Natural Computing Project
Neuroevolution with Snake

Valentin Kotov, s4629701 Nicole Walasek, s4629310
Luuk Scholten, s4126424

June 2016

Contents

1 Introduction

This report contains the results of our project of applying NeuroEvolution of Augmenting Topologies (NEAT) (Stanley & Miikkulainen, 2002) to convolutional neural networks (CNNs), thereby "evolving" an AI that learns to play Snake. Snake is an indeterministic 2D arcade game in which the player maneuvers a line of growing length with the line itself being one of the primary obstacles. The advantage of evolving neural networks is that in contrast to supervised learning algorithms, neuroevolution does not require correct input-output pairs during the training phase as it actually builds on reinforcement learning.

By combining genetic programming with neural networks, NEAT is able to optimize not only the weights of the neural network but also its topology using an entire population of networks and some task specific fitness measure. There are various variants of and abstractions from NEAT, and numerous implementations are publicly available[1]. NEAT is, for instance, being extensively used in Game AI design (Risi & Togelius, 2014), and caught our attention through an application to the Nintendo game Super Mario[2].

The NEAT algorithm is an important algorithm in the field of reinforcement learning. The field of reinforcement learning deals with agents taking actions in an environment to maximize some notion of reward. This is a very broad area with connections to fields as diverse as game theory, control theory, swarm intelligence, robotics, and others.

Learning an agent to play a game does not seem to be relevant to society at first, but the controlled environment and 'known' outcomes make it a perfect testbed for trying out and developing algorithms that may be useful in other, real-world scenarios. This has been acknowledged in the scientific world and many attempts have been made to "solve" difficult games. A concrete application of agents that act as general game players might be in the area of robotics. A robot navigating through unknown territory or in need of solving novel tasks might very well benefit form general game playing approaches. Many complex tasks can be broken down or "gamified" and thereby tackled by some form of general game player. Another application might be the use of these "intelligent" agents in simulations. When a new car is being developed and the engineers want to test new technical advances *in-silico*, it might be useful to do this by letting an "intelligent" agent drive the car in various scenarios.

Our main contribution to the already existing work on NEAT will be the incorporation of convolutions. We will discuss our ideas in more detail in the following parts of the report.

2 Neuroevolution of Augmenting Topologies

NEAT – Neuroevolution of Augmenting Topologies – is a genetic algorithm, developed by Stanley and Miikkulainen (2002), to evolve artificial neural networks. While neuroevolution in the classical sense is primarily concerned with optimizing the weights for a fixed topology, NEAT aims at optimizing both the weights and topology of the network at the same time, thereby balancing the fitness of evolved structures and their structural diversity. Stanley and Miikkulainen (2002) were able to outperform the best fixed-topology method (at that time) on a challenging benchmark reinforcement learning task. The authors employ three key techniques: (1) allowing crossover among differing topologies, (2) applying speciation to preserve innovations and (3) complexifying solutions, by incrementally developing topologies from existing structures.

[1]`http://eplex.cs.ucf.edu/neat_software/#NEAT`

[2]`https://www.youtube.com/watch?v=qv6UVOQ0F44&list=PL2E35EBCF273E6B8D&index=5&feature=iv&src_vid=S9Y_I9vY8Qw&annotation_id=annotation_4250647787`

2.1 Related Work

An excellent review of the use of NEAT and other methods for neuroevolution in games is given by (Risi & Togelius, 2014).

Within video games, NEAT has among other things been applied to driving a simulated car in The Open Car Racing Simulator (TORCS) (Cardamone, Loiacono, & Lanzi, 2009), playing Unreal Turnament (Schrum, Karpov, & Miikkulainen, 2011) and moving Mario through Super Mario World[2]. Several extensions of NEAT exist, for example SNAP-NEAT, which improves over NEAT's performance on so called highly fractured problems (problems in which the correct action varies discontinuously from state to state, (Kuipers, Ghosh, & Stone, 2009)) by combining two extensions that respectively allow for radial basis function nodes (RBF-NEAT) and introduce cascades (Cascade-NEAT) (Kohl & Miikkulainen, 2012).

Another extension of NEAT suggests that scaling to higher-dimensional representations such as using the raw game screen as input, can be realized by the use of hypercube-based encoding for evolving large-scale neural networks (HyperNEAT) (Stanley, D'Ambrosio, & Gauci, 2009).

Most importantly, our work is based on the idea not to train a neural network and convolutions in the classical manner (through supervised learning) but by learning the weights and topology of the network by means of genetic evolution and reinforcement. For our specific application of playing a 2-D game with neuroevolutionary algorithms (such as NEAT or HyperNEAT) we would regard Seth Bling's implementation[2] as an intuitive demonstration of the state of the art regarding NEAT. However, different realizations of this approach for different games (with different constraints) exist, e.g. a HyperNEAT-based Atari general game player (Hausknecht, Khandelwal, Miikkulainen, & Stone, 2012) or a real time application of neuroevolution which incorporates player feedback and thereby allows for an interactive learning process in the agents (Stanley, Bryant, Karpov, & Miikkulainen, 2006).

All previous work we are aware of have in common, however, that they extract the interesting features using direct access to the game emulator's memory. Taking high-dimensional data such as visual data as input remains an open challenge to date (cf. Risi and Togelius (2014) section VIII, subsection C).

A different approach is taken by Koutník, Schmidhuber, and Gomez (2014), who separately train the weights of a convolutional network ("feature extractor") in an unsupervised setting, and evolve a recurrent network ("controller"), whereby the ouput of the former is the input to the latter. The controller therefore capitalizes on the input reduction afforded by the feature extractor.

Of all previous approaches, ours is the most similar to this one, although in our approach, both the "feature extractor" and the "controller" evolve at the same time by means of neuroevolution.

3 Goal: Can we improve NEAT by adding Convolutions?

Our main idea for this work is to successfully integrate convolutions into the classical NEAT setup and to investigate whether this improves the original NEAT framework for 2D arcacde games.

From competitions like ImageNet we know that convolutional neural networks work very well for image classification problems. As our input to NEAT will be a 2D-game state for each move, we decided to view playing a game as a series of 2D images. Our hypothesis is that using convolutions will generate useful transformations of the input game state, now representing distinct features learned by the convolutions, which can be used as a new input to regular NEAT.

We will try to contribute to the field of neuroevolution by incorporating convolutional neural network techniques. In more technical terms we aim at implementing a flexible framework for adding convolutional layers to an already existing implementation of NEAT.

4 Experimental Setup

To reach the goals set out in this report, we have to make several design decisions. First of all, we have to define the classes of games to test and the programming languages and packages to use. From this, we can derive our implementation of ConvNEAT, and run our experiments.

4.1 Games

In the beginning of our work for this project we came to a point where we were wondering whether it was actually possible to develop useful features by convolutions based on an indeterministic, constantly changing input. We hypothesized that using convolutions might actually be advantageous for learning to play indeterministic games in contrast to regular NEAT. As we found this to be an interesting research question on its own we integrated it into our project by defining two classes of games on which experiments are performed:

Indeterministic games Indeterministic games are games where the output of the game can be different each time the player performs the same action. In these games there is some form of randomness which determines the outcome of the game.

One of the simplest indeterministic games is the famous 'Snake' game. In this game a player tries to get a higher score by moving a snake on a game board towards a piece of food. The player receives a point and the snake's length is increased by one when the player reaches this piece of food. The food is then randomly placed on a different position on the game board. The game ends when the player runs into a wall or when the player runs into itself.

Deterministic games Deterministic games are games where the output is always the same given an action of the player. One example of a deterministic game that was mentioned earlier on is Super Mario World. If a player walks through the level twice at the exact same timing, the enemies and power-ups will always spawn at the same points. We predict that deterministic games are easier to learn for genetic algorithms; whether it be NEAT or ConvNEAT. This is because the algorithm can just optimize the playing of the game by trial and error, without needing to generalize for any indeterminism.

We use two deterministic games in our experiments. The first one is a deterministic version of Snake, where the piece of foods spawns in the top left corner of the game board and the bottom right corner of the game board in an alternating matter. The player then only has to learn to walk to the top left corner, and the bottom right corner, without running into himself or the wall. The second deterministic game is an adaptation of the Snake game, where no food is involved. Instead, the snake starts from a certain position and grows with one at each step, leaving the tail of the body on the original starting position. The goal of this game is for the player to fill the game board without running into walls. In this 'growing Snake' game, walls are also placed on the game board.

We have implemented these three games using the python language. Our implementation is heavily built on the simple Snake implementation written by Lis (2013). Our version is built to run without a human player, and without any GUI at all. Whenever the game needs a decision from the player, it propagates the input through the neural network currently being tested. One noteworthy adaptation is the mapping of the game state to the input of the network. Instead of feeding the network the direct game state (the pixel values of the game), the input is transformed so that the head of the snake is always in the center of the game state. In this way the algorithm doesn't have to optimize for the actual position of the snake, but can make a decision based on the surroundings of the snake. The major disadvantage of this approach is the fact that the input size is significantly enlarged. For a game board of $N \times N$, the input to the network will transform to an input with size $2N - 1 \times 2N - 1$.

4.2 Available Packages and Implementations

Although NEAT implementations can be found on various platforms we chose Python as all of us feel most comfortable with this language. Choosing a familiar language also allows us more easily to make adaptations to already existing NEAT implementations. In the very beginning of our work we were also considering Super Mario World as a deterministic game. However, we would have been forced to use Lua and moreover to implement regular NEAT ourselves because of existing conflicts between platforms for using a game simulator and for using packages in Lua. Resolving these issues was out of the scope of this project. Consequently, these problems triggered our decision to focus on a game that could be easily implemented in our language of choice, we found one realization of such a game in Snake.

We were able to find an already existing NEAT implementation, called NEAT-python (McIntyre, 2016). The NEAT-python package implements NEAT based on (Stanley & Miikkulainen, 2002) and is still under constant development. The current version implements feedforward and recurrent networks, as well as spiking neurons and more variants of these.

To evolve a solution to a problem, the user must provide a fitness function which computes a single real number indicating the quality of an individual genome: better ability to solve the problem means a higher score. The algorithm progresses through a user-specified number of generations, with each generation being produced by reproduction (either sexual or asexual) and mutation of the most fit individuals of the previous generation. The reproduction and mutation operations may add nodes and/or connections to genomes, so as the algorithm proceeds, genomes (and the neural networks they produce) may become more and more complex. The algorithm terminates when the preset number of generations is reached, or when at least one individual exceeds the user-specified fitness threshold.

5 ConvNEAT

The most important components of NEAT consist of genomes, genes, speciation and heritage tracking through historical markings. Any extension to NEAT must therefore certainly consider these important building blocks. Our extension adds a new concept to the actual network structure and we must therefore define a new genotype.

5.1 Genotype

Our newly introduced genotype is called the ConvGenome, which is an extension of the regular feedforward genotype. The architecture of the ConvGenome and the corresponding network is roughly depicted in figure 1. A ConvGenome, like a feedforward genome, consists of a set of node genes, which represent hidden nodes, input nodes and ouput nodes, and a set of connection nodes, which represent a connection between two nodes. The ConvGenome additionally consists of a set of ConvGenes, which represent convolutional layers.

We assume that the convolutions will always use 'valid' padding, and the output of one convolutional layer will therefore be of smaller size than the input. Because of the nature of the ConvGenome, we cannot know beforehand how many convolutional layers there will be, which makes the output size of the convolutional section uncertain. NEAT, however, needs a set number of input nodes for performing structural evolution. Our solution for this problem is adding one layer with N hidden nodes after the last convolutional layer, which always reduces the output of the convolutional section to a fixed size. The ConvGenome therefore also contains a set of weights, that maps the output of the last convolutional layer to the N hidden nodes that were introduced. Admittedly, this does introduce many new parameters to optimize.

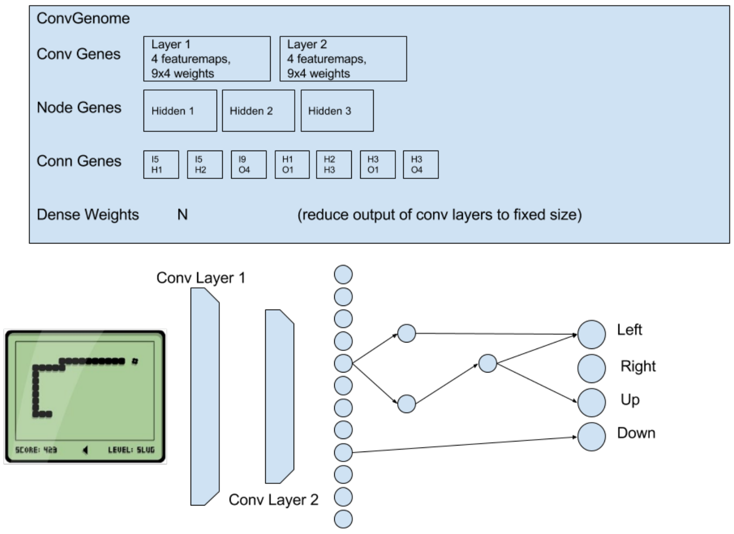


Figure 1: A sketch of the ConvGenome, with it's corresponding convolutional neural network

5.2 Phenotype

Each convolutional genome represents a specific network configuration, which is represented by the convolutional phenotype. The convolutional phenotype expands upon the normal feedforward phenotype of NEAT. When an instance of this phenotype is created from a genome, the weights for the convolutional layers and the set of weights for the layer of hidden nodes are passed to the network, along with the nodes and connections of the classical NEAT section of the genome. With all these weights, a convolutional neural network is created and input can be fed through the network to get the new direction of the snake. The input must be a 2D game state, which then is passed through the network. Each convolutional layer calculates the activations with the weights of the genome. After each convolution, the activations get passed through a leaky rectified linear unit (leaky ReLU), keeping all positive activations and multiplying all negative activations with 0.01. After the convolutions, the input is flattened to a one-dimensional vector and are activated with the weights of the fully connected layer. Once again, the leaky ReLU function is applied and finally the outputs of this is passed through the conventional NEAT network configuration.

5.3 Mutation and Mating

For our convolutional genotype we had to make a few design choices with respect to what structures and parameters we want to mutate and which we want to keep fixed.

Fixed (Hyper)Parameters We define a set kernel size for all layers, for the majority of the experiments we used a 3×3 kernel. Similarly, the number of feature maps corresponds to a predefined value and is the same for all layers. Throughout the convolutional layers a stride of 1 and valid padding is applied. Moreover, we set the number of hidden units for the dense layer to a fixed value. Lastly, we also need to specify a mutation chance for both adding a convolutional layer, and the adaptation of the kernel weights.

Mutation of Convolutional Layers We decided to mutate the number of convolutional layers with a probability that is predefined. In this case we add a randomly initialized convolutional layer with the predefined number of feature maps to the current network right before the dense layer.

Mutation of Kernel Weights Furthermore, we have a predefined mutation probability for the mutation of the kernel weights. For each layer we go through each feature map and mutate each individual weight based on this predefined probability. We add a small value drawn from a standard normal distribution multiplied by a predefined learning rate to the current weight. In analogy to the mutation of the connection weights in the regular NEAT implementation we also experimented with replacing the current kernel weight by a value randomly drawn form a standard normal distribution in case we exceed a certain replacement probability. However, our experiments suggested that even if the probability for replacement is chosen to be very low it harms our performance to the extend that the network does not seem to learn any more. We believe that this demonstrates the structural differences between a feedforward network and a network also containing convolutions. It is observable that the convolutional part of our network is much more sensitive to drastic changes than the regular NEAT part.

Speciation In order to define different species for preserving innovation we were required to implement the distance between ConvGenomes. Finding an appropriate distance function was crucial for our approach to work. Adding a randomly initialized convolutional layer constitutes a rather drastic change to the current network structure. In terms of each individual's fitness it will result in a decay in most of the cases as the weights of the kernels first need to be optimized. This issue can be resolved to some extent by exploiting speciation and thereby preserving the newly added layer. The implementation of an adequate distance function was very challenging as we were not able to find guidelines for the distance between convolutional layers in the literature and it was hard to evaluate the effect of different distance functions. In the end we settled for the normalized, absolute difference between kernel weights of the minimum number of feature maps in both parents. Using the terminology of the original NEAT paper these genes, i.e. layers can be called *matching*. Those layers that do not line up may be either *disjoint* or *excess* depending on whether they occur within or outside of the other parent's innovation number. Formula 1 shows the calculation of the distance, where c_0, c_1 and c_2 correspond to predefined scaling parameters while E and D stand for *excess* and *disjoint*, respectively, $\widetilde{K}$ refers to the absolute normalized kernel difference and N corresponds to the number of genes in the larger ConvGenome.

$$\delta = \frac{c_0 * E}{N} + \frac{c_1 * D}{N} + c_2 * \widetilde{K} \tag{1}$$

Mating For each matching layer (matching in terms of the number of layers), i.e. for each homologous gene, the ConvGene of the child will randomly inherit feature maps from both parents. Given that the fitter parent features more layers, the child will copy the additional layers from that parent. As this mating scheme enforces that the child and the fitter parent have the same number of feature maps the dense layer weights from the fitter parent are adopted completely to ensure a certain harmony when merging the structures.

6 Experiments and Results

In this section, we will report on a couple of experiments that aim to verify, following our initial research interest, whether our expansion to NEAT, ConvNEAT, can indeed be regarded as an improvement over classical NEAT for the task of game playing on certain levels. We have run three experiments, two of which compare how ConvNEAT performs compared to NEAT on an indeterministic and on a deterministic game, and one which examines whether ConvNEAT works as a standalone method without making use of the structural evolution of classical NEAT added on top of the ConvGenes.

6.1 ConvNEAT versus NEAT playing an indeterministic game

For our first experiment we will be comparing classical NEAT to our ConvNEAT implementation when we teach them to play the regular indeterministic Snake game. Based on our assumptions and intuition we've come up with two hypotheses:

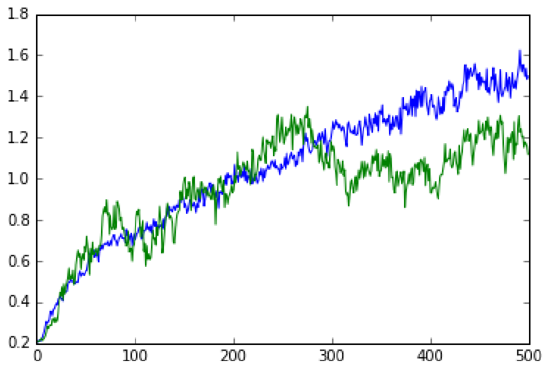
Hypothesis 6.1 *ConvNEAT will develop a different strategy than regular NEAT for playing an indeterministic game*

We think that ConvNEAT will learn a different strategy than regular NEAT for playing the game. The reasoning is that the convolutions transform the input space to a smaller input space for the NEAT algorithm. It will therefore not learn to make decisions based on the pixel values, but learn to make decisions based on the abstractions of the convolutional layers. This hypothesis will be tested by just looking at the results, as an empirical test for 'different strategy' is hard to engineer.

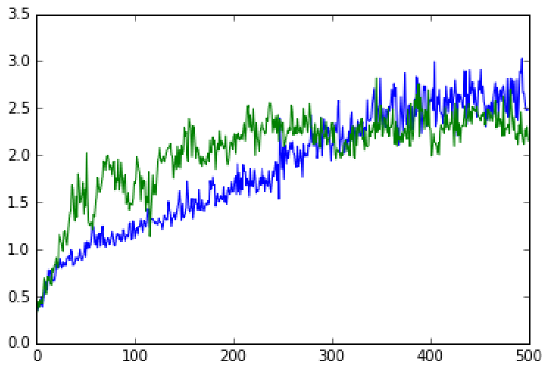
Hypothesis 6.2 *ConvNEAT will outperform regular NEAT in a reasonably sized indeterministic game*

Before testing this hypothesis, we have to carefully define two parts of the hypothesis. The first part is 'reasonably sized'; we expect the advantages of convolutions to not appear unless the game size is large enough. Classical NEAT could easily learn to 'solve' a small snake game by just trying out every possible game state. Because it is no longer possible when the game is large enough, we expect the power of convolutions to kick in at a larger input space. We will perform this experiment on a 11×11 board, which admittedly still only gives a game size of 21×21 pixels. The second and most important definition is the definition of 'outperform'. The experiments are computationally very expensive to run, so a full statistical analysis is unfortunately impossible. For this experiment, our method for checking whether ConvNEAT outperforms regular NEAT consists of comparing the fitness evolution graphs of two full runs. We will compare the average fitness and the best fitness in each generation, and we will derive our conclusions from that.

The experiments were performed with a 11×11 game board. Both classical NEAT and ConvNEAT were run for 500 generations using a population size of 100 individuals. The hyperparameter setting of NEAT and ConvNEAT were exactly the same, apart from the hyperparameters that concern only ConvNEAT. We've used 3×3 convolutions, 4 feature maps and started with 2 convolutional layers for our convolutional experiment. Both kernel weights and dense layer weight mutation have a mutation rate of 0.1. 40 hidden units were used in the fully connected layer of the ConvGenome. The fitness of an individual in the population is calculated by performing 50 runs of the game with the given genome.



(a) Average fitness of the whole population



(b) Fitness of the best genome in the population

Figure 2: Fitness over all 500 generations, blue line is regular NEAT, green line is ConvNEAT

Figure 2 summarizes the results of our experiment. We can see that regular NEAT is gradually performing better, and after 500 generations has no indication that the fitness will stop increasing. ConvNEAT on the other hand, seems more unstable. Around generation 250, the average fitness seems to be decreasing for ConvNEAT, which is obviously not what we want. Based on the graphs, we state that hypothesis 6.2 has been disproven; normal NEAT performs better than ConvNEAT for learning the indeterministic 11×11 Snake. We can see that ConvNEAT does indeed get a higher fitness faster than ConvNEAT, but it stops improving relatively soon while regular NEAT is still increasing in fitness.

What is more interesting however, is the analysis of hypothesis 6.1. For both these experimental runs, we have saved the genome that achieved the highest average fitness over 50 runs. When watching the classical NEAT network play the game[3], we can see that it has learned a very specific pattern. Most of the times, it seems to be going straight to the right wall, and starts to go down. From this point, it goes to the left whenever there is food on that row. It fails most of the times when a piece of food spawns above the snake. The convolutional network, however, almost always gets the first piece of food, and generally shows more seeking behaviour. We believe this behaviour is due to the abstractions created in the convolutional layers. Our conclusion is that hypothesis 6.1 is indeed confirmed, as ConvNEAT shows more 'natural' behaviour than regular NEAT.

6.2 ConvNEAT versus NEAT playing a deterministic game

Finding a solution to regular Snake is a statistical problem, as Snake is a game that depends on a random placement of food. The solution, i.e. the solving strategy/player encoded by the

[3]Please read the 'README.txt' file of the code to find out how to view the genomes play the game.

genome, which is meant to be found by the genetic algorithm has to generalize to a good portion of the possible food spawning sequences the player could encounter, which are n^l (with n the size of the board and l the length of sequences that is considered). This makes evaluating the fitness of any single mutation/genome a lot more difficult than it would be in other games, which indicates that Snake is not well suited for a simple comparison of the two methods.

For this reason, we implemented another game that, while still a version of Snake, does not require an aggregate fitness measure. In “Foodless Snake”, the fitness is not determined by the amount of food eaten, but by the length of the path traveled (i.e. the time survived). Food, and thus its random location, is entirely taken out of the equation.

Hypothesis 6.3 *ConvNEAT will develop a better strategy than regular NEAT for playing “Foodless Snake”.*

Ideally, this would be easily observable by looking at and interpreting the key statistics of a few solutions found during multiple runs of each algorithm. We defined an acceptable solution that can be compared in this way as the first genome that either exceeds a fitness level $fl = 100$ in any respective run, or that is the best genome after 200 generations.

To evaluate hypothesis 6.3, we have run both NEAT and ConvNEAT on a 11×11 board of Foodless Snake on which three obstacles have been placed in a similar fashion as in the experiment reported in section 6.3. Particularities of this game include the fact that decisions very early on constrain movements very late into the game.

Due to time constraints, we only ran each algorithm thrice, which already took several hours overall. While self-evidently, this does only give a small picture of the expected performance of either algorithm over many runs, it does allow a glimpse.

6.2.1 Regular NEAT

A population of 500 genomes was evolved until either the best genome exceeds a fitness of 100 (number of visited squares/length of path travelled), or until generation 200 is reached.

All populations had evolved for 200 generations before any genome exceeded a fitness of 100.

- run 1: best fitness: 66. Population average in 200th generation: 29.4 with $std = 16.46$. Throughout all generations, 8 species have existed, all of which survived until the end.
- run 2: best fitness: 58. Population average in 200th generation: 23.29 with $std = 13.21$. Throughout all generations, 13 species have existed, 12 of which survived until the end.
- run 3: best fitness: 77. Population average in 200th generation: 32.11 with $std = 21$. Throughout all generations, 24 species have existed, 22 of which survived until the end.

6.2.2 ConvNEAT

A population of 500 genomes was evolved in the same way as in the setup for pure NEAT.

Again, all runs reached generation 200 before reaching the fitness threshold.

- run 1: best fitness: 41. Population average in 200th generation: 13.38 with $std = 8.27$. Throughout all generations, 18 species have existed, of which 8 survived until the end.
- run 2: best fitness: 45. Population average in 200th generation: 13.97 with $std = 8.18$. Throughout all generations, 11 species have existed, of which 9 survived until the end.

- run 3: best fitness: 37. Population average in 200th generation: 12.99 with $std = 6.69$. Throughout all generations, 362 species have existed, of which 52 survived until the end.[4]

As can be seen, NEAT reaches the same performance in fewer generations, while simultaneously burning through fewer species, i.e. discovering and subsequently discarding fewer partial solutions. One might say that in our experiment, ConvNEAT lacked focus, being led astray onto too may false trails.

This gap in performance, we feel, does however not allow an outright disapproval of ConvNEAT, as the higher turnover of species implies that too many 'bad' evolutionary branches are explored, which impairs exploitation of the few that were good. This makes it much more likely that we are dealing with a parameter optimization problem. Indeed, finding the right speciation threshold in our current implementation of ConvNEAT is difficult, because the distance between kernels is not optimally normalized. As speciation plays a crucial role in the algorithm, this particular experiment seems to reveal this as a good candidate for future improvement.

Hypothetically, one would expect ConvNEAT to be able to find better solutions, because it can mutate on more "levels", which gives some of its mutations greater significance with regard to expected behavioral changes than single hidden neuron or connection mutations in regular NEAT would have. We currently have no way of quantifying such qualitative differences however, as we will discuss in section 7.

An unrelated, but potentially important observation made during this experiment concerned the fitness of the best individual: It did not monotonically increase over the generations, unlike what one would expect in a fully-deterministic game. This might, perhaps, be due to an error in our (or neat-python's) code, as the elitism used throughout all experiments (set, in this experiment, to carry on the best 5 individuals from any generation to the next) should actually have precisely this effect.

6.3 Fully Convolutional NEAT

In this subsection we will present the results of experiments investigating (1) whether it was possible to play a game solely based on input transformation via convolutions followed by a final dense layer activated by SoftMax, i.e. without any usage of original NEAT and (2) whether our implementation of the ConvGenome would allow us to train and develop a "classical" convolutional neural network, i.e. a network first consisting of a few convolutional layers, followed by a dense layer and a fully connected network on top of that. The fully connected network is evolved by regular NEAT where we set the probabilities of deleting and adding nodes to zero.

The following experiments were all conducted using 5×5 convolutions, 4 feature maps, 2 fixed convolutional layers to begin with and a game board of size 7×7. For both kernel weight and dense layer update we used a learning rate of 0.3. In the convolutional neural network 15 units were used in the dense layer. Population size and number of generations are subject to variation depending on the exact experimental setup and will be illustrated for each scenario individually.

In none of the following scenarios the best individual added another convolutional layer on top of the already existing ones. However, this does not deem the convolutional layers to be unnecessary. In contrary, we believe that they are essential for reaching a good performance. As it takes a few generations to train the weights to a point where they proof useful in terms of the fitness the evolutionary pressure for not adding layers is high. It might be that a more sophisticated distance function could solve this problem.

[4]This elevated number of species seems strange and one would expect there to be a misconfiguration of parameters, but in fact the settings in all runs were identical. We have no explanation for this high contrast in behavior.

Hypothesis 6.4 *ConvNEAT can still learn to play a(n) (in)deterministic game if there is no mutation after the fully connected layer.*

Hypothesis 6.5 *It is possible to train the weights of a convolutional neural network for playing a(n) (in)deterministic game by neuroevolution.*

Deterministic Game This section presents the results of both training only convolutions and training a convolutional network on a deterministic game. For both experiments we used 500 generations with a population size of 500.

Classical Convolutional Network After 500 generations with a population size of 250 the best individual was able to achieve a score of 29 in "growing snake" with a strategy that can be seen in figure 3a.

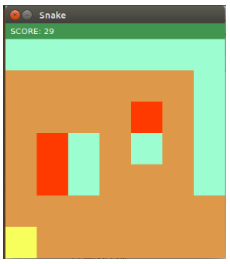
With only 500 generations this is already a decent result. Moreover, it confirms hypothesis 6.5 in that it shows the possibility of training a convolutional network by neuroevolution to learn to play a deterministic game. The result also shows that it would not be possible to solve the entire board if following generations would copy the initial strategy of this particular individual. However, we are positive that given more individuals and especially more generations it would be possible to win the game. One also has to acknowledge that the current board game is quite difficult for our evolutionary algorithm as failure to win the game due to the red walls raised within the board might only occur at the very end which makes it hard for the individuals to adapt their strategy in early generations.

Only Convolutions In analogy to the previous experiment we present the results of only training convolutional layers followed by a final dense layer with a SoftMax activation in figure 3b. The score of this fittest individual was 21. In comparison to the classical convolutional network the increase in the average population fitness is less steady and subject to larger variations. Unfortunately, we did not have enough time to experiment more with only training the convolutional layers to be able to draw more general conclusions. Nevertheless, we would argue that these results can be regarded as competitive with respect to the previous setup.

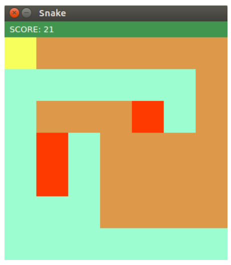
Furthermore, we are of the opinion that this result still confirms hypothesis 6.4, as it is possible to observe that the fittest individual not only managed to fill a portion of the board without hitting the walls but also developed a strategy that filled the lower right quadrant in an intelligent manner. However, it can be also seen that continuing from here on it would not be possible to fill the entire board.

Indeterministic Game For the indeterministic setup we chose the median over 70 runs per individual as fitness function.

Classical Convolutional Network We ran the algorithm with a population size of 250 individuals for 800 generations. During training we were able to observe a steady increase in the average population fitness. Figure 4a shows the scores, i.e. the length of the snake, achieved with the fittest individual that emerged from that run. These results confirm hypothesis 6.5. We therefore succeeded in showing that it was possible to train a convolutional neural network by neuroevolution and reinforcement learning to play an indeterminstic game without any use of backpropgation and supervised learning. Although the performance seems not to be perfect yet, one has to consider that the network has to optimize all weights of the convolutional layers, as well as all



(a) Result of best convolutional network.



(b) Result of best individual using only convolutions.

Figure 3: Achieved scores on "growing snake" after 500 generations with a population size of 250.

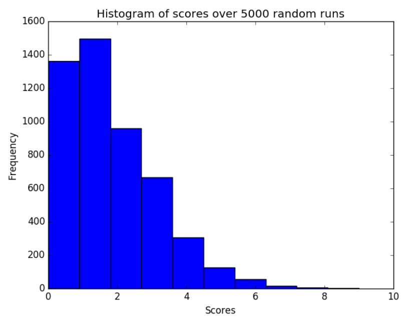
weights from the last dense layer and the weights of the fully connected network on top of that. Therefore, one can assume that given a larger number of iterations and a larger population size the average performance will increase.

Only Convolutions Due to time and computational constraints we were only able to run the algorithm for 100 generations with a population size of 250 individuals. During training a slow increase of the average population fitness was observable. It might be the case that more experiments with different learning rates would already improve the results. Figure 4b illustrates the scores, i.e. the length of the snake, achieved with the fittest individual that emerged from that run. In comparison to the results achieved with the classical convolutional network we would argue that the results after only 100 generations look promising. Most importantly, they also confirm hypothesis 6.4 and demonstrate that it was possible to train the weights of consecutive convolutional layers, followed by a final dense layer with a softmax activation by neuroevolution to play an indeterministic game. Of course, it would have been nice to contrast this setup in a fairer comparison, i.e. same number of generations with the previous results. An interesting feature of this setup is that it shows a lower standard deviation within random runs. However, this might be an artifact of only training for 100 generations. In the end, we believe that these results can be seen as a "proof of concept" of our successful implementation of the ConvGenome.

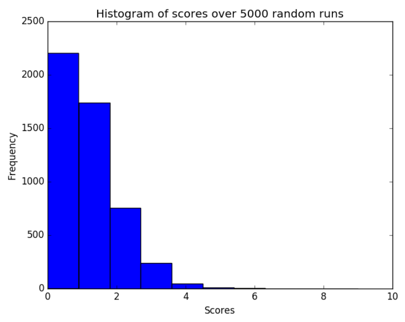
7 Discussion

This project addressed two primary objectives: (1) adding convolutions to NEAT, and (2) pointing to scenarios in which ConvNEAT has advantages over classical NEAT.

Concerning objective 1, we have developed ConvNEAT, which is a basic, yet general, flexible and customizable framework. It is an extension of NEAT-python (McIntyre, 2016). Our contribution lies in giving ConvNEAT the ability to evolve a form of convolutional net within a number of constraints that have been described in detail. The code base is currently not very well-documented, but we might still decide to take this additional step and then make it available online. Achieving objective 2, in turn, was much more difficult than we expected. The critical issue on which approaching this objective lingers is: What makes for a fair comparison? Although we have come up with multiple ideas, we haven't had the time to fully incorporate them given the limited scope of this project.



(a) Scores of best convolutional network trained for 800 generations. $M = 1.5622, MD = 1.0, SD = 1.47$.



(b) Scores of best convolutional output trained for 100 generations. $M = 0.85, MD = 1.0, SD = 0.96$.

Figure 4: Achieved scores on snake within 5000 runs, with a population size of 250.

7.1 Problems and Difficulties

We had to deal with several unexpected difficulties during the execution of this project.

We actually started out with the goal to tackle convolutions using Super Mario, but this turned out to be infeasible due to a complex and daunting web of technical constraints. After reconsidering this choice and recognizing Python as the best tool, we still had to deal with the fact that game emulation in Python is almost nonexistent in the way that originally made Mario attractive, and we settled for implementing the vintage game Snake for flexibility and adaptability reasons.

Another issue is the evaluation of fitness during evolution. Games that have a strong non-deterministic force such as Snake require a statistical approach to fitness, anything else will fail to capture the utility of a specific combination of genes in an unpredictable environment. The approach we chose is only very simple (an average of the performance over a number of runs per individual), but this could be researched much more extensively.

In particular, it was non-trivial to find a good yard stick that takes into account both the differences and the similarities between NEAT and ConvNEAT, and that would allow one to compare both methods, either in the Snake / Foodless Snake context, or elsewhere.

7.2 Ideas for Improvement

While it has to be pointed out that we were only able to really think about questions such as these at the very end of the available time (once we had enough functioning code to work with), we had a number of ideas about how to address especially the last point. What is for sure is that both ConvNEAT and regular NEAT can create networks that represent strategies to solve the given game. And in the indeterministic use case, strategies are not so easily compared: what exactly is it that we want to measure and compare performance in? A generalizable winning strategy should do well across as many runs as possible, but we might conceive of this in many different ways - because doing well "overall" is a very multifaceted notion.

Some of the points we think one has to consider when designing a criterion to compare regular NEAT to ConvNEAT are the following:

- The most fundamental theoretical problem has to do with the hyperparameters: We have quite some new parameters besides those already present in NEAT-python, and it is less than clear which combination is optimal for a given game. This is a big caveat for any preliminary comparison between the two, as we have no way of knowing how representative any chosen hyperparameter settings are. Insights into this could theoretically be gained through random search through the parameter space and subsequent functional Analysis of Variance (fANOVA) (Hutter, CA, & Leyton-Brown, 2014), although this would be, of course, a brute force approach.

- This aside, one could contrast a pair of NEAT and ConvNEAT setups by comparing the first solutions found by either of the two that achieve a fitness of level l by the histogram of their performance across n random runs. This disregards the number of generations needed to reach a certain level, which unsurprisingly is higher for ConvNEAT given its much larger parameter search space. Insights to be gained from interpreting the shape of the distribution include notions of generalization of the solutions to the respective problem domain, which in turn can serve as the basis for any sort of recommendations regarding the design and application of (both NEAT and) ConvNEAT.
- Following this line of thought, it might be very useful to constrain the source of randomness producing the game environments during evolution (e.g. the food spawning locations in Snake) or at least for comparison runs (e.g. use the same sequence of locations). This way, evolutional trajectories would still be open to a certain variation (as they have their own sources of randomness), but at least the comparison of the best solutions can be made more insightful e.g. through pairwise comparisons over all runs and statistical analysis.
- To further corroborate any conclusions reached on the basis of fixed-seeded evolution, it is conceivable to quantify to what degree the difference in performance between ConvNEAT and NEAT solutions can be attributed to variance in evolutionary trajectories. This could be done by evolving multiple populations of both kinds from the same initial conditions (mutation rates, speciation criteria etc).
- Independent from all of the above points, as has been hinted at in the previous subsection, the fitness measure by which mutations are rated during evolution could be exchanged for another distribution statistic, depending on the kind of solution one is interested in.

Independently of the search for a comparison criterion that generalizes well to both methods, it might be useful to change the representation of the board which is given as input to NEAT/ConvNEAT: While it is “snake-centric”, such that the environment moves around the snake and not the other way around, it might be helpful for any algorithm if it also shared the perspective of the snake, i.e. flipped 90° left or right whenever the snake makes either move. Like this, other activation patterns become useful, and in ways that might make it more easy to generalize a partial solution from one part of the snake-centric snake-oriented input to another part of it.

Furthermore, given more computational resources, it would become feasible to run both algorithms with a higher count of iterations per individual genome to calculate the fitness. As can be seen in the graphs in experiment 1, the fitness of the best individuals, even though they are carried from one generation to the next (save for errors in the code) shows a high degree of noise, especially in later generations. This might be important to consider, as the selection mechanism becomes a lot more stochastic than one may expect.

What is more, in the deterministic Snake experiment, we have identified the genome’s currently used distance function, and its kernel distance coefficient in particular, as a possible reason for concern.

Lastly, a game like Snake seems to be a good example for the "fractured problems" dealt with by SNAP-NEAT, which was mentioned earlier. Looking at Snake in particular, but also at the general case, a future update of ConvNEAT informed by the insights from SNAP-NEAT would be better suited to deal with such fractured problem domains.

7.3 Conclusion

This report shows that we have indeed created a flexible and extensible framework for the incorporation of convolutions in Neuroevolution of Augmented Topologies. We are able to evolve convolutional neural networks to play both deterministic and indeterministic games, from the raw pixel values of the game. Our experiments have indicated that ConvNEAT does not always come to a better network for playing the games, however they have demonstrated that ConvNEAT does find solutions that qualitatively differ (strategies look very different to a human observer) from those obtained with NEAT. Furthermore, ConvNEAT still works when the NEAT approaches for evolving the hidden units and connections is taken out of the equation, proving the robustness of our method.

Another interesting dimension of our approach is the fact the our method is able to train a convolutional neural network by reinforcement learning in contrast to classical supervised training by gradient descent. Since the deep learning breakthrough in the ImageNet competition many improvements have been made on supervised learning tasks. While some work has been done on using deep learning techniques for reinforcement learning, it has not yet received as much attention as the approaches used for supervised learning. We hope that the creation of ConvNEAT might contribute to this field.

References

Cardamone, L., Loiacono, D., & Lanzi, P. L. (2009). Evolving competitive car controllers for racing games with neuroevolution. In *Proceedings of the 11th annual conference on genetic and evolutionary computation* (pp. 1179–1186).

Hausknecht, M., Khandelwal, P., Miikkulainen, R., & Stone, P. (2012). Hyperneat-ggp: A hyperneat-based atari general game player. In *Proceedings of the 14th annual conference on genetic and evolutionary computation* (pp. 217–224).

Hutter, H. H., Frank, CA, U., & Leyton-Brown, K. (2014). An efficient approach for assessing hyperparameter importance. *Proceedings of The 31st International Conference on Machine Learning*, 754–762.

Kohl, N., & Miikkulainen, R. (2012). An integrated neuroevolutionary approach to reactive control and high-level strategy. *IEEE Transactions on Evolutionary Computation*, *16*(4), 472–488.

Koutník, J., Schmidhuber, J., & Gomez, F. (2014). Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. In *Proceedings of the 2014 annual conference on genetic and evolutionary computation* (pp. 541–548).

Kuipers, B., Ghosh, J., & Stone, P. (2009). Learning in fractured problems with constructive neural network algorithms.
(Dissertation)

Lis, M. (2013). *pyqt_snake.* `https://github.com/mlisbit/pyqt_snake`. GitHub. (b95cd9baf6f)

McIntyre, A. (2016). *Neat-python.* `https://github.com/CodeReclaimers/neat-python`. GitHub. (2162f3f72c1c)

Risi, S., & Togelius, J. (2014). Neuroevolution in games: State of the art and open challenges.

Schrum, J., Karpov, I. V., & Miikkulainen, R. (2011). Ut$\hat{2}$: Human-like behavior via neuroevolution of combat behavior and replay of human traces. In *2011 ieee conference on computational intelligence and games (cig'11)* (pp. 329–336).

Stanley, K. O., Bryant, B. D., Karpov, I., & Miikkulainen, R. (2006). Real-time evolution of neural networks in the nero video game. In *Aaai* (Vol. 6, pp. 1671–1674).

Stanley, K. O., D'Ambrosio, D. B., & Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, *15*(2), 185–212.

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