Machine Learning Engineer Nanodegree

Capstone Project

Prasoon Shukla
January 31, 2017

Definition

Project Overview:

Reinforcement learning is a subfield of Machine Learning in which a machine agent learns the optimal behaviour in an environment in a trial-and-error manner. The agent is put in an environment and allowed to interact with it. Any action the agent performs results in a reward which can be either positive and negative and our intent is to train the agent to perform actions that result in the highest long term cumulative reward. As such, the agent learns on its own - there is no training data as is the case with supervised learning.
The problem of reinforcement learning can be framed mathematically as a *Markov Decision Process (MDP)* [1] where the environment - as perceived by the agent - is represented by the state space ***S***. The agent, ***P***, can perform any action from the set of all actions, ***A***, to gain some reward, ***r***. Then, the objective is for the agent to learn the optimal action to perform in any given state that maximizes the total long term reward to the agent. One of the ways to achieve this is by using Q-Learning [2]. In Q-Learning, we calculate the Q function, ***Q(s, a),*** for each state-action pair in an iterative manner and figure out the optimal action using this Q-value.
In recent years, there has been an increased focus on 'general purpose AIs' - AIs that can learn much like a human would. Reinforcement learning is a good tool for building such a general purpose AI but unfortunately, the state space for any non-trivial task is prohibitively large. If we were to use the Q-learning iterative algorithm to teach our agent, it would take us, in some cases, thousands of years to converge to the optimal solution. This has led to the (very recent) development of the Deep Q Learning algorithm which uses deep neural networks to approximate the Q function[3][4]. The neural net helps to extract meaningful features from the data thereby drastically reducing the effective state space and also learns the Q-values by incorporating the reward into the loss function [5]. We will therefore use Deep Q-Learning to teach an agent to play PacMan. It should be noted that while Deep Q-Learning is a new technique for solving MDPs, a slew of newer, far more efficient and much better performing algorithms have been introduced in the recent years which will produce better results in solving PacMan[6][7]. However, Deep Q-Learning is still the fundamental model on which most new techniques are built and therefore, a solution using DQNs can be modified to implement new algorithms, all of which use Neural Networks.

We'll be using the OpenAI Gym environment for PacMan for data collection and testing our solution.

Problem Statement:

Given to us is the Ms. Pac-Man game. The game consists of a maze with 'pellets' scattered around and 'ghosts' which chase the main player (PacMan). The objective of the game is to acquire all pellets without running into the ghosts. Some special pills, called 'energizers', allow the main player to attack and kill 'ghosts' for a small duration. The agent has three 'lives' and our objective is to make a player bot that can successfully navigate this maze, collecting as many pellets as possible in one game level. Initially, the expectation was that we would try to complete at least one level. However, as we will describe in the later part of this report, we are now aiming to try and score as many points as possible in a 20 episode average.

Solution: We will be using Deep Q-Network to solve this problem. Solution steps:

- Set up OpenAI Gym with the Atari environments
- Create a Deep Q-Network using tensorflow and baselines that takes in the image of the game as input and produces the optimal action to be performed as the output. The model will be explained in detail in the next few sections.
- Train this network using the PacMan Gym Environment by selecting good values for the hyperparameters. Run the trained model on a new game and record the final score of each game and pick the best 20 game average score.

Metrics: Unlike many other atari games, PacMan doesn't have a specific threshold where it is considered solved - the aim is simply to score as high as possible. In this vein, we accumulate the rewards that the player receives until all lives are exhausted. These rewards are consistent with the in-game score. The cumulative score at the end of the game is considered the "total score" for the player and it is this number that is used as the evaluation metric for measuring how well the model performed. This same metric is also used for the benchmark model for comparison. The simulation is run multiple times and we take the highest 20 episode running average as the score indicating the performance.

Analysis

Data Analysis: Since we are training an agent using reinforcement learning, we do not have a dataset. Instead, we will extract the data as we go from the game environment itself. We will be using the OpenAI Gym 'MsPacman-v0' environment to collect the data from the simulation. OpenAI Gym is an easy way to simulate a game environment and to retrieve the state and reward. The environment is provided as the RGB image of each frame of size (210, 160, 3). A typical frame from the game contains maze walls, pellets, ghosts, number of lives remaining and the score. The major part of the screen is occupied by the actual maze within which the game is being played.

Fig. 1

The bottom 20% of the frame is a status bar that contains the current score, the number of lives remaining and the cherries (which are essentially irrelevant because of their infrequent appearance).

The agent can go in one of the 8 possible directions - left, right, up, down, left-up, left-down, right-up, right-down - or do nothing. The rewards are calculated based on the result of the agent's actions - eating a pellet, killing a ghost all result in positive rewards. Getting killed by the ghost should have ideally resulted in a negative reward however since the environment is set-up to replicated the game, we actually do not get any negative rewards on the 'death' of the agent and therefore the event of 'getting caught by the ghosts' isn't used in the DQN to optimize the Q function. If we

were to manually implement a negative reward everytime the game ended, it still wouldn't make any difference to the optimization of the Q-function since this is a very rare event and therefore the frequency of its occurrence in a training batch is very small. So, we only train the DQN on the positive rewards and the weights of the neural net will adjust so that the DQN can compute a good approximation of the optimal Q function [4].

Since we have the image of the frame, we have all the information needed by the neural net to construct the state. Just as in any other RL problem (like smartcab), the data isn't pregenerated - we simulate the data at training time. It should be noted here that while the dimensionality, and therefore the state space, appears quite large for this problem, it's not actually as large as it first appears. Just as in the in the MNIST data set where the actual 28 * 28 = 768 dimensional image can be projected down to a much smaller subspace, similarly we can project the 210 * 160 * 3 = 100800 dimensional data in a PacMan game frame to a much smaller subspace without losing any information. This dimensionality reduction occurs automatically within the neural network since neurals nets are quite good at finding equivalent representations of data in lower dimensional spaces and the reason why the DQN can do this effectively is because a lot of pixels in the image are immovable and unchanging and therefore don't contribute to the change in state.

Once we have a frame, we preprocess it by converting it to grayscale, removing the status bar at the bottom and scaling down the size to (86, 80) to reduce the dimensionality without losing much information followed by scaling the image data to [0, 1] range. The only information we lose from this is the number of lives which shouldn't affect the optimal strategy of avoiding the ghosts by much. We also lose information about the cherries but that is an element for which it's very difficult to train the network because of its rare appearance in the maze. This preprocessed image will then be passed to the neural net to decide the optimal action. Once the agent decides on an action and passes it to the environment, the action is executed on the next 2, 3 or 4 (the number of frames is chosen randomly by the environment). Once the execution is completed, the agent is provided with a reward. We use the reward and other Q-values to calculate loss and train the neural net.

Exploratory Visualization: The game frame look as shown in **Fig. 1** and the transformed frame looks as follows (scaling not being showed as the image becomes difficult to see at such small scale):

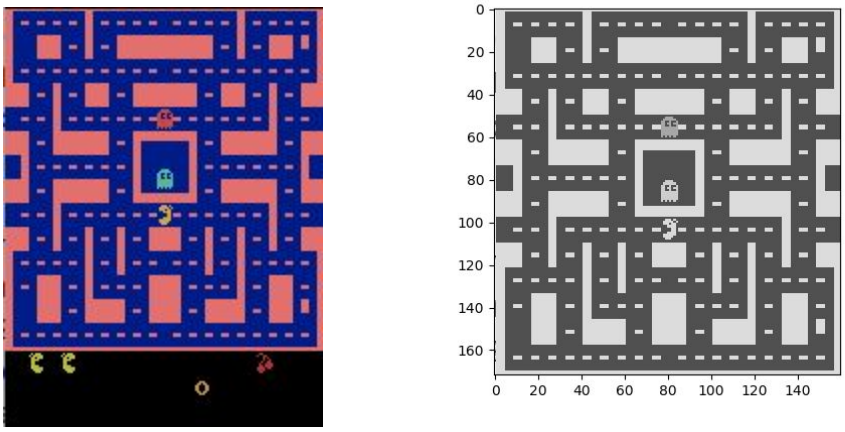


Fig. 2

Algorithms and Techniques:

As discussed in the Project Review, we will be using Deep Q-Learning to solve this problem. This technique uses neural networks for calculating the Q-values of all states and the neural net architecture is called a Deep Q-Network (DQN). In essence, the neural net is used for approximating the Q-function such that given a state, the DQN returns Q-values for all actions possible in that state.

The DQN was first made by a team at Deepmind and was popularized by their paper 'Playing atari with deep reinforcement learning'[3]. The DQN is a deep convolutional neural network with three convolutional layers and two fully connected layers:

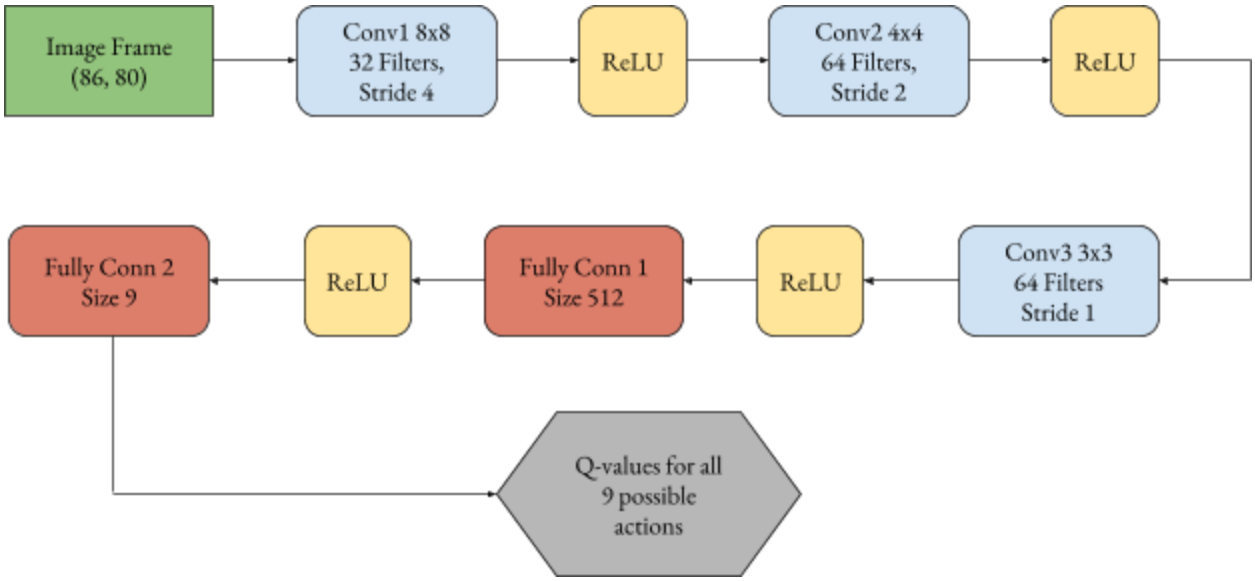


Fig. 3: The architecture of the Deep Q-Network

We use the following loss function:

$$L = \frac{1}{2}[r + \gamma(max_{a'}Q(s', a')) - Q(s, a)]^2$$

Fig. 4: The loss function used in a DQN

This loss function is based on the Bellman Equation for calculating Q-values iteratively[1]. We subtract the old Q-values from the network-calculated, decayed new Q-value and add the reward followed by squaring this quantity to calculate the loss for one state/action pair. A *naive* training algorithm can be described as follows:

1. Initialize the DQN with small random weights.
2. For each step:
    a. Get state
    b. Decide action using the exploration parameter. Feedforward the state to the DQN to decide the best action to take.
    c. Take the action and observe the new state and the reward.
    d. Use the DQN to calculate Q-values for all actions in the new state.
    e. Use the loss function of **Fig. 4** to calculate the loss, backpropagate the gradients through the network.
    f. Repeat until desired network performance.

If we were to use this naive algorithm and feed the state to the DQN one at a time, the network will perform very poorly. The most recent states are always highly correlated because of the sequential nature of the problem. This causes the function to often gets stuck in a local minima since it's only 'looking' at the most recent state(s) of the network to optimize weights. Also, it turns out that approximation of Q-values using non-linear functions is not very stable[5]. The behaviour, therefore, is erratic and to the casual eye, the actions appear to be chosen completely at

random. The function does not converge to a good minima and takes a long time in training. To mitigate this behaviour, the Deepmind team suggests a few techniques:

Experience Replay:

We build a 'replay buffer' in which we store past experiences ***<s, a, s', r>*** and then, when training the neural net, we sample a batch of past experiences randomly from the replay buffer. This helps in breaking correlation.

Note: We are using an advanced version of experience replay, called 'Prioritized Replay'. In this version, we don't select past experiences randomly. Instead we use some priority order on the experiences and sample based on that priority order [8].

Double Q-Learning

It was observed that even with experience relay in place, the policy changes quite drastically with small changes to Q-values. In other words, the policy function, $\pi = argmax_{a'}Q(s,\ a')$ isn't stable. To counter this, we use two sets of weights, called the target weights and the online weights. We use the online weights for calculating the Q-values of each state and the target weights to calculate the target values (the $r\ + \gamma max_{a'}Q(s,\ a')$ part in the **Fig. 4**). We update the online weights each time we backpropagate through the DQN. After a set of samples (say 10,000), we update the target weights with the online weights and the process begins again. The online weights are represented by w and the target weights are represented by w^-. Then, the loss function changes to:

$$L = \frac{1}{2}[r + \gamma(max_{a'}Q(s, a', w^-)) - Q(s, a, w)]^2$$

Fig. 5: Loss function with target and online weights

This method of using two sets of weights is call the Double Deep Q-Learning [9].

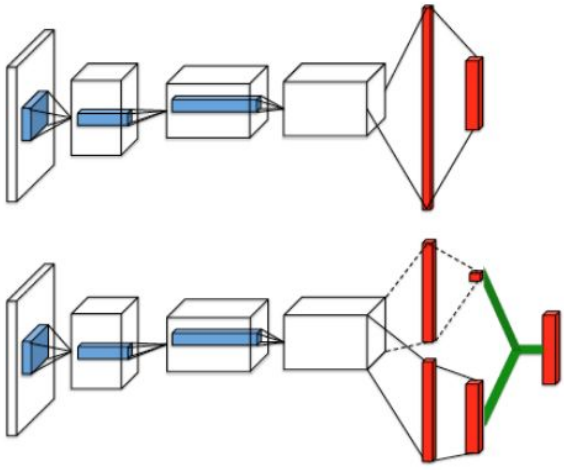


Fig. 6: Top: DQN without dueling. Bottom: DQN with duelling

Dueling Deep Q-Networks

The Q-values for any MDP can be further decomposed into two parts:

$$Q(s,\ a)\ =\ V(s)\ +\ A(s,\ a)$$

Here, the first part is the value function and the second part is the advantage function. The value function simply determines how good of a state the agent is in - it puts a number on the benefit of being in *a given state* vs being in any other state. The advantage function then tells only how much better is it to choose an action *a* in a given state compared to all other actions.

The Dueling DQN bifurcates the network after the convolutional layers into a Value part and an Advantage part. It corrects the issue with overestimation of Q-values by learning Q-values for states where actions have little to effect. [10]

Finally, our architecture looks like **Fig. 6** shown above.

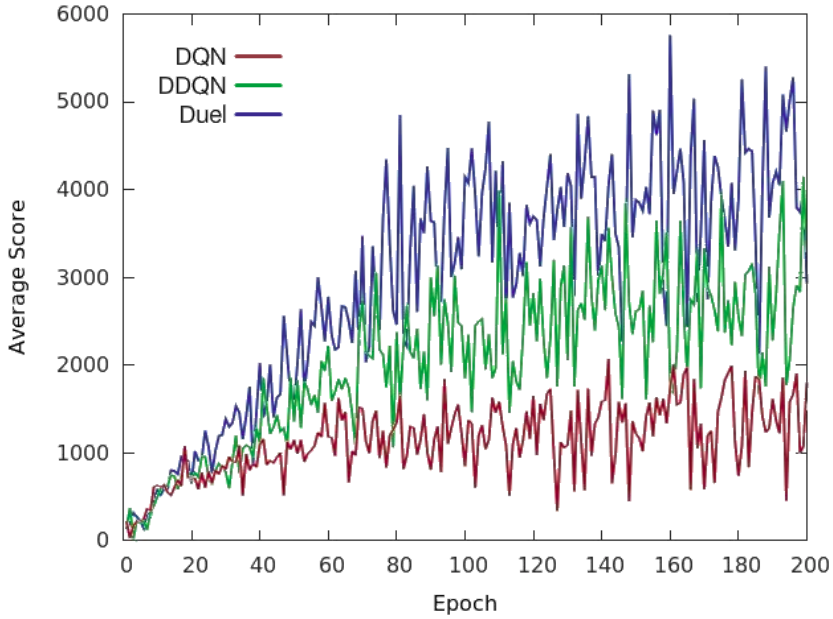


Fig. 9: Performance of DQN, Double DQN and Dueling DQN on Space Invaders. Image taken from torch blog [11]

Training on multiple frames: In the techniques discussed till now, the state fed to the DQN consists of one frame only. However, this poses a challenge in that it is impossible to determine the flow of time from the state itself. What this means is the DQN cannot make decisions based on whether, for example, the ghost is coming towards the agent or whether it's going away. For this reason, we stack together a number of frames (usually 2 to 5 frames) and we call this stack of frames the state. This enables the network to determine the best action across time-steps by taking into account the movement of various elements in the game. This is not a necessary step for static games where movement doesn't matter but in a game such as PacMan, predicting movement of ghosts is crucial for a good score.

Algorithm

Finally, the algorithm we use is as follows:

1. Initialize replay memory ***D***.
2. Initialize DQN for calculating ***Q-values*** with random weights.
3. For each time-step:
    a. Get 4 frames, preprocess and build stack of 4 frames. This is the state.
    b. With probability ε, select a random action. Otherwise, use the online-DQN to get the best action. Perform this action.
    c. Get the reward and the new frame. Modify the existing 'stack' of frames by removing the oldest frame and adding the newest frame.
    d. Store the transition in replay memory.
    e. At a fixed frequency of time-steps, select a sample of transitions from the replay memory. Forward propagate the batch through both online and target DQN weights and calculate loss using the equation in **Fig. 5**.
    f. Backpropagate the gradients calculated from the loss but only update the online-DQN weights.
    g. At a fixed frequency, update the weights of the target-DQN with the weights of the online-DQN.
    h. Repeat until performance is satisfactory.
4. Once the network is trained, save weights which can be used to to play games in the environment.

Benchmark:

Initially, we were planning on using the best performing model on the OpenAI Gym as the benchmark to beat. However, with a lot more reading, it is clear that the model described in the link above will be impossible to beat with DQNs. This benchmark model uses the A3C LSTM (Asynchronous Advantage Actor-Critic) model to train the agent which is a far superior model to the DQN and it came out in 3 years after the DQN model. The A3C model uses several agents training at once which also considerably decreases training time. Also, instead of using a stack of frames to determine the notion of time, this model uses a LSTM instead for this purpose. For all these reasons, comparing a DQN against a well trained A3C LSTM is pointless.

We will therefore be comparing our full model against a vanilla DQN (this was also suggested by the reviewer of the Capstone Proposal). The vanilla DQN will **not** include the Double DQN weights and the Dueling DQN architecture. Also, it will operate on one frame at a time (so no stacking of frames).

We will compare a 20 episode average on both models after training each of them for 2 million time steps. The data for rewards in each episode will be generated as a CSV file for each model and this data will then be used to compute the 20 episode average.

We will **not** be using the OpenAI Gym environment collection process which compares a 100 episode average. The reason for this is that both the vanilla DQN and the Full DQN have a very jittery performance. The 100 episode average remains consistently low and doesn't show the better performing episode very well. For a more consistently scoring model, the 100 episode average would be a good choice however it isn't a good choice for high variance scores.

Methodology

Note: While the recommended length of this section is 3-5 pages, we have already discussed quite a lot of the methodology already in the Analysis section including the implementation of Double DQN, Dueling DQN, the stacking of frames and the algorithm. As such, this section will be smaller than recommended as there be no repetition of the already discussed topics. Instead, a reference will be provided to the relevant item in the previous sections.

Data Preprocessing:

Data preprocessing has already been discussed in the Analysis subsection. We convert the frame into grayscale, crop the bottom status bar and scale down the image to the new size of (86, 80) These frames are then stacked together in groups of 4 to create a state. There are no outliers in this data as this is an RL problem and we don't have a pre-prepared dataset. Any extreme states, if they occur at all, are considered part of the dataset and cannot be removed as outliers since they are valid states in the game.

Implementation: The algorithm for the Full DQN (with all modification discussed in the Analysis section) can be found in the Analysis > Algorithm subsection. The implementation was done using the OpenAI Baselines module. This module is a wrapper over tensorflow and has built-in methods that can:

- Create the computational graph in tensorflow
- Generate the 'train_fn' function that can be called for training the DQN by passing in the training data
- Generate the 'act_fn' function that can be used to determine the action using the online-DQN
- Generate the 'update_target_fn' function that can be called to update the weights of the target-DQN in the double DQN.

The overall procedure which was used in creating the code is as follows:

- Create a 'q_function_nn' function that takes in tensor and creates the computational graph as described in **Fig. 6**
- Call 'deepq.build_train' method from baselines module and pass it the input size (size of the state), the 'q_function_nn', the action space size (9 in this case), the optimizer (we're using Adam with learning rate 0.001) and the decay parameter 'gamma' (seen in **Fig. 5**, we're using a 0.99 value since the actions are deterministic) and the 'double_q' flag which specifies whether or not to use a Double DQN.
- This function will then generate the following functions:
    - 'train_fn', 'act_fn', 'update_target_fn' whose use has been described above.
    - We also get 'debug_fn' which can used for debugging purposes.
- Once we have these three functions, we follow the algorithm described in the Analysis > Algorithm section to train the network. There are a few edge cases that need to be handled:
    - When starting the simulation for the first time, we stack a copy of the initial frame 4 times to create the state.
    - If we want to load a previously saved model, we need to restore not only the saved model weights but also the replay memory and the iteration steps.
    - When an episode ends, we log to a CSV file the total reward of that episode.
    - When an episode ends, the next 90 frames are static so we skip these frames.

Refinement

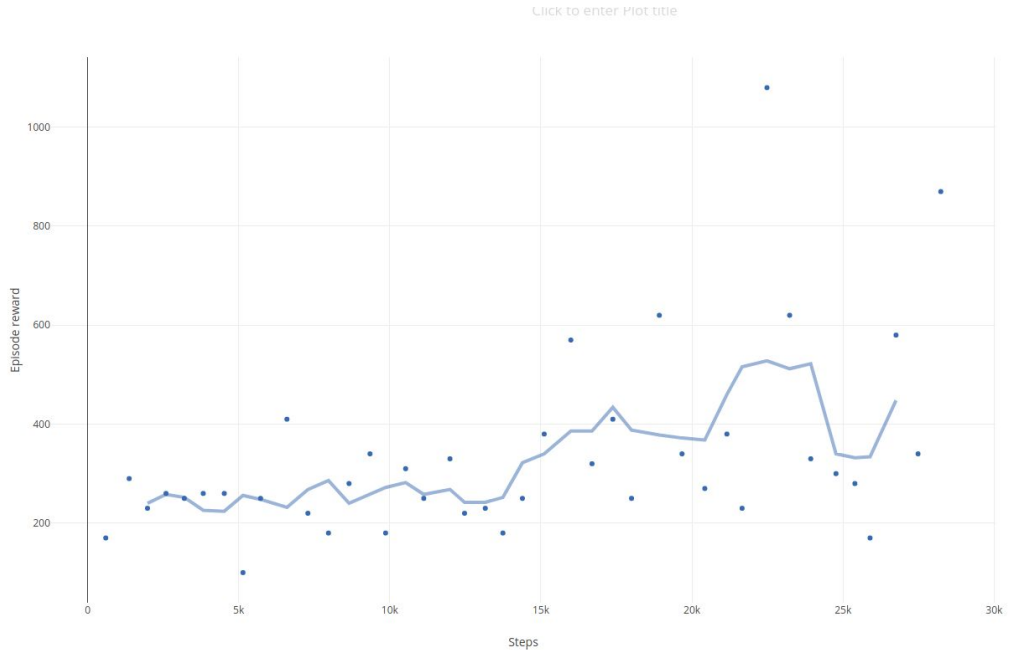
During the course of writing and testing the code for this project, several small changes were made to make the project work better. These can be classified into two parts: changes to the model and changes to the parameters. Changes to the model include:

- Modifying the initial model to use Double DQN
    - Already discussed the justification behind this change in detail in the Analysis > Double Q-Learning subsection.
    - The use of two sets of weights - online and target weights - reduce the oscillation in the policy on changing Q-values. Makes the policy more robust.
- Adding the Dueling layers at the end of the model
    - Justification discussed in Analysis > Dueling Deep Q-Networks subsection.
    - Network learns better in situations where all actions produce similar rewards [10].

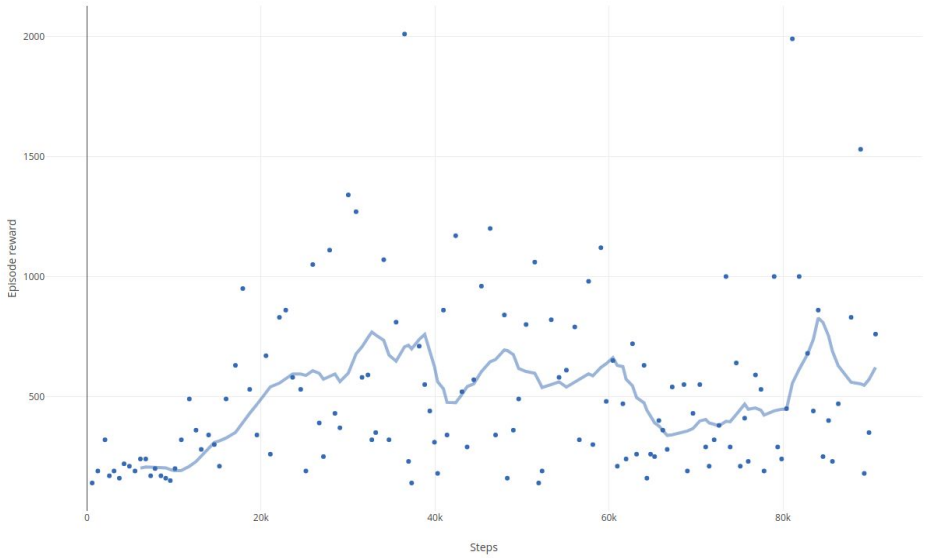
- Adding batch normalization to the Q-values at the end of the two ‘branches’ of the dueling network
    - The rewards produced can be quite large and thus can make the gradients flowing into the network very large. This can lead to dead neurons through which gradient never flows. Batch normalization avoids this problem by normalizing rewards and thus allows the network to train for a longer duration of time.

The other set of refinements are from choosing good hyperparameters. For the most part, we are using the same set of parameters as used by the Deepmind team in their original paper. However, due to limited resources, we could only train the network for a much short time-period (about 1/10th) and we could only keep our replay buffer small (about 1/10th in size). Both of these limitations resulted in worse results than provided by the original paper. The good thing, however, is that we were able to train this network on a small machine with a 2GB graphic Nvidia 860m graphic card which is good for easy prototyping.

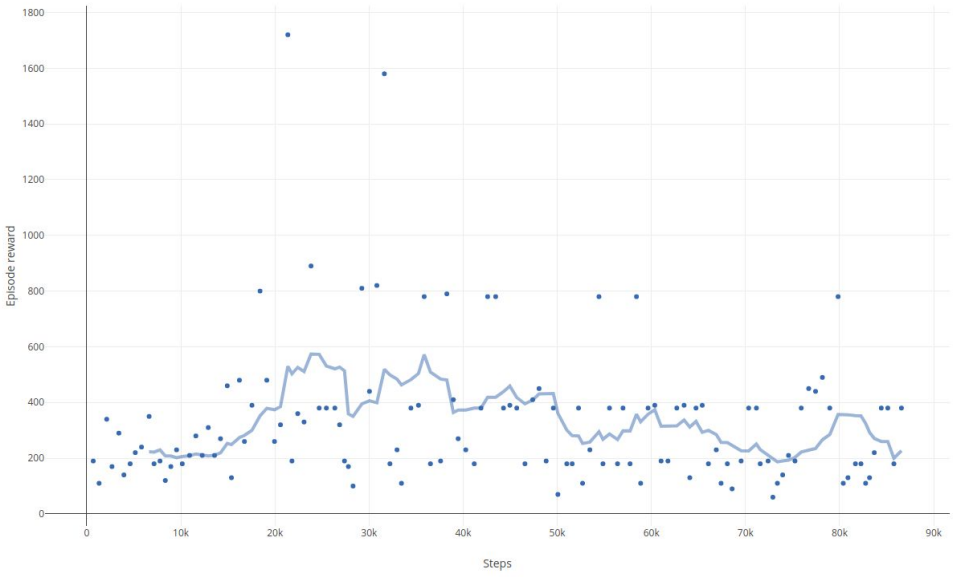
Hyperparameter tuning took a disproportionately long time since it takes a long time to train the DQN to be measurable. For each parameter, we changed its levels to reasonable values and recorded the episode rewards along with the time-steps. Then, we plotted these rewards and selected the one which provided both better results in terms of 10 episode moving average and was closer to values used in the original Deepmind paper. Some graphs are provided below.



Batch size 25



Batch size 50



No dueling

A set of graphs and the CSV files required to generate these graphs will be provided in the GitHub link[12].

In the end, using these graphs, we found what we think are a good set of parameters that train relatively quickly on a small machine.

Results

Model Evaluation and Validation

There have been several new techniques in the past few years that target solving of general purpose problem using reinforcement learning. Deep Q-Learning was one of the first models to do this using neural networks and the field of reinforcement learning has exploded by leaps and bounds since then. Using Deep Q-Networks is not the best solution

to the problems that we are trying to solve but it is the simplest solution. As discussed in the Analysis section, we found that the DQN described in the original paper [3] used an enormous amount of computational resources to perform reasonably well (which is still worse than a good human player [13]). Therefore, we are evaluating out model not by comparing it against the state-of-the-art solution but by comparing it against a vanilla DQN that is trained with the same resource and time constraints.

As discussed previously, we found that the vanilla DQN performs very poorly (see the no dueling graph above; even with Double DQN, the performance is barely better than for a random agent). Therefore, after reading up on more research on DQNs, we have came up with the final Full DQN model:

- We stack together frames to encode in the state the passage of time.
- We used a Double Deep Q-Network by having two sets of weights, one for online-DQN and one for target-DQN. We update the weights frequently from online to the target DQN and this keeps the policy function robust to small changes in Q-values.
- We used Dueling Deep Q-Networks to handle the case where the network was overestimating the Q-value in a state [10]. We also handled the case where all actions have similar results such as when PacMan has no ghosts close to it and has pellets in all directions.

All these changes, added together, make the network at least somewhat competent at playing PacMan and makes it much better that a vanilla DQN. It should be noted that PacMan is one of the problems which is hard for Reinforcement Learning agents to effectively solve because of the strategic component involved (a far more extreme version of this is the game Montezuma's Revenge where the RL agent performs worse than a random agent). Even the most advanced model can only clear 2-3 levels of the game consistently [55].

Game	Worst Demo Score	Best Demo Score	Number Transitions	Number Episodes	DQfD Mean Score	PDD DQN Mean Score	Imitation Mean Score
Ms Pacman	31781	55021	21896	3	**4695.7**	3684.2	692.4

The first three scores are by human players whereas the last three scores are from RL agents. Clearly, RL agents have a lot of catching up to do for this game.

For our metrics, here are the graphs for the 20 episode average for both vanilla DQN and Full DQN run on a total of 500 episodes.

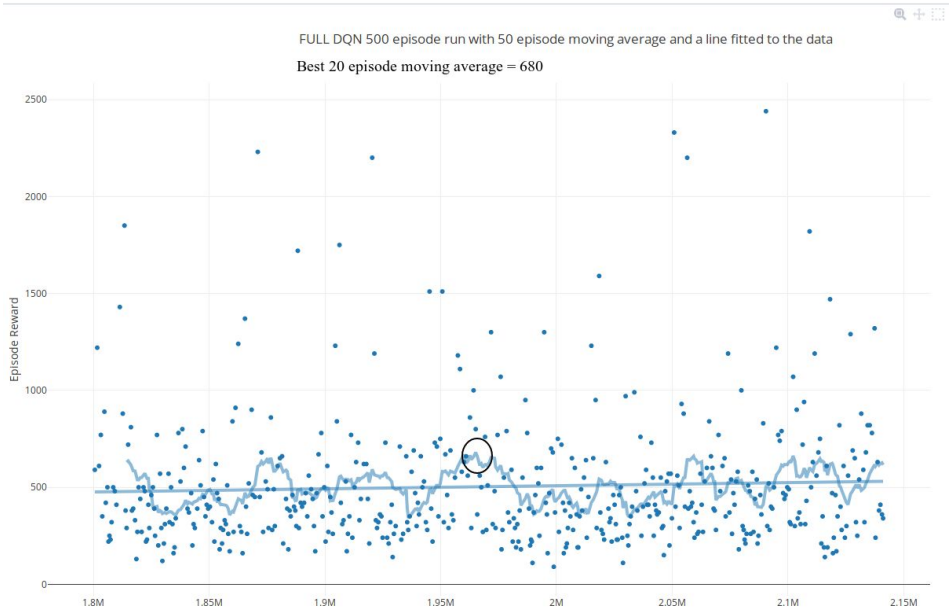


Fig. 10.a : Performance of the Full DQN on 500 episodes. 20 episode MA = 680

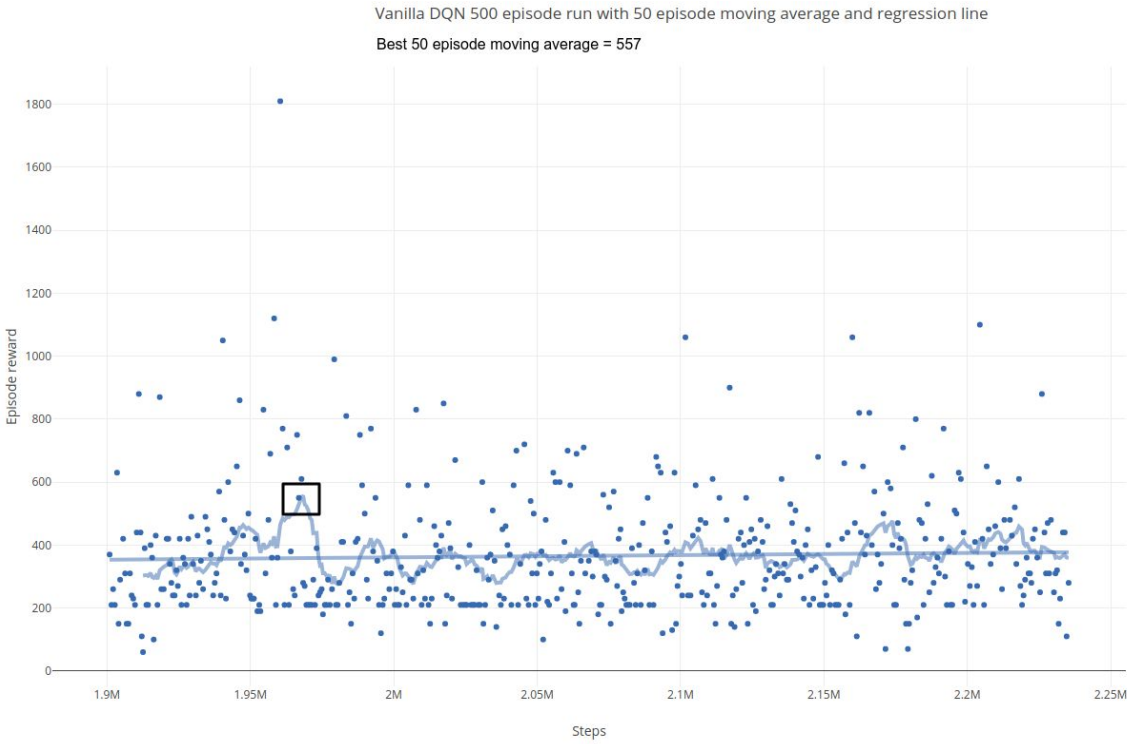


Fig. 10.a : Performance of the Vanilla DQN on 500 episodes. 20 episode MA = 557

As we can see, the Full DQN performs better than the vanilla DQN and the best 20 episode average is 680 (see the area in **Fig. 10.a** highlighted in a circle) which is better that the best 20 episode average for the vanilla DQN which is 557 for a 500 episode run which was what we were hoping to prove. As far as the question of generalization and robustness goes, we have already discussed that since this is an RL problem and the data is completely computer generated, we must consider all inputs to be valid. That is to say that if the agent reaches a state which an observer would classify as 'extreme' for that environment, this state should still be used for learning by the agent and shouldn't be removed as an outlier. The model is not as robust as we would like even after adding Double DQN and Dueling DQN - the jitter in the total episode reward is still much higher than the state-of-the-art model [56]. However, we did manage to improve the performance of the model and on a small machine. As mentioned, the full episode rewards have a high variance but it the results can be trusted since the environment enforces correct behaviour.

Justification

As evident from the graphs in, the Full DQN model performs better than the vanilla DQN model on a 20 episode mean score. Statistically,

- Vanilla DQN
    - Mean: 366.08
    - Std Dev: 189.72
    - Best 20 episode MA: 557
- Full DQN
    - Mean: 503.7
    - Std Dev: 334.60
    - Best 20 episode MA: 680

Calculation of these values can be found in this Google Sheet. We have used values generated from 500 episode runs of both Full DQN and Vanilla DQN, both also stored as CSV files in the code submitted (see README.me).
The mean of Full DQN is clearly better than the Vanilla DQN. The Full DQNs mean is 0.72 standard deviations away from the mean of the Vanilla DQN- while it won't clear a p-value test - it's still much better than a vanilla DQN.

Conclusion

Free-form visualization

Here is a video of the agent playing PacMan. This is on a random collection of episodes and the scores vary widely: https://vimeo.com/254426113

From the video, it's very clear that we haven't trained the agent long enough (even though it was trained for 15+ hours) since the agent sometimes gets stuck at corners and only leaves the corner when the state changes somewhat (so if a ghost comes nearby). In a well trained model, the agent would know not to execute actions that make it bump against the wall. Also evident from this video is that the agent has not learned to avoid ghosts very well. Sometimes, the agent runs from ghosts while at other times, the agent collides head-first into the ghosts. This is a sign of two things - a model that needs more training and a reward system that doesn't de-incentivize running into ghosts.

Reflection

There have been many learning opportunities in this project. When I started writing the Capstone Proposal for this project, I was uncertain of which field to do the Capstone Project in. Initially, I was leaning towards doing something using Convnets but in the end, I chose to do a project in RL because of its focus on general purpose AIs. After some research, which included reading sections of papers and many blog posts, I decided that I could write an agent that plays PacMan. The expectation was that the agent would be good enough to beat or at least come close to the best solution on the OpenAI Gym. Unfortunately, I had underestimated the sheer magnitude by which the research in this field has grown over the last 4+ years.

At first, I started reading in more depth on Deep Q-Learning and related topics. It became increasingly clear to me that the simplistic DQN model that I was thinking of making would be unable to to solve PacMan. It may have been able to solve some other atari games but PacMan, due to its deep strategic component, is a hard problem to solve for even the state-of-the-art RL models. Nevertheless, I decided to build the simple DQN model that I had proposed. Initially, I wanted to do everything in tensorflow but it soon became evident that using a purpose build RL library would be a better choice to avoid bugs and for easier prototyping.

The initial model was built using OpenAI baselines python library which is a wrapper over tensorflow. Unfortunately, the simple DQN (which we've been calling vanilla DQN in this report) performed far more poorly than I had expected.

It was then that I finally realized that there is only so much we can do with a DQN base. Then, I changed the project plan to a more realistic one - try to make improvements to the vanilla DQN to make it better. A good amount of reading later I had discovered Double DQN, Dueling DQN and frame stacking. In the next iteration of the model, all these techniques were implemented and the model seemed to perform better than the vanilla DQN. Then, I made some improvements in the hyperparameters which also improved the result by a small amount.

There were several deeply interesting aspects of this project not least of which was learning about the several newer models that have been published by Deepmind and other universities. I ended up watching the AlphaGo documentary and learned about the model-based approach to RL where the agent has access to a model of the environment and can therefore 'see' what will happen should it take a certain action. This kind of a 'look-ahead' move was most prominently used in AlphaGo itself and it would work very well on any deterministic system. I also learned about how making small changes to the model (such as the Dueling DQN) can improve results across all use cases.

There were also several challenges in this project. Understanding why some of these techniques work or why they don't work was quite challenging and to be frank, I'm still not completely certain about a few things. The feedback delay was also a huge problem since iterating different solutions taken several hours as I could only check how well my modifications worked after the model had trained for a few hours.

While the model did not achieve the original goal of beating the top submission on OpenAI Gym (which in hindsight appears a highly unrealistic goal), it still was able to beat vanilla DQN and we were able to show that (relatively) rapid prototyping can be done on decently powerful laptops.

Improvement
As stated several times before, this model was trained on relatively underpowered hardware. Compared to the state-of-the-art benchmarks which train on hardware that can be upto 100 times better would certainly help produce better results. Given more time and more computing resources, the hyperparameters can be tuned to produce the best possible results which is not the case at the moment. As far as the general purpose RL problem is concerned, it would be best to move away from DQNs and explore the new research on this topic such as A3C, ACER and ACKTR.

References

[1] van Otterlo M., Wiering M. (2012) Reinforcement Learning and Markov Decision Processes. In: Wiering M., van Otterlo M. (eds) Reinforcement Learning. Adaptation, Learning, and Optimization, vol 12. Springer, Berlin, Heidelberg

[2] Watkins, C.J.C.H. & Dayan, P. Mach Learn (1992) 8: 279. https://doi.org/10.1007/BF00992698

[3] Playing Atari with Deep Reinforcement Learning by Volodymyr Mnih et. al. https://arxiv.org/abs/1312.5602

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

[5] Tambet Matiisen, Demystifying Deep Reinforcement Learning, https://www.intelnervana.com/demystifying-deep-reinforcement-learning/

[6] Asynchronous Methods for Deep Reinforcement Learning by Volodymyr Mnih et. al. https://arxiv.org/abs/1602.01783

[7] https://blog.openai.com/baselines-acktr-a2c/

[8] [50] Prioritized Experience Replay by Tom Schaul, John Quan, Ioannis Antonoglou, David Silver https://arxiv.org/abs/1511.05952

[9] [51] Deep Reinforcement Learning with Double Q-learning by Hado van Hasselt, Arthur Guez, David Silver https://arxiv.org/abs/1509.06461

[10] [52] Dueling Network Architectures for Deep Reinforcement Learning by Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas https://arxiv.org/abs/1511.06581

[11] [53] http://torch.ch/blog/2016/04/30/dueling_dqn.html

[12] [54] https://github.com/prasoon2211/pacman-RL

[13] [55] Deep Q-learning from Demonstrations https://arxiv.org/pdf/1704.03732.pdf

[14] [56] https://gym.openai.com/evaluations/eval_8Wwndzd8R62np8CxVQWEeg/