non-denoising autoencoder might give a high error and thus a high bonus. With denoising, all frames are corrupted to some extent and thus minute differences due to the stochastic nature of a game will not affect performance drastically.

5.4 Model Selection

As the case with most machine learning algorithms, there are many hyperparameters used in DQNs. The ones we used were the ones used by Mnih et al [1]. These hyperparameters seem to give good results in most games. However, it is quite likely that certain sets of hyperparameters work better with different games.

The hyperparameters for the reward bonuses were not specified in the paper by Sadie et al [2], however, after trying a few different parameters, we found that a decay constant of around 0.9 and beta multiplier of 0.05 seem to give the best results and hence these are the ones we used.

In the denoising autoencoder, we chose a dropout probability of 0.5 without doing any hyperparameter tuning. This is primarily because 0.5 seems to be the default parameter used in many other applications, and it seemed to give a good error during evaluation of the autoencoder after training.

6 Conclusion

6.1 Strengths

DQNs have been shown to achieve human level control in different games [1] and this result clearly illustrates that DQNs can learn policies for tasks that involve complex spatio-temporal input. They are also able to learn policies for a different variety of games as well and in a model-free manner. Adding the idea of reward bonuses, we can encourage exploration in domains where exploration is crucial. Finally, incorporating the idea of denoising in the autoencoders used for calculating reward bonuses helps further in the goal of awarding reward bonuses.

6.2 Weaknesses

As with most neural networks, there are many local minima and it is possible for the agent to get stuck in these local minima leading to less than optimal policies. This also can vastly increase the training time and produce a lot of noise in the performance measure as seen from the plots shown previously. In fact, each run of our DQN replication (without additional changes like AEs) took about 3 days to run. The DQN replications with AEs and DAEs took about the same time as well. Hence, the efficiency of DQNs is still a cause for concern. Additionally, DQNs have numerous hyperparameters and choosing these can be time consuming as well. Finally, the use of autoencoders for providing reward bonuses while beneficial to training also makes the problem nonstationary, this can be a problem from a theoretical standpoint though empirically the results indicate otherwise.

6.3 Summary

Thus, we have replicated DQNs and illustrated their use in finding policies for playing Atari video games. We extended on DQNs using autoencoders for extracting representations of state and using them along with actions to predict next state. This gives us an error measure that can be used to encourage exploration. This modification of DQN seems to perform well and we were able to replicate the results of Stadie et al [2]. Introducing denoising, we showed how it could further help improve performance in stochastic type games. We also provided theory behind the algorithms where appropriate and discussed the results obtained along with the strengths and weaknesses of the methods used.

Individual Contribution

We all equally contributed to equally to the DQN implementation and the introduction of autoencoders and denoising autoencoders. In particular, since the code involves many parts that combine together to form a whole, we each took separate parts of the code and implemented them.

References

[1] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.

[2] Stadie, Bradly C., Sergey Levine, and Pieter Abbeel. "Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models." arXiv preprint arXiv:1507.00814 (2015).

[3] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[4] Tsitsiklis, J. N. & Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *Automatic Control, IEEE Transactions on*, 42(5), 674-690.

[5] Riedmiller, M. (2005). Neural fitted Q iterationfirst experiences with a data efficient neural reinforcement learning method. In *Machine Learning*: ECML 2005 (pp. 317-328). Springer Berlin Heidelberg.

[6] Lange, S. & Riedmiller, M. (2010, July). Deep auto-encoder neural networks in reinforcement learning. In *Neural Networks (IJCNN), The 2010 International Joint Conference on* (pp. 1-8). IEEE.

[7] Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. (2012). The arcade learning environment: An evaluation platform for general agents. *arXiv preprint arXiv*:1207.4708.

[8] Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[9] "Convolutional Neural Networks (LeNet)." Convolutional Neural Networks (LeNet) DeepLearning 0.1 Documentation. N.p., n.d. Web. 17 Dec. 2015.