university of groningen / faculty of science and engineering

Neural Networks for AI

Semester Project

July 2021

Solving the Lunar-Landing Problem: Neuroevolution vs. Backpropagation

Team 22

Stefania Radu (s3919609)

Satchit Chatterji (s3889807)

Ruhi Mahadeshwar (s4014456)

Andreea Minculescu (s3932222)

Abstract

Landing a spacecraft autonomously is a fundamental task in space exploration: it has to be precisely executed or it could lead to tremendous loss in terms of resources and human lives. As a result, computer precision has proved to be a strong candidate in solving the lunar landing task. In this present paper, we explore two different multilayer perceptron (MLP) architectures in a simplified 2-dimensional rocket control simulation with the final goal of landing the rocket as close as possible to a randomly generated target. The architectures have the same overall structure but differ in 1) the learning algorithm, one being evolutionary-based and the other using the error backpropagation method and 2) the kind of training, one being trained by playing games against itself and the other on human generated data. Although both architectures solve the lunar landing task, the evolutionary-based MLP is more robust in terms of the solution(s) found, while the backpropagation-based MLP reaches a good solution after a lower number of iterations.

CONTENTS

I. Introduction

The rocket landing problem has become increasingly popular over the past few years, especially after the successful landings of SpaceX boosters on the surface of Earth and their equally successful media coverage. Being able to train a spacecraft to land autonomously is a fundamental issue if one wants to explore the solar system and then return safely to Earth. Thus we will also refer to this as the 'lunar landing problem'. The goal can be expressed in simple words: find an optimal landing sequence from your current position to the target before running out of fuel. However, the computational solutions are not trivial at all. For example, in 2010, the MIT engineer Lars Blackmore proposed a solution to this challenge, by treating it as a "convex optimization problem" and using mathematical tools, chose the best path from a geometric shape, containing all possible landing scenarios. (Blackmore, Açikmeşe, & Scharf, 2010). In a real-world scenario, a reliable solution to this problem is both important and incredibly difficult to find. Nevertheless, our goal for this project is not so bold.

A. *Problem definition*

We focused our attention on a simpler 2D simulation written in Python, where a rocket is taught to land on a flat surface at a specified landing location. Previous work has been conducted to solve landing problems of this type.

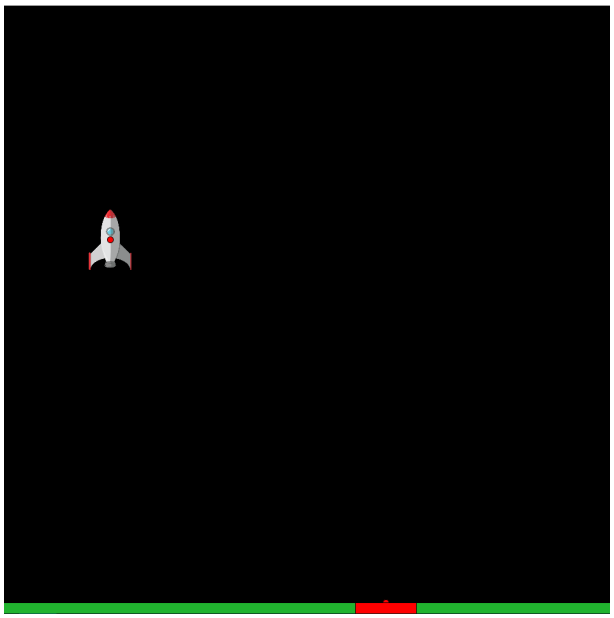


Figure 1: A visual representation of the lunar landing game. For more information, see text.

Reinforcement learning (RL) is a popular approach for this. Gadgil, Xin, and Xu (2020) compared two different RL techniques: Sarsa and Deep QLearning, while training a spacecraft to land after introducing different amounts of uncertainty to the environment. Other techniques make use of a control-model-based approach which learns the model parameters (Peters, Stewart, West, & Esfandiari, 2019), or guided genetic algorithm with extra RL components (Liu, 2006).

Echo state networks (ESN) (Jaeger, 2001) can also be used to solve this kind of autonomous control problems. These networks are based on a reservoir computing architecture, which use a recurrent neural network that is excited by the presentation of inputs and outputs. In the case of ESNs, only the weights connecting the reservoir and the output neurons are trained, which makes the overall training process more efficient. Tilburg (2021) developed a model using ESNs, which learns how to fly an aircraft (of a more sizeable complexity than our own simulation).

The current project analyzes two training methods for artifical neural networks to play this lunar landing game, one using a backpropagation-based paradigm trained on human data, and the other being a neuroevolution-based technique, which uses evolutionary algorithms to find optimal neural network parameters.

B. Simulation

The simulation used in this study is a 1000x1000 pixel 2D scene. Here, our aim is to train a neural network to control a rocket to land safely on a randomly initialized target before the rocket runs out of fuel. To keep things simple, we assume that the fuel is consumed by the rocket at a constant rate when the engine is on. Figure 1 is a visualization of the lunar landing game. The frame has four components: the sky, indicated by a black background, the ground, which is green, the target, highlighted in red, and the rocket itself[1]. The red dots on the rocket and the target are the corresponding centers of gravity.

When the simulation is launched, the rocket and the landing target are both initialised randomly, the first in the sky and the latter on the ground. There are several factors to consider when landing the rocket on the target – notably, one must control the rocket engine (i.e. the thrust) and the direction in which the rocket is flying. These factors are controlled using keys on the keyboard and each key influences the rocket's behavior in some way. The specification of these controls is as follows: 'w' - engine on, ' ' (space bar) - engine off, 's' - stop rotation, 'a' - turn left, 'd' - turn right, 'r' - reset game. All other key inputs are ignored. The state of the rocket at each frame is determined by Eq. 1.

$$state \rightarrow \begin{cases} \text{rocket position on x} \\ \text{rocket position on y} \\ \text{velocity on x} \\ \text{velocity on y} \\ \text{thrust on x} \\ \text{thrust on y} \\ \text{rotation} \\ \text{engine} \\ \text{target position x} \\ \text{target position y} \\ \text{fuel} \end{cases} \tag{1}$$

Landing a rocket safely implies that the rocket does not crash. We call this a *successful* landing. To determine if the landing was successful or not, we define a score. This score is determined by factors such as the distance between the rocket and the center of the target, the velocity of the rocket when it is landing, the relative rotation of the rocket and the fuel left after landing the rocket. We want this score to be as low as possible, that is the distance between the rocket and target is minimal, the landing velocity of the rocket should be minimal, the rocket should be upright and the rocket should not use more fuel than necessary. Based on this score, we labeled the games as either successful or failed and used them as training data for the backpropagation-based MLP, presented in the following sections.

Worth mentioning here is that, while the initial goal of the project was to optimize all four features (i.e. distance from target, rotation, velocity, fuel consumption) we soon understood this to be too involved of a task. Therefore, in the remainder of the report we will focus on minimizing just the landing distance to target. Note that we did not change the conditions of what would be considered a "successful" landing in the case of the human training data because we want this data to model a landing as close to "ideal" as possible.

[1]The rocket sprite is downloaded from `www.kindpng.com`, under license.

The code for this simulation, as well as for both training methods, can be found at `https://github.com/satchitchatterji/NeuralNetworksProject`. Instructions can be found in each file and the readme.md file. A recording of a run can be found at `https://youtu.be/zxtcC-INAH0`. We would recommend playing the simulation yourself – it is quite addictive!

II. Data

In this section we will describe the training data used to train the backpropagation-based models, the gathering process and preprocessing techniques.

The backpropagation-based model was trained with data gathered from human players. We managed to gather a total of 412 plays, of which 154 were successful landings. A successful landing implies that the rocket lands close to the target (within 50 pixels either side), its final speed is no more than 2 pixels/frame, and its relative rotation is less than 15 degrees.

Each play was recorded in a csv file, which, on the first line indicates whether the game was successful or not. After an informative header, the file contains a list of states as shown in Eq. 1, recorded at each time frame, each of them followed by a label. The label represents the key pressed by the user. The total number of labels is 6 and it matches the number of possible keys: ‘w’, ‘a’, ‘s’, ‘d’, ‘ ’, ‘p’. The ‘p’ key is included to represent the ”do nothing” action here, and is ignored by the simulation, and in case the player accidentally presses an invalid key. It is worth mentioning that once one key is pressed, all the following simulated states will use the same key as a command, unless a different key is pressed. Each file contains a different number of rows, depending on the length of the game. The initial position of the rocket and the target position were randomly generated for each game. The game ends when the rocket impacts the ground.

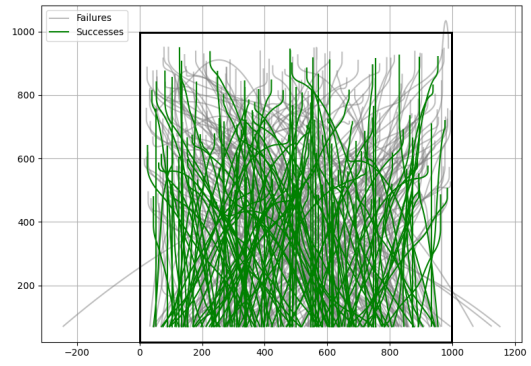


Figure 2: A visual representation of the paths from the training set, where the green lines are successful games and the grey ones are failures. The 800x800 black box represents what was visible to the player. The starting position and targets are randomised uniformly.

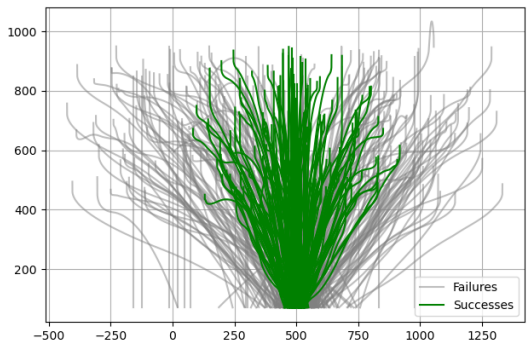


Figure 3: A visual representation of the paths from the training set, where the green lines are successful games and the grey ones are failures. The paths are repositioned such that the target is placed in the center (at the point $(550, 0)$).

The saved flight paths, when plotted, result in Figure 2, where the green lines are the successful games and the grey lines are the unsuccessful ones. We see that the initial configurations are uniformly distributed and the paths fill the screen. We cannot determine very much from this graph about the behaviour of human game play. However, Figure 3 is another visual representation of the paths taken by the rockets from the training set. All the paths are translated in the $x - dimension$ such that the target is placed in the center of the screen ($x = 500$). It can be noticed in the plot

that, while not all rockets manage to reach the goal position, the behavior is consistent overall, as all the rockets are trying to approach the target. We also see that humans tend to land the rocket successfully when the initial position is not too far away laterally from the target.

For training, we decided to use successful games only, so that the network learns to generate a sequence of keys, which will create a valid landing path to the target. This means the dataset we used contained 73079 data points (each game state - move pair, recorded at each time frame for all games). Seeing as the input features had different ranges, we normalized the data, such that features of higher degree would not necessarily be considered more important. For that, we used the *MinMaxScaler*[2] class from the *sklearn* library (Pedregosa et al., 2011), which scales the data to a given range (in this case, $[0, 1]$). Interestingly enough, scaling the input resulted in approximately the same performance level and overall behaviour as not conducting any scaling at all.

III. Methods

In this section we will discuss the main network architecture used, and two methods employed for training it. We have divided this section into three subsections, the first laying out the network, and the other two each discussing a training method.

A. Architecture

For the first approach, we treated the lunar landing game as a multi-label classification problem and trained it using the human dataset (see Section II). The main idea behind the classifier is as follows:

Given: a game state (i.e. rocket position, velocity, rotation, fuel, target position) at time t.

Wanted: a function that takes the game state as input parameters and outputs the next move.

Output: one of the 6 control keys

The neural network class we examined for this task is the multilayer perceptron (MLP), a relatively simple feedforward model. In terms of the architecture of the network, we used one input layer of 11 neurons, one output layer of 6 neurons and one hidden layer with 11 neurons (Figure 4). We also experimented with more hidden layers, but this added complexity made the training slow and did not significantly improve the performance. The network is fully connected. Though different activation functions were experimented with, for the sake of comparison, we decided that the hidden layer should have a rectified linear unit (ReLU) activation function, defined at Eq. 2. Empirically, this did fairly well for both training paradigms.

$$ReLU(x) = \begin{cases} 0, x < 0 \\ x, x \geq 0 \end{cases} \tag{2}$$

B. Backpropagation

Conventionally, MLPs are supervised learning models that generate input-output mappings such that the initial input-output underlying trends of the training data are preserved (Goodfellow, Bengio, & Courville, 2016). Hence, it is easy to see why such an architecture would be suitable for our

[2]Documentation is at `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html`

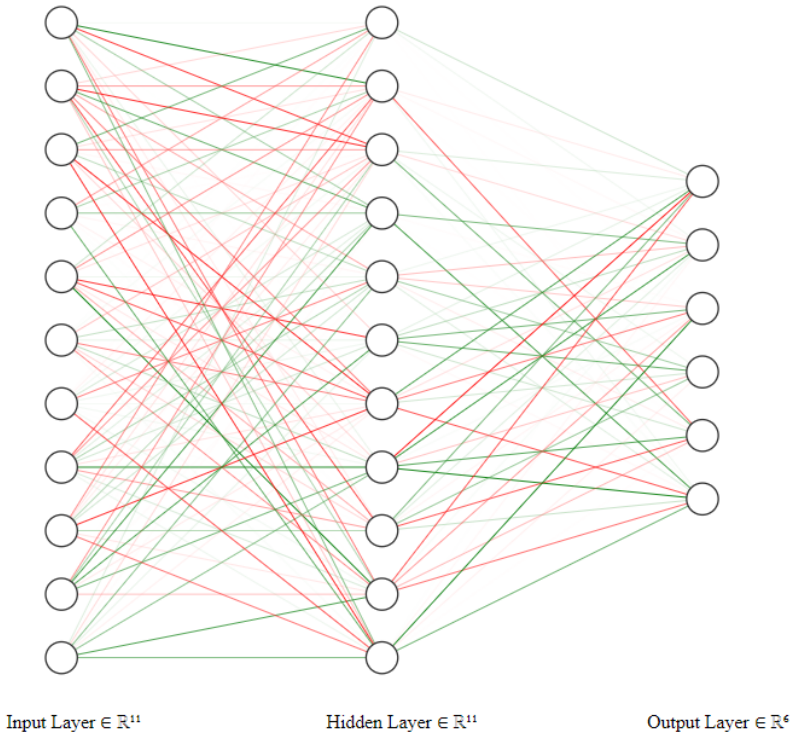


Figure 4: A schematic representation of the network used for the current task.

task: it should, in theory, be able to reproduce the strategy humans use in driving the rocket towards the target.

Thus, the first training method we discuss is backpropagation, with the network in Figure 4 being trained in the following way: it iteratively adjusts trainable weights such that some measure of loss is minimized (Jaeger, 2021). The minimization problem is solved, in this case specifically but not only, using stochastic gradient descent (sgd). This method produces a series of models with decreasing empirical risk values, following these rules: model $n + 1$ is generated by applying slight modifications to model n and the starting model is generated randomly. In this way, in theory, the series of models should converge towards a model with an empirical risk as close to the possible minimum empirical risk as feasible, within a finite number of iterations. We call this final model "optimal" and finding it is the goal of the backpropagation algorithm.

Backpropagation is applicable in a supervised learning context with a suitable training dataset used to compute the loss measurement so we now return to the problem at hand. We initially designed two backpropagation-based MLPs using two different libraries, `sklearn` (Pedregosa et al., 2011) and `tflearn` (Damien, 2016) with the same architecture, since we were interested in whether the two networks would solve the same task using different strategies. As it turns out they displayed similar overall behaviour and for the sake of brevity, we will only discuss the tflearn-based model from here on.

The single hidden layer in the network architecture is associated with a dropout core layer, with a dropout probability of 0.8, used to avoid overfitting. This just means that at each training step, there is a probability for each neuron in the layer to be set to zero. The output layer has a softmax activation function. A regression estimator layer performs the back propagation algorithm: the "adam" optimizer minimizes the "categorical_crossentropy" loss. Biases were used and initialized

to zero. For an overview and a more detailed explanation of the hyperparameters see the remainder of the subsection.

$$Hyperparameters = \begin{cases} \text{activation = relu/softmax,} \\ \text{optimizer = adam,} \\ \text{learning\_rate = 0.001,} \\ \text{loss = categorical\_crossentropy,} \\ \text{bias = True,} \\ \text{n\_epoch = 15,} \\ \text{batch\_size = 128} \\ \text{shuffle = True} \\ \text{validation\_set = 0.1} \end{cases} \tag{3}$$

a. **Activation**: the activation function of the layers regulates how the weighted sum of the input of a neuron should be transformed into an output value. If the output value is higher than some threshold, the neuron is activated. In this case, the ReLU activation function gave the best result for hidden layer(s): it adds non-linearity to the neural network and, due to its relatively simple form, its computation time is low. In the case of the output layer, the softmax function was used because it turns a series of arbitrary n real-valued numbers into a series of n real-valued numbers that sum up to 1. Thus, it computes the probability of the input to belong to each of the six possible output labels.
b. **Optimizer**: algorithm used to update the internal parameters of the neural network (i.e. weights, biases) such that the loss function is minimized. Here, the sgd-based "adam" optimizer was used, due to its efficient optimization technique. In simple words, *adam* uses adaptive learning rates and momentum to converge faster and therefore, it is suitable for large datasets.
c. **Learning_rate**: global control parameter of the error backpropagation algorithm that regulates the step size of the optimization algorithm at each iteration in the pursuit towards the minimum of the loss function. This parameter implies a trade-off between stability of the error backpropagation algorithm and convergence speed towards a local minimum. It was simply due to experimentation that we settled for the 0.001 value, in the present case.
d. **Loss**: a number indicating how bad a prediction made by the algorithm is. The goal is to identify a set of weights and biases with an average low loss value, across all training examples. The loss function used here is categorical crossentropy, a standard loss function used in multi-class classification. It computes the difference between two probability distributions (i.e. training dataset vs prediction), using formula Eq. 4,

$$Loss = -\sum_{i=1}^{output\_size} y_i \cdot log(\hat{y}_i) \tag{4}$$

where y represents the actual output and $\hat{y}$ is the predicted value. This loss measurement is most often used in combination with the softmax activation function, since it transforms real values into probabilities.
e. **Bias**: an intercept that shifts the activation function such that it generalizes better to the training data.
f **N_epoch**: the number of epochs. An epoch represents training the neural network with (all) training examples for one iteration. In this case, training ends after 15 epochs. We selected this

number in order to avoid overfitting and underfitting, under the consideration of the relatively small number of trainable weights.

g **Batch_size**: the number of training examples used during one iteration, during training. An epoch is generally made up of one or more batches.

h **Validation_set**: the percentage of the data to be used for validation. The validation set is used to tune the hyperparamters of the neural network.

C. Neuroevolution

To motivate our choice for the second training method, we need to delve a little bit more into the nature of the task. As we can discern from the *Data* section (section II), gathering training samples by recording games played by humans is a laborious task that eventually leads to a modest amount of data. We also noticed that the human players' successful landing attempts have very similar flight paths.

Additionally, for our task, training data cannot be simply generated or simulated using a function either (for example by adding noise to an ideal flight path generated by such a function), since none are known to us. To analytically derive one is also incredibly hard, given the complexity of the feature landscape. We assume there exists a function of our input variables that can generate a suitable decision at each game state such that it lands the rocket within certain constraints. There indeed may be several suitable functions, causing different behaviours that result in a variety of flight paths, but all adequately being able to land the rocket. Training paradigms that explore a wide range of candidate functions may stumble or settle upon a function approximator that is suitable for the task.

Thus, a training method that does not need teaching data is a viable alternative to a supervised method such as backpropagation to find a set of neural network parameters that can land a rocket. This is beneficial both in terms of needing no data, which is tiresome to gather, and to potentially find interesting solutions in the potentially large pool of solutions to this task, different from a human's approach. Evolutionary algorithms fit this description perfectly.

Evolutionary algorithms are a set of general methods that are used to solve optimisation tasks that are inspired by Darwinian evolution (Stanley, Clune, Lehman, & Miikkulainen, 2019). These algorithms are general purpose, having been used in biology, economics, mathematics, music and art generation, and other disparate fields (Romero, Romero, & Machado, 2008; Hui, Landi, Minoarivelo, & Ramanantoanina, 2018; Mitchell, 1998). Much work on this has been done in the past in neural network research, being succinctly known as neuroevolution (Yao, 1993). Ronald and Schoenauer (1994) attempted to train a neural network on a task similar to ours, trying to land a rocket on the surface of the moon, albeit with a much simpler set of inputs and output controls, while looking for optimal parameter values for a set of fixed networks. A more recent algorithm, called *neural evolution of augmented topologies* attempts to find a suitable neural network architecture in addition to changing just the trainable parameters (Stanley & Miikkulainen, 2002). A more atomistic approach was designed to generate a population of neural fragments or individual neurons instead of complete networks (Gomez, Schmidhuber, Miikkulainen, & Mitchell, 2008). In our project, we used a relatively simple genetic algorithm (GA) approach to search through the state space of parameters (weights and biases) of a fixed MLP network topology (specifically, that of Figure 4).

The code was written from scratch, only using the numerical libraries `numpy` (Harris et al., 2020) and `scipy` (Virtanen et al., 2020) for random sampling and common statistical formulae. The MLP was also programmed from scratch, and only includes a forward pass. There are many customisable and arbitrary hyperparameters with respect to the GA itself. These, as well as the outline of the training process, are presented below.

- **The overall process:** A set with cardinality *n* of initial networks are created (called a *population*). They then are used to control *n* rockets in the simulation for *t* trials. Each rocket is assigned a score based on how close it gets to landing successfully. The networks that performed the best (had the highest *fitness*) have a higher chance of reproducing into the next generation. Once parents from the current generation are chosen on the basis of fitness, they reproduce to create children, whose parameters (the *genes*) are then slightly mutated. The same process occurs for the new population, and the network population gets better at landing the rocket over time.
- **Initial population:** A population of *n* MLPs of the architecture described are initialised with random weights (uniformly in the range $[0, 1)$) and biases set to zero. The random initial population ensures we have a wide selection over the state space of possible trainable parameters. Initial testing showed that networks even without biases performed fairly well for simpler tasks (e.g. tasks with initial constant target positions and rocket starting points), thus, we initialised all biases to be zero, but mutable, in order for 'evolution' to decide[3] whether or not to include them. Over time, we observed that biases were included in well-performing parameter sets.
- **Score/Fitness:** On the basis of the rocket's final state (when it collides with the ground), a *score* is computed on the basis of its final game state – specifically with respect to its distance to the target. Given this constraint, the task of the GA is to minimise this score – the ideal score is zero. The initial game configuration (starting position and target) are the same for all rockets during each trial, but are changed over the course of *t* trials to maintain a variety of starting configurations. The rocket's score is then defined as the average score over all trials. Initially, the target and starting position were both randomised, but the GA never found an algorithm that could land the rocket effectively. Thus, to simplify the task, the starting positions for all trails were constant – high in the air in the centre of the screen – and the target positions were varied over trials.

  A real, non-negative valued fitness is calculated on the basis of the score for each network, something akin to to a loss function in backpropagation. Since the selection method selects individuals with a high fitness, an arbitrary function was chosen to calculate a value inversely proportional to the score. In this case, we used a Gaussian probability density function centred around zero with a standard deviation that tends to decrease over time and is dependant on the overall population score. This standard deviation is somewhat analogous to selection pressure in real evolution (Mitchell, 1998). Over time, the population of rockets tend to do better on average (mean), so a constant selection pressure may not be suitable to decide good and bad parents for all cases. Thus, the selection pressure is set to be proportional to the score of the best performing rocket over may generations (the selection gets stronger over time – making for larger differences between good and bad networks). We used the function `norm.pdf` from the `scipy` library to enable this[4]. The formula for this is given in Eq. 5, where $x =$ *individual score*, $\mu = 0$ (the best possible score) and $\sigma \propto$ (best score in generation):

$$fitness(x) = f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \tag{5}$$

[3]The algorithm and evolution in general do not have agency and cannot *decide* at all, since they do not actively look at the specifics of the parameters – just the fitness – but implicitly do so, e.g. via the probability of reproduction if a set of parameters does better than another set. However, for brevity, we will continue to use this phrase.

[4]Documentation at `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html`

- **Selection:** A "roulette-wheel" or "fitness-proportional" selection is used to decide parents of the next generation. This is a fairly common selection method for GAs. Essentially, the relative probability of selection is proportionate to the fitness of the individual with respect to the population. An in-depth mathematical examination of this method can be found in Schmitt (2001), but for now we will rely on this high-level, but sufficient, understanding. The corresponding function to find probabilities for the selection of an individual x_i into the next generation is fairly simple, defined in Eq. 6.

$$P\left(x_{i_{parent}}\right) = \frac{fitness(x_i)}{\sum_{j=0}^{n} fitness(x_j)} \tag{6}$$

- **Reproduction:** Several reproduction methods can be designed for this task. Multiple parents may be chosen to create a new child that contains the genes/parameters of the parents parents, for example by partitioning two network graphs arbitrarily and splicing together their values. This is generally called *crossover* (Mitchell, 1998). However, for our algorithm, we opted to use a *clonal* model for reproduction, that is, a single candidate parent is copied over to the next generation (i.e. it is cloned). The population size remains the same over all generations. Bad-performing networks die out early on, and better ones are rewarded by being cloned forward more frequently, propagating the famous Darwinian "survival of the fittest" aphorism. This is far less computationally complex than crossover, and seemed to converge to a solution much more quickly based on early tests we did on simpler networks.
- **Mutation:** Once the new population is created, its parameters are changed by a small amount. This facilitates searching through the large network state space. The amount to be added to a given parameter value is sampled from a normal distribution centred around 0, with a hand-picked standard deviation, what we called the *strength* or *magnitude* of mutation. If this magnitude is too low, the algorithm may converge to a solution (local minimum in the fitness state space) very slowly and which may not be very good. If it is too high, it may not converge at all, with the behaviour of the population becoming essentially indefinite. This is analogous to the learning rate hyperparameter in backpropagation. The rate of mutation (the proportion of parameters that are mutated) was set to 1. Though this seems extreme, with a small enough standard deviation, our testing showed that it was a suitable value with which to find fairly good network parameters.
- **Generations:** The algorithm is run for a number of generations, which cannot be determined easily analytically. Empirically, we look at the scores or fitnesses to see where they plateau, which indicates a convergence to a solution (similar to looking at accuracy/loss curves for backpropagation). If run for several more generations past convergence, there is a possibility that the population of networks find different solutions, since mutation still occurs after each generation. This is more likely to result in a better solution (higher fitnesses) due to the nature of our chosen selection method. One notable benefit to GAs of this form is that we can tune the hyperparameters on the basis of the current generation. For example, if we see the population is converging to a good solution but is still erratic over generations, by example looking at score graphs, we can fine-tune it by reducing the mutation magnitude.

IV. Results

In this section we will discuss the results of the two methods. The section contains three subsections, the first two describing one of the training methods respectively. The final subsection compares the two methods and draws final conclusions.

A. *Backpropagation*

The backpropagation training resulted in *accuracy* = 73% and *loss* = 0.88 on the training set and *accuracy* = 77% and *loss* = 0.76 on the validation set. From Figures 5 and 6, it is clear that the network learns and improves itself over time: the accuracy function increases exponentially from a value of 44% and the loss function decreases exponentially from a value of 1.5. Additionally, towards the end of the training, both functions converge towards constant values, a clear indication of the fact that increasing the number of epochs would result in little to no improvement. Conversely, a smaller number of epochs of training results in lower accuracy and higher loss (e.g. on the training set: for n_epoch = 5, *accuracy* = 55% and *loss* = 1.2; for n_epoch = 10, *accuracy* = 70% and *loss* = 0.95).

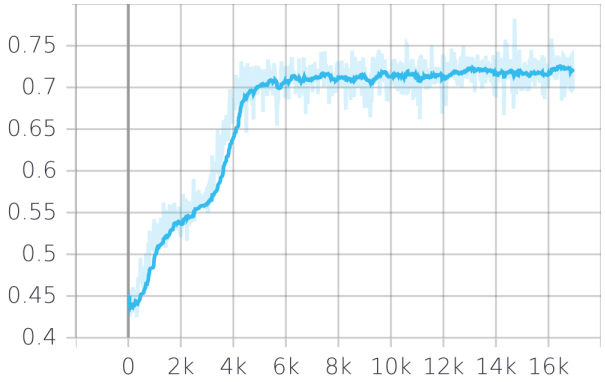


Figure 5: The accuracy curve of the *tflearn* MLP. The y-axis shows the value of the accuracy function and the x-axis shows the step of the training process.

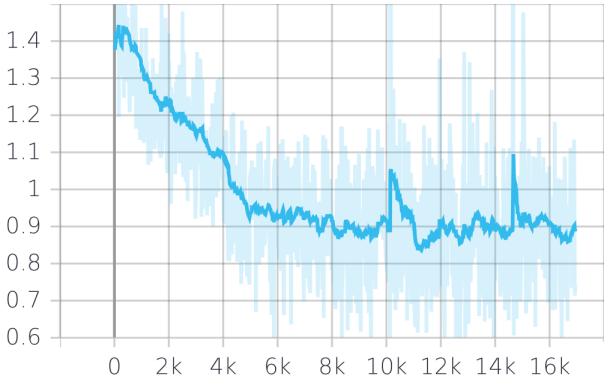


Figure 6: The loss curve of the *tflearn* MLP. The y-axis shows the value of the loss function and the x-axis shows the step of the training process.

We next studied the behaviour of the rocket controlled by the trained neural network. The process was as follows: newly generated games with random starting position of the rocket and random position of the target. The flight paths are presented in Figure 7 (right plot). Apart from a number of outliers, the rocket shows indisputable purposefulness in the trajectory towards the target: it identifies the position of the target relative to its own position (left or right), turns on the engine and rotates towards the target, turns off the engine when above the target, then falls to the ground. Worth noting is that the MLP does not seem to understand the concept of inertia, as it slightly overshoots when the target is further away and slightly undershoots when it is closer to its initial position. Moreover, it is interesting to note from Figure 7 (left plot) that the MLP seems to do better for targets on the left that for targets on the right, as the function increases for target position values larger than 500 (i.e. center of the screen).

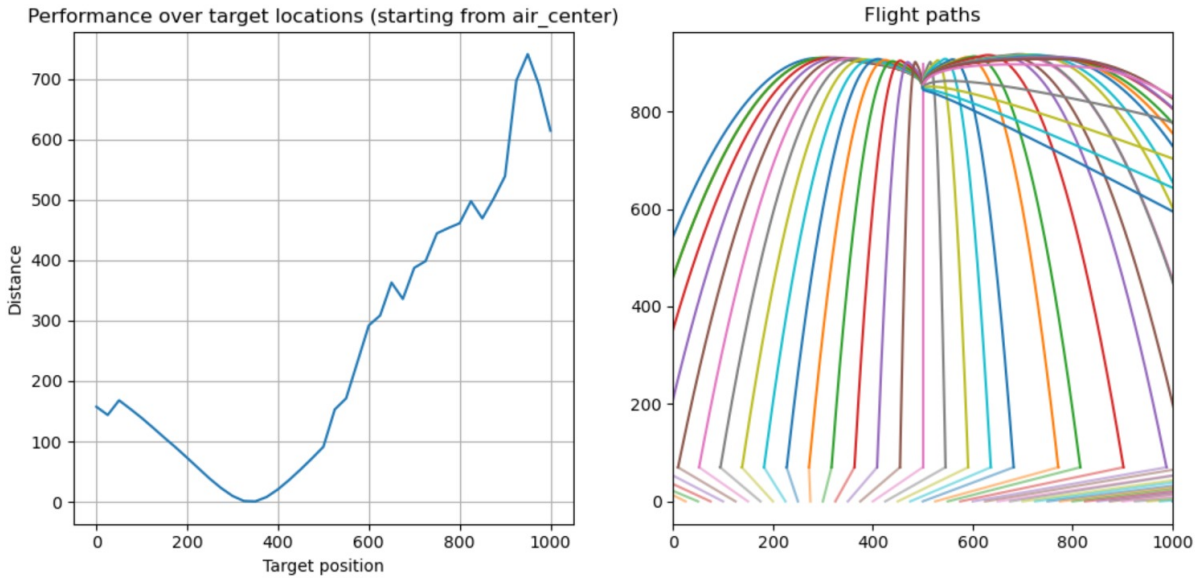


Figure 7: Left plot: the performance of the MLP in terms of the distance when landing, using a ReLU activation function. The y-axis shows the absolute value difference between the coordinates of the target and the coordinates of the rocket landing, the x-axis shows the position of the target (500 is the center of the screen); Right plot: the flight paths of the rocket, with starting position as the center of the screen. The axes represent the coordinates of the screen. The rocket lands at around $y = 80$, and the targets are placed at $y = 20$. The landing positions and ideal targets are connected with a straight line at the bottom of the parabolic flight paths.

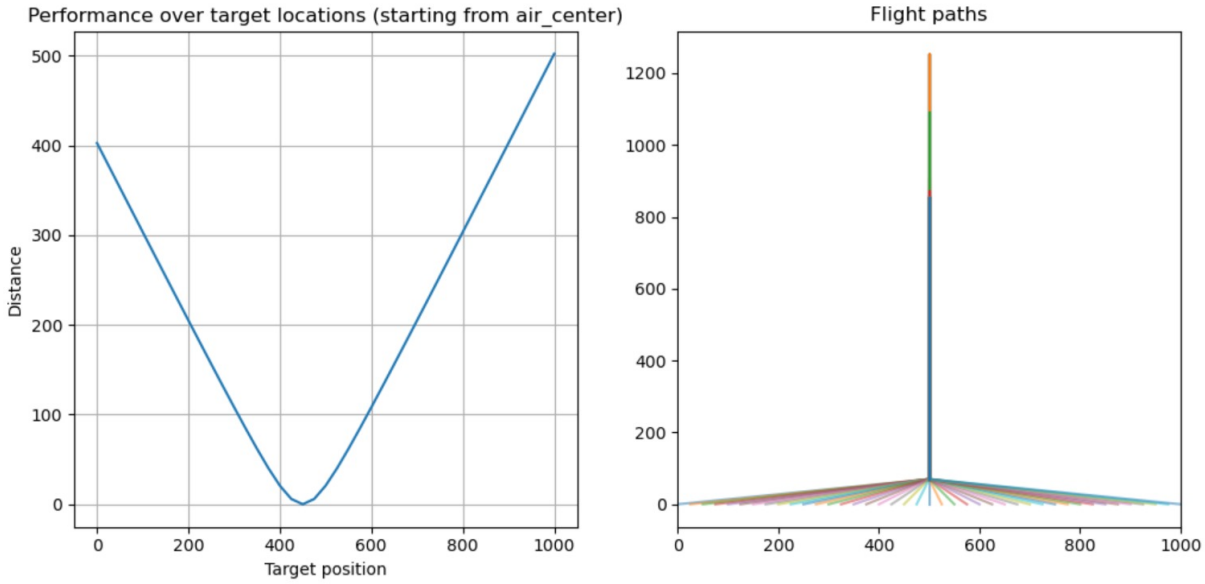


Figure 8: Left plot: the performance of the MLP in terms of the distance when landing, using a tanh activation function. The y-axis shows the absolute value difference between the coordinates of the target and the coordinates of the rocket landing, the x-axis shows the position of the target (500 is the center of the screen); Right plot: the flight paths of the rocket, with starting position as the center of the screen. The axes represent the coordinates of the screen

Another interesting thing to note is the impact of the activation function in determining the behaviour of the rocket. We replaced the ReLU function with the tanh function, modeled by formula Eq. 7.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{7}$$

As can be seen in Figure 8, this results in entirely different behaviour, as the rocket simply falls to the ground, entirely failing the task. Other activation functions gave rise to similar behaviour as the tanh. Therefore, we can empirically conclude that ReLU is most suited for this task.

Overall, while the behaviour could be improved, we can conclude that the training process was successful: the neural network shows clear signs of learning and identifying patterns from the training examples, as it identified the goal (i.e. landing on the target) without explicit instructions in a sense and found an efficient strategy for solving the task.

B. Neuroevolution

In general, the GA did well in finding neural networks that could steer accurately towards an arbitrary target within a few dozen generations at most. However, the original training task of to minimising distance to the target, rotation, velocity and fuel turned out to be a hard task, determined after a large number of tests with several variations on hyperparameters. As such, in the end, we set the task to just minimise distance to target (effectively creating a population of well-trained missiles, instead of rockets).

Several GA training sessions on different initial populations and configurations all usually resulted in "goal-directed behaviour" from the rockets (they turned and flew towards the target). For now, we will discuss one training session as an example of a specific configuration of hyperparameters that worked well.

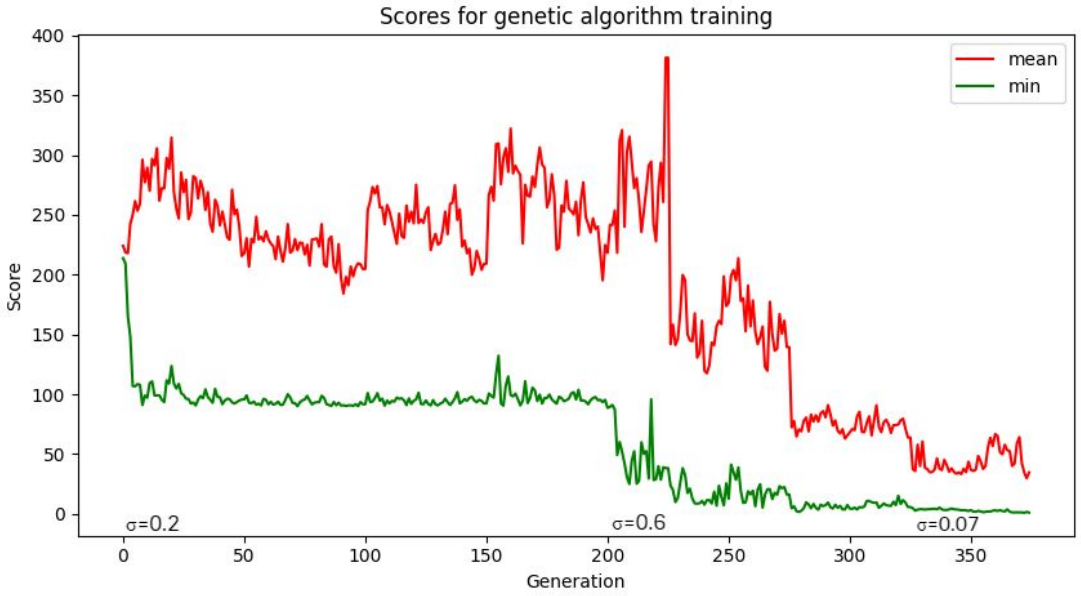


Figure 9: The population mean (higher line, red) and minimum scores (lower line, green) over 375 generations of training (the lower the score the better). The mutation strength σ starts off at 0.2, and is increased to 0.6 when a plateau is reached. This causes the population to find a new, better solution, so the scores fall. This hyperparameter is gradually reduced to its final value of 0.07 for the last 50 or so generations.

In Figure 9, we can see the mean and minimum scores of each generation of a population of 100 rockets over time, with each generation being tested on 10 trials of varying target positions. The mutation magnitude was changed over time to help with convergence. Figure 10 shows the final flight paths of the very best rocket from that generation starting from high up on screen in the horizontal centre (hereon referred to as the *air_center* of the screen). We also see how far away the rocket lands with respect to the target position in the graph in Figure 10. As we see here, they land very close to the target indeed (less than around 20 pixels away for each target).

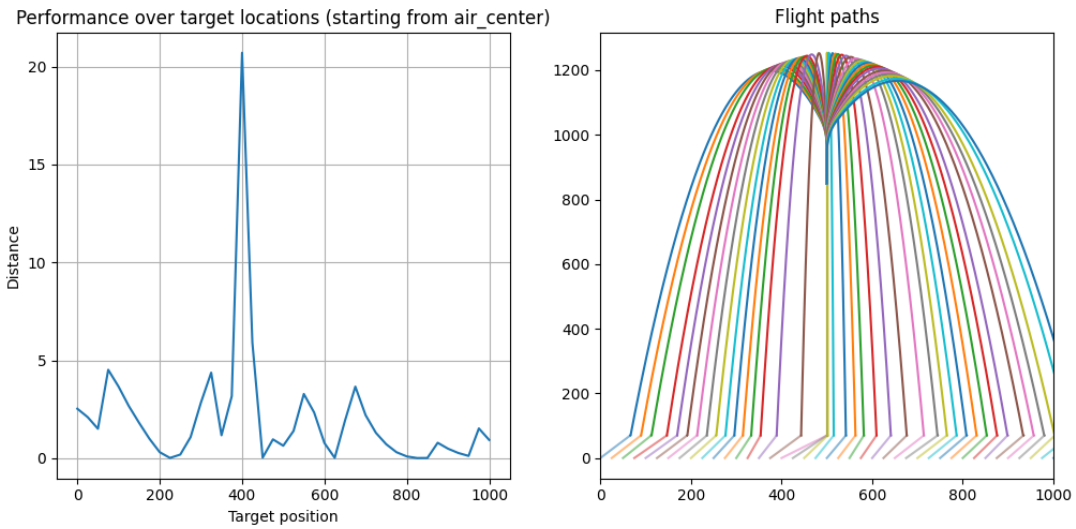
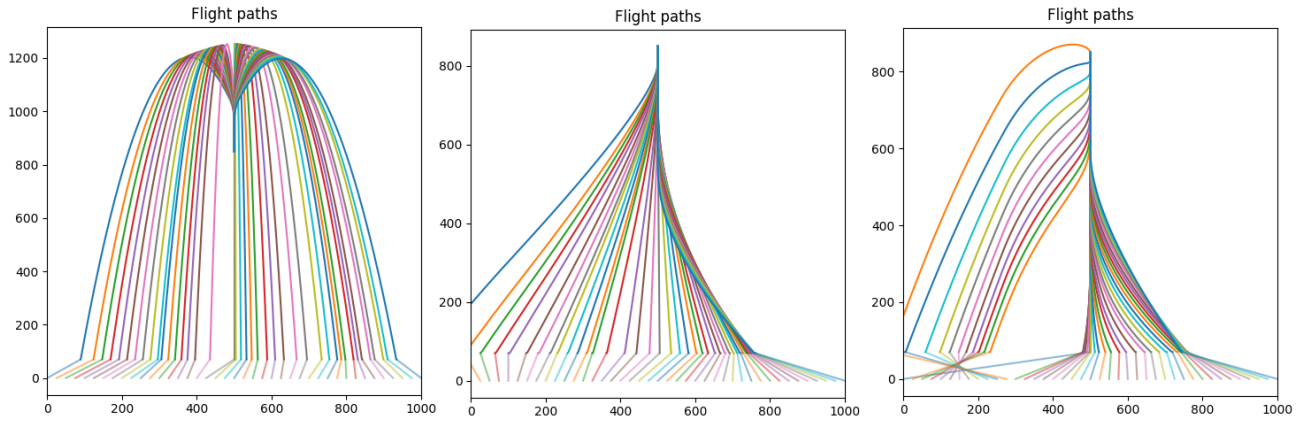


Figure 10: Left plot: the performance of the MLP in terms of the distance when landing. The y-axis shows the distance between the coordinates of the target and the coordinates of the rocket landing, the x-axis shows the position of the target (500 is the center of the screen). An ideal rocket would have a plot that looks like a flat line near zero; Right plot: the flight paths of the rocket, with starting position as the center of the screen. The axes represent the coordinates of the screen. The rocket lands at around $y = 80$, and the targets are placed at $y = 20$. The landing positions and ideal targets are connected with a straight line at the bottom of the parabolic flight paths.

What is fascinating about this training method however, is that it found ways to solve the training problem of minimising distance in ways that are entirely separate from what we saw with the human data. In particular, any given population seemed to hone in on one of two techniques: either fly very high, finish all its fuel at the beginning of the run while turning (Figure 11a, what we refer to as the "fountain" method), or fall to the ground, turn, and then turn on the thrust (Figure 11b, what we call the "falling" method). Usually, rockets that use the fountain method overshoot the target and slowly get closer to the target over time. Those that use the falling method tend to undershoot the target and also get better with time. Both tended to be biased towards one side or another (targets placed to one side of the screen would have better results than those placed on the other side). We saw that we could also coerce one method to turn into another by inducing large mutations while training. In one instance, it even did both simultaneously, depending on where the target was relative to the rocket (Figure 11c). If the target lay to the left, it showed a fountain behaviour, if it were to the right, it showed a falling one. Thus, with an even more complex optimisation task, such as including minimising landing velocity and angle, and enlarging the network to have more trainable parameters (and a thus larger search space) we are convinced that several interesting and non-intuitive solutions can be found. Perhaps these are better than a human's approach! Thus, supervised learning may not be the optimal training methodology.



(a) Example of ‘fountain’ behaviour. (b) Example of ‘falling’ behaviour. (c) Example of ‘mixed’ behaviour.

Figure 11: Various behaviours observed when training the MLP using a genetic algorithm. Note the larger heights (scaled y-axis) of flights in Figure 11a compared to the others. All rockets start at $(500, 850)$.

C. Comparison

Since the main aim of this study is to compare learning methods, we made sure that both methods used the same (or nearly the same) neural architecture to reduce any performance differences due to the architecture. Thus, both training methods are using a multilayer perceptron with 3 layers: one input layer (with 11 neurons), one hidden layer (with 11 neurons) and one output layer (with 6 neurons) as shown in Figure 4. Additionally, both learning methods have ReLU as their activation function.

We introduce a few notable differences between them:

- **Behaviour:** The most significant and perhaps most interesting difference between the two training methods were the behaviours of the final networks. Humans tend to turn the rocket towards the target, and fly directly towards it. Thus, the backpropagation-trained model seemed to do something similar (Figure 7). The GA-trained models had several possible solutions, as discussed earlier, preferring either a ‘fountain’ or ‘falling’ behaviour (Figure 11). The human-trained model initially looks like a fountain-esque behaviour, but it flies much more horizontal than the fountain one does. There is no reason to believe the anthropogenic behaviour of landing a rocket is ideal, and other solutions can be found with training methods such as GAs. We see this, in fact – from Figures 7 and 10, we note that the GA-trained networks land much closer to the intended targets than the backprop-trained ones. However, the currently observed behavioural differences could also be due to differences in training goals.
- **Training goals:** Initially, we aimed to *land* the rocket: minimize distance to target, velocity, and land it upright. Additionally, the initial position of the rocket was intended to be arbitrary. However, we soon learnt that the GA could not train well under these constraints, as far as we experimented with it. Thus, the final goal of the GA was just to minimise distance, and start from the “air_center” of the screen (point $(500, 850)$). The humans, on the other hand, understood the more complex task, which was then somewhat approximated by the backprop-trained network. They only gently propelled themselves towards the target and conserved fuel for landing, regardless of starting position. In the end, neither training method could perfectly land the rocket successfully, though we believe it is possible with perhaps a more complex network (see *Further improvements*, section V-A).
- **Training speed:** The training time for each training method to plateau (GA: scores don’t

improve; backpropagation: loss does not decrease) depends drastically on their respective hyperparameters – however, it seems like backpropagation reached this point faster than GAs. Though this could be due to library-specific optimisations, evolutionary models generally are slow, but may be made to run in parallel, though we did not implement this. Training with backpropagation took time on the order of a few minutes, whereas the GA sometimes took several hours.

V. Discussion

In this section we will discuss possible further improvements of the two learning paradigms and our general and concluding impressions.

A. Further improvements

There are several improvements and further research that we believe may improve results and enable different learning tasks. Here, we discuss improvements to particular training paradigms, and general improvements at the end.

1) General: Both learning methods learnt to fly towards the target, but not much else. This could be due to the structure of the network itself, regardless of the training paradigm. Thus, a more complex network may be useful.

- **Other MLP Architectures:** The first of improvements that can be tried relate to the architecture of the networks. Both designs that we have presented use one hidden layer with 11 neurons. Deeper neural networks with more than two hidden layers could perform better, given the task's complexity. Tuning the hyperparameters such as the learning rate and the regularization term by using, for example, a grid search should also lead to better results.
- **Other deep learning paradigms:** Using a different deep learning technique like reinforcement learning does also seem to lead to a better behavior in the lunar landing problem. Gadgil et al. (2020) obtained a 80% accuracy in the action selection by using Q-learning.
  An even more challenging task and a matter for future research is to train a convolutional neural network (CNN) with frames from each played game. This method would indeed make the network learn just like a human does, by visually observing different games. (Sutskever & Nair, 2008) trained a CNN to play Go, by feeding it with images representing the current state of the game. They achieved a 34% accuracy when predicting the moves of professional Go players, which is impressive, considering the highly complex state space of a Go game. Also, Bellemare, Naddaf, Veness, and Bowling (2013) used CNNs to train a model to play Poker. Eventually, the network was able to play the game with human experts. This shows just how much can be achieved by using CNNs in game applications.

2) Neuroevolution: There are a number of improvements or further research that can be done with the evolutionary algorithm, apart from those in the architecture. Some are listed below:

- **Hyperparameter variation:** The genetic algorithm has a number of hyperparameters that can be tweaked that may result in more reliable landing behaviour than what we attained. For example, a different selection algorithm, fitness functions, or mutation parameters may help with converging to good local minima.
- **Speed increase**: A grid search over hyperparameters does not seem viable with the current version of the code, since it is serial. When training, we also noted that the backpropagation tended to be much faster than an equivalent GA training session. Of course, libraries like sklearn and TensorFlow have highly optimised these procedures, but in general, GAs are known to be quite slow if run serially. These libraries ran training sessions over a few minutes at most,

whereas the GA took up to an hour per generation. However, a major benefit to GAs or any agent-based approach is that it can be meaningfully parallelised, with each thread handling a subset of individuals. This is not done in the current version of the code.

- **Reproduction methods:** We only use a clonal (asexual) model of reproduction here, which, though being the simplest method, did reasonably well. However, it may be fruitful to examine different sexual reproduction methods as well. Crossover is especially useful to maintain a wide search space for each generation, but was deemed unnecessary by us for the current task.
- **Other neuroevolution algorithms**: Other neuroevolution algorithms apart from clean GAs exist, such as the previously mentioned NEAT (Stanley & Miikkulainen, 2002). This incrementally augments both the trainable parameter values, and the topology of the network, slowly making it more complex. Since we were comparing training methods, we decided to settle on a fixed topology, and thus did not consider other methods for this project.

3) Backpropagation: In the case of the backpropagation-based model, there are a number of smaller improvements that one can make in order to increase the overall performance of the network.

- **Training Data:** The primary set of improvements is related to the dataset that was used for training. Considering that the task at hand is quite involved, it is difficult to expect that the network will perform well after training with only 154 successful games. Therefore, more data would most likely help the network. Training with a suitable architecture should also make the network improve when using unsuccessful games. Even in the failed games, the user is still trying to approach the target, so the network should pick up this behavior regardless.
- **PCA:** Another idea is to perform a principal component analysis (PCA) in order to reduce the number of dimensions and make the data more compact. Usually, in PCA, we achieve this dimension reduction by projecting each data point on the first principal components. These principal components are the eigenvectors of the covariance matrix formed by the dataset. The decomposition of X is given at Eq. 8, where W is the matrix of weights, whose columns are the eigenvectors of $X^T X$. This would be especially useful if the task gets more complex and more candidate input variables are created.

$$T = XW \tag{8}$$

B. Conclusion

Both training paradigms, we examined, backpropagation and neuroevolution, did quite well with respect to flying close to the target. Though there were notable technical differences in the two methods, as we saw in the *Comparison* section (section IV-C), they both showed target-oriented flight paths. The differences in behaviour were also surprising, with the backpropagation-trained networks having a human-esque flight pattern (arising from the training data trends) and the neuroevolution networks finding multiple different solutions. We think with more complex networks, more complex behaviour can be learnt, however, we did not have time to adequately explore this.

We also had an enjoyable learning experience during this project. All code was documented and shared on GitHub. Using resources like Google Colab was also very useful, being directly integrated with cloud storage. Implementing an MLP by hand for the evolution training paradigm clearly shone light upon the central role of linear-algebra in neural networks. Learning how to use libraries such as TensorFlow and sklearn also gave us a glimpse of state of the art learning methods, and wrangling with their specific requirements and documentation is no doubt useful for future work.

REFERENCES

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Blackmore, L., Açikmeşe, B., & Scharf, D. P. (2010). Minimum-landing-error powered-descent guidance for mars landing using convex optimization. *Journal of guidance, control, and dynamics*, *33*(4), 1161–1171.

Damien, A. (2016). *Tflearn.* `https://github.com/tflearn/tflearn`. GitHub.

Gadgil, S., Xin, Y., & Xu, C. (2020). Solving the lunar lander problem under uncertainty using reinforcement learning. In *2020 southeastcon* (Vol. 2, pp. 1–8).

Gomez, F., Schmidhuber, J., Miikkulainen, R., & Mitchell, M. (2008). Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, *9*(5).

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... Oliphant, T. E. (2020, September). Array programming with NumPy. *Nature*, *585*(7825), 357–362. doi: 10.1038/s41586-020-2649-2

Hui, C., Landi, P., Minoarivelo, H. O., & Ramanantoanina, A. (2018). *Ecological and evolutionary modelling*. Springer.

Jaeger, H. (2001). The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, *148*(34), 13.

Jaeger, H. (2021). *Neural Networks (AI) (WBAI028-05) – Lecture notes.* Personal collection of H. Jaeger. University of Groningen, Groningen, The Netherlands.

Liu, Z. (2006). A guided genetic algorithm for the planning in lunar lander games.

Mitchell, M. (1998). *An introduction to genetic algorithms*. MIT press.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... others (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, *12*, 2825–2830.

Peters, C., Stewart, T. C., West, R. L., & Esfandiari, B. (2019). Dynamic action selection in openai using spiking neural networks. In *The thirty-second international flairs conference.*

Romero, J., Romero, J. J., & Machado, P. (2008). *The art of artificial evolution: A handbook on evolutionary art and music*. Springer Science & Business Media.

Ronald, E., & Schoenauer, M. (1994). Genetic lander: An experiment in accurate neuro-genetic control. In *International conference on parallel problem solving from nature* (pp. 452–461).

Schmitt, L. M. (2001). Theory of genetic algorithms. *Theoretical Computer Science*, *259*(1-2).

Stanley, K. O., Clune, J., Lehman, J., & Miikkulainen, R. (2019). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, *1*(1), 24–35.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, *10*(2), 99–127.

Sutskever, I., & Nair, V. (2008). Mimicking go experts with convolutional neural networks. In *International conference on artificial neural networks* (pp. 101–110).

Tilburg, G. v. (2021). *Echo state network based helicopter control.* University of Groningen.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., ... SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. doi: 10.1038/s41592-019-0686-2

Yao, X. (1993). A review of evolutionary artificial neural networks. *International journal of intelligent systems*, *8*(4), 539–567.