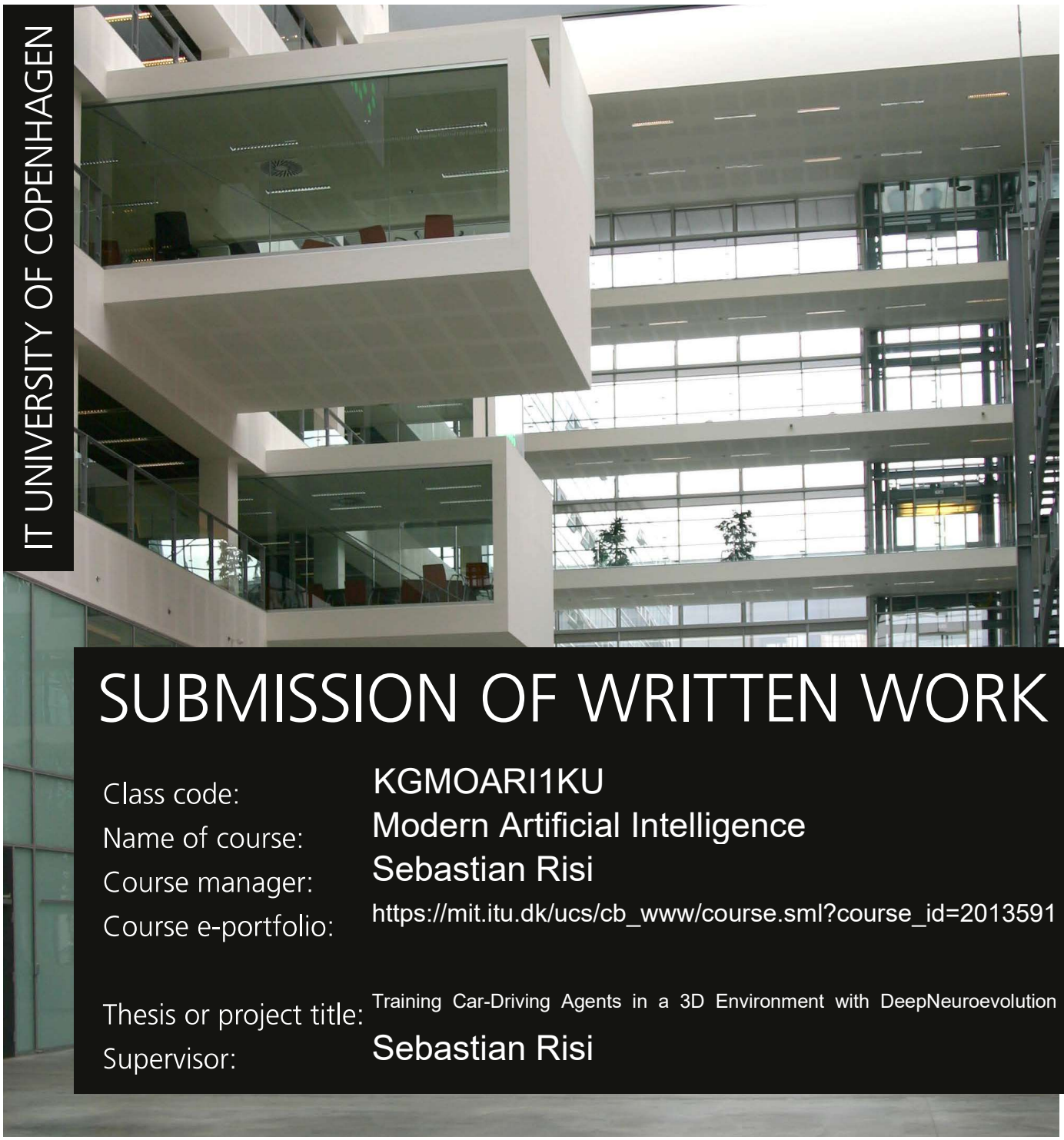
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:	KGMOARI1KU
Name of course:	Modern Artificial Intelligence
Course manager:	Sebastian Risi
Course e-portfolio:	https://mit.itu.dk/ucs/cb_www/course.sml?course_id=2013591
Thesis or project title:	Training Car-Driving Agents in a 3D Environment with DeepNeuroevolution
Supervisor:	Sebastian Risi

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Emilio Capo	05/03-1995	emca@itu.dk
2.	David Oppenberg	03/09-1991	daop@itu.dk
3.	Winfried Baumann	06/11-1993	wiba@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk
8.			@itu.dk

Training Car-Driving Agents in a 3D Environment with Deep Neuroevolution

Winfried Baumann
wiba@itu.dk

Emilio Capo
emca@itu.dk

David Oppenberg
daop@itu.dk

***Abstract*— The task of image-input based game controllers has been popularized most notably by the paper "Playing Atari with Deep Reinforcement Learning", in which agents are trained to interpret an image input with Reinforcement Learning and stochastic gradient descend. Deep Neuroevolution has proven to be a competitive alternative to gradient-based training of Convolution Neural Networks, albeit only in the domain of 2D games. We therefore propose two image-based controllers exploiting two different CNN architectures, which have been trained with Deep Neuroevolution to navigate a virtual car in a 3D environment. The results show that, even if the agents did not manage to learn the task completely, this approach is promising when applied to moderate-sized networks. However, there is large space for improvement, since our work was significantly limited by computational resources and time.**

I. Introduction

The use of video games for experimenting machine learning techniques for diverse purposes has received great attention in the last decade [1]. Among the different areas of application is Non-Player Character (NPC) behaviour modelling, which focuses on modelling agents that are able to play a game well. The main learning paradigm used for this approach is reinforcement learning, which is based on a reward system informing the agent about how effective a given action is in a determined situation, or state. This state is usually represented by the game visual as a raw pixel image. This approach, contrarily to supervised learning, does not require any human-recorded data. Moreover, the agent is able to learn the target task even without having a model of the environment, which in most games is very complex, but simply exploring it multiple times and storing information about each run, or episode. The efficiency of this approach, from its first applications in *Mnih et al.* [2], has been increasingly refined showing impressive results in games like Go [3] and the classic Atari games [4].

Another way to model the behaviour of an agent is the evolutionary paradigm, which stimulates learning by selecting the best candidates out of a finite population of samples. However, this approach requires a consistent amount of time, which grows proportionally to the space of the different characteristics, or chromosomes, that each candidate can assume. The very large space of parameters that represent the main image-input based computational model, which is Convolutional Neural Networks, has often discouraged a direct evolutionary approach. However, the recent paper by Uber AI Labs [5] shows that evolution is a competitive alternative to reinforcement learning for some Atari games. Our objective is to see how the same approach performs in a 3D environment, applying it to the task of driving a car without going off road.

II. Background

A. Deep Learning

Deep Learning is a novel Artificial Intelligence technique, based on the concept of Neural Network, a computational model resembling the human brain [6]. It consists of a series of elements called "neurons". These neurons are organized in layers, which sequentially exchange information. In the case of a fully-connected layer, each neuron receives all the outputs from the neurons of the previous layer, elaborates them through an activation function and forwards its outputs to the following layer.

Compared to "Shallow" Neural Networks, which are composed of three layers (input-hidden-output), Deep Neural Networks feature a larger number of hidden layers [7, Chapter 6]. This generally allows for higher abstraction capabilities, meaning more complex tasks can be learned, but this comes at the cost of requiring a larger amount of data to learn such tasks.

B. Convolutional Neural Networks

Convolutional Networks (CNN) are a type of Deep Neural Networks which are suitable for computer vision problems, due to their ability to detect spatial relation within data [8]. The concept of CNNs, as well as their architecture, are based on the model of the mammal visual cortex proposed by Hubel and Wiesel. Their model distinguishes between simple cells, responsible for feature extraction, and complex cells, which combine local features of spatial proximity [9].

Similarly, the layers of CNNs consist of convolutional filters that extract low level features such as edges and curves, which are then abstracted to a higher level representation throughout the network. Instead of using a handcrafted feature representations, CNNs use a 2D matrix as an input. Common architectures for CNNs consists of multiple convolutional layers, each of them followed by a non-linear activation function and a pooling layer, which downsamples their output. The output of the final convolutional layer is flattened into a one-dimensional shape and given to a fully-connected dense layer, which performs the final computation.

CNNs match human performance on benchmark tasks such as the recognition of handwritten digits or traffic signs [10].

C. Genetic Algorithms and Neuroevolution

Genetic Algorithms are optimization techniques inspired by biological evolution [11]. After defining the parameters, or "chromosomes", of the population to evolve, each candidate is generated by drawing the values of their chromosomes out of some defined probability distribution. The candidates of the population are then evaluated on the task they have to learn according to a fitness function. The best candidates are allowed to proceed to the next generation and to produce offspring according to different techniques, such as mutation or crossover, while the worst candidates are usually eliminated.

The application of genetic algorithms to Neural Networks takes the name of NeuroEvolution (NE). This technique usually takes as chromosomes the weights of the Neural Network, its topology or both. A survey of this technique [12] highlights one of the open issues of the field, which is learning from high-dimensional/raw input, that we had to face in this work. As suggested, we scaled the image down for our visual controller. A famous application of NE is NeuroEvolution of Augmenting Topologies (NEAT) [13], where the network weights and topology are evolved together.

III. Previous works

The DeepMind research team pioneered the concept of training a convolutional neural network with raw pixels input, by introducing their paper "Playing Atari with Deep Reinforcement Learning" in 2013 [2]. Instead of using handcrafted feature representations, the DeepMind team uses Deep Reinforcement Learning by approximating a Q-function, in order to learn a control policy for stochastic 2D Atari games. With regards to our experiment, the paper offers two main takeaways:

- *Network architecture*: The network consists of 3 convolutional layers with 32, 64, and 64 filters respectively. The convolutional layers use 8 8, 4 4, and 3 3 filters with strides of 4, 2, and 1, respectively The output of the last convolutional layer is flattened and connected to a hidden layer with 512 units followed by a ReLU activation function.The deep layer is finally connected to the output nodes. As the Uber paper refers to this architecture as well, we decided to adapt the architecture for our model (architecture B; see chapter IV).
- *Perceptual aliasing*: As isolated frames can be ambiguous, the paper proposes to define a sequence of 4 consecutive frames as a state. In the context of our game, it is difficult to infer information about speed or direction of motion from an isolated image. Therefore, we adapt the solution of treating 4 stacked consecutive frames as one state (architecture B; see chapter IV).

The paper published by the Uber AI Labs in 2017 [5] introduces the idea that genetic algorithms are a competitive alternative to gradient-descend based training methods in the domain of deep reinforcement learning problems. The non-gradient-based evolutionary algorithm performs well on benchmark problems such as Atari 2600 and Humanoid Locomotion in the MuJoCo simulator. It indicates that GAs perform well on the scale of deep neural networks with more than 5 million parameters. Apart from the previously mentioned network architecture, we adapted the following design decisions of the GA implementation explained in the paper.

- *Truncation Selection*: The 10% best performing individuals of a generation become the parents of the next generation.
- *Mutation by additive gaussian noise*: The selected parents are duplicated 10-fold. Out of these folds, 9 are mutated by adding noise sampled from a truncated gaussian distribution. The tenth fold remains unaltered, a process called elitism.

It is crucial to mention that the described experiment operates on a much larger scale, as shown by the parameter table. The population sizes range from 1000 for the Atari problem, to 12.500 for the Humanoid Locomotion. Due to limited computational resources, we are not able to match those parameters.

IV. Methodology

A. Environment

We built our environment using Unreal Engine 4 [14], a game engine which offers a racing game template with some built-in vehicles. Physics is completely taken care of by the engine, thus we mainly focused on building the assets we needed to setup the simulation.

First, we built a tool that allowed us to quickly create and modify tracks [15]. This tool renders the track based on a spline, whose points represent the edges of the track pieces. Each spline segment is thus a piece of the track. Each piece has different properties that can be set, among which is the possibility to dynamically add guardrails, which will turn out to be an important part of our evolution algorithm. It is also possible to adjust length, width and rotation of the track pieces, which helped in shaping the tracks more precisely. The tool is built so to allow an easy implementation of a random track generator, a feature we briefly considered in order to improve the generalization capabilities of agents, but we later discarded this option to concentrate our efforts on a single, cleverly designed track.

The track we used (Fig. 1) was designed to sequentially teach the agent different sub-tasks. The first challenge it is presented with is a right turn. We then put a short straight road to penalize agents who just continuously turn right. After the second right turn, the next challenge is a left turn. We then added a sharp right curve to make sure that the agent is able to steer continuously. A long straight road then serves the purpose to teach the agent to accelerate in speed

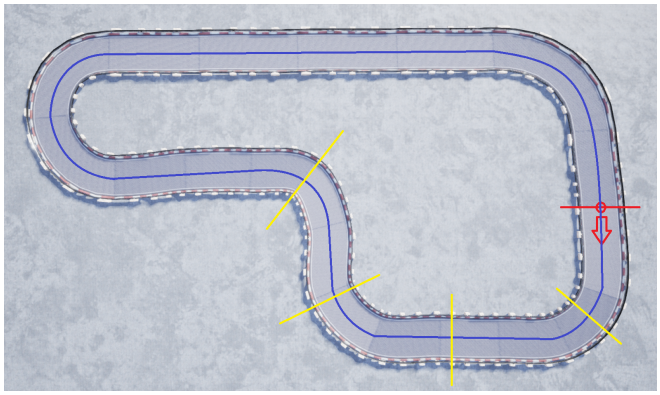


Fig. 1. The track used for evolution. The red line represents the goal. The correspondent red dot represents the point where the candidates start the evaluation. The direction is indicated by the red arrow. The blue line along the track represents an approximation of the spline we used to compute the fitness of each candidate. The yellow lines approximately show the distance corresponding to fitness values 0.1, 0.2, 0.3 and 0.4, respectively, to better contextualize the obtained results.

and eventually brake before the last turn.

We then setup the visual of the vehicle. The virtual camera was put on the bonnet, aligned with the centre, so that the agents have a full visual of the road. Moreover, to allow the simulation of more than one agent at the same time, we set the camera not to render the other vehicles on the scene and disabled collisions between them. This way, each agent has the illusion to be the only one running on the track. This gimmick allowed us to speed-up the evolution process.

B. Network Architectures

We built our network using the UnrealEnginePython plugin [16] and the tensorflow-ue4 plugin [17] available for Unreal Engine 4, based on the machine learning framework TensorFlow [18]. Specifically, we built it at a high level using the Keras module. We tried two different network architectures for our Deep Convolutional Neural Network, in order to vary the size of the parameters space.

We started from a simple architecture (A), a sequential model with two convolutional layers, whose output is flattened and then fully-connected to the output layer. The second architecture (B) features an additional convolutional layer and an additional dense layer before the output layer. To keep our network consistent with the one proposed in [5], we decided not to use any pooling layer.

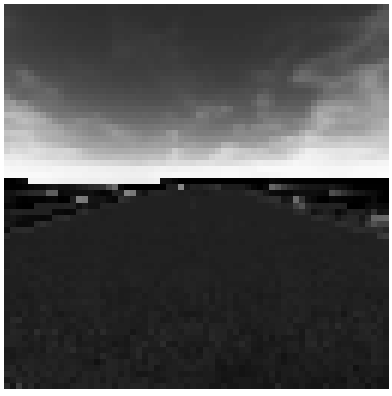


Fig. 2. An example of input for the network.

Architecture B receives four 64x64 pixels grayscale images as an input, with normalized pixel values. These images correspond to the last four frames received. This way the agent is able to derive velocity and acceleration information from its input. Architecture A only uses one image instead. The input is further reshaped to align with the batch format of the Keras implementation.

In architecture A, the first convolutional layer features 64 8x8 filters, a stride of 1 and a ReLu activation function. The second convolutional layer only differs by the number of filters, which are 32. In architecture B, the first convolutional layer features 32 8x8 filters, a stride of 4 and a ReLu activation function. The second convolutional layer features 64 4x4 filters, a stride of 2 and a ReLu activation function. The third convolutional layer features 64 3x3 filters with a stride of 1 and a ReLu activation function. The additional dense layer receives the flattened output of the final convolutional layer, applies a ReLu activation function and outputs 512 values. Each of these values is then fully-connected to 5 outputs applying a Sigmoid activation function. The 5 outputs are thus defined in the range [0, 1] and interpreted as probability values. These output values describe:

- Right direction;
- Left direction;
- Forward motion;
- Backward motion;
- Handbrake pulled.

The five outputs are mapped to the five keyboard keys controlling the car. For the activation values to translate into these boolean control inputs, we defined hard-coded thresholds. The first two values are interpreted respectively as the probability to turn right or left. These two values are compared, taking into consideration the highest one. If this value is greater than 0.5, then the car will turn into the correspondent direction, otherwise it will keep a straight direction. The third and fourth values are interpreted respectively as the probability to accelerate or decelerate. Again, the two values are compared and if the highest one is greater than 0.5, then the car will perform the related action, otherwise it won't move. Finally, the fifth value is interpreted as the probability to pull the handbrake. If that value is greater than 0.5, then the handbrake will be pulled.

C. Genetic Algorithm

Our genetic algorithm has been implemented as two separated parts, one using Unreal's blueprint visual scripting [19] and the other one using a python script. The blueprint script constitutes the genetic algorithm at a high-level, managing the different evolution phases as well as computing the candidates' fitness and saving the information as a csv file

for visualization. The python script, on the other hand, works as a container for a candidate, knowing all the operations that can be performed on it. It knows the chromosomes of each candidate and is able to switch between them when it receives the related call by the blueprint. This calls are managed through an interface which offers the following methods:

- *initialize(id)* - Loads the chromosomes information of the candidate with the correspondent id;
- *onJsonInput(jsonInput)* - Receives the $jsonInput$, which consists in the input frame(s) in json format [20] and returns the prediction made by the network. This prediction has already been processed, thus from the 5 outputs of the network, the blueprint receives three values:
  - $right$: [1, 0, -1] = [turn right, no steering, turn left]
  - acc : [1, 0, -1] = [accelerate, no input, brake]
  - $handbrake$: [True, False] = [pull, don't pull]
- *mutate(id)* - Apply mutation to the candidate with the correspondent id.

This way, the genetic algorithm blueprint does not need to know about the genome structure and can be used independently of the population. Moreover, since the data related to the latest generation is always fully stored by the python script, it is possible to stop the evolution and resume it later. This allowed us to recover from some crashes we sometimes experienced.

The beginning population's chromosomes, which are their weights and bias vectors, are randomly sampled from a truncated Gaussian distribution. Each candidate is evaluated once on a 33 seconds performance on the track. The fitness function measures the maximum distance that the car covered along the spline of the track, within the time limit, divided by the total distance. It is thus the percentage of the track explored by the agent.

Our algorithm performs truncation selection, in which the top 10% (which in our case corresponds to 30 candidates) best performing individuals of each generation are selected as the parents of the new generation. The selected elite is duplicated 10-fold, ensuring to keep the population size constant. One fold remains unaltered, while the other 9 folds are subsequently mutated. Mutation occurs by adding noise sampled from a truncated Gaussian distribution both to the weights and to the bias vector. The new population is then fully evaluated.

Initially, we enabled guardrails on the whole track, in order to avoid the agent to fall off. Moreover, the engine physics would slow the agents down on collision, penalizing those who would collide with the guardrails. However, by running some quick experiments, we noticed that some agents used the guardrails to bounce, rotate the car in the curve direction and then start driving backwards. Some other times, the agents managed to slide along the guardrails and make the turn. To avoid these cheating behaviours, as well as to speed up evolution, we decided to stop the evaluation when colliding with the guardrails. It must be noted, however, that using this highly punitive solution probably made the evolution process harder, since both minor and major collisions lead to the same penalty.

Additionally, we decided to also stop the evaluation when the agent would stay still for 1 second. This allowed to speed up the evaluation in some cases, such as in generation 0, where most randomly initialized agents would not move at all.

Finally, thanks to our setup of the agents' camera, we were able to simulate three agents at the same time and still keep each evaluation independent from the other. A higher number of agents resulted in quite a low rendering framerate, which would impact on the experiments. Even with three simultaneous agent being evaluated, we were still able to significantly reduce the time required to evaluate the population.

V. Experiments

We run two experiments, one for each architecture. In both cases, we used a population of 300 candidates, whose chromosomes were initialized using a truncated normal distribution with mean 0 and standard deviation 0.05, generating weights in the interval $[-0.1, 0.1]$. Each of them was evaluated on a single run of the track. Mutation was applied by adding random noise sampled from a normal distribution with mean 0 and standard deviation 0.005. In both cases, the populations were evolved for 305 generations. The results of the experiments are shown in Fig. 3 and Fig. 4.

It is important to remark that we had different performance issues while we were running the evolution process. Framerate drops and seldom crashes surely had an impact on the experiments, causing interruption in the evaluation and resulting in some candidates being evaluated poorly. However, in the case of architecture A, this did not prevent the agents from improving on the task.

VI. Results

Architecture A showed more promising results in a smaller amount of generations, as many agents already learned how to turn right around generation 200, whereas the candidates using architecture B were still struggling around generation 300. We associate this outcome with the increased amount of parameters that architecture B has, due to its additional input frames, convolutional layer and number of filters, as well as its additional dense layer.

The results on architecture A show that the agents progressively change their behaviour. The very first generations contain a lot of agents that just don't move at all. These are quickly eliminated by those who start driving straight. At this point, the process of learning to turn right is quite slow, because most agents didn't learn to tune their

Fig. 3. A graph showing a fitness boxplot for the elite of each generation (best 30 candidates, 305 generations in total). It is possible to notice that around generation 200, when agents learned to turn right, there was a positive change of trend in the evolution process. The graph suggests that the agents could further improve their ability to perform the task.

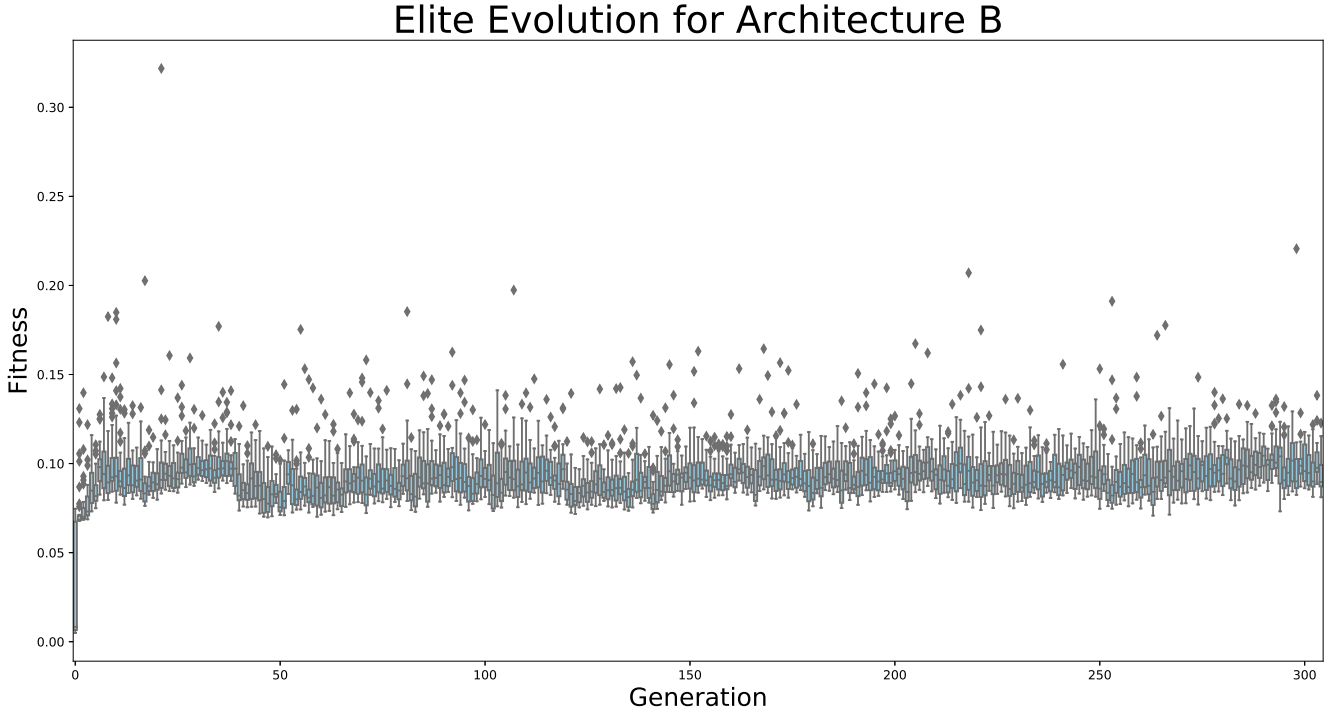


Fig. 4. A graph showing a fitness boxplot for the elite of each generation (best 30 candidates, 305 generations in total). The graph shows there was no relevant improvement in the agents' performance. This could be due to the larger parameters space compared to Architecture A.

speed and are not able to make the turn at a high speed. As generations pass, it is possible to notice that agents that move slower are those preferred by evolution, as they manage to make the first turn. The best agent managed to do both the first and the second right turn, showing a fitness of over 0.4.

It is hard to compare our result to those of Uber AI Labs [5], since our work was realized on quite a smaller scale. Our approach using the architecture they proposed (architecture B) did not bring to positive results given the population size that we use and the generation we evolved it for.

A video showing the performances of part of the final populations for both architectures is available on `youtu.be/TdbkxcSYNW0`.

VII. Conclusion

We evolved a population of car-driving agents for more than 300 generations, trying two different network architectures. The results suggest that, much like what is expected by genetic algorithms, the simpler network (architecture A) showed better results having a smaller chromosomes space to explore. However, none of the two architectures managed to completely learn the task.

It must be noted that, due to computational and time limitations, our work can be greatly improved in different ways. Results suggest that agents based on architecture A are actually learning by showing a slow increase in the fitness measure we defined. It would thus be interesting to run the experiment longer to see how continuous this learning process is.

Beyond running for a limited number of generations, we could also only afford one evaluation per candidate. This resulted in a variable highest fitness throughout generations, as the elite did not always perform coherently from one generation to the other. Averaging over different evaluations could help in finding the actual best candidates and speed up the learning process.

For this experiment, we used a single instance of Unreal Engine running three agents at the same time. A significant improvement in evolution time could be using more instances of Unreal Engine, potentially accessed through a server. This would however add complexity in managing synchronization between the different processes.

It could also be argued that the genetic algorithm we used was quite simple, using blind mutation over a large number of parameters. One interesting direction of improvement would be identifying a more guided evolution procedure, which could either use some clever mutation or crossover, or even reduce the number of the elite to allow a deeper exploration of the best chromosomes.

On the other hand, our algorithm was very punitive in case of idling or collision with the guardrail, again a measure that was introduced also to speed the evolution up. A more clever punishment would be, for instance, only stopping those agents who hit the guardrail with a certain direction, using dot product between the direction of the car and the normal of the guardrails to filter out agents that hit it directly, as well as tracking frequency and duration of collisions.

Finally, once found a good setup to learn the task, it would be interesting to compare evolution on a single track to evolution using multiple tracks, to see if agents trained using the first technique already gain good generalization capabilities.

References

[1] G. N. Yannakakis and J. Togelius, "A panorama of artificial and computational intelligence in games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 4, pp. 317–335, Dec 2015.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–, Oct. 2017. [Online]. Available: http://dx.doi.org/10.1038/nature24270

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[5] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *CoRR*, vol. abs/1712.06567, 2017. [Online]. Available: http://arxiv.org/abs/1712.06567

[6] R. Lippmann, "An introduction to computing with neural nets," *IEEE ASSP Magazine*, vol. 4, no. 2, pp. 4–22, Apr 1987.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[8] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14539

[9] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *Artificial Neural Networks – ICANN 2010*, K. Diamantaras, W. Duch, and L. S. Iliadis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–101.

[10] D. C. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," *CoRR*, vol. abs/1202.2745, 2012. [Online]. Available: http://arxiv.org/abs/1202.2745

[11] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[12] S. Risi and J. Togelius, "Neuroevolution in games: State of the art and open challenges," *I E E E Transactions on Computational Intelligence and A I in Games*, vol. 9, no. 1, pp. 25–41, 2015.

[13] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. [Online]. Available: https://doi.org/10.1162/106365602320169811

[14] Epic Games, "Unreal Engine," *Online: https://www.unrealengine.com*, 2007.

[15] "Using splines and spline components, live training, unreal engine." [Online]. Available: https://www.youtube.com/watch?v=wR0fH6O9jD8

[16] J. Kaniewski, "Unrealenginepython." [Online]. Available: https://github.com/getnamo/UnrealEnginePython

[17] J. Kaniewski, “tensorflow-ue4.” [Online]. Available: https://github.com/getnamo/tensorflow-ue4
[18] Google Brain team, “Tensorflow.” [Online]. Available: https://www.tensorflow.org/
[19] Epic Games, “Blueprints visual scripting.” [Online]. Available: https://docs.unrealengine.com/en-us/Engine/Blueprints
[20] “json.” [Online]. Available: https://www.json.org/