Playing MsPac-man with Deep Reinforcement Learning

Stephen Charles
Department of Computer Science
University of Bath
Bath, BA2 7AY
scm64@bath.ac.uk

Max Ashton-Lelliott
Department of Computer Science
University of Bath
Bath, BA2 7AY
maal21@bath.ac.uk

Peter Amante-Roberts
Department of Computer Science
University of Bath
Bath, BA2 7AY
ptear20@bath.ac.uk

Kwabena Ohene-Bonsu
Department of Computer Science
University of Bath
Bath, BA2 7AY
kgob20@bath.ac.uk

Hiba Lubbad
Department of Computer Science
University of Bath
Bath, BA2 7AY
hl2377@bath.ac.uk

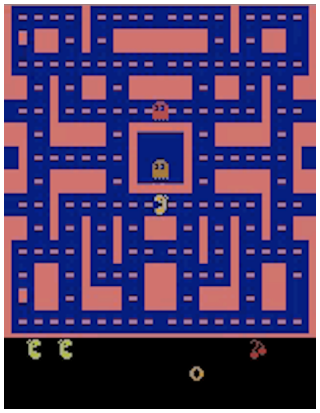

Figure 1: The Atari 2600 game MsPac-man.

Abstract

The first version of the Deep Q-Network by DeepMind (Huang, 2013) was capable of human-level performance on a number of classic Atari 2600 games. The algorithm used a CNN architecture which based its strategy from vision, much like a human player. Training from scratch with no prior knowledge of the environment, it discovered strategies that enabled it to exceed human benchmarks. Since then, many refinements and optimisations have been attempted, and we aim to benchmark some of the recent advancements with the MsPac-man environment from the python package `Gym` (Brockman et al., 2016).

1 Problem Definition

1.1 MsPacman Atari 2600

MsPac-man (Figure 1) is a game where the title character is tasked with eating pellets in an enclosed maze while avoiding four ghosts. Contact with a ghost causes the player to lose a life. Larger power pellets are available, which let the player eat the ghosts. When the larger pellet is consumed the ghosts turn blue and flee. Bonus fruit also appears, and when eaten increases the reward multiplies. When all the pellets are consumed, the level ends and the next one begins.

As rounds increase, the speed increases and the large power pellets have a reduced timer for the ghosts vulnerability phase. Four mazes appear in different color schemes of which follow

- Pink: levels 1 and 2,
- Blue: levels 3, 4 and 5,
- Brown: levels 6-9,
- Dark Blue: levels 10-14.

Three of the mazes have two sets of warp tunnels, when the player enters and appears from the alternate side.

The ghosts have different colors, and behave in different ways. Blinky (red) directly chases the player, Pinky (pink) and Inky (blue) try to position themselves in front of the player and Sue (orange) switches between chasing and fleeing from the player.

2 Background

2.1 Actor Critic

The first approach we take is utilising an asynchronous advantage actor-critic (A3C) (Mnih et al., 2016) agent. This is an actor-critic method, which combines learning approximations of both policy and value functions (Sutton, 2018). This differs from the exclusively value-based approach of DQNs (Mnih et al., 2013). The learned policy is referred to as the 'actor' and the learned value function is referred to as the 'critic'.

As covered by Sutton (2018), integrating direct approximations of action probabilities via the 'actor' enables the agent to find a stochastic optimal policy, which an action-value method could not do naturally. We might expect this to be of particular benefit to our MsPac-man environment since, as mentioned above, the orange ghost switches between chasing and fleeing from the player - and all ghosts switch back to chasing after a time following the consumption of a large pellet. Hence, an element of stochasticity might allow our agent to better avoid ghosts, as it will not necessarily assume that a ghost is running away.

A3C achieved much better results than previous GPU-based algorithms and showed better efficiency at the time Mnih et al. (2016) introduced the method. The asynchronous advantage actor-critic (A3C) has proved to perform well on Atari games while using a forward view instead of the typical view often used by techniques such as eligibility traces.

The asynchronous RL framework uses asynchronous actor-learners that use multiple CPU threads on a single machine instead of multiple machines. Instead of relying on replay memory, this framework relies on parallel actors that employ different exploration policies thus maximizing diversity and stabilizing learners. This method is advantageous as it allows for training deep neural networks reliably without large resource requirements while also removing communication costs between CPUs.

2.2 Deep-Q Networks

Neural Networks have proven to be universal function approximators (Hornick, 1989). DQNs build on this success by splitting the architecture into constituent parts (Brockman et al., 2016). Convolutional layers learn to detect abstract features of the game screen, then a dense classifier creates a map of the

features present from the currently observed screen to an output layer equal to the size of the action space.

The outputs generated by the action space represent the agent's best guess of the reward it expects if it were to take that specific action. This process is known as *Value Based Learning*, defined by the action-value function $Q^*(s, a)$ (Equation 1). Put simply, it represents the total value of taking a state action pair (s, a) which is the sum of all future rewards R, scaled by the discount factor γ, which defines how far into the future the agent can plan.

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\left[R_t + \gamma_t R_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi\right] \tag{1}$$

The inability for the agent to see future states is solved by the deep learning process, mapping the pixel images (game screen) to the Q-values and creating an approximation of the Q-function itself. This allows the agent to make predictions on future state action pairs and their associated rewards.

The loss by which this is minimised through gradient descent is a variable of *Temporal Difference* (Equation 2). Its loss is the difference between the true Q values and the estimation of them, where the goal is to achieve the reward plus the Q value of the next state action pair.

$$L_i(\theta_i) = \left[R_{t+1} + \gamma_{t+1} \max_{a'} Q_{\theta}^{-}(S + t + 1, a') - Q_{\theta}(S_t, A_t)\right]^2 \tag{2}$$

As the fundamental nature of RL creates an ever-changing dataset, modifications to the basic DQN have been made over the years to stabilize the Q table approximations. We aim to implement all of the modifications published by the Rainbow (Hessel et al., 2018) paper where the performance improvements are shown in Figure 2.

A few of the modifications are:

- *Dueling* (Wang et al., 2015): Two identical branches are created where one branch deals with the target values, and the original processes the estimations. Only the original is trained, and the target value branch receives updated weights in a piecemeal fashion.
- *Experience Buffer* (Schaul et al., 2015): This buffer represents all of the agents past experience, defined as $(S_t, A_t, R_{t+1}, \gamma_{t+1}, S_{t+1})$ (state, action, reward, terminal, next state). Two new terms are saved, which lets the agent remember when the terminal state is reached, and what happened in its next state when it took an action from its current state last time.

As a result of the architectural decisions made, the DQN is classified as an *off-policy, model-free* algorithm. It learns to predict the value associated with position, but doesn't build a model of its environment in order to make those predictions (model-free), and the training examples are created from a previous version of itself (off-policy).

To account for exploration, DQNs initialise a variable epsilon $\epsilon = 1.0$. For each step, a random number is chosen $\in (0, 1)$, which if less than ϵ will cause the agent to take a completely random action, ignoring the Q table entirely. In the initial stages, this happens every time, due to the initialisation of $\epsilon = 1.0$. As training continues though, ϵ is annealed down to a much lower figure, decreasing the odds of the agent taking a random action, and instead following the Q table. Such exploration is known as *Epsilon Greedy*.

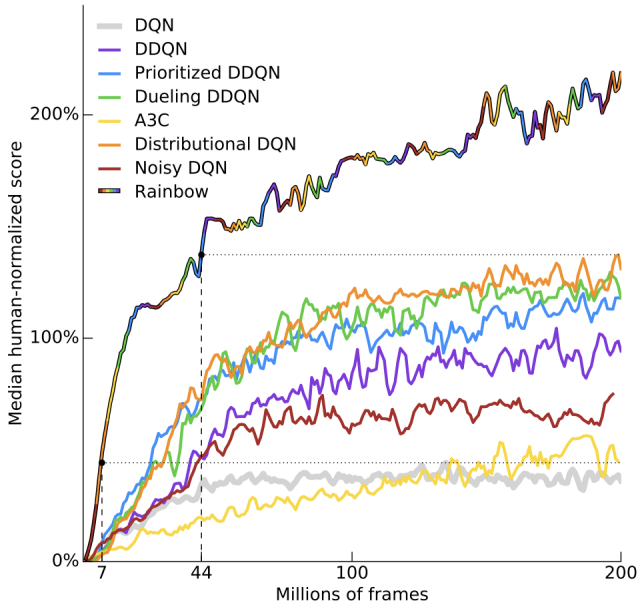

Figure 2: The median human-normalized performance across 57 Atari games (Hessel et al., 2018).

3 Method

The two methods we chose to implement are Asynchronous advantage actor-critic (A3C) and Rainbow.

According to Figure 2, from Hessel et al. (2018), these were the highest performing methods when applied to Atari games (at the time of writing). This provided the motivation for our choices: proven historical performance. Furthermore, Hessel et al. (2018) shows that the variety of modifications that Rainbow introduces are all beneficial for MsPac-man specifically (for some Atari games this is not the case).

As for A3C, and as mentioned in Section 2.1, we would expect a stochastic policy introduced by the 'actor' part of A3C to benefit our agent; given the stochastic behaviour of the ghosts.

3.1 Actor Critic - A3C

As covered above, our first agent uses the A3C algorithm introduced by Mnih et al. (2016), which maintains a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V(s_t;\theta_v)$. A3C has proven to perform well on Atari games while using a forward view instead of the previously typical view often used by techniques such as eligibility traces (Sutton, 2018). The majority of code that we have found that implements this algorithm is often messy, complex, and difficult to understand. Therefore, we decided to utilise a much more concise and simple code for our project.

The policy and the value function for this method are updated after every t_{max} action or when a terminal state is reached, where t_{max} a hyper-parameter we choose (chosen to be 20, as opposed to 5 used by Mnih et al. (2016)). The update performed by the algorithm can be seen as:

$$\nabla_{\theta'} log\pi(a_t|s_t;\theta^{'})A(s_t,a_t;\theta,\theta_v) \tag{3}$$

Included in our implementation is entropy regularization, as suggested by Mnih et al. (2016), who found that including this "improved exploration by discouraging premature convergence". Including this feature, the gradient of the full objective function including the entropy regularization term with respect to the policy parameters takes the form (with H as entropy):

$$\nabla_{\theta'} log\pi(a_t|s_t;\theta^{'})(R_t - V(s_t:\theta_v)) + \beta\nabla_{\theta'} H(\pi(s_t;\theta^{'})) \tag{4}$$

To implement A3C as outlined above, we adapt code from Greydanus (2017). For the actor and critic neural network architectures, a series of convolution layers are followed by GRU cells. A GRU cell is used "because it has fewer parameters, uses one memory vector instead of two, and attains the same performance as an LSTM cell" (Greydanus, 2017). The only difference between the architectures is that the activation layers have 9 and 1 outputs for actor and critic respectively. This is because the actor model needs to be used to obtain probabilities for each of the 9 actions, whereas the critic only needs to output a single value estimate.

Preprocessing is performed on the raw pixel inputs representing the state the game is in. Firstly, the bottom section of the screen is cropped out, as this is almost completely static (it only contains the number of lives and items the agent has obtained). Then, the 3 array layers representing the rgb input are flattened into a single gray-scaled layer. This decision was made to improve efficiency of training. However, we might expect that this will marginally impact performance, noting that the different coloured ghosts behave slightly differently. Finally, some resizing and normalising is performed.

The optimizer used to train the actor and critic model weights is Adam (Kingma and Ba, 2014). This is in contrast to the RMSProp optimizer used in the original Mnih et al. (2016) paper for A3C. Kingma and Ba (2014) illustrate that Adam effectively integrates the ideas of RMSProp and improves upon it via a variety of metrics. Hence, we would expect that - had A3C been developed after Adam was introduced - Mnih et al. (2016) would also have used it.

As with the Rainbow approach detailed below, a modification to the environment we make for our A3C agent is the inclusion of a death penalty. It was hoped that this would improve performance by encouraging the agent to avoid the ghosts. We hypothesized that the agent might sometimes utilise the ghosts as a means to 'teleport' back to the starting location. Or on a more basic level, it might not see the ghosts as being detrimental to its reward seeking objective.

3.2 Rainbow

Deep-Q Networks (Huang, 2013) are an algorithm subset of *Reinforcement Learning* (Sutton and Barto, 2014). In RL, an agent is placed into an environment and seeks to maximize a pre-defined reward. An action is taken, which alters the environment and a reward is given in alignment with the altered state. The new state is then observed and a new action taken, completing the cycle until a termination condition is met inside the environment. Such a decision process is known as a *Markov Decision Process*.

Video games such as MsPacman however, are technically *partially observable* MDPs, as you are forced to make choices based on the screen rather than the underlying code defining the environment itself. Games such as these are constrained by a finite amount of input actions and frame rates which allow us to map large observation spaces into more manageable discrete action spaces.

The Rainbow paper (Hessel et al., 2018) by DeepMind examined six enhancements to the DQN algorithm, and studied different combinations of their implementation. The combination provided state-of-the-art performance, of which the components are outlined in Appendix E. The implemented base DQN architecture is shown in Figure 6, and the rainbow architecture is shown in Figure 2.

3.3 Modifications to Rainbow

Penalising Death

Mnih et al. (2013) states that

> "Since the scale of scores varies greatly from game to game, we fixed all positive rewards to be 1 and all negative rewards to be -1, leaving 0 rewards unchanged. Clipping the rewards in this manner limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. At the same time, it could affect the performance of our agent since it cannot differentiate between rewards of different magnitude."

We chose to not use this scaling feature in our use case as we wanted to penalise death significantly to allow the agent to avoid ghosts. The wrapper used for this is shown in Listing .1.

Color Channels

As the individual ghosts have separate AI based upon their colour, we chose a strategy of adding the colour channels to allow the agent to detect this variation.

The rainbow paper used a sequence of four game frames stacked together, making the dimension $(4, 84, 84)$. The action that agents choose depends on a prior sequence of events. This allows the detection of direction and movement.

There is also a frame skipping parameter, of which each state consists of 4 consecutive overlapping frames. For example, state $s_1 = (x_1, x_2, x_3, x_4)$ is used as one state, and the next state uses $s_2 = (x_2, x_3, x_4, x_5)$. Lastly the component wise maximum is taken between the last two consecutive frames, as some atari games render information every other frame. This step is not as crucial now, as no frame skip environments have been added to gym recently.

We chose to use this frame stacking approach and incorporate the colour channels into an input of $(12, 84, 84)$, of which we use frame skipping of 9 to have 12 consecutive overlapping frames.

4 Results

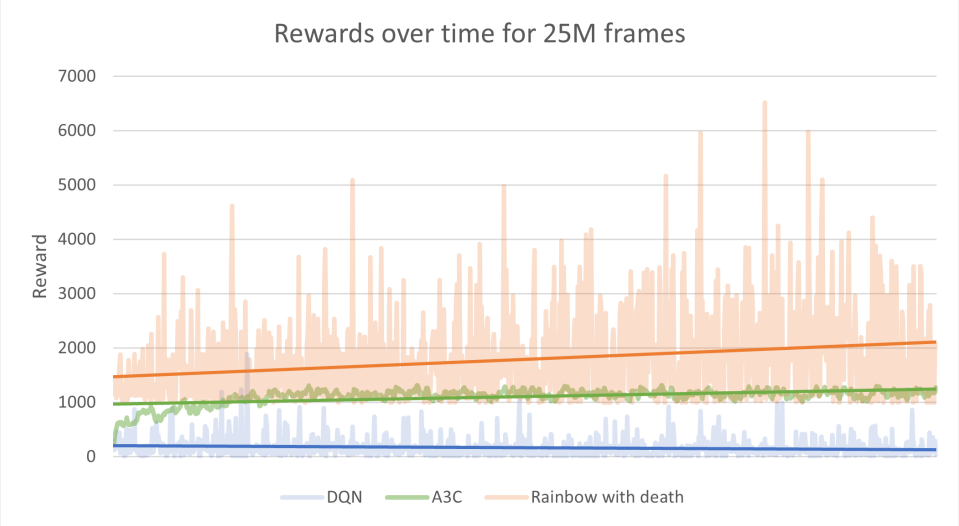

Figure 3: Average undiscounted return for 25m frames of training

Figure 3 shows the results of our agents' learning. The basic DQN agent with no modifications performs very poorly. Whilst achieving a high score over 1000, it does not learn at all (as evidenced by the flat curve).

Our A3C agent converges fairly quickly after around 8 million frames - with a high score of 3020 (not illustrated in 3 as this displays only average rewards). This compares favourably with the A3C agent from the original Mnih et al. (2016) paper, which achieves a high score of 850.7.

The Rainbow agent performs much better, achieving a high score of 7070 (as shown in the video). This does not appear in Figure 3 due to the difference in performance from training to testing parameters - more 'exploitation' is encouraged during testing. We can also observe that the agent has not yet converged to a policy - this provides encouragement that better performance could be achieved, given more time to train.

Our best overall score of 7070 compares very well with the results from a number of agents for MsPac-man. As can be seen in Stojnic (2022), this places our performance at 12th on the ranking list. This places it above DDQN, A3C (as shown by our implementation and the original paper), A2C and Gorila, to name a few. The score of 7070 is roughly comparable to that of the average human (we obtained an average performance of 6300 within our group).

5 Discussion

Overall, we can see that our high score of 7070 is better than all but one paper released prior to 2018 (Stojnic, 2022). Our implementation of Rainbow can therefore be said to be utilising reinforcement learning methods at the forefront of current research. As shown by Hessel et al. (2018), who achieve a high score of around 6000 using Rainbow, the various modifications included in Rainbow are all beneficial for MsPac-man. The suitability of Rainbow to our problem is a large contributing factor to our success in this project.

Our A3C agent obtains a score higher than that of the original paper (Mnih et al., 2016). This improvement can be attributed to the improvement in optimizer (the original paper used RMSProp whereas our implementation uses Adam), and the inclusion of a death penalty. We can also assume that the results obtained by Mnih et al. (2016) in the A3C paper follow the same [-1, 1] clipping procedure as in DQN (Mnih et al., 2013), which will contribute to its weaker performance for MsPac-man. Overall, we can say that our agent is more suited to the specific MsPac-man environment, as it applies modifications more specific to the problem, which the agent in the original paper does not.

Both our Rainbow agent and A3C agent perform very well compared to the scores reported in their respective papers, as mentioned above in the Results section. Therefore, we can say that our implementations have been successful, particularly considering the fact that our high score of 7070 is comparable to the performance of a human. However, when compared to the 'solved' score of 999990, achieved by van Seijen et al. (2017), we are still a long way off. As a side note, it is worth mentioning that van Seijen et al. (2017) also incorporate a death penalty (a -1000 reward); validating further the similar approach we took.

One factor that we should also take into account when considering our performance is the length of training for each agent. We had limited time and resources to train our agents and therefore fell far short of most papers training lengths which can number in the 200M (see Mnih et al. (2016)) - 20B (see 'MuZero' (Schrittwieser et al., 2020)) frames of training. Considering that the main weakness of our final agent was that it became confused in a new environment it may be that further training to this industry convention could enable higher performance. However, an ideal agent would be one which learns from as little interaction with the environment as possible, so this requirement for further training may indicate that another method or further modifications would also benefit our performance.

6 Future Work

Attempting to implement an Importance Weighted Actor-Learner Architectures (IMPALA) (Espeholt et al., 2018) agent for MsPac-man would be the most obvious choice for future work with regards to this project. In their paper, Espeholt et al. (2018) show that a deep IMPALA agent achieves a score of 7342.32 on MsPac-man. As covered in the paper, IMPALA vastly improves on the performance of older actor-critic methods such as A3C. This is borne out in the performance of our A3C agent, as well our Rainbow agent; both of which do not achieve a score as high as that reported in the IMPALA paper.

One of the key improvements IMPALA makes is the ability to "scale to thousands of machines without sacrificing data efficiency or resource utilisation" (Espeholt et al., 2018). It is shown in the paper that IMPALA can process around 250000 frames per second, compared to around 50000 frames per second for a3c, over a distributed, multi-machine setup. Whilst speed is not necessarily of vital importance, when combined with the improvement in overall score, it can be said that IMPALA is a universally superior method when compared to our a3c and Rainbow agents.

However, one key feature of IMPALA that would not be applicable to our problem is the "multi-task" approach IMPALA takes. Via this approach, IMPALA trains on multiple environments at once - for example the 57 ATARI games in the ALE (Bellemare et al., 2013) suite - achieving "positive transfer" between tasks. Since we are only interested in the MsPac-man environment, this would not be necessary.

The random network distillation (RND) (Burda et al., 2018) method which integrates "intrinsic rewards" would also be of interest. Burda et al. (2018) test the performance of RND on several ATARI games, but not MsPac-man. Therefore, it would be illuminating to test an implementation of

RND on MsPac-man. In particular, it might be expected to improve upon performance in the later stages of a game; where our current A3C and Rainbow agents might struggle to deal with sparse rewards. In addition, when the environment changes slightly upon clearing a level, it might benefit the agent to be curious to explore new areas.

Modifying the frame-stacking approach used by DeepMind (Huang, 2013) may also prove to be a beneficial approach for MsPac-man specifically. As each of the ghosts have unique AI, it could prove advantageous for the agent to find new strategies associated with predictive movements and behaviours. We attempted to implement this, in a similar vein using frame stacking, but the performance was less optimal given our constraints on training time. A better approach may be to have three separate networks for each RGB channel which then share information between each other and update the Q tables with the most beneficial states and actions.

7 Personal Experience

Peter: My experience of the project was a largely positive one. Working on the actor-critic agent allowed me to solidify my knowledge of this family of methods, whilst attempting to train an agent for MsPac-man improved my understanding of how and why RL methods struggle. My only disappointment would be the failed attempt at training an agent to play the "Amazons" (Huang and Li, 2019) board game.

Max: As most of the rainbow agent training was done on my machine I probably observed the most games of MsPac-Man being played. While often infuriating watching the agent constantly miss optimal plays or in some extreme cases choose to change direction straight into a ghost. It was interesting to see the impacts that each of our modifications had on the agent. Going from essentially guessing what to do at each step to strategically avoiding the ghosts and using power-ups as a shield.

Stephen: Initially we attempted to write an environment from scratch for the "Game of the Amazon" (Huang and Li, 2019), but this proved too difficult to implement within the time scale, and the focus of the assignment was implementing the policy itself. I worked on writing the DQN and agent, of which we extended to include all of the Rainbow (Hessel et al., 2018) components as the ablation studies showed that all components positively contributed to agent performance. My attempt to incorporate all RGB channels was not successful, but for this environment in particular I feel perhaps that each channel could have its own independent branch, to detect variations in AI from each of the ghosts. I'm pleased with the replication of the Rainbow paper though, and I feel the assignment helped solidify my understanding of DQNs in general.

Kwabena: Working on this project improved my understanding of the RL methods implemented. Comparing the performance of the actor-critic and rainbow methods in our case, saw the high score of the agent move from 3020 in the actor-critic, to 7070 in the rainbow method with the effect of the death penalty having a greater impact in the rainbow method.

Hiba: I don't have much to add to what has already mentioned. I thoroughly enjoyed the project. Most of the difficulties I encountered involved working with OpenAI Gym!

References

Marc G. Bellemare, Will Dabney, and Rémi Munos. 2017. A Distributional Perspective on Reinforcement Learning. https://doi.org/10.48550/ARXIV.1707.06887

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research* 47 (jun 2013), 253–279. https://doi.org/10.1613/jair.3912

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie a Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv* (2016). arXiv:1606.01540

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. 2018. Exploration by Random Network Distillation. https://doi.org/10.48550/ARXIV.1810.12894

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. 2018. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. https://doi.org/10.48550/ARXIV.1802.01561

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. 2017. Noisy Networks for Exploration. https://doi.org/10.48550/ARXIV.1706.10295

Sam Greydanus. 2017. Baby A3C: solving Atari environments in 180 lines. https://github.com/greydanus/baby-a3c.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2018. Rainbow: Combining improvements in deep reinforcement learning. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018* (2018), 3215–3222. arXiv:1710.02298

Kur Hornick. 1989. Multilayer Feedforward Networks are Universal Approximators. *Practical Neural Network Recipies in C++* (1989), 77–116. https://doi.org/10.1016/b978-0-08-051433-8.50011-2

Hebin Huang and Shuqin Li. 2019. The Application of Reinforcement Learning in Amazons. In *2019 International Conference on Machine Learning, Big Data and Business Intelligence (MLBDBI)*. 369–372. https://doi.org/10.1109/MLBDBI48998.2019.00083

Yanhua Huang. 2013. Deep Q-networks. *Deep Reinforcement Learning: Fundamentals, Research and Applications* (2013), 135–160. https://doi.org/10.1007/978-981-15-4095-0_4

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. https://doi.org/10.48550/ARXIV.1412.6980

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. (2016). https://doi.org/10.48550/ARXIV.1602.01783

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. https://doi.org/10.48550/ARXIV.1312.5602

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized Experience Replay. https://doi.org/10.48550/ARXIV.1511.05952

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. 2020. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (dec 2020), 604–609. https://doi.org/10.1038/s41586-020-03051-4

Robert Stojnic. 2022. Atari Games on Atari 2600 Ms. Pacman. https://paperswithcode.com/sota/atari-games-on-atari-2600-ms-pacman.

A Sutton and A Barto. 2014. Reinforcement Learning: An Introduction. https://doi.org/10.48550/ARXIV.1312.5602

Richard S Sutton. 2018. *Reinforcement learning : an introduction* (second edition. ed.). Bradford Books, Cambridge, Massachusetts.

Hado van Hasselt, Arthur Guez, and David Silver. 2015. Deep Reinforcement Learning with Double Q-learning. https://doi.org/10.48550/ARXIV.1509.06461

Harm van Seijen, Mehdi Fatemi, Joshua Romoff, Romain Laroche, Tavian Barnes, and Jeffrey Tsang. 2017. Hybrid Reward Architecture for Reinforcement Learning. https://doi.org/10.48550/ARXIV.1706.04208

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. 2015. Dueling Network Architectures for Deep Reinforcement Learning. https://doi.org/10.48550/ARXIV.1511.06581

Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Hang Su, and Jun Zhu. 2021. Tianshou: a Highly Modularized Deep Reinforcement Learning Library. https://doi.org/10.48550/ARXIV.2107.14171

A Problem definition

A.1 Objective

Our objective in this project was to make an agent which could maximise its rewards before reaching a game over. A game over occurs when MsPac-Man has lost all 3 lives by coming into contact with a ghost 3 times. The main difficultly in this game comes from navigating each of the game's 14 levels (4 mazes) while avoiding the ghost enemies who roam around it. Each ghost has a different AI and rewards vary for each type of action meaning there is a large amount of information for the agent to consider. Additionally once lives are lost there is no way to get them back meaning that if an agent clears a maze with only one life remaining they will still only have one life going forward into the next maze.

A.2 Ghost AI

: There are 4 main ghosts in MsPac-Man, Blinky (red), Pinky (pink), Inky (cyan) and Sue (orange). These ghosts have have 2 distinct phases in their AI, scatter and normal behaviour.

During scatter:

1. Blinky and Pinky move randomly.
2. Inky and Sue move to designated corners of the level.

This scatter phase usually lasts for only the first few seconds of the level. Then they move permanently into their normal movement patterns, with very occasional bursts of changing direction to make their patterns unlearn-able.

During normal movement:

1. Blinky alternates between random movement and directly chasing MsPac-Man around the maze.
2. Pink chases towards the spot 2 Pac-Dots in front of Ms. Pac-Man trying to get close.
3. Inky's target is move complex, his target is relative to both Blinky and Ms.Pac-Man, where the distance Blinky is from Pinky's target is doubled to get Inky's target.
4. Sue chases directly after MsPac-Man, but tries to head to her Scatter corner when within an 8-Dot radius of her, which is at the bottom left of the maze.

A.3 Rewards

There are 3 main ways for MsPac-Man to earn rewards:

1. Picking up Pac-Dots these are the small rectangular blocks (small square shapes in the actual game) and are worth 10 points each. Additionally there are Power-Pellets these are the glowing square shaped blocks (circles in the actual game) which grant 50 points and turn ghosts into a fearful state.
2. When ghosts are in this fearful state they begin to move away from MsPac-Man who can now eat them to imprison them in the centre of the map and gain points. The first ghost MsPac-Man eats is worth 200 points and subsequent ghosts are worth double. For example if MsPac-Man ate all 4 ghosts during 1 power-up she would gain 200+400+800+1600 = 3000 points.
3. Finally every level at some point spawns a piece of fruit which moves around the maze. Each piece of fruit is worth a variable amount of points:
    - Cherry: 100 points.
    - Strawberry: 200 points
    - Orange: 500 points
    - Pretzel: 700 points
    - Apple: 1000 points
    - Pear: 2000 points
    - Banana: 5000 points

A.4 Movement options

MsPac-man has only 4 movement options up, down, left, and right. If she collides with a wall she immediately stops moving until another input is given. There are also tunnels that exist within most of the maps which allow for MsPac-Man to travel to the other side. While ghosts are also able to use this feature, during the first 3 rounds of the game doing so temporarily reduces their movement speed. Finally when a ghost is eaten while in a fearful state as defined above they are transported into the central area of the maze where they are held for a few seconds before being released back into the maze.

B Hyper-Parameter Selection

B.1 A3C

Hyper-parameters are largely taken from those used by Mnih et al. (2016) and Greydanus (2017). Alterations include: death penalty (0 death penalty in the papers -> -500 for our environment) and Tau (Greydanus (2017) uses 1.0, we use 0.999). Explanations of hyper-parameters are available in Mnih et al. (2016). Hyper-parameters were chosen by a review of the literature and consideration of the environment.

Hyper-parameter	Value
Grey-scaling	True
Observation down-sampling	(80, 80)
Reward clipping	None
Processes	28
Optimizer learning rate	0.0001
NN hidden units	256
Gamma	0.99
Tau	0.999
Horizon	0.99
T_{max}	20
Terminal on loss of life	False
Death Penalty	-500
Max frames per episode	10k

Table 1: **A3C Hyper-parameters**

B.2 Rainbow

We used many of the same processing steps and hyper-parameters from the Rainbow paper (Hessel et al., 2018), but made alterations to better suit our chosen environment. The environment preprocessing consisted of:

Hyper-parameter	Value
Grey-scaling	True
Observation down-sampling	(84, 84)
Frames stacked	4
Action repetitions	4
Reward clipping	None
Terminal on loss of life	False
Death Penalty	-500
Max frames per episode 108K	100k

Table 2: **Pre-Processing**: Observations are grey-scaled and resized to 84×84 pixels. Four consecutive frames are stacked to represent each state. Every action selected is repeated four times by the agent. No reward clipping occurs to ensure impact of the death penalty specific to this environment.

Hyper-parameter	Value
Q network: channels	32, 64, 64
Q network: filter size	(8 × 8), (4 × 4), (3 × 3)
Q network: stride	4, 2, 1
Q network: hidden units	512
Q network: output units	9
Discount factor	0.99
Memory size (PER)	1,000,000
Replay period	every 4 agent steps
Minibatch size	32
Optimizer	Adam
Learning rate	0.0001
Noisy Layers std	0.5
n-step	3
Update Frequency	500
Discount Factor	0.99
Atoms	51

Table 3: **Hyper-parameters**: The base DQN has 3 convolutional layers with 32, 64 and 64 channels. Each layer uses $(8 \times 8), (4 \times 4), (3 \times 3)$ filters with strides $4, 2$and1 respectively. The dueling architecture with value and advantage branches each have a hidden layer size of 512 units. The output of the network is the available number of actions of the game, in this case 9. The Adam optimiser has a learning rate of 0.0001. The number of atoms for the categorical DQN is 51. When noisy layers are used, the epsilon greedy exploration parameter ϵ is set to 0.

C Experimental Details

C.1 A3C

To replicate the training of our A3C agent:

- Copy "`a3c_agent_group9`" folder
- `pip install -r requirements.txt` in "`a3c_agent_group9`" folder (ideally to a fresh virtual env)
- `python main_a3c.py` within "`a3c_agent_group9`" folder
- If rendering is desired, change the global "`RENDER`" variable from "`False`" to "`True`"
- Note: the "`PROCESSES`" global variable has been set to `8` to limit computation load on the computer, but was `28` in our trained version (from which the results were obtained)

C.2 Rainbow

To replicate the training of our Rainbow agent:

- Copy "`rainbow_agent_group9`" folder
- `pip install -r requirements.txt` in "`rainbow_agent_group9`" folder (ideally to a fresh virtual env)
- `python main_rainbow.py` within "`rainbow_agent_group9`" folder
- In order to enable GPU usage an additional command "`pip3 install torch==1.10.0+cu113 -f https://download.pytorch.org/whl/cu113/torch_stable.html`" must be run
- If rendering is desired, change `render` to be `True` inside of `atari_wrapper.py` and `death` to `False` in order to get the real scores
- In the event of a paging file error you may need to reduce the number of environments being created, these can be found on line 408, 409, 499 and 500

- In the event of an unable to allocate memory error please reduce the memory size found on line 458

D Algorithm Pseudocode

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 4: The pseudo-code procedure for A3C (Mnih et al., 2016).

Algorithm 1 Deep Q-learning with Experience Replay

1million frames
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
Second "target network" added in a later version of the paper
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
Each x_t consists of 4 frames
ϕ : RGB to grayscale, downsampling, cropping,
  for $t = 1, T$ **do**
Annealed from 1 to 0.1
"Frame-skipping": agent sees and select actions on every k^{th} frame
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
ϵ-greedy strategy
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
FIFO queue
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
batch size=32
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 5: The pseudo-code procedure for a DQN with Prioritized Experience Replay (Mnih et al., 2013).

E Rainbow Components

Double Q-Learning

Double DQN (van Hasselt et al., 2015) fixes the overestimation bias of Q-learning by separating the selection and evaluation of bootstrapped actions. Combining this method with the base DQN gives us the loss

$$L_i(\theta_i) = \left[R_{t+1} + \gamma_t Q_\theta^- \left(S_{t+1}, \arg\max_{a'} Q_\theta(S_{t+1}, a') \right) - Q_\theta(S_t, A_t) \right]^2. \tag{5}$$

Dueling

Dueling (Wang et al., 2015) is a Network architecture which enables actions to be generalised by representing state values and action advantages separately. These value and advantage streams share a convolution encoder, which merges by the following aggregator

$$Q_\theta(s,a) = v_\eta\left(f_\varepsilon(s)\right) + a_\psi(f_\varepsilon(s),a) - \frac{\sum_{a'} a_\psi(f_\varepsilon(s),a')}{N_{actions}}. \tag{6}$$

Our implementation is contained within the `Rainbow` class which inherits directly from the base `DQN` object (Listing

.3).

Distributional DQN

Bellemare et al. (2017) states that a Distributional DQN learns an approximate categorical distribution of discounted returns $Z(s,a)$, instead of approximating the expected return $Q(s,a)$. $Z(s,a)$ satisfies a distributional Bellman equation:

$$Z(s,a) = R(s,a) + \gamma Z(s',a'). \tag{7}$$

This distribution of returns is discrete and parameterized by the number of atoms $N \in \mathbb{N}$ and $V_{min}, V_{man} \in \mathbb{R}$. The support for the distribution of atoms is

$$\left\{ z_i \overset{\text{def}}{=} V_{min} + i\Delta z : 0 \leq i < N \right\} \text{ for } \Delta z \overset{\text{def}}{=} \frac{V_{max} - V_{min}}{N-1}. \tag{8}$$

The atom probabilities are predicted using a softmax distribution

$$Z_\theta(s,a) = \left\{ z_i \text{ with probability } p_i = \frac{e^{f_i(s,a)}}{\sum_j e^{f_j(s,a)}} \right\}. \tag{9}$$

The network is then trained to minimise the Kullbeck-Leibler divergence between the current distribution and the one-step update

$$D_{KL}\left(\Phi(R + \max_{a'} Z(s',a') || Z(s,a))\right). \tag{10}$$

Noisy DQN

A Noisy DQN (Fortunato et al., 2017) adds stochastic network layers for exploration. This adds a noisy stream to the deterministic weights in the form of

$$\boldsymbol{y} = (\boldsymbol{b} + \boldsymbol{W}\boldsymbol{x}) + (\boldsymbol{b}_{noisy} \odot \epsilon^b + (\boldsymbol{W}_{noisy} \odot \epsilon^\omega)\boldsymbol{x}) \tag{11}$$

where ϵ^ω and ϵ^b are random variables. The noise ϵ is a factorised Gaussian in matrix form. If $\epsilon_{i,j}$ corresponds to a layer with i, j inputs and outputs, we can generate independent noise ϵ_i, ϵ_j for input and output neurons where

$$\epsilon_{i,j} = f(\epsilon_i) f(\epsilon_j) \tag{12}$$

for $f(x) = \text{sign}(x)\sqrt{|x|}$. When noisy layers are used, ϵ-greedy is no longer used, and instead are entirely greedy.

Our implementation is contained with the `NoisyLinear` class (Listing .2), adapted from here.

Prioritized Experience Replay

Prioritized experience replay (Schaul et al., 2015) allows an agent to replay transitions from which there is more to learn. Samples are drawn with probability p_t relative to the last encountered absolute temporal difference error.

$$p_t \propto \left| R_{t+1} + \gamma_{t+1} \max_{a'} Q_\theta^-(S_{t+1},a') - Q_\theta(S_t,A_t) \right|^\omega \tag{13}$$

We chose to use the existing data structures in `Tianshou` (Weng et al., 2021) as they allow for efficient computation using the `PrioritizedVectorReplayBuffer` class.

N-step Temporal Difference

Used in A3C (Mnih et al., 2016), learning from multi-step bootstrap targets (Sutton and Barto, 2014) shifts the bias-variance trade off which enables newly observed rewards to propagate to previously visited states. The truncated n-step return from a state S_t is

$$R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}. \tag{14}$$

The multi step variant loss then becomes

$$L_i(\theta_i) = \left[R_t^{(n)} + \gamma_t^{(n)} \max_{a'} Q_{\theta}^{-}(S_{t+n}, a') - Q_{\theta}(S_t, A_t) \right]^2. \tag{15}$$

F Code

```python
# Returns reward - 100 when hitting a ghost
class PenaliseDeath(gym.RewardWrapper):
    """Penalise suiciding into ghosts!"""
    def __init__(self, env):
        super().__init__(env)
        self.total_lives = 3

    def reward(self, reward):
        self.total_lives_left = self.env.unwrapped.ale.lives()
        if self.total_lives_left < self.total_lives:
            self.total_lives -= 1
            return reward-500
        return reward

    def reset(self):
        if self.was_real_done:
            obs = self.env.reset()
        else:
            obs = self.env.step(0)[0]
        self.total_lives = self.env.unwrapped.ale.lives()
        return obs
```

Listing .1: Returns reward - 500 when hitting a ghost.

```python
class NoisyLinear(nn.Module):
    """Implementation of Noisy Networks. arXiv:1706.10295.
        Adapted from https://github.com/ku2482/fqf-iqn-qrdqn.pytorch/blob/master
        /fqf_iqn_qrdqn/network.py .
    """

    def __init__(self, in_features, out_features, noisy_std = 0.5):
        super().__init__()

        # Learnable parameters.
        self.mu_W = nn.Parameter(torch.FloatTensor(out_features, in_features))
        self.sigma_W = nn.Parameter(torch.FloatTensor(out_features, in_features))
        self.mu_bias = nn.Parameter(torch.FloatTensor(out_features))
        self.sigma_bias = nn.Parameter(torch.FloatTensor(out_features))

        # Factorized noise parameters.
        self.register_buffer('eps_p', torch.FloatTensor(in_features))
        self.register_buffer('eps_q', torch.FloatTensor(out_features))

        self.in_features = in_features
        self.out_features = out_features
        self.sigma = noisy_std

        self.reset()
        self.sample()

    def reset(self):
        bound = 1 / np.sqrt(self.in_features)
        self.mu_W.data.uniform_(-bound, bound)
        self.mu_bias.data.uniform_(-bound, bound)
        self.sigma_W.data.fill_(self.sigma / np.sqrt(self.in_features))
        self.sigma_bias.data.fill_(self.sigma / np.sqrt(self.in_features))

    def f(self, x):
        x = torch.randn(x.size(0), device=x.device)
        return x.sign().mul_(x.abs().sqrt_())

    def sample(self):
        self.eps_p.copy_(self.f(self.eps_p))
        self.eps_q.copy_(self.f(self.eps_q))

    def forward(self, x):
        if self.training:
            weight = self.mu_W + self.sigma_W * (
                self.eps_q.ger(self.eps_p)
            )
            bias = self.mu_bias + self.sigma_bias * self.eps_q.clone()
        else:
            weight = self.mu_W
            bias = self.mu_bias

        return F.linear(x, weight, bias)
```

Listing .2: The NoisyLinear Layer.

```python
class DQN(nn.Module):
    """Reference: Human-level control through deep reinforcement learning."""

    def __init__(
        self,
        c, # Channels - In this case, stacked frames.
        h, # Height
        w, # Width
        action_shape, # Action Space
        device="gpu",

        output_dim=None,
    ):
        super().__init__()
        self.device = device
        self.DQN = nn.Sequential(
            nn.Conv2d(c, 32, kernel_size=16, stride=4), nn.ReLU(inplace=True),
            nn.BatchNorm2d(32),
            nn.Conv2d(32, 64, kernel_size=12, stride=2), nn.ReLU(inplace=True),
            nn.BatchNorm2d(64),
            nn.Conv2d(64, 64, kernel_size=4, stride=1),
            nn.ReLU(inplace=True),
            nn.Flatten()
        )
        with torch.no_grad():
            self.output_dim = np.prod(self.DQN(torch.zeros(1, c, h, w)).shape[1:])

        if output_dim is not None:
            self.DQN = nn.Sequential(
                self.DQN, nn.Linear(self.output_dim, output_dim),
                nn.ReLU(inplace=True)
            )
            self.output_dim = output_dim

    def forward(self, obs, state=None, info={}):
        r"""Mapping: s -> Q(s, \*)."""
        obs = torch.as_tensor(obs, device=self.device, dtype=torch.float32)
        return self.DQN(obs), state

class Rainbow(DQN):
    """Reference: Rainbow: Combining Improvements in Deep Reinforcement Learning."""

    def __init__(
        self,
        c, # Channels - In this case, stacked frames.
        h, # Height
        w, # Width
        action_shape, # Action Space
        num_atoms = 51,
        noisy_std = 0.1,
        device = "cpu",
    ):
        super().__init__(c, h, w, action_shape, device)
        self.action_num = np.prod(action_shape)
        self.num_atoms = num_atoms

        self.Double_Q = nn.Sequential(
            NoisyLinear(self.output_dim, 512, noisy_std), nn.ReLU(inplace=True),
            NoisyLinear(512, self.action_num * self.num_atoms, noisy_std)
        )

        self.Dueling = nn.Sequential(
            NoisyLinear(self.output_dim, 512, noisy_std), nn.ReLU(inplace=True),
            NoisyLinear(512, self.num_atoms, noisy_std)
        )
        self.output_dim = self.action_num * self.num_atoms

    def forward(self, obs, state = None, info = {}):
        r"""Mapping: x -> Z(x, \*)."""
        obs, state = super().forward(obs)
        # Split the Q table into dueling
        q = self.Double_Q(obs)
        q = q.view(-1, self.action_num, self.num_atoms)
        v = self.Dueling(obs)
        v = v.view(-1, 1, self.num_atoms)

        logits = q - q.mean(dim=1, keepdim=True) + v
        probs = logits.softmax(dim=2)
        return probs, state
```

Listing .3: DQN and Rainbow Classes.

```python
class QContinuumRainbow(DQN):
    """Reference: Rainbow: Combining Improvements in Deep Reinforcement Learning."""

    def __init__(
        self,
        c, # Channels - In this case, stacked frames.
        h, # Height
        w, # Width
        action_shape, # Action Space
        num_atoms = 51,
        noisy_std = 0.1,
        device = "cpu",
    ):
        super().__init__(c, h, w, action_shape, device)
        self.action_num = np.prod(action_shape)
        self.num_atoms = num_atoms

        self.Double_Q = nn.Sequential(
            NoisyLinear(self.output_dim, 512, noisy_std), nn.ReLU(inplace=True),
            NoisyLinear(512, self.action_num * self.num_atoms, noisy_std)
        )
        self.Triple_Q = nn.Sequential(
            NoisyLinear(self.output_dim, 512, noisy_std), nn.ReLU(inplace=True),
            nn.Dropout(0.1),
            NoisyLinear(512, self.action_num * self.num_atoms, noisy_std)
        )

        self.Dueling = nn.Sequential(
            NoisyLinear(self.output_dim, 512, noisy_std), nn.ReLU(inplace=True),
            NoisyLinear(512, self.num_atoms, noisy_std)
        )
        self.Double_Dueling = nn.Sequential(
            NoisyLinear(self.output_dim, 512, noisy_std), nn.ReLU(inplace=True),
            nn.Dropout(0.1),
            NoisyLinear(512, self.num_atoms, noisy_std)
        )
        self.output_dim = self.action_num * self.num_atoms

    def forward(self, obs, state = None, info = {}):
        r"""Mapping: x -> Z(x, \*)."""
        obs, state = super().forward(obs)
        # Split the Q table into dueling
        q = self.Double_Q(obs)
        q = q.view(-1, self.action_num, self.num_atoms)
        q2 = self.Triple_Q(obs)
        q2  = q2.view(-1, self.action_num, self.num_atoms)
        v = self.Dueling(obs)
        v = v.view(-1, 1, self.num_atoms)
        v2 = self.Double_Dueling(obs)
        v2 = v2.view(-1, 1, self.num_atoms)
        logits = q - q.mean(dim=1, keepdim=True) + v +\
            q2 - q2.mean(dim=1, keepdim=True) + v2
        probs = logits.softmax(dim=2)
        return probs, state
```

Listing .4: Triple DQN, Double Dueling and Rainbow.

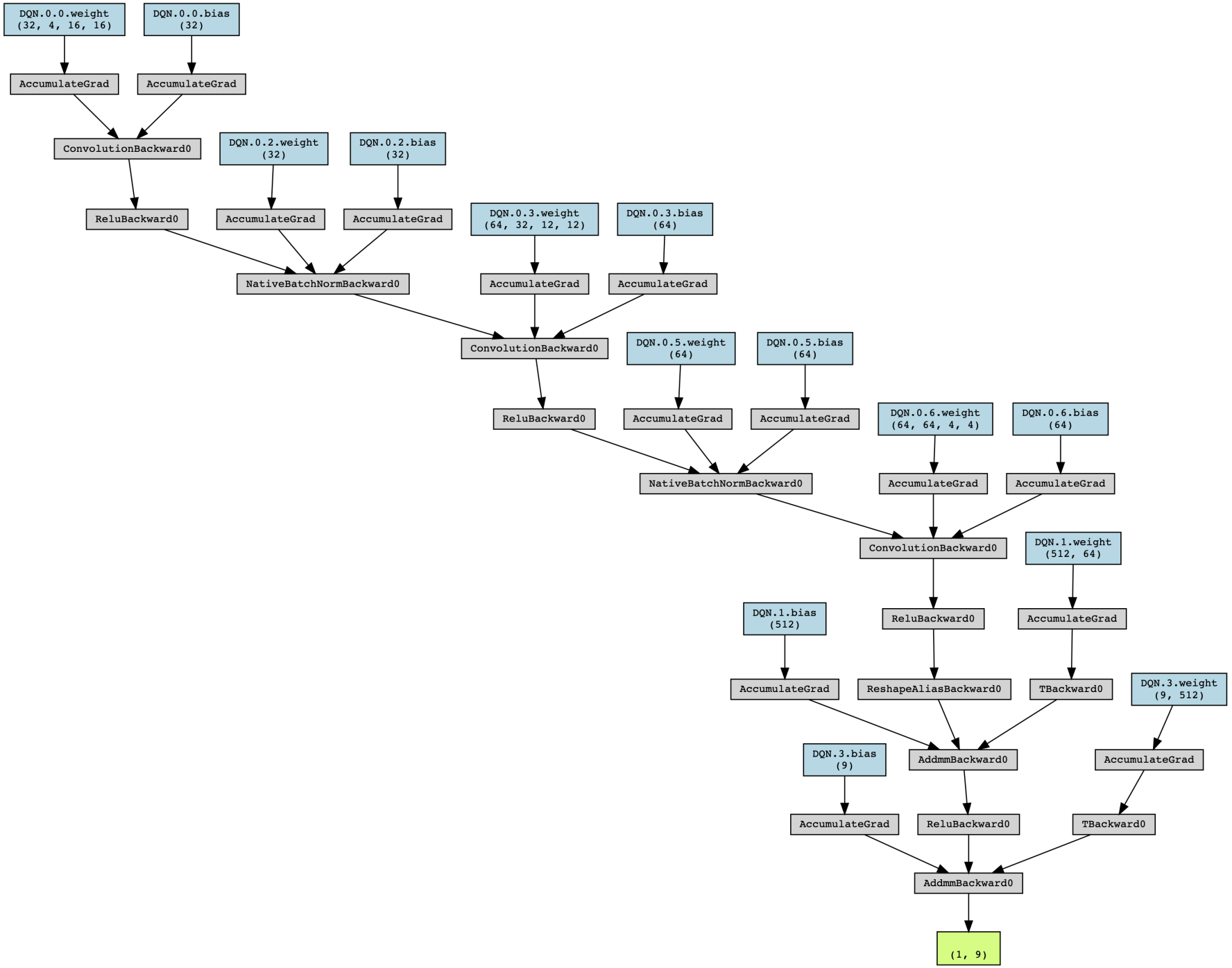

Figure 6: Deep-Q Neural Network.

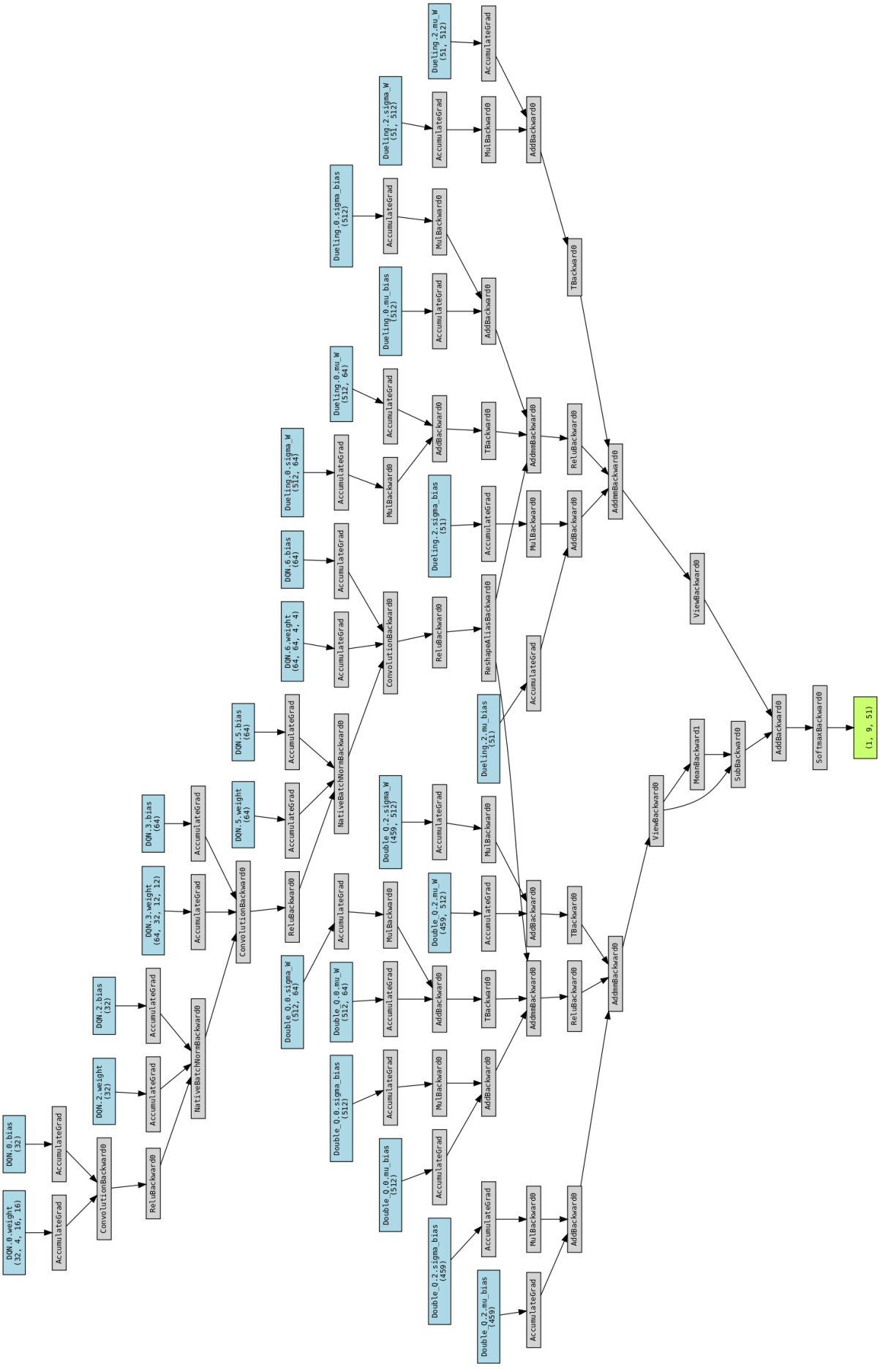

Figure 7: Rainbow Deep-Q Neural Network.

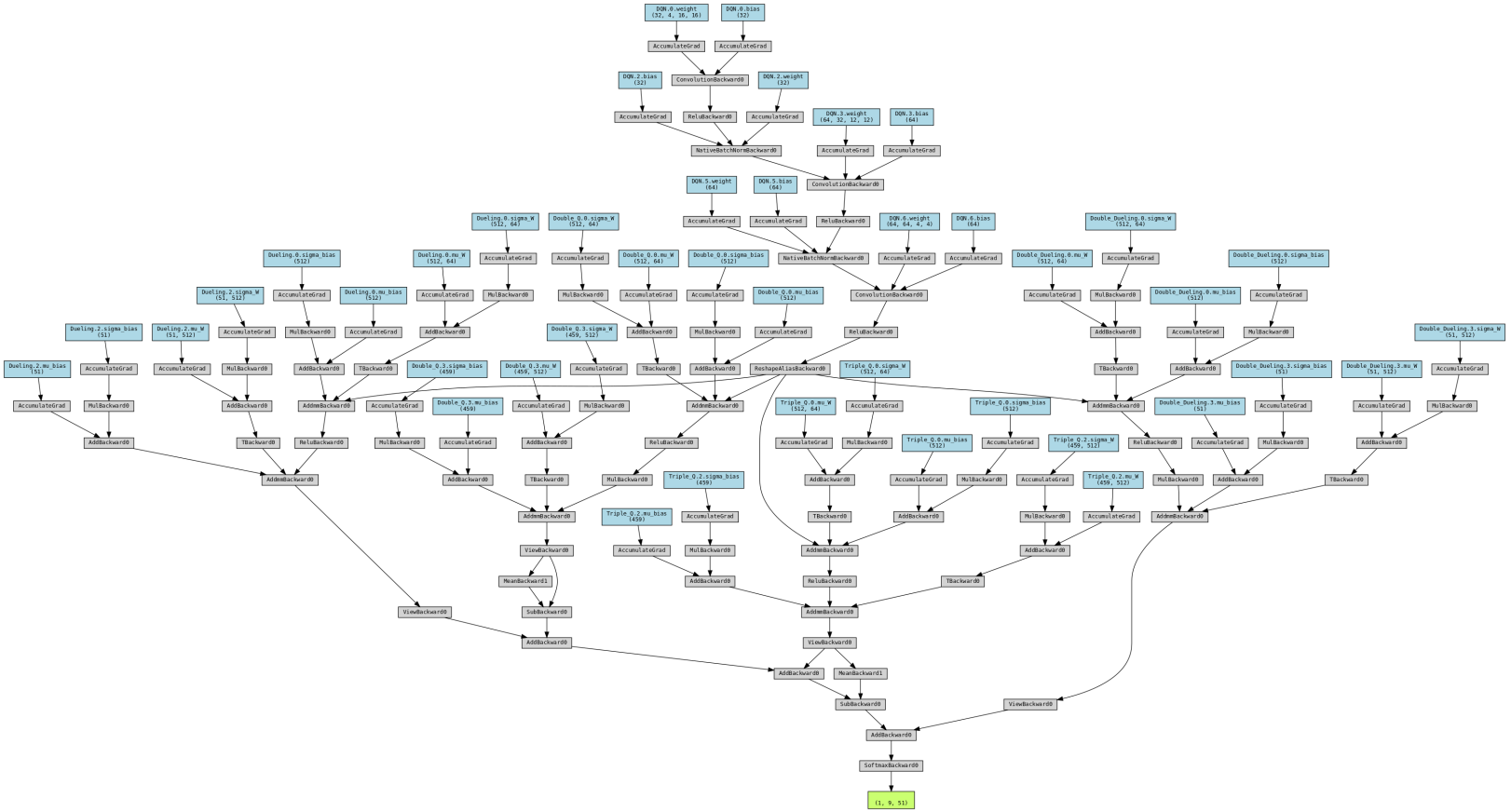

Figure 8: Double Dueling, Triple-Q Rainbow Neural Network.

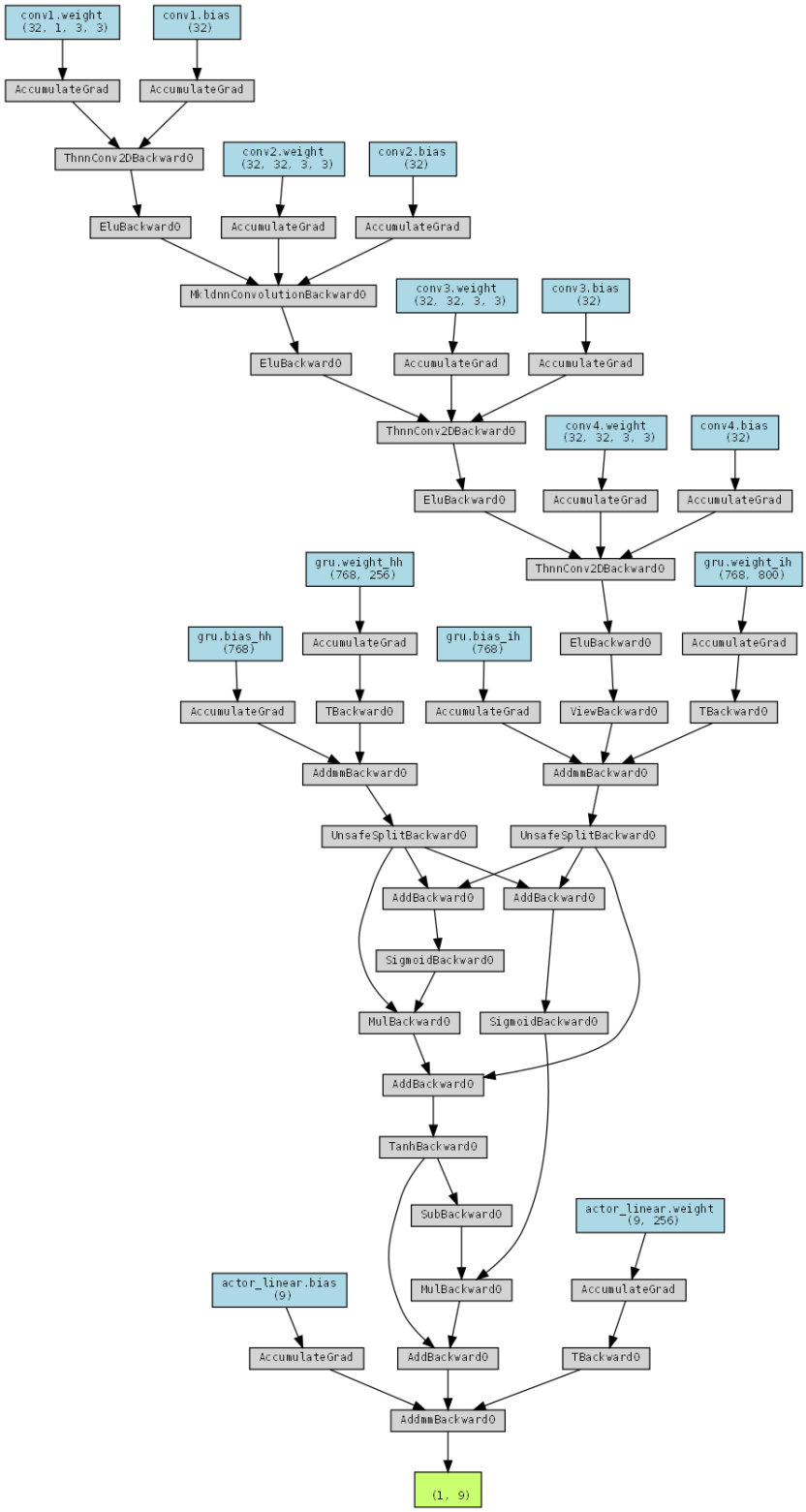

Figure 9: Actor Neural Network for A3C

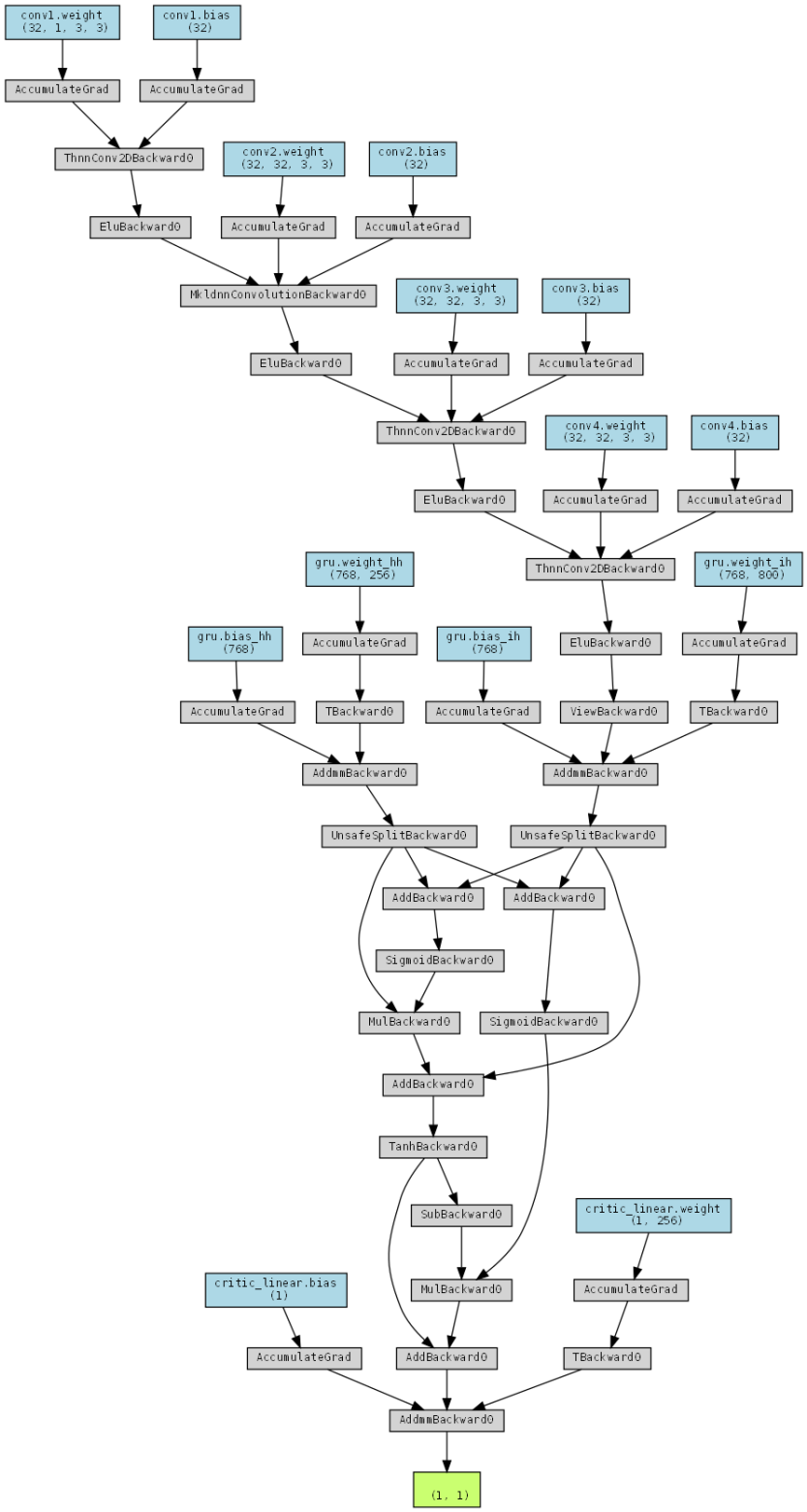

Figure 10: Critic Neural Network for A3C