Journal of Next-Generation Research 5.0 (JNGR 5.0)
ISSN: **3075-2868** Website: www.jngr5.com Email: editor@jngr5.com

Reinforcement Learning Guided Engineering Design: from Topology Optimization to Advanced Modelling

Giorgi Tskhondia

Independent researcher, Tbilisi, Georgia

Abstract
Applying reinforcement learning for topology optimization is a novel approach to engineering design. In this work, I produced 6 by 6 and 5 by 5 grid topologies by the PPO algorithm and 4 4 by the HRL algorithm in adequate compute wall-clock time. I have also addressed increasing the calculation speed for artificial intelligence-driven topology optimization by combining genetic algorithms and reinforcement learning in a straightforward sequential (one method after another). First, I apply genetic algorithms to get an outline of the topology, and then I 'fine-tune' or refine the obtained topology by reinforcement learning approach. This way, I can obtain more optimal topologies and reduce wall-clock time. In particular, I optimized topology for a 10 by 10 grid, which can be seen as an improvement over a 6 by 6 topology obtained by reinforcement learning alone. The genetic algorithm alone could not produce such an optimal topology as a combination of reinforcement learning and genetic algorithms in comparable wall-clock times.

Keywords: Topology Optimization, Reinforcement Learning, Generalizable, Engineering Design, Genetic Algorithms

1. Introduction

1.1. Motivation

Rapid development of artificial intelligence facilitates engineering design by creating advanced tooling to assist an engineer. The pace of computer technological development dictates a modern designer to equip herself with at least basic understanding of current artificial intelligence algorithms and methods. From this perspective, this work is an attempt to provide some guidelines to artificial intelligence aided engineering design with focus on topology optimisation by reinforcement learning.

Topology optimisation (TO) got traction in recent years and is incorporated in almost all main commercial CAD systems nowadays. Optimisation approaches used by these systems are mostly based on genetic or gradient optimisation combined with finite element methods. But

little attention has been paid to using reinforcement learning (RL) for topology optimisation. Only few scientific groups have attempted to apply RL to topology design so far. Recent example of applying reinforcement learning to TO can be found in [1, 2]. Main advantages of RL include being gradient-free approach and its generalisability. Whereas the disadvantages are sample inefficiency of RL and scalability problem for huge finite element meshes. On the one hand, RL takes a lot of wall clock time to train, and, on the other hand, finite element analysis might take lots of time to converge. This drawback should be overcome with the advances in better hardware (like GPUs or targeted AI chips) and better, more sample efficient RL algorithms in the future. In fact, reinforcement learning itself can be used in computer chip design [3].

Likewise, domain conceptual design has a lot of value for creating unique machinery and cherishing the future design methods. Some hope that domain design can be solved with large language models (LLM) like GPT, [4], which relies on reinforcement learning from human feedback (RLHF), [5], or Deepseek which relies on chain of thought (CoT) and mixture of experts (MoE), [6]. These LLMs show ability to generate programming code that compiles and is logically correct. The same LLMs can be used to generate a 'language' or 'code' (akin to Unified Modelling Language, [7]) that describes the geometry of mechanical structures in parametric models, component diagrams, structural diagrams, etc. This 'code' would represent unique structures and designs. One example of this could be LLM producing the input file for finite element model, then one would run FE analysis with this input file via utilizing a FE engine (not LLM), and then the produced output were again fed to LLM for postprocessing.

Additionally, an interesting approach would be using LLMs on graphs to design mechanical structures. Evolution of a mechanical structure during generative design could be seen as time varying graph (finite element mesh is a graph in essence), hence applying graph neural networks (e.g. graph transformers, [8]) to the design process seems to be an interesting direction of research.

Finally, some diffusion probabilistic models [9] that create photorealistic images can be used for an engineer's inspiration. Here, for generating sketches of a design one needs to provide a prompt using natural language only. These sketches can later be refined with detailed drawings or fed to a multimodal LLM, [10], for further design tailoring.

1.2. Focus of this article

Learning how to optimize topology has major importance in structural design for aerospace, automotive, offshore, and other industries. Despite outstanding successes in topology optimization methods, [11], complexity of design space continues to increase as the industry faces new challenges. A viable alternative to conventional topology optimization methods might be deep reinforcement learning approach, [12]. Reinforcement learning offers gradient free, learning based, generalizable topology optimization paradigm suitable for non-convex design spaces. It can be seen as a smart search in solution space.

Deep reinforcement learning has had great success in artificial intelligence applications. Among them, beating the champion of the game of Go in 2016, mastering many Atari games, [13], and

optimizing the work of data centers. In this work, I have combined deep reinforcement learning (RL), genetic algorithms (GA), and finite element analysis (FEA) for the purpose of topology optimization. I have experimented with 10x10, 6x6, 5x5 and 4x4 FE grids, tested generalizability, applied simple reward function for the RL agent, applied density field-based topology optimization, and used simpler input features' vector compared to other works. I have experimented with PPO, [14], HRL [15], and GA algorithms. Also, my code implementation (see Availability of data and materials) is scalable and robust.
The RL agents were able to find optimal topologies (more optimal than by gradient methods) for a wide range of boundary conditions and applied loads [16 - 18].
To the best of my knowledge, I am the first to apply PPO, HRL and combination of GA and RL to the problem of topology optimization.

2. Reinforcement Learning Background

2.1. Markov Decision Process (MDP) framework

I consider the standard reinforcement learning setting where an agent interacts with an environment E over several discrete time steps. At each time step t, the agent receives a state s_t and selects an action a_t from some set of possible actions A according to its policy π, where π is a mapping from states s_t to actions a_t. In return, the agent receives the next state s_{t+1} and receives a scalar reward r_t. The process continues until the agent reaches a terminal state after which the process restarts. The return $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is the total accumulated return from time step t with discount factor $\gamma \in (0, 1]$. The goal of the agent is to maximize the expected return from each state s_t.
The action value $Q^{\pi}(s, a) = E[R_t | s_t = s, a]$ is the expected return for selecting action a in state s and following policy π. The optimal value function $Q^{*}(s, a) = \max_{\pi} Q^{\pi}(s, a)$ gives the maximum action value for state s and action a achievable by any policy. Similarly, the value of state s under policy π is defined as $V^{\pi}(s) = E[R_t | s_t = s]$ and is simply the expected return for following policy π from state s, [19].
In topology optimization settings, the finite element model represents an environment to which an agent applies actions and from which it gets observations and rewards. An agent uses neural network to decide on its actions. Actions change topology and the new topology is then subjected to finite element analysis (FEA). Finite element analysis produces the state, which then is fed to neural network. And the process repeats itself. The agent gets rewards if it meets an optimization objective of minimizing compliance. The end result of the modelling (after inference stage) is an optimized topology. The inference stage is a usual greedy inference where an agent makes actions of altering the topology based on observations only.

2.2. Formulating engineering design as full reinforcement learning problem

I argue that engineering design in general and topology optimization in particular should be formulated as full reinforcement learning but not as just a deterministic problem where the next state depends on previous state and action alone but not also on random noise.

Journal of Next-Generation Research 5.0 (JNGR 5.0)
ISSN: 3075-2868 Website: www.jngr5.com Email: editor@jngr5.com

For the purpose of my argument, I will use terms dynamic programming and reinforcement learning interchangeably as per [20]. In its simple form, dynamic programming can be described as solution to the equation of full reinforcement learning (Figure 1):

$$x_{n+1} = f(x_n, a_k, w_k), \qquad k = 0, 1, 2, \dots, N-1$$

k – *indexes discrete time,*
x_k – *state of the system,*
a_k – *action,*
w_k – *noise,*

w_k *can* depend on x_k *and* a_k, *i.e.* $w_k = g(x_k, a_k)$.

Equation of full reinforcement learning, (1)

Figure 1. Equation of Full Reinforcement Learning (1)

I will also use words engineering design and topology optimization interchangeably to some extent. However, my argument can be understood in a broader sense when applied to all forms of engineering design but not only to topology optimization. For instance, I will touch such processes as metal stamping and designing computer chips layouts.

Let's start from mechanical engineering and metal pressure treatment. In topology optimisation, especially 3D printed structures, you can have a large range of fatigue allowable stresses that depend on the thermal history experienced by the material in each point. Basically it means that your component will experience fatigue based on history and ordering of the structure's 3D printing. To some degree, fatigue initiates at random after certain amount of cycles of exploited component. Here we can say that actions of producing the structure in a sequence, whether during design or actual manufacturing, affect the outset of fatigue of the component in the future. In terms of equation (1), if *wk* is random outset of fatigue, it should depend on N-2(nd) state and N-2 (nd) action. Or more broadly on all previous states and actions in a sequence in a sense that N-2(nd) depend on N-3(d) and so on down to state number zero. Hence, equation of full reinforcement learning (1) can be applied to both manufacturing and design (as a way for future manufacturing steps) of the component.

Another example is metal stamping where metal stamping mold's shape would depend on the topology of the component being produced. The mold would install different micro defects into the component after metal pressure treatment (stamping). These defects will be installed at random. Think *wk* is micro defects. Again, defects would depend on the topology of the component, on the shape of the mold. These defects affect metal fracture in the future component exploitation, and might even have non-stationary evolution that leads to fatigue. Hence, equation (1) can be used and the problem should be formulated as full reinforcement learning.

Now, let's say a few words about electrical engineering and chip design. If you design chips you need to deal with something called parasitic capacitance (an unavoidable and usually unwanted capacitance that exists between the parts of an electronic component or circuit simply because of their proximity to each other). This makes transition dynamics (how the voltage and current

in different parts of the circuit change over time in response to different inputs) stochastic and to some degree random. Once more, we have here *wk* as parasitic capacitance. The layout of the chip you design would affect implicitly the parasitic capacitance you get in the future. Hence, again, the chip design can be seen as a full reinforcement learning problem here.

2.3. Proximal Policy Optimization (PPO)

PPO is an architecture that improves an agent's training stability by avoiding too large policy updates. To do that, it uses a ratio that indicates the difference between the current and old policy and clips this ratio from a specific range $[1-\epsilon, 1+\epsilon]$.

PPO seeks to improve over other solutions by introducing an algorithm that attains the data efficiency and reliable performance of Trust Region Policy Optimization [21], while using only first-order optimization. PPO authors propose a novel objective with clipped probability ratios, which forms a pessimistic estimate (i.e., lower bound) of the performance of the policy. To optimize policies, they alternate between sampling data from the policy and performing several epochs of optimization on the sampled data, [14].

2.4. Hierarchal Reinforcement Learning (HRL)

Hierarchical agents have the potential to solve sequential decision-making tasks with greater sample efficiency than their non-hierarchical counterparts because hierarchical agents can break down tasks into sets of subtasks that only require short sequences of decisions. To realize this potential of faster learning, hierarchical agents need to be able to learn their multiple levels of policies in parallel so these simpler subproblems can be solved simultaneously. Yet, learning multiple levels of policies in parallel is hard because it is inherently unstable: changes in a policy at one level of the hierarchy may cause changes in the transition and reward functions at higher levels in the hierarchy, making it difficult to jointly learn multiple levels of policies. In the original paper, the authors introduce a new Hierarchical Reinforcement Learning (HRL) framework, Hierarchical Actor-Critic (HAC) that can overcome the instability issues that arise when agents try to jointly learn multiple levels of policies. The main idea behind HAC is to train each level of the hierarchy independently of the lower levels by training each level as if the lower-level policies are already optimal. HAC authors demonstrate experimentally in both grid world and simulated robotics domains that their approach can significantly accelerate learning relative to other non-hierarchical and hierarchical methods. Indeed, their framework is the first to successfully learn 3-level hierarchies in parallel in tasks with continuous state and action spaces, [15].

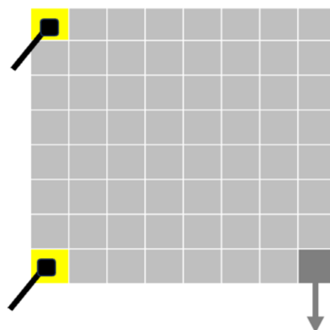
2.5. Genetic algorithms

According to Mitchell (1996) and Whitley (1994), in computer science and operations research, a genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimisation and search problems by relying on biologically inspired operators such as mutation, crossover and selection. In a genetic algorithm, a population of candidate solutions (called individuals, creatures, or phenotypes) to an optimisation problem is evolved toward better solutions. Each candidate solution has a set

of properties (its chromosomes or genotype) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible (with e.g., continuous variables). The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimisation problem being solved. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.
A good introductory book for GAs, extensively used in this article, is by [22]. It uses python as a primary programming language and DEAP framework for implementing genetic algorithms. DEAP is a novel evolutionary computation framework for rapid prototyping and testing of ideas. It seeks to make algorithms explicit and data structures transparent. It works in perfect harmony with parallelisation mechanism such as multiprocessing and SCOOP [1].

3. Experiments

Using codebase from [23], I have tried to rework its topology optimization approach by replacing its conventional gradient based optimization method with reinforcement learning and genetic algorithms.
Topology optimization aims to distribute material in a design space such that it supports some fixed points or "normals" and withstands a set of applied forces or loads with the best efficiency. To illustrate how we can formulate this, I concentrate on one of design problems from [1], describing elementally discretized design domain as per (Figure 2).



[1] https://deap.readthedocs.io/en/master/

Figure 2. 2D Representation of an Elementally Discretized Starting Topology of an 8 × 8 Cantilever Beam

The large gray rectangle here represents the design space. The load force, denoted by the downwards-pointing arrow, is being applied at the bottom right corner. There are fixed points here as well, which are at the top and bottom left corners of the design space corresponding to a normal force from some external rigid joint (Figure 2).

Finite elements. Elastic materials have continuous physics, but a computer can only deal with discrete approximations. This means that we have to divide the design space into a discrete number of regions or finite elements that can mimic the behaviour of an elastic solid as closely as possible. We can connect their boundaries with a set of nodes and let these nodes interact with each other as if they were joined by springs. This way, when a force is applied to one node, it passes on a part of that force to all the other nodes in the structure, causing each to move a little and, in doing so, deform the finite elements. As this happens, the whole structure deforms as if it were an elastic solid.

There are many ways to choose the arrangement of these finite elements. The simplest one is to make them square and organize them on a rectangular grid.

In the diagram above, there are 64 elements with four nodes per element and two degrees of freedom (DOFs) per node. The first is horizontal and the second is vertical. The numbering scheme proceeds column wise from left to right so that the horizontal and vertical displacements of node n are given by DOFs $2n - 1$ and $2n$ respectively. This grid structure is useful because it can be exploited "…in order to reduce the computational effort in the optimization loop…" It also simplifies the code [23].

The density method. After parameterizing the design space, we need to parameterize the filling material. At a high level, each finite element will have a certain amount of material given by some number 0 or 1. We will use this amount to determine the element stiffness coefficient Ee, also known as Young's modulus. In the nodes-connected-by-springs analogy, this coefficient would control all the spring stiffnesses.

Reinforcement learning. I have applied PPO implementation from stable baselines 3 library and reworked HRL implementation from [24]. For both PPO and HRL, I have created custom OpenAI Gym environments.

The action space consisted of N^2 actions of 'filling void space (actually with the density of 1e-4) with an element (density of 1)', where N by N is dimension of the grid.

I have experimented with 6 by 6 and 5 by 5 grids for PPO, and 4 by 4 grid for HRL. For PPO setting, I have tested generalizability of the model by randomizing the places of force application across different episodes and being different in training and inference. The environment for this setting was the topology on the grid.

I experimented with squared rooted and squared reciprocals of compliances as reward functions. I trained my model for 1M steps (for about 80min of wall-clock time) on 2,9 GHz Dual-Core Intel Core i5 computer, for PPO. And for 5000 steps on Apple M1 Pro, for HRL.

Journal of Next-Generation Research 5.0 (JNGR 5.0)
ISSN: 3075-2868 Website: www.jngr5.com Email: editor@jngr5.com

For HRL setting, hyperparameters (including network architecture) were obtained by *optuna*. For PPO settings, the exact architecture is as per *stable baselines 3* implementation. Interested reader is referred to (Availability of data and materials) section for more details on the neural network's architecture.

The results of applying PPO and HRL to topology optimization of Cantilever beam is presented in Figure 3 below.

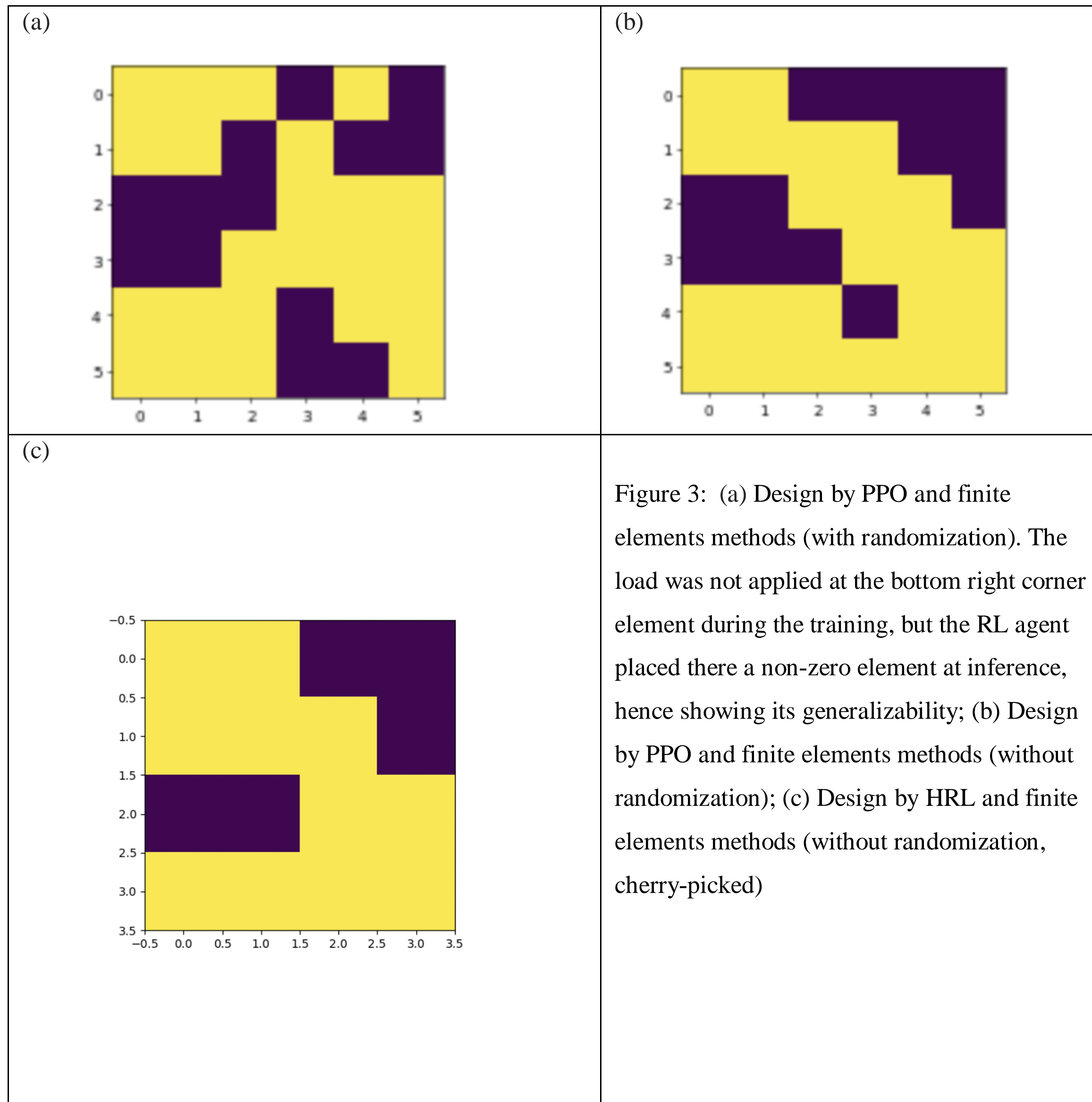


Figure 3: (a) Design by PPO and finite elements methods (with randomization). The load was not applied at the bottom right corner element during the training, but the RL agent placed there a non-zero element at inference, hence showing its generalizability; (b) Design by PPO and finite elements methods (without randomization); (c) Design by HRL and finite elements methods (without randomization, cherry-picked)

I have experimented with a 2-level hierarchical agent in 4 by 4 grid and did not test generalizability for HRL setting. My intention with HRL was to speed up the learning.

RL was able to reliably find near optimal topologies for a range of different boundary conditions and applied forces. In terms of performance my implementation of HRL is much slower than PPO if we account for HPO (hyper parameter optimization), but seems to be faster if we do not

account for HPO. Another limitation of HRL is that you need to manually set the target state, which is a kind of drawback because you do not know the target state in advance and can only set it by experimentation.

Number of calculation steps is much less than the number of possible combinations (states) on the grid, hence the neural network is not merely learning by heart the most optimal solution (Figure 4).

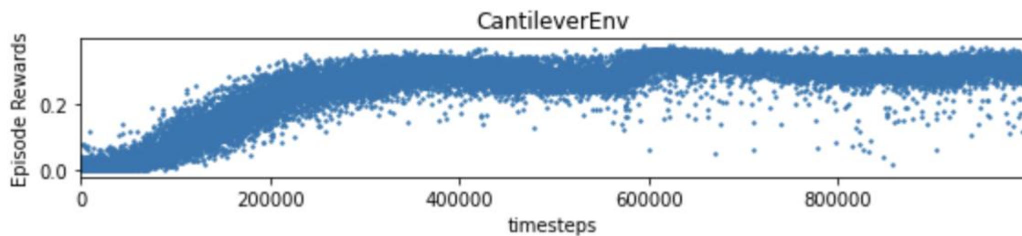


Figure 4: Episode Reward for PPO Setting

In other set of experiments, I have combined genetic algorithms and reinforcement learning in a straightforward sequential (one method after another) way. First, I apply genetic algorithms to get an outline of the topology and then I 'fine-tune' or refine obtained topology by reinforcement learning approach. In this way, I can obtain both more optimal topologies and reduce wall-clock time. I was able to optimize topology for 10 by 10 grid, which can be seen as an improvement over 6 by 6 topology obtained by reinforcement learning alone. It should be mentioned that genetic algorithm alone was not able to produce such an optimal topology as by combination of reinforcement learning and genetic algorithms in adequate wall clock times.

I combined genetic algorithm (GA) and reinforcement learning to optimize a 10 by 10 grid in the following manner:

1. I have run fast GA (for about 1 min wall-clock time), where actions were placing an element in the void on the grid (void is 0 and an element is 1).
2. I have set an action space for the RL problem as removal of boundary elements obtained in (1) only (Figure 5). All other actions (grid elements) were not considered in RL problem. Hence, I considerably reduced combinatorial space for the RL task.
3. I ran RL PPO algorithm for about 32 min wall clock time.
4. Experiments showed that GA pre-training + RL refinement was better than GA alone (in terms of a compliance metric) for the same wall clock run times. And, obviously, GA+RL was better than RL alone.

I trained my model on Apple M1 Pro. I have used DEAP framework for the GA part. And, I have used stable_baselines3 and OpenAI gym for PPO algorithm's implementation for the RL part.

Journal of Next-Generation Research 5.0 (JNGR 5.0)
ISSN: 3075-2868 Website: www.jngr5.com Email: editor@jngr5.com

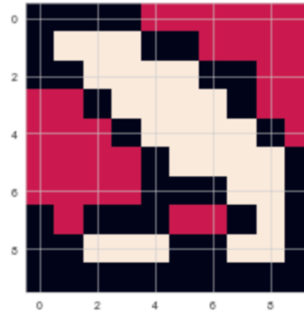


Figure 5. Topology with Boundary Layer (Action Layer) in Black, Internal Material in Yellow and Empty Space in Red

I applied downward load at the bottom right corner and fixed an element on the left side alike cantilever beam (Figure 2). Objective function was a compliance in topology optimisation sense i.e. an inverse strength of a material. RL improved compliance metric of GA from 19.3654 to 18.2786 (the lower the better). Near optimal topologies are depicted in Figure 6 below.

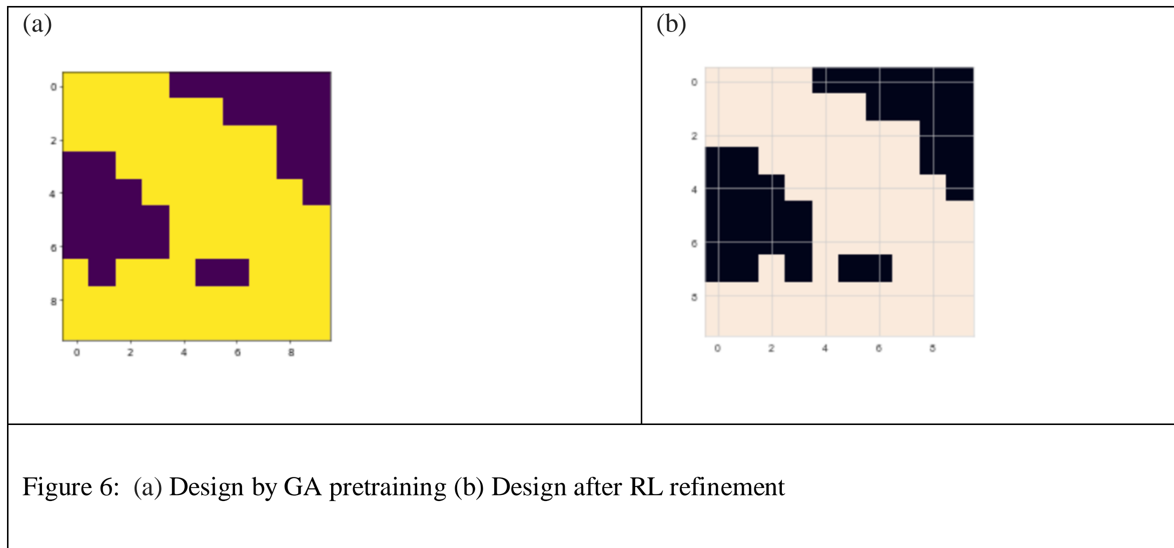


Figure 6: (a) Design by GA pretraining (b) Design after RL refinement

Learning progress for RL refinement can be seen in Figure 7.

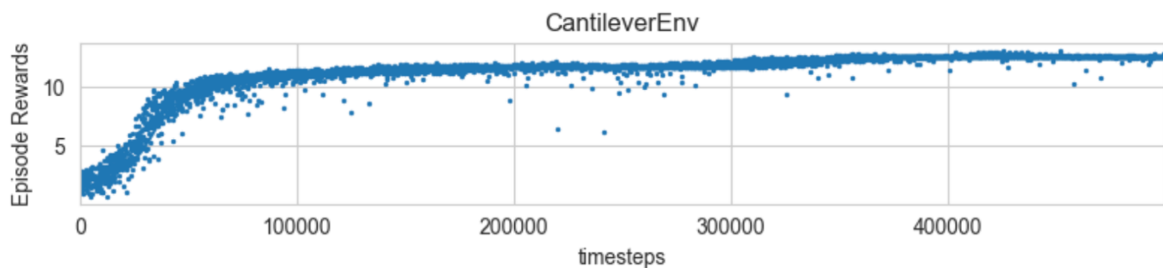


Figure 7: Episode Reward for PPO Refinement

4. Discussion

One of the main problems when applying reinforcement learning to topology optimisation is scaling up from smaller geometries to larger design spaces. In this work, I have tried to apply HRL, PPO and GA to solve the scale up problem. My intention was to speed up the learning hence potentially to be able to solve larger topologies, especially with non-convex objectives.

As stated earlier, HRL is a technique that utilizes hierarchies for a better sample efficiency of RL algorithms. Hierarchies can be spatial, temporal or between the abstraction levels. In the case of finite element analysis, hierarchy can be different scales (micro, macro) or it can be different comprehensive levels of mathematical model (e.g. analytical solution — plane stress — 3D model).

At first, my reasoning for HRL was that an agent at higher level would choose one of the sub sections within the design space grid first, and a low-level agent then would choose one of the cells within the sub-section chosen at higher level. But that would make decisions sequential and interdependent between the hierarchical levels. From the literature analysis, it turned out that most promising HRL algorithms are those who make decision within a hierarchical level relative to other level not sequentially but in 'parallel', since it greatly speeds up the learning. Making my initial 'sequential' views on HRL applications to structural design a bit off. Nevertheless, the initial idea of different hierarchical levels doing different things, for example, on a higher level an agent moving to the location on the grid, and on the lower level, the agent changing the topology around that location, should be investigated further, because it can allow reducing number of computations required and speeding up the learning as an RL agent would need to learn doing 'local actions' only. Some experimentation related to this can be found in the codebase for this article (see Availability of Data and Materials).

There were a few tricks that I used to solve the HRL environment:

- hyperparameter optimisation (HPO; done with *optuna*)
- tweaking neural network architecture (especially adding dropout layer)
- reward engineering
- making neural network aware of the design space geometry (Figure 8)

```
self.penalty_coeff = 0.10 # 0.25
action=action*(1-self.penalty_coeff*self.x.reshape(len(action),))
```

Figure 8. Code Snippet for HRL 'Trick'

Penalty coefficient is used to give the agent an ability to do the same actions (Figure 8).

The continuous environment in this HRL model is a vector of [mean density, compliance], and the action is putting a material into the cell of the grid. The number of actions equals the number of cells in the grid. Time horizon to achieve a subgoal should be less than maximum number of steps in the episode to benefit from abstraction that HRL provides. It should be noted that the environment vector is not uniquely maps to a topology.

I am not feeding the topology to the neural network explicitly during the learning, hence I applied the above trick of making neural network 'aware' of the design space geometry (Figure 8). It is kind of applying penalty to already taken actions. This restriction on the action distribution might actually not be the optimal way to do it, but, even if it is a faulty reasoning, a neural network should adjust its decisions and make right actions, resulting in an optimal policy that accounts for the 'bug'.
Also, it should be noted that relying only on hyperparameters tuning might have overfitted the domain.
Apart from applying the penalty, at first, I wanted to make the neural network to build an understanding of an environment by remembering actions taken (because I did not explicitly input the grid into the neural network, and the neural network did not know any changes happening to the topology). I experimented with adding LSTM in the first layer of actor network as a form of memory, but it reduced the speed of learning by a large margin, and potentially required accumulating some number of actions hence mixing up the initial HRL algorithm. So, I abandoned this idea and used the above trick of 'masking' (applying the penalty) action distribution instead. But I might return to the LSTM idea later since some papers indicate that entering even one frame of the grid might be sufficient for 'remembering' [25].
Both PPO and HRL produced near optimal topologies. Some benchmarks can be found in the Appendix.
By number of states, I mean number of combinations of cells in a grid.
The promise that RL can speed up topology optimization is high. If fully solved, it can allow to go from 2D to 3D topologies with RL especially for larger design spaces and non-convex objectives. Other ways to scale/speed up to larger topologies might be progressive refinement method [1], where one gradually increases the complexity of design space without requiring additional computation, and asynchronous deep reinforcement learning [19] that uses asynchronous gradient descent for optimization of deep neural network controllers and enables half the training time on a single multi-core CPU instead of a GPU. In the future, speeding up the calculation might be accomplished: by making neural networks 'smaller' via using sparsity, quantization, or distillation techniques; by applying physics informed neural networks [2] in place of finite element methods; by using evolutionary strategies [26] that can run in parallel; or, on the hardware side, by using analog chips [3].
I believe, the idea of hierarchical reinforcement learning can also lead to the systems of the full design cycle, where at the lowest level, and the RL agent designs material. One abstraction level above, the RL agent designs a component (assembly of materials), one level above that, the RL agent produces a product (assembly of components). All it is done concurrently. At each level, the RL agent makes higher and higher level design decisions. At the end, an engineer gets her final design that was generated in a hierarchical fashion. As for the effectiveness of current

[2] https://arxiv.org/abs/1711.10561
[3] https://www.youtube.com/watch?v=GVsUOuSjvcg

implementation of hierarchical RL, it was estimated that for each iteration there were ~11 actions that the agent was making. For 4500 iterations (min number of iteration at which we can get optimal solution), it is 11*4500 = 49500 actions. On the other hand, for 4 by 4 grid we have 2^16=65536 possible states (combination of cells). Hence, accounting for the fact that from that 11 actions that the agent was making some were repetitive, HRL is much more effective than a brute force approach (not in terms of wall clock time for 4x4 grid, but in terms of number of operations).

Although traditional TO will almost always be faster than RL in wall clock time meaning, reinforcement learning can account for a unique constraint where minimising gradients is tough or impossible to do (for non-convex objectives). Reinforcement learning accounts for these objectives and constraints through the reward function. Besides, generalisability of RL and transfer learning can make RL to be more compute efficient than classical TO in total once you train the RL model. Because, one does not need to rerun calculations for each new constraint, but just run RL inference instead. Additionally, every time you run inference after RL training, it gives you a slightly different topology. Hence, RL can be leveraged at inference time as it produces not a single design but several different designs from which you can choose the best. It happens since at each inference time a neural network fires slightly different weights thus producing a slightly different design.

Other experiments show that GA pre-training + RL refinement is better than GA alone (in terms of a compliance metric) for the same wall clock run times. And, GA+RL is better than RL alone. Modelling of this kind gives hope to apply my algorithms for a larger design spaces producing optimal topologies within an adequate wall-clock time.

Now, let's try to estimate practicality of the proposed approach if a supercomputer is available. First, let's try to estimate a supercomputer's compute capacity. In the article [4], on the macro placements of Ariane picture, there are ~ (32 by 34) grid with ~6 different colouring. That means that there are 6^1088 states that can be handled by Google supercomputer in an adequate wall clock time. On the other hand, we have a binary grid in topology optimization. Hence to understand how many nodes we can process, we need to solve the following equation: 6^1088=2^x, from which it follows that x~2812 elements. However, it should be noted that the google paper [3] used only reinforcement learning and did not apply finite element methods in their algorithm. I instead used combination of RL and genetic algorithm (that can handle more elements in less time) and applied finite element methods. Let's assume that these two factors compensate each other (in terms of calculation time) and we are still at 2812 elements that can be handled by a supercomputer. From the literature it was reported that gradient topology optimization can handle up to 30000 elements max, but nevertheless gradient topology optimization would not provide such optimal results as a combination of RL and GA presented in this paper.

[4] https://research.google/blog/chip-design-with-deep-reinforcement-learning/

Potential practical application of topology optimization with RL for small grids can be microscopic topology optimization, [27], and metamaterials design, [28].
Finally, an open research question in topology optimisation is how to make the produced designs suitable for a wide range of manufacturing methods but not only for additive manufacturing and 3D printing. I propose to approach this issue by guiding topology optimisation with reinforcement learning by techniques from AI safety research. For example, by learning a reward function from human feedback (favouring the designs that are manufacturable) and then to optimize that reward function, [5].

References

[1] Deep reinforcement learning for engineering design through topology optimization of elementally discretized design domains, Nathan K. Brown et al, https://doi.org/10.1016/j.matdes.2022.110672
[2] Reinforcement learning for optimal topology design of 3D trusses, Kazuki HAYASHI and Makoto OHSAKI, Proceedings of the IASS Annual Symposium 2020/21 and the 7th International Conference on Spatial Structures
[3] A graph placement methodology for fast chip design, Azalia Mirhoseini et al, Nature volume 594, pages 207–212 (2021)
[4] Improving Language Understanding by Generative Pre-Training, Alec Radford et al, OpenAI
[5] Deep reinforcement learning from human preferences, Paul Christiano et al, arXiv:1706.03741
[6] DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, DeepSeek-AI, arXiv:2501.12948
[7] Unified Modeling Language 2.5.1. OMG Document Number formal/2017-12-05
[8] Graph Transformers: A Survey, Ahsan Shehzad et al, arXiv:2407.09777
[9] High-Resolution Image Synthesis with Latent Diffusion Models, Robin Rombach et al, arXiv:2112.10752
[10] A Survey on Multimodal Large Language Models, Shukang Yin et al, arXiv:2306.13549
[11] Andreassen, E., Clausen, A., Schevenels, M., Lazarov, B. S., and Sigmund, O. Efficient topology optimization in matlab using 88 lines of code. Structural and Multidisciplinary Optimization,43(1):1–16, 2011.
[12] Richard Sutton and Andrew Barto. Reinforcement Learning: An Introduction. Second Edition, MIT Press, Cambridge, MA, 2018
[13] Playing Atari with Deep Reinforcement Learning, Mnih et al, 2013
[14] Proximal Policy Optimization Algorithms, Schulman et al, 2017.
[15] Learning Multi-Level Hierarchies with Hindsight, Andrew Levy et al, 2019
[16] On the Quest to Achieve Fast Generalizable Topology Optimization with Reinforcement Learning, Giorgi Tskhondia, 2023 (preprint)

[17] Topology optimization by genetic algorithms pre-training andreinforcement learning refinement, Giorgi Tskhondia, 2025 (preprint)
[18] Formulating engineering design as full reinforcement learning problem, Giorgi Tskhondia, 2025 (preprint)
[19] Asynchronous Methods for Deep Reinforcement Learning, Volodymyr Mnih et al, 2016
[20] Dynamic programming and optimal control by D.P. Bertsekas
[21] Trust Region Policy Optimization, John Schulman et al. 2017
[22] Hands-On Genetic Algorithms with Python: Applying genetic algorithms to solve real-world deep learning and artificial intelligence problems. Wirsansky Eyal, 2020
[23] A Tutorial on Structural Optimization, Sam Greydanus, May 2022, DOI: 10.48550/arXiv.2205.08966
[24] @misc{pytorch_hac, author = {Barhate, Nikhil},
title = {PyTorch Implementation of Hierarchical Actor-Critic},
year = {2021}, publisher = {GitHub}, journal = {GitHub repository}, howpublished = {\url{https://github.com/nikhilbarhate99/Hierarchical-Actor-Critic-HAC-PyTorch}},}
[25] Deep Recurrent Q-Learning for Partially Observable MDPs, Matthew Hausknecht and Peter Stone, 2017
[26] Evolution Strategies as a Scalable Alternative to Reinforcement Learning, Tim Salimans et al., 2017
[27] Microscopic stress-constrained two-scale topology optimisation for additive manufacturing, Xiaopeng Zhang et al, January 2025Virtual and Physical Prototyping 20(1)
[28] Deep reinforcement learning for the design of mechanical metamaterials with tunable deformation and hysteretic characteristics, Nathan K. Brown et al, Materials & Design Volume 235, November 2023, 112428

Declarations

Availability of data and materials

Code from this paper is available at https://github.com/gigatskhondia/gigala
Some of the presented ideas first appeared on my scientific blog at https://gigatskhondia.medium.com/

Competing interests

The author declares no financial or non-financial interests that are directly or indirectly related to this work.

Funding

Not applicable

Authors' contributions

Giorgi Tskhondia is a solo contributor to this paper, including ideas, modelling and presenting the results.

Acknowledgments

I would like to acknowledge all artificial intelligence researchers and structural engineers on top of whose ideas I have built my work. Also, I would like to acknowledge my family who provided me with immense support throughout all these years.

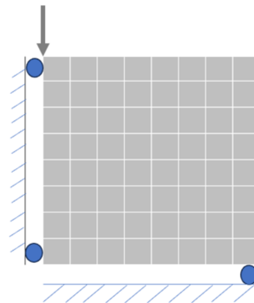
Authors' information

Giorgi Tskhondia (aka Georgy Tskhondiya): Independent researcher, PhD in mathematical modeling, ex subsea pipelines installation engineer, data science professional.

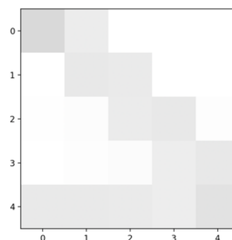
Appendix – Benchmarking for 5 by 5 grid

Measured by compliance as per codebase [23]; and by codebase [11].

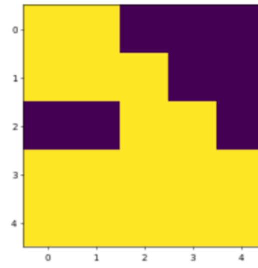
Loading schedule:



a). Topology obtained by Python version (topopt_cholmod.py; python3 topopt_cholmod.py 5 5 0.05 1 2 1) of [11]: compliance (as per [23]) is 89.63; wall clock time ~ 1 sec.



b). Topology obtained by the reinforcement learning (PPO): compliance (as per [23]) is **33.6**; wall clock time ~ **18 min**



c). Brute force approach: compliance (as per [23]) is **29.6**; wall clock time ~ **5.8 hr.**