Decision Making Under Uncertainty DAT235

RL Competition 2014 - Helicopter

Peng-Kun Liu	920102T095	pengkun@student.chalmers.se
Pedro Matias	9302087433	mpedro@student.chalmers.se

January 22, 2015

Contents

1 Our solution to the helicopter domain

After having tried value function methods (see section 2), we switched to policy search approaches and got much better results. The main problem with our previous attempts was to find a good resolution of both states and actions spaces. In this approach, we don't need to discretize any space, because there is no value approximation for state-action pairs and the next action is automatically given to us without the need of choosing from a set of actions.

Our approach is based on a **neuroevolution** algorithm (see [2]). The idea is to apply a genetic algorithm to evolve neural networks.

1.1 The algorithm

Our algorithm follows a policy search method, by means of a genetic algorithm that evolves neural networks. Each of this networks corresponds to a genome which represents a policy: the approximation function maps from *states* to *actions*.

The general genetic algorithm has a series of independent stages, so one can have a slightly modified algorithm, by trying different implementation methods for each stage.

1.1.1 Genetic algorithm

Stages of the genetic algorithm and our implementations follow:

1. **Initial seed**

   Corresponds to the original neural network. See subsubsection 1.1.3 for more information on the kinds of neural networks tried.

2. **Initial generation**

   Responsible for creating the first generation given the seed. The resulting population is a series of genomes mutated from the initial seed (see mutation below).

3. **Genome choice**

   Because one genome corresponds to a policy, it is used throughout one episode. As long as there are unevaluated genomes (policies) in the current generation, one of them is chosen randomly. If all genomes have been evaluated, the next generation is computed.

4. **Next generation**

   We tried the following two methods:

i. N pairs of individuals are selected (N is the population size) and for each pair we generate a new genome through crossover and, if it happens, mutation.

ii. We assume $N = 20$. We then select the 6 best individuals from the current population and generate the new offspring for all possible pairs of individuals - $\binom{6}{2}$. The next generation is then composed of the offspring (after crossover and mutation) and the 5 best genomes of the previous population.

- **selection**

  Stage responsible to select pairs of individuals during the next generation stage. We tried both:

  i. roulette wheel method: choose the pair of individuals with a probability proportional to its fitness.

  ii. best individual: choosing the fittest pair of individuals.

  – **fitness**: since each individual corresponds to a policy used during one episode, the fitness of each individual is proportional to the average reward at the episode end.

- **crossover**

  The new genome is a neural network with the same topology as the parents, but each connection weight is a linear combination of the corresponding weights of the parents: $w(\text{offspring}) = \alpha \cdot w(\text{parent}_1) + \beta \cdot w(\text{parent}_2)$ and $\alpha + \beta = 1$. We tried several values of α and β, both constant and dynamic. The best approach was to choose the values proportional to the parents fitness: the fittest the individual the closest it is to the offspring.

- **mutation**

  Each gene in the genome mutates with a probability of `mutation_rate`. A gene corresponds to a connection weight in the neural network and the mutation is a new gene with distribution $w_{\text{new}} \sim \mathcal{N}(w_{\text{old}}, \texttt{mutation\_variance})$.

1.1.2 RL-Glue agent implementation

We give a high-level overview of the required function implementations of the *rl-glue* agent.

---

function AGENT_INIT

```
    generation = initial_generation(seed)
end function
```

```
function AGENT_START(observation)
    genome = choose_genome(generation)
    return agent_step(0, observation)
end function
```

```
function AGENT_STEP(reward, observation)
    if observation is far from optimal observation (1) then
        next_action = weak_controller.next_action()
        genome.train(observation, next_action) (2)
    else
        next_action = genome.next_action()
    end if
    return next_action
end function
                                    ▷ (1)assuming optimal state is {0}[12]
                                    ▷ (2)training with back propagation
```

```
function AGENT_END(reward)
    genome.fitness = average_reward()
    if there are no unevaluated genomes then
        generation = next_generation()
    end if
end function
```

The underlined methods correspond to the implementations of the genetic algorithm stages, described above.

Note: In the very first episode we train the seed neural network using backpropagation under supervision of the *weak baseline controller*. Only then we start using the genetic algorithm. Before the training, the initial weights of the network are randomly assigned. We also tried skipping the training of the network, but, not surprisingly, we didn't achieve good results.

1.1.3 Neural network topologies

In this section we describe the kinds of neural networks used: neural networks A and B, and, in the end of this section, a neural network with a dynamic topology.

1.1.3.1 Neural Network A

The topology of this network is described in Table 1

Neural network A

layers	*activation function*	*#neurons*
input	linear	12
hidden	linear	12
output	hyperbolic tangent	4

Table 1: Topology of Neural network A (NN-A). Between layers, all neurons are connected to each other.

In the following plots you can see how the average reward (calculated in the end of each episode) evolves as more and more episodes are run. We tested in a total amount of 500 episodes, given that it takes already a substantial amount of time to compute. From episode to episode the current population is kept.

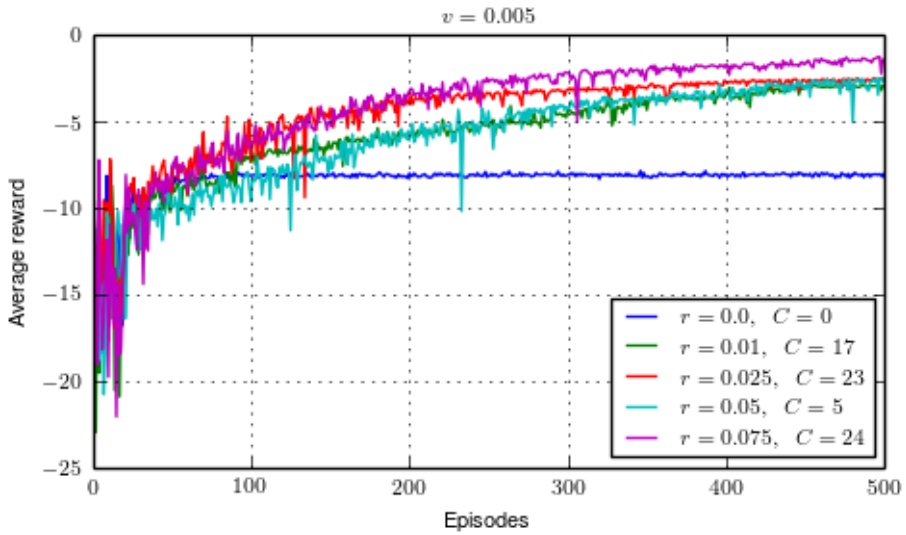

Figure 1: Average reward of NN-A for different `mutation_rate`=r and `mutation_variance`=v. C corresponds to the number of times the helicopter crashed during all episodes (for simplicity, the average reward is not displayed when the helicopter crashed).

As we can see from the plot in Figure 1, the larger is the mutation rate r, the higher is the average reward. However, as r increases, does also the time of convergence. On another hand, if r is large, the average reward obtained in the first episodes is much more unstable. In general, the number of crashes C also increases with r. Not surprisingly, $C = 0$ when $r = 0$: if there are no mutations, the helicopter behaves according to the weak controller policy. The average reward in this case is quite low.

Since larger values of mutation rate appear to be "better", we decided to try plotting the average reward for different values of `mutation_variance`, see Figure 2.

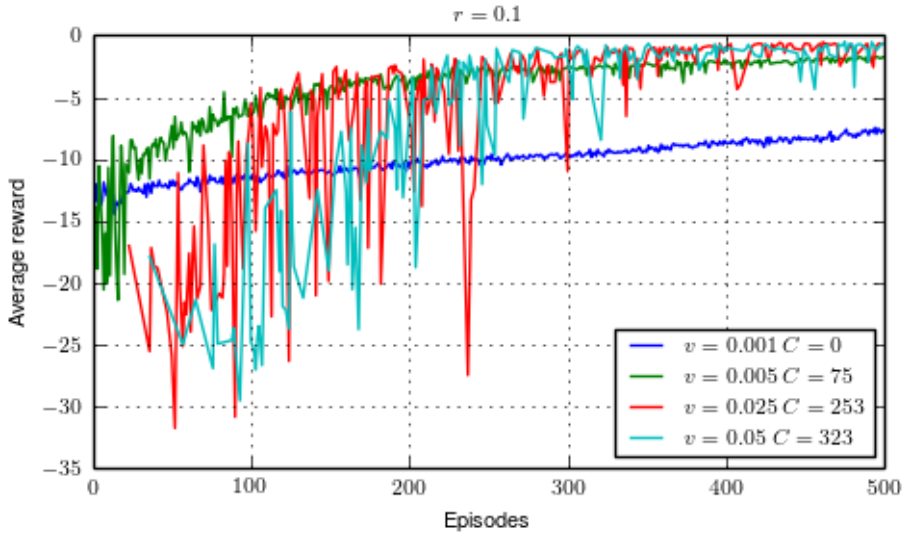

Figure 2: Average reward of NN-A for different `mutation_variance`=v and `mutation_rate`=r. C corresponds to the number of times the helicopter crashed during all episodes (for simplicity, the average reward is not displayed when the helicopter crashed).

By analysis of the plot in Figure 2, we can conclude the same facts stated in the analysis of Figure 1 (both in convergence and stability, except using `mutation_variance` as the argument). We can also see, more clearly, that the higher is the `mutation_variance`, the higher is the number of crashes C, specially for $v = 0.05$, where the helicopter crashes more than half times.

We also tried a neural network with the same topology as NN-A, but with a linear activation function in the output layer. However the results were worse.

1.1.3.2 Neural Network B

The topology of this network is described in Table 2.

In order to test this topology, we considered 3 different `mutation_rate` and, for each of this 3 different `mutation_variance`. The results of running the experiments are in Figure 3.

Neural network B

layers	*activation function*	*#neurons*
input	linear	12
output	linear	4

Table 2: Topology of Neural network B (NN-B). Between layers, all neurons are connected to each other.

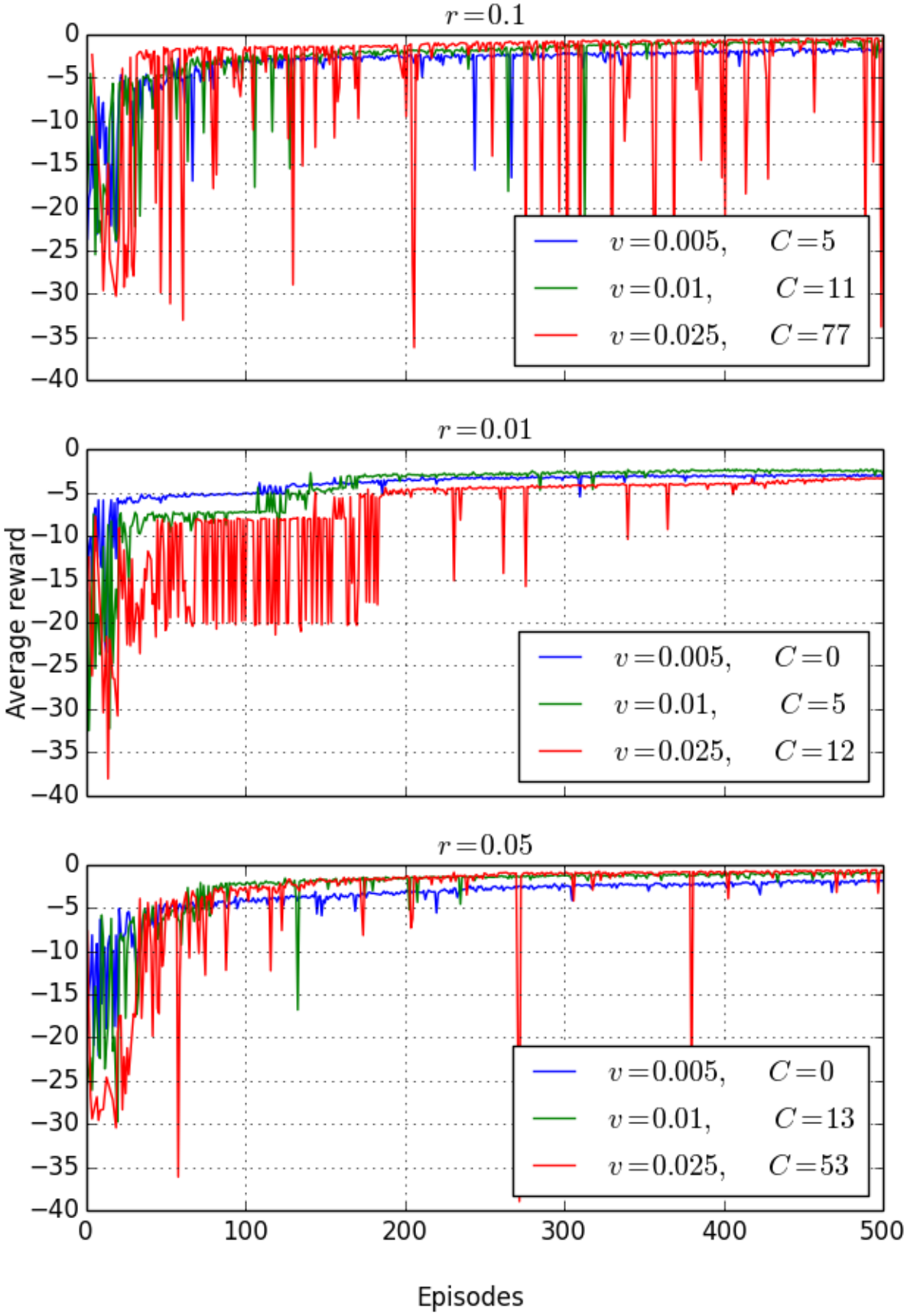

Figure 3: Average reward of NN-B for different `mutation_variance`=v and `mutation_rate`=r. C corresponds to the number of times the helicopter crashed during all episodes (for simplicity, the average reward is not displayed when the helicopter crashed).

From this results we can say that:

- NN-B converges faster than NN-A
- NN-B is in, general, more stable than NN-B, despite the average reward plots presenting some "spikes" with greater values of `mutation_variance`
- NN-B has less number of crashes C (if one compares similar parameters, e.g. $r = 0.1$ and $v = 0.005$ or $v = 0.025$)
- The average reward in the last episodes is quite good and similar to NN-A

The fact that NN-B has a simpler topology (only 48 neuron connections) explains the fast convergence and the existence of "spikes" in the plot when `mutation_variance` is big - a mutation is much more effective in NN-B, given that the number of neuron connections is much smaller. However, a mutation also happens less often in NN-B, explaining perhaps the lower number of crashes C.

We can observe that a topology as simple as this is enough to achieve a good average reward in the end of the episodes.

Nevertheless, the fact that we got different results for different topologies suggested us that we could try combining topologies, see next section.

1.1.3.3 Dynamic topology

The fact that NN-A has a slower convergence rate suggests us that its topology might find policies that are given better rewards in future episodes. By combining this idea with the fact that NN-B is a fast and stable converger, one might start the policy search with NN-B and then, later in the future, switch to NN-B. The results of running the experiment are in Figure 4. For comparison purposes, we run again experiments for the static topologies NN-A and NN-B.

Following is the new implementation of the *rl-glue* function called in the beginning of each episode:

function AGENT_START(*observation*)

if last average reward > -2 **then**

genome = new neural network with topology NN-A

genome.train(past observations and actions) [(1)]

$r = 0.01$

$v = 0.001$

else

$r = 0.05$

$v = 0.01$

genome = choose_genome(generation)

end if

return agent_step(0, *observation*)

end function

▷ [(1)]training with back propagation

As we can observe in Figure 4, we achieved quite good results: the number of crashes C dropped significantly and the average reward is higher. After having found a stable policy, the switch to a more complex network is the only way one can improve the average reward. Moreover, after the switch the current neural network is very sensitive to small changes. Hence, both the mutation rate r and the mutation variance v have to decrease.

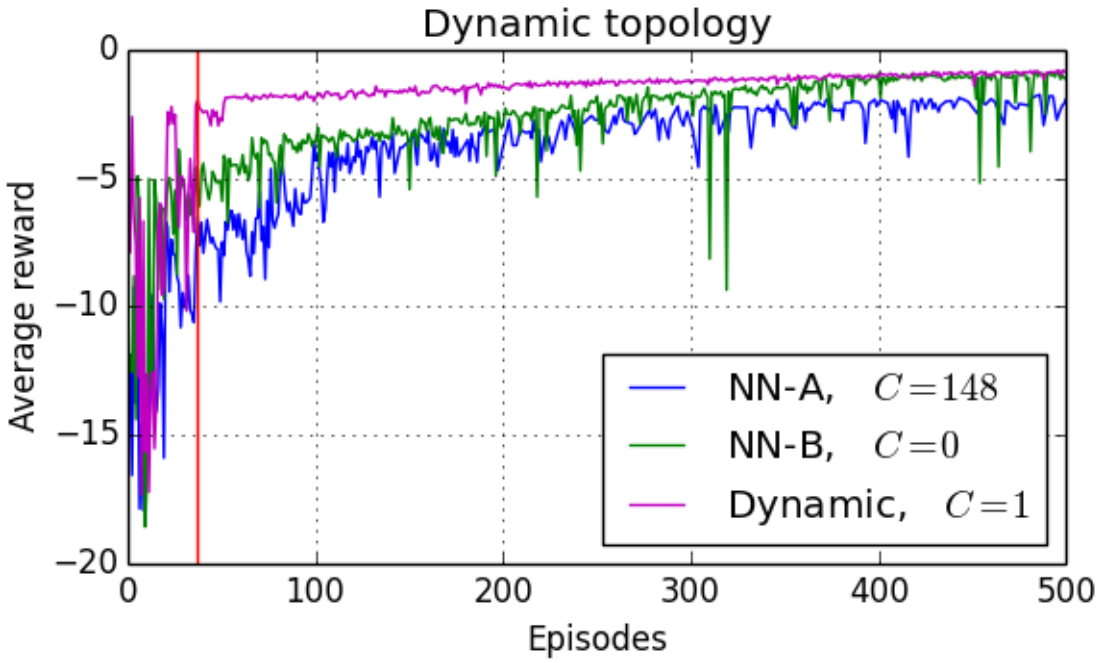

Figure 4: Average reward of networks with (i) dynamic topology, (ii) static topology NN-A and (iii) static topology NN-B. Before the red line, in the dynamic network: topology=NN-B, $r = 0.05$, $v = 0.01$. After the red line: topology=NN-A, $r = 0.01$, $v = 0.001$. The values of r and v are the same for the corresponding static topologies.

C corresponds to the number of times the helicopter crashed during all episodes (for simplicity, the average reward is not displayed when the helicopter crashed).

2 Previous approaches

In the beginning, we decided to try value function approaches. Overall, the main difficulties encountered were the fact that the space is continuous and the huge space dimension that results from combining both states and actions spaces. We tried both direct *kernel-function approximations* and *Q-learning* (see [4]).

2.1 Kernel-function approximations

We tried approximating both the *reward* and *transition* functions using the following methods:

2.1.1 Least squares

Given our inexperience in the subject we started by just trying simple variations of linear regressions using the least squares method for both:

- *transition function*: $s_{t+1} = s_t + A \cdot a$, where A is a 12×4 matrix of weights (to be approximated) and a is the vector representing the action to take
- *reward function*: $r = R \cdot s_t$, where R is the vector of weights to be approximated.

We quickly realized that this method wouldn't work. By still assuming smoothness of the transition function, we tried the following: approximate a transition function $s_{t+1} = s_t + A \cdot a + N$ where N is now a vector representing the (non-linear) influences of the environment and we assume this vector does not change drastically. The overall idea would be to approximate a non-linear function by a series of subsequent linear functions in intervals of Δ timesteps. To get N for a given period we just send a null action and observe the new state, giving us $N = s_{t+1} - s_t$. For the next Δ timesteps we consider that N is constant and we use linear regression for updating A. However it is quite easy to see how this policy can be so bad, specially taking into account the huge state space. We never did more than 20 steps.

2.1.2 Using Neural networks

We used two neural networks, each trained with backpropagation under supervision of the *weak baseline controller*. The networks were used to approximate both the transition and reward functions. We also didn't got good results since we didn't find a smart way to choose actions from its continuous space. Here, we tried sampling over a gaussian distribution with the bell

curve centered in the action that would be taken under the weak controller policy. Besides, the topology of the network can have great influence in the results. Although, the results were not consistent, the number of steps was improved to 1000.

2.2 Q-learning

The obvious main problem here was to find good ways to solve the space continuity problem (both in states and actions).

For the **state space** we tried 1-tiling discretization, but it was difficult to find a good resolution. If it's very low, the results are not accurate at all. If it is high, we might not be able to store all Q-tables entries and, most of the times, each entry is never updated, because it is computed once (and randomly).

For the **action space** we tried both 1-tiling discretization and sampling with gaussian distribution centered in the last action. The intuition behind this was the fact that in real life, consecutive actions do not change drastically.

We tried several combinations of "resolutions" (of the both spaces above) and discount values (γ). Moreover, we also tried these combinations with SARSA and Q-learning with *controllability* (see [1]). Besides, we also tried this method using *eligibility traces* (see [4]). However, overall, all the results were not satisfying: we never did better than 40 steps.

2.2.1 Neural Network approximation

To avoid the resolution problem in the state space, we tried approximating the q-learning function using a neural network trained trained with backpropagation under supervision of the *weak baseline controller*. However, it was still difficult to properly discretize the action space and choose an action that maximizes the computed value.

3 Conclusion and further work

Based on what we have tried, we conclude that it is very hard to start from scratch using value function methods. We experienced difficulties in finding a good policy with a high average reward and, at the same time, prevent crashes within the first 1000 episodes. Instead, by searching directly in the policy space, it is much easier to have good results: prevent crashes and at the same time have a high average reward.

Given the results we obtained in the neuroevolutionary approach, we conclude that this method is very sensitive to the following factors: how to crossover genomes, mutate genes and create the next generation. But also, how to initialize the seed and which network topology should be used.

Despite the *weak baseline controller* giving already good results, we noticed that the average reward was quite low, compared to the average reward given by the neuroevolution method. We concluded that the weak controller, despite avoiding crashes in general, doesn't seek to find good states.

It would be interesting to see an algorithm that can be more adaptative and more responsive to the environment, e.g., an algorithm that can guess the network topology that would be the best fit for the current state in the environment. We just considered two fixed network topologies.

Moreover, by having prior-knowledge of this problem, one could more effectively guide the evolution process in order to achieve faster convergences and less crash rates.

References

[1] A. Asbah, A. Barreto, C. Gehring, J. Pineau, D. Precup.
Reinforcement Learning Competition: Helicopter Hovering with Controllability and Kernel-Based Stochastic Factorization. In the Reinforcement Learning Competition, 2013

[2] R. Koppejan, S. Whiteson.
Neuroevolutionary reinforcement learning for generalized control of simulated helicopters. Evol. Intell., 4(4):219–241, 2011.

[3] HJA Martín, de Lope J.
Learning autonomous helicopter flight with evolutionary reinforcement learning. 12th international conference on computer aided systems theory (EU- ROCAST), pp 75–82, 2009.

[4] RS Sutton, AG Barto.
Reinforcement learning: an introduction. MIT Press, Cambridge, 1998.