Evolving Recurrent Neural Network Controllers by Incremental Fitness Shaping

Kaan Akinci[1], Andrew Philippides[2]

[1,2]Informatics, University of Sussex

[1]ka373@sussex.ac.uk

[2]andrewop@sussex.ac.uk

Abstract

Time varying artificial neural networks are commonly used for dynamic problems such as games controllers and robotics as they give the controller a memory of what occurred in previous states which is important as actions in previous states can influence the final success of the agent. Because of this temporal dependence, methods such as back-propagation can be difficult to use to optimise network parameters and so genetic algorithms (GAs) are often used instead. While recurrent neural networks (RNNs) are a common network used with GAs, long short term memory (LSTM) networks have had less attention. Since, LSTM networks have a wide range of temporal dynamics, in this paper, we evolve an LSTM network as a controller for a lunar lander task with two evolutionary algorithms: a steady state GA (SSGA) and an evolutionary strategy (ES). Due to the presence of a large local optima in the fitness space, we implemented an incremental fitness scheme to both evolutionary algorithms. We also compare the behaviour and evolutionary progress of the LSTM with the behaviour of an RNN evolved via NEAT and ES with the same fitness function. LSTMs proved themselves to be evolvable on such tasks, though the SSGA solution was outperformed by the RNN. However, despite using an incremental scheme, the ES developed solutions far better than both showing that ES can be used both for incremental fitness and for LSTMs and RNNs on dynamic tasks.

Introduction

While deep feed-forward neural networks have been used very successfully in static problems where there is no temporal dependence between inputs, non-Markovian problems such as controllers for robots or games could potentially benefit by temporally extended networks (networks with a temporal element). Long short-term memory (LSTM) networks – which have complex forms of memory - are interesting networks because of their potential to capture long term temporal dependencies and have been used successfully in a number of tasks. (Sutskever, at al. 2014). However, it is harder to find the optimal settings for these networks using things like back-propagation due to exploding gradient problems (Salimans, at al. 2017). Indeed, to train LSTMs, people typically use back-propagation through time or reinforcement learning (Bakker, 2001). Recently, evolutionary optimisation has been used as an alternative to reinforcement learning to develop solutions since it require less computational power per episode and memory (Salimans, at al. 2017). Here we therefore see if it is possible to evolve an LSTM for a lunar lander game using either a steady-state genetic algorithm (SSGA) or an evolutionary strategy (ES). While ES's have been shown to outperform GAs on a number of tasks, it is not clear if it will be true for such dynamic networks operating in irregular, noisy, fitness landscapes.

Many different evolutionary algorithms have been effective at finding solutions for robotics tasks, not least as they provide a very flexible approach. For example, it is relatively easy in such problems to encourage robustness to problem variations and generalisability across starting conditions through evaluating the agent on multiple trials. In the lunar lander problem, this robustness is necessary as the landing surface and starting conditions are randomly generated for each instantiation. Steady-state genetic algorithms are particularly good at encouraging robustness because solutions that perform well stay in the population and are evaluated many times over the course of evolution, meaning they experience a very wide range of starting conditions without incurring the computational cost of evaluating all solutions in the population over the same number of trials. As evolutionary strategies work by having multiple copies of a single individual, which then moves through fitness space, it is not clear if they will lead to a similar level of generalization. Another way in which the flexibility of evolutionary algorithms helps in dynamic optimisation tasks, is that the issue of local optima in the search space can be ameliorated by techniques such as fitness shaping and incremental evolution in which the designer can guide the solution to the types of behaviour desired. Again, such schemes have been used successfully with SSGAs, it is not clear that they will also function well with ESs. As the lunar lander game with the default fitness function has local optima issues (as the agent can gain a reasonable score by doing nothing) as well as requiring solutions robust to starting conditions, we here use it as test-bed to see if an incremental fitness scheme can be as effective for an ES as it is for an SSGA and if so, whether the solutions generated are robust to changes in conditions.

Despite the issues of a large local optima and very noisy fitness evaluations, we show that LSTM networks can successfully be evolved with both evolutionary algorithms. However, through comparison with an RNN evolved with NEAT (used as a benchmark and to derive the network morphology for the evolutionary algorithms) and ES, the SSGA, while able to produce a viable network, does not appear to be taking advantage of the rich dynamics provided by the LSTM. In contrast, despite not being population-based in the same way as the SSGA, the ES generates robust controllers demonstrating both that these algorithms can

function effectively with both incremental fitness functions, noisy evaluations and highly dynamic LSTM networks.

Background and Methods

In this section we briefly describe the methods that were used during the experiments.

Long-Short Term Memory (LSTM) Networks

Long Short-Term-Memory (LSTM) networks are advanced versions of RNN networks that can selectively forget and update hidden states. In a basic LSTM perceptron there are four different gates that determine the output and hidden states (Gers, at al. 2002). These gates are commonly referred as "forget", "input", "recursive memory" and "output". Additionally, LSTM networks have a hidden state and a memory. These properties allow LSTM networks to be aware of past actions and experiences thus enabling long-term temporal dependencies in the decision making process.

During the experiment, all of the parameters controlling the shape of the LSTM network were kept constant as evolving the LSTM shape as well would require a bigger study and more computational power and we were here interested in whether ESs would be able to work in noisy spaces with fitness shaping. In order to choose the morphology, initial experiments were conducted using NEAT and RNNs. Following tests, the size of the layers and depth of the network were chosen to be slightly bigger than the best NEAT derived RNN. Specifically, the LSTM comprised two ReLU layers of 20 and 25 units followed by a fully connected softmax layer.

Genetic Algorithm

Genetic algorithms are random heuristic search algorithms that are inspired by biology. Genetic algorithm are commonly used in multi-variable optimisation tasks (Gers, at al. 2002). They rely on continually evaluating different combinations of variables on an optimisation task and using the result as feedback to improve the variable choice. In order to use these algorithms, variables to be optimised are represented by genes and solutions are represented by genomes. A genome thus consists of all variables and their corresponding values. The result of the optimisation for the given genome is referred to as the fitness of the genome and the set of the genomes are referred to as the population (Whitley, 2001). Genetic algorithms use an analogue of an evolutionary process to iteratively improve the population. Many variations of genetic algorithms exist, but the majority work by using the fittest individuals in a population to generate the next population via selection, crossover and mutation.

In our experiments, the weights of the LSTM or RNN are used as the genotype meaning it is 7,040 variables long for the LSTM and 1,835 long for the RNN. As a crossover operator we used uniform crossover which produces two offspring, where the first offspring has 80 percent chance per gene to get genes from first parent and 20 percent chance to get from second parent, and second offspring has 80 percent chance from first parent and 20 percent chance from second parent. For mutation we used uniform mutation in the range of [-1,1] with 0.35% mutation chance per genome. We also used a steady state genetic algorithm, which instead of replacing all of the population at once, as in a generational GA, iteratively selects two parents and produces two offspring which replace two members of the current population. The original replacement method of the population was replacing the parents, as shown in Pramanik and Setua (2014). However, we found the method proposed by Gilbert Syswerda (1991) to work better in our experiments (Pramanik and Setua, 2014; Syswerda, 1991). Specifically, in order to choose the individuals for mating, proportional roulette wheel selection was used and in order to choose the individuals to replace, reverse proportional roulette wheel was used. The architecture of the networks were as described above and a population size of 30 was used.

Evolutionary Strategy

An Evolutionary Strategy (ES) is a nature-inspired algorithm (Salimans, at al. 2017) and is a variant of a GA in which new individuals are not generated by random variation of one or two parents via crossover and mutation. Instead, an ES uses the statistics of the current population within the fitness space to determine new individuals and thus the direction of improvement in fitness space. An ES thus effectively uses mutation only and no crossover. Here we used the ES algorithm that is described by Salimans (Salimans, at al. 2017). Specifically, at each evaluation there is only one 'real' individual and N-1 (where N is the population size) simulated individuals which surround the real individual. Each gene of each simulated individual is constructed by adding a small amount of Gaussian-distributed noise to the corresponding gene of the real individual. The noise of each gene of each individual is scaled by its fitness value. The product is added to the genes of the real individual resulting in a new real individual (Salimans, at al. 2017). N-1 simulated individuals are created around this one individual and the process repeats.

The architecture of the networks was the same as the steady state network and a population size of 30 was used.

Algorithm 1: Evolutionary Strategies

for t=0,1,2,… **do**
Sample $\epsilon_1, \ldots, \epsilon_i$ individuals using Gaussian noise
Compute fitness $F_i = F(\theta_t + \sigma\epsilon_i)$ for i=1,..n
Set $\theta_{t+1} = \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
end for

Algorithm 1 displays how each generation was created. n represents the population size, θ_t represents the real individual's genes, α represents learning rate, σ represents variance of the Gaussian noise, F_i represents the fitness value of i'th individual. The variance of the Gaussian distribution was chosen to be two and the learning rate was chosen to be 0.5.

NEAT (Neuroevolution of Augmenting Topologies)

In order to get an idea of what network size might be effective for evolution and thus avoid complications due to over/under sized network structure, we first trained an RNN with NEAT, an approach which has proved successful previously (Stanley and Miikkulainen, 2002). NEAT is an advanced version of a

Weight Evolving Artificial Neural Network (TWEANN), a specialised genetic algorithm which focuses on optimisation of the weights of an artificial neural network (Stanley and Miikkulainen, 2002). The major difference between NEAT and TWEANN is topological evolution. In other words, the composition and the architecture of the network evolve over time together with its weights. Since the NEAT library does not support LSTMs, RNNs were used for these initial experiments, using the NEAT algorithm described by Stanley and Miikkulainen (2002). Due to limited computational power, we used a relatively small population of 30 with minimum species size 3 and 2 elites per deme. ReLU was used as the activation function.

Lunar Lander Game

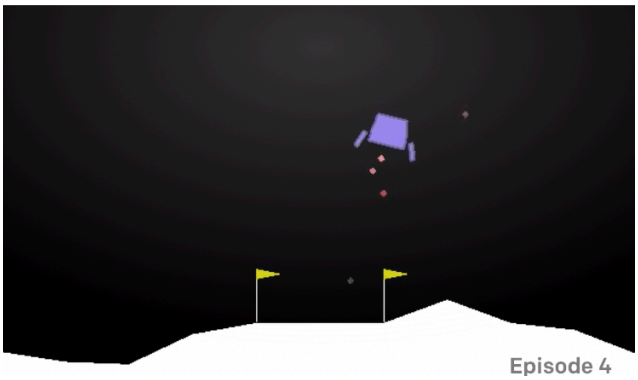

Figure 1: OpenAI Gym Lunar Lander Environment

The Lunar Lander game is a simulation provided by OpenAI (Brockman, at al. 2016) in which controllers have to land a spacecraft in a designated landing area indicated by the flags in Figure 1. Specifically the flags are at coordinates (-0.25,0) and (0.25,0) where (0,1) is the starting position of the spacecraft. The problem is made more difficult as the surface of the moon is randomly generated on every evaluation. The simulation area is bounded and if the spacecraft exits the boundary, the simulation stopped. While the spacecraft always starts from the same coordinates, (0,1), the starting angle is random as is the initial velocity (both magnitude and direction).

The environment is frictionless but subject to a constant gravity force towards the surface of the moon. The spacecraft has three thrusters, bottom, left and right. The bottom truster increases spacecraft's speed in the direction it is facing, while left and right make it rotate clockwise/counter-clockwise respectively. Each thruster produces a constant thrust and the spacecraft is subject to basic momentum constraints. In any given instance, the spacecraft could only use one of its thrusters. The goal was to land the spacecraft without crashing (landing with more than -0.6 vertical velocity or not upright) and using the minimum amount of fuel. The total fuel was unlimited however; the spacecraft was limited to 1000 action commands.

The Lunar Lander game expects one command for input at each step. These commands are coded as "*0*","*1*","*2*" or "*3*" which correspond to: *do nothing, use left thruster, use bottom thruster* and *use right thruster,* respectively. After the game receives one of those commands, it returns four parameters, which are *observation*, *score*, *done* and *info*. The observation parameter, which represents the state of the environment, has eight variables: "x" and "y" coordinates, speed in vertical and horizontal axes, facing angle, angular velocity, and two Booleans, leg 1 and leg 2, which state whether a leg touches the ground in the goal zone (and which we do not use in network training though the default fitness function does). The *score* parameter is the value of default fitness function and *done* is a Boolean indicating the simulation's stop condition.

Fitness Shaping Through Incremental Evolution

The default fitness function (displayed below) provided by the simulation discourages fuel consumption ($TrusterPower$ terms) while encouraging every action that shortens the distance between the spacecraft and the goal position and penalising every action that increases the distance between the spacecraft and the goal location (first two terms of the equation). Touching the goal position rewards 10 points per leg of the spacecraft. Also attempting to takeoff after landing is discouraged and causes negative points. The formula of the default fitness function is given as;

$$fitness_t = -100\sqrt{X_t^2 + Y_t^2} - 100\sqrt{X_{acc(t)}^2 + Y_{acc(t)}^2} - 100angle_t + 10leg1 + 10leg2 - fitness_{t-1} - 0.3MainTrusterPower - 0.03SideTrusterPower$$

Because of the penalization of fuel consumption (Main/ Side Truster Power term in the equation), attempts to evolve a solution with the default fitness were subject to issues with local optima (see Results) in which the spacecraft does not apply any thrusters and simply falls down. We explored different fitness functions to get around this issue (e.g. applying multiple thresholds to the action commands and discouraging use of the "0" action command) but were not successful.

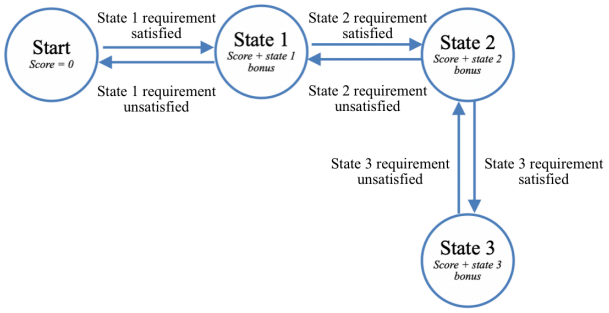

Figure 2: Incremental Fitness Function State Diagram

Thus we designed a fitness function based on different behavioural states, as illustrated at Figure 2. This fitness function was designed to define the problem in a more behavioural way, eliminate the issue of fuel consumption and decreasing the number of variables in the fitness function. Instead of trying to measure the properties of the agent and to determine if it is good or not, this fitness function categorises the current state of the agent and gives bonuses to certain actions/behaviours. With this approach we aimed to add action priorities into the evolutionary process. However the key

feature is that the agent has to perform well at a given state, meaning a certain type of behaviour has been developed, before the next state is evaluated. These states and transitions are:

State 1: Flying state. Detected if the agent was flying. The detection was made by thresholding. If the vertical velocity of the agent was between 0.0 and -0.80, the agent was rewarded 0.5 points and state 2 was enabled. If the agent's velocity was less than -0.80 but it was using its bottom thruster, it was rewarded 0.1 point. However, the transition to state 2 was disabled.

State 2: Horizontal stabilisation state. The agent was rewarded 0.5 points and state 3 was enabled if the angle of the agent was between 0.8π and -0.8π, which translates to being perpendicular to the ground as 0 is the angle of the normal to the ground, with some offset. Also the agent was rewarded 0.1 points if the angle was off-perpendicular but it used one of its side thrusters to correct. If neither of these conditions were met, 0.1 was subtracted.

State 3: Flight route minimisation state. This state rewards the flight trajectory of the agent. Every step that took the agent closer to the landing location gave $100 * displacement$ points. If the displacement was negative (going away from the landing area), nothing was rewarded.

The rationale behind these states were that the agent should be in control of its speed in order to fly, to be able to fly to control its orientation, and to be able to guide its orientation to fly towards the landing pad, which is the desired final behaviour.

We initially tried implementing the behavioural fitness function with all states used together i.e. without gating the states by only evaluating if it passed the previous one, but it is overly complicated and does not produce good results. In order to reduce the complexity of the heuristics, as suggested by Barlow (at al. 2004), an incremental fitness function was implemented. In this approach the behavioural fitness function is gradually modified. The heuristic fitness function was thus divided into three stages, hover, stabilise and orient (States 1, 2 and 3). The first stage of the incremental fitness function prioritises the flight time and the transition between State 1 to State 2 is blocked. For the second stage, the stabilisation phase, the fitness function was allowed to move from State 1 to State 2, but transition to State 3 was blocked. At the third stage, orient, the transition to the third state was enabled. This approach enabled the agent to develop certain behaviours easily while retaining previously learned behaviour.

Resampling and Noise Reduction

As the simulation environment randomly changed at every evaluation, some environments were more suitable for landing while some environments were less. As Pietro (2004) states, noise in the environment decreases the learning rate and the population has a chance to forget what it has learned (Di Pietro, at al. 2004). However, completely eliminating randomly generated landscapes and performing trials on a single static landscape would encourage overfitting and would prevent generalisation of the network. Thus, each network performed on n different environments and the fitness scores were sorted in an ascending order and combined as described below:

$$fitness = \sum_{i=1}^{n} \frac{fitness_{i-1}}{i+1}$$

This function decreases the effect of the best episode and increases the effect of the worst episode. The individuals that perform well at all episodes perform significantly better than those that are only good at certain situations. This fitness function contributes to the generalisation of the network. The result of 10 episodes will be referred as a trial.

Since the environment was regenerated in every trial, the complexity of the solution varies between each trial. There were three different elements that contribute to the characteristics of the environment which were: the landscape, the starting angle of the agent and the starting vector. The diverse set of starting angles and the starting vectors ensured the network could not overfit to any action command or to find a fixed set of actions that lead to success. Also, the landscape alters the vector effect of the thrusters, e.g. the bottom thruster may have produced velocity on the horizontal axis due to an obstacle in the environment. All of these factors increase the complexity and high variance due to noise of the simulation. In order to solve the issue with the landscape, every generation performed "n" different episodes, where the k'th trial of i'th individual had the same random factors as the k'th trial of j'th individual. "n" randomly generated landscapes were selected with "n" different starting vectors which were picked from uniformly distributed random values. In order to prevent bias in the starting angles, the starting angles were divided into three groups, left, right and middle. The starting angle varied in between "$-\pi$" and "π", where "0" is perpendicular and "π" and "$-\pi$" is horizontal to the ground. The starting angles for the left group were chosen from randomly distributed values between "-0.53π" and "-0.2π". The angles for the right group were chosen from randomly distributed values between "0.2π" and "0.53π". The angles for the middle group were chosen from randomly distributed values between "-0.2π" and "0.2π". The number of left angles and right angles were distributed equally while the ratio of the middle group was one in seven trials. This method was implemented to give a range of starting angles of similar difficulty, hence as the middle angle was the easiest to solve the issue, the ratio of the middle angle was the lowest.

Results and Analysis

We started using the default fitness function but it soon became evident that there was a local minimum in the fitness space. We ran the Steady State Genetic Algorithm with LSTM network with the default fitness function, for 5,200 trials (52,000 evaluations), where each trial consists of 10 resampled episodes however, the fitness does not improve and behaviour of the resulting network was the same in all runs.

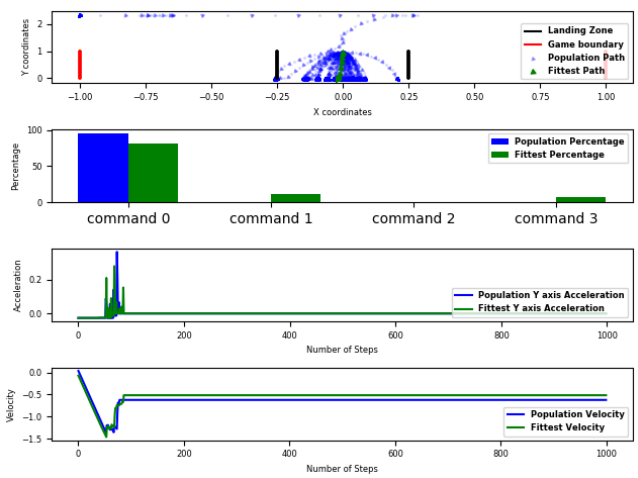

Figure 3: SS GA evolved by default fitness function. The top panel shows the flight trajectories, while the second panel shows the percentages of the actions that are being used during the flight. Third and fourth panels display the acceleration and velocity over time.

Figure 3 illustrates the behaviour of SS GA. The top panel shows the flight trajectories, while the second panel shows the percentages of the actions that are being used during the flight. The third panel displays the acceleration and velocity of the agent. The default fitness function has an unavoidable local optimum, where the agent abuses command 0 and does not use any other command and during the training process, the fitness of the population didn't improved. The reason behind the behaviour was the punishment of fuel usage. In other words, if the agent does not do anything, it will not consume any fuel and since fuel consumption was discouraged, the agent will perform better than the ones that were trying to fly.

With the introduction of the incremental fitness function and fitness shaping, we were able to avoid this local minimum. During the training with the incremental fitness function the fitness was improved. Thus, we decided to run the algorithm longer periods in order to obtain a well developed model. After, running the algorithm with the incremental fitness function for 20,000 trials (250,000 evaluations), the behaviour of the network had improved.

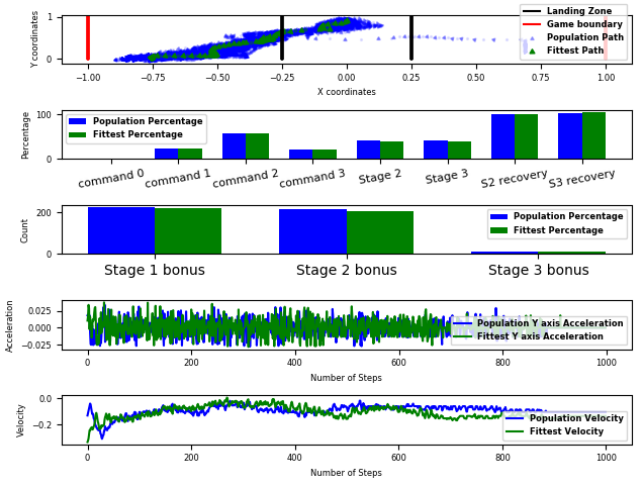

Figure 4: SS GA evolved by incremental fitness function. As in figure 3, The top panel shows the flight trajectories, while the second panel shows the percentages of the actions that are being used during the flight. However it additionally shows the amount of time the agents spends in stages 2 and 3 as well as the amount of steps that led to a transition to step 2 or 3 as step 2/3 recovery respectively, which gives insight into how much of the time the spends in these stages. The third panel shows the additional stage bonuses accrued the fourth and fifth panels display the acceleration and velocity over time. The same conventions are used for the following figures.

The behaviour of the steady state LSTM algorithm is displayed in Figure 4. The algorithm had significant improvements in acceleration and velocity controlling aspects as well as horizontal stabilisation. The agent learned to slow down its speed and slowly glide down. However, it failed to control its flight path. The points earned from each stage in the incremental fitness function are displayed in the middle panel of Figure 4. The stage 1 bonus (descending points) and stage 2 bonus (angle control) were high while stage 3 bonus (flight path) was very low. This behaviour was due to the lack of generalisation at stage 3.

To see if this behaviour was caused by a fundamental problem with using an LSTM network or network size, we ran the same experiment with an RNN network using the NEAT algorithm.

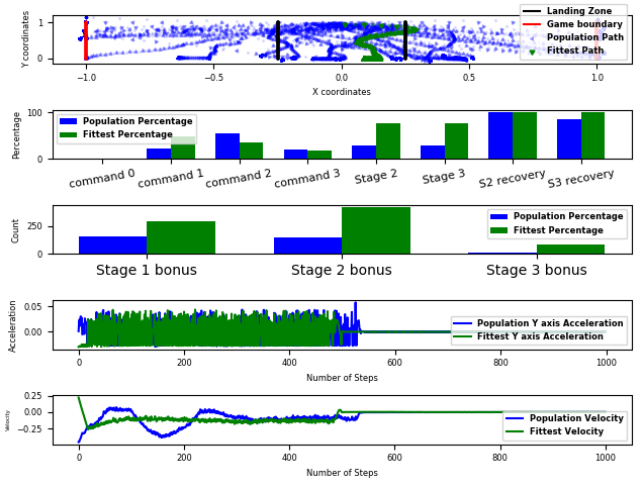

Figure 5:NEAT RNN evolved by incremental fitness function

Figure 5 displays the behaviour of the RNN NEAT network evolved using the incremental fitness function over ~360,000 trials (3,600,000 evaluations). The NEAT network managed to learn all stages showing that the fitness function is viable. In particular, when Figure 4 and Figure 5 are compared, while the LSTM performs similarly in stage 1, RNN with NEAT performs far better at stage 2 and stage 3, which leads to a better control over its orientation and flight path. This lead us to believe that using an LSTM is a viable option but that perhaps it is the type of evolution which is holding performance back. We thus repeated the same experiment with an ES algorithm using the LSTM and RNN network.

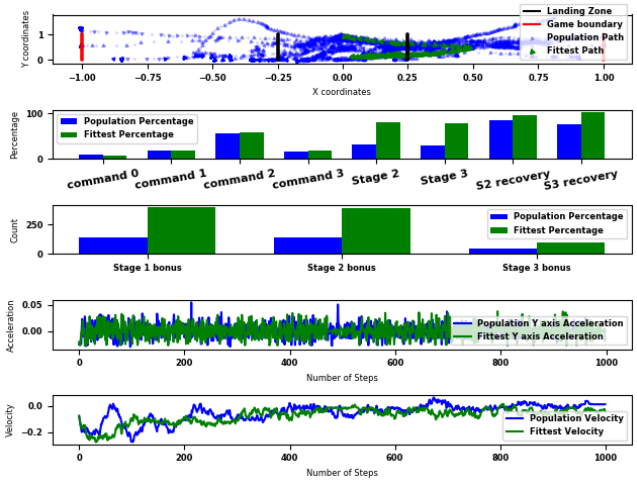

Figure 6: ES LSTM evolved by incremental fitness

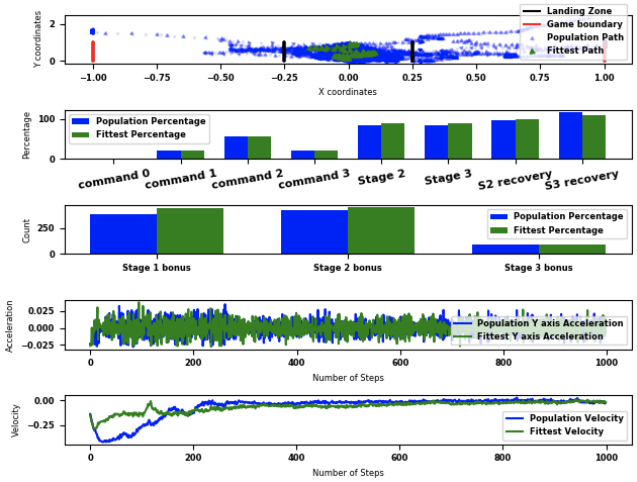

Figure 7: ES RNN evolved by incremental fitness

The behaviour of the evolutionary strategy algorithm with LSTM and RNN network is displayed in Figure 6 and Figure 7 respectively. The agent figured out how to change its flight path towards the landing point while kept its velocity and acceleration in control. The algorithm was run for 6,000 trials (60,000 evaluations) with LSTM and RNN.

The accumulated stage 1 bonus of the ES RNN and LSTM was far higher than NEAT RNN algorithm's while stage 2 and stage 3 bonus were similar. However, note that while the behaviour is complex, the networks actually finds a loophole in the fitness function. This trick was never landing and slowly tilting up and down near the landing area, which cheats the fitness function to think the agent is performing a landing and earns points. The behaviours of ES LSTM and ES RNN were similar. However, ES LSTM was utilising command 0 more than SS LSTM, NEAT RNN and ES RNN algorithms. Also, ES RNN algorithm's population wise fitness was far higher than ES LSTM. This might indicate existence of a wider optimum at a lower dimension.

	Stage 1 Average	Stage 2 Average	Stage 3 Average	Peak Fitness
SS GA LSTM (Population)	-244.37± 125.36	127.51±14.37	251.42±29.74	262.20±12.25
NEAT RNN (Population)	2.25±0.71	3.82±1.54	3.33±16.24	127.82±36.69
ES LSTM (Fittest)	212.21±28.23	783.92±78.91	1674.06± 119.85	1600.25± 1164.59
ES LSTM (Population)	65.38±13.42	332.55±123.18	330.62±171.42	471.68±446.78
ES RNN (Fittest)	167.06±30.69	1509.56±83.46	1662.15±80.90	2127.93± 1255.64
ES RNN (Population)	89.29±15.46	495.85±101.09	481.60±368.68	832.60±549.33

Table 1: Fitness Evaluations

To compare the performance of the algorithms more generally, Table 1 display average fitnesses of ES LSTM, ES RNN, NEAT RNN and SS GA algorithms over 60,000 evaluations with five random restarts. While ES RNN, ES LSTM and SSGA had similar fitness values, NEAT RNN had lower fitness scores. NEAT algorithm alters the composition of the network during the process of evolution and it starts from one hidden layer which has one node. While this attribute enables the population to produce more unique individuals, it increases the time of convergence. However, ES was the fastest algorithm in both network types and reached a better optimum corresponding to a qualitatively better solution. Even though ES LSTM had lower average population fitness, fittest individuals of both ES RNN and ES LSTM had similar scores.

However, while the ES clearly develops better more dynamic behaviour than the SSGA, as noted in the introduction, one of the benefits of the SSGA is that as good individuals remain in the population and are re-evaluated, good individuals are tested on many different configurations of the task which increases robustness. As the ES works in a different manner, it is not clear whether this robustness accrues in as direct a way as it does for the SSGA. To test for dynamic stability, we thus decreased the left thruster power to 35% of its original power. This resulted in the following figures.

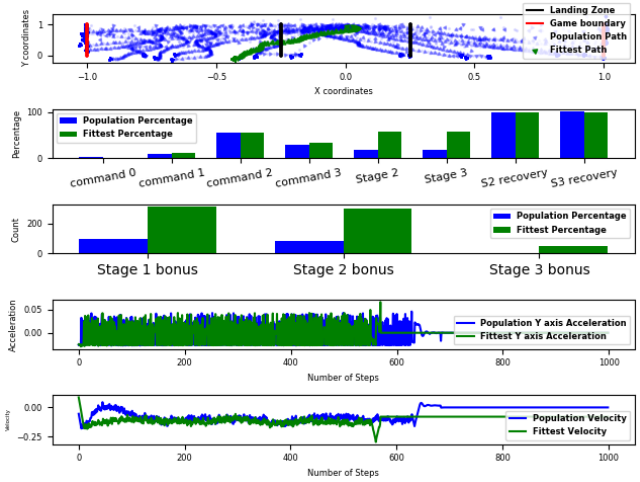

Figure 8: NEAT behaviour while left thruster was crippled

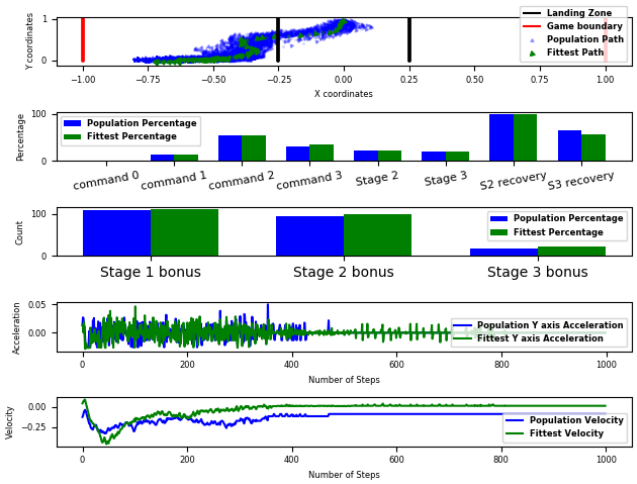

Figure 9: SS GA behaviour while left thruster was crippled

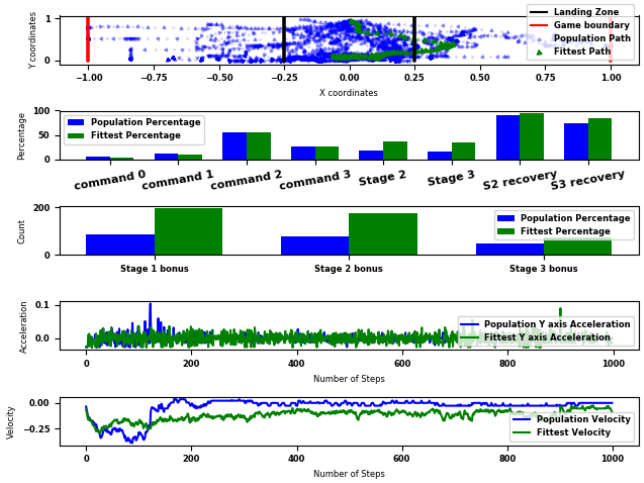

Figure 10: ES LSTM behaviour while left thruster was crippled

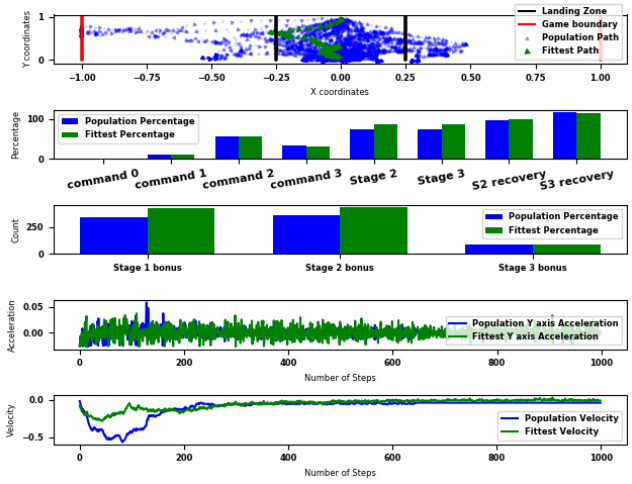

Figure 11: ES RNN behaviour while left thruster was crippled

Overall all algorithms used more Command 3 to compensate the loss of left thruster power (Figures 8, 9, 10,11). NEAT lost most of its control over the agent (Figure 8). SSGA performed similar behaviour as it had before the manipulation (Figure 9). However, when Figures 4 and 9 are compared, the velocity pattern of SSGA can be seen to be changed, indicating it is having a hard time controlling its velocity. In contrast, the ES LSTM and RNN were mostly unaffected (Figure 10, 11). Thus the ES has indeed developed robustness exhibiting homeostatic behaviour even to perturbations not experienced during evolution.

Summary and Discussion

In this experiment, LSTM and RNN networks were evolved using an incremental fitness function to solve a complex controller task: the Lunar Lander game. The steady state algorithm with LSTM was found to be less useful when the landscape was complex and noisy and found the local optimum faster than the NEAT algorithm with RNN network, as the faster converge rate of the population decreased the chance of finding other possible optima. The NEAT algorithm with RNN had a good balance between promoting diversity and was found to produce decent results. However using an ES with LSTM or RNN network was the fastest network and produced adaptive and dynamically stable behaviour even exploiting the fitness function to hover for a long time.

Overall the incremental fitness function was able to eliminate the issue of a local optimum in the fitness space both for a standard GA and for the ES. In addition increasing the complexity gradually helps the population to learn faster perhaps by reducing population convergence. Since the solution space of LSTM networks were more complicated (each neuron has eight weights excluding the bias weights) than RNNs, optimising LSTM networks is a harder task. However, reducing the complexity of the fitness landscape using incremental fitness function showed that LSTM networks can be evolved as fast as RNN networks.

The question of which type of memory in the LSTM networks was beneficial in this task, or whether different gates of the LSTM network require different evolution techniques, is not yet answered. To understand this, we would need to know more about why the LSTM network's behaviour is different to the RNN's. Since LSTM network cells are advanced versions of RNN cells, during the optimisation process they could be turned into RNN's by constraining the parameters correctly. In order to explore how the more complex memory might be valuable, the LSTM network can be initialised with parameters that make it act as an RNN or that allow different memory types only. By analysing the final evolved networks and seeing how much they vary from the standard RNN, the benefits that the different memory types of the LSTM bring to the table can be observed.

However, there are many other factors that affect the LSTM network's performance, such as network size, activation function and population size. The effect of these factors are not yet explored. A more flexible evolutionary method which explores these variables combined with ES algorithms, or a NEAT type process, might boost its performance and produce better results as suggested by Rawal Miikkulainen (2016) who showed that an LSTM network can be evolved using NEAT.

References

Bakker, B. (2001). Reinforcement Learning with Long Short-term Memory. Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic.

Barlow, G., Oh, C., and Grant, E. (2004). Incremental evolution of autonomous controllers for unmanned aerial vehicles using multi-objective genetic programming. In IEEE Conference on Cybernetics and Intelligent Systems 2004. volume 2, pp. 689–694.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Zaremba, W. (2016). OpenAI Gym. CoRR.

Di Pietro, A., While,. L. and Barone, L. (2004). Applying evolutionary algorithms to problems with noisy, time-consuming fitness functions. Proceeding of Congress on Evolutionary Computation, 19-23 June 2004, pp. 1254-1261 10.1109/CEC.2004.1331041

Gers, F.A., Schraudolph, N.N. and Schmidhuber, J. (2002). Learning precise timing with LSTM recurrent networks. Journal of Machine Learning Research (JMLR), 3:115–143.

Pramanik, S. and Setua S. (2014). A steady state genetic algorithm for multiple sequence alignment. Proceedings of the 2014 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2014. pp. 1095-1099. 10.1109/ICACCI.2014.6968251.

Rawal, A. and Miikkulainen, R. (2016). Evolving Deep LSTM-based Memory Networks using an Information Maximization Objective. The Genetic and Evolutionary Computation Conference (GECCO'16), July 20-24, Denver, Colorado, USA.

Salimans, T., Ho, J., Chen, X., Sutskever, I. (2017). Evolution Strategies as a Scalable Alternative to Reinforcement Learning. ArXiv e-prints, 2017.

Stanley, K.O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. IEEE Trans. Evol. Comput. 10(2):99–127.

Sutskever, I., Vinyals, O., Le, Q. V. (2014). Sequence to sequence learning with neural networks. Proceedings of the 27th International Conference on Neural Information Processing Systems.

Syswerda, G. (1991). A study of reproduction in generational and steady-state genetic algorithms. In Gregory J. E. Rawlins, editor, Foundations of Genetic Algorithms. volume 1, pp 94-101, Morgan Kaufmann Publishers.

Whitley, D. (2001). An overview of evolutionary algorithms: practical issues and common pitfalls. Information & Software Technology, 43:817-831.