Analysis of Policy Gradient Methods for Neuroevolution of Augmenting Topologies Networks in Reinforcement Learning

Archit Rungta
Department of Mathematics
Indian Institute of Technology Kharagpur
18MA20008

Shreyansh Darshan
Department of Mathematics
Indian Institute of Technology Kharagpur
18MA20044

***Abstract*—In this paper we consider the application of different methods developed to improve stochastic policy gradient algorithms to Neuroevolution of Augmenting Topologies(NEAT) [SM02]. NEAT is a genetic algorithm developed to evolve neural network topologies and weights. Policy gradient [Sut+00] is a very interesting algorithm developed to maximize non-differentiable functions. While Neural Networks are extremely good at optimizing a large array of functions, they require gradients. This is where Policy Gradient comes in. Alternatively, NEAT is also able to optimize Neural Network based function approximators without requiring gradients. In the past decade, many new algorithms [Zha+11; Sil+14; PS08] have been created that improve the stability and performance of Policy Gradients. We apply a subset of these methods and analyse the effect on NEAT networks.**

I. Introduction

Today, Neural Networks [BDH96] have emerged as a go-to standard for function approximation. These parameterized approximators are largely inspired from biological brains and can be easily trained to approximate any function through first order derviatives. Neural Networks can now beat human level performance at many tasks such as image classification [ZK16]. However, a major challenge still remains to optimize non-differentiable functions. To this end, a class of methods broadly known as Policy Gradient has been developed. These methods are able to provide an expected value of the gradient for non-differentiable functions. Coupled with Neural Networks, policy gradients form the backbone of modern Reinforcement Learning. Fueled by advancements in computing power and more sophisticated algorithms, modern Reinforcement Learning can now defeat human players in games previously considered unapproachable by computers [Bor16] and programs that took decades of human engineering [Gib17]. Because of a concentration on research in policy gradient based methods to optimize neural networks, we feel that the alternative approach of using Genetic Algorithms as derivative-free optimizer has remained under explored. To that end, we investigate the application of some modern advancements on policy gradients to NEAT evolved neural networks and compare performance.

NeuroEvolution of Augmenting Topologies (NEAT) is a genetic algorithm [SM02], which evolves both the topology and weights of a neural network. While most neuroevolution methods [Yao99] evolve only the weights of a network with fixed topology, NEAT discovers an appropriate topology automatically while evolving weights. This approach is highly effective: NEAT outperforms other neuroevolution methods on many difficult learning tasks [SM02; SM04].

Since the objective of this paper is only to find out the effect of these improvements, we will introduce these changes to the default NEAT algorithm and compare end accuracy as-well as the learning curves. To do this we are using OpenAI gym [Bro+16] environments. The environments are characterized by a vector that represents the state of the environment and an actor in the environment. To solve any environment, the Neural Network, that represents a policy on which the actor acts, must predict which action to take. This has been explained in more detail in II. Whenever the actor takes any action, the environment gives a reward that shows how good the action was and returns a vector corresponding to next state. The goal of Neural Network is to decide actions such that the sum of rewards over the entire trajectory is maximized. Note that this reward function is not differentiable, so we must use NEAT to optimize our networks. The sum of rewards over the entire trajectory averaged over multiple trajectories is taken as the fitness value for any particular genome.

II. Mathematical Preliminaries

A. Markov Decision Process

We first define a Markov Decision Process.
$\mathcal{M} = \{ \mathcal{S}, \mathcal{T} \}$
$\mathcal{S}$: State Space
$\mathcal{A}$: Action Space
$\mathcal{T}$: Transition Operator
$\mathcal{E}$: Emission Probability $p(\mathbf{o}_t|\mathbf{s}_t)$
r: $(\mathcal{S} \times \mathcal{A} \to \mathbb{R})$ Reward Function
Let $\mu_{t,j} = p(s_t = j)$, $\xi_{t,k} = p(a_t = k)$, $\mathcal{T}_{i,j,k} = p(s_{t+1} = i | s_t = j, a_t = k)$, then

$$\mu_{t,i} = \sum_{j,k} \mathcal{T}_{i,j,k} \mu_{t,j} \xi_{t,k} \tag{1}$$

Using the Markov chain we can get

$$\underbrace{p_\theta\left(\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T\right)}_{p_\theta(\tau)} = p\left(\mathbf{s}_1\right) \prod_{t=1}^{T} \pi_\theta\left(\mathbf{a}_t|\mathbf{s}_t\right) p\left(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t\right) \tag{2}$$

B. RL Definition

The goal of RL is to find the optimal policy

$$\theta^\star = \arg\max_\theta J(\theta) \tag{3}$$

where J is

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r\left(\mathbf{s}_t, \mathbf{a}_t\right) \right] \tag{4}$$

The value of J can also be used to find how good our policy is.

C. Final Remarks

We use neural networks to approximate p_θ corresponding to the optimal policy. For the $\arg\max$ step we will be using NEAT in this paper instead of first-order gradient ascent.

III. Methodology

We compare the following different improvements to the standard objective of policy-gradient based reinforcement learning and compare the final values of $J(\theta)$ [4]. Since we can not find the expectation over all trajectories possible, we use a monte-carlo approximation of the expectation as our proxy. Each method describes a modification to the J function that is used to evaluate and find the optimal policy. Our comparison metric will still be the vanilla J as that is the goal of reinforcement learning. In genetic algorithm terminology, the value $J(\theta)$ is the fitness of a genome characterized by θ.

A. Vanilla

This is the standard implementation against which we compare other methods.

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r\left(\mathbf{s}_t, \mathbf{a}_t\right) \right] \tag{5}$$

B. Advantage Standardization

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} \frac{r\left(\mathbf{s}_t, \mathbf{a}_t\right) - \mu}{\sigma} \right] \tag{6}$$

where

$$\mu = \mathbb{E}_{s,r \sim p_\theta(\tau)} \left[r\left(\mathbf{s}_t, \mathbf{a}_t\right) \right] \tag{7}$$

$$\sigma^2 = \mathbb{E}_{s,r \sim p_\theta(\tau)} \left[r\left(\mathbf{s}_t, \mathbf{a}_t\right)^2 \right] - \mathbb{E}_{s,r \sim p_\theta(\tau)} \left[r\left(\mathbf{s}_t, \mathbf{a}_t\right) \right]^2 \tag{8}$$

C. Reward To Go

Inspired from [MT03]. Instead of directly optimizing J over the entire trajectory, we change optimization form to the following.
Before

$$J_i(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r\left(\mathbf{s}_t, \mathbf{a}_t\right) \right] \tag{9}$$

to

$$J_i(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=i}^{T} r\left(\mathbf{s}_t, \mathbf{a}_t\right) \right] \tag{10}$$

Thus, now the fitness value at a particular point in trajectory depends only on the rewards obtained after that point and not on the previous rewards of the trajectory.

D. Discount Factor

Inspired from [MD08].

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} \gamma^t \cdot r\left(\mathbf{s}_t, \mathbf{a}_t\right) \right] \tag{11}$$

γ is a number chosen from $[0, 1]$ and called the discount factor.

E. Monte Carlo Approximation

Inspired from [Lev18]. All the of the previous equations depend on $\mathbb{E}_{\tau \sim p_\theta(\tau)}$. However, taking the expectation of this variable is not practically feasible because of high dimensionality and continuous state spaces. As such we use the below approximation for all expectations based on Monte Carlo method.

$$\mathbb{E}_{\tau \sim p_\theta(\tau)} \left[f(\tau) \right] = \sum_{i=1}^{N} \frac{f(\tau)}{N} \tag{12}$$

Here τ are trajectories sampled from p_θ. N is a pre-decided constant that determines the accuracy of our expectation. It is often referred to as batch size and that is what we will mean by batch size from here on wards.

IV. Implementation

For implementing these different methods we rely on multiple libraries: neat-python [McI+], OpenAI-gym [Bro+16], and pytorch [Pas+17]. We test these different methods on 3 control environments: CartPole-v1[BSA83], LunarLander-v2, and Pendulum-v0. The three environments are in ascending order of difficulty.

The below algorithms constitute a rough sketch of the training process. Note that we have not detailed the NEAT

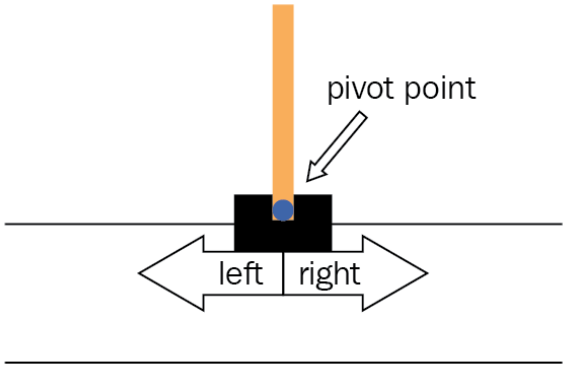

Fig. 1. CartPole-v1 environment

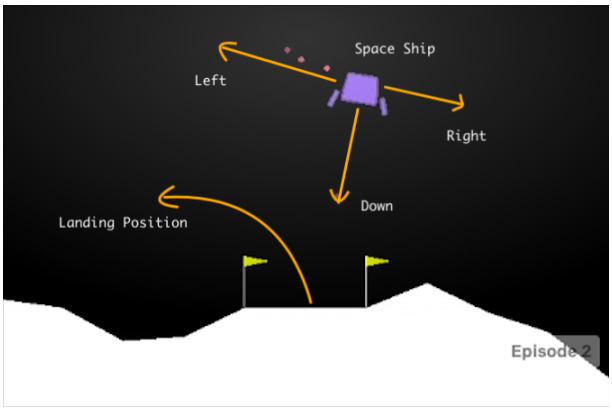

Fig. 2. LunarLander-v2 environment

genetic algorithm. Implementation for that can be referred from [SM02].

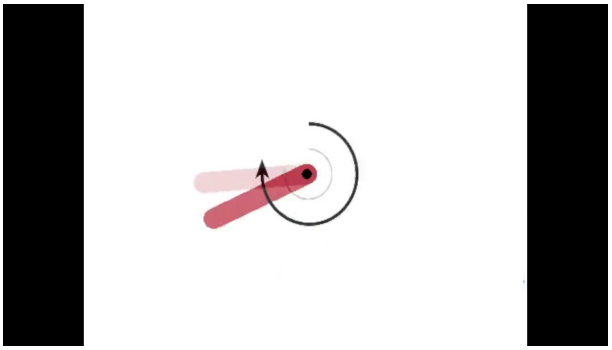

Fig. 3. Pendulum-v0 environment

Algorithm 1: Evaluates genome with environment and batch size returning fitness

```
function fitness;
  Input  : Genome, positive integer batch size=N, env, J(θ)
  Output: fitness
fitness ← 0
for i ← 1 to N do
    while env.state() ≠ env.finish() do
        action ← genome.getaction(env.state())
        reward ← env.step(action)
        fitness ← fitness+J(action, env.state(), reward)
    end
end
return fitness
```

Algorithm 2: Runs the genetic algorithm to create genomes and evaluate fitness

```
function train;
  Input  : positive integer batch size=N, env, J(θ), config for NEAT
  Output: best genome and fitness
current_gen ← 1
best_fitness ← 0
previous_gen_genomes ← ϕ
previous_gen_fitnesses ← []
while config.num_of_gens < current_gen do
    current_gen ←1+ current_gen
    genomes = NEAT.get_genomes(previous_gen_genomes, previous_gen_fitnesses, config)
    current_gen_fitnesses ← []
    for genome in genomes do
        current_gen_fitnesses.append(fitness(genome, N, env, J))
        best_fitness ← max(best_fitness, current_gen_fitnesses.last())
    end
    previous_gen_genomes ← genomes
    previous_gen_fitnesses ← current_gen_fitnesses
end
return best_fitness
```

We have abstracted some of the details away for multi-threading and detailed logging using TensorBoard. The exact code used for this paper can be found at here. However, the above two algorithms suffice to understand the implementation.

The only changes for different configurations is to change the inputs of env and J to Algorithm 2. All computation was done on 16-core AWS EC2 m5.4xlarge server with 16 different configurations running in parallel for any fixed environment. The data logging was done using Tensorboard.

V. Results

For results, we graph and compare the performance of our genetic algorithm in all different environments and variations of J function. We also vary the batch size from $1, 10$ to see the effect of batch size on the performance of the different algorithms. For each combination, we have plotted the value of the best genome performance. Note that the values of fitness plotted here are the vanilla J since that is the goal of Reinforcement Learning. The different modifications that we have introduced are merely to find better proxies to J that may be easier to maximize. All configurations were designed to run for 1000 generations but many of them were stopped earlier either because of reaching max reward or because of extreme stagnation in performance.

A. CartPole

We start with the easiest environment. In CartPole-v0, the agent has to control a rod that is placed on top of a cart. It can control the linear velocities of the cart in an attempt to keep the rod balanced. For every timestamp that the rod hasn't fallen the agent gets a reward of $+1$. The environment terminates after 200 time steps, so the maximum possible reward is 200.

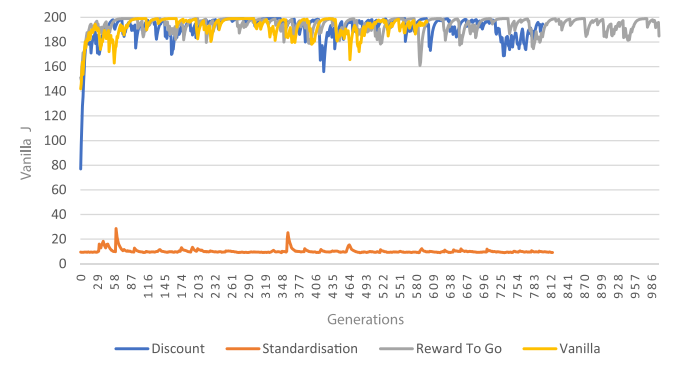

Fig. 4. CartPole-v1 with $N = 1$

In both batch sizes 1 and 10 [4,5] we notice that all of the methods except standardisation very quickly reach the maximum value of 200. The version with $N = 1$ has more noise because of the extremely stochastic nature of our estimation approximation. Expectantly, standardisation method performs the worst of all. Since all rewards are 1, the variance is zero. Now division by zero causes the proxy J function to give out garbage results. This means that standardisation method doesn't perform any better than random in this scenario.

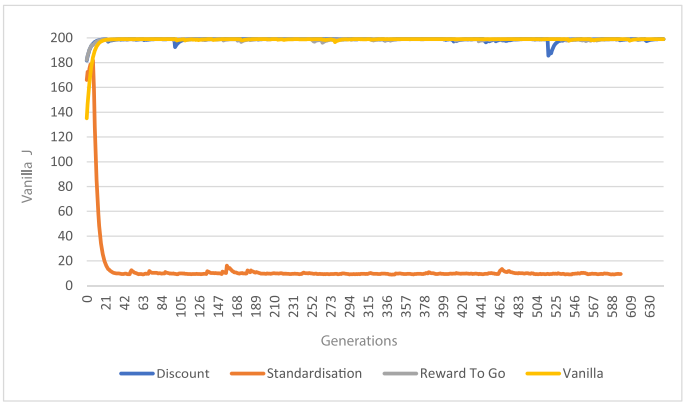

Fig. 5. CartPole-v1 with $N = 10$

B. LunarLander

LunarLander-v2 is a more complicated environment. We notice very strong results in this environment depending upon the batch size and the algorithm chosen. However, in all scenarios, the more advanced methods that we propose in this paper beat the vanilla results. In this environment, the agent is responsible to control a lander trying to land at a designated part. Depending upon velocity and successful land or crash different rewards is given. This environment is considered solved if reward achieved by agent crosses 200.

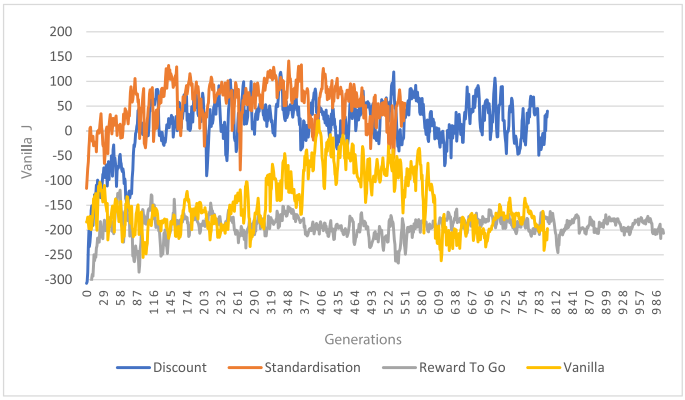

Fig. 6. LunarLander-v2 with $N = 1$

The results for $N = 1$ [6] show both standardisation and discounted methods performing far better than vanilla and reward-to-go. The former two methods were able to get their fitness above 0 very rapidly and then stagnated around 50 to 100 while vanilla and reward-to-go methods were stuck in the negative region.

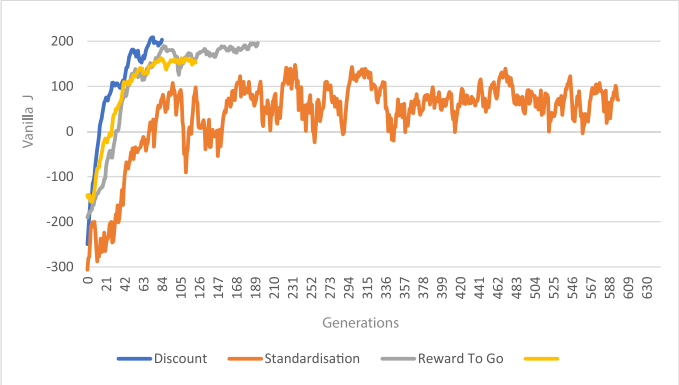

Fig. 7. LunarLander-v2 with $N = 10$

The results for $N = 10$ [7] have lesser spectacle. All three methods except standardisation quickly solved the environment and reached the max reward values. Out of these, discount algorithm was the fastest and the stablest. Surprisingly, a lead performer of the previous round: Standardisation did the worst with an increase in batch size. We suspect this is because the buffer allocated for calculating mean and variance was too small and couldn't accommodate the increased number of samples in each generation.

C. Pendulum

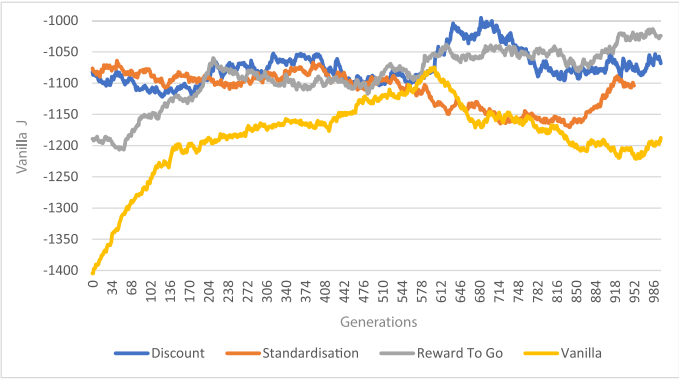

Fig. 8. Pendulum-v0 with $N = 1$

Pendulum-v0 is the hardest environment. The goal of this environment is to balance an inverted pendulum upright with the minimum amount of force possible. In this environment, batch size was a very important factor for good performance.

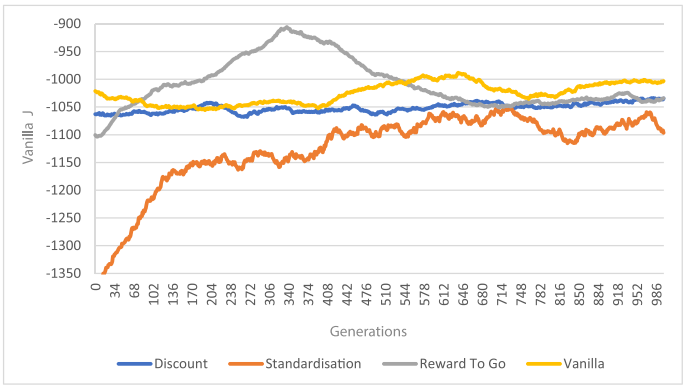

Fig. 9. Pendulum-v0 with $N = 10$

VI. Related and Future Work

At both batch sizes, the best performer is reward to go. In $N = 10$, reward-to-go manages to beat all other methods by over 150 points before losing performance because of stagnation. In $N = 1$ it has a close battle with discount method and overtakes it around the end. The performance of standardisation and vanilla methods is mixed and about the same. As with LunarLander, standardisation performs significantly worse at increased batch sizes probably because of a small buffer. We expect its performance to improve with increase in available memory to solve problems.

There has been a lot of work in the field of parameter optimization for reinforcement learning using Genetic Algorithms [Seh+19; PE02]. There also has been much more recent research work on directly using Genetic Algorithms for Reinforcement learning tasks [Suc+17; MHH07]. This paper belongs to the latter class.

A future extension of this work would consider more complex variations of NEAT networks that are still evolved using Genetic Algorithms such as HyperNeat[DGS14], HyperNeat-GCP[Hau+12] and ES-HyperNeat[SR11]. Another area of future research is looking into even more advanced methods such as actor-critic[KT00], deterministic policy gradient [Sil+14], baseline subtraction and more. The former two methods in particular form the corner-stone of many of modern RL's greatest achievements.

Conclusion

We have evaluated the questions of whether genetic algorithms can solve reinforcement learning problems and whether they can benefit from advanced in reinforcement learning algorithms. We answer both of these with an affirmative and see major improvements across all environments with the modified algorithms proposed in this paper. We also show that modern reinforcement learning problems can be tackled with genetic algorithms. This paper opens up the avenue of more research in importing complicated methods such as deterministic policy gradient [Sil+14] and actor-critic policy gradient methods [KT00] to genetic algorithms.

Acknowledgment

The authors want to thank Professor Nirupam Chakrobarty for giving them the opportunity to work on this paper and for inspiration to investigate genetic algorithms in a reinforcement learning setting. This paper was partly possible because of the free compute resource provided by Google Colab [Bis19] and AWS Educate. The authors also wish to thank the developer of Open AI gym, PyTorch[Pas+17], and neat-python[McI+] without which this paper would have been impossible. Finally, the authors express gratitude towards Prof. Sergey Levine and the University of California at Berkeley for the CS285[Lev18] - Deep Reinforcement Learning course, parts of which have been used as reference material for this paper.

References

[BSA83] Andrew G Barto, Richard S Sutton, and Charles W Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE transactions on systems, man, and cybernetics* 5 (1983), pp. 834–846.

[BDH96] Hagan Demuth Beale, Howard B Demuth, and MT Hagan. "Neural network design". In: *Pws, Boston* (1996).

[Yao99] Xin Yao. "Evolving artificial neural networks". In: *Proceedings of the IEEE* 87.9 (1999), pp. 1423–1447.

[KT00] Vijay R Konda and John N Tsitsiklis. "Actor-critic algorithms". In: *Advances in neural information processing systems*. 2000, pp. 1008–1014.

[Sut+00] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems*. 2000, pp. 1057–1063.

[PE02] James E Pettinger and Richard M Everson. "Controlling genetic algorithms with reinforcement learning". In: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. 2002, pp. 692–692.

[SM02] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[MT03] Peter Marbach and John N Tsitsiklis. "Approximate gradient methods in policy-space optimization of Markov reward processes". In: *Discrete Event Dynamic Systems* 13.1-2 (2003), pp. 111–148.

[SM04] Kenneth O Stanley and Risto Miikkulainen. "Competitive coevolution through evolutionary complexification". In: *Journal of artificial intelligence research* 21 (2004), pp. 63–100.

[MHH07] Shingo Mabu, Kotaro Hirasawa, and Jinglu Hu. "A graph-based evolutionary algorithm: Genetic network programming (GNP) and its extension using reinforcement learning". In: *Evolutionary computation* 15.3 (2007), pp. 369–398.

[MD08] Gianluigi Mongillo and Sophie Deneve. "Online learning with hidden Markov models". In: *Neural computation* 20.7 (2008), pp. 1706–1716.

[PS08] Jan Peters and Stefan Schaal. "Natural actor-critic". In: *Neurocomputing* 71.7-9 (2008), pp. 1180–1190.

[SR11] K Stanley and S Risi. "Enhancing ES-HyperNEAT to Evolve More Complex Regular Neural Networks". In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2011). New York, NY: ACM. http://eplex. cs. ucf. edu/papers/risi_gecco11. pdf*. 2011.

[Zha+11] Tingting Zhao et al. "Analysis and improvement of policy gradient estimation". In: *Advances in Neural Information Processing Systems*. 2011, pp. 262–270.

[Hau+12] Matthew Hausknecht et al. "HyperNEAT-GGP: A HyperNEAT-based Atari general game player". In: *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. 2012, pp. 217–224.

[DGS14] David B D'Ambrosio, Jason Gauci, and Kenneth O Stanley. "HyperNEAT: The first five years". In: *Growing adaptive machines*. Springer, 2014, pp. 159–185.

[Sil+14] David Silver et al. "Deterministic policy gradient algorithms". In: 2014.

[Bor16] Steven Borowiec. "AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol". In: *The Guardian* 15 (2016).

[Bro+16] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[ZK16] Sergey Zagoruyko and Nikos Komodakis. "Wide residual networks". In: *arXiv preprint arXiv:1605.07146* (2016).

[Gib17] Samuel Gibbs. "AlphaZero AI beats champion chess program after teaching itself in four hours". In: *The Guardian* 8 (2017).

[Pas+17] Adam Paszke et al. "Automatic differentiation in pytorch". In: (2017).

[Suc+17] Felipe Petroski Such et al. "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning". In: *arXiv preprint arXiv:1712.06567* (2017).

[Lev18] S Levine. "Lecture notes in deep reinforcement learning, cs 285". In: *University of California, Berkeley* (2018).

[Bis19] Ekaba Bisong. "Google Colaboratory". In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 2019, pp. 59–64.

[Seh+19] Adarsh Sehgal et al. "Deep reinforcement learning using genetic algorithm for parameter optimization". In: *2019 Third IEEE International Confer-*

ence on Robotic Computing (IRC). IEEE. 2019, pp. 596–601.

[McI+] Alan McIntyre et al. *neat-python*. https://github.com/CodeReclaimers/neat-python.