Steven Smits
s4232763

Tjalling Haije
s1011759

Tido Bergmans
s4373561

Kai Standvoss
s4751744

Four Horsemen — Final Assignment
Natural Computing

From Genes to Behavior: Solving Super Mario Bros — Twice

Abstract

Modeling an agent that is able to form policies such that it can navigate through and interact with a high-dimensional environment remains to be a assiduous goal of Artificial Intelligence. This study models such an agent using two strategies: Neuroevolution of Augmenting Topologies (NEAT) and Deep Q-learning. The two techniques are compared in their performance on Super Mario Bros, a game with a complex environment. It was found that both an agent with policies constructed by NEAT and Deep Q-learning outperform an agent using random policies to interact with the game environment. Further, policies constructed by NEAT and Deep Q-learning did not significantly differ in their performance playing the game. Thus, the results show both NEAT and Deep Q-learning to be promising models for developing agent cognition in terms of policies to navigate and interact with a high-dimensional environment.

Contents

1 Introduction

Teaching an autonomous agent to navigate through and interact with an environment is an important landmark in the fields of robotics, control theory and AI. Such an agent should have an conceptual understanding of the goal, the environment and the possible interactions between itself and the environment. An agent that fulfills these requirements can be deployed to execute tasks ranging from simple physical labor to logistics and strategic planning in an autonomous manner. Concrete examples include autonomous vehicles [1] and robotic surgery [2].

A remaining challenge is to provide the agent an appropriate representation of the environment based on low level sensory input. This representation should exclude noise and irrelevant features so the agent can use relevant features to decide what actions to deploy. The performance of an agent relies directly on the quality of the representation [3], but sensory input is often noisy, multi-modal and high-dimensional. Supervised learning methods therefore require large training datasets to train agents in different environments and different situations. For unsupervised learning this problem is different, and the main challenge is to create a higher level feature representation of the environment from low level sensory input.

In this study we trained an agent to navigate in a simple environment, finding it's way to a destination while avoiding problematic encounters with other agents in the environment. We used two different machine learning strategies to train an agent to play levels of Super Mario Bros, a role playing game in which the player has to get Mario to the end of the level, while grabbing bonuses and avoiding dangerous enemies. Both strategies rely on the processing of visual input to select an appropriate action for the agent. The two dimensional Mario environment is less detailed and less noisy than a three dimensional real world environment, but the principle of extracting higher order features from low level input remains the same.

The strategies that were implemented both use reinforcement learning, a method based on the general principle of rewarding good behavior and adapting behavior to increase the reward. Reinforcement learning is a form of unsupervised machine learning and only needs an environment, a set of possible actions and a reward function in order to start the learning process. It is inspired by how learning works in animals, which are also agents interacting with an environment. One of two straightforward examples of reinforcement learning in animals is Pavlovian conditioning: model-free learning through experience, and learning the predictive value of environmental input. The other is instrumental conditioning: where an animal can learn an arbitrary policy to obtain rewards [4].

There are many different techniques for reinforcement learning, such as Q-learning, deep reinforcement learning and a variety of evolutionary strategies. The first strategy we employed on the Mario game is Deep Q-Learning (DQL) [3]. Here a deep neural network processes low level visual input and extracts higher level features to get an abstract representation of the environment and generate an appropriate action. This action changes the state of the environment leaving the agent in a better or worse situation than before. This change is quantified as a reward or score which is used as feedback for the neural network. The neural network then uses a gradient method to change its weights and

improve the agent's behavior. This particular method was previously applied to a set of Atari 2600 games, a reinforcement learning testbed with diverse and challenging tasks, with unprecedented performance [5].
The second strategy combines reinforcement learning with an evolutionary strategy [6][7]. In this case the neural network is not updated using a gradient method based on the reward, but rather the reward serves as a fitness score to determine what networks from a pool of networks be discarded or selected for the next generation. After evaluation of the fitness, the best networks in a pool are reproduced, recombined and mutated into a new pool of networks which will again be evaluated, each generation increasing the fitness and thus converging towards the optimal behavior.
In particular, we used NeuroEvolution of Augmenting Topologies (NEAT), an algorithm that not only changes network weights during evolution, but also network topology. This is contrary to more traditional neuroevolutionary strategies, where the evolutionary algorithm searches for the best network weights, but where the network topology is predetermined and remains fixed throughout the process. NEAT provides a way to generate neural networks in an unbiased way, starting out with networks with a minimal topology and introducing more complex topological features only if it improves performance. The models and programming code can be found on our GitHub.[1]

2 Methods

2.1 Super Mario Bros

Super Mario Bros (SMB) is a 1983 platform video game by Nintendo. In the game the player controls the character Mario, an Italian plumber, who needs to rescue his kidnapped girlfriend Princess Peach. In order to do this, Mario has to cross several side-scrolling levels filled with enemies, obstacles, points and bonuses. Some examples of levels with different themes are shown in Figure 1. The player's job is to get Mario to the end of each level, while overcoming the obstacles, avoiding the enemies and collecting points and bonuses. There are many different enemies that move in different patterns and can be beaten in different ways, e.g. by jumping on top of them. There are gaps in the floor that have to be jumped over, bonuses that yield superpowers and obstacles that are too high to jump over. When Mario touches an enemy, falls into a gap or gets stuck, he dies and the level restarts.
During the game, the primary goal is to clear the level, a secondary goal is to gain a high score by killing enemies or collecting points. For this study, only the first level of the first world of Super Mario Bros was used.
The player controls Mario with only 6 keys; left, right, up, down, jump and dash. Since combination of actions are allowed the total set of allowed actions here employed was 14. The game resolution is 256x224 pixels, and most objects are placed in a 16x14 grid with cells of size 16x16. The game draws frames at a rate of 60 Hz. The low resolution makes this game suitable for AI training without needing an excessive amount of memory and computational time. This is comparable to the Atari 2600 games that have been

[1] `https://github.com/thaije/Natural-Computing/tree/master/project`

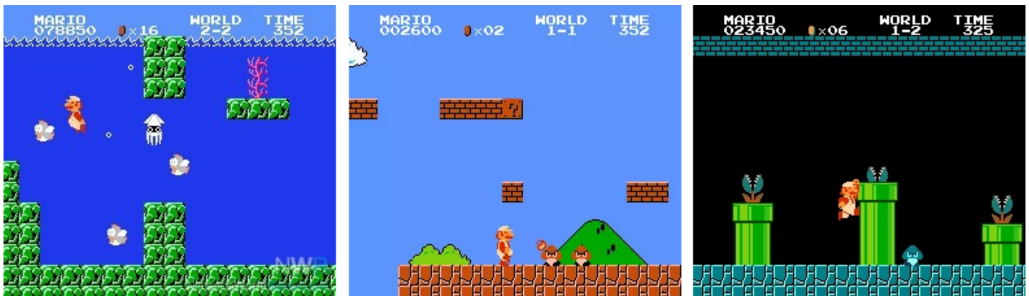
MARIO 078850 ×16 WORLD 2-2 TIME 352
MARIO 002600 ×02 WORLD 1-1 TIME 352
MARIO 023450 ×06 WORLD 1-2 TIME 325

Figure 1: Examples of different levels in Super Mario Bros.

extensively used for training autonomous agents [3] [5].
The game originally runs on the Nintendo Entertainment System (NES), but applications are available that emulate the system on a modern computer. To be specific, the Fceux emulator[2] was used. For training the agent we used the OpenAI Gym framework, a toolkit for developing and comparing reinforcement learning algorithms [8]. OpenAI Gym provides a game environment that can be used to train and benchmark agents, and provides functions for the agent to interact with said environment. The framework does not include the agent itself; only the infrastructure to train one.

2.2 Deep Q-Learning

2.2.1 Background

The first technique to tackle the Super Mario Bros games is called Deep Q-learning [3]. This method is a variant of the standard Q-learning approach used in reinforcement learning tasks. In reinforcement learning, an agent $\mathcal{M}$ interacts with an environment $\mathcal{E}$ in which $\mathcal{M}$ observes environmental states x_t, does an action a_t, and subsequently obtains a reward r_t. In this study $\mathcal{E}$ is the Super Mario Bros NES environment which emits a state x_t that is a matrix directly representing the screen as an image. Since in reality a task is poorly understood in only its current state, instead of single observations x_t sequences of state and action observations $s_t = \{x_0, a_0, ..., x_{t-1}, a_{t-1}, x_t\}$ are used as the observed state for the agent $\mathcal{M}$. Now, the goal of the agent is to maximize its future reward received by the environment based on the agents action responses to the states. Specifically, this means that the agent aims to maximize $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} \cdot r_{t'}$, where γ is a reward discounting factor that takes into account that future states and therefore potential rewards are stochastic. By evolving a function $\mathcal{Q}^*(s;a)$ that applies a policy π on the sequence of observations s_t, the agent is able to maximize the expected value of R_t. Thus the main function to estimate in Q-learning is:

$$\mathcal{Q}^*(s;a) = E_{s'}\left[r + \gamma \cdot \max_{a} \mathcal{Q}^*(s',a')|s,a\right] \tag{1}$$

Traditional Q-learning requires to learn all action reward values for all states separately, in an iterative procedure such that our estimation function $\mathcal{Q}_i(s,a) \rightarrow \mathcal{Q}^*(s;a)$ as $i \rightarrow \infty$. In a complex game such as Super Mario Bros this procedure becomes impractical as it

[2]http://www.fceux.com/web/home.html

doesn't generalize because not all states can be observed. The solution we use to this problem is using a neural network (Q-network) as a non-linear function approximator $Q(s, a; \theta)$, with θ being the the weights of the Q-network. This Q-network aims to minimize some loss function $\mathcal{L}_i(\theta_i)$, which in this study is the Huber loss defined as [9]:

$$\mathcal{L}_i(y_i, Q_i(s, a; \theta_i)) = \begin{cases} \frac{1}{2}(y_i - Q_i(s, a; \theta_i))^2 & \text{if } |y_i - Q_i(s, a; \theta_i)| \leqslant \delta, \\ \delta|y_i - Q_i(s, a; \theta_i)| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

where $y_i = Q^*(s; a) = E_{s'}\left[r + \gamma \cdot \max_a Q^*(s', a'; \theta_{i-1})|s, a\right]$ is the target for iteration i. This means that the target values depend on the weights of the Q-network itself as a consequence of reinforcement learning being an unsupervised learning technique. This loss can be minimized using a method for stochastic optimization called Adam that at each time-step uses s_t and r_t obtained from $\mathcal{E}$ [10].
The method so far described is just some realization of standard Q-learning. Deep Q-learning augments this method by using a deep convolutional multi-layered network that outputs a vector Q with action-values for each possible action given s_t, as such predicting the expected reward for each action from the given state. Furthermore, key features of deep Q-learning are experience replay and reward restriction. Experience replay stores the agent's $\mathcal{M}$ experiences as $e_t = \{s_t, a_t, r_y, s_{t+1}\}$ in dataset $\mathcal{D} = \{e_1, .., e_N\}$, with N being some finite number, out of which the Q-network can extract batches to learn from during each time step. Thus, before acting, the agent experiences replay on some random samples $e \sim \mathcal{D}$ drawn with uniform probability. This approach of experience replay prevents samples within mini-batches to be strongly correlated, thereby reducing the variance of each Q-network update. To elucidate, if the action that maximizes the reward within some connected time-sample mini-batch is dominated by an action that moves to the left, then it is easy for the Q-network to get stuck on only predicting going to the left. This is prevented by taking random samples to create mini-batches with a more diverge actions that maximize the reward. To further reduce correlation between replay experiences, a warm up might be done of the replay memory before the Q-network is trained on the replay experiences. During the warm up a random agent is used to traverse the level for a certain number of frames to prefill the replay memory with a diverse set of experiences. Finally, rewards should be clipped to some interval such that the gradients of the Q-network remain well conditioned and don't diverge too much.

2.2.2 Implementation

The agent The agent is defined as a model that at every time step t receives $\{x_t, r_{t-1}\}$ and consists of general policies, a memory, a Q-network and acts upon the input. The memory makes it so that the agent at any time has access to $\mathcal{D} = \{e_1, .., e_N\}$ with $e_t = \{s_t, a_t, r_y, s_{t+1}\}$ being stored experiences. When called to act based on the current state x_t, the agent first stores the reward r_{t-1} in its memory to the corresponding x_{t-1} such that the memory stacks $s_t = \{x_0, a_0, ..., x_{t-1}, a_{t-1}, x_t\}$ to its experiences. Next, the agent takes random batches from its memory for the Q-network to experience replay. Finally,, the agent has general policies to ensure a balance between exploration-exploitation. Specifi-

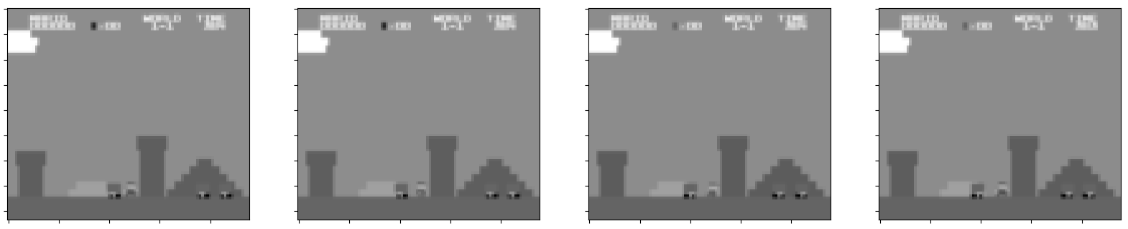

Figure 2: Example input for the Q-network. Four consecutive frames, chronological from left to right. Mario is located left of the large pipe, with one enemy on the left, and two enemies on the right side of the pipe.

cally, the agent will explore (i.e. execute random actions) with probability

$$\epsilon = \begin{cases} 1 - \frac{1-0.15}{T_{explore}} \cdot (t - T_{warmup}) & \text{if } 1 - \frac{1-0.15}{T_{explore}} \cdot (t - T_{warmup}) > 0.15, \\ 0.15 & \text{otherwise} \end{cases}$$

where t is the current time-step, $T_{explore}$ is the number of time-steps to decay to a minimum exploration rate and T_{warmup} is the time-step at which the exploration probability starts to decrease. This ensures that the agent explores at the beginning of learning and increasingly becomes more greedy in its policy (i.e. use the Q-network to decide an action), with a minimum exploration probability of 0.15.

The Q-network The Q-network is defined as a deep neural network that gets as input s_t and aims to maximize future reward (see section 2.2.1). The Q-network used in this study mainly works as aforementioned, with some slight additions. That is, the Q-network will aim to minimize the Huber loss and will experience replay as described. Regarding the input s_t, the Q-network will receive $s_t = \{x_{t-3}, x_{t-2}, x_{t-1}, x_t\}$, in which each x_i is a to gray-scale and down-sampled to size (84, 96) from (224, 256) preprocessed image of the Super Mario Bros game screen. An example of an four input frames is given in figure 2. The main advantage of giving multiple frames as input, is that speed and acceleration can be inferred by comparing the frames. As such, in figure 2 can be seen that the enemy on the left walks away from Mario, and the two enemies on the right walk toward Mario.

Now, because previous states and rewards are stored in memory, the network could be able to maximize future reward more than only one time-step ahead. This means the new target becomes:

$$y_i = Q^*(s;a) = E_{s'}[r + \sum_{t'=t}^{T} \gamma^{t'-t+1} \cdot \max_a Q^*(s^{t'}, a^{t'}; \theta_{i-1})|s, a]$$

with T being the amount of steps the Q-network is set to maximize for in the future. The Q-network will use Adam with a learning rate of 0.001 to minimize its loss.

The deep convolutional network architecture consists of:

1. Input layer with $4 \times 84 \times 96$ pixels

2. Normalization layer with $4 \times 84 \times 96$ pixels
3. 1st convolutional layer, with 32 filters, kernel size of (8,8), strides of (4,4) and a ReLu activation function
4. 2nd convolutional layer, with 64 filters, kernel size of (4,4), strides of (2,2) and a ReLu activation function
5. 3rd convolutional layer, with 128 filters, kernel size of (3,3), strides of (1,1) and a ReLu activation function
6. Flatten layer that creates a 22528 node mapping
7. Fully connected linear layer consisting of 512 neurons that outputs Q-values for all 14 possible actions

The architecture of the Q-network is depicted in figure 3.

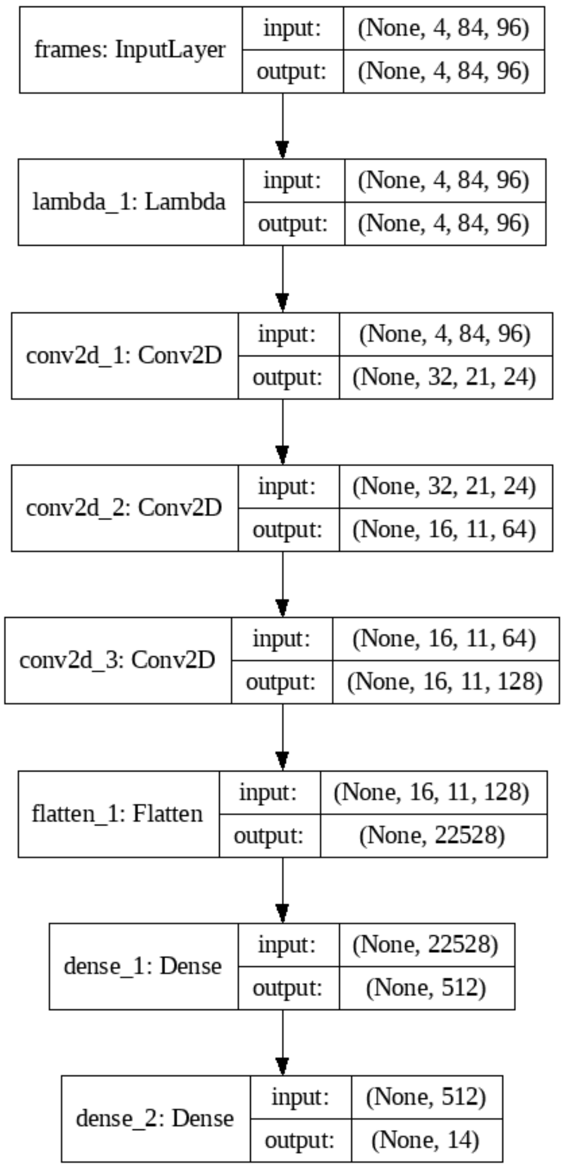
frames: InputLayer | input: (None, 4, 84, 96) | output: (None, 4, 84, 96)
lambda_1: Lambda | input: (None, 4, 84, 96) | output: (None, 4, 84, 96)
conv2d_1: Conv2D | input: (None, 4, 84, 96) | output: (None, 32, 21, 24)
conv2d_2: Conv2D | input: (None, 32, 21, 24) | output: (None, 16, 11, 64)
conv2d_3: Conv2D | input: (None, 16, 11, 64) | output: (None, 16, 11, 128)
flatten_1: Flatten | input: (None, 16, 11, 128) | output: (None, 22528)
dense_1: Dense | input: (None, 22528) | output: (None, 512)
dense_2: Dense | input: (None, 512) | output: (None, 14)

Figure 3: Representation of the Q-network.

Reward function The original Super Mario Bros environment as provided by the OpenAI Gym environment gives rewards that represent pixels moved to either the left, right or not moved. Score for killing enemies or gaining coins is thus not included. This means that rewards between -10 and 10 are typically observed and a sudden reward that can easily be $\ll -100$ when the agent dies. This high meaningless variance in rewards may cause the Q-values to diverge too much, thus a clipped reward function was created such that the new rewards become:

$$r_{new} = \begin{cases} +2 & \text{if finish level,} \\ +1 & \text{if } r \geqslant 8 \quad \text{(quickly walk right),} \\ +0.5 & \text{if } 0 < r < 8 \quad \text{(slowly walk right),} \\ -0.1 & \text{if } r = 0 \quad \text{(idle),} \\ -1 & \text{if } -8 < r < 0 \quad \text{(slowly walk left),} \\ -1.5 & \text{if } r \leqslant -8 \quad \text{(quickly walk left),} \\ -3 & \text{if } r \leqslant -8 \quad \text{(death),} \end{cases}$$

2.3 NeuroEvolution of Augmenting Topologies

2.3.1 Background

Next to reinforcement learning in the form of Deep Q-Learning, an evolutionary strategy has been implemented. Similarly to reinforcement learning, evolutionary algorithms are loosely inspired by natural processes [11]. Yet, unlike reinforcement learning in animals, which occurs on the time scale of hours or days within a single individual, evolutionary processes in nature operate on much larger time scales and across entire populations and species. This difference is mirrored in the training of evolutionary algorithms which aims to generate better adapted solutions over a sequence of generations through reproduction of *fit* individuals and random mutations.
In the context of Artificial Neural Networks a series of evolutionary algorithms have been proposed [12], commonly termed *Neuroevolution* (NE). All of these solutions have in common that they seek to generate increasingly better adapted neural network solutions to the task at hand, yet they differ in the neural network components that are optimized and the way these are evolved. NE algorithms are of particular interest to unsupervised or semi-supervised problem domains were no clear target values can be identified, since the objective is to maximize a user-defined fitness function which can be evaluated more sparsely than individual forward-passes through the neural network as in classical Deep Learning. Hence, specifically in the here investigated complex setting of controlling a Super Mario Agent, NE strategies offer a promising solution.
In this paper, the so-called NeuroEvolution of Augmenting Topologies (NEAT) [7] algorithm was implemented. Unlike many other NE algorithms, which pre-define a fixed network topology and evolve the networks connection weights, NEAT co-evolves both the network connectivity and its topology simultaneously. While it has been shown that neural networks with a single hidden layer and infinite width can theoretically approximate

any computable function [13], in practice network structure often considerably affects learning speed [14]. Therefore, NEAT offers a valuable solution for complex learning tasks.
NEAT makes use of a direct genetic encoding of network structure for evolution, meaning that the phenotype, i.e. the neural network, is explicitly represented in the genotype. For that, individual genes in a genome encode either a single network weight between two nodes, or an individual artificial neuron. Through random mutations, the network can be altered by introducing new nodes or connections or perturbing existing weights. Through reproduction or *crossover*, fit individuals/networks can pass on their genes to the next generation, seeking to increase the total fitness $\mathcal{F}$ of the population.
In that processes, NEAT deals with the so-called *Competing Conventions Problem* [15] by keeping track of a gene's "historical origin" [7]. Competing conventions, refers to the problem that structurally different neural networks, can represent exactly the same function, e.g. through permutation of hidden nodes. For that reason, crossover of network substructures can lead to an effective loss of representational capacity. By keeping track of a genes history, NEAT makes sure that only compatible structures of the same heritage are crossed.
Another important novelty of the NEAT algorithm is directly inspired by the process of *niching* in natural evolution. Since structural *innovations* that occur in the course of neuroevolution can cause an initial decrease in fitness, newly developed structures are protected through *speciation*. If an offspring network is structurally different enough it will form a new species that can no longer crossover with genomes from another species. Thereby, it is ensured that multiple solutions are evolved alongside and no single local optimum dominates the evolutionary process.

2.3.2 Implementation

The population pool The implementation of the NEAT algorithm in this work is based on the lua implementation of "MarI/O" by Seth Bling [6]. The main structure of the algorithm is the population *Pool*. The pool holds all so far evolved individuals and keeps track of the global *innovation* number. The purpose of the innovation number is two-fold. It marks a genes historical origin and thereby allows for crossover of network structures without facing the problem of competing conventions. Furthermore, the innovation number can be used to determine the structural similarity of two networks to allow for speciation. Each time the network topology is altered through a structural mutation, the new gene encoding either the new network node or weight is tagged with the current innovation number which is then incremented. If the number of disjoint genes, indicated by the number of genes with non-matching innovation numbers, and the absolute weight difference between the network and the networks of the already existing species surpasses a pre-defined threshold a new species is created within the pool. Future crossover between individuals is only possible within a species, ensuring that new structures have sufficient time to evolve in order to show their adaptiveness. A new species is created for network g when

$$\frac{\delta_D \cdot D_{gs}}{N} + \delta_W \frac{\sum_{i \in C} |W_{ig} - W_{is}|}{|C|} < \delta_t \forall s \in S$$

with δ_t being the speciation threshold, D being the total number of disjoint network nodes and the respective weight δ_D, s indicating a species-defining network of a species in the set of all species S, C the set of indices of coincident weights in both networks, δ_W the weighting of the weight difference factor, and N being the maximum number of network nodes in both networks. It is sufficient to calculate the difference to a single network within each network to determine the speciation [7].

The genome As previously mentioned, the genotype directly encodes the structure and weights of a neural network. In our implementation the *genome* holds a list of *genes* which encode weights and their connection strengths between two artificial neurons. Further, a *network* structure holds the network's *neurons*. Each gene in the genome is tagged by an innovation number. A genomes fitness is evaluated by employing the neural network it represents to navigate the mario agent in the simulation environment. More details on the evaluation is described below. When all genomes in all species have been evaluated, a new generation of genomes is formed by randomly mutating genes and crossover between fit individuals. Possible mutations are:

- **Point Mutation**: Randomly perturbs a connection weight or replaces it by a new random value
- **Link Mutation**: Randomly creates a new connection between two nodes
- **Node Mutation**: Randomly creates a new neuron between two existing nodes
- **Enable/Disable Mutation**: Randomly disables (or enables) a gene, temporarily removing the connection

For crossover, genes that match according to their innovation number are inherited randomly of either parent, while excess genes are taken over from the parent with higher fitness. The genomes for crossover are selected randomly. At each new generation the weakest individuals (according to their fitness) in each species are removed as well as any entire species that has not improved the maximal fitness of its genomes in a certain number of iterations. The entire population is replaced by the new genomes.

Evaluation In each generation, all individuals, i.e. genomes, in all species are evaluated in the game environment. To that end, the network represented by a genome is constructed and used to control the mario agent. As in the Q-learning implementation, the network input is a gray-scale image of the game screen. The network output is one of fourteen possible actions. The game is simulated until the agent has not advanced in the level for T time steps where the factor $\frac{\#\mathtt{Frames}}{4}$ is added to T, allowing networks that have already progressed far into the level to run for more time. The fitness of the genome is the total distance in pixels traveled to the right minus the current frame number divided by two. Subtracting the frame number divided by two encourages faster progression

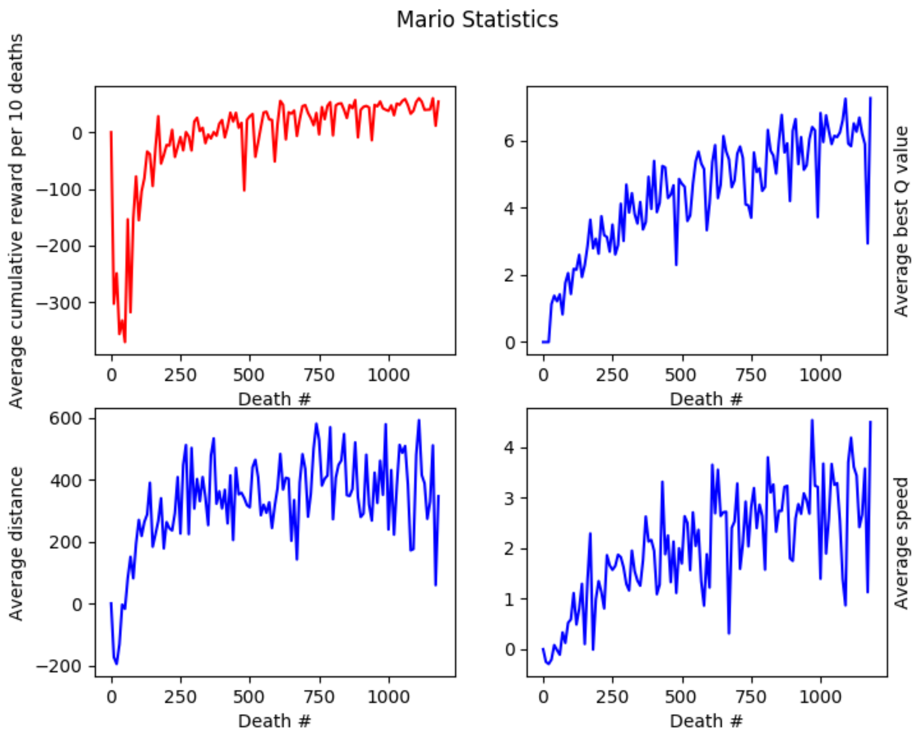

Figure 4: Statistics for the Q-learning algorithm.

through the level. The network is continued to be evaluated when the agent dies but if no checkpoint has been reached, it is usually difficult to improve the best position within the given time limit.

2.4 Analysis/comparison

For the analysis of the algorithm the trainings success of the individual approaches is evaluated. For that the progress over the course of training is visualized.
Additionally the two algorithms are directly compared to each other. To that goal, the best models after training are evaluated in the game environment for 10 runs. The models are scored by the cumulative reward obtained from the environment. To establish a baseline, additionally a random agent is evaluated that selects actions non-deterministically at the same frequency as the other agents. As a second metric, the average speed of the agents is reported. The speed is defined as the average distance traveled divided by the number of frames since the last dead. This measure provides additional information on how well the agent manages to progress through the level. Finally, the agents are evaluated qualitatively.

3 Results

3.1 Q-Learning

The Q-learning algorithm was run for 362403 frames, taking approximately 10 hours of computation time on a computer using a GTX 1080 Ti GPU. In this time the agent had 1172 deaths. Four statistics were measures for every death-run and averaged over 10

runs: 1. average cumulative reward, 2. average best Q-value, 3. Average distance, and 4. Average speed (calculated as distance divided by frames in that run).
The results are visualized in Figure 3. Regarding the cumulative reward (as r_{new}), it can be seen that the agent improves from many negative rewards to slightly dominated by positive rewards at later deaths. This indicates that the Q-agent improved performance in the game mainly at the beginning of learning. Secondly, looking at the average Q-value per death shows that this increases over time. This implies that the Q-network favors a subset of choices giving them higher an higher Q-values output. Next, the average distance — the real measure of interest — increases much over the first deaths and seems to converge at later runs. This confirms the relationship over deaths as the reward does, thus implying that the reward function has a good carry-over effect to game performance. Similarly, speed per run increases which implies the Q-agent finds a way to be increasingly faster at the Super Mario Bros game. It has to be noted that distance in this graph was calculated including the penalty at the time of death. This means that the actual maximum distances reached were higher than depicted in the figure. However, the figure still depicts the progression over time to the same extent. Figure 5 depicts the distance of the last 600 runs, categorized in bins. Equal to figure 3 was the distance used inclusive of the big death penalty, which ranges between -80 to -350, explaining the negative values in figure 5. Looking at the histogram, only 4% of the runs resulted in a high distance (>1000), while 21% resulted in an early death giving a negative distance.

Qualitative analysis The main strategy which seemed to be employed by the Q-agent was to run right very fast,gaining high rewards, with an occasional jump. The Q-agent seemed to have developed a level of skill in passing the pipes in the level, which were passed in all cases after training. Likewise was there some developed skill in dealing with enemies and small holes in the ground, although this was not consistent due to for instance wrongly timed jumps. The Q-agent did not seem to be able to deal with large holes well, mostly running or jumping straight down. Another reoccurring observation was that the Q-agent was particularly bad at avoiding enemies coming from the left, which might be a result of enemies mostly coming from the right during running.

3.2 NEAT

The NEAT algorithm was run for 320 generations, taking approximately 20 hours of computation time on a personal laptop computer. At each generation the maximum and average fitness across all species was evaluated. Figure 6 shows these quantities over the course of training. It can be observed that both curves remain relatively flat over generations indicating that no systematic increase in fitness occurred. Yet infrequent peaks in maximum fitness can be seen, corresponding to individual solutions that perform very well.
Another visualization can be seen in Figure 7. It depicts the cumulative fitness over all generations per species. Most species show relatively flat fitness values indicating in most generations the network did not successfully control the agent. Few species show a relatively steady increase in fitness indicating consistent success in the game, but these species mostly die out after some generations, presumably because the species failed

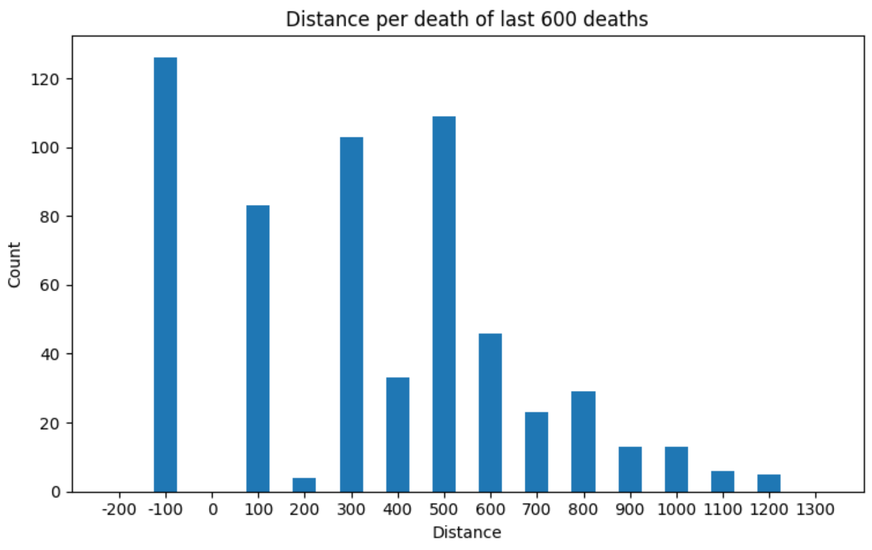

Figure 5: For the last 600 runs of the Q-learning algorithm, the distance of each run categorized in distance bins of 100.

to improve in fitness for too many successive generations. Some species consistently performed very poorly and still survived until the end of the training.
Furthermore, Figure 8 shows a histogram over the distances traveled by the agent in the game. It can be seen that most networks performed quite poorly and remained at the very beginning of the level or even went to the left of the starting position. Only a few networks managed to travel some distance along the required direction.

Qualitative analysis Similar to the Q-agent, the main strategy employed by the NEAT algorithm seemed to be to move to the right as fast as possible by jumping frequently. These strategy can prove very successful but is limited since it does not take into account much of the information provided in the input. Most agents evolved even late during training do not move or move to the left of the level right at the start. Since an evaluation is terminated after a number of consecutive trials without improvement they receive a very low fitness value.
Those agents that successfully progress in the level often fail at later structures in the game, like holes or new kinds of enemies. No agent during training has managed to finish the level.

3.3 Comparison

In this section the two learning algorithms are compared. As mentioned, a baseline is established by letting an agent choose actions randomly. Table 1 shows the mean resulting cumulative rewards and average speed with one standard deviation for the three algorithms after 10 evaluations. It can be seen that both solutions of the NEAT algorithm and the Q-learning agent perform comparably. NEAT on average proceeds further into the level than the reinforcement learning agent but also has a higher standard deviation, indicating that for some runs it performs more poorly. The Q-learning agent on the other hand consistently has a higher average speed in which it traverses the level. The baseline

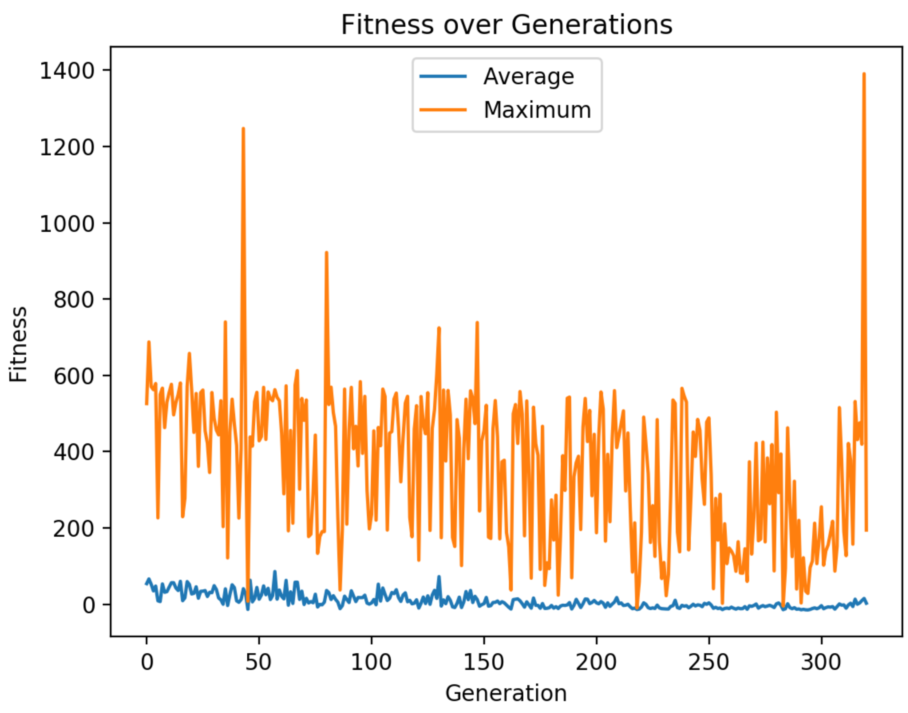

Figure 6: Maximum and average Fitness over generations.

random agent performs worse on both metrics. It moves very slowly through the level since it performs many actions that cancel each other. Furthermore, it does not get very far and the standard deviation is of the same magnitude as the average distance traveled indicating very inconsistent results.
In order to establish whether the observed differences in performance are significant, t-tests were performed between all pairs for both metrics respectively. Since the underlying variances are not assumed to be equal, Welch's t-test for unequal variances was used. For all tests an alpha level of 0.05 was set. For the difference between the Q-agent and the NEAT algorithm no significant difference was found for either Performance t(16) = 0.3627, $p > 0.7$ or Speed t(17) = 1.0072, $p > 0.3$. For the comparison between NEAT and the baseline, a significant difference for Performance t(15) = 2.4131, $p < 0.03$ and Speed t(12) = 5.3431, $p < 0.001$ was found. Similarly, significant results were obtained for the Performance of the Q-agent compared to baseline t(17) = 2.3993, $p < 0.03$, as well as for Speed t(13) = 7.7675, $p < 0.001$.

Table 1: Comparison between different Algorithms

Algorithm	Performance	Speed
Q-learning	627.8±343.5	6.29±1.9
NEAT	692.14±443.49	5.33±2.34
Random	293.86±275.17	1.05±0.97

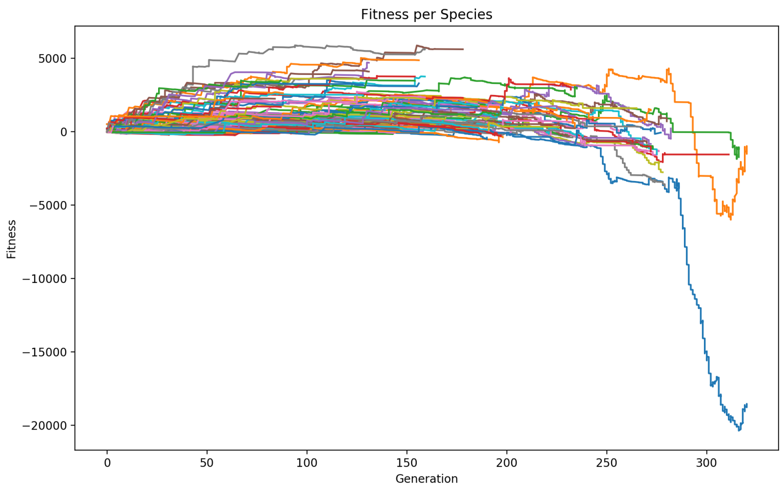

Figure 7: Fitness over generations per species.

4 Discussion

The goal of this project was to train an agent to complete levels of Super Mario Bros using two reinforcement learning strategies, and compare the results. We successfully implemented the Deep Q algorithm and NEAT algorithm and trained the agent on the first level of the game. There were some complications, for example the first level contains a pipe that allows Mario to enter an underground bonus level. This level has a different layout and color scheme than the original training level and the model can't deal with this well. Instead, the model has learned to ignore the pipe and jump over instead of entering it. Aside from the bonus level, the original level itself also has a challenging color scheme for the agent. The floor, the enemies and parts of Mario all have the same color brown. This makes it difficult for the networks to recognize the pixels as separate objects, and since the actions of the agent rely directly on the processing of the visual input this can hinder the performance.

We chose to train the agent on the first level because it introduces new gameplay elements as the level progresses. The obstacles encountered by the agent early in the level, like jumping over pipes and enemies, are encountered many times and the agent learns to overcome them. Later in the level there are other obstacles like gaps in the ground. Because these obstacles are seen much less often by the agent it doesn't get trained much on them and has a hard time overcoming them. For Deep Q-learning this can be seen from Figure 5, where only 14.8% of the runs made it further than a distance of 500, which is the location of the first gap in the ground.

For the Q-learning algorithm, the results show that the agent has learned the goal of the game (i.e. completing the level) and learned to avoid enemies and jump over obstacles. In Figure 4, the average distance plot shows that the average distance of the agent stops increasing near the end of the training, and in fact decreases a bit. However, the cumulative reward keeps growing. Since the reward depends both on the traveled distance and the average speed, this indicates that while the agent did not learn to overcome new obstacles at the end, it learned to get through the levels at a much faster pace than before.

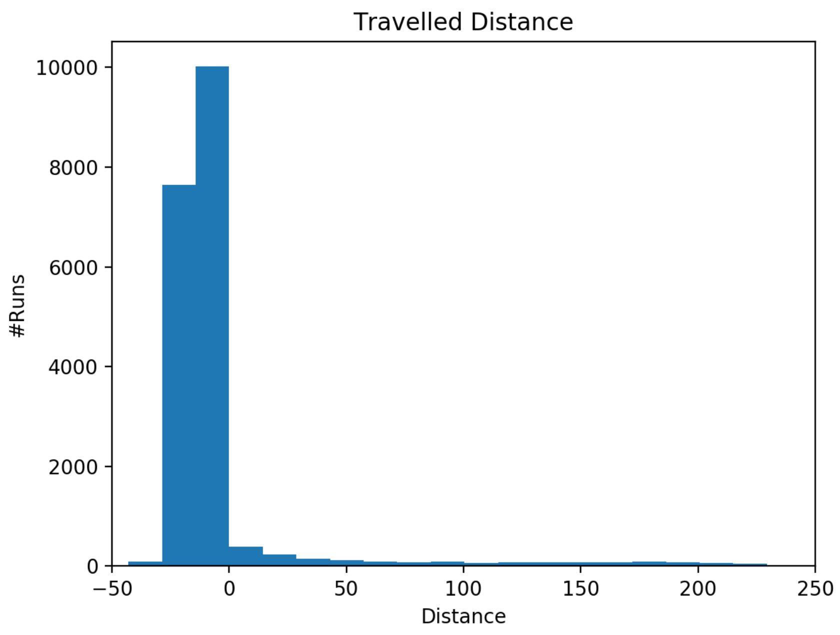

Figure 8: Histogram of distances traveled by all NEAT agents.

The evolutionary strategy generally seems to be mostly running to the right as fast as possible and jumping a lot. Sometimes this goes well but there is no consistency. Sometimes the agent seems to recognize and avoid enemies, but more often it runs straight into them. One explanation for this is that the agent simply requires more training. Training is computationally heavy with models that grow rapidly over the generations. A previous implementation of NEAT with Super Mario Bros [6] was done with 24 hours of training, but since no hardware was specified we could not compare our training time to theirs.
While the average performance did not improve significantly after training, there are some individual agents that perform very well. This is likely caused by speciation during evolution, the mechanism that allows for exploration of many different solutions in parallel. In hindsight, training could have been more efficient if the balance between exploration and exploitation was shifted towards exploitation. This would favor good solutions and give them a chance to further evolve. Evaluations are terminated if the agent does not move, and this has caused potentially good structures to stop evolving and be purged from the gene pool.

5 Conclusion

We fulfilled our goal to train an agent to play levels of Super Mario Bros using reinforcement learning. Both Deep Q-learning and NeuroEvolution of Augmenting Topologies are appropriate techniques and they consistently perform significantly better than the random agent.
To improve our model more levels could be added to make the agent more flexible with handling its input, and more time should be reserved for training. The models can be further optimized with parameter tweaking, rebalancing exploitation versus exploration, changing the neural network architecture or changing the reward function. Furthermore,

many similar techniques exist that might yield even better results. Reinforcement learning is a promising method for training autonomous agents and many variants are being developed. An example is the use of Spatial Transformer Networks [16], which can make CNNs spatially invariant to Mario's position. Other examples are Double Q-learning [17] and curiosity-driven learning with internal rewarding [18].

6 Project evaluation

6.1 Reflection

We started out with a different research question. The initial goal of the project was to train a generative adversarial network (GAN) to generate new levels of Super Mario Bros. Even though we knew that training GANs often poses considerable difficulties, especially with a limited time frame we decided that we would be able to manage within the scope of the project. In order to make the project as likely to succeed as possible we decided to stay as close as possible to the implementation of the original paper. Unfortunately even after consultation with the authors we were not able to reproduce any sensible results. After spending too much time on a desperate project, we decided to shift focus. The second part of our project proposal consisted of implementing an evolutionary agent to play the generated models. We decided to make this the main part of our final project. Since we thereby omitted a significant part of the initial workload, we decided to extent the research question. During the course we often pondered how the presented algorithm fare in comparison to more prominent machine learning algorithms. Therefore, a natural extension of our project was to use a contemporary reinforcement learning implementation to compare the evolutionary algorithm to. Reinforcement learning receives a lot of research attention in current times, yet similar to evolutionary strategies is inspired by a natural process. Hence, combining these approaches in our project both addressed our curiosity as well as it seemed to do the theme of the course justice. The advice to be careful when considering working with GANs will be kept in mind.

6.2 Task distribution

Tasks were uniformly distributed as:

$$F(k; a, b) = \frac{\lfloor k \rfloor - a + 1}{b - a + 1}$$

Two distinct subtasks were identified. On the one hand a reinforcement learning agent had to be implemented. Additionally, the NEAT algorithm was developed. To that goal, we split into teams of two. Steven and Tjalling worked on the deep Q-agent, while Tido and Kai implemented the evolutionary strategy. All group members contributed equally to the creation of the report. Furthermore, all members were involved in the initial attempt to create a GAN to generate super Mario levels.

References

[1] M. Carreras, J. Yuh, J. Batlle, and Pere Ridao. A behavior-based scheme using reinforcement learning for autonomous underwater vehicles. *IEEE Journal of Oceanic Engineering*, 30(2).

[2] Takayuki Osa, Naohiko Sugita, and Mamoru Mitsuishi. Online trajectory planning and force control for automation of surgical tasks. *IEEE Transactions on Automation Science and Engineering*, 15(2).

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[4] Peter Dayan, Yael Niva, Ben Seymour, and Nathaniel D. Daw. The misbehavior of value and the discipline of the will. *Neural Networks*, 19:1153–1160, 2006.

[5] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47(2).

[6] Seth Bling. MarI/O machine learning for video games, 2015. URL `https://www.youtube.com/watch?v=qv6UVOQ0F44`.

[7] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[8] Greg Brockman, Vicki Cheung andLudwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv:1606.01540*.

[9] Peter J Huber. Robust estimation of a location parameter. *The annals of mathematical statistics*, pages 73–101, 1964.

[10] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[11] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[12] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[13] Franco Scarselli and Ah Chung Tsoi. Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results. *Neural networks*, 11(1):15–37, 1998.

[14] Hrushikesh Mhaskar, Qianli Liao, and Tomaso Poggio. Learning functions: when is deep better than shallow. *arXiv preprint arXiv:1603.00988*, 2016.

[15] Nicholas J Radcliffe. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing & Applications*, 1(1):67–90, 1993.

[16] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. In *Advances in neural information processing systems*, pages 2017–2025, 2015.

[17] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv:1509.06461*.

[18] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *ICML 2017*.