Technical Documentation: Advancing RL in "Super Mario Bros." through Reward Engineering and Enhanced Double Deep Q-Network Architectures

Anthony Khaiat akhaiat@mit.edu
Jan Philipp Girgott girgott@mit.edu
Valentin Pinon vpinon@mit.edu

1 Technical Description of Baseline Method

Our baseline method uses a Double Deep Q-Network (DDQN) architecture with a convolutional neural network (CNN). The model uses the `SIMPLE_MOVEMENT` action space, which consists of seven discrete actions, each represented as a list of inputs Mario can execute at time t (see slides).

Since our actions are not dependent on the RGB input values of the environment, we use wrappers to pre-process the environment data before sending it to the agent. We use a `GrayScaleObservation` wrapper which transforms the RGB image to greyscale. We also use a `ResizeObservation` wrapper to downsample observations into square images. Next, the `SkipFrame` wrapper is inherited from `gym.Wrapper` and skips intermediate frames without information loss. The last step of pre-processing the image data is the `FrameStack` wrapper which merges consecutive frames into one observation point, and this allows us to determine whether Mario jumps or lands based on previous frames. After applying all of our wrappers to our environment, we obtain a final wrapped state consisting of 4 stacked gray-scaled consecutive frames.

Next, we implement a Mario class (our agent) which can **act** according to the optimal action policy based on the state, **remember** experiences, and **learn**. For any state, Mario can choose to do random exploration or the most optimal action, and this parameter is controlled by `self.exploration_rate`.

For Mario's memory, the agent can `cache()`, where Mario stores an experience in the format of *state, action, reward, next state, and done*, or the agent has the function `recall()`, allowing them to randomly sample a memory to learn the game.

The agent uses the DDQN algorithm to learn. DDQN uses two CNNs, Q_{online} and Q_{target} to approximate the action-value function.

There are two values for learning: 1) **TD Estimate** and 2) **TD Target**. **TD Estimate** is the predicted optimal Q^* for a state:

$$TD_e = Q^*_{online}(s, a)$$

TD Target is the aggregate of current reward and the estimated Q^* in the next state s':

$$a' = \arg\max_a Q_{online}(s', a) \qquad TD_t = r + \gamma Q^*_{target}(s', a')$$

Since we are unaware of the next action a', we take the argmax of Q_{online} at the next state.

As Mario samples inputs from the replay buffer, we compute the values for TD_t and TD_e and back-propagate this loss down Q_{online} to update its parameters θ_{online}. We also use α, which is the learning rate of the ADAM optimizer with SmoothL1Loss. θ_{target} does not update through backpropagation. Instead, we periodically copy θ_{online} to θ_{target}:

$$\theta_{target} \leftarrow \theta_{online} \leftarrow \theta_{online} + \alpha\nabla(TD_e - TD_t) \text{ and } \theta_{target} \leftarrow \theta_{online}$$

Finally, we run the training loop for 40 episodes and record the mean score for all episodes and the computational time.

1.1 Baseline Reward Function (R_t^{base})

The reward function in the `gym-super-mario-bros` environment is the sum of three main components designed to encourage efficient and safe level navigation:

1. **Horizontal Progress** $H(x_t, x_{t-1})$: The agent receives rewards for moving right which equals the difference between the current x-position x_t and the previous x-position x_{t-1}.
2. **Time Penalty** $P_T(t_{now}, t_{prev})$: A linear penalty is applied for the passage of time, motivating the agent to complete levels more quickly.
3. **Death Penalty** $P_D()$: Dying incurs a significant negative reward of -25.

This provides us with the full formulation of the baseline reward function.

$$R_t^{base}(\cdot) = \underbrace{(x_t - x_{t-1})}_{H(\cdot)} + \underbrace{(t_{prev} - t_{now})}_{P_T(\cdot)} + \underbrace{\mathbf{1}_t^{\text{is dead}} * (-25)}_{P_D(\cdot)} \quad \forall t \in [1, T]$$

1.2 Baseline Neural Network Architecture

The baseline CNN architecture consists of 3 convolutional layers with a stride of 4 pixels and kernel size of 8 pixels. Moreover, we use RELU activation functions.

2 Technical Description of Changes

2.1 Changes to Reward Functions

2.1.1 Reward Function 1 (R_t^1): Use Information From Dictionary

The pieces of information from the dictionary that can be used for reward design are `coins`, `score`, and `status`. For coins and score respectively, we will add a small incremental reward for each additional coin collected ($\Delta_t^c = \texttt{coins}_t - \texttt{coins}_{t-1}$) and score earned ($\Delta_t^s = \texttt{score}_t - \texttt{score}_{t-1}$). These small incremental rewards are called $\mu_c = 0.5$ and $\mu_s = 0.1$. For status, the agent will receive an additional reward of 0.5 if `status = tall` and an additional reward of 1 if `status = fireball`. These considerations lead to the following reward function.

$$R_t^1(\cdot) = R_t^{\text{base}} + 0.5\Delta_t^c + 0.1\Delta_t^s + \begin{cases} 0.5, & \text{if } \texttt{status} = \text{tall} \\ 1, & \text{if } \texttt{status} = \text{fireball} \\ 0, & \text{otherwise} \end{cases} \quad \forall t \in [1, T]$$

2.1.2 Reward Function 2 (R_t^2): Decrease Importance of Time

The basic reward function gives equal weight to progress along the x-axis and speed. Since the primary goal is to complete the game without dying, one might decide to change the current linear time penalty. We propose a quadratic option, so that the time penalty starts small and increases as the time limit of the game (T = 1,000) is approached. To also decrease the overall magnitude of the time penalty, we add a scaling factor k = 0.5.

$$R_t^2(\cdot) = \underbrace{(x_t - x_{t-1})}_{H(\cdot)} + \underbrace{0.5\left(\frac{t}{1000}\right)^2}_{P_T'(\cdot)} + \underbrace{\mathbf{1}_{\text{is dead}} * (-25)}_{P_D(\cdot)} \quad \forall t \in [1, T]$$

2.1.3 Reward Function 3 (R_t^3): Make Later Progress More Rewarding

The base reward function assumes that rewards for stepping to the right are equally rewarding in all stages of the game. However, there might be value in making the later steps of the game more rewarding as they lead to the completion of the task. Therefore, we make the incremental reward quadratic. To control for its magnitude, we add a scaling factor k = 0.5.

$$R_t^3(\cdot) = \underbrace{(x_t - x_{t-1}) * 0.5x_t}_{H'(\cdot)} + \underbrace{(t_{prev} - t_{now})}_{P_T(\cdot)} + \underbrace{\mathbf{1}_t^{\text{is dead}} * (-25)}_{P_D(\cdot)} \quad \forall t \in [1, T]$$

2.1.4 Combined Reward Function ($R_t^{combined}$): Make Later Progress More Rewarding

In addition to the individual reward functions described before, we also introduce a version that combines all changes made before.

$$R_t^{combined}(\cdot) = \underbrace{(x_t - x_{t-1}) \cdot 0.5 \cdot x_t}_{H'(\cdot)} + \underbrace{0.5\left(\frac{t}{1000}\right)^2}_{P'_T(\cdot)} + \underbrace{\mathbf{1}_t^{\text{is dead}} \cdot (-25)}_{P_D(\cdot)} + 0.5\Delta_t^c + 0.1\Delta_t^s$$

$$+ \begin{cases} 0.5, & \text{if } \texttt{status} = \text{tall} \\ 1, & \text{if } \texttt{status} = \text{fireball} \\ 0, & \text{otherwise} \end{cases} \quad \forall t \in [1, T]$$

2.2 Changes to Neural Network Architectures

To advance the current Double Deep Q-Network architecture applied in the Mario domain, we explored the integration of the following neural network models.

1. **ResNet50:** Residual networks like ResNet50 are known for their deep architecture with shortcut connections, which can help capture intricate features and spatial relationships in complex game environments, potentially leading to more robust and efficient learning. The ResNet50 model from the `torchvision.models` library was pre-trained on ImageNet. We modify the first convolutional layer of the mdoel to accomodate for the different number of input channels; the reason behind this is to deal with images of different sizes than those expected by ResNet50.
2. **Vision Transformer (ViT):** Vision Transformers have shown promise in capturing long-range dependencies and spatial relationships in image data, which can be advantageous in understanding the layout of levels, detecting obstacles, and planning paths. In our case, we use the `vit_small_patch16_224` model from Hugging Face, which has also been pre-trained on ImageNet.
3. **AlexNet:** Although an older architecture compared to ResNet50 and ViT, AlexNet's simpler structure and fewer parameters might offer computational advantages. We use the AlexNet model from the `torchvision.models` library and we modify the first convolutional layer to adjust for our modified input image.
4. **PPO:** Leveraging a straightforward convolutional neural network (CNN) architecture, PPO offers a streamlined approach for efficient training and learning in diverse environments.

3 Results

To address the computational demands of training image-based RL agents, we use the Google Colab Pro's T4 GPUs. We are using a streamlined action space and run each algorithm on 40 episodes.

3.1 Mean In-Game Score

Algorithm	$R_t^{base}(\cdot)$	$R_t^1(\cdot)$	$R_t^2(\cdot)$	$R_t^3(\cdot)$	$R_t^{combined}(\cdot)$	**Mean**
$DDQN : VanillaCNN$	215.0	275.0	332.5	165.0	250.0	247.5
$DDQN : ResNet50$	215.0	235.0	202.5	200.0	227.5	216.0
$DDQN : ViT$	230.0	288.75	215.0	197.5	242.5	234.75
$DDQN : AlexNet$	237.5	285.0	317.5	252.5	337.5	286.0
$PPO : VanillaCNN$	215.0	225.0	285.0	190.0	265.0	236.0
$PPO : ResNet50$	185.0	177.0	222.5	202.5	225.0	202.4
$PPO : ViT$	195.0	202.0	230.0	200.0	242.5	213.9
$PPO : AlexNet$	235.0	218.0	250.0	203.0	240.0	229.2
Mean	215.44	238.09	256.88	201.25	253.75	–

3.2 Training Time

Algorithm	$R_t^{base}(\cdot)$	$R_t^1(\cdot)$	$R_t^2(\cdot)$	$R_t^3(\cdot)$	$R_t^{combined}(\cdot)$	**Mean**
$DDQN : VanillaCNN$	179.065	227.849	254.487	139.224	201.630	200.451
$DDQN : ResNet50$	321.271	385.643	203.379	289.225	432.549	326.413
$DDQN : ViT$	257.522	391.074	278.460	291.624	281.981	300.132
$DDQN : AlexNet$	226.688	300.971	369.925	277.470	380.941	311.199
$PPO : VanillaCNN$	187.345	289.370	257.210	235.950	179.250	229.825
$PPO : ResNet50$	334.670	338.520	216.260	257.140	447.120	318.742
$PPO : ViT$	233.360	372.070	248.140	265.820	421.950	308.268
$PPO : AlexNet$	329.690	291.020	304.550	248.830	375.410	309.900
Mean	258.701	324.564	266.551	250.660	327.603	–

3.3 Score-to-Time Ratio

Algorithm	$R_t^{base}(\cdot)$	$R_t^1(\cdot)$	$R_t^2(\cdot)$	$R_t^3(\cdot)$	$R_t^{combined}(\cdot)$	**Mean**
$DDQN : VanillaCNN$	1.20	1.21	1.31	1.19	1.24	1.23
$DDQN : ResNet50$	0.67	0.61	0.99	0.69	0.53	0.70
$DDQN : ViT$	0.89	0.74	0.77	0.68	0.86	0.79
$DDQN : AlexNet$	1.05	0.95	0.86	0.91	0.89	0.93
$PPO : VanillaCNN$	1.15	0.78	1.11	0.81	1.48	1.07
$PPO : ResNet50$	0.55	0.52	1.03	0.79	0.50	0.68
$PPO : ViT$	0.84	0.54	0.93	0.75	0.57	0.73
$PPO : AlexNet$	0.71	0.75	0.82	0.82	0.64	0.75
Mean	0.89	0.81	0.98	0.83	0.90	–

4 Discussion of Results

The evaluation of different network architectures and reward functions revealed significant differences in the ability of each configuration to navigate and succeed in the game environment. Notably, the AlexNet architecture combined with the $R_t^{combined}(\cdot)$ reward function yielded the highest score of 337.5, suggesting that certain architectures might better leverage complex reward strategies for improved decision-making in dynamic environments. Moreover, utilizing more complex pre-trained models such as ResNet50 or ViT did not yield improvements, which could suggest that the models' inductive bias may cause it to perform poorly on a new, game-like environment such as Mario.

The modification of reward functions had an impact on the behavior and success of the architectures. The introduction of $R_t^1(\cdot)$, which added bonuses for coins and player status, led to noticeable improvements in scores across most architectures, demonstrating the effectiveness of incorporating game-specific incentives into the reward design.

However, the $R_t^3(\cdot)$ reward function, which aimed to make later progress more rewarding, generally did not perform well, indicating that the increased rewards did not effectively motivate better performance toward the end of the game. This could be due to the increased complexity and risks associated with later stages of the game, which are not sufficiently offset by the increased rewards from reaching the goal state.

DDQN with the $R_t^2(\cdot)$ reward function, which decreased the importance of time, achieved a significantly higher score compared to the baseline and other reward functions. This suggests that reducing the pressure of time constraints may allow the agent to make more thoughtful navigation decisions, enhancing overall performance without the rush associated with harsh time penalties.

While AlexNet provided the highest scores, its computational demand, as reflected by the compute times and score-to-time ratios, was among the highest. This highlights a trade-off between computational efficiency and performance, with AlexNet configurations requiring more resources, which poses scalability challenges for longer training sessions or larger action spaces.

PPO: Vanilla CNN demonstrated the highest ratio in the combined reward setting, indicating that while its absolute performance was not the best, it offered the best balance between score achieved and computational resources used. Thus, PPO with CNN could be particularly advantageous in environments where both computational efficiency and agent performance are critical.

5 Team Contributions

Table 1: Team Contributions

Team Member	Contributions
Anthony Khaiat	Implementation of DDQN with baseline CNN, ResNet50, ViT, and AlexNet Writing part of mid-term and final report Presenting in video presentation
Jan Philipp Girgott	Reward Engineering Creation of presentation Video cutting Writing part of mid-term and final report Presenting in video presentation
Valentin Pinon	Implementation of PPO with baseline CNN, ResNet50, ViT, and AlexNet Ideation and initial research on Super Mario Bros environment Writing part of mid-term and final report Presenting in video presentation

6 References

- "Deep Reinforcement Learning with Double Q-learning", i.e., the baseline.
- "Proximal Policy Optimization Algorithms," by Schulman et al., provides an examination of PPO, which we intend to implement for improved learning stability and efficiency.
- "Playing Atari with Deep Reinforcement Learning," Mnih et al., introducing one of the first deep learning models to successfully learn policies directly from high-dimensional sensory input using RL.
- "A Survey on Transfer Learning for Multiagent Reinforcement Learning System" by Silva and Costa, explores training agents in various environments, emphasizing the importance of generalization.
- "Reward Shaping for Improved Learning in Real-time Strategy Game Play" by Kliem and Dasgupta, investigates the effect of reward shaping in improving the performance of reinforcement learning in the context of the real-time strategy, capture-the-flag game.
- Train a Mario-playing RL Agent by Yuansong Feng, Suraj Subramanian, Howard Wang, Steven Guo
- Playing Super Mario Bros. with Reinforcement Learning by Sohum Padhye
- OpenAI Mario Gym Environment
- Train a Mario-playing RL Agent PyTorch Tutorial, i.e. the baseline DDQN with CNN