1 Methods

In this section I will describe the methods that I've been using to evolve multitasking agents. Since most of these methods is build atop other methods, I've decided to start off with a description of the basics. I then use this as a foundation to describe the next methods that is building upon that. This bottom-up approach is then repeated until I've described all the methods used in this project.

1.1 Evolutionary Algorithms

Evolutionary Algorithms (EA) is a group of algorithms that are based on implementing the principles of evolution. The general idea is to mimic real life, and let the algorithm "discover" the best solutions. This is done by having a population of solutions be evaluated on a defined fitness parameter. We then let the fittest solutions survive, and combine them with another fit solution to produce a new offspring solution. The offspring then has a possibility to mutate and is placed in a new generation. We then repeat the procedure on the new generation until we have a good enough solution.

An overview of the algorithm can be seen in figure (x) and the following will give a more in-depth explanation of the components used to implement an EA.

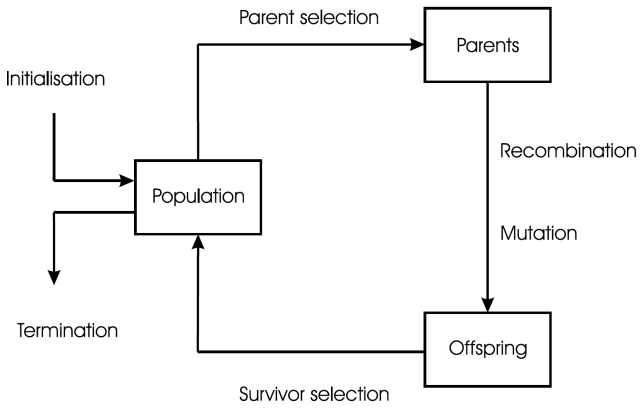

Figure 1: [ref Eiben-Smith]

1.1.1 Initialization

To start using EA, we first need to have an initial population of individuals and decide how we want to represent these individuals. The representation of the individuals that the EA works with is called the genotype, and an individual that can be used on the actual problem domain is called the phenotype.

The phenotype space can be different from the genotype space, and choosing a genotype

space is important, since it's here the evolutionary search takes place. The mapping from phenotype to genotype is called encoding and it is possible to have multiple genotypes that encode the same phenotype. The reverse is called decoding, however one genotype can only produce one phenotype.

With a representation chosen we can now initialise the first population. The initialisation of an EA is often kept simple, and the first population is therefore mostly generated randomly [p.23 Eiben Smith].

1.1.2 Evaluation and Parent Selection

With an initial population in place we now want to evaluate the whole population and choose which individuals will be used for parents. To do this, we introduce a fitness function. The fitness function is usually applied in the phenotype space (as it's here we want to find the best solution). The fitness of the genotype is therefore assigned by first decoding the genotype and then applying the fitness function.

Dependent on the problem being a minimisation or maximization problem we value the individuals with lowest or highest fitness values respectively. The idea for parent selection is to let the higher quality individuals be the base for the new generation. However we don't want to make the selection solely on the fitness, but instead use a probabilistic approach where individuals with higher fitness still gets favoured. This means that other "weaker" individuals still get a chance to get picked. This is done to avoid the search being greedy and getting stuck on local optimum [p.21 Eiben-Smith].

1.1.3 Recombination and Mutation

Recombination merges two genotypes into one or more genotypes, and the way parts from the parents gets chosen, is decided in a randomly fashion. It is possible to use more than two parents but is not common as doesn't have a biological equivalent [p.22 Eiben-Smith]. Using only one parent is called a-sexual reproduction and can produce a different offspring dependent on the variation operator [ref?].

With a new offspring created there is a chance that a mutation might happen. A mutation will change the offspring's genotype in random unbiased way. This is done to fill the gene pool with "fresh blood" [p. 21 Eiben-Smith].

1.1.4 Survivor Selection

Survivor selection is similiar to parent selection, but is not stochastic. Instead it selects in a deterministic way by choosing from the unified multi set of parents and offspring the one with highest fitness.

1.1.5 Termination

When to terminate the EA can be chosen in many different ways. Some examples include: set a limit of number of generation or check if the diversity in the population drops under a certain a threshold.

1.2 Neuroevolution

Neuroevolution is evolutionary algorithms applied on artificial neural networks (ANN). In the following I will give an overview of what an ANN is and how they work.

1.2.1 ANN

ANN is a model that is inspired by biological neural networks - as observed in the brain. ANN is a collection of interconnected neurons and can be thought of as messages getting exchanged between each neuron. A single neuron can be defined in the following way.

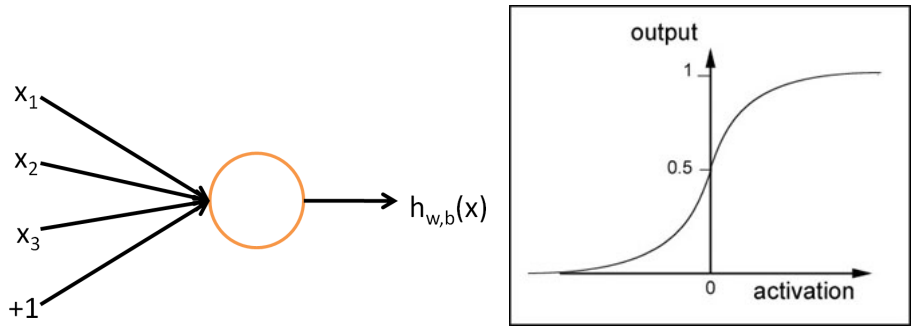

Figure 2: [ref http://ufldl.stanford.edu]

As seen in figure (x), the neuron takes a number of inputs given by vector X and add them together. It then inputs the sum into the the function h - also called an activation function. The idea with the activation function is... [ref].

The activation function can vary depending on the setup but often a sigmoid function is chosen. The one input to the neuron that is 1, is a bias and can be grouped together with the vector X by setting $X_0 = 1$.

With the definition of a single neuron we can now connect them together with other neurons to get a larger network of neurons. However before we do that, we rewrite the input vector X as the product of a weight w and the output z of another neuron.

Each neuron can then be computed in the following way.

$$z_k = h\Big(\sum_j w_{kj} z_j\Big) \tag{1}$$

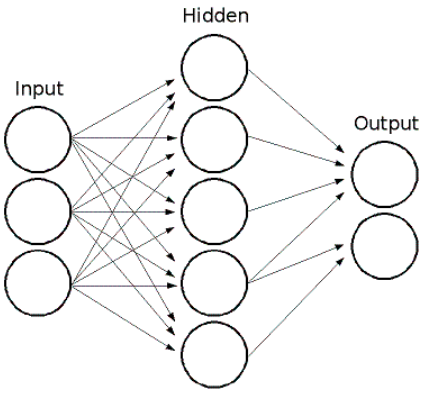

Figure 3: [ref - don't use this]

ANN has three different types of layers as shown in figure (x). An input layer that takes the input often along with a bias. A hidden layer that connects the input layer to either the output or a new hidden layer. An output layer then uses the connections from the hidden (and input - if any connection) layer to produce the output.

It's possible to have cyclic connections, ex. a neuron connected to itself, but I won't be focusing on these, as the networks I train in this project is feed forward and don't have any cycles.

Neural networks can also be used in standard classification task by feeding the network an input and compare the output with a target output using an error term. This error term can then be minimized by setting the weights using the back propagation algorithm. However in neuroevolution we don't set the weights using back propagation, but instead let an evolutionary algorithm decide.

1.3 NEAT

Now that I've given a brief description of both evolutionary algorithms and neural networks, it's time to put the things together and use it to evolve neural networks. To evolve the networks I use NeuroEvolution of Augmenting Topologies (NEAT), since it manages to handle many of the problems that arise when evolving neural networks. Unlike other algorithms that has a fixed structure (ex. 1 hidden layer), NEAT starts of with a minimum structure and evolves by the use of historical markings and speciation. A more detailed explanation of how NEAT works is explained in the following.

1.3.1 Representation and crossover

NEAT uses a direct encoding by letting the genotype have two lists representing the network, one for the nodes and another for the connections. The reason NEAT is using a direct encoding instead of an indirect, is that an indirect encoding tend to bias the search in unpredictable ways [stanley p. 102].

One of the problems with doing NeuroEvolution is dealing with the Competing Convention Problem [ref stanley p. 103]. It's a problem that arise when you have multiple genomes that produces the same solution but have different encoding. When doing crossover on these, it's likely to produce damaged offspring - since they are permutation of one another, you could end up getting the same "information" from both parents.

To deal with this problem along with the problem of matching two structurally different networks, NEAT introduces a historical marking for each gene. This is done to keep track of all the genes, and match those that share the same origin. An example of this is given in the following.

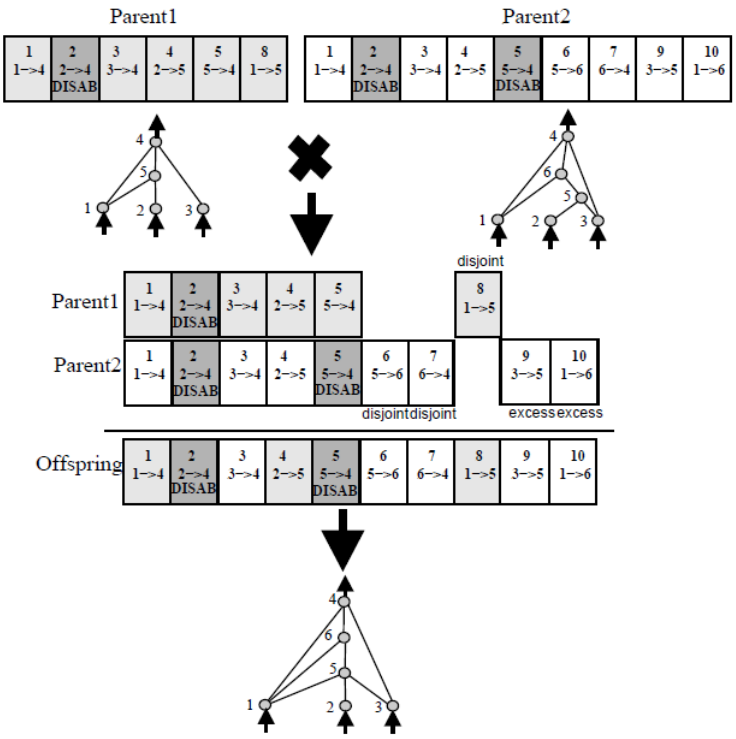

Figure 4: [ref - don't use this]

NEAT has two types of structural mutation, adding a new node or adding a new con-

nection. When adding a new connection, a new connection gene is added to the end of the list and gets assigned the next innovation number. If another genotype gets a similar connection during the same generation, NEAT will give it the same innovation number. When adding a new node, a chosen connection gene will be disabled and split into two new connection with each new connection having new innovation numbers.

When crossing two genotypes, the genes that have identical innovation numbers are matched up, and those that don't match up is categorised into either disjoint or excess genes. The disjoint genes are those that don't match in the middle and the excess are those that don't match in the end. For the matching genes the offspring chooses randomly from both parents, and for the disjoint and excess genes the more fit parent is chosen.

1.3.2 Species

Another key feature of NEAT is the introduction of species. The idea of having different species is to protect structural innovation. Since an initial change in topology will likely decrease the fitness, the network would most likely not survive the next generation. Therefore NEAT groups the networks into different niches to protect different topologies. To do this, NEAT uses the number of excess E and disjoint D genes as a measure for their compatibility distance. The compatibility distance is normalized with regards to the number of genes N and also uses the difference in matching weights $\overline{W}$. The three coefficents c_1, c_2, c_3 is used to weight the importance of the three factors [stanley p. 110].

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{2}$$

Each species is represented by a random genome inside the species. A genome g is placed into the first species it's compatible to, if the the distance measure δ between g and the representive genome for that species is within a certain threshold δ_t. If g is not compatible with any species, a new species will be created with g as a representative.

To avoid letting one species take over the entire populations, NEAT uses an explicit fitness sharing. This is done by introducing an adjusted fitness f_i' that is the normal fitness f_i divided by the number of genomes in the species that the target genome exist in. For reproducing the species first eliminate the lowest performing genomes in the population and then let the entire population be replaced by the new offspring.

1.3.3 Minimum structure

NEAT has an initial structure of no hidden neurons, and so new topology is gained with mutation. This has the advantages of the dimensionality staying low, and strengthen the analogy to nature, where something simple gets evolved into something more complex during evolution.

1.4 HyperNEAT

HyperNeat extends the NEAT method, by encoding the network by another network called a Compositional Pattern Producing Network (CPPN) which is then evolved using NEAT. HyperNeat uses a substrate which usually is defined as a 2 dimensional plane that ranges from -1 to -1 on both x and y axis. You then place the neurons for your final network on the substrate and let the connection weights be determined by querying the CPPN.

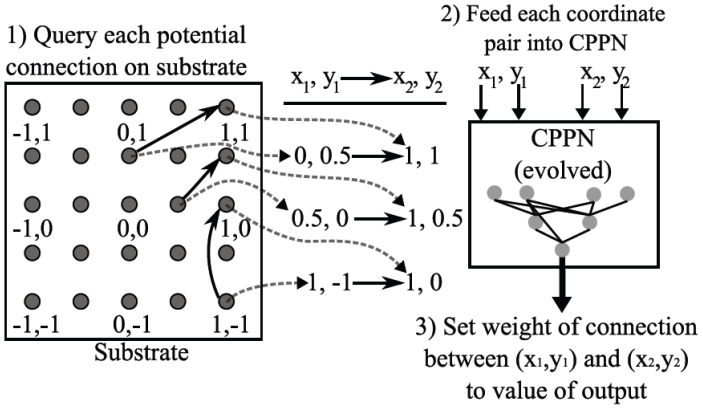

Figure 5: [ref don't use this]

To get the weight between neuron a and neuron b, you use a's position in the substrate a_x, a_y as the first two parameters, and b's position b_x, b_y as the last two parameters for the CPPN. The output from the CPPN is then used as the weight between neuron a and b. Figure x shows an example of this.

1.4.1 Compositional Pattern Producing Network (CPPN)

CPPN uses an abstraction that is based on patterns that are typical seen in nature. This include: repetition, repetition with variation, symmetry, imperfect symmetry, elaborated regularity and preservation of regularity. As an example the human body is symmetric along the center of the body and has repetition with variation for the fingers on hands and feet.

With development encoding you use local interaction and temporal unfolding in simulations to mimic the development in nature. However with CPPN the development encoding is abstracted away and CPPN are instead using functions to achieve the patterns mentioned above. The idea is that the phenotype can be viewed as a distribution of points in a multidimensional Cartesian space and that any phenotype produced through a temporal progression (development encoding) is possible to be represented using a functional description. To get symmetry, a Gaussian function or $f(x) = abs(x)$ can be used (since they are mirrored around the y-axis). For repetition, functions like sin(x) or

modulus can be used.

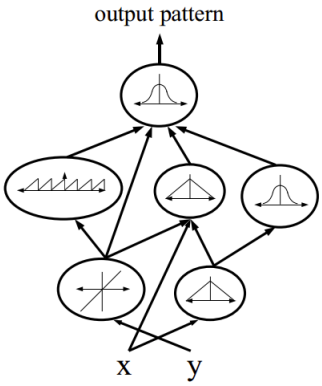

Figure 6: [ref don't use this]

These functions can then be put together in composition to get more advanced phenotypes. This is done using CPPN-NEAT which is a modified version of the NEAT algorithm. The reason that NEAT can be modified is that artificial neural networks (ANN) and CPPN structurally are the same.

1.5 Single Unit Pattern Generator (SUPG)

To generate oscilliations I will be using Single Unit Pattern Generator (SUPG) by Morse & Risi [ref].

Single Unit Pattern Generator (SUPG) is a special neuron that is placed on the hyperneat substrate, and works as a modified version of the CPPN. The unique thing for the SUPG, is that along with the substrate position, it also takes a linear signal as a input. The linear input works as a timer, and is increased by each clock tick. The time span from 0 to 1 and can be reset at any time using a trigger. The trigger is a way for the oscilliation to be dependent by events in the world - so that when a trigger is fired the cycle will reset and the timer will be set to 0. The main motivation for the SUPG, is that since CPPNs are good at encoding spatial pattern, it might also be good at encoding patterns across time.

SUPG was introduced by Morse & Risi [ref], where they used them to evolve gaits for a four-legged virtual robot. The robot had three motors for each leg (knee, hip in/out, hip forward/back). They then spread the different motor types across the substrate so that each motor type is grouped together with other legs that have the same type. The reason for this, is that hyperneat uses an indirect encoding, so that neurons grouped closely together will ensure that each motor type produce similar motion trajectories.

For the four-legged robot the trigger is getting reset every time a leg touches the ground.

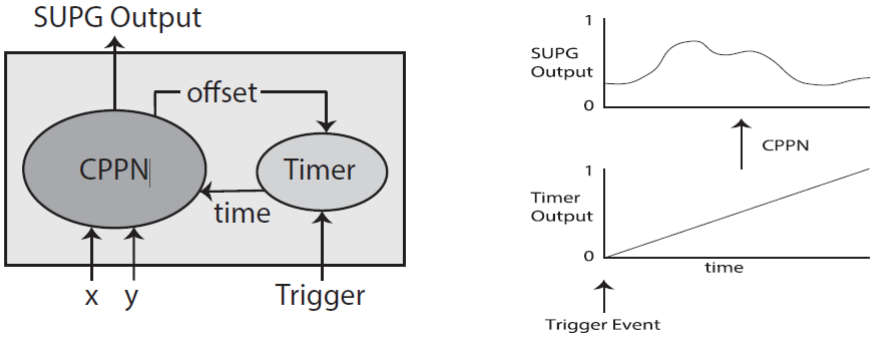

Figure 7: [ref don't use this]

1.6 MM-NEAT [ref]

In this paper they train nine different modular networks to play Ms. Pac-Man. The nine methods can be grouped into three main approaches.

The first one is called a multitasking network. The idea is to choose how many modules the network should have and how it should switch between the modules. Each module has its own policy output neurons, and it's up to the user to tell the network when it should switch between the policies. This approach is biased, since it's up to the user to choose the number of modules and how to split the tasks. So to avoid how to split the task, another network is used which introduces an extra neuron for each module called a preference neuron.

The idea is to let the preference neuron decide which module to choose. This is done by choosing the module with the preference neuron of highest value. Although this approach removes the human-bias on how to switch between tasks, the user still has to decide on how many modules should be used. So to account for this, a third approach called module mutation is used.

Module mutation starts with a single module and new modules with policy and preference neurons are then later added. There are different ways to add a new module. MM(P) lets the module be connected to a previous module outputs. MM(R) randomly chooses input links used for the policy and preference neurons. MM(D) duplicates the same links used for previously policy neurons, and choose a random link for the new preference neuron.

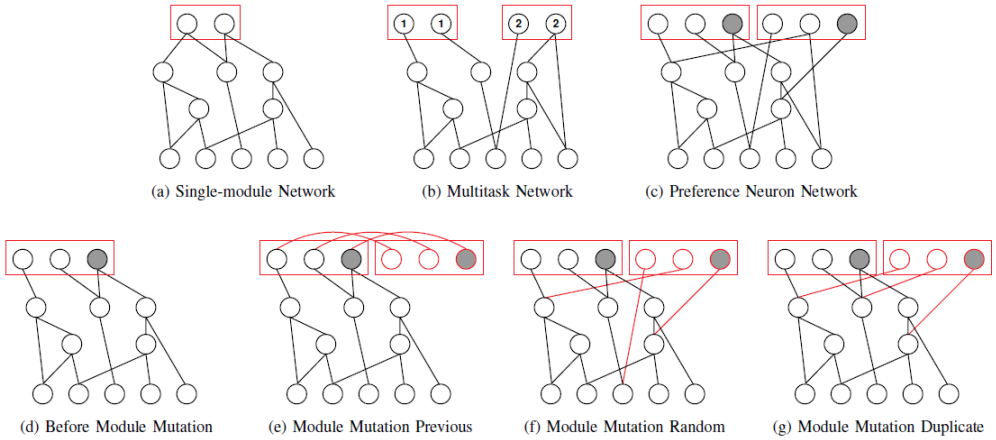

Figure 8:

1.7 MB-Hyper-NEAT [ref]

As a comparison for how the new methods perform, I will be using an extension to Hyper-NEAT called MB-Hyper-NEAT by Schrum[]. MB-Hyper-NEAT is inspired by the direct encoding MM-NEAT, which introduces methods to generate modules to NEAT. The idea with MB-Hyper-NEAT is to use multiple "brains" (MB) by encoding multiple networks on different substrates. In the group of MB-Hyper-NEAT these networks can be generated in different ways, however I will only be focusing on two of the methods: namely Multitask CPPNs and Preference Neurons.

1.7.1 Multitask CPPNs [ref]

Recall that a regular CPPN has one output to query weights and another output for bias. This setup is fine if we only want to encode one network, however for Multitask CPPN we are interested to encode multiple networks. Therefore Multitask CPPN introduces n extra pairs of weight/bias-outputs, that can be used to encode the same nodes on the substrate but with different weights and biases. This way Multitask CPPN's generate different networks, so that one network can be used for one task and another network for a different task. However when to switch between the different networks must still be specified by a human - which can be problematic as it require some insight into the domain and can also bias the final result. This problem is addressed in the other MB-HyperNEAT method using Preference Neurons.

1.7.2 Preference Neurons [ref]

This method builds upon multitasking CPPN, and introduces a way that let HyperNEAT discover when to switch between the different networks. This is done by having an extra output on the substrate network called a preference neuron. The preference neuron exist

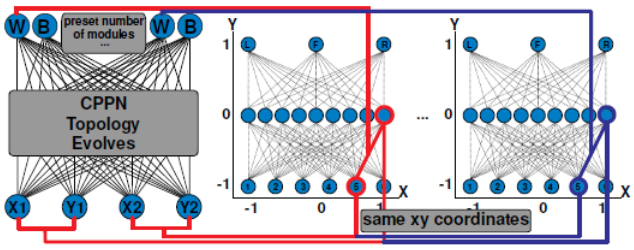

Figure 9:

on each network, and the one with the heighest input is the network that get chosen. The connections weights to the preference neuron gets encoded by it's own output from the CPPN. An example can be seen in figure (x).

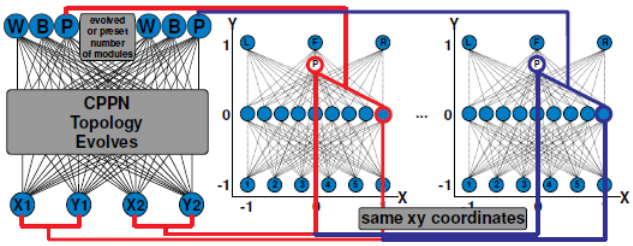

Figure 10: