

**MSc in Artificial Intelligence and Machine Learning**

**CS6482 – Deep Reinforcement Learning**

**Assignment 3: Sem2 AY 23/24 - DQN for Atari**

**Pratik Verma – 23007575**

**Siddharth Prince – 23052058**

**Table of contents**

[1. Why Reinforcement Learning? 2](#_Toc1284496726)

[2. The Gym Environment: ALE/Breakout-v5 3](#_Toc1810025742)

[3. Implementation 5](#_Toc765775660)

[3.1 Capture and Pre-processing of Data 5](#_Toc1624368738)

[3.2 Network structure and hyperparameters 5](#_Toc1263823510)

[3.3 Q-learning updates 8](#_Toc100443514)

[3.3.1 Action Policy function 8](#_Toc1741273357)

[3.3.2 The Training Step 8](#_Toc1895153469)

[3.3.3 Other concepts: Replay buffer 9](#_Toc823389279)

[4. Results 10](#_Toc405326196)

[5. Experimentation: Dueling DQNs 11](#_Toc1997143462)

[5.1 Dueling DQN concept and architecture 12](#_Toc1432104924)

[5.2 Results and comparison 13](#_Toc1603530136)

[6. References 14](#_Toc1318603627)

# 1. Why Reinforcement Learning?

Reinforcement Learning (RL) is used to solve problems such as the one tackled in this report when the problem lacks any straightforward classification angle either due to the lack of labelled data for the task or the very nature of the task needing actions to be taken or interactions to be made with an environment. In other words, there is no precise solution to the problem and the model (agent) needs to produce approximations of an optimal strategy (policy). Training data is generated on the fly based on what the environment generates as the agent acts on the environment and explores different strategies that works and does not building up its knowledge or “experience”. As it builds up a set of winning strategies, it gets better and better at navigating the environment/tackling the task set out for it.

Speaking to this task specifically, the Atari Breakout game environment like other ALE Atari game environments requires that an agent/model learn how to play the game in the most optimal way. There is no scope for pre-determined labelled data because of the sheer number of possibilities with how the ball moves around the space. It is also difficult to think about how to label the frames of the game environment manually. Thus, it makes sense to go with a reinforcement learning approach with the specific RL approach here being [Deep Q Networks (DQNs)](https://arxiv.org/pdf/1312.5602).

# 2. The Gym Environment: ALE/Breakout-v5

The Gym environment of choice for this assignment is Breakout-v5. It is an Atari arcade environment where the goal is to destroy the brick wall at the top by hitting and thereby destroying each brick with a ball until all are destroyed. There is a floating paddle at the bottom of the space which can be moved left or right to bounce the ball that is served at the start of the round. There are a total of 5 balls (lives) that are available in each game episode. One life is lost if the ball falls into the space below the paddle because the paddle missed bouncing it. The detailed specifications for this environment are as follows [1].

**Action Space:**

* The action space is discrete.
* By default, all actions that can be performed on an Atari 2600 are available in this environment. However, for the Breakout environment, we will only take a subset of 4 actions from the total 18 controller inputs. These actions are:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Value** | **Meaning** | **Value** | **Meaning** | **Value** | **Meaning** |
| 0 | NOOP | 1 | FIRE | 2 | RIGHT |
| 3 | LEFT | - | | | |

* The above actions are self-explanatory with LEFT and RIGHT to move the paddle across the screen in the respective directions. FIRE is to serve a ball and get a round started. The player has 5 balls per game. NOOP is the usual “no operation”.

**Observation Space:**

* The observation space is an RGB image (colour image) with dimensions (210, 160, 3).
* Each pixel in the image represents a colour value (ranging from 0 to 255) for red, green, and blue channels.
* It can also be observed by the grayscale version of the image, which has dimensions (210, 160).

**Rewards:**

* Points are scored by destroying bricks in the wall.
* The reward for destroying a brick depends on the colour of the brick. They are as follows as seen from the [Atariage documentation for Breakout](https://atariage.com/manual_html_page.php?SoftwareID=889).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Brick Colour** | **Points** | **Brick Colour** | **Points** | **Brick Colour** | **Points** |
| Red | 7 | Orange | 7 | Yellow | 4 |
| Green | 4 | Aqua | 1 | Blue | 1 |

**Episode Termination:**

The episode finishes if

* The ball goes out of bounds. Though there are 5 lives, losing one life terminates it ensure the agent learns to not loose lives.
* All the bricks in the wall are destroyed and the maximum score of 864 is reached.

The starting state of this environment can be seen in figure 2.1 below which shows the paddle starting at a random position and the ball not yet served.

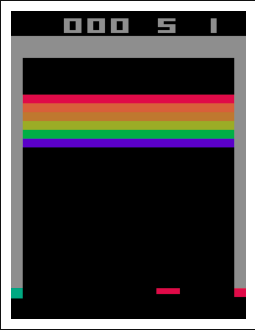


Fig 2.1: Starting state of the ALE/Breakout-v5 environment (seed=42)

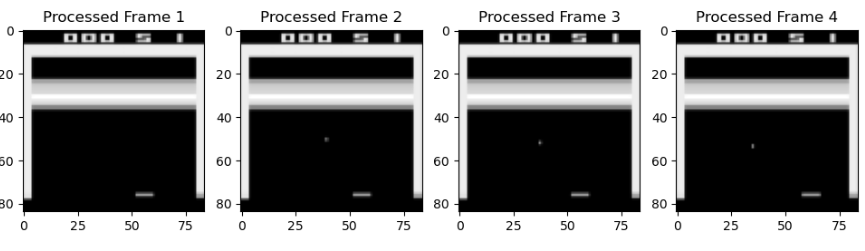
# 3. Implementation

The DQN agent implementation to reliably be able to solve the problem posed for the environment is discussed in this chapter.

## 3.1 Capture and Pre-processing of Data

As discussed in the specifications above, the observations are made from frames at each given step in the form of an image in a training episode. The default RGB image of the frame is again shown in Figure 2.1.

When it comes to actual training on these captured frames of data, we do not need it to have the feature complexity of it being an RGB image. It can be converted to grayscale to get the job done. Also, resizing the frame to an even 84x84 square shape simplifies the input which will be fed into the neural network. Another important aspect to consider is that there will not be enough context for the agent to know the properties of the ball such as the direction of travel with just a single frame. Hence, we have a frame buffer where four frames are always taken as the observation instead of just one. Figure 3.1 below shows the four pre-processed frames in a single frame buffer plotted.

Fig 3.1: Pre-processed frames in frame buffer

## 3.2 Network structure and hyperparameters

For the neural network architecture, we referred to the architecture described in the [original DeepMind paper](https://arxiv.org/pdf/1312.5602). In the paper, the number of output channels in the first convolutional layer is set to 16 and that of the second layer as 32. However, in almost all practical implementations and examples we came across for the network, the number of output channels are doubled to 32 and 64 in the respective layers. We tried searching for a paper that proposed this change to the original paper but were unsuccessful in doing so. But given the prevalence of this choice in practice, we decided to go for it taking the liberty of chalking this up to empirical exploration by the research community. The Figure 3.2 architecture summary.

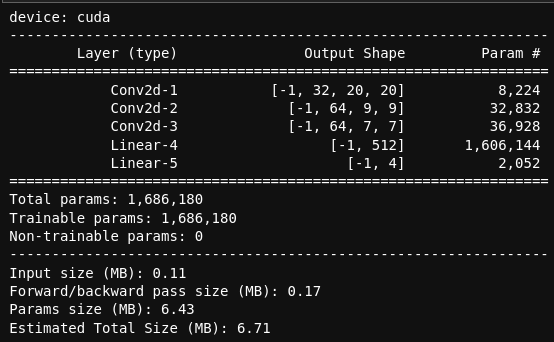


Fig 3.2: DQN architecture summary

Getting into the architecture itself, there are three convolutional layers *conv1*, *conv2* and *conv3* that process the raw pixel frames (grayscale images) from the emulated environment. These layers extract features from the images by applying convolutional filters. The kernel sizes (8, 4, and 3) and strides (4, 2, and 1) control the spatial down sampling and feature extraction.

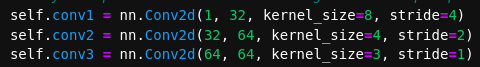


Fig 3.3: Convolutional layer definitions for the DQN

The fully connected linear *fc1* and *fc2* layers take the flattened output from the convolutional layers and produce Q-values for each action as the output of the final fully connected layer based on the action space defined for the environment. Layer *fc1* has 512 hidden units while *fc2* has 4 because the action space for Breakout is only 4 as discussed in the environment specifications in Section 2. The flattened output dimensions are 64 \* 7 \* 7 since the stride and kernel size of the 3rd layer being 1 and 3 leads to feature maps of dimension 7 x 7.



Fig 3.4: Fully connected layer definitions for the DQN

The activation function of choice is Rectified Linear Unit (ReLU) which is applied for all convolutional layers and the first fully connected layer. ReLUs are a popular choice for deep neural networks in general because it introduces non-linearity and helps the network learn complex representations. The ReLU activation functions being called in the forward() function definition can be seen in the code snippet in Figure 3.5.

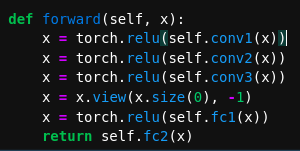


Fig 3.5: Forward function inside the DQN PyTorch implementation

Also in the forward function from Figure 3.5 is the code line,

*x = x.view(x.size(0), -1)*

This line above is responsible for flattening the output of *conv3* layer. Flattening converts the 2D output feature maps into a 1D vector so that it can be using in the linear fully connected layer below.

In this implementation there is the use of two DQN networks (double DQN) where one is the policy network and the other is the target network. The “older” target network is used to compute the target value for the state while the policy network is the one with the latest weight updates which is used to choose the next action to be taken based on the current state for every step by predicting the corresponding Q value for each action. The target network is updated with the weights of the policy network after a set number of episodes defined by the condition, `*(episode + 1) % TARGET\_UPDATE == 0`* as seen in Figures 3.6 and 3.7 below which essentially is updating the network every n number of times as defined by the hyperparameter, TARGET\_UPDATE.



Fig 3.6: Code snippet of the target network update method being called

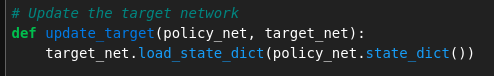
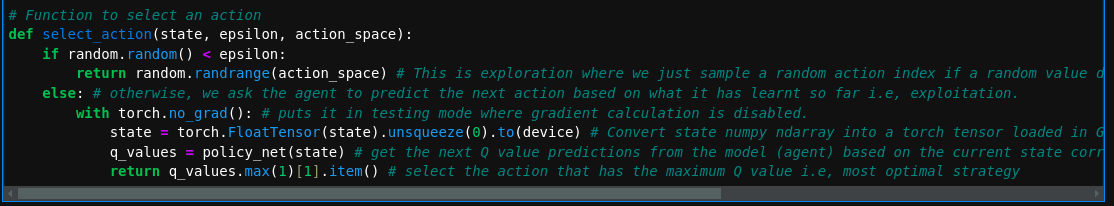


Fig 3.7: Code snippet of the target network’s weights being updated

## 3.3 Q-learning updates

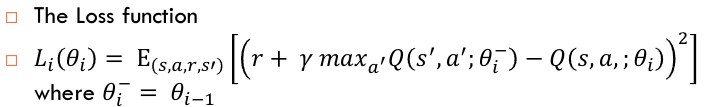
### 3.3.1 Action Policy function

The policy function determines the answer to exploration vs exploitation. This condition hinges on the set Epsilon value. We are using a decaying epsilon where its value is high when the agent is in its nascent stages and does not have a minimum number of samples in the memory buffer or lacks “experience”. This increases the probability of the agent sampling random actions from the action space instead of using a predicted value from the model. But as the number of episodes run increases, the epsilon value decreases till it reaches a minimum value that is set manually by updating the current epsilon value using the EPSILON\_DECAY hyperparameter i.e., multiplying current epsilon with the decay parameter. At this point, the focus is more on the agent “exploiting” the learning it has done from the saved experience. The code snippet in figure 3.8 below demonstrates this.

Fig 3.8: Epsilon Greedy Policy definition

### 3.3.2 The Training Step

The optimise\_model() function is responsible for getting the RL DQN agent trained by calculating the target Q value for all actions in the action space given the current state via the target network. The loss between the predicted Q values from the current state and the predicted Q values from the target network and the [formula below (week 9 - DQN for Atari)](https://learn.ul.ie/d2l/le/lessons/17967/topics/654775).

Fig 3.9: Loss function for DQN

The code snippet in Figure 3.10 below is the implementation of the training and optimisation of the DQN.

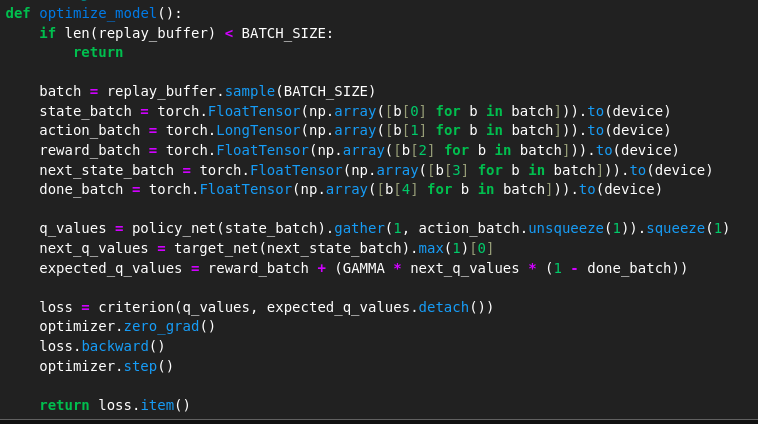


Fig 3.10: Code snippet of the optimize\_model function

The mean square error loss function is used over the difference of the predicted and expected Q values. `*criterion`* is just a variable place-holder for the `*nn.MSELoss()*`loss function defined further below in the notebook. The code line,

*expected\_q\_values = reward\_batch + (GAMMA \* next\_q\_values \* (1 - done\_batch))*

Represents the calculation of the target/expected q values after passsing the next state to the target network.

### 3.3.3 Other concepts: Replay buffer

The replay buffer is used to store already explored states related information such as the corresponding action that was applied to the state, reward received after action was applied, the next state the action resulted in and if the current action resulted in an episode termination because of failure or success. A random batch of these saved “experiences” is then sampled according to the BATCH\_SIZE hyperparameter defined whose state, action, done status and reward batches are used to get predicted and target Q values as seen in Figure 3.10 in the optimisation step.

The ReplayBuffer class defined in the code snippet shown in Figure 3.11 initialises a deque data structure according to the max length defined i.e., the BATCH\_SIZE parameter. There are methods to add samples and get random samples from the replay buffer.

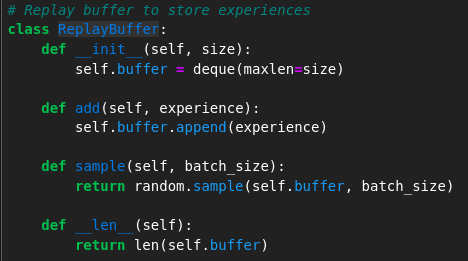
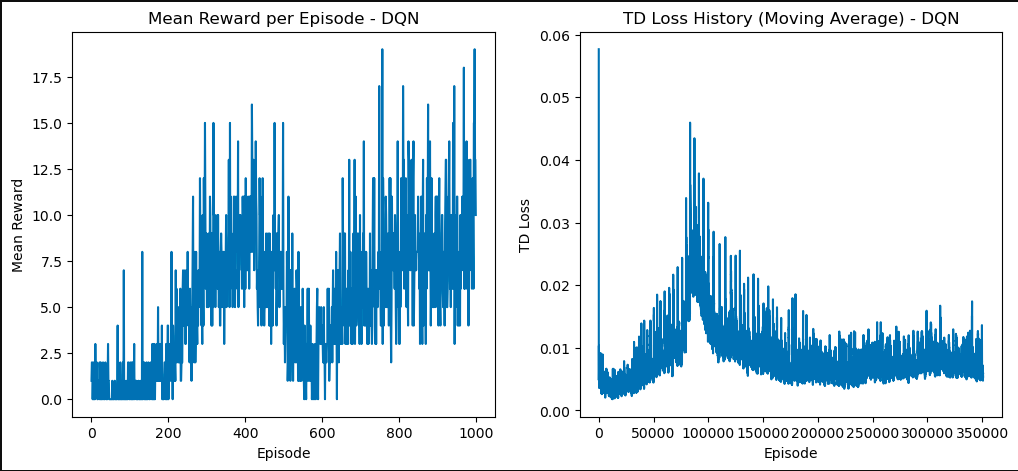


Fig 3.11: Code snippet of ReplayBuffer class

# 4. Results

The training was run only for a thousand episodes which is not much in terms of getting the agent to learn the optimal policies for a complex game environment such as Atari’s Breakout relatively speaking for a machine learning RL agent. The results below as seen in figure 4.1 show the mean reward achieved per episode of training and the computed loss.

Fig 4.1: Reward and loss result metrics

For the small number of episodes, the rewards from episode to episode are very erratic and seem to be varying by a large margin. This is probably because of the exploration and that the agent had not found a good enough stable policy for it to converge upon. The maximum achieved reward value is also about only 19 which is exceptionally low and not optimal at all considering the possible high score in the game being 864 when all bricks have been destroyed. Some hyper parameter tweaking should be able to improve this. For example setting the minimum epsilon value to a much lower 0.01 probability rather than the 0.1 percentage could encourage it to focus more on the stable and already built-up set of policies. Also, just running the training for more episodes significantly to about 10,000 to 50,000 episodes could frankly increase the resulting efficacy of the agent in being able to play the game. But that is currently difficult for us given the hardware and time constraints.

# 5. Experimentation: Dueling DQNs

When exploring the more recent development of Dueling DQNs, we understood that they are an extension of the standard DQN architecture. They employ separate “value” and “advantage” functions to improve learning efficiency. The value function estimates the state’s overall value, while the advantage function captures action-specific advantages. By decoupling these components, Dueling DQNs in theory enhance performance in reinforcement learning tasks.

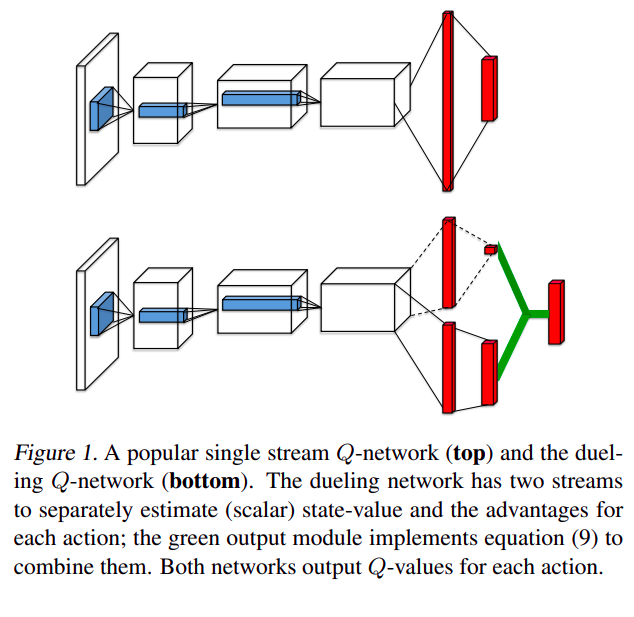


Fig 5.1: Difference between a normal DQN and Dueling DQN architecture [(ref)](https://arxiv.org/pdf/1511.06581)

## 5.1 Dueling DQN concept and architecture

To implement a Dueling DQN in our code, we needed to make a minor change to the existing DQN architecture (see Figure 5.3 for the code snippet). We needed to add an extra `*value\_stream*` layer parallel to the usual fully connected `*advantage\_stream*` (*fc2* layer from the normal DQN architecture) which gets the outputs for the actions in the action space. Finally, the values are combined using the following formula which is the Q value output.

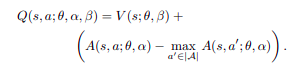
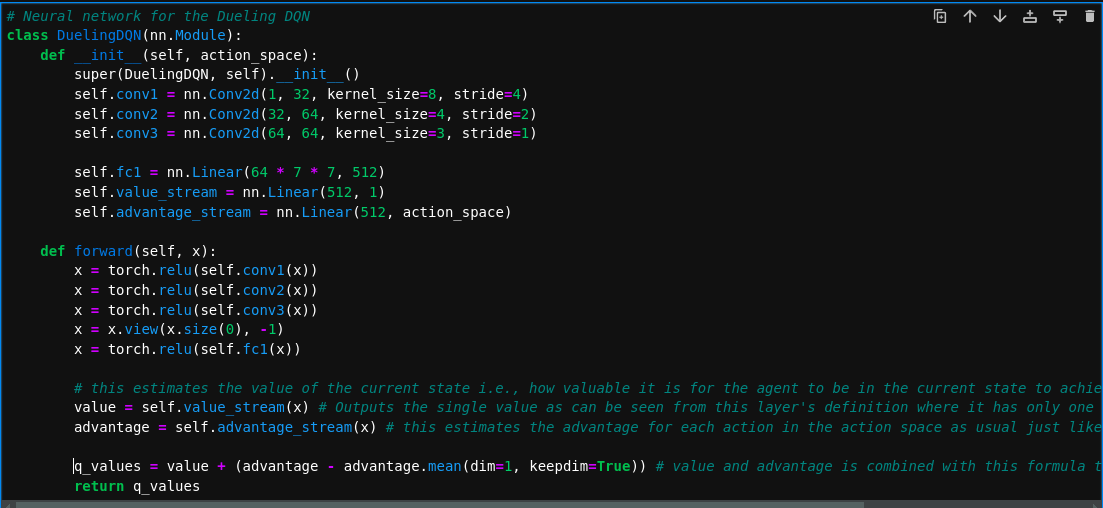
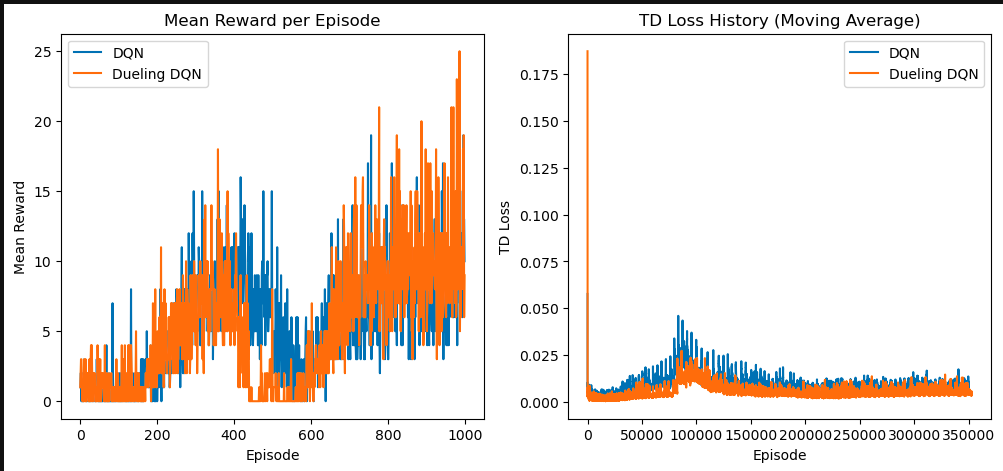


Fig 5.2: Dueling DQN formula [(ref)](https://arxiv.org/pdf/1511.06581)

Fig 5.3: Code snippet of the Dueling DQN architecture

## 5.2 Results and comparison

When testing our implementation on the same 1000 episode we had better success compared to just the Double DQN approach before. The maximum achieved reward was noticeably higher than before as can be seen from the comparison graph that has been plotted in Figure 5.4.

Fig 5.4: Comparison plot between the performance of Double DQN and Dueling DQN implementations

As we can see the mean reward per episode reached a maximum of 25 with the Dueling DQN as opposed to 19 for the Double DQN implementation. The loss was also lower on average across the board. This proves that this innovative approach has furthered the field of DQNs in reinforcement learning.

# 6. References

1. The Gymnasium documentation for Mountain Car environment: <https://gymnasium.farama.org/environments/classic_control/mountain_car/>
2. A Geron’s ML notebook for Reinforcement Learning: <https://github.com/ageron/handson-ml2/blob/master/18_reinforcement_learning.ipynb>
3. PyTorch tutorial for DQN: <https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html>
4. GitHub repository containing RL implementation of DQN with PyTorch: <https://github.com/lazavgeridis/LunarLander-v2/blob/main/train.py>
5. Efficient memory-based learning for robot control, Andrew William Moore; University of Cambridge Technincal Report – November 1990: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-209.pdf>