



Universiteit
Leiden
The Netherlands

Bachelor Datascience and Artificial Intelligence

Optimizing Deep Reinforcement Learning
Architectures for Pacman: A Comparative Study
of Manual Design and Evolutionary Algorithms

Keith Iqbal

First supervisors: Marcello Bonsangue and Thomas Moerland

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

1/Jul/2025

Abstract

Deep reinforcement learning has achieved impressive results on Atari games by using a fixed convolutional neural network architecture. In a seminal study, agents learned to play directly from raw pixel data using the Deep Q-Network (DQN). This thesis adopts and builds upon some of the techniques introduced in that study. Our experiments focus on optimizing neural network architectures to train autonomous agents to play the classic arcade game Pacman. We use the Arcade Learning Environment for simulation. This study compared two approaches: manually designed deep reinforcement learning models and models optimized using a genetic algorithm.

The models were trained using Deep Q-learning. The input to the convolutional neural network consisted of channel tensor maps. These channels represent preprocessed semantic features extracted from key elements of the raw game frames. All other variables were kept constant. The optimization process focused on finding the number of layers and nodes that resulted in the highest game scores. Despite significant efforts to design neural networks manually, the genetic algorithm was able to find a network configuration that outperformed them.

Models generated by genetic algorithms consistently achieved higher average scores than manually designed ones. The findings demonstrate that evolutionary search can efficiently uncover compact and high-performing neural architectures for reinforcement learning tasks. Applying a genetic algorithm to multivariable scenarios could be prohibitively expensive. However, carefully selecting which hyperparameters to optimize and applying the genetic algorithm to a few hyperparameters at a time could be a practical strategy for solving complex tasks efficiently and reliably.

The results showed that, while manually designed models can perform well with appropriate tuning, architectures optimized using genetic algorithms achieved competitive performance with less manual intervention. This demonstrated the potential of genetic algorithms for automating neural network design in deep reinforcement learning.

Acknowledgements

I would like to thank everyone who motivated and supported me in making it this far. The conception of this thesis would not have been possible without the expertise, guidance, and support of my main supervisor, Marcello Bonsangue. I am also grateful to my professors—Max van Duijn, Elena Raponi, Roy de Kleijn, and Anne Urai—who provided an unforgettable learning experience. A special thanks to Bart Nikkelen for being a brilliant and inspirational teacher. Lastly, I would like to thank my brother Andis for his constant support and encouragement, and my friend Nico for his valuable companionship throughout this process.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Reinforcement Learning	1
1.3	Convolutional Neural Network	3
1.4	Related Work	4
2	Methods	5
2.1	Environment	6
2.2	Deep Q-Learning	7
2.3	Genetic Algorithm	8
3	Implementation	9
4	Results	13
4.1	Comparison	17
4.2	Statistical Analysis	17
5	Discussion	18
5.1	Conclusion	19
5.2	Further Research	20
	References	21
	Appendices	22
	Glossary	22
	Notations	23
	Notes	24
	Images	25
	Graphs	28
	Tables	31

1 Introduction

Video games are a significant part of life for many children and adults. In addition to providing entertainment, video games help children develop cognitive skills such as memory, attention, and problem-solving. The practice of automated game playing contributes to the development and understanding of intelligent algorithms. Game simulation is an ideal testing ground for existing and new algorithms because modeling in simulation is much more cost-effective. In this study, neural networks were trained to play Pacman. Models were trained using two separate strategies: manual design and a Genetic Algorithm (GA). The performance of the two strategies was then compared to determine which resulted in higher Pacman scores.

Pacman¹, originally called Puck Man, is a game introduced in the 1980s. It was first released in Japan then later in the United States [NAM]. The game was developed by Toru Iwatani with his team. A variant called Ms. Pac-Man, featuring a pink color scheme and a ribbon, was later created in the US. The objective of the video game is for the player to guide Pacman to eat all the pellets (dots) in the maze while avoiding ghosts. Four different colored ghosts chase Pacman, and if any of them catch him, Pacman loses a life. Pacman has a total of three lives in which he must finish eating all pellets. There are 154 pellets in total, including 4 special large pellets (power pellets). When Pacman eats one of these power pellets, the ghosts temporarily become vulnerable and can be eaten by Pacman.

The primary objective of this thesis is to explore how different neural network architectures impact the performance of Deep Reinforcement Learning (DRL) agents when playing the classic arcade game Pacman. While traditional DRL models use manually designed networks, this study investigates whether an automated search approach using a Genetic Algorithm (GA) can uncover more effective configurations.

This leads to the central research question: **Can Genetic Algorithms discover neural network architectures for DRL that outperform manually designed models in terms of gameplay performance in Pacman?**

To answer this, we compared the performance of 60 manually designed DRL models with 60 architectures discovered through a Genetic Algorithm, using the same training setup and evaluation criteria. By analyzing and statistically validating the outcomes, we aimed to assess the effectiveness and efficiency of using evolutionary optimization in the design of DRL architectures.

1.1 Motivation

Efficiency and automation are fascinating concepts. Combined, they have the potential to address major global challenges. Our aim was to explore a project that reflected our interest in these areas, while remaining feasible within our time frame. Pacman is a well-known game with simple rules and strategic complexity, making it an ideal environment for reinforcement learning experiments. This paper employs three main techniques. Together, these techniques form a process for creating an intelligent Pacman agent.

1.2 Reinforcement Learning

All animals learn. Instead of theorizing how they learn, we approach the problem from an artificial intelligence (AI) perspective. Reinforcement learning is based on the idea of learning through



Figure 1: Original Pacman screen

interaction with the environment [Sut18]. All activities have cause and effect; that is, every action has a consequence. Interactions with the environment produce a response, providing an opportunity to learn by determining whether the response was desirable.

There are three crucial elements for this to work: agent, environment, and reward. An agent is an entity that takes actions. These actions should be based on the outcomes of previous interactions, or, if no past experience exists, the agent should take random actions. The ability to make decisions in a given state is referred to as a policy, usually denoted by π . The decision to take random actions, known as exploration, is important for gaining experience. Exploitation, on the other hand, involves using past experience to make better decisions, which eventually leads to learning. The environment is the space in which the agent operates. It provides responses to the actions taken by the agent. The response received is assigned a value, known as the reward. The reward is positive for a desirable response and negative for an undesirable response. This measure dictates how useful the action taken was. The higher the reward, the more the system perceives the chosen action as a good choice in a given state. When the environment is in a particular configuration it is referred to as a state.

Reinforcement learning (RL) in machine learning is different from supervised and unsupervised learning. Unlike supervised or unsupervised learning, RL does not require labeled or unlabeled datasets; instead, it generates and evaluates its own data through interactions with the environment. This process is repeated to improve the policy. The generated data can be stored in single- or multi-dimensional arrays. However, due to the sheer volume of possible states and calculations, this approach quickly becomes infeasible. Deep reinforcement learning (DRL) is an advanced form of RL in which neural networks are used to approximate policies or value functions, enabling agents to predict responses to actions and generalize from past experiences. A multi-layer neural network can handle complex and large datasets, provided it has an appropriate number of layers and nodes². A deep neural network consists of an input layer, hidden layer(s), and an output layer. The input layer receives the data, which in this case is a 210 x 160 pixel image. The hidden layers receive calculated values from the preceding layers and forward them to the next layer. The final layer is the output layer, which represents the possible actions. In this case, there are nine possible actions to choose from. The information flow can be visualized as moving from one side to the other, typically from left to right (see Figure 3). For a neural network to receive input $X \in \mathbb{R}^n$ and produce a predicted output, commonly annotated as \hat{y} , several parameters need to be adjusted.

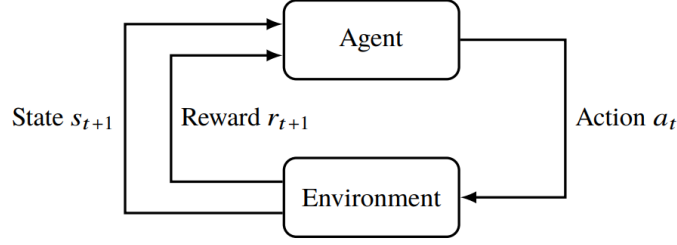


Figure 2: Reinforcement learning agent in an environment taking actions and receiving reward [Sut18]

These include the learning rate (η), activation functions, batch size, among others. Additionally, decisions must be made regarding the number of hidden layers and the number of nodes² (neurons or units) in each layer to achieve optimal performance.

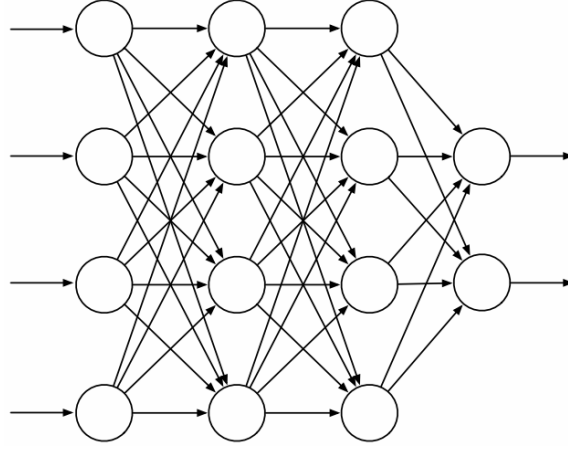


Figure 3: A typical feedforward artificial deep neural network depicts an input layer with 4 nodes, two hidden layers of 4 nodes each and the last layer as output layer with 2 nodes [Sut18]. The calculation here flows from left to right.

1.3 Convolutional Neural Network

Inspired by the visual cortex, Convolutional Neural Networks (CNNs) were designed to process images. In CNNs, kernels (also called filters) slide over the input to extract features such as edges or textures. Kernels are usually much smaller than the input dimensions—for example, 3 x 3 pixels. Other important elements of CNNs include stride and padding. Stride refers to how many pixels the kernel moves at each step during convolution. Padding involves adding extra pixels around the border of the input to preserve edge information and prevent excessive shrinking of the output. CNNs automate the feature extraction process, eliminating the need for hand-crafted features and improving pattern recognition performance[LBBH18]. They demonstrated their spatial capabilities, highlighting their ability to extract local features by restricting the receptive field. They may also use pooling layers to reduce the number of parameters and to analyze the structure of an image more efficiently. While the original paper focused on digit recognition, their spatial learning aspects

are applicable to any image. For Pacman, this could mean extracting the locations of Pacman, ghosts, and pellets within the environment. As the positions of Pacman and the ghosts change, the CNN can detect these changes, which aids the learning process. This technique allows raw images to be fed directly into the network, enabling feature extraction. The extracted features are then passed to fully connected (FC) layers, which produce Q-values representing the expected future rewards of possible actions. “

CNNs were successfully used in "Playing Atari with Deep Reinforcement Learning" [MKS⁺13, MKS⁺15]. In this work, the authors processed game screen data to train a model to play Atari games. They used video input (frames) directly, without hand-crafting visual features or providing internal state information about the game. The only preprocessing applied was reducing the RGB (Red Green Blue) image input from 210 x 160 pixels to 84 x 84 pixels to shrink the state space. They noted that it was impossible to understand the current game state from a single frame. To address the lack of temporal information in single frames, they stacked four consecutive frames to form each input state. This stacking helps the agent approximate Markovian observability, ensuring that the environment satisfies the Markov property required to apply Q-learning. Q-learning is a model-free algorithm that enables agents to learn the optimal actions. Markov Decision Process (MDP) is a mathematical framework used to model decision-making where outcomes are partly random and partly under the control of an agent. It consists of states, actions, transition probabilities, and rewards, and assumes that the next state depends only on the current state and action, Markov property is fundamental requirement to apply Q-learning.

After the convolution process, a non-linear activation function is applied to allow the network to learn complex patterns. The most commonly used activation function in CNNs is the Rectified Linear Unit (ReLU). Non-linearity is important; otherwise, the network would be linear regardless of the number of nodes and layers, as it could all be compressed into one or more constant multipliers. A linear network cannot solve linearly inseparable problems. Pooling layers, such as max pooling, reduce the spatial dimensions of the input while preserving important features. Lastly, inputs processed by the CNN are flattened into a one-dimensional vector and passed to FC layers. During training, the network uses backpropagation to update the network by comparing actual rewards with predicted rewards for possible actions. Backpropagation is a supervised learning algorithm used to update the weights of a neural network by propagating the error from the output layer backward through the network. It uses gradient descent to minimize the loss function by adjusting weights based on the error gradient. The difference between these two is called the loss, which is calculated using a loss function such as Mean Squared Error (MSE). Huber loss is often preferred over MSE, as it is less sensitive to outliers in the target values, which can occur in reinforcement learning. While their paper used MSE, Huber loss is a common and often beneficial modification.

1.4 Related Work

The 'Playing Atari with Deep Reinforcement Learning' paper published in 2013, and its follow-up 'Human-level control through deep reinforcement learning' by Mnih, are directly related to our experiment. In their first paper, their approach outperformed all previous methods on six Atari games and even surpassed human expert performance on three of them. As suggested in 'Gradient-Based Learning Applied to Document Recognition' [LBBH18]. They refrained from using hand-crafted features. They used raw pixel data (visual frames) as input to a CNN combined with reinforcement learning to achieve impressive results. They applied Q-learning with stochastic

gradient descent to update the network weights. To address the credit assignment problem, they implemented experience replay (ER), which stores previous experiences and random samples from them during training to break correlations and improve learning efficiency. The credit assignment problem in reinforcement learning refers to the challenge of determining which specific actions taken by an agent are responsible for observed outcomes or rewards. They preprocessed the frames by converting them into grayscale and resizing them to 84 x 84 pixels, stacking the four most recent frames as input to the network. Their network architecture consisted of 16 convolutional filters of size 8 x 8 in the first layer, followed by 32 convolutional filters of size 4 x 4 in the second layer. This was followed by a FC layer with 256 nodes, and finally an output layer with one neuron for each possible action. They started with an epsilon-greedy (ϵ -greedy) value of 1, which they reduced linearly to 0.1 over the first million frames. Training then continued for another nine million frames with ϵ set to 0.1. In practice, ϵ -greedy strategy selects the best-known action with probability of $1-\epsilon$ and a uniform normal random action with probability of ϵ . For ER, they stored the most recent 10 million frames. They also used frame skipping, where the agent repeated the same action for four consecutive frames before selecting a new action. This technique reduces computational load and helps stabilize gameplay.

A recent study from 2024, "Bridging Evolutionary Algorithms and Reinforcement Learning: A Comprehensive Survey on Hybrid Algorithms" has introduced new hybrid approaches that combine the strengths of two paradigms. The authors categorize hybrid algorithms into parallel, sequential, and embedded strategies, aiming to improve exploration, sample efficiency, and robustness in training agents [LHT⁺24]. Their findings support the idea that EAs can help address limitations of gradient-based RL, particularly in discovering neural network structures and navigating sparse or deceptive reward environments. This aligns directly with the motivation and methodology of our work, where GAs were used to optimize the architecture of deep Q-learning networks to play Pacman. Our approach is consistent with the class of hybrid methods described as evolutionary architecture search, a strategy highlighted as particularly useful in neural design for RL agents. EAs are powerful tools for improving deep RL agents by enhancing architectural search and escaping local minima in policy space.

2 Methods

Various configurations were tested to determine which enabled Pacman to achieve the highest scores. For this comparative study, 60 different manual configurations and 60 configurations generated by a GA were tested to see which resulted in higher scores. As a baseline, the game was played using random moves and also played manually by human participants. In random play, the highest score was approximately 3,215, with an average score of 100. After reward shaping, these scores corresponded to 309 and -2.54, respectively (see Figure 8). Reward shaping involved assigning negative rewards when Pacman lost a life and positive rewards for progressing toward goals. Reward shaping refers to the practice of assigning artificial rewards to encourage specific objectives. On average, a typical Pacman game would score around 1,000 points without negative rewards. Rewards³ were shaped by dividing any points received by 10, subtracting 0.01 for each time step, and deducting 15 points for each lost life. As human players, we achieved a high score of approximately 6,500, with an average score of about 1,500. With this information, we were able to assess the performance of the trained models and judge how well they were performing.

Following the approach in the Mnih papers, RGB images of size 210 x 160 pixels were converted to grayscale and resized to 84 x 84 pixels [MKS⁺13, MKS⁺15]. This input tensor was then fed into a CNN-based deep reinforcement learning (DRL) network consisting three convolutional layers with 16 to 64 nodes and kernel sizes ranging from 3 to 12. The output from CNN was then flattened and fed into a FC layer with either 32 or 256 nodes. Like the other parameters, the number of nodes and kernel sizes were chosen according to common industry practice, through trial and error, and through heuristics. The ϵ value was annealed using the ϵ -greedy method, decreasing from 1.0 to 0.10 over the first 100,000 frames. The model was then trained for up to 1 million frames initially during which their performance were evaluated at 100, 250, 750 thousand and finally at 1 million steps. Training was limited to 100,000 steps, as this took an average of three hours to complete, and was considered manageable given our time constraints. Over the course of the experiment, many incremental changes were made, and all modifications were documented as thoroughly as possible.

2.1 Environment

The Arcade Learning Environment (ALE) for the Pacman simulator is freely available from the Farama Foundation [Fou18]. ALE provides detailed outputs of the game state, including internal information. A state refers to the specific configuration of the game at any given moment. For example, when Pacman starts the game, this initial configuration can be considered state 1. If the game runs at 20 frames per second (FPS), there would be 20 different states per second, with each frame representing a separate state, assuming the game changes between frames. The number of possible states increases with the number of elements in the game. Pacman, for instance, has nine possible actions: up, left, down, right, up-right, up-left, down-left, down-right, and a default action (such as no movement). Each possible action can lead to a different subsequent state.

To understand the system’s complexity, the total number of possible states is estimated. For comparison, Chess has a state space of approximately 10^{50} , Go has 10^{172} (see Figure 4), and Pacman is estimated to have 10^{62} possible states (see Figure 21). This estimate considers key game objects such as Pacman’s position, the positions of normal and scared ghosts, the remaining number of lives, and the number of pellets left.

ALE provides state information in the form of random access memory (RAM), RGB images, or grayscale images. The RAM output consists of 128 bytes (1,024 bits) that describe the game state. This information may include Pacman and the ghosts’ locations, pellet status, the number of lives remaining, current score, and rewards received. The Pacman game does not provide any negative rewards. This 128-byte data is much easier to work with, as the information is compact compared to the 210 x 160 pixels (33,600 bits) required for a grayscale image or 210 x 160 x 3 pixels (100,800 bits) for an RGB image. However, in this study, visual information is prioritized because most real-world applications require visual sensory input. For this reason, only RGB images were used as the input.

ALE offers many Atari game simulators, some of which are easy to model, while others are more challenging. According to one paper [BHW13], Pacman is one of the more difficult games to build a model for. Several challenges were encountered during the training process. For example, the wall and pellet colors in the game were identical, making it difficult to differentiate them during preprocessing. This issue was addressed by detecting pellet shapes rather than relying solely on color.

Game	Board size	State space
Go	19 x 19	10^{172}
Chess	8 x 8	10^{50}
Checkers	8 x 8	10^{18}

Figure 4: State space calculation for famous board games [YA12].

2.2 Deep Q-Learning

The Bellman equation, Equation 1, defines optimal Q-function where Q-learning uses the Bellman optimality update rule which is an approximation to iteratively update the Action-Value function, Equation 2, which estimates the expected return for taking an action in a given state and following the optimal policy thereafter. This process involves storing the values gained from different actions and recalling them when needed. In a greedy policy, the action that gives the highest expected return is selected. However, in the long run, this could lead to suboptimal returns due to the exploration-exploitation problem. Exploration is when actions are taken at random, and exploitation is when actions are taken to maximize the outcome based on available information.

$$v_{\pi}(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma v_{\pi}(s')]$$

Equation 1: Bellman equation.

$$q_{\pi}(s, a) = \sum_{s', r} p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_{\pi}(s', a') \right]$$

Equation 2: Action-value function for Q-learning.

During training, an RL agent discovers actions that may return different outcomes, which can be high or low, positive or negative. For example, if Pacman attempts to eat a ghost and loses a life, this results in a negative reward, signaling to the network that this action was undesirable. On the other hand, if Pacman eats pellets and receives a positive reward, this creates an incentive to repeat that action. If Pacman later has the option to eat pellets and chooses to do so because it has learned this yields a positive reward, this is an example of exploitation. If Pacman instead tries a new action, this is an example of exploration.

Over many training episodes, Pacman alternates between exploitative and exploratory moves to collect enough data to make informed decisions. This training process can be repeated many times to develop the model, which ultimately results in a learned policy. Playing the game using the learned policy is called policy evaluation. In reinforcement learning, there are on- and off-policies. Q-learning is considered an off-policy algorithm because it learns the optimal Q-values independently of the policy used to generate actions, often maintaining separate target and behavior policies.

During evaluation, the model only makes greedy, exploitative moves to try to achieve the highest score possible. In order to balance exploration and exploitation, reinforcement learning implements an ϵ -greedy strategy. This allows agents to occasionally select random actions and discover new paths with potentially higher rewards.

To stabilize Q-learning, a separate target network is maintained, which is a copy of the current Q-network. The target network is updated less frequently than the main network. This helps prevent oscillations or divergence in training by providing a stable target for the loss function. For loss, the Huber loss function was used to calculate the difference between the predicted rewards and the actual rewards.

2.3 Genetic Algorithm

Hyperparameter optimization is needed to find the configuration values that are needed in image preprocessing, environment settings, reward shaping, neural network parameters to name a few. In order to train a high performing Pacman agent, optimal hyperparameter values need to be found that will result in a high performing agent, however this is a challenging part of the process primarily because there are infinitely many combinations of hyperparameters. Trying out so many combinations of parameters to see which ones perform and which ones not is a matter of trial and error and time consuming. When starting out heuristics are used to set initial hyperparameters and then refine them over trials. Needless to say, heuristics are not always available nor always best.

Evolutionary Algorithm (EA) is a type of optimization technique inspired by how species evolve in nature by natural selection. Genetic Algorithm (GA) is a type of EA introduced in 1975 in *Adaptation in Natural and Artificial Systems* [Hol10] which aims to find optimal solutions through probabilistic search and selection. The approach applied in this study is not to be confused with Neuroevolution which is artificial evolution of neural networks using GA which searches through space of behaviors for networks [SM02] which is different because many neuroevolution explore neural architecture and weight space without using backpropagation. Our GA operates as a black-box optimizer that explores hyperparameter combinations stochastically, unlike grid search which exhaustively evaluates a fixed parameter grid.

GA is a population-based optimization technique that is well-suited for solving complex problems where the solution space is large, poorly understood, or not differentiable. Unlike gradient-based methods, which require a continuous and differentiable function to guide optimization, GA operates on a set of discrete candidate solutions and does not rely on gradient information. In a GA, a population of individuals represents potential solutions to the problem. Each individual is evaluated using a fitness function, which quantifies how well it performs in relation to the task—in this case, the average game score of a neural network agent after training. The individuals with higher fitness are more likely to be selected to reproduce and pass their characteristics to the next generation. Through evolutionary operations such as selection, mutation, and sometimes crossover, the algorithm introduces variation into the population while preserving high-performing traits.

The selection process chooses top-performing individuals to serve as parents. Mutation introduces random changes to the architecture of these parent networks—such as altering the number of layers or adjusting the size of each layer—to explore new solutions. Crossover combines parts of two parent individuals to create offspring, although in this project crossover was omitted to keep the process simpler. This cycle of evaluation, selection, and mutation is repeated over several generations, allowing the population to evolve toward better-performing neural network configurations.

Genetic algorithms offer several advantages for neural architecture search. They can optimize over complex search spaces, do not require differentiability, and support exploration of diverse architectural configurations. Their population-based nature allows for the parallel evaluation of multiple candidate networks, which is beneficial when computational resources allow batch training. In this study, GA was used to evolve deep reinforcement learning architectures for the Pacman game, and its performance was compared to manually designed networks. The goal was to determine whether an automated method like GA could discover architectures that outperform or match the effectiveness of human-designed models.

```

procedure [P] = standard_EA(pc, pm)
  initialize P
  f ← eval(P)
  P ← select(P, f)
  t ← 1
  while not_stopping_criterion do,
    P ← reproduce(P, f, pc)
    P ← variate(P, pm)
    f ← eval(P)
    P ← select(P, f)
    t ← t + 1
  end while
end procedure

```

Figure 5: A standard evolutionary algorithm [Nun06].

Pseudocode (Figure 5) illustrates a general EA procedure which evolves a population of solutions over time to optimize a given objective. The algorithm begins by initializing a population 'P', and then evaluating it using a fitness function 'eval' to produce fitness scores 'f'. A selection function then filters the population based on fitness values, creating a new, more promising population. The main loop of the algorithm runs until a predefined stopping criterion is met. In each iteration of the loop, a new population is created through the 'reproduce' function, which may involve crossover using crossover probability 'pc'. Then, mutation is applied via the 'variate' function, controlled by mutation probability 'pm'. The new population is evaluated and the best individuals are selected again. The generation counter 't' is incremented at each step. This iterative process continues until the stopping condition is satisfied, gradually evolving the population toward better solutions.

3 Implementation

All experiments were conducted on a desktop computer with an Intel i7-12700k 3.6 GHz processor, 64 gigabytes of RAM and a NVIDIA RTX 3050 GPU with 8 gigabytes of memory. The setup followed the approach described in [MKS⁺13], where the DQN takes as input a 210 x 160 RGB image, which is downsized to an 84 x 84 grayscale image. The first convolutional layer consists of 16 filters with a kernel size of 8 x 8 and a stride of 4, followed by a second convolutional layer with 32 filters, a kernel size of 4 x 4, and a stride of 2. Both layers used the ReLU activation function. The final hidden layer was a FC layer with 256 nodes, followed by an output layer with nine outputs, one for each available action.

Moreover, when ghosts become edible, they start flashing blue (see Figure 15). However, since the models initially used grayscale images, this information was lost, making it more difficult for the models to distinguish between edible and non-edible ghosts. To counter this problem, the model was provided with RGB input frames. However, this solution increased the input space threefold, which consequently required longer training times and a larger number of frames. This conflicted with our goal of identifying configurations that could train efficiently within limited computational resources.

To address the problem of increased input size, the frame size was reduced from 84 x 84 pixels to smaller dimensions. Several different combinations were tested, such as 42 x 42 pixels and 21 x 21 pixels, but these did not result in any improvement. It was considered that excessive downsampling might prevent the model from recognizing important features, but this was not a straightforward conclusion. Models were then trained on 84 x 84 pixel RGB frames for one million frames to determine whether the model could learn effectively.

After many attempts, some preliminary promising results were documented. The maximum score achieved during training was 1040, and the average score⁴ over 30 game plays after training was 701, measured across 10 episodes in Model E (see Figure 9). Several models (A, B, C, D, and E) were trained, achieving average scores between 370 and 500. These scores outperformed the random agent baseline (mean = 100), confirming that learning had occurred, even though the score was not that high. This was expected, as the models were initially trained for only one million frames, compared to the 10 million frames used in the Mnih experiments. However, once again, a more efficient method was needed to achieve high performance with fewer training frames due to time constraints.

Many image frame transformations were tried in order to keep the input data small while not losing features information too much for the model to be able to learn. For example, excessive downsampling of image frames leads to a loss of important game information (see Figure 10a). Walls were removed by detecting only pellets, which were identified by searching for small rectangular shapes of the pellet color (see Figure 11b). The size of the pellets was also increased (see Figures 12b and 13b) before downsampling the image. This adjustment prevented the pellets from disappearing, as had occurred in previous attempts (see Figure 11c). Some of our image frame transformation techniques were documented (see Figures 10, 11, 12). Although these transformations produced visually interesting results, the models did not perform as well as expected. Some of these transformations were promising, but it was believed that successful implementation would require training on larger frames leading to longer training times. In order to apply color filters an analysis of all the colors used in the game were identified and documented (see Figures 16, 22).

After many unsuccessful image transformation techniques, a multi-channel tensor was created, with each channel clearly marking objects of interest (see Figure 13b). With this technique, model performance improved much more quickly. Initially, five different channels were created—one each for Pacman, ghosts, pellets, scared ghosts, and cherries. It was soon realized that edible and non-edible channels could be merged to reduce the number of channels, thereby decreasing the input size. A visual representation of a such 3 channel input tensor is depicted in Figure 13c. This proved beneficial because the network can learn faster with a more condensed and meaningful state space. Edible ghosts and cherries do not appear frequently, the data in those channels were mostly zero, for this reason they were merged into other channels. From the nine possible actions, four diagonal actions—up-right, up-left, down-left, and down-right—were removed. These changes made the output space smaller, which in theory should make learning more efficient.

Walls were left out deliberately to avoid excessive hand-crafted features, which was precisely what we wanted to avoid. The input tensor size needed to match or exceed the number of pellets vertically (top-down) and horizontally (across). The pellets in the two-dimensional image were not evenly spaced or equally distributed; however, this was not an issue. The maximum number of pellets was 15 vertically and 18 horizontally, so the input tensor size needed to be at least 15 x 18. The input tensor size was adjusted during our manual training runs. We tested several combinations, including 16 x 18, 22 x 20, 14 x 20, 16 x 20, 35 x 32, 28 x 36, 30 x 26, and 32 x 32. There was no immediate noticeable difference in performance when changing the input tensor size, but the training time increased for larger tensor sizes. For GA models, the input tensor size was kept constant at 20 x 24. Three-channel tensors were used to train various models, while most other variables remained unchanged. For hand-crafted models, additional techniques were tested to determine if better results could be achieved. For example, the number of repeated actions taken every k-th step was varied, but this parameter was mostly kept at 1, unlike the Mnih papers, which used a value of 4. This choice was made to enable Pacman to react more quickly to ghosts.

After creating multiple manual models, a GA was set up to automatically find optimal network configurations. Many parameters could have been optimized. However, optimizing more parameters requires additional time and computing resources. The most fundamental decision when designing a neural network is choosing the number of layers and the number of nodes in each layer. This study is primarily focused on these aspects.

In addition to the number of layers and nodes, other relevant hyperparameters include the activation function, dropout rate, learning rate, optimizer type, batch size, discount factor, and gradient clipping value. For the exploration strategy, relevant parameters include the ϵ -greedy rate, its decay rate, and the minimum ϵ value. For memory management, important parameters are the replay buffer size, training frequency, and target network updated every 10,000 steps. This corresponds to about ten Pacman game plays, so the target network is updated roughly every ten games. This improves training stability and avoids Q-value overestimation. Additional considerations include whether to use a Deep Q-Network (DQN), and whether to implement Priority Experience Replay (PER). A DQN is a Q-Network with a neural network architecture used in deep reinforcement learning (DRL). PER is a variant of ER that offers better performance but has a more complex setup. Other factors include memory buffer settings, reward shaping, the number of frames to stack, and the number of frames for which a selected action is repeated. For the genetic algorithm, relevant parameters are population size, mutation rate, crossover rate, selection method, fitness function, elitism, and convergence criteria. All of these parameters could be encoded in the genetic algorithm for optimization. However, optimizing so many combinations would require significant processing time, which is impractical.

After selecting which parameters to tune, we defined several limitations for the genetic algorithm to operate within. In our network, there are two types of layers: CNN layers and FC layers. The search space was limited to allow between zero and three CNN layers, and between one and three FC layers. This means that each network must have at least one layer and no more than six layers in total. CNN layers were optional, so the network could include them or not, but at least one FC layer was required. The number of nodes in FC layers could range from 16 to 1024, while CNN layers could have between 8 and 128 nodes. Setting these limits may restrict the full potential of the GA, but this was necessary to complete the experiment within the available time frame.

For the GA parameters, 25% was selected as the δ , which denotes the upper and lower range used when choosing a new value during mutation for the number of nodes. The population size was set

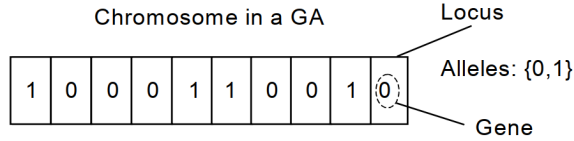


Figure 7: Bitstring of a chromosome in standard genetic algorithm. Each locus can assume either 0 or 1 [Num06].

to 20, with elitism of six, five always brand-new random individuals, and the remaining individuals selected from the top-performing models of previous generations. In the very first generation, all 20 individuals were initialized randomly. In the next generation, the six top-performing models were mutated within the δ range. When mutating a model, there was a 20% chance of flipping the state—either adding or removing a CNN layer. For both FC and CNN layers, there was a 50% chance of either adding a new layer at a random position or removing an existing one. Six generations were evolved, producing a total of 120 models. For our statistical evaluation, only the latter 60 models were considered. This is because the GA starts with random models; including them in the assessment would be unfair.

Finally, the parameters are encoded for processing. One common method is to create a bitstring (see Figure 7), where each bit represents a binary feature, and each unique bitstring corresponds to an individual in the population. In our setup, a custom variant of a bitstring chromosome was created using a Python dictionary with three keys: *fc_layers* (an array), *is_cnn* (a boolean), and *cnn_layers* (an array). The arrays hold the count of nodes in each layer. For example, an individual could consist of one FC layer with 128 nodes and two CNN layers with 64 nodes each. This configuration can be represented by a 'fingerprint' such as [128],1,[64,64]. This custom data structure allowed us to easily manage and process the population.

To reduce the creation of divergent models, recombination was not applied. Instead, models were mutated directly by modifying parts of the sequence to create new individuals. Finally, after training, the scores from 30 games were averaged to determine each individual's overall score. This average score served as the fitness function for our GA, which measures how well an individual performs compared to others in the population.

4 Results

The three-channel tensor appeared to be effective, as the average scores increased significantly. An effective method was identified for training high-performing models with fewer timesteps. Scores are calculated as the average over 30 games after training for 100,000 steps, unless stated otherwise. The results⁵ from the manual network design implementation, with 60 models trained, are detailed in Figures 23, 24, and 25. Results from the GA, with 120 models trained, are presented in Figures 26, 27, 28, and 29. Each test has been given a unique ID to properly store and document its results. For manual network design, models were trained up to 1 million frames to determine whether performance would continue to improve with more training frames. This was not the case; in fact, the models usually resulted in poor scores most likely due to overfitting. Overfitting occurs when the network learns the training data so well that it performs poorly on new, unseen data. However, performance also depended on the number and size of layers, as well as changes

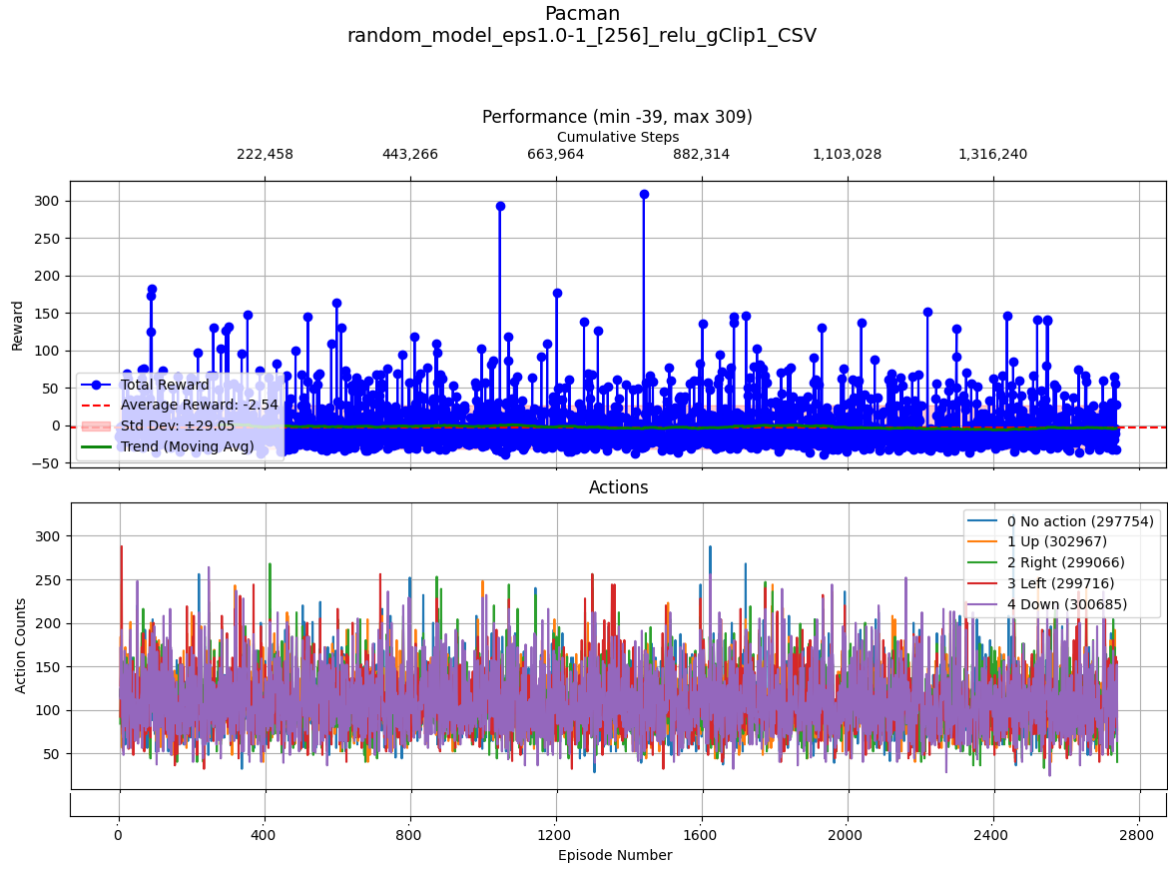


Figure 8: Performance of random play over 2700 episodes and 1.5 million frames. Scores are scaled down and reward shaped³.

in input tensor dimensions across different models. The documentation includes maximum scores, overall average scores, and average scores over 30 games at 100,000; 250,000; 500,000; 750,000; and 1,000,000 frames. Finally, the tables show the configurations used to achieve these results. Each configuration includes the size of the two-dimensional tensor used, as well as the size and number of both FC and CNN layers.

From the 60 manually designed networks, the highest score was achieved by model ID 47, which used a 16 x 20 grid tensor and consisted of two FC layers with 256 nodes each and no CNN layer. Over the course of 1 million frames, this model showed fluctuating performance. Despite this, its highest score was 474, which is lower than that of model ID 55, which achieved 784 points with two FC layers of 512 nodes each and no CNN layer. This was a remarkable result, as this network achieved the highest average score among all 60 hand-crafted models.

Across models ID 1 to 60, several parameters were varied—including input tensor size, number of frames stacked, use of flicker-free frames, and number of available output actions—in an attempt to increase model performance. The results tables are color-coded, with red highlights indicating lower scores and green indicating higher scores. It appears that models with only FC layers and a larger number of nodes performed better. The worst results came from networks with fewer nodes, such as model ID 54, which had two FC layers of 16 nodes each and resulted in an average score of only 340. This is likely due to underfitting, as the information from Pacman exceeded what the network could process, leading to poor performance.

On the other hand, larger networks, such as model ID 60 with three FC layers of sizes 3840, 1920, and 960, could be considered undertrained. It initially scored 664, but this increased to 1,173 after training for 250,000 steps. However, this model’s performance declined with further training, which could be due to overfitting.

Although adding more layers might improve performance, the number of layers was limited to between one and three to avoid vanishing gradient problems and to keep the comparative study manageable. Vanishing gradient is when due to too many layers the values in the nodes become so small they become very small. This prevents additional training from having any further effect.

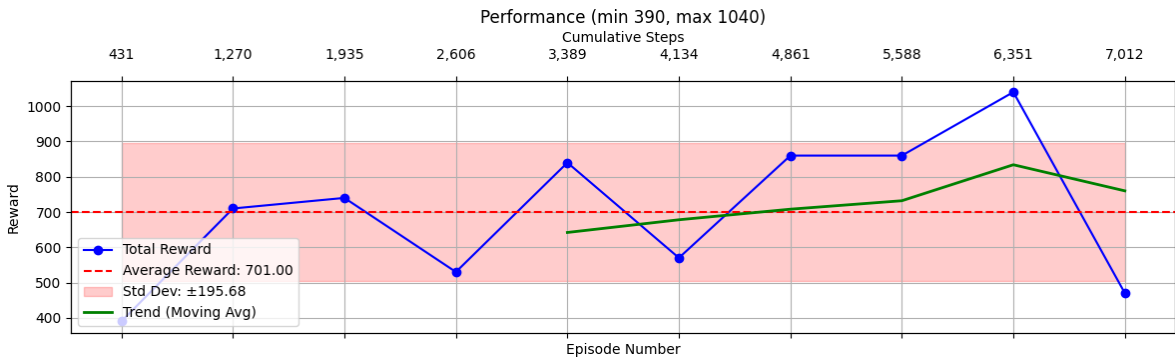


Figure 9: Model E performance results over 10 episodes.

For model IDs 22 and 23, only a single FC layer of 256 and 512 nodes, respectively, was used, resulting in very poor scores of 311 and 364. This was expected because a single layer is too shallow to capture complex patterns. For moderate scores, there needed to be at least two FC layers with 256 nodes each, as seen in model ID 49, which achieved an average score of 567. This configuration of two FC layers with 256 nodes each was tested multiple times using different tensor sizes. For

example, model ID 3 used a 16 x 18 input tensor and achieved a score of 413. However, a different input tensor size, such as 22 x 20 in model ID 10, resulted in a lower average score of 392. With a smaller input tensor of 14 x 20, model ID 17 produced a score of 411. This score improved in model ID 38, where an input tensor size of 35 x 32 resulted in a much higher score of 701. This suggests that having a larger input tensor size is beneficial. This may occur because, on a smaller two-dimensional input tensor, many cells are condensed into one. As a result, the system may not know exactly how many steps are needed to take the action required to continue moving.

To understand this, consider an example where the tensor matches the image exactly, pixel for pixel. If the cropped version of the frame (with the bottom part displaying scores removed) is 170 x 160 pixels, and the input tensor is also sized 170 x 160, then each action taken by Pacman moves it by a predefined distance, which is assumed to be one pixel in each direction. This means that if Pacman is at coordinates $x=1, y=1$, then one step to the right (east) will move it to $x=2, y=1$.

Now, imagine the tensor size is reduced by a factor of five, resulting in an input tensor of 34 x 32. When Pacman is at $x=1, y=1$ in the input tensor, it aligns with its actual position. However, if Pacman moves right (east), its actual position becomes $x=2, y=1$, but in the condensed tensor, it still appears to be at $x=1, y=1$. This creates ambiguity for the deep reinforcement learning (DRL) system, as it cannot determine how many steps are required for the action "move right" to actually result in a change to $x=2, y=1$ in the tensor representation.

To address this issue, the same action was repeated k times, similar to the approach used by Mnih in their reports. However, their reasoning may have been different, as their downsampled frame was 84 x 84 pixels, which, based on the original dimensions (210 / 4 and 160 / 4), does not directly correspond to 84 x 84 pixels.

One noticeable change from the manual network design was that the tensor size was increased to 20 x 24 pixels. This change was made because, as shown earlier, larger input tensors resulted in better outcomes for the same network configuration. The models generated by the GA are from model IDs 61 to 180, resulting in 120 GA-trained networks—twice as many as the manually designed networks. For a fair comparison, it would be inappropriate to include the first few GA networks, as the GA starts with completely random models and only produces improved models after several iterations. It is up to us to decide which GA models are admissible for benchmarking and comparison. Since there were 60 manual networks, the latest 60 GA models out of the 120 will be used for statistical analysis. However, it is also important to analyze how the GA progresses from the beginning.

The GA produced model ID 61, which starts off with three FC layers of 645, 427, and 473 nodes, and two CNN layers of 109 and 68 nodes, resulting in a meager score of 70. A person designing a network would probably never create this configuration, as it appears quite random. This highlights why algorithms such as GA can discover novel solutions—they are not limited by preconceived assumptions. Humans are prone to bias; for this reason, manually constructed networks may exhibit suboptimal performance.

For the most part, the average scores fluctuate greatly, with only a few models scoring over 1,000. These include model IDs 77, 83, 127, 131, 134, 141, 145, 147, 151, 153, 158, 159, 164, 167, 168, 169, 172, 173, 174, 175, 176, 177, 178, 179, and 180. From this sequence, it is clear that almost all models scoring higher than 1,000 appear in the later stages of training. The midpoint of the GA neural network experiment is model ID 120. Between model IDs 61 and 120, only two models—IDs 77 and 83—scored above 1,000. This demonstrates that there was a clear improvement in scores as the GA progressed.

Among all GA models, the highest average score was achieved by model ID 172, with 1,664 points,

using only one FC layer of 867 nodes and one CNN layer of 131 nodes. This model also outperformed all manual models. This was quite a discovery because the network used a minimal number of layers yet achieved a high score. This was unexpected, and it demonstrates why GA is so useful—because of its ability to discover unexpected solutions. Most of the later GA models also scored quite highly, which was a very positive outcome. It is also worth noting that the last nine models all included CNN layers, suggesting that networks with CNN layers perform better for this application than those with only FC layers.

4.1 Comparison

In this section, a comparison is made between manually designed neural networks and GA-generated neural networks. Of the 60 manual networks, only three achieved an average score higher than 1,000: model IDs 50, 51, and 55. In contrast, the GA created twenty-five models that scored higher than 1,000. This indicates that only 5% of manual models achieved average scores above 1,000, whereas 41% of GA models did so. One weakness of the manual models is that most were composed only of FC layers, due to our preconceived idea that an FC network would perform better with the tensor input than a CNN network. This assumption was incorrect, as demonstrated by the GA.

Another unexpected discovery was made during the design of manual models. It was previously assumed that achieving high scores required a large number of nodes in the network, but model IDs 176, 177, and 180 prove that this is not the case. Model ID 176 scored 1,395 using only one FC layer with 168 nodes and one CNN layer with 129 nodes. Model ID 177 scored 1,035 with one FC layer of 442 nodes and one CNN layer of 55 nodes. Model ID 180 scored 1,267 with one FC layer of 313 nodes and one CNN layer of 55 nodes. The total number of nodes for these models is only 297, 497, and 368, respectively. To our understanding, having fewer layers and nodes creates a computationally efficient model, requiring fewer calculations. These are remarkable results from the GA.

Lastly, despite training all manual models extensively—up to 1,000,000 frames—the highest average score was 1,493, achieved by model ID 19 after being trained for 750,000 frames. This shows that longer training does not necessarily lead to better results. With these outcomes, the preliminary judgment is a clear win for the GA. Nevertheless, a statistical analysis was conducted to confirm this.

4.2 Statistical Analysis

In this section, the Mann-Whitney statistical test is used to determine whether there is a significant difference between two independent groups. This statistical test was chosen because the performance scores of the models were not normally distributed, making non-parametric methods more appropriate. It allows comparison between two independent groups (manual and GA models) without assuming equal variances or normality.

The scores from both groups can be ranked, and the data are not normally distributed. Here, μ_1 represents the mean score of manual models, and μ_2 represents the mean score of GA models.

$H_0 : \mu_1 = \mu_2$ or in other words manual model scores are equal to GA model scores.

$H_1 : \mu_1 \neq \mu_2$ or in other words manual model scores are not equal to GA model scores.

The statistics calculations were as follows:

$$\begin{aligned} n_1 &= 60, n_2 = 60, R_1 = 2833.5, R_2 = 4426.5 \\ U_1 &= 2596.5, U_2 = 1003.5, U = 1300.5 \\ \sigma_U &= 190.526, \mu_U = 1800 \\ z &= -4.181, p \approx 0.000, \alpha = 0.05 \\ r &= \frac{z}{\sqrt{N}} = \frac{-4.181}{\sqrt{120}} \approx -0.381 \end{aligned}$$

The test yielded z-score of -4.181 and two-tailed p-value ≈ 0.000 . This z-score falls well beyond the critical z-value for $\alpha = 0.05$ (± 1.96), indicating strong evidence against the null hypothesis and hence we rejected the null hypothesis. Furthermore, to understand the variability in model scores, we calculated the standard deviation of the U distribution to be 190.526. Although Mann-Whitney U is a non-parametric test and does not directly yield a confidence interval, the z-score and p-value provide inference strength equivalent to a 95% confidence level or higher, confirming that the performance difference is statistically significant.

To measure the effect size, we computed the rank-biserial correlation, denoted as r , which was -0.381—an effect size metric for the Mann-Whitney U test. The absolute value of $r = 0.381$, which indicates a moderate to large effect size. This means the difference between the manual and GA groups is practically meaningful. GA not only showed a higher likelihood of outperforming manual models, but did so with substantial performance gaps. According to these findings, GA models performed better than manual models, and this statistical analysis confirmed this result.

5 Discussion

The results made it clear that using a GA model is much more effective than designing the network manually. However, it would not have been possible to begin directly with GA, as some base code demonstrating initial progress was required. The GA requires a functioning codebase and cannot set itself up automatically. Even with only three parameters being tuned, each generation in the GA takes a significant amount of time because it depends on a population. The larger the population, the higher the chance of discovering better combinations of hyperparameters. However, the biggest obstacle is the time required to train for 100,000 frames. As mentioned earlier, there are many hyperparameters that could be tuned, and it is impractical to implement all of them in the GA because it would require an excessive amount of time.

For the RL algorithm to be effective, it is important to have a reward shaping strategy in place, as this encourages models to pursue a given goal. In Pacman, it was found that there were no negative rewards. Without this, it may not have been possible to achieve these results. The choices made in formulating reward shaping may or may not lead to an optimal outcome. It is up to the designer to determine this through trial and error. The process is imperfect and time-consuming. Hypothetically, this task could also be assigned for the GA to solve. However, this would add more hyperparameters and further exacerbate the problem of time efficiency, as it would require even more training time. Graphics Processing Units (GPUs) are commonly used to train AI models. More advanced GPUs, or multiple GPUs used simultaneously, could reduce training time, but this could be a costly proposition.

When replaying trained models, some recurring patterns were observed, such as Pacman tending to follow the same path almost every time. This behavior is unlike that of a human player, who

tends to take different paths. However, after extensive training, the network may have learned the best path. As a result, it appears to us that Pacman is simply repeating the same path, but in reality, the network has identified what it believes to be the optimal route, which may be only a handful—or even just one—optimal path.

This fixed path becomes problematic for Pacman because the ghosts move randomly. This is why a well-trained model might perform well in one game but poorly in others. This stochasticity in the environment is the primary challenge for networks to overcome.

While watching gameplay, Pacman was sometimes seen hitting a wall and just staying there. The reasons behind this strange behavior are not fully understood, and we can only speculate as to what might be happening. One theory is that, due to the mapping of frames onto the tensor, there were some mismatched pixels, which could place Pacman in a position where it stands still next to a wall. However, this explanation seems unlikely, as Pacman must always take one of four possible actions. Nevertheless, it was frequently observed that Pacman remained in the same spot on the map until it was eaten.

Another possible reason is that, since wall information is not available, Pacman cannot differentiate between a clear path and an obstruction. Nevertheless, it was not entirely clear why Pacman sometimes chose to come to a standstill.

The idea of learning directly from the pixels in frames is promising and may unlock solutions for a variety of applications. The biggest challenge is the sheer amount of data generated and the processing required for a neural network to learn to optimize a given objective function. Having access to fast computers can improve this process. Just over a decade ago, the computing power available today did not exist. Computing power is expected to continue to increase in the coming years. With greater and faster computing resources, neural networks could be trained more quickly, saving time.

These findings have significant implications beyond games. For example, autonomous robots—such as delivery drones, self-driving cars, logistics robots, and surveillance systems—must make optimal decisions with limited information, often relying on visual sensory inputs similar to those demonstrated in Pacman. Using GA to optimize neural network architectures can accelerate the discovery of effective models without requiring exhaustive manual tuning, thereby saving resources. Learning from visual inputs remains a compelling and realistic approach, especially in scenarios where raw image data is often the primary or only available input. The challenge lies in using visual data efficiently with robust algorithmic modeling. The combination of DRL, CNNs, and GA forms a powerful toolkit capable of addressing a variety of complex real-world scenarios.

5.1 Conclusion

GA-optimized models achieved an overall ⁶ average score of 903, outperforming manually designed models, which averaged 681 overall. This confirms the efficiency of evolutionary search in discovering optimal network configurations under limited training time. Algorithms such as DRL, used in playing Pacman, work but are far from efficient compared to human players. One hundred thousand frame steps correspond to about 180 game plays. This is quite a lot of games for a person to play in one sitting. Human players do not need to play that many games to become proficient.

Human players can understand the game objective, available actions, and their consequences even before starting to play. This is an area where current artificial intelligence falls short. To compensate for this flaw, algorithms are trained on a large number of examples (data) to cover as

many possibilities as possible, and this process is termed 'learning'. However, this kind of learning is far from having an understanding of the underlying mechanics and the ultimate objective of the game.

The idea of exploring the state space through exponential numbers of game plays requires exponential computational resources, and this cannot be considered a smart solution. More efficient methods are needed to better replicate the intelligence of the human mind. The lack of both advanced algorithms and efficient computing power is what holds us back from creating more intelligent systems.

5.2 Further Research

The success of this study was due to using a hybrid solution that involved detecting game features—such as Pacman, pellets, and ghosts—mapping them into an input tensor, and feeding this tensor into a neural network with a Q-learning algorithm. The process of manually encoding features could be eliminated by creating a system that can automatically detect features to be mapped into the tensor. Automating this part of the process could save time and effort, and also improve the mapping of additional characteristics that may be useful for model training.

CNNs were particularly effective in producing high scores with fewer network nodes. For all of these networks, a kernel size of 3 and a stride of 1 were used. A further study could be conducted to explore different kernel sizes and strides to determine their effects.

For memory, ER was used, but another variant, PER, exists, which some papers claim delivers superior results. An attempt was made to implement it in this study, but it was abandoned due to its complexity. A comparison of the performance of ER and PER in Pacman could be explored in future work.

Finally, during the progression of the study, novel approaches were explored, such as the 'flicker free' method we coined. In this approach, multiple tensors are combined over j frames to try to capture ghosts that temporarily become invisible. This idea is analogous to frame stacking, but we believe it is better because it does not require the input tensor to grow; the information from j tensors is combined into a single tensor. This provides the network with transitional information about moving objects within a single tensor.

However, this approach was abandoned during the experiment because the processing was too time-consuming due to the way it was coded. To explain this in more detail, the ghosts can disappear for a maximum of eight frames. If we were to create a composite tensor that adds information on top of the previous tensor over nine frames, the model would retain information about the ghost's last position before being updated again after nine frames. This would eliminate the element of surprise for the model.

The implementation involved creating a queue that held the last eight frames, with the latest frame added to the top of the queue. A single consolidated tensor would then capture the movement and transitions. This required processing eight frames every time, which significantly increased the training time. However, it was later realized that this was not the optimal approach. Instead of creating a queue that held RGB frames, we should have used the previous eight tensors, as tensors are lightweight and would have required only matrix multiplications for the GPU, rather than additional image processing.

References

- [BHW13] Luuk Bom, Ruud Henken, and Marco Wiering. Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. *CiteSeer X (The Pennsylvania State University)*, 04 2013.
- [Fou18] Farama Foundation. Ale documentation. https://ale.farama.org/environments/ms_pacman/, 2018.
- [Hol10] John H Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. Cambridge, Mass. Mit Press [Ca, 2010.
- [LBBH18] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278–2324, 2018.
- [LHT⁺24] Pengyi Li, Jianye Hao, Hongyao Tang, Xian Fu, Yan Zhen, and Ke Tang. Bridging evolutionary algorithms and reinforcement learning: A comprehensive survey on hybrid algorithms. *IEEE Transactions on Evolutionary Computation*, pages 1–1, 01 2024.
- [MKS⁺13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.
- [MKS⁺15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharmashan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 02 2015.
- [NAM] History the official site for pac-man - video games and more.
- [Nun06] Leandro Nunes. *Fundamentals of Natural Computing*. CRC Press, 06 2006.
- [SM02] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 06 2002.
- [Sut18] Richard S Sutton. *Reinforcement Learning, Second Edition : An Introduction : An Introduction*. The Mit Press, 2018.
- [YA12] Arturo Yee and Matías Alvarado. Pattern recognition and monte-carlotree search for go gaming better automation. *Lecture notes in computer science*, pages 11–20, 01 2012.

Appendices

Glossary

ALE	Arcade Learning Environment
CNN	Convolutional Neural Network
DQN	Deep Q-learning Network
DRL	Deep Reinforcement Learning
EA	Evolutionary Algorithm
ER	Experience Replay
ENAS	Evolutionary Neural Architecture Search
FC	Fully Connected
FPS	Frames per Second
GA	Genetic Algorithm
ID	Identification
MSE	Mean Squared Error
MDP	Markov Decision Process
NEAT	Neuroevolution of Augmenting Topologies
PER	Priority Experience Replay
PIL	Python Imaging Library
RAM	Random Access Memory
RL	Reinforcement Learning
ReLU	Rectified Linear Unit
RGB	Red Green Blue
SGD	Stochastic Gradient Descent

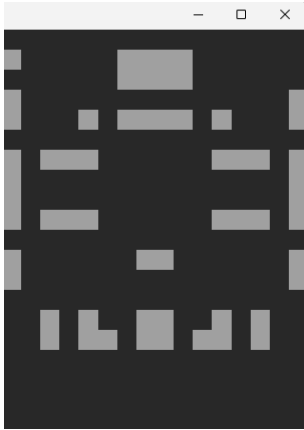
Notations

X	input matrix
\in	element of
\mathbb{R}^n	real number with n dimensions
\hat{y}	prediction
ε	probability of taking random action in ε -greedy policy
γ	discount rate
λ	decay factor
η	learning rate
δ	delta
a	action
a'	next action
s	state
s'	next state
r	reward
t	timestep
π	policy
p	probability
$p(s', s, a)$	probability of transitioning into s' from state s by taking action a
$r(s, a)$	expected reward from state s after taking action a
$v_\pi(s)$	value of state s under policy π
$q_\pi(s, a)$	value of taking action a in state s under policy π
$q_\pi(s', a')$	value of taking next action a' and arriving in next state s' under policy π
$\pi(a s)$	probability of taking action a in state s
$\pi(a' s')$	probability of next action in next state
$p(s', r s, a)$	transition probability from state s to state s' by action a with reward r
n_1	number of experiments in manual networks (group 1)
n_2	number of experiments in GA networks (group 2)
R_1	sum of ranks in manual networks (group 1)
R_2	sum of ranks in GA networks (group 2)
U_1	U statistic from manual networks (group 1)
U_2	U statistic from GA networks (group 2)
U	U statistic which is lower of U_1 and U_2
μ_U	mean of U
σ_U	standard deviation of U
α	significance level of rejecting the null hypothesis
$z = \frac{U - \mu_U}{\sigma_U}$	z-score, standardized test statistic score
H_0	null hypothesis
H_1	alternative hypothesis

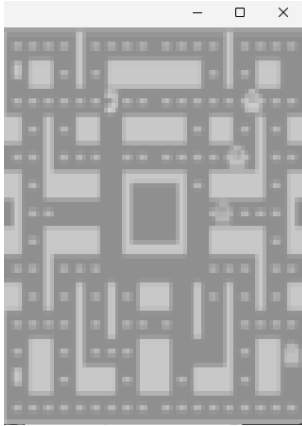
Notes

1. Variant of Pacman, Ms. Pacman was used but always referred to as Pacman.
2. Nodes, neurons, units could be used interchangeably as they refer to the same thing.
3. Reward shaped calculations are calculated as, any reward received divided by 10, -0.01 for every step, and -15 for losing life. The step deduction amount is not included because it is very small and negligible.
4. Scores given refer to average of 30 game plays after 100 thousand frame training unless stated otherwise.
5. Results in Figures 23 to 29 are based on 2 dimensional grid modeled detecting Pacman, ghosts, and pellets with learning rate of 0.0001 with starting epsilon of 1.0 decreasing linearly over 100,000 frames and 0.10 thereafter and RELU as activation function. The cells have color grading to indicate ranking for numbers where in each column lowest values are marked with red, medium in orange and large in green color.
6. The GA optimized model average score of 903 is based on the latter 60 average game score and manually designed model average score of 681 is based on 60 average games scores.

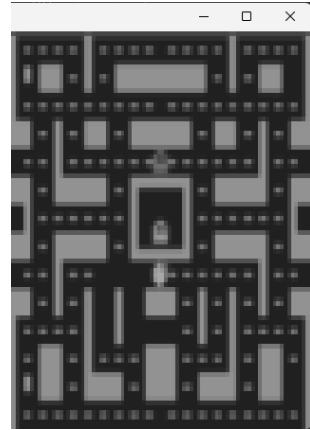
Images



(a) Downsampled grayscale frame of where most of information is lost

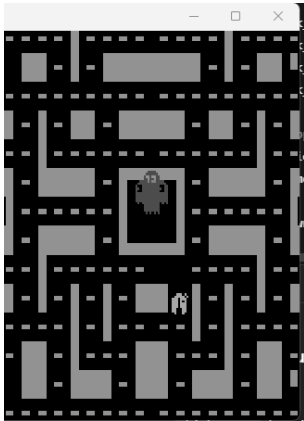


(b) Downsampled grayscale frame with some blur

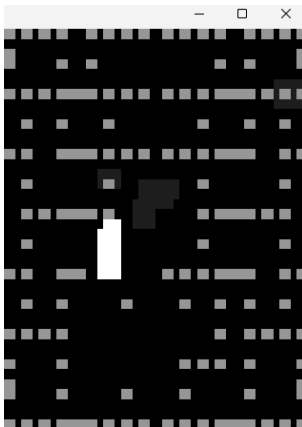


(c) Downsampled grayscale frame using bicubic filter.

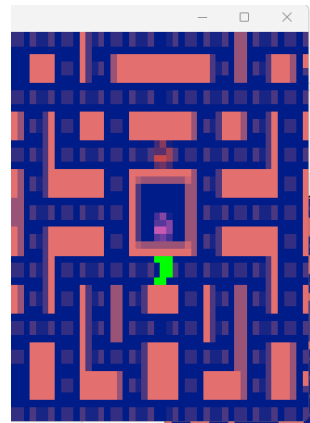
Figure 10



(a) Grayscale frame using PIL format.

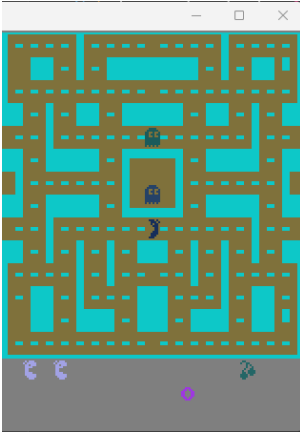


(b) Using a 2 dimensional grayscale grid canvas to draw boxes where pellets are detected.

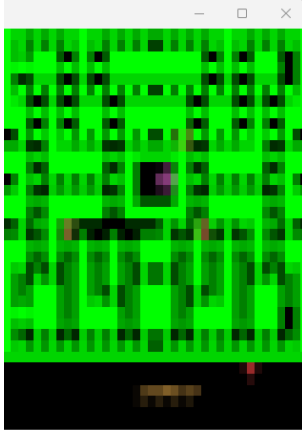


(c) Downsampled color image to emphasize Pacman by bright green color.

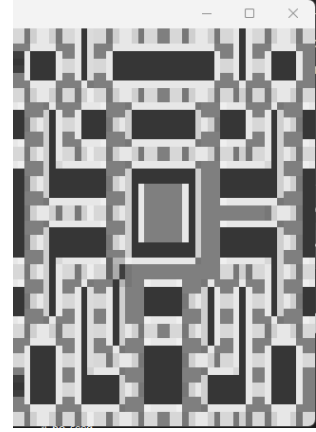
Figure 11



(a) Image frame transformation by mixing up different color channels.

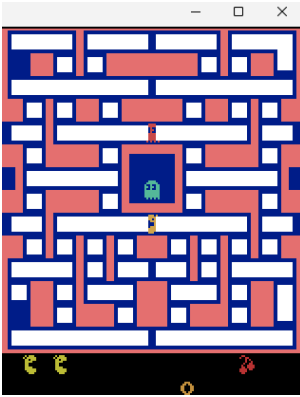


(b) Downsampled color image frame with pellet color

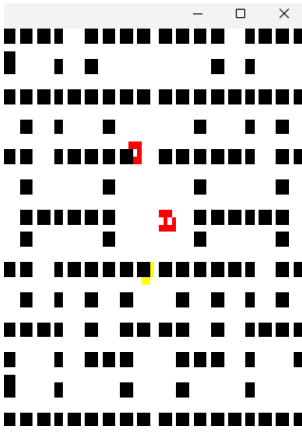


(c) Downsampled grayscale image with white squares drawn where pellets are detected

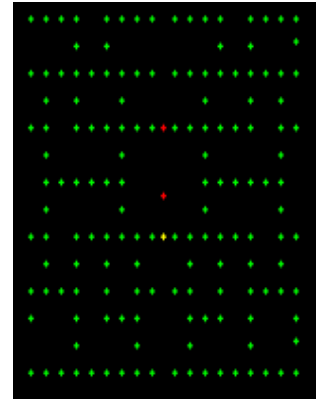
Figure 12



(a) Downsampled color image with white squares drawn where pellets are detected



(b) Using a blank 2 dimensional grayscale grid canvas to draw boxes where pellets are detected and draw colors at ghost and Pacman locations effectively removing objects considered noise.



(c) Visual representation of the 3 channels combined. The 3 channel tensor is fed into the network as input, not the visual representation.

Figure 13

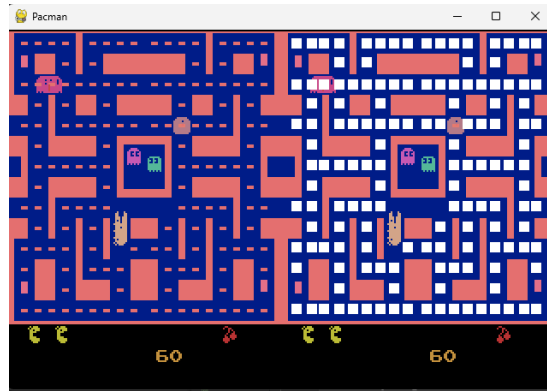


Figure 14: Image overlay with 4 frames stacked to detect motion and pellets overlaid with white boxes.

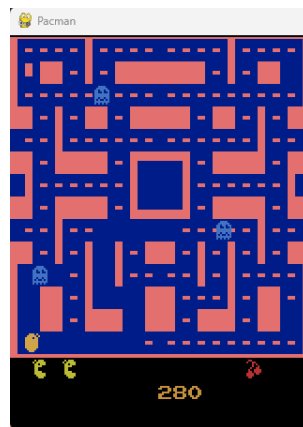


Figure 15: Game state with scared ghosts



Figure 16: Pacman image color pallet.



Figure 17: Pixel count of a pellet. The measurement shows that a pellet is made up of 8 pixels.

Graphs

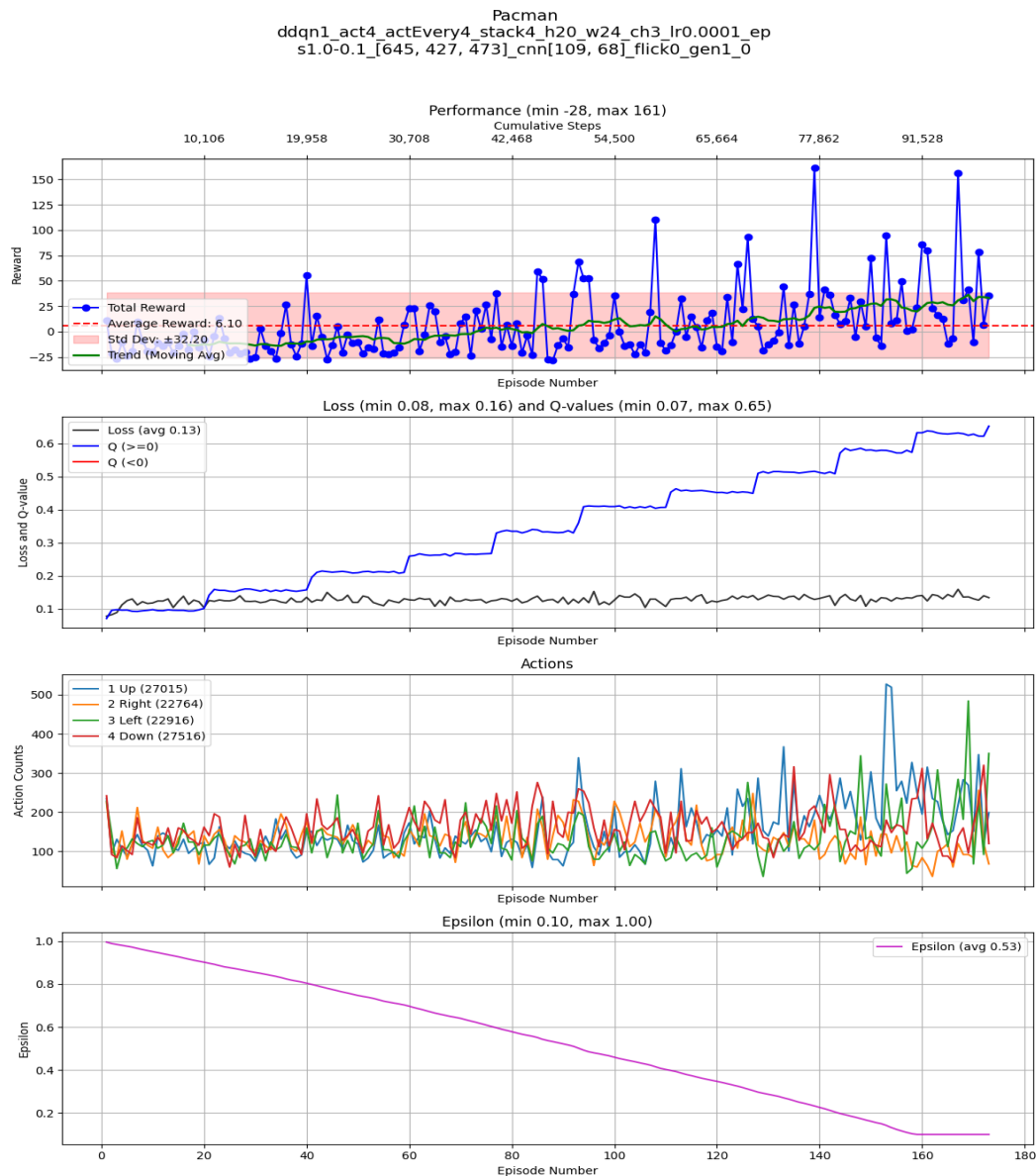


Figure 18: Performance and analysis graphs of experiment model ID 61 that was the very first model by GA.

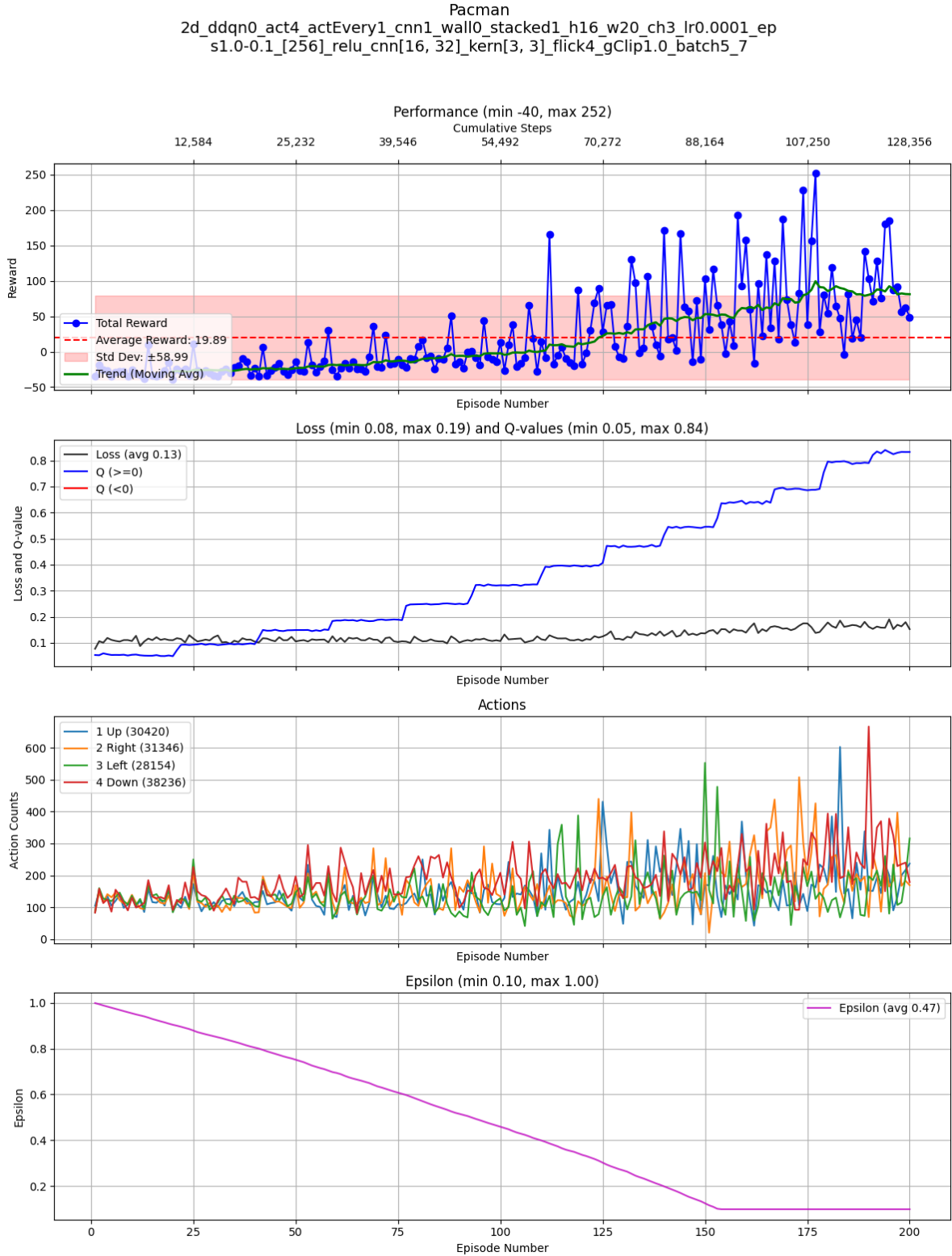


Figure 19: Performance and analysis graphs of experiment model ID 51 manually designed which produced the highest average score over 30 game plays of 1113 after training for 100 thousand frames.

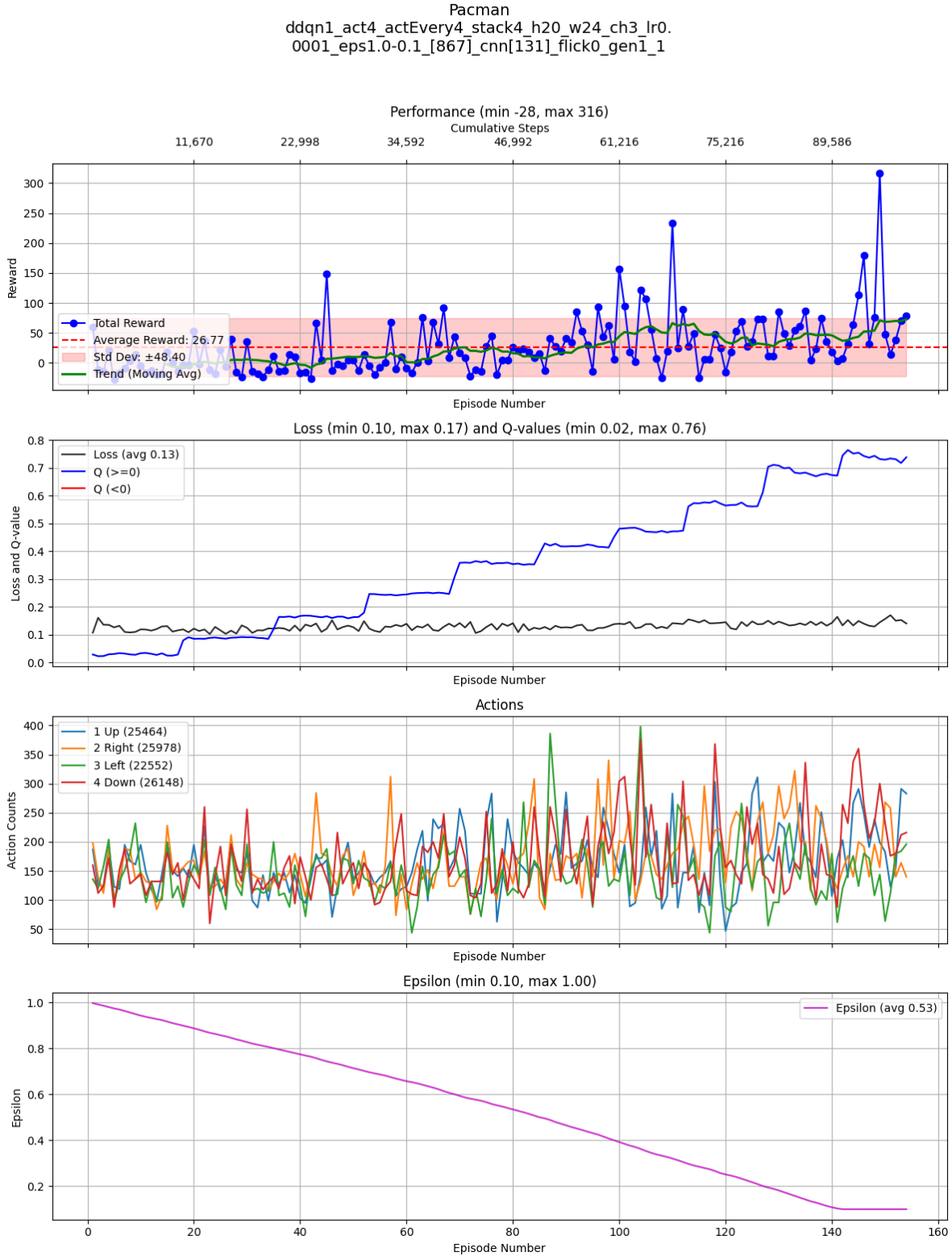


Figure 20: Performance and analysis graphs of experiment model ID 172 created by GA which produced the highest average score over 30 game plays of 1664 after training for 100 thousand frames.

Tables

Calculation	Estimated count	Estimated count (scientific notation)
Board size (210 width - 40 crop) = 170 pixels x 160 pixels height, downsized to 20 width x 24 height = 480 positions	-	-
9 Pacman movement directions	9	9
Pacman position over 480 tiles	480	480
4 normal ghosts or scared ghosts over 480 tiles x 2 = 960	960^4	$\sim 8.493 \times 10^{11}$
154 pellets status either eaten or not	2^{154}	$\sim 2.283 \times 10^{46}$
Number of lives	3	3
Total state space		$\sim 2.513 \times 10^{62}$

Figure 21: State space calculation for Pacman.













RGB Code	Color	Object
(210, 164, 74)		Pacman
(228, 111, 111)		Walls and pellets
(84, 184, 153)		Blue ghost
(180, 122, 48)		Yellow ghost
(198, 89, 179)		Pink ghost
(200, 72, 72)		Red ghost
(66, 114, 194)		Edible ghosts
(187, 187, 53)		Pacman lives
(184, 50, 50)		Cherry
(195, 144, 61)		Score number
(0, 28, 136)		Maze background
(0, 0, 0)		Scoreboard background

Figure 22: RGB color codes for colors used in Pacman.

Test ID	Size	FC Layers	CNN Layers	Average score (100k frames)	Max score (100k frames)	100k frames trained*	250k frames trained*	500k frames trained*	750k frames trained*	1m frames trained*	Overall Average score	Overall Max score
1	16x18	64,64	-	380	1440	465	930	1071	1164	1054	876	4060
2	16x18	128,128	-	388	1860	487	411	503	528	763	532	4080
3	16x18	256,256	-	413	2180	692	702	953	1068	1192	827	4070
4	16x18	512,512	-	422	2530	606	891	1066	1097	1085	925	5090
5	16x18	1024,1024	-	437	1910	647	1013	1067	1217	891	990	4720
6	22x20	16,16	-	353	1020	334	401	911	694	633	584	3570
7	22x20	32,32	-	418	2040	611	561	352	703	786	496	4070
8	22x20	64,64	-	413	1840	474	588	721	718	661	695	3030
9	22x20	128,128	-	413	3660	656	601	1064	738	1068	776	4150
10	22x20	256,256	-	392	1400	491	771	854	938	1156	759	5990
11	22x20	512,512	-	470	1950	673	874	1060	989	1237	937	4040
12	22x20	1024,1024	-	488	2450	798	995	976	1266	1165	993	3900
13	14x20	16,16	-	395	1330	452	458	629	468	749	546	3540
14	14x20	32,32	-	406	1780	419	504	639	888	718	605	2830
15	14x20	64,64	-	378	1600	421	880	711	908	1120	720	5240
16	14x20	128,128	-	411	1140	544	729	987	1360	915	840	3910
17	14x20	256,256	-	411	3460	517	737	1112	963	1193	781	4490
18	14x20	512,512	-	494	1910	756	858	1197	1394	1213	1069	5580
19	14x20	1024,1024	-	441	3970	708	953	1057	1493	1432	991	5020
20	14x20	256,256	-	500	1940	697	1019	-	-	-	812	4220

Figure 23: Manual neural network (Result IDs 1-20)

Test ID	Size	FC Layers	CNN Layers	Average score (100k frames)	Max score (100k frames)	100k frames trained*	250k frames trained*	500k frames trained*	750k frames trained*	1m frames trained*	Average score	Max score
21	14x20	512,512	-	498	2170	840	934	925	1187	1299	1039	4540
22	16x20	256	-	311	1900	357	430	321	278	286	338	3370
23	16x20	512	-	364	1580	283	408	410	360	349	334	3180
24	35x32	512,128,32	-	472	1920	614	618	579	596	829	665	4330
25	35x32	256,256	-	474	1910	709	933	1194	1194	1039	1001	4100
26	28x36	256,256	-	459	2340	774	595	1017	1067	957	846	4080
27	28x36	512,512	-	423	1430	562	474	731	878	885	719	4650
28	30x26	16,16	-	401	1440	453	593	460	484	616	456	2240
29	30x26	32,32	-	431	1980	656	500	496	727	942	566	4100
30	30x26	64,64	-	464	1660	741	729	1003	917	1007	946	4070
31	30x26	128,128	-	433	2210	581	991	1129	981	1378	994	4180
32	30x26	256,256	-	429	1140	543	581	816	1145	775	846	4620
33	30x26	512,512	-	434	1560	651	737	1156	1387	1309	973	5690
34	30x26	1024,1024	-	485	2190	745	692	1080	1104	852	945	4240
35	32x32	256,256	-	436	1910	571	512	1079	1191	1138	852	5490
36	32x32	512,512	-	441	1900	600	526	874	1409	1223	842	4030
37	35x32	264,64,16	-	397	1890	453	745	765	644	543	604	3500
38	35x32	256,256	-	701	4440	763	646	646	814	833	754	4440
39	35x32	256,256	-	421	1980	643	1112	921	1002	697	899	4480
40	35x32	512,512	-	522	2480	870	796	920	835	1089	857	5350

Figure 24: Manual neural network (Result IDs 21-40)

Test ID	Size	FC Layers	CNN Layers	Average score (100k frames)	Max score (100k frames)	100k frames trained*	250k frames trained*	500k frames trained*	750k frames trained*	1m frames trained*	Average score	Max score
41	16x20	256	16,32	535	2200	869	970	806	676	752	758	3720
42	16x20	512	32,64,64	531	2340	837	847	776	718	654	735	4100
43	16x20	256	16,32	518	2140	728	1184	976	797	783	870	4390
44	16x20	512	32,64,64	528	2170	976	1061	702	586	722	786	3930
45	16x20	512,512	-	401	1890	618	882	846	1109	896	850	4280
46	16x20	512,512	-	417	1480	724	1083	1283	957	904	988	4830
47	16x20	256,256	-	474	4620	834	906	798	1105	884	851	5080
48	16x20	512,512	-	497	2220	876	1016	730	960	734	871	4860
49	16x20	256,256	-	567	2740	975	904	880	956	830	877	3870
50	16x20	512,512	-	547	2600	1107	863	1032	1095	902	1018	6180
51	16x20	256	16,32	569	2460	1113	1294	1017	1079	852	959	5220
52	16x20	512	32,64,64	522	2520	857	1003	718	811	820	761	3170
53	16x20	16,16	-	444	2210	687	707	580	603	566	577	2230
54	16x20	16,16	-	340	1980	377	766	538	503	617	553	3460
55	16x20	512,512	-	784	3680	1010	999	1161	1285	1197	1164	4970
56	16x20	512,512	-	724	2680	986	1009	1154	1418	1275	1145	4260
57	16x20	1024,1024	-	720	2900	937	1247	1132	1231	1192	1155	4750
58	16x20	1024,1024	-	676	2240	888	1195	1167	1066	1257	1066	4950
59	16x20	1920,480,120	-	629	2560	703	798	865	694	718	770	3310
60	16x20	3840,1920,960	-	664	2320	942	1173	1023	1028	1083	986	4770

Figure 25: Manual neural network (Result IDs 41-60)

Test ID	Size	FC Layers	CNN Layers	Average score (100k frames)	Max score (100k frames)	100k frames trained*
61	20x24	645, 427, 473	109,68	568	2130	70
62	20x24	216, 371, 982		627	2200	478
63	20x24	555, 406, 249		569	2050	524
64	20x24	384, 261		669	2570	765
65	20x24	447, 433, 149		622	2170	681
66	20x24	873, 287		689	2240	657
67	20x24	882, 833, 998		649	2820	684
68	20x24	672, 332, 149		577	1900	502
69	20x24	554, 624	121, 15	637	2300	921
70	20x24	181	116	780	3790	834
71	20x24	841, 857, 434		743	3830	396
72	20x24	497, 295		666	2680	795
73	20x24	506, 970	107	779	2500	858
74	20x24	389, 380		718	2610	788
75	20x24	364, 523, 35		577	1410	70
76	20x24	431	39, 92, 108	642	2330	865
77	20x24	378, 941		718	2740	1,007
78	20x24	973, 203		709	1880	929
79	20x24	73, 281, 730	119,33,112	556	2020	345
80	20x24	949, 921, 450		637	2280	572
81	20x24	87, 853		687	2340	547
82	20x24	367, 100	23,37	625	3670	796
83	20x24	218, 817		713	3380	1,003
84	20x24	620, 889, 869		633	2090	692
85	20x24	946, 326		681	4080	751
86	20x24	152, 954	62,24,123	598	3160	489
87	20x24	896, 610	64,74,113	717	3990	531
88	20x24	521, 609, 700		681	1960	772
89	20x24	985, 566, 62		622	2290	240
90	20x24	896, 610, 780	74,113	580	2140	339

Figure 26: Genetic algorithm neural network (Result IDs 61-90)

Test ID	Size	FC Layers	CNN Layers	Average score (100k frames)	Max score (100k frames)	100k frames trained*
91	20x24	32, 776, 748		600	2070	495
92	20x24	837, 563, 692	107,104	625	2320	547
93	20x24	353, 573, 383	9,185	612	2570	802
94	20x24	353, 573	9	683	2520	951
95	20x24	373, 272, 464	23,53	626	2140	140
96	20x24	884, 287		667	2200	911
97	20x24	946, 326, 275		584	2780	747
98	20x24	326, 926		681	2330	909
99	20x24	282, 719		733	2700	883
100	20x24	32	53	658	2220	484
101	20x24	206, 1016, 964		697	2400	812
102	20x24	674, 95, 849		641	2350	637
103	20x24	651	17	729	3750	595
104	20x24	180, 269, 339	59,97	560	2680	70
105	20x24	330, 843, 607		623	1074	655
106	20x24	167, 129		682	3520	464
107	20x24	582, 115		621	2020	575
108	20x24	504, 609, 700		640	2260	857
109	20x24	168	121	718	2910	947
110	20x24	1021, 636, 676	64,65	561	2040	316
111	20x24	985, 699, 516	73,43	571	2230	651
112	20x24	170, 129		597	1590	966
113	20x24	95, 521		660	2200	655
114	20x24	784, 942	115	780	2480	858
115	20x24	800, 871, 22		637	2500	70
116	20x24	857, 284, 319		674	2810	475
117	20x24	477, 902, 199	15,26	578	2200	70
118	20x24	123, 296	48,105,49	583	3940	352
119	20x24	117, 1014		728	2490	979
120	20x24	383, 113		631	3660	758

Figure 27: Genetic algorithm neural network (Result IDs 91-120)

Test ID	Size	FC Layers	CNN Layers	Average score (100k frames)	Max score (100k frames)	100k frames trained*
121	20x24	493, 276, 612		567	2330	895
122	20x24	643, 452	39	711	3540	987
123	20x24	560, 257, 454		602	1640	702
124	20x24	678, 619, 938	12,30	610	2780	372
125	20x24	116	63,78,29	627	2150	579
126	20x24	873, 287, 575		625	1152	727
127	20x24	176	142	-	-	1,135
128	20x24	890, 467		651	3840	969
129	20x24	134	15, 32, 73	602	2370	858
130	20x24	739, 30, 319	11	526	1960	537
131	20x24	867	123	750	2500	1,270
132	20x24	547, 393, 144	54,118	655	3540	70
133	20x24	296, 643		614	1088	818
134	20x24	886	123	-	-	1,270
135	20x24	461	87, 31	726	2380	885
136	20x24	330, 894		756	2670	870
137	20x24	377, 697, 119		653	2340	462
138	20x24	854, 45		630	2520	850
139	20x24	866, 585	59,89,28	610	2400	257
140	20x24	613, 962	55,91,20	626	3390	418
141	20x24	168	130	722	2800	1,135
142	20x24	181	119, 888	780	3950	941
143	20x24	928, 604, 29	109,41	574	3510	242
144	20x24	884, 287, 676		605	1650	619
145	20x24	869, 250		773	3190	1,033
146	20x24	111, 122, 88	12	540	2260	230
147	20x24	155, 619	51	753	2280	1,285
148	20x24	42, 625, 817		573	1820	294
149	20x24	713, 256, 835		600	2040	926
150	20x24	174, 284, 430		648	2690	414

Figure 28: Genetic algorithm neural network (Result IDs 121-150)

Test ID	Size	FC Layers	CNN Layers	Average score (100k frames)	Max score (100k frames)	100k frames trained*
151	20x24	894, 454		691	3530	1,036
152	20x24	214, 349	17	610	1136	803
153	20x24	201, 233		604	2090	1,031
154	20x24	106, 684		603	2200	924
155	20x24	231, 154, 909		635	2320	302
156	20x24	494, 768		780	3130	813
157	20x24	919, 433	56, 59	675	2330	726
158	20x24	366	55, 126	810	4120	1,319
159	20x24	619, 904		713	2420	1,219
160	20x24	201, 968, 77		599	2250	878
161	20x24	223, 358		660	2910	838
162	20x24	510, 268		679	2440	855
163	20x24	913	66, 10	646	2420	945
164	20x24	316	20	769	2310	1,217
165	20x24	547, 499, 525		647	2620	724
166	20x24	625, 426, 464	86	683	2280	942
167	20x24	378	82	737	2680	1,071
168	20x24	234, 446		627	1056	1,177
169	20x24	701, 943		751	3730	1,177
170	20x24	475, 821, 485		651	4650	907
171	20x24	855, 472		673	2490	981
172	20x24	867	131	783	3690	1,664
173	20x24	181, 1010	116, 888	778	3850	1,522
174	20x24	181, 1010, 899	116	672	1830	1,522
175	20x24	181	118, 888	791	4020	1,522
176	20x24	168	129	751	2560	1,395
177	20x24	442	55	735	2910	1,035
178	20x24	1010, 899	127	689	3720	1,064
179	20x24	867, 443	128	725	2570	1,277
180	20x24	313	55	747	2580	1,267

Figure 29: Genetic algorithm neural network (Result IDs 151-180)