Genetic training of neural networks in Flappy Bird

Rasmus Hogslätt[1]

Abstract
This report delves into the application of Genetic Algorithms, GAs, for training Neural Networks, NNs, in a dynamic environment, exemplified by the game Flappy Bird, where traditional gradient-based training methods are not viable. The exploration is aimed at understanding how GAs can optimize NNs and how a few different parameters influence the performance of this approach. The implementation is carried out in Rust, utilizing the Bevy game engine for game creation, and devises a structured approach to evolving NNs, with the population of birds being evaluated, selected, crossed over, and mutated across generations to improve gameplay performance. Two methods of generating new populations, named *average* and *thirds*, are introduced alongside a decreasing mutation rate strategy to balance exploration and exploitation over time. The analysis of varying mutation rates and neural network architectures under both generation methods reveals that the 'thirds' method, especially with a mutation rate of 0.125 and a [4, 3, 3, 2, 1] architecture, achieves superior performance, hinting at a promising trade-off between exploration and exploitation and a well-suited architecture complexity. Conversely, the 'average' method exhibited lower average performance, indicating potential convergence issues or the risk of being trapped in local minima. While the current implementation demonstrates encouraging results, further exploration of other parameters such as population size, more nuanced generation methods and additional environment inputs is suggested as further exploration that could potentially yield improvements in performance when training neural networks using genetic algorithms.

Source code: https://github.com/RasmusHogslatt/GeneticFlappyBird

Authors
[1] *Media Technology Student at Linköping University, rasho692@student.liu.se*

Keywords: Neural Network — Genetic Algorithm — Rust

Contents

1. Introduction

Neural networks have become an important part of today's everyday life and is increasingly encountered in various types of applications. In order for neural networks to be useful, they need to be trained. This is typically done by utilizing gradient descent and back-propagation, requiring knowledge of the cost function's gradient[Kostadinov(2019.)]. However, in dynamic and non-deterministic environments, for example the game Flappy bird, outcomes of an action are not always known. Thus, the gradient function is hard or difficult to obtain, meaning that backpropagation is not always trivial to implement. Reinforcement learning methods, such as genetic algorithms, are often employed rather than backpropagation methods. This is further discussed in the section *Advantages of Reinforcement learning* by Bajaj in his article *What is reinforcement learning*[Bajaj(2023.)]. This report aims to implement and explore how genetic algorithms can be utilized to train neural networks, and also explore how various parameters can impact the performance of such implementation. Further, the implementation will be done for the game Flappy bird.

2. Theory

2.1 Neural Networks

Neural Networks are computing systems inspired by the human brain's interconnected neuron structure. They comprise layers of nodes or "neurons," with each node in a layer connected to all nodes in the preceding and succeeding layers. These connections have associated weights, which are adjusted during training to minimize the error between the network's output and the desired output.

- **Architecture:**
  - **Input Layer:** Receives the raw data and passes it onto the subsequent layers.
  - **Hidden Layers:** Intermediate layers where the data is processed through weighted connections and activation functions.
  - **Output Layer:** Delivers the final output of the network.
- **Training:**
  - **Forward Propagation:** Input data is passed through the network, layer by layer, until it reaches the output layer.
  - **Error Calculation:** The error is calculated by comparing the network's output with the desired output.
  - **Backpropagation:** The error is then propagated back through the network, and the weights are adjusted to minimize the error.
  - **Epochs:** This process is repeated for a specified number of iterations or until a desired level of accuracy is achieved.
- **Activation Functions:** Functions like the sigmoid or rectified linear unit (ReLU) that introduce non-linear properties to the system, enabling the network to learn from the error.

2.2 Genetic Learning for Neural Networks

Genetic Algorithms, GAs, are optimization algorithms based on the process of natural selection and where proposed in 1989 by a group of researchers at Stanford and California University of Technology[Miller and Hegde.(1989.)]. They can be used to train neural networks, particularly when the problem domain is complex, the data is noisy, or the error landscape is non-convex and full of local minima. In this case, they can be used to replace the backward propagation in traditional training of neural networks.

- **Initialization:** A population of neural networks is initialized with random weights.
- **Fitness Evaluation:** Each network's fitness is evaluated based on a predefined criterion, often the performance on a task.
- **Selection:** Networks are selected for reproduction (crossover and mutation) based on their fitness, with fitter networks having a higher chance of being selected.
- **Crossover:** Pairs of networks are combined to create offspring networks, inheriting characteristics (weights) from both parent networks.
- **Mutation:** Some weights in the offspring networks are randomly altered to introduce variability.
- **New Generation:** The offspring networks form a new generation, and the process is repeated until a stopping criterion is met.

Genetic learning can be particularly useful when traditional training methods like gradient descent are infeasible or ineffective. By evolving networks over several generations, GAs facilitate exploration of wide search spaces which can lead to the discovery optimal weight configurations. They are also able to run in parallel, allowing for faster training than training one agent at a time. Moreover, GAs do not require the cost function to be differentiable, a requisite for methods like backpropagation, making them applicable to a broader range of problems.

3. Method

The implementation of the project was done using the Rust programming language.

3.1 Making a neural network

As described in section 2.1, neural networks consist of layers, with some amount of nodes, each consisting of weights and biases. Therefore, the structure of the neural network was achieved by creating a vector of layers, each layer holding a vector of biases and a matrix of weights.

Each layer L_i in the neural network is represented as

$$L_i = \{W_i, b_i\} \tag{1}$$

where,

- W_i is the weights matrix for layer L_i, with each element W_{ij} representing the weight from the jth neuron in layer L_{i-1} to the ith neuron in layer L_i.
- b_i is the bias vector for layer L_i, with each element b_{ik} representing the bias of the kth neuron in layer L_i.

Further, a forward function was implemented according to the equation

$$Z_i = W_i A_{i-1} + b_i$$
$$A_i = g(Z_i)$$

where,

- Z_i is the pre-activation value at layer L_i.
- A_i is the post-activation value at layer L_i, obtained by applying the activation function $g(\cdot)$ to Z_i. The activation function implemented was the sigmoid function.
- A_0 is the input to the neural network.

This structure and forward propagation function facilitates the training of the neural network after incorporating the genetic algorithm, enabling the optimization of weights and biases to minimize the loss function and make accurate predictions on unseen data.

3.2 Making a game

Making the game was done using the *Bevy* game engine in Rust. It uses an entity-component-system model, ECS. A bird was defined by a body, in this case a red square, and a neural network. The bird was affected by gravity and its only action was to ”flap” its wings, represented by an upwards force.

Pipes, the obstacles in the game, were made from two squares with a gap between and moving at fixed velocity from right to left of the screen.

Further, a system was implemented that continuously fed each bird information about its state in the environment. This became the input to the neural network and were given by

1. Horizontal distance to nearest pipe
2. Vertical distance to the center of the gap of nearest pipe
3. The bird's vertical position
4. The bird's vertical velocity

which meant that the neural network architecture had to have 4 input neurons and one output neuron.

3.3 Fitness

Each bird was also given a fitness score, which was calculated as the time the bird had been alive. Each bird also had a score, increased by 1 for each pipe it managed to pass. However, the score was merely for ease of visualizing performance, and not to determine the fitness of the birds, despite their strong correlation.

3.4 Selection

Each generation assumed a population size of at least two birds. A pair of birds, *best birds*, were kept around to keep track of the best performing birds out of all generations. The last two birds of each generation were those that were the fittest for the given generation. The fitness of these birds were checked against the *best birds* fitness. If the fitness of the current generation exceeded that of the fitness values in *best birds*, the neural networks and current fitness values were used to update *best birds*.

3.5 Crossover

The crossover algorithm, called when spawning a new population, was implemented as the average weights and biases of the two fittest birds' neural networks.

3.6 Mutation

The mutation of a bird was reliant on two parameters, *probability* and *rate*. The *propability* was the propability that the bird should mutate or not, whereas the *rate* was used to generate a random value in range $[-rate, rate]$ which was added to the weights and biases.

Initially, the *rate* was set to 1.0, producing large mutations. Every 10th generation, it was multiplied by 0.9, resulting in a decreasing rate over time. Thus, exploration was favored initially and exploitation was more favored in later parts of the training.

3.7 New generation methods

Two algorithms for generating a new population were implemented. These were the *average* and *thirds* methods.

The *average* method simply averaged the weights and biases of the two best neural networks for each bird in the new population. Essentially using the crossover method previously described. This was followed by applying the mutation schema to each bird.

The *thirds* method instead divided the new population into thirds. One third was cloned from the best bird and another third from the second best bird. These birds were then mutated according the the mutation schema described. The last third was generated entirely randomly.

4. Result

Figure 1 depicts the game during training of a population of 400 birds.

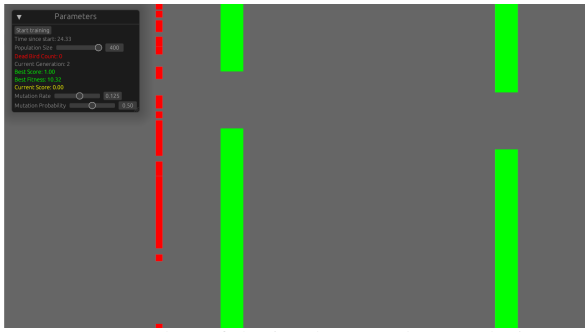

Figure 1. Screenshot of the implemented game during training.

Further, for a population of 50 birds, eight tests were ran with varying mutation probabilities and neural network architectures. The tests ran until no further significant improvement was seen. This resulted in the tests running for approximately 4 hours each. The score of each trained generation and model is shown in Figure 2 and a summary of the models' performances is seen in table 1.

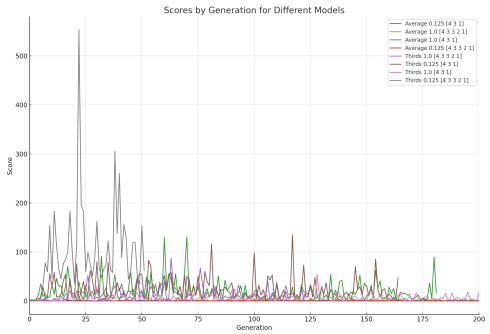

Figure 2. Scores over several generation with varying parameters.

Table 1. Key metrics of performance of various parameter configurations.

Method	Probability	NN Architecture	Max Score
Average	0.125	[4, 3, 1]	8
Average	0.125	[4, 3, 3, 2, 1]	47
Average	1.0	[4, 3, 3, 2, 1]	5
Average	1.0	[4, 3, 1]	131
Thirds	0.125	[4, 3, 1]	135
Thirds	0.125	[4, 3, 3, 2, 1]	553
Thirds	1.0	[4, 3, 3, 2, 1]	88
Thirds	1.0	[4, 3, 1]	54

5. Discussion

The training of neural networks using genetic algorithms to play the Flappy Bird game yielded varying results, dependent on the mutation rate, method of generating new generations and neural network architecture.

The *thirds* method, especially with a mutation probability of 0.125 and [4, 3, 3, 2, 1] architecture, achieved the most impressive performance with a score of 553. This suggests a balanced exploration and exploitation trade-off and an architecture complexity that was rather well suited to capture the necessary patterns for playing the game well. The decreasing mutation rate over generations contributed to the stabilization of learning, initially promoting exploration and later focusing on fine-tuning the promising candidates, commonly referred to as *exploitation*.

Contrarily, the *average* method, particularly with a 1.0 mutation probability and [4, 3, 1] architecture, exhibited a lower average performance, indicating potential issues with convergence or the possibility of being trapped in local minima. This was especially noticeable in cases where no birds would survive past the first pipe. This was likely due to no the fact that the average of two unfit birds typically result in another unfit bird. This could possibly be resolved by only starting applying the *average* crossover method if the fittest bird passed the first pipe.

Further, higher mutation rates yielded more exploration, minimizing the risk of getting stuck in local minima. They could, however, introduce an unnecessary amount of exploration, causing the algorithm to take longer to achieve good results. Thus, a trade-off between exploration and exploitation is to be made. The current approach of taking 90% of previous mutation rate every 10 generations seemed to achieve good results, but could perhaps be further tuned to achieve faster convergence to a good bird, whilst also avoiding local minima.

6. Conclusion

The implementation shows that genetic algorithms can be a viable method of training neural networks for tasks like game playing, where traditional training methods might face challenges. In particular, they proved to be useful for dynamic environments, where gradient functions are unknown.

Further, the choice of parameters and how each new population is generated was shown to greatly affect the time it takes to train a well performing network. Many other parameters could have been explored further, such as the population size, neural network architecture, more nuanced generation methods, other environment inputs etc.

References

[Kostadinov(2019.)] Simeon Kostadinov. Understanding backpropagation algorithm. *Towards Data Science*, 2019.

[Bajaj(2023.)] Prateek Bajaj. What is reinforcement learning, 2023. URL `https://www.geeksforgeeks.org/what-is-reinforcement-learning/`.

[Miller and Hegde.(1989.)] Peter M. Todd Miller, Geoffrey F. and Shailesh U. Hegde. Designing neural networks using genetic algorithms. *ICGA. Vol. 89.*, 1989.