TUM Data Innovation Lab

Munich Data Science Institute (MDSI)

Technical University of Munich

&

TUM Professorship of Data Science in Physics

&

Chair of Aerodynamics and Fluid Dynamics

Final report of project:

ADOPT: Topology Optimization using Reinforcement Learning in JAX-FEM

Authors	Eray Yildiz, Ioan Craciun and Kartik Bali
Mentors	Prof. Dr. Lukas Heinrich and Msc. Ludger Paehler
Project Lead	Dr. Ricardo Acevedo Cabra (MDSI)
Supervisor	Prof. Dr. Massimo Fornasier (MDSI)

Jul 2023

Abstract

With the advancements in machine learning algorithms and increased computational efficiencies, engineers now have access to new capabilities and tools that can be applied to engineering design. One category of such tools are machine learning (ML) models which can approximate complex functions and make them useful for various tasks in the engineering design workflow. This paper explores the use of reinforcement learning (RL), a subset of machine learning, to automate the designing of 2D discretized topologies. RL agents are trained to complete a task by accumulating experiences in an interactive environment. In this proposed environment, the RL agent can make sequential decisions to design a topology by removing elements to satisfy compliance minimization objectives best. After each decision, the agent receives feedback by evaluating how well the current topology satisfies the design objectives. This report explains a proof of concept study performed based on "Deep reinforcement learning for engineering design through topology optimization of elementally discretized design domains", which aims to train an RL agent that performs described 2D discrete topology optimization scenarios similarly or better than traditional gradient-based topology optimization methods. For comparison, the method of moving asymptotes (MMA) is used as a gradient-based optimizer in this study.

Contents

1 Introduction

1.1 Problem definition and goals of the project

Machine learning algorithms have made significant advancements in recent years and have become widely used in various industries and applications, including engineering design processes such as topology optimization (TO). Topology optimization is a computational design methodology in engineering that aims to determine the optimal distribution of material within a given design domain, subject to specific performance and manufacturing constraints. It employs mathematical and computational algorithms to systematically explore and evaluate different material layouts or configurations by iteratively redistributing material or void regions, to minimize or maximize desired performance metrics, such as load-bearing capacity, stiffness, or weight limitations. A topology optimization process typically consists of two main steps. The first step involves discretizing the design space into smaller elements and utilizing mathematical models, such as the finite element method (FEM), to evaluate the structural behavior and performance of the design. The second step includes an optimization algorithm that tries to improve the performance of the design based on the FEM results of obtained topology after each design update regarding the defined objectives and constraints. The output of a topology optimization process is often a design that exhibits an optimal material layout, resulting in enhanced structural efficiency, improved performance, or reduced material usage.
Gradient-based and evolutionary optimization algorithms are popular choices for topology optimization problems. However, these optimizers have limitations in certain cases. For example, gradient-based optimization tools require a relationship between the design parameters and the objective function(s) via gradient calculation to find the path toward the optimal solution, but such gradients may not exist for complex systems. Additionally, gradient-based algorithms are prone to get stuck in local minima or maxima for non-convex problems which may be far away from the global optima. Evolutionary algorithms (e.g., particle swarm, genetic algorithms, etc.) are preferred to overcome these challenges. These methods can slowly push the candidate designs toward global optima by sampling globally. Nevertheless, these techniques have a higher sample-complexity, and hence require more samples of the model and exhibit substandard performance when the quantity of design variables is substantial. At this point ML based optimization methods offer a potential solution to the mentioned traditional design optimization methods by approximating the complex mapping between the input design and the best design modification to fulfill objective and constraint requirements [30].
This report involves a proof of concept study focusing on design optimization tasks for 2D discretized topologies using Reinforcement Learning (RL) based on the paper titled "Deep reinforcement learning for engineering design through topology optimization of elementally discretized design domains" [4]. As an additional contribution, we make a performance comparison between the applied RL algorithm and MMA algorithm on a specific TO scenario.

1.2 State of the art approaches and algorithms in TO

ML-based methods have gained popularity recently in TO applications in addition to classical gradient-based methods and genetic algorithms. Most ML methods used in solving TO problems are structured on deep learning (DL) frameworks. DL-based methods are commonly used because they provide modularity in design and are highly adaptable for various tasks.
Figure 1 provides an overview of the state-of-the-art learning-based methods with their training strategies and how they are applicable to TO applications. Supervised and unsupervised learning are the most commonly considered strategies for fixed datasets, while reinforcement learning is an experience-based approach, and is part of a different class of algorithms. Transfer learning is another popular technique which can be used in any of the aforementioned approaches.

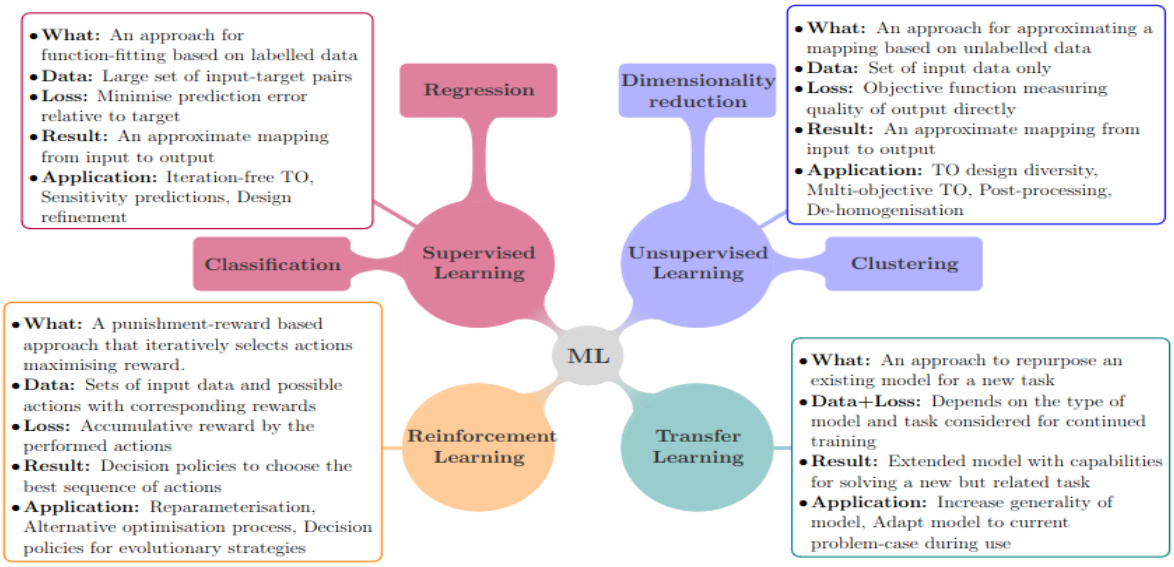


Figure 1: An overview of the common ML-based learning strategies used in TO [26].

Supervised Learning is commonly applied in cases where iteration-free topology optimization or efficient sensitivity approximations are desired, such as optimizing structures based on boundary conditions and loads or evaluating structural designs using element density values and computing displacement or strain energy density.

Unsupervised Learning is valuable for detecting underlying patterns without predefined outputs, as the loss function drives the learning process. It is particularly advantageous when multiple useful outputs exist for each input. However, this method requires an appropriate function for measuring the quality of the output in order to select the best output among all.

Reinforcement Learning (RL) is a branch of machine learning that enables an agent to learn and perform tasks in an interactive environment through feedback and rewards.

It focuses on training the agent to find an optimal design that satisfies the constraints in the design space.

Transfer Learning uses pre-trained models to improve performance on specific problems under different conditions or constraints. The success of this method depends on the similarity of the initial task to the new task.

In the literature there are varied examples of machine learning approaches being used to solve topology optimization scenarios, which differ both in the architecture of the learning agent as well as in the exact problem they are trying to solve. Sosnovik et al. [21] solved the topology optimization problem by using CNNs that are able to compare images of incomplete topologies with target optimal topologies. Another CNN approach include the one by Kollmann et al. [12], where by using deep neural networks the authors were able to increase or decrease various physical constraints, by creating different unit cells designs. The implementations do not always involve solving the whole problem. For example Ulu et al. [23] use a trained model to successively predict lower dimensional topologies, which are afterwards solved by a gradient-based method. Other approaches include data generation with the help of moving morphable component method, that could be later used for learning the designs (see Lei et al. [13]). This method includes solving a constraint problem.

As we can see these problems resemble the TO problem partially or even totally. But, none of them reach their goal with the help of reinforcement learning. However, reinforcement learning has already been applied in different varied domains: from predicting protein folding structures [16] or using deep R.L. for *beating* different computer games [20] until implementations in the field of robotics [11] or medicine [6].

2 Structure of the paper

In Background we present the basic fundamentals of the concepts we are going to deal throughout the project: e.g. topology optimization. Next, in Methods we will show popular ways in which one can deal with the problems from the previous chapter: e.g. FEA-based MMA for topology optimization. Implementation brings the rationale behind the code that implements the methods. Evaluation and Results present the modality in which we assess the quality of our product and the results, respectively.

3 Background

3.1 Reinforcement Learning

We are going to begin by presenting the theoretical fundamentals behind reinforcement learning, starting with a basic intuitive description of it. For this, we are going to make use of a simplified toy example where the lower part of a pole is attached to a hinge on cart, which can freely move from left to right. The scope of this scenario is to keep the pole in a vertical stance. Following this we will continue with proper definitions that define the goals of this type of machine learning.

3.1.1 Intuitive explanation

Reinforcement learning is a machine learning paradigm in which an agent learns to interact with a predefined environment according to the previous agent's interactions. Usually one can distinguish the following 5 components of a reinforcement learning problem:

- Environment: the *world* in which the learning takes place. In our example this would be the cart and the pole. These objects contain attributes such as the position of the cart and the angle at which the pole is currently at.
- Agent: the entity which interacts with its environment with the goal of maximising its reward function. The agent of the toy example is the cart which can move from side to side in order to keep the pole steady.
- Actions: every possible action that an agent can take in order to influence its environment. In our simplified example, this are the left and right commands given to the cart, which in turn will influence the position of the pole as well.
- States: the agent action renders the environment in different conditions. For example, by choosing to move the cart to the left, we will influence the position of the cart, but also rotate the pole to the right due to inertia.
- Goal and Reward function: The problem is defined by a goal, which in turn can be quantified by a reward function. In our toy example the goal is to be able to keep the pole steady in a vertical stance. The reward could be a function that return positive values the closer we are getting the pole to a stable state and negative values when the pole seems to get unstable. The main goal of the agent is to learn (after numerous repeated experiences) a productive policy. The policy is a mapping between the current state and an action i.e. given the current state which is the action that will reward the agent the highest amount in the long term.

3.1.2 Q-Learning

Having covered the intuitive meaning behind reinforcement learning in the last part, we are now going to explain the way in which the learning is done in a thorough manner. In the following we are going to reproduce explanations from [18] by Minh et al. In this paper, the authors make use of an environment where they can simulate simple old Atari games, where they realize an agent that is able to learn to use the controls in such a way that after the training is over, it is able to apply a sequence of control that result in high scores.
In the following we are going to denote the environment of the agent as $\mathcal{E}$. The action space is formalized as

$$\mathcal{A} = 1, ..., K \tag{1}$$

The agent does not observe complex data such as for example the Atari emulator internals, but uses a set of observations instead. In the case of [18] these are images represented as vectors $x_t \in \mathbb{R}^d$, where t is the timestep of the emulator and x_t represents the changes according to the actions taken by the agent. To store state representation, and store

successive actions, and observation in chronological order, the authors create the following state representation

$$s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t \tag{2}$$

Performing an action a_t when we are observing x_t will not only determine the new state x_{t+1}, but also a reward r_t whose goodness will be decided by a separate reward function. When an agent is performing an action it does not need to think only about the current reward, but also about the potential future rewards. In this way one can make sure that the agent will not be deceived by instantaneous large rewards, only to reach a dead end afterwards. For this, a cumulative *discounted reward at time t* is defined as

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}, \tag{3}$$

where γ represents the discount factor (which in practice is usually set to 0.9) and T is the final timestep. Next, an optimal-value action function which returns the maximal return possible after observing the state s and performing an action a is defined:

$$Q^*(s, a) = max_\pi \mathbb{E}[R_t | s_t = s, a_t = a, \pi, \tag{4}$$

where π is the policy. The optimal action-value function follows an identity called the *Bellman equation*, which allows us to rewrite the preceding value function as:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}[r + \gamma max_{a'} Q^*(s', a') | s, a] \tag{5}$$

The intuitive meaning behind this formulation is that after following a_t we will achieve the maximal reward only by following the action a_{t+1} which maximizes the reward at state s_t. Finally, the Bellman identity can then be rewritten as an iterative update:

$$Q_{i+1}(s, a) = \mathbb{E}[r + \gamma max_{a'} Q^*(s', a') | s, a] \tag{6}$$

. It can be proven that $Q_i \to Q^*$ as $i \to \infty$. However, such limit approach is infeasible in practice. That is why, the usual workaround is to use a function approximator $Q(s, a; \theta) \approx Q^*(s, a)$. This function approximator tends to be constructed with the help of a neural network with weights θ, which is usually referred to as a *Q-Network*. The common approach for finding a suited network is to define and minimize a loss function by using stochastic gradient descent. The loss of the neural network when using the weights θ_i at the i-th iteration is defined in the following way:

$$L_i(\theta_i) = \mathbb{E}_s, a \sim \rho(\cdot)[(y_i - Q(s, a; \theta)_i)^2] \tag{7}$$

where y_i is the target for the i-th iteration and is defined as:

$$y_i = \mathbb{E}_{s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a] \tag{8}$$

and $\rho(\cdot)$ is a distribution over sequences and actions. We can see, that in order to get the target we are using a previous iteration of the DQN in order to be able to use the future rewards. Finally, one obtains the following gradient:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i)] \tag{9}$$

Said approach has many degrees of freedom left such as:

- How do we perform the sampling for the mini-batches?
- How do we choose the next action in the best possible way?

We will develop these aspects in the latter part of this report.

3.2 Finite Element Method

FEM is a numerical method for finding approximate solutions to boundary value problems for PDEs. The central concept of FEM is the discretization of the large continuous problem with infinite degrees of freedom (DOF) into a problem with a finite number of DOF using idealized mathematical elements such as triangles, or hexagons to cover the to-be-simulated domain. The simple equations that model cells are then assembled into a larger system of equations that models the entire problem. The finite element method then uses techniques based on the calculus of variations to approximate a solution by minimizing an associated error function. As a last step of the FEM, displacements values at each nodes of associated cells are computed and other physical properties of the system are calculated based on these displacement values. A simplified workflow of FEM is given in Figure 2 below.

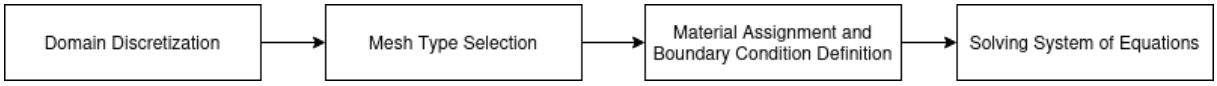


Figure 2: Simplified flowchart of finite element method framework.

For the linear static structural problems the finite element method is based on the matrix equation given in Equation 10. Here $[\mathbf{K}]$ is the stiffness matrix, $\{\mathbf{u}\}$ is the displacement vector and $\mathbf{P}$ is the applied force.

$$[\mathbf{K}]\{\mathbf{u}\} = \mathbf{P} \tag{10}$$

The stiffness matrix is calculated based on the selected mesh type, assigned material properties, boundary conditions and the performed discretization. Since applied loads are also defined at the start, given equation is solved for obtaining an unknown displacement field. Other derived quantities such as stress, strain, etc. can be computed using found displacement field values.

3.3 Topology Optimisation

Rosinha et al. [19] define Topology Optimization as *a mathematical method which spatially optimizes the distribution of material within a defined domain, by fulfilling given constraints previously established and minimizing a predefined cost function. For such an optimization procedure, the three main elements are design variables, the cost function and the constraints.* We will expand upon these 3 constraints in the Methods section. In practice, topology optimization is usually reduced to a discrete domain. In our case, where we will only deal with two dimensional shapes, the design domain is reduced to a grid-shaped form, with equal numbers of cells along both the horizontal and the vertical

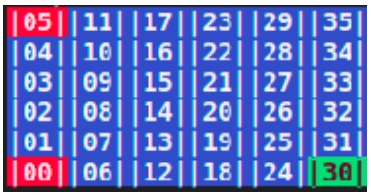


Figure 3: Initial Topology

direction. In the following we will reproduce the mathematical formulation of topology optimization from [4]:

$$\begin{aligned} &\min_{\rho} F \\ F(\mathbf{u}(\rho), \rho) &= \int_{\Omega}^{\Omega} f(\mathbf{u}(\rho), \rho) dV \\ s.t. \ : G_0 &= \int_{\Omega}^{\Omega} \rho dV - V_0 \leq 0 \\ \rho(x) &= 0 \ or \ 1, \forall \mathbf{x} \in \Omega \end{aligned} \tag{11}$$

The goal of optimization in 5 is to minimize F i.e. the compliance, which in turn will result in maximizing the stiffness of the structure. The design domain is represented by Ω and ρ represents a binary variable for each of the cells in the domain taking into account the presence or absence of material at that location. The first constraint represents the volume constraints G_0 which ensures that the volume will always be under a certain threshold V_0.
Solving a topology optimization problem with the help of reinforcement learning is a relatively unexplored idea. The most important example we have found is Brown et al. in [4], which we have used as a guideline for our project. Other works that combine the 2 concepts together tend to opt for approaches where the domain is more simplified than in our case, e.g. Hayashi and Ohsaki [8] where they solve a binary truss optimisation problem. Here, the domain is discretized as a set of binary trusses, who have to be removed in order to get to a minimal count.

3.4 Methods of Moving Asymptotes (MMA)

MMA is one of the most common algorithms to solve topology optimization problems. The main concept of the MMA algorithm is replacing the difficult nonlinear, non-convex optimization problem with a sequence of approximate convex subproblems that are easier to solve. The main algorithm proposed and explained in [22],and a detailed explanation is beyond our scope, since it's directly adopted from [2] to create a result to demonstrate the TO performance of a gradient-based method. Since JAX-FEM is a differentiable solver, it allows required gradient computations for MMA.

4 Methods

As mentioned in section 1.1 briefly, a typical topology optimization process is an iterative two step process. A topology optimization process begins with a predefined set of param-

eters, including the design parameters for which the structure will be optimized. Then in the first step, an FEM framework is used to evaluate design performance based on the defined design parameters. In the second step, an optimizer is used to modify the design parameters to advance the model performance regarding the set objectives and defined constraints.
In this study, we perform topology optimization for a 2D solid structure using deep reinforcement learning. For the design evaluation we used JAX-FEM [29]. As an optimizer, we develop a double deep q-learning (double DQN) agent to perform the optimal TO using OpenAI-Gym to define the environment and Keras for the implementation of the DQN [3, 5]. This section provides detailed explanations for our TO framework and its components.

4.1 Related Work

In the recent years a tremendous amount of FEM numerical solvers have emerged which boast of superior analysis accuracy at which they compute solutions or the range of problems that they can address [9]. Lately, however, there has been a surge in the use of automatic differentiation and machine learning based solvers in the scientific community. This is evident due to the fact that there exists an upper limit to accuracy when it comes to numerical schemes no matter how high order the scheme is. Automatic differentiation (AD) and ML frameworks compute gradients via the advent of chain rule and thus compute absolute derivatives with machine order precision [7]. An example of the former approach can be seen in [1]. Due to the success of DL based approaches in pattern recognition, Computer Vision and Language processing tasks, a lot of recent works now focus on integrating ML frameworks into the solver for their AD capabilities and their pattern recognition ability to simulate stochastic phenomenon [25], [14]. Apart from harnessing the power of AD, a lot of works have also resorted to exploring neural network architectures for accelerating existing solvers [28] and using novel architectures to compute solutions in material science [27], [10]. Our work aims to integrate Reinforcement learning to solve TO using the already AD based JAX-FEM solver.

4.2 FEM Framework

4.2.1 JAX-FEM

JAX-FEM is an open-source, differentiable FEM library for design optimization constructed on top of Google JAX [2], a rising ML library focused on high-performance numerical computation. JAX-FEM is written provides an accelerated framework for structural analysis. It is validated for several structural mechanics problems, including linear elastic problems, by comparing the results obtained with the state-of-the-art FEM solvers FEniCSx and Abaqus. This study uses JAX-FEM to create a structural linear elastic design for TO, and evaluate the design performance after each optimization step. The workflow in JAX-FEM is the same as in regular FEM software if the user wants to perform non-gradient optimization such as DQN. The workflow is explained in section 3.2, and practical utilization of JAX-FEM will be described more detailed in section 5. An illustration of a created 2D cantilever beam using JAX-FEM is given in Figure 4 below.

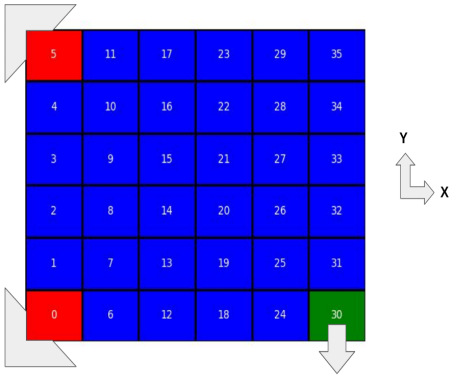


Figure 4: Initial topology of the 2D cantilever beam. Here the red cells represent the bounded cells, the green cell represents the loaded cell and the blue cells represent cells that contain material. The numbers indicates the cell indices. Cell indices are ordered with respect to the JAX-FEM logic. The reference coordinate system of JAX-FEM is given in the right hand side of the figure.

4.3 RL Framework

In order to implement an RL-based optimizer for for the discretized TO problem, the problem needs to be reformulated as a Markov Decision Process (MDP). MDPs are a discrete-time stochastic decision-making process. They use a mathematical framework to model the decision-making of a dynamic system in scenarios where the results are taken either randomly or controlled by a decision-maker sequentially in time. An MDP is built upon four elements, a state space (S), am action space (A), a transition probability function (P), and a reward function (R). If a TO problem can be expressed in terms of these elements, then the optimal TO design problem can be an MDP. For the MDP formulation we created an environment that contains S, A, P and R definitions and a DQN agent that interacts with the environment for optimal decision-making.
In the remainder of this subsection the environmental and agent related components of the RL framework will be described in detail.

4.3.1 State space

The state space, denoted as S, encompasses the complete set of possible observations that an agent can encounter during its interaction with the environment. The current observation of the agent depends on the current topology, boundary conditions, and loading conditions. Each state space observation is constructed as arrays with dimensions N x N x 3, where N denotes the number of cells along one dimension.
The state information about the current topology of the design can be represented by the stress distribution on the current topology. The von Mises stress is a widely utilized measurement in engineering design to describe the current stress state of an object. The calculation of von Mises stress for each cell can be performed using equation 12, where σ_x, σ_y and τ_{xy} are found using JAX-FEM. This information is encoded in the first channel of the state observation matrix in the form of normalized inverse von Mises values calculated using equation 13, where $\sigma_{VM,i}$ represents the von Mises stress for cell with index i, and σ_{VMmax} represents the highest valued von Mises stress among the cells in observation

space. The normalization is performed in order to prevent unbounded stress values.

$$\sigma_{VM} = \sqrt{{\sigma_x}^2 + {\sigma_y}^2 - \sigma_x \sigma_y + 3\tau_{xy}} \tag{12}$$

$$\sigma_{VMinv,i} = \left(\frac{\sigma_{VM,i}}{\sigma_{VMmax}} \right)^{-1} \tag{13}$$

The second and third channels of the observation state matrix consist of the boolean representations of the fixed and loaded elements in order. Fix and loaded elements are assigned a value of 1, while the unfix and unloaded elements are assigned a value of 0. An example of a simple 6x6x3 observation state matrix under a multi-loaded topology is given in Figure 5 below.

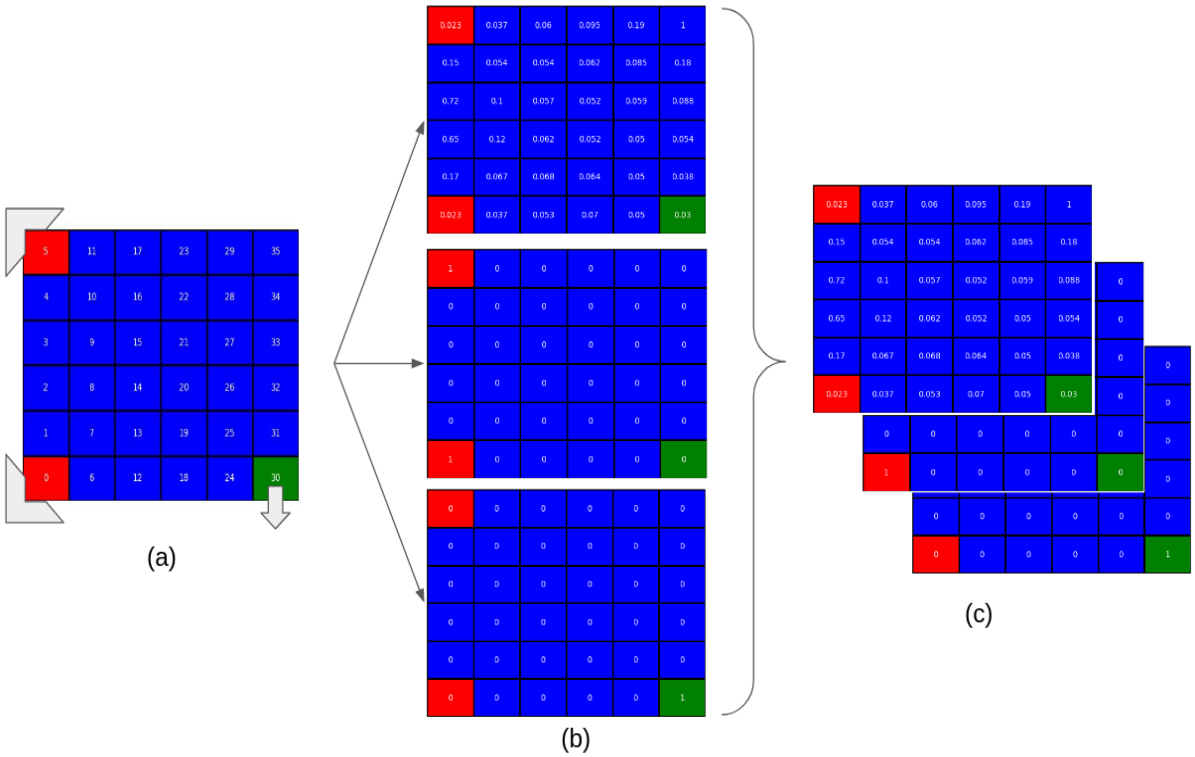


Figure 5: Observation representation. (a) Topology with cell indices. (b) Channels of state observation matrix in order. (c) State tensor used as input for DQN. (red = bounded cells, green = loaded cell, blue = cells with assigned material.)

4.3.2 Action space

The action space refers to all the possible actions the agent can take at each time step. In the context of our TO environment, an action corresponds to toggling a selected cell from material to void by changing the material density, in other words, the elasticity of the selected cell from 1 to 1e-4, since we assume that for this study, the elasticity of each

cell equals to the assigned material density of each cell. The action space size for an N x N topology environment is equal to N^2. In our scenario, the action space is an array that contains the index number of each cell in the topology. Some of the possible actions are defined as illegal actions in the action space. These illegal actions include removing a bounded, loaded, or previously voided cell or removing a cell that causes a singularity in the topology as a result. Such actions are demonstrated in the Figure 6.

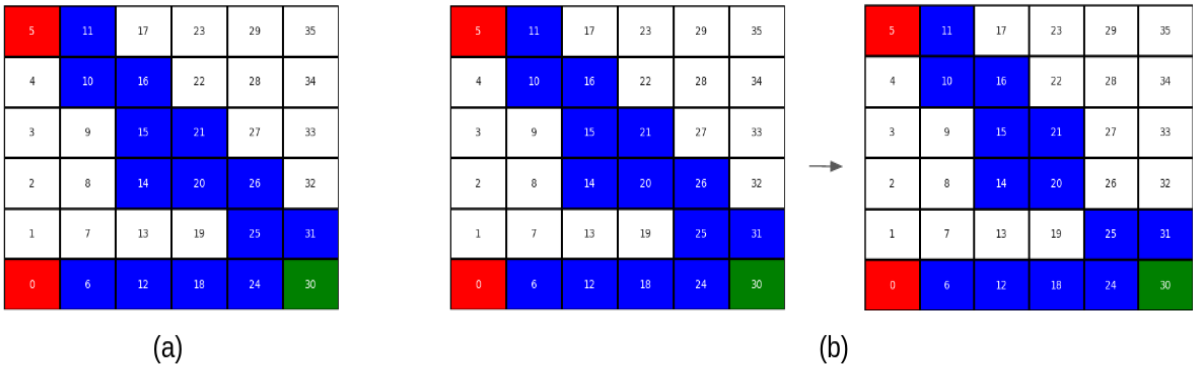


Figure 6: Illegal actions. (a) Removing voided (white), loaded (green), bounded (red) cells. (b) Action results in non-singular body such as removing cell 26 for given example.

4.3.3 Reward

The reward formulation is motivated by encouraging the agent to take effective actions that improve the design into the optimal design. For taking the desired actions, the agent gains positive rewards, whereas for taking the illegal actions, it is penalized by negative rewards. The agent always seeks to accumulate the most possible reward during the episodes and tries to learn the best action sequence to achieve this goal. For this TO problem, the proposed reward function is used in [4] as in equation 14. Here c_s is the initial strain energy of the solid block topology, c_t is the strain energy of the current topology at the time step, t, α_t is the number of voided elements at timestep t, and N^2 gives the total number of cell in the topology. This equation means that the agent is assigned more reward if the topology exhibits minimal increases in strain energy after each taken action while reducing volume fraction.

$$r_t = \left(\frac{c_s}{c_t}\right)^2 + \left(\frac{\alpha_t}{N^2}\right)^2 \tag{14}$$

4.3.4 Training

In this part we will describe the basic workflow of the DQN Agent while training. Firstly, we initialize a solid 6x6 block of material which is discretized into 36 cells. Afterwards, we randomly generate the Dirichlet conditions, which represents the points at which we are fixing out structure. We do this for 2 cells, whose points are randomly selected. If the 2 cells have no common edges we will choose 2 points for each cell, if they have a common edge we will choose only one point for each cell and if one of the cell is a corner

we will choose the point that represents one of the 4 corners of the structure. Next, we are randomly choosing a cell which will contain the Neumann boundary condition i.e. the cell where the force is applied. The direction (vertical, horizontal in or against the positive coordinate of the axis) as well as its value are randomly sampled. It is worth mentioning that in the debugging phase of this part we usually used fixed boundary and force conditions in order to be able to track the progress of our implementation better.
Secondly, we use the FEA Implementation provided by JAX in order to calculate the required quantities of the current configuration: inverse von Mises, displacement, strain energy. By using the results of the FEA, we create the following 3x6x6 state matrix of the current timestep t: the first dimension contains the normalized inverse von Mises, the second one and the third dimensions are both binary matrices and contain the value 1 only where the cell is used for the Dirichlet boundary condition and the Neumann conditions, respectively. However, due to time constraints we aimed at training our DQN model for the configuration stated in 4. Thirdly, we choose a cell to be removed. This decision can be computed in 2 different ways. The first procedure selects a cell at random, while the second one makes use of the DQN model. The training procedure starts with an empty memory buffer. This is why, in the beginning we are encouraging the use of random actions, which we are calling *exploratory*, over actions that are recommended by the DQN, which we name *exploitative*. Hence, we are introducing a new condition: the memory buffer should have a given minimal length, on which the DQN has previously trained in order for the DQN o be used. If this is not he case, we will use the exploratory approach. However if the DQN was trained at least once we will use the aforementioned greedy policy to decide if we are taking an exploratory or an exploitative approach. No matter the approach, this step will return us the ID of a cell that has to be removed. If the removal of the cell does not render a non-singular body, we calculate with the help of JAX the new quantities for the given body. We can now enhance the memory by adding a new element made of: the previous and current state, the action the model took (i.e. the removed cell) and the reward that was achieved by this action. However, if the action was illegal or if after the removal of the last cell the volume fraction has fallen under a predetermined margin we consider this episode over. If the episode is not over, we will repeat a cell removal step.
Finally, after the episode is over we retrain the main DQN model on a sampled batch from the memory buffer. Moreover if the episode count has reached a certain limit (5000) we end the training. However, if the episode number is also divisible through 100 we assign the weights of the DQN model to the auxiliary model which we use for calculating the targets.

4.3.5 Testing

Until now we have only trained our agent on a 6x6 scenario. What makes topology optimization different from the usual tasks that are solved by reinforcement learning is the action space and the consequences a single action has on the whole state space. When we compare our scenario with other typical scenarios from reinforcement learning we can see this difference better. We take for example the classic Atari game called *Breakout* or the toy example we have talked about in 3. We see the the action space is very small and it is represented only by actions such `left` or `right`. Now if we look at our case we

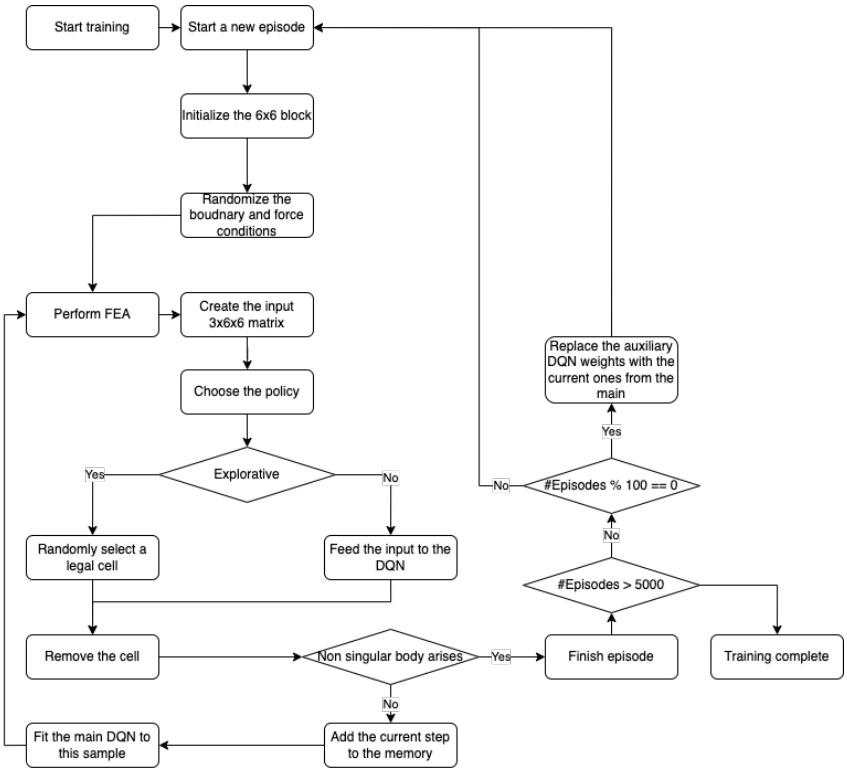


Figure 7: Episodic training flowchart (adapted from [4])

will see that we theoretically have 6x6=36 actions i.e. we can remove any of the 36 cells. We reduce this number to 33 by not considering the boundary conditions. Moreover, the number keeps getting smaller with every step because we do not consider already removed cells as viable action. However, in topology optimization removing a cell will have a direct impact on every other subsequent step. This is not true for most of the reinforcement learning scenarios. Take for example the cartpole example where the states can become periodical. Our scenario cannot present such behaviour.

One of the our goals was to try implement the cell removal procedure for bigger scenarios than 6x6, more specifically for a 24x24 scenario. However, this means an action space of 24x24=576. Running the DQN agent directly on this would prove infeasible as it would become increasingly difficult to generalize on these dimensions. In order to solve this problem, Brown et al. present in [4] a procedure called *Progressive Refinement*. The procedure functions in the following way: it starts with a 6x6 solid block where the boundary conditions are defined. Following this we apply the DQN in order to remove the unnecessary cells, which will return us a new design. We upscale this new design to 12x12. After repeating the same procedure for the 12x12 and the 24x24 case we will get an optimized design for the final dimensions.

In both the 12x12 and 24x24 scenarios we are not training the agent anymore and we are only using the DQN model we have trained for the 6x6 case. Even though the in- and output dimensions are different, the DQN will still function for these 2 cases thanks to the convolutions ratio it has. Intuitively, the progressive Refinement procedure would mean that a cell in the 6x6 scenario represents 4 cells in the 12x12 scenario and 16 cells in the 24x24 scenario. This means that if we are performing the operations with a volume

fraction goal given in the 24x24 space we have to redefine it in the 12x12 and 6x6 case accordingly. Unfortunately, we were not able to test the Progressive Refinement due to lack of time.

4.3.6 DL Architecture

In our case the task of the neural network is to return an array of shape 6x6 that represent the approximated Q values. The Q value of a cell represents the expected Q value of the former state in combination with the action of removing the respective cell.
We have chosen to replicate the DQN Model Architecture used by Brown et al. in [4]:

```
class _build_model(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.conv = Sequential()
        self.conv.add(Conv2D(16,(3,3),padding='same' ,activation='relu',
        ↪  input_shape=state_size))
        self.conv.add(Conv2D(8,(3,3),padding='same',activation='relu'))
        self.conv.add(Conv2D(4,(3,3),padding='same',activation='relu'))
        self.conv.add(Conv2D(1,(3,3),padding='same', activation='relu'))
        ...
```

Listing 1: DQN Model Architecture

5 Implementation

In Background we defined the basic concepts of our project such as reinforcement learning and topology optimization. The following section 4 presented known solutions to these concepts. In this section we are going to present how these solutions were implemented into the project code.
Firstly, we are going to present the `fem_model.py` where we are creating `jax` methods for our goals. Secondly, there is `problem.py` which contains the most important elements of our simulation, that allow us to randomly create scenarios for the topology optimization problem. Thirdly, we will present `env.py`, which will serve as an interface between our DQN Agent and the `jax` environment. Lastly, we are going to present the way in which the reinforcement learning is performed with the help of a DQN in `optimizer.py`.

5.1 fem_model.py

This python file contains a class called `Elasticity` that uses inheritance from the `FEM` class located in the JAX-FEM `core` module. This module performs main FEM calculations in order to solve second-order elliptic PDE problems including linear elasticity as defined in [29].
The `Elasticity` class is used for structural model definition. We used a linear elastic model defined in the `get_tensor_map_box` method to perform the required stress calculations for the creation of the first channel of the observation state tensor that contains

the normalized inverse von Mises stress values for each cell in the topology and the reward function calculation.

5.2 problem.py

This python file contains a class called `ProblemSetup`. The class is used for setting up the topology optimization problem. It inherits the `Elasticity` class that we mentioned in the previous section.
Firstly, we set up the topology optimization problem. This class creates a discretization of the topology based on the passed variables that define the desired discretization properties, such as the number of cells along the dimensions denoted as N, the total length of the topology along each dimension denoted as L, the `dim` variable that defines the dimensionality of the TO problem. Also, we hard-coded the element type as *'QUAD'* under the `__init__` class to discretize the topology with homogeneous quadratic meshes. Discretization is performed automatically under the `__init__` method each time the class is instantiated.
Secondly, this class has several methods to create randomized boundary conditions for TO problem in the beginning of the each episode as it is described in the Training section. This randomized selection process is applied according to the defined methodology by Brown et al. [4]. `select_bounded_and_loaded_cells` method is used to select random cells among the topology domain for imposing Dirichlet (fixation) and Neumann (loading) boundary conditions. It uses the defined `num_bounded_cell` and `num_loaded_cell` parameters and a helper method called `_categorize_cells`. The `_categorize_cells` method categorizes the cells based on their locations in the topology and returns this information, such as corner cells, edge cells, outer cells and inner cells. Based on this information the bounded cells are randomly selected in order among the outer cells and assigned to `bounded_cell_inds` variables. These cells are then removed during the random loaded cell selection and loaded cells are selected randomly among the remaining cells. Since the boundary conditions cannot be applied on the cells but on the nodes, `select_bounded_and_loaded_points` method is used to select the corresponding nodes around the selected cells to assign boundary conditions. After the corresponding node selection, `set_dirichlet_bc` and `set_neumann_bc` methods are called to create proper inputs for the boundary condition assignment in JAX-FEM. After creating the proper inputs with the desired format, these inputs are passed to the `problem_define` method to create an `Elasticity` instance as a last step of the randomized TO problem definition.
Thirdly, this class has some methods that are useful for training. One of them is called `problem_solve`. This method takes the objective design parameter cell material density array as an input and performs a one time-step forward simulation using the JAX-FEM solver. This method is called in the training and testing at each step during the each episode to perform state update based on the selected action using modified material density array called *rho*. Another useful methods is `create_state_space_tensor`, which creates the required observation state input described in State space for the DQN model training.

```
1 def problem_solve(self, problem, rho: np.ndarray):
2     fwd_pred = ad_wrapper(problem, linear=True, use_petsc=True)
```

```
    rho = rho.reshape((-1,1))
    return fwd_pred(rho)
```

Listing 2: problem_solve method

`update_density` is an another method that automatically updates the material density value for a selected cell. It is used to modify the material density array after each taken action during training and testing. Also, positive reward calculation after each valid action of the agent during the training is done by calling the `positive_reward` method. Lastly, the illegality of each taken action is controlled by calling `check_illegal` method. It performs the validity check based on the criteria described in the Action space section and returns true if the termination criteria have been met; otherwise returns false.
Finally, the `ProblemSetup` class has some helper functions for array and matrix manipulation. The `_state_matrix_from_array` method e.g. takes an array and converts it into a matrix in the same order with JAX-FEM indexing logic. `_state_array_from_matrix` on the other hand is used for performing the reverse operation. These methods are used when we need to convert the operations in the JAX-FEM indexing logic and vice-versa.

5.3 env.py

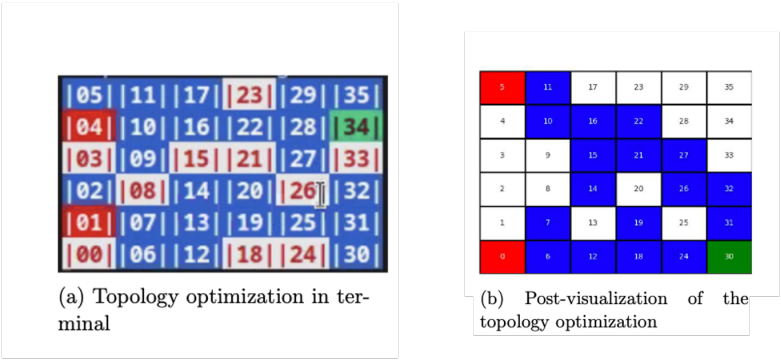
For the environment creation, we chose to use the `Gym` package, which is a toolkit developed for reinforcement learning research. It includes a diverse collection of pre-built scenarios (called environments) and enables to create new custom environments with a common interface.
A new environment can be constructed by creating a class that inherits from the `gym.Env` class of OpenAI Gym. This created class has two main attributes called `observation_space` and `action_space`, and three main methods called `reset`, `step` and `render`.
The `observation_space` and `action_space` attributes give the format of valid observations and actions in the environment. In our case these are the 3-dimensional matrix containing the values of the FEM simulation along with the boundary and force conditions and the 36 possible cells respectively.
Since the MDP is a sequential process, the `reset` method is called when the agent takes an action that triggers termination criteria to automatically reset the environment to an initial state, and returns the initial observations at the beginning of each episode. In this way, we ensure that we start every episode with all the parameters reset.
The `step` method advances the dynamics of the environment by one timestep (see figure 13) when it is called. It contains strategic information that the agent follows during the training and later in the testing phase. It takes an action as an input and returns a tuple that contains the `current_observation_state`, the `current_action`, the `reward` that is gained based on the taken current action, the `next_observation_state` as a result of the taken action, and the `termination` information as a boolean whether the termination criteria has been reached after taking the current action. In the following lines we present snippets from the definition of `step` function, where one can see the interaction it has with the `jax` model. The reader is advised to see the Code of env.py section for the whole method.

```
def step(self, action): # action is the cell to be removed
    #...
```

```
3       if self.jax_model.check_illegal(self.rho_matrix, cell_to_be_removed,
        ↪  self.current_state_tensor_check, self.nb_removed_cells,
        ↪  self.max_num_step):
4       else:
5           #...Recompute the new von Mises values...
6           rho_vector, rho_matrix =
            ↪  self.jax_model.update_density(self.rho_vector,
            ↪  cell_to_be_removed)
7           self.rho1d = rho_vector.reshape((-1,1))
8           solution = self.jax_model.problem_solve(self.problem, rho_vector)
9           von_mises = self.problem.compute_von_mises_stress(solution)
10          #...Calculate the reward...
11          reward = self.jax_model.positive_reward(self.init_SE,
            ↪  self.curr_SE, self.nb_removed_cells, self.size_x*self.size_y)
12          #...Recompute the state tensor...
13          self.next_state_tensor_DQN, self.next_state_tensor_check=
            ↪  self.jax_model.create_state_space_tensor(rho_vector,
            ↪  von_mises, self.bounded_cells, self.loaded_cells)
14          #...
15      return self.current_state_tensor_DQN, action, reward,
        ↪  self.next_state_tensor_DQN, terminated
```

Listing 3: Step method

The `render` method is used for visualization of the action of the agents and the results of the environment. This method is optional, and the absence of it does not affect the learning performance of the agent. However, we have chosen 2 alternative ways to this. The first visualization (see figure 8) mode we are using is the real-time one. We are using the `colorama` package that allows us to color the ANSI characters in the terminal. Whenever we are printing the current step we are printing additional details with it such as: the reward we get from that step, the cell that was removed, the strain energy etc. The color scheme we are using is the following: red for the boundary conditions, green for the force location, blue for simple cells that are not yet removed and the removed cells are white. Moreover we are also annotating these cells with their index in our matrix.



(a) Topology optimization in terminal

(b) Post-visualization of the topology optimization

Figure 8: Visualisation modes

Even though this type of visualization allows us to inspect the DQN Agent in real time, it is not a good option for post-visualization or for storing old episodes. This is why we have implemented a second type of visualization (see 8). This is an extension to the current visualization function. Every time a new step is taken we create a plot which uses the visualization matrix from the last paragraph along with a color map using the `heatmap` method of the `seaborn` library. This newly created plot is saved as a `.jpg` in a special folder for the current episode. After an illegal move was performed, and RL trajectory subsequently terminated, we go through all the steps of the current episode and combine them into a `.gif` in the correct chronological order. In this type of visualization we are not using the indexing anymore.

5.4 optimizer.py

The DQN agent is created in the Python class. We used the Keras library to build the CNN based DQN model, which takes state observation tensor as an input and returns the possible best action as an output by learning the possible best policy for the TO problem during training. The DQN agent class has three main methods called `remember`, `train` and `act` and some attributes related to DL model, some to the TO environment and some to the learning policy.
In the beginning of the file we define some variables whose value will remain constant throughout our implementation, such as the dimensions of the body or the density of the voided elements. We create an instance of our `jax` simulator by instantiating a `ProblemSetup` object. After this we create the environment with which the DQN Agent will interact by instantiating a `TopOptEnv`. Finally we create our neural network architecture which we have presented earlier with the help of `tf.keras`. We also add an additional post-processing step to the architecture, that rearranges the elements of the matrix in the order we have presented in previous figures.
In the following part we are creating the `DQNAgent`'s class, the most important part of the reinforcement learning paradigm. In the initialization of the agent we specify both the state and action space size, these are useful in keeping track of their dimensions. Other important attributes, whose values were fixed are:

- `memory = deque(maxlen=30000)`: the data structure that stores tuples of past experiences in the `(state, action, reward, next_state, done)` format. Through the method `remember(self, state, action, reward, next_state, done)` the agent is able to enhance it after taking a new action an observing the new state.

- `gamma = 0.1`: the value with which we are discounting the future experiences, its role was already explained in Background.

- `epsilon = 0.9`: the initial value for `epsilon`, representing the probability of taking a random action by the agent, i.e. not using the neural network.

- `epsilon_decay = 3.5e-4`: the decay factor for `epsilon`. After learning to approximate the outcomes of a new action, the `epsilon` is given the following value: `1 - episode_num * epsilon_decay`. This ensures that with time, the agent will use the Q-Network more often.

- `epsilon_min = 0.01` if `epsilon` falls below this value, it will be assigned this value. In this way we ensure, that the agent will always test new actions, but not as often as in the beginning.

- `learning_rate = 2.5e-3` the value we are using for the gradient descent

Now we are going to describe the way in which the DQN Agent helps create a network capable of predicting the best action according to the current state. In order to do this, we will present the learning loop and explain every important method along the way.
We are going to simulate 5000 episodes. An episode is considered finished when the DQN Agent is trying to perform an illegal action. When the Agent is using a random action all the illegal actions with the exception of the non-singularity inducing ones are filtered out. However, when using a Q-Network prediction every type of illegal action can be returned. Each training loop begins by calling the `reset` function of the environment. This sets the two dimensional body to its complete state and, if we are not hard-coding boundary- and force-conditions, new locations for the fixed points and the force are also generated. We set the `done` variable to `False` (it will be set to `True` when the agent will have taken an illegal action. While the boolean does not change its value we are performing the following steps:

1. `action = agent.act(state)`: This method will return the action to be taken according to the epsilon-policy. The epsilon policy chooses with probability `epsilon` a random cell of the body (in this case the cells will already be filtered in order to exclude the most illegal actions). However if the randomization results in a different outcome, we will use the Q-Network in order to predict the new action. The network will assign a Q-value for each of the cells and we are choosing the highest one.

2. Knowing what action we have to apply we will perform `state, action, reward, next_state, done = env.step(action)` i.e. we simulate this action in the environment and we also register the additional information.

3. We enhance the agent's memory with the results of the current step through `agent.remember(state, action, reward,next_state, done)`

After `done` becomes `True` we have completed the present episode. After the episode we call the training function which we describe in the following paragraph. If the episode number is higher than 0 and a multiple of 100 we set the weights of the auxiliary model to the same weights as those of the main model. Moreover, at every 50th episode we are saving the model's weights in a `.hdf5` format model snapshot.
After we are done with simulating the steps of an episode, the last remaining task for this episode is learning the new transitions and their rewards, we do this through `agent.train(batch_size, episode_number)`. The first argument is used to sample a batch of samples of that size from the memory, while the second argument is used only for decaying the epsilon. Due to the fact that every sample is formed out of the state and its successor, the action, the reward and the finished state we are creating the following variables which contain a list of each element type: `states, states_nxt, actions, rewards,`
`dones`. In lines 3 and 4 from the code listing 4 we calculate the Q-values with the **main**

model. In the following line we do the same for the next states with the **auxiliary** model. As previously stated the auxiliary model uses old weights of the main model. Van Hasselt et al. [24] suggest the use of the Double DQN which implies the existence of a second model in order to reduce the overestimation from the main model. If the sample is a finished one we assign the q-value in that step the reward value, as an increase is no longer possible. If however the batched step is not finished, we reinterpret the iterative Bellman equation 6 where Q^* is represented by the **auxiliary** model and Q by the **main** model. After the discounted reward is calculated we call the `.fit` function of the Keras model.

```
1     def train(self, batch_size, episode_num):
2     ...
3         targets = onp.array(self.model(states))
4         targets_nxt = self.model(states_nxt)
5         targets_val = self.model_target(states_nxt)
6         for i in range(batch_size):
7             if dones[i]:
8                 targets[i][actions[i]] = rewards[i]
9             else:
10                a_max = np.argmax(targets_nxt[i])
11                targets[i][actions[i]] = rewards[i] + self.gamma *
                      ↪  targets_val[i][a_max]
12
13        self.model.fit(states, targets, epochs=1)
14    ...
```

Listing 4: Double DQN Training

5.5 MMA implementation

MMA implementation performed using `MMA.py` module in [2] after creating the same TO problem for the RL training calling the required methods from `problem.py`. Also a volume fraction variable *vf* is defined as a constraint to the MMA optimization that defines ratio of the allowable remaining total material density after optimization to the total material density in the initial topology.

6 Evaluation

In this section we describe the way in which we are assessing the quality of our predictions. We will first introduce a qualitative method and afterwards a quantitative method.
Topology optimization is traditionally realized through analytical gradient-based methods. One of the possible implementations is the *Method of Moving Asymptotes (MMA)*. Due to the fact that this is only a qualitative assessment we have not created a metric for this part. Instead we want to find out if our scenario is able to recreate the basic geometry of an optimized topology. In the following section we will conduct an analysis by comparing the 2 results: one from the gradient-based topolgy optimization and the other one created by our reinforcement learning model.

The second modality in which we are testing the performance of our model is by assessing the quality of the predictions. We will follow a metric similar to the one used by Minh et al. [18] in their paper about Atari games. The first quantity that comes into one's mind when trying to characterize the *goodness* of a prediciton is the average reward per episode. However, as Minh et al. state in their paper, the evolution of the reward can be extermely noise. This is most probably because of the instantaneous nature of the reward i.e. the reward changes between every consecutive steps. In order to solve this issue, we are moving our attention to a cumulative metric, i.e. the Q-value function. We will average the returned values by the Q-value functions during every episode (this only happens when we are using the DQN Network for predictions). One needs to highlight the fact that as the episode index increases we will also see more DQN calls inside an episode, hence the values should become more stable.

7 Results

The Double DQN model was trained to optimize the topology given in 4. The initial topology comprised of 6×6 elements, with the fixed Dirichlet boundary conditions along the left vertical and loaded at the bottom right as shown in 3. One of the main reasons this task of topology optimization is difficult is because of the increased action space of the system (33) and consequently the high number of possible states due to those actions. It is critical that the hyperparameters are tuned to maximize the learning performance of the agent. After exhaustive experimentation and careful tuning, the RL agent was able to learn the removal of optimal elements from the topology which resulted in minimal increases in strain energy at every step.
In the initial stages of development, the agent was having a hard time learning the strategy. After exploring and revisiting various concepts in Double DQN based RL, we were able to optimize the hyperparameters and in this process boost the performance of the learning strategy significantly.
Contrary to popular Double DQN implementations which employ a discount factor γ close to 1, in our instance it was possible for the agent to learn by focusing more on the immediate rewards. Hence $\gamma = 0.1$ gave considerably better results than $\gamma = 0.9$, which is primarily used for Double DQN based RL where it is to penalize near-term rewards, and instead prioritize long-term future rewards, as commonly seen in reinforcement learning applied to Atari games ([18]). This can be fairly easily understood by studying the input inverse of the von Mises state given to the network (see figure 9). The value of the normalized inverse von Mises stress at every step is the highest where removing the element is most optimal, since the highest inverse von Mises stress corresponds to the least loaded cell in the topology at any time. In the figure 9 its evident that the first element to remove is element number 35 since it has highest normalized inverse von Mises stress value equal to 1.

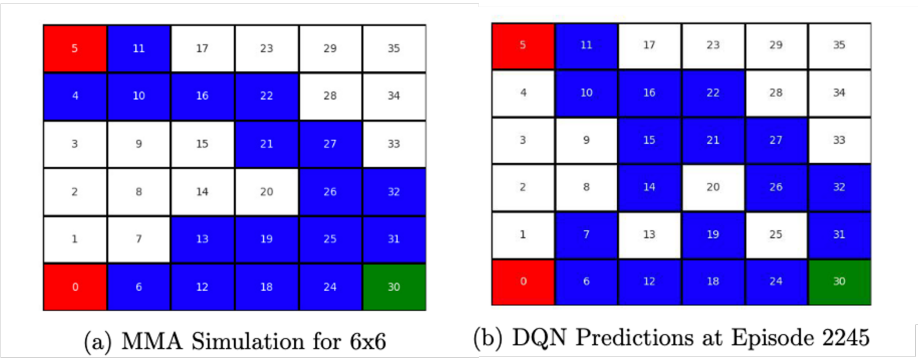
$$\begin{bmatrix} 0.02308 & 0.03743 & 0.06002 & 0.09509 & 0.19225 & 1. \\ 0.14993 & 0.05408 & 0.05367 & 0.06164 & 0.08537 & 0.18436 \\ 0.72457 & 0.10116 & 0.05704 & 0.05193 & 0.05894 & 0.08756 \\ 0.64509 & 0.11533 & 0.06246 & 0.05178 & 0.05005 & 0.05383 \\ 0.16998 & 0.06736 & 0.06823 & 0.06441 & 0.04954 & 0.0378 \\ 0.0234 & 0.03709 & 0.05334 & 0.06962 & 0.05002 & 0.03028 \end{bmatrix}$$

Figure 9: Normalized inverse von Mises stresses

It was observed during the development stages that the target model was getting updated more frequently than intended. This was leading to overestimation of future reward values. A deque memory buffer of size 30000 was kept to store recent states, rewards, actions and termination of the episodes.

7.1 Qualitative assessment

Due to the MMA approach being a traditional numerical approach we can assume that the design it delivers is optimal. That is why we consider that, when a model manages to replicate the basic shape of that solution, it is on the right path. For our specific 6x6 example where the fixed boundary points and the force origin are placed on opposite sides, the material that has to be removed first is located in the upper right corner and in the space between the 2 bounded cells. This means that in the end, our DQN agent has to create a topology that resembles the topology created through the MMA method (see figure 10). We can see that even after 2245 episodes the model starts to create designs that resemble the correct topology. For example in the following figure the differences between the 2 topologies can be resolved by moving 4 of the cells.



(a) MMA Simulation for 6x6 (b) DQN Predictions at Episode 2245

Figure 10: Comparison of solutions

7.2 Quantitative Assessment

As stated by Mnih et al. [18] a good way to assess the performance of the model is by plotting the rewards and Q-values averaged by episodes. The reward plot is more noisy

than the Q-values plot, but this is due to the fact that the reward depends only on the current step.
Therefore, a better candidate for the quantitative assessment is the Q-value function. One reason for this is because the Q-value considers both the reward of the current step and the discounted rewards of he following steps. In the figure 11, it is seen that Q-values are noisy during the initial episodes and then later seem to stabilize as the agent starts to transition into the exploitation phase. The former is because most of the actions are taken at random to explore more the action space and the number of DQN calls are lower in number. The values are still noisy during the end as the agent needs to still train for a large number of episodes to attain a more stable Q-value. However as seen initially the value become lesser noisy as agent trains and its further understood that the values would stabilize for episodes in the range of 10,000. It should however be noted that the two plots are from different training loops, which went on for 5000 11 and 2000 12 episodes respectively.

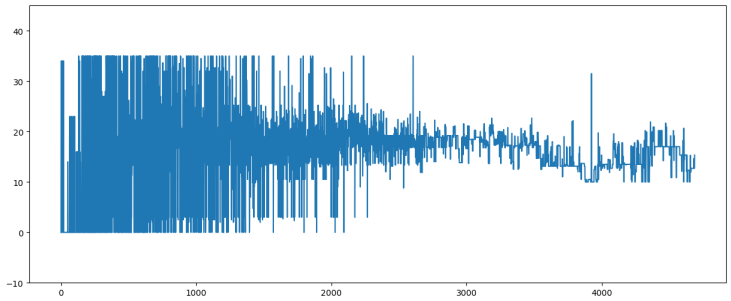


Figure 11: Averaged Q-Values for the first 4000 episodes

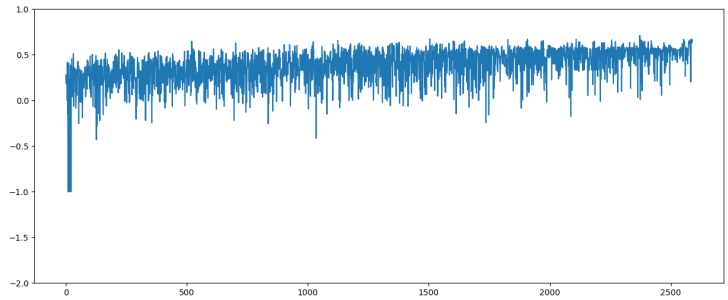


Figure 12: Averaged Rewards Values for the first 2000 episodes

8 Conclusion

We used an existing FEA simulator from JAX-FEM and successfully integrated it with a Double DQN based RL model written in Keras closely adopted along the lines of [4]. The simulator was used to generate the state for the environment and the DQN agent worked towards achieving a design objective by taking it as an input. It was observed that the Double DQN does a good job at learning the topology optimization procedure, and is able to remove elements based on the FEA based reward. By further exploring the

hyperparameters space by varying the γ, memory size, auxilliary model update frequency, epsilon decay rate and network architecture, better training performance is possible. The trained Double DQN thus gives topologies on par with the numerical MMA from the authors of JAX-FEM, however it still has significant work left to be done to make it as accurate and efficient to traditional numerical approaches.
Given the versatile nature of Reinforcement learning, it can be adapted to solving many such physical problem, if physics aware environments and rewards are cleverly designed.

9 Future Work

Future work includes changing the model architecture and introducing a more informative input to the Double DQN, like adding resnet based skip connections and dense layers, and introducing an additional feature channel that stores the voided elements respectively. One problem we stumbled upon while training the DQN model was that it repeatedly tried to remove voided cells. We believe that, by enhancing the input of the model with a mask of the already voided cells, the predictions will be able to lower the Q-value for the removed elements. Hence, the probability of removing a voided element for a second time would be reduced.
The model trained with these approaches can then be used and experimented on larger topologies. For example after the current TO results in an optimal state the topology can be expanded by a factor of 2 or more and the resulting topology can once again become an input to the trained network to handle more realistic topology optimization problems successively [4], by performing the aforementioned *Progressive Refinement*. A good future direction would be to take the same optimization based problem and extend it to aerodynamic shape optimization where the agent receives the forces on the initial topology as input and successively removes elements to optimize for the lift to drag ratio. However, work still needs to be done to make it as accurate, if not more, than traditional approaches to solve physical problems, and even better, to integrate it any a way that it assists the traditional approaches to compute faster and more accurate solutions.
Improvements can also be done on the computational side. Our approach use the solutions provided by Keras for creating the learning model. However, it uses only the CPU for computation. Lately, there have been improvements in the field of GPU-based solvers. Some noteworthy examples are Isaac Gym by Makoviychuk et al. [17] where both the physics simulations and the learning on the agent side are done simultaneously on the GPU by allowing the two to communicate via tensors buffers or PureJaxRL by Lu et al. [15], which is able to run a large number of RL agents in parallel on GPUs. Redesigning our implementation according to any of these frameworks could prove more efficient not only in the time required for the learning, but also in the accuracy of the solutions.

References

[1] Bezgin, D. A., Buhendwa, A. B., and Adams, N. A. Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. *Computer Physics Communications 282* (2023), 108527.

[2] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018.

[3] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

[4] Brown, N. K., Garland, A. P., Fadel, G. M., and Li, G. Deep reinforcement learning for engineering design through topology optimization of elementally discretized design domains. *Materials & Design 218* (2022), 110672.

[5] Chollet, F., et al. Keras. `https://keras.io`, 2015.

[6] Eckardt, J.-N., Wendt, K., BornhÃ¤user, M., and Middeke, J. M. Reinforcement learning for precision oncology. *Cancers 13*, 18 (2021).

[7] Griewank, A., and Walther, A. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[8] Hayashi, K., and Ohsaki, M. Reinforcement learning and graph embedding for binary truss topology optimization under stress and displacement constraints. *Frontiers in Built Environment 6* (2020).

[9] Kamensky, D., and Bazilevs, Y. tigar: Automating isogeometric analysis with fenics. *Computer Methods in Applied Mechanics and Engineering 344* (2019), 477–498.

[10] Kanno, Y. A kernel method for learning constitutive relation in data-driven computational elasticity. *Japan Journal of Industrial and Applied Mathematics 38* (2021), 39–77.

[11] Kober, J., Bagnell, J., and Peters, J. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research 32* (09 2013), 1238–1274.

[12] Kollmann, H. T., Abueidda, D. W., Koric, S., Guleryuz, E., and Sobh, N. A. Deep learning for topology optimization of 2d metamaterials. *Materials & Design 196* (2020), 109098.

[13] Lei, X., Liu, C., Du, Z., Zhang, W., and Guo, X. Machine learning driven real time topology optimization under moving morphable component (mmc)-based framework. *Journal of Applied Mechanics 86* (08 2018).

[14] Lindsay, A., Stogner, R., Gaston, D., Schwen, D., Matthews, C., Jiang, W., Aagesen, L. K., Carlsen, R., Kong, F., Slaughter, A., et al. Automatic differentiation in metaphysicl and its applications in moose. *Nuclear Technology 207*, 7 (2021), 905–922.

[15] Lu, C., Kuba, J., Letcher, A., Metz, L., Schroeder de Witt, C., and Foerster, J. Discovered policy optimisation. *Advances in Neural Information Processing Systems 35* (2022), 16455–16468.

[16] Lu, J. Protein folding structure prediction using reinforcement learning with application to both 2d and 3d environments. Association for Computing Machinery.

[17] Makoviychuk, V., Wawrzyniak, L., Guo, Y., Lu, M., Storey, K., Macklin, M., Hoeller, D., Rudin, N., Allshire, A., Handa, A., and State, G. Isaac gym: High performance gpu-based physics simulation for robot learning, 2021.

[18] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning, 2013.

[19] Rosinha, I. P., Gernaey, K. V., Woodley, J. M., and Kruhne, U. Topology optimization for biocatalytic microreactor configurations. In *12th International Symposium on Process Systems Engineering and 25th European Symposium on Computer Aided Process Engineering*, K. V. Gernaey, J. K. Huusom, and R. Gani, Eds., vol. 37 of *Computer Aided Chemical Engineering*. Elsevier, 2015, pp. 1463–1468.

[20] Shao, K., Tang, Z., Zhu, Y., Li, N., and Zhao, D. A survey of deep reinforcement learning in video games, 2019.

[21] Sosnovik, I., and Oseledets, I. Neural networks for topology optimization, 2017.

[22] Svanberg, K. The method of moving asymptotes a new method for structural optimization. *International Journal for Numerical Methods in Engineering 24* (1987), 359–373.

[23] Ulu, E., Zhang, R., and Kara, L. A data-driven investigation and estimation of optimal topologies under variable loading configurations. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization 4* (08 2015), 1–12.

[24] van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning, 2015.

[25] Vigliotti, A., and Auricchio, F. Automatic differentiation for solid mechanics. *Archives of Computational Methods in Engineering 28*, 3 (2021), 875–895.

[26] Woldseth, R. V., Aage, N., Bærentzen, J. A., and Sigmund, O. On the use of artificial neural networks in topology optimisation. *Structural and Multidisciplinary Optimization 65*, 10 (oct 2022).

[27] Xue, T., Adriaenssens, S., and Mao, S. Learning the nonlinear dynamics of soft mechanical metamaterials with graph networks. *arXiv preprint arXiv:2202.13775* (2022).

[28] Xue, T., Gan, Z., Liao, S., and Cao, J. Physics-embedded graph network for accelerating phase-field simulation of microstructure evolution in additive manufacturing. *npj Computational Materials 8*, 1 (2022), 201.

[29] Xue, T., Liao, S., Gan, Z., Park, C., Xie, X., Liu, W. K., and Cao, J. JAX-FEM: A differentiable GPU-accelerated 3d finite element solver for automatic inverse design and mechanistic data science. *Computer Physics Communications 291* (oct 2023), 108802.

[30] Yang, T. Advancing non-convex and constrained learning: challenges and opportunities. *AI Matters 5*, 3 (2019), 29–39.

A Project Code

A.1 Code of problem.py

```python
import numpy as onp
import jax
import jax.numpy as np
import os
import glob
import matplotlib.pyplot as plt
from scipy.ndimage.measurements import label

from jax_am.fem.core import FEM
from jax_am.fem.solver import solver, ad_wrapper
from jax_am.fem.utils import save_sol
from jax_am.fem.generate_mesh import get_meshio_cell_type, Mesh
from jax_am.common import rectangle_mesh
from fem_model import Elasticity

os.environ["CUDA_VISIBLE_DEVICES"] = "2" # --> Only activate when there is a CUDA-device in the system

def _clear_previous_output_files(self):
    """
    Clears previous outputs in the current folder.
    """
    data_path = os.path.join(os.path.dirname(__file__), 'data')
    files = glob.glob(os.path.join(data_path, f'vtk/*'))
    for f in files:
        os.remove(f)

class ProblemSetup(Elasticity):
    def __init__(self, Nx, Ny, Lx, Ly, num_bounded_cell=2, num_loaded_cell=1, filled_density=1., void_density=0., dim=2, vec=2):
        self.Nx, self.Ny = Nx, Ny
        self.Lx, self.Ly = Lx, Ly
```

```
31          self.num_bounded_cell, self.num_loaded_cell = num_bounded_cell,
            ↪ num_loaded_cell
32          self.filled_density, self.void_density = filled_density,
            ↪ void_density
33          self.vec = vec
34          self.dim = dim
35          self.ele_type = 'QUAD4'
36          self.cell_type = get_meshio_cell_type(self.ele_type)
37          self.meshio_mesh = rectangle_mesh(Nx=Nx, Ny=Ny, domain_x=Lx,
            ↪ domain_y=Ly)
38          self.mesh = Mesh(self.meshio_mesh.points,
            ↪ self.meshio_mesh.cells_dict[self.cell_type])
39          self.cells = self.mesh.cells
40          self.points = self.mesh.points
41          self.cell_inds = np.arange((Nx * Ny), dtype=np.int32)
42          self.cell_inds_matrix =
            ↪ self._state_matrix_from_array(self.cell_inds, self.Nx,
            ↪ self.Ny)
43          self.point_inds_matrix =
            ↪ self._state_matrix_from_array(np.arange(len(self.points)),
            ↪ self.Nx+1, self.Ny+1)
44
45
46      def _state_matrix_from_array(self, state_array: np.ndarray, num_row:
        ↪ int, num_column: int) -> np.ndarray:
47          """
48          Converts given array to matrix form with same topological
    ↪   representation of JAX-FEM format.
49                                                  [2   5   8]
50          e.g. [0, 1, 2, 3, 4, 5, 6, 7, 8] -->    [1   4   7]
51                                                  [0   3   6]
52          Args:
53              state_array (np.ndarray): input array
54              num_row (int)           : number of row of output matrix
55              num_column (int)        : number of column of output matrix
56          Returns:
57              Output matrix in predefined geometry represantation defined
    ↪   by Jax-Fem
58          """
59          return onp.rot90(onp.reshape(state_array, (num_row, num_column)),
            ↪ k=1, axes=(0, 1))
60
61
62      def _state_array_from_matrix(state_matrix: np.ndarray) -> np.ndarray:
63          """
64          Converts a state matrix into state vector.
```

```
65                  [2   5   8]
66          e.g.    [1   4   7] --> [0, 1, 2, 3, 4, 5, 6, 7, 8]
67                  [0   3   6]
68          Args:
69              state_matrix (np.ndarray): input state matrix represents
    ↪  topological indexing format of Jax-FEM
70          Returns:
71              Array representation of given matrix.
72          """
73          return onp.reshape(onp.rot90(state_matrix, k=1, axes=(1, 0)),
            ↪  (-1))
74
75
76      def _catagorize_cells(self):
77          """
78          Categorizes the cells wrt their topological locations.
79          Note: Left-Top-Right-Bottom edge cell indices do not contain
    ↪  corner element indices at these edges
80                  [2   5   8]
81          e.g.    [1   4   7]
82                  [0   3   6]
83          Args:
84          Returns:
85              inner_cell_inds         :   e.g. [4]
86              outer_cell_inds         :   e.g. [0, 1, 2, 3, 5, 6, 7, 8]
87              outer_corner_cell_inds  :   e.g. [0, 6, 8, 2]
88              left_edge_cell_inds     :   e.g. [1]
89              top_edge_cell_inds      :   e.g. [5]
90              right_edge_cell_inds    :   e.g. [7]
91              bottom_edge_cell_inds   :   e.g. [3]
92          """
93          cell_inds_matrix = self._state_matrix_from_array(self.cell_inds,
            ↪  self.Nx, self.Ny)
94          inner_cell_inds = onp.reshape((cell_inds_matrix)[1:self.Nx-1,
            ↪  1:self.Ny-1], -1)
95          outer_cell_inds = onp.delete(self.cell_inds, inner_cell_inds)
96          outer_corner_cell_inds = onp.array((cell_inds_matrix[self.Nx-1,
            ↪  0], cell_inds_matrix[self.Nx-1, self.Ny-1],
            ↪  cell_inds_matrix[0, self.Ny-1], cell_inds_matrix[0, 0]),
            ↪  dtype=int)
97          left_edge_cell_inds = cell_inds_matrix[1:self.Ny-1, 0][::-1]
98          top_edge_cell_inds = cell_inds_matrix[0, 1:self.Nx-1]
99          right_edge_cell_inds = cell_inds_matrix[1:self.Ny-1,
            ↪  self.Nx-1][::-1]
100         bottom_edge_cell_inds = cell_inds_matrix[self.Ny-1, 1:self.Nx-1]
```

```
101         return inner_cell_inds, outer_cell_inds, outer_corner_cell_inds,
            ↪ left_edge_cell_inds, top_edge_cell_inds,
            ↪ right_edge_cell_inds, bottom_edge_cell_inds
102
103
104     def select_bounded_and_loaded_cells(self):
105         """
106         Performs random cell selection to assign boundary conditions on
    ↪ their choses points.
107         Note: Number of cells to be selected for Dirichlet and Neumann
    ↪ boundary conditions are passed as input for the class.
108         Args:
109             self.num_bounded_cell   (int)
110             self.num_loaded_cell    (int)
111         Returns:
112             bounded_cell_inds       (list)
113             loaded_cell_inds        (list)
114         """
115         _ , outer_cell_inds, _, _, _, _, _ = self._catagorize_cells()
116         bounded_cell_inds = onp.random.choice(outer_cell_inds,
            ↪ self.num_bounded_cell, replace=False)
117         cell_inds = onp.delete(self.cell_inds, bounded_cell_inds)
118         loaded_cell_inds = onp.random.choice(cell_inds,
            ↪ self.num_loaded_cell, replace=False)
119         #loaded_cell_inds = [30]
120         return bounded_cell_inds, loaded_cell_inds
121
122     def select_bounded_and_loaded_points(self, bounded_cell_inds:
        ↪ np.ndarray, loaded_cell_inds: np.ndarray) -> list:
123         """
124         Performs point selection to assign boundary conditions for given
    ↪ cells.
125         Note : It contains a part to be update. For now it is
    ↪ implemented such to be work properly in neumann bc application, but
    ↪ deviates from the paper!
126         Args:
127             bounded_cell_inds   (np.ndarray):   Cell indices selected
    ↪ for Dirichlet BC assignment.
128             loaded_cell_inds    (np.ndarray):   Cell indices selected
    ↪ for Neumann BC assignment.
129         Returns:
130             bounded_cell_inds   (list): Selected point indices wrt given
    ↪ cell indices for Dirichlet BC assignment.
131             loaded_cell_inds    (list): Selected point indices wrt given
    ↪ cell indices for Neumann BC assignment.
132         """
```

```
133             inner_cell_inds, outer_cell_inds, corner_cell_inds,
                ↪ left_edge_cell_inds, top_edge_cell_inds,
                ↪ right_edge_cell_inds, bottom_edge_cell_inds=
                ↪ self._catagorize_cells()
134
135             bounded_point_inds = []
136             loaded_point_inds = []
137             for bounded_cell in bounded_cell_inds:
138                 if bounded_cell in corner_cell_inds:
139                     bounded_point =
                        ↪ self.cells[bounded_cell][onp.where(bounded_cell ==
                        ↪ corner_cell_inds)]
140                     bounded_point_inds.append(int(bounded_point))
141                 else:
142                     if bounded_cell in left_edge_cell_inds:
143                         bounded_point1, bounded_point2 =
                            ↪ self.cells[bounded_cell][0],
                            ↪ self.cells[bounded_cell][3]
144                     elif bounded_cell in top_edge_cell_inds:
145                         bounded_point1, bounded_point2 =
                            ↪ self.cells[bounded_cell][2],
                            ↪ self.cells[bounded_cell][3]
146                     elif bounded_cell in right_edge_cell_inds:
147                         bounded_point1, bounded_point2 =
                            ↪ self.cells[bounded_cell][1],
                            ↪ self.cells[bounded_cell][2]
148                     else:
149                         bounded_point1, bounded_point2 =
                            ↪ self.cells[bounded_cell][0],
                            ↪ self.cells[bounded_cell][1]
150                     bounded_point_inds.append(bounded_point1)
151                     bounded_point_inds.append(bounded_point2)
152             for loaded_cell in loaded_cell_inds:
153                 if loaded_cell in corner_cell_inds:
154                     # loaded_point =
                        ↪ cells[loaded_cell][onp.where(loaded_cell ==
                        ↪ outer_corner_cell_inds)]
155                     # loaded_points.append(loaded_point)
156                     #index = onp.random.randint(0, 4)
157                     index = 0
158                     loaded_point1, loaded_point2 =
                        ↪ self.cells[loaded_cell][index],
                        ↪ self.cells[loaded_cell][(index+1)%4]
159                     loaded_point_inds.append(loaded_point1)
160                     loaded_point_inds.append(loaded_point2)
161                 else:
```

```
162                 if loaded_cell in bottom_edge_cell_inds:
163                     loaded_point1, loaded_point2 =
                        ↪ self.cells[loaded_cell][0],
                        ↪ self.cells[loaded_cell][1]
164                 elif loaded_cell in left_edge_cell_inds:
165                     loaded_point1, loaded_point2 =
                        ↪ self.cells[loaded_cell][0],
                        ↪ self.cells[loaded_cell][3]
166                 elif loaded_cell in top_edge_cell_inds:
167                     loaded_point1, loaded_point2 =
                        ↪ self.cells[loaded_cell][2],
                        ↪ self.cells[loaded_cell][3]
168                 elif loaded_cell in right_edge_cell_inds:
169                     loaded_point1, loaded_point2 =
                        ↪ self.cells[loaded_cell][1],
                        ↪ self.cells[loaded_cell][2]
170                 else:
171                     #index = onp.random.randint(0, 4)
172                     index = 0
173                     loaded_point1, loaded_point2 =
                        ↪ self.cells[loaded_cell][index],
                        ↪ self.cells[loaded_cell][(index+1)%4]
174                 loaded_point_inds.append(loaded_point1)
175                 loaded_point_inds.append(loaded_point2)
176         return sorted([*set(bounded_point_inds)]),
            ↪ sorted([*set(loaded_point_inds)])    # FIX IN THE FUTURE (for
            ↪ now if the loaded_points has 1 element, causes error!)
177
178     def _cell_point_relation_check(self):
179         """
180         Easy check for confirmation of selected cell and selected point
    ↪ relations.
181         Returns:
182             Prints points inds and its index for given cell, otherwise
    ↪ returns a warning message.
183         """
184         pass
185
186     def set_dirichlet_bc(self, selected_points: list) -> list:
187         """
188         Creates required Dirichlet boundary input for Jax-FEM solver for
    ↪ given points.
189         Note : It assigns 0 displacement to given points in 2
    ↪ direction.
190         Note : This method includes hardcoding and right now work for
    ↪ len(selected_points) = 2 or 3 or 4 cases.
```

```
191         Args:
192             selected_points (list): Selected points for Dirichlet BC
    ↪ assignment
193         Returns:
194             Required list for JAX-FEM solver contains fix point
    ↪ locations, vectors (in which directions the displacement should be
    ↪ applied), value list (displacement value)
195         """
196         fix_location_list = []
197         vector_list = []
198         dirichlet_value_list = []
199         if len(selected_points) == 2:
200             fix_location1 = lambda point:
                ↪ np.logical_and(np.isclose(point[0],
                ↪ self.points[selected_points[0]][0]), np.isclose(point[1],
                ↪ self.points[selected_points[0]][1]))
201             fix_location2 = lambda point:
                ↪ np.logical_and(np.isclose(point[0],
                ↪ self.points[selected_points[1]][0]), np.isclose(point[1],
                ↪ self.points[selected_points[1]][1]))
202             vector_list = [0, 1, 0, 1]
203             dirichlet_value = lambda point: 0.
204             fix_location_list = [fix_location1, fix_location1,
                ↪ fix_location2, fix_location2]
205             dirichlet_value_list = [dirichlet_value, dirichlet_value,
                ↪ dirichlet_value, dirichlet_value]
206         if len(selected_points) == 3:
207             fix_location1 = lambda point:
                ↪ np.logical_and(np.isclose(point[0],
                ↪ self.points[selected_points[0]][0]), np.isclose(point[1],
                ↪ self.points[selected_points[0]][1]))
208             fix_location2 = lambda point:
                ↪ np.logical_and(np.isclose(point[0],
                ↪ self.points[selected_points[1]][0]), np.isclose(point[1],
                ↪ self.points[selected_points[1]][1]))
209             fix_location3 = lambda point:
                ↪ np.logical_and(np.isclose(point[0],
                ↪ self.points[selected_points[2]][0]), np.isclose(point[1],
                ↪ self.points[selected_points[2]][1]))
210             vector_list = [0, 1, 0, 1, 0, 1]
211             dirichlet_value = lambda point: 0.
212             fix_location_list = [fix_location1, fix_location1,
                ↪ fix_location2, fix_location2, fix_location3,
                ↪ fix_location3]
```

```
213             dirichlet_value_list = [dirichlet_value, dirichlet_value,
                ↪  dirichlet_value, dirichlet_value, dirichlet_value,
                ↪  dirichlet_value]
214         if len(selected_points) == 4:
215             fix_location1 = lambda point:
                ↪  np.logical_and(np.isclose(point[0],
                ↪  self.points[selected_points[0]][0]), np.isclose(point[1],
                ↪  self.points[selected_points[0]][1]))
216             fix_location2 = lambda point:
                ↪  np.logical_and(np.isclose(point[0],
                ↪  self.points[selected_points[1]][0]), np.isclose(point[1],
                ↪  self.points[selected_points[1]][1]))
217             fix_location3 = lambda point:
                ↪  np.logical_and(np.isclose(point[0],
                ↪  self.points[selected_points[2]][0]), np.isclose(point[1],
                ↪  self.points[selected_points[2]][1]))
218             fix_location4 = lambda point:
                ↪  np.logical_and(np.isclose(point[0],
                ↪  self.points[selected_points[3]][0]), np.isclose(point[1],
                ↪  self.points[selected_points[3]][1]))
219             vector_list = [0, 1, 0, 1, 0, 1, 0, 1]
220             dirichlet_value = lambda point: 0.
221             fix_location_list = [fix_location1, fix_location1,
                ↪  fix_location2, fix_location2, fix_location3,
                ↪  fix_location3, fix_location4, fix_location4]
222             dirichlet_value_list = [dirichlet_value, dirichlet_value,
                ↪  dirichlet_value, dirichlet_value, dirichlet_value,
                ↪  dirichlet_value, dirichlet_value, dirichlet_value]
223         return [fix_location_list, vector_list, dirichlet_value_list]
224
225     def set_neumann_bc(self, selected_points: list) -> list:
226         """
227         Creates required Neumann boundary input for Jax-FEM solver for
    ↪  given points.
228         Args:
229             selected_points (list): Selected points for Neumann BC
    ↪  assignment
230         Returns:
231             Required list for JAX-FEM solver contains load point
    ↪  locations, force values in each axis assigned in random directions
232         """
233         load_location_list = []
234         neumann_val_list = []
235         load_location = lambda point: np.logical_and(
```

```
236             np.isclose(point[0], (self.points[selected_points[0]][0] +
                ↪  self.points[selected_points[1]][0])/2, atol= 1e-5 +
                ↪  onp.abs(self.points[selected_points[0]][0] -
                ↪  self.points[selected_points[1]][0])),
237             np.isclose(point[1], (self.points[selected_points[0]][1] +
                ↪  self.points[selected_points[1]][1])/2, atol= 1e-5 +
                ↪  onp.abs(self.points[selected_points[0]][1] -
                ↪  self.points[selected_points[1]][1])))
238         #neumann_val = lambda point: np.array([100., 100.]) *
            ↪  onp.random.choice([1, -1], 2)
239         neumann_val = lambda point: np.array([0, -0.1])
240         load_location_list.append(load_location)
241         neumann_val_list.append(neumann_val)
242         return [load_location_list, neumann_val_list]
243
244     def problem_define(self, dirichlet_bc_info: list, neumann_bc_info:
        ↪  list):
245         """
246         Creates an Elasticity instance by passing required inputs.
247         """
248         return Elasticity(mesh=self.mesh, vec=self.vec, dim=self.dim,
            ↪  ele_type=self.ele_type, dirichlet_bc_info=dirichlet_bc_info,
249                           neumann_bc_info=neumann_bc_info,
                              ↪  additional_info=('box',))
250
251     def problem_solve(self, problem, rho: np.ndarray):
252         """
253         Advances one step the given problem instance through solver by
    ↪   taking rho as design input and returns the solution.
254         """
255         fwd_pred = ad_wrapper(problem, linear=True, use_petsc=True)
256         rho = rho.reshape((-1,1))
257         return fwd_pred(rho)
258
259     def create_state_space_tensor(self, rho_vector: np.ndarray,
        ↪  von_mises: np.ndarray, bounded_cell_inds:np.ndarray,
        ↪  loaded_cell_inds: np.ndarray) -> np.ndarray:
260         """
261         Creates required DQN input 3 x N x N  state tensor
262         Args:
263             rho_vector (np.ndarray)          : density vector
264             von_mises (np.ndarray)           : von mises vector
265             bounded_cell_inds (np.ndarray)   :
266             loaded_cell_inds (np.ndarray)    :
267         Returns:
```

```
268             state_tensor_DQN                : NxNx3 tensor which will be
    ↪ used in DQN training
269             state_tensor_check              : 3xNxN tensor used for
    ↪ illegality check and visualization
270         """
271
272         inverse_von_mises_array = np.zeros_like(von_mises)
273         max_VM = np.max(von_mises)
274         print(f'max_VM here :{max_VM} at {np.argmax(von_mises)}')
275         for i in range(len(von_mises)):
276             if rho_vector[i] > self.void_density:
277                 inverse_von_mises_array =
                    ↪ inverse_von_mises_array.at[i].set(max_VM/von_mises[i])
278
279         inverse_von_mises_array =
            ↪ inverse_von_mises_array.at[inverse_von_mises_array>1e3].set(0.)
280         inverse_von_mises_array =
            ↪ inverse_von_mises_array/np.max(inverse_von_mises_array)
281         bounded_cells_state_array =
            ↪ self.cell_inds[onp.where((self.cell_inds ==
            ↪ bounded_cell_inds[0]) | (self.cell_inds ==
            ↪ bounded_cell_inds[1]), 1, 0)]  # hard_coded to return 2
            ↪ cells
282         loaded_cells_state_array =
            ↪ self.cell_inds[onp.where((self.cell_inds ==
            ↪ loaded_cell_inds), 1, 0)] # hard_coded to return 1 cell
283         inverse_von_mises_matrix =
            ↪ self._state_matrix_from_array(inverse_von_mises_array,
            ↪ self.Nx, self.Ny)
284         bounded_cells_state_matrix =
            ↪ self._state_matrix_from_array(bounded_cells_state_array,
            ↪ self.Nx, self.Ny)
285         loaded_cells_state_matrix =
            ↪ self._state_matrix_from_array(loaded_cells_state_array,
            ↪ self.Nx, self.Ny)
286         state_tensor_DQN = np.stack((inverse_von_mises_matrix,
            ↪ bounded_cells_state_matrix, loaded_cells_state_matrix),
            ↪ axis=2)
287         state_tensor_check = np.stack((inverse_von_mises_matrix,
            ↪ bounded_cells_state_matrix, loaded_cells_state_matrix),
            ↪ axis=0)
288         return state_tensor_DQN, state_tensor_check
289
290     def check_illegal(self, rho_matrix: np.ndarray, new_point: int,
        ↪ state_tensor: np.ndarray, nb_step: int, nb_max_step: int) ->
        ↪ bool:
```

```
291         """
292         Checks whether the selected point can be removed
293         Args:
294             rho (np.ndarray)                    : The boolean mask of
    ↪ the topology (shape Nx x Ny)
295             new_point (int)                     : The index of the cell
    ↪ to be removed
296             state_matrix (np.ndarray)           : The state matrix that
    ↪ contains inv_von_mises, bounded_cells, and loaded_cells arrays
    ↪ (shape : 3 x Nx x Ny)
297             self.cell_inds_matrix (np.ndarray)  : The matrix contains
    ↪ cell indices in the order that represents geometry (x = 0 is at the
    ↪ LEFT, y = 0 is at the BOTTOM) (shape: Nx x Ny)
298             self.filled_denstiy (float)         : The material intensity
    ↪ value for filled cells
299             self.void_density (float)           : The material intensity
    ↪ value for void cells
300 
301         Returns:
302             True if the cell can be removed and False otherwise
303         """
304         _, bounds, forces = state_tensor
305         Nx, Ny = rho_matrix.shape
306         new_point_inds = onp.argwhere(self.cell_inds_matrix ==
            ↪ new_point)[0]
307         x, y = new_point_inds
308 
309         # (A) If the to-be-removed point has the coordinates of a
            ↪ boundary condition or of a force origin
310 
311         if bounds[x, y]==self.filled_density or forces[x,
            ↪ y]==self.filled_density:
312             print("\nIllegality check --> False")
313             print(f"You are trying to remove bounded or loaded cell
                ↪ number {new_point}.")
314             return True
315 
316         # (B) If the to-be-removed point has already been removed
317 
318         if rho_matrix[x, y] == self.void_density:
319             print("\nIllegality check --> False")
320             print(f"You are trying to remove already removed cell number
                ↪ {new_point}.")
321             return True
322 
323         # (C) Making sure for only one connected component
```

```
324             new_rho_matrix = onp.floor(rho_matrix)
325             new_rho_matrix[x, y] = 0
326             labeled, ncomponents = label(new_rho_matrix)
327             if ncomponents > 1:
328                 print("\nIllegality check --> False")
329                 print(f"More than one component by removing cell number
                    ↪ {new_point}.")
330                 print(labeled)
331                 return True
332 
333             if nb_step > nb_max_step:
334                 return True
335 
336             # If everything complies to the rules:
337             return False
338 
339         def test_check_illegal(self, state_tensor: np.ndarray,
            ↪ pre_created_rho: np.ndarray=None,
            ↪ pre_selected_cell_to_be_removed: int=None):
340             """
341             Created for testing check_illegal function using predefined or
        ↪ random scenarios.
342             """
343             rho_matrix = onp.where(onp.random.randint(2, size=(self.Nx,
                ↪ self.Ny))==0, self.void_density, self.filled_density)
344             cell_to_be_removed =
                ↪ (self.cell_inds_matrix[onp.random.choice(onp.arange(self.Nx)),
                ↪ onp.random.choice(onp.arange(self.Ny))])
345             if pre_created_rho is not None:
346                 rho_matrix = pre_created_rho
347             if pre_selected_cell_to_be_removed is not None:
348                 cell_to_be_removed = pre_selected_cell_to_be_removed
349             check = self.check_illegal(rho_matrix, cell_to_be_removed,
                ↪ state_tensor)
350             if check:
351                 print("LEGAL ACTION!!!")
352             else:
353                 print()
354                 print("ILLEGAL ACTION!!!")
355                 print(f"Selected cell '{cell_to_be_removed}' can not be
                    ↪ removed.")
356                 print(f"Cell_indices_matrix = \n{self.cell_inds_matrix}")
357                 print(f"Rho matrix = \n{rho_matrix}")
358                 print(f"Bounded cell matrix = \n{state_tensor[1]}")
359                 print(f"Loaded cell matrix = \n{state_tensor[2]}")
360 
```

```
361
362     def update_density(self, rho_vector: np.ndarray, cell_index: int) ->
        ↪  np.ndarray:
363         """
364         Updates selected index of density vector with
    ↪   self.void_density.
365         Args:
366             rho_vector (np.ndarray)             : Density vector that
    ↪   contains denstiy values for each cell
367             new_point (int)                     : The index of the cell
    ↪   to be removed
368         Returns:
369             Updated rho vector and its state represantation formatted
    ↪   rho matrix
370         """
371         rho_vector[cell_index] = self.void_density
372         rho_matrix = self._state_matrix_from_array(rho_vector, self.Nx,
            ↪  self.Ny)
373         return rho_vector, rho_matrix
374
375
376     def positive_reward(self, strain_energy_initial: float,
        ↪  strain_energy_current: float, num_of_voided_cells: int,
        ↪  num_of_total_cells: int) -> float:
377         """
378         Updates selected index of density vector with
    ↪   self.void_density.
379         Args:
380             init_von_mises (float)          : Von mises stresses at
    ↪   initial state
381             current_von_mises (float)       : Von mises stresses at the
    ↪   current state
382             num_of_voided_cells (int)       : Number of voided cell
    ↪   including the current state
383             num_of_total_cells (int)        : Number of total cell in
    ↪   the topology
384         Returns:
385             Positive reward value after each successful action
386         """
387
388         return ( strain_energy_initial / strain_energy_current) ** 2 +
            ↪  (num_of_voided_cells / num_of_total_cells) ** 2
```

A.2 Code of fem_model.py

```
import numpy as onp
import jax
import jax.numpy as np

from jax_am.fem.core import FEM


class Elasticity(FEM):
    def custom_init(self, case_flag):
        self.cell_centroids = onp.mean(onp.take(self.points, self.cells, axis=0), axis=1)
        self.flex_inds = np.arange(len(self.cells))
        self.case_flag = case_flag
        if case_flag == 'freecad':
            self.get_tensor_map = self.get_tensor_map_freecad
        elif case_flag == 'box':
            self.get_tensor_map = self.get_tensor_map_box
        elif case_flag == 'multi_material':
            self.get_tensor_map = self.get_tensor_map_multi_material
        elif case_flag == 'plate' or case_flag == 'L_shape' or case_flag == 'eigen':
            self.get_tensor_map = self.get_tensor_map_plane_stress
            if case_flag == 'eigen':
                self.penal = 5.
            else:
                self.penal = 3.
        else:
            raise ValueError(f"Unknown case_flag = {case_flag}")

    def get_tensor_map_plane_stress(self):
        def stress(u_grad, theta):
            # Reference:
            # https://engcourses-uofa.ca/books/introduction-to-solid-mechanics/
            # constitutive-laws/linear-elastic-materials/plane-isotropic-linear-el
            Emax = 70.e9
            Emin = 1e-3*Emax
            nu = 0.3

            penal = self.penal

            E = Emin + (Emax - Emin)*theta[0]**penal
            epsilon = 0.5*(u_grad + u_grad.T)

```

```
41              eps11 = epsilon[0, 0]
42              eps22 = epsilon[1, 1]
43              eps12 = epsilon[0, 1]
44
45              sig11 = E/(1 + nu)/(1 - nu)*(eps11 + nu*eps22)
46              sig22 = E/(1 + nu)/(1 - nu)*(nu*eps11 + eps22)
47              sig12 = E/(1 + nu)*eps12
48
49              sigma = np.array([[sig11, sig12], [sig12, sig22]])
50              return sigma
51          return stress
52
53      def get_tensor_map_freecad(self):
54          # Unit is not in SI, used for freecad example
55          def stress(u_grad, theta):
56              Emax = 70.e3
57              Emin = 70.
58              nu = 0.3
59              penal = 3.
60              E = Emin + (Emax - Emin)*theta[0]**penal
61              mu = E/(2.*(1. + nu))
62              lmbda = E*nu/((1+nu)*(1-2*nu))
63              epsilon = 0.5*(u_grad + u_grad.T)
64              sigma = lmbda*np.trace(epsilon)*np.eye(self.dim) +
                ↪  2*mu*epsilon
65              return sigma
66          return stress
67
68      def get_tensor_map_box(self):
69          def stress(u_grad, theta):
70              nu = 0.3
71              E = theta[0]
72              mu = E/(2.*(1. + nu))
73              lmbda = E*nu/((1+nu)*(1-2*nu))
74              epsilon = 0.5*(u_grad + u_grad.T)
75              sigma = lmbda*np.trace(epsilon)*np.eye(self.dim) +
                ↪  2*mu*epsilon
76              return sigma
77          return stress
78
79      def get_tensor_map_multi_material(self):
80          def stress(u_grad, theta):
81              Emax = 70.e3
82              Emin = 70.
83              nu = 0.3
84              penal = 3.
```

```
85
86              E1 = Emax
87              E2 = 0.2*Emax
88
89              theta1, theta2 = theta
90              E = Emin + theta1**penal*(theta2**penal*E1 + (1 -
                ↪  theta2**penal)*E2)
91
92              mu = E/(2.*(1. + nu))
93              lmbda = E*nu/((1+nu)*(1-2*nu))
94              epsilon = 0.5*(u_grad + u_grad.T)
95              sigma = lmbda*np.trace(epsilon)*np.eye(self.dim) +
                ↪  2*mu*epsilon
96              return sigma
97          return stress
98
99      def set_params(self, params):
100         full_params = np.ones((self.num_cells, params.shape[1]))
101         full_params = full_params.at[self.flex_inds].set(params)
102         thetas = np.repeat(full_params[:, None, :], self.num_quads,
            ↪  axis=1)
103         self.full_params = full_params
104         self.internal_vars['laplace'] = [thetas]
105
106     def compute_compliance(self, neumann_fn, sol):
107         boundary_inds = self.neumann_boundary_inds_list[0]
108         _, nanson_scale = self.get_face_shape_grads(boundary_inds)
109         # (num_selected_faces, 1, num_nodes, vec) * #
            ↪  (num_selected_faces, num_face_quads, num_nodes, 1)
110         u_face = sol[self.cells][boundary_inds[:, 0]][:, None, :, :] *
            ↪  self.face_shape_vals[boundary_inds[:, 1]][:, :, :, None]
111         u_face = np.sum(u_face, axis=2) # (num_selected_faces,
            ↪  num_face_quads, vec)
112         # (num_cells, num_faces, num_face_quads, dim) ->
            ↪  (num_selected_faces, num_face_quads, dim)
113         subset_quad_points =
            ↪  self.get_physical_surface_quad_points(boundary_inds)
114         traction = jax.vmap(jax.vmap(neumann_fn))(subset_quad_points) #
            ↪  (num_selected_faces, num_face_quads, vec)
115         val = np.sum(traction * u_face * nanson_scale[:, :, None])
116         return val
117
118     def get_von_mises_stress_fn(self):
119         def stress_fn(u_grad, theta):
120             Emax = 70.e9
121             nu = 0.3
```

```
122             penal = 0.5
123             E = theta[0]**penal*Emax
124             mu = E/(2.*(1. + nu))
125             lmbda = E*nu/((1+nu)*(1-2*nu))
126             epsilon = 0.5*(u_grad + u_grad.T)
127             sigma = lmbda*np.trace(epsilon)*np.eye(self.dim) +
                ↪  2*mu*epsilon
128             return sigma
129
130         def vm_stress_fn_helper(sigma):
131             dim = self.dim
132             s_dev = sigma - 1./dim*np.trace(sigma)*np.eye(dim)
133             vm_s = np.sqrt(3./2.*np.sum(s_dev*s_dev))
134             return vm_s
135
136         if self.case_flag == 'plate' or self.case_flag == 'L_shape':
137             def vm_stress_fn(u_grad, theta):
138                 sigma2d = stress_fn(u_grad, theta)
139                 sigma3d = np.array([[sigma2d[0, 0], sigma2d[0, 1], 0.],
                    ↪  [sigma2d[1, 0], sigma2d[1, 1], 0.], [0., 0., 0.]])
140                 return vm_stress_fn_helper(sigma3d)
141         else:
142             def vm_stress_fn(u_grad, theta):
143                 sigma = self.get_tensor_map()(u_grad, theta)
144                 return vm_stress_fn_helper(sigma)
145
146         return vm_stress_fn
147
148     def compute_von_mises_stress(self, sol):
149         # (num_cells, 1, num_nodes, vec, 1) * (num_cells, num_quads,
            ↪  num_nodes, 1, dim) -> (num_cells, num_quads, num_nodes, vec,
            ↪  dim)
150         u_grads = np.take(sol, self.cells, axis=0)[:, None, :, :, None] *
            ↪  self.shape_grads[:, :, :, None, :]
151         u_grads = np.sum(u_grads, axis=2) # (num_cells, num_quads, vec,
            ↪  dim)
152         vm_stress_fn = self.get_von_mises_stress_fn()
153         vm_stress = jax.vmap(jax.vmap(vm_stress_fn))(u_grads,
            ↪  *self.internal_vars['laplace']) # (num_cells, num_quads)
154         volume_avg_vm_stress = np.sum(vm_stress * self.JxW, axis=1) /
            ↪  np.sum(self.JxW, axis=1) # (num_cells,)
155         return volume_avg_vm_stress
156
157     def compute_4_points_polygon_area(self, A, B, C, D):
158         '''
159         D---C
```

```
160         |   |
161         A---B
162         '''
163         x_A, y_A = A
164         x_B, y_B = B
165         x_C, y_C = C
166         x_D, y_D = D
167 
168         xs = [x_A, x_B, x_C, x_D]
169         ys = [y_A, y_B, y_C, y_D]
170 
171         area = 0.0
172         for i in range(4):
173             area += xs[i] * ys[(i + 1)%4] - xs[(i + 1)%4] * ys[i]
174         area *= .5
175 
176         return abs(area)
177 
178     def compute_4_points_polygon_centroid(self, A, B, C, D):
179         '''
180         D---C
181         |   |
182         A---B
183         '''
184         x_A, y_A = A
185         x_B, y_B = B
186         x_C, y_C = C
187         x_D, y_D = D
188 
189         xs = [x_A, x_B, x_C, x_D]
190         ys = [y_A, y_B, y_C, y_D]
191 
192         abs_area = self.compute_4_points_polygon_area(A, B, C, D)
193 
194         c_x = 0.0
195         c_y = 0.0
196 
197         for i in range(4):
198             c_x += (xs[i] + xs[(i + 1)%4]) * (xs[i] * ys[(i + 1)%4] -
                ↪  xs[(i + 1)%4] * ys[i])
199         c_x /= 6 * abs_area
200 
201         for i in range(4):
202             c_y += (ys[i] + ys[(i + 1)%4]) * (xs[i] * ys[(i + 1)%4] -
                ↪  xs[(i + 1)%4] * ys[i])
203         c_y /= 6 * abs_area
```

```
204
205         c = [c_x, c_y]
206
207         return c
```

A.3 Code of env.py

```
1  import gym
2  import numpy as onp
3  import jax.numpy as np
4
5  from colorama import init, Fore, Back, Style
6  from problem import ProblemSetup
7
8  class TopOptEnv(gym.Env):
9
10     metadata = {"render_modes": ["human", "rgb_array"], "render_fps": 4}
11
12     def __init__(self, size_x:int = 6, size_y:int = 6, render_mode=None,
       ↪  jax_model=None):
13         # Dimensionality of the grid
14         self.size_x, self.size_y = size_x, size_y
15         self.window_size = 512
16         self.initial_rho_vector = onp.ones((self.size_x * self.size_y,
           ↪  1))
17         self.jax_model = jax_model
18         self.points = self.jax_model.points
19         self.cells = self.jax_model.cells
20
21         # Our 3-dimensional array that stores the strain, boundaries
           ↪  points and force-load points
22         self.observation_space = gym.spaces.Dict(
23             {
24                 "strains": gym.spaces.Box(low=0.0, high=1.,
                   ↪  shape=(size_x,size_y), dtype=onp.float32),
25                 "boundary": gym.spaces.Box(low=0, high=1,
                   ↪  shape=(size_x,size_y), dtype=int),
26                 "forces": gym.spaces.Box(low=0, high=1,
                   ↪  shape=(size_x,size_y), dtype=int),
27             }
28         )
29
30         self.action_space = gym.spaces.Discrete(size_x * size_y)
31         self._render_image = onp.ones((size_x, size_y))
32
```

```
33      def _coloring(self):
34          add_bounded_mask = onp.zeros((self.size_x, self.size_y))
35          for bounded in self.bounded_cells:
36              add_bounded_mask += onp.where(self.jax_model.cell_inds_matrix
                ↪  == bounded, 1, 0)
37
38          add_loaded_mask = onp.zeros((self.size_x, self.size_y))
39          for loaded in self.loaded_cells:
40              add_loaded_mask += onp.where(self.jax_model.cell_inds_matrix
                ↪  == loaded, 2, 0)
41
42          self._render_image = onp.ones((self.size_x, self.size_y)) +
            ↪  add_bounded_mask + add_loaded_mask
43
44
45      def _remove_cell_color(self, x, y):
46          self._render_image[x, y] = 0
47
48      def _get_obs(self):
49          self._strains, self._bounds, self._forces =
            ↪  self.state_tensor_check[0,:,:],
            ↪  self.state_tensor_check[1,:,:],
            ↪  self.state_tensor_check[2,:,:]
50          return {"strains": self._strains,
51                  "boundary": self._bounds,
52                  "forces": self._forces}
53
54      def _get_info(self):
55          return self.rho_matrix
56
57      def reset(self, seed=123, options=123):
58          super().reset(seed=seed)
59
60          self.bounded_cells, self.loaded_cells =
            ↪  self.jax_model.select_bounded_and_loaded_cells()
61          self.max_num_step = len(self.cells) - (len(self.bounded_cells) +
            ↪  len(self.loaded_cells))
62
63
64          self.bounded_cells = [0,5]
65          self.loaded_cells = np.array([30])
            ↪  #[onp.random.choice([30:35],size=1)]
66          self.bounded_points, self.loaded_points =
            ↪  self.jax_model.select_bounded_and_loaded_points(self.bounded_cells,
            ↪  self.loaded_cells)
67
```

```
68          self.dirichlet_bc =
            ↪ self.jax_model.set_dirichlet_bc(self.bounded_points)
69
70          self.neumann_bc =
            ↪ self.jax_model.set_neumann_bc(self.loaded_points)
71          self.problem = self.jax_model.problem_define(self.dirichlet_bc,
            ↪ self.neumann_bc)
72
73          self.rho_vector = onp.copy(self.initial_rho_vector)
74          self.rho_matrix =
            ↪ self.jax_model._state_matrix_from_array(self.rho_vector,
            ↪ self.size_x, self.size_y)
75          self.solution = self.jax_model.problem_solve(self.problem,
            ↪ self.rho_vector)
76
77          self.init_SE = 0
78          for elem_nb in range(self.size_x*self.size_y):
79                  elem_node_1 = int(elem_nb + elem_nb/self.size_x)
80                  elem_nodes = [elem_node_1, elem_node_1 + 1, elem_node_1 +
                    ↪ self.size_x+1, elem_node_1 + self.size_x]
81                  self.init_SE +=
                    ↪ (self.solution[elem_nodes,:].reshape((-1,1)).T *
                    ↪ self.rho_vector[elem_nb]) @
                    ↪ self.solution[elem_nodes,:].reshape((-1,1))
82          self.init_SE = onp.float(self.init_SE)
83          print(f'Initial Strain Energy : {self.init_SE}')
84          self.initial_von_mises =
            ↪ self.problem.compute_von_mises_stress(self.solution)
85          self.state_tensor_DQN, self.state_tensor_check =
            ↪ self.jax_model.create_state_space_tensor(self.rho_vector,
            ↪ self.initial_von_mises, self.bounded_cells,
            ↪ self.loaded_cells)
86
87          print(f'Check the tensor : {self.state_tensor_check}')
88          # List with the cells that we have already removed
89          self.removed_cells = []
90
91          self.current_state_tensor_DQN, self.current_state_tensor_check =
            ↪ self.state_tensor_DQN, self.state_tensor_check
92
93          self.nb_removed_cells = 0
94          self._coloring()
95
96          observation = self._get_obs()
97          info = self._get_info()
98
```

```
99          self.special_print(0)
100         return self.current_state_tensor_DQN
101
102
103     def step(self, action):
104
105         ## Action reperesents the cell number to remove from topology
106         cell_to_be_removed = action
107         reward = 0
108         self.next_state_tensor_DQN = None
109         if self.jax_model.check_illegal(self.rho_matrix,
            ↪  cell_to_be_removed, self.current_state_tensor_check,
            ↪  self.nb_removed_cells, self.max_num_step):
110             reward = -1
111             terminated = True
112             indices =
                ↪  onp.argwhere(self.jax_model.cell_inds_matrix==cell_to_be_removed)
113             index_x, index_y = indices[0][0], indices[0][1]
114             self._remove_cell_color(index_x, index_y)
115             self.special_print(f"{self.nb_removed_cells + 1} -->
                ↪  illegal")
116         else:
117             terminated = False
118             if self.nb_removed_cells > 1:
119                 self.current_state_tensor = self.next_state_tensor_DQN
120
121             self.nb_removed_cells += 1
122             rho_vector, rho_matrix =
                ↪  self.jax_model.update_density(self.rho_vector,
                ↪  cell_to_be_removed)
123             self.rho1d = rho_vector.reshape((-1,1))
124             solution = self.jax_model.problem_solve(self.problem,
                ↪  rho_vector)
125             von_mises = self.problem.compute_von_mises_stress(solution)
126
127             self.curr_SE = 0
128             for elem_nb in range(self.size_x * self.size_y):
129                 elem_node_1 = int(elem_nb + elem_nb/self.size_x)
130                 elem_nodes = [elem_node_1, elem_node_1 + 1, elem_node_1 +
                    ↪  self.size_x+1, elem_node_1 + self.size_x]
131                 self.curr_SE +=
                    ↪  (solution[elem_nodes,:].reshape((-1,1)).T) @
                    ↪  solution[elem_nodes,:].reshape((-1,1))
132             self.curr_SE = onp.float(self.curr_SE)
133             print()
134             print(f'Current Strain Energy: {self.curr_SE}')
```

```
135             print()
136 
137             reward = self.jax_model.positive_reward(self.init_SE,
                ↪  self.curr_SE, self.nb_removed_cells,
                ↪  self.size_x*self.size_y)
138             self.next_state_tensor_DQN, self.next_state_tensor_check=
                ↪  self.jax_model.create_state_space_tensor(rho_vector,
                ↪  von_mises, self.bounded_cells, self.loaded_cells)
139 
140             print(f'Check the current tensor :
                ↪  {self.next_state_tensor_check}')
141             indices =
                ↪  onp.argwhere(self.jax_model.cell_inds_matrix==cell_to_be_removed)
142             index_x, index_y = indices[0][0], indices[0][1]
143             self._remove_cell_color(index_x, index_y)
144             self.special_print(self.nb_removed_cells)
145             self.removed_cells.append(cell_to_be_removed)
146 
147         return self.current_state_tensor_DQN, action, reward,
            ↪  self.next_state_tensor_DQN, terminated
148 
149 
150     def special_print(self, counter):
151         def aux(value: int):
152             if value < 10:
153                 return (f"0{value}")
154             else:
155                 return (f"{value}")
156         index = -1
157         print(f"Step: {counter}")
158         for i in range(self.size_y):
159             for j in range(self.size_x):
160                 index += 1
161                 if self._render_image[i][j] == 0:
162                     print(Style.BRIGHT + Back.WHITE + Fore.RED +
                        ↪  f"|{aux((self.size_y-i-1) + (self.size_x*j))}|",
                        ↪  end="") # White background
163                 elif self._render_image[i][j] == 1:
164                     print(Style.BRIGHT + Back.BLUE + Fore.RED +
                        ↪  f"|{aux((self.size_y-i-1) + (self.size_x*j))}|",
                        ↪  end="") # Blue background
165                 elif self._render_image[i][j] == 2:
166                     print(Style.BRIGHT + Back.RED + Fore.RED +
                        ↪  f"|{aux((self.size_y-i-1) + (self.size_x*j))}|",
                        ↪  end="") # Red background
167                 elif self._render_image[i][j] == 3:
```

```
168                     print(Style.BRIGHT + Back.GREEN + Fore.RED +
                        ↪  f"|{aux((self.size_y-i-1) + (self.size_x*j))}|",
                        ↪  end="") # Green background
169                 else:
170                     print(Style.BRIGHT + Back.MAGENTA + Fore.RED +
                        ↪  f"|{aux((self.size_y-i-1) + (self.size_x*j))}|",
                        ↪  end="") # Magenta background
171             print()
172         print()
```

A.4 Code of optimizer.py

```
1  from typing import Type
2  import numpy as onp
3
4  import jax
5  import jax.numpy as jnp
6
7  from colorama import init, Fore, Back, Style
8
9  from problem import ProblemSetup
10
11 from env_withgif import TopOptEnv
12 import tensorflow as tf
13 import random
14 import gym
15 import numpy as np
16 from collections import deque
17 from keras.models import Sequential
18 from keras.layers import Dense, Conv2D, Flatten, Layer
19 from keras.optimizers import Adam
20 import os
21
22 from tensorflow.python.ops.numpy_ops import np_config
23 np_config.enable_numpy_behavior()
24
25 # constant decleration for problem setup
26 Nx, Ny = 6, 6
27 Lx, Ly = 6, 6
28 num_bounded_cell = 2
29 num_loaded_cell = 1
30 filled_density = 1.
31 void_density = 1e-4
32 dim = 2
33 vec = 2
```

```
# design variable initialization
num_of_cells = Nx * Ny
vf = 1
init_rho_vector = vf*onp.ones((num_of_cells, 1))
# optimization paramaters decleration
num_episodes = 10
num_steps = num_of_cells - (num_bounded_cell + num_loaded_cell)

simulator = ProblemSetup(Nx=Nx, Ny=Ny, Lx=Lx, Ly=Ly, num_bounded_cell=num_bounded_cell, num_loaded_cell=num_loaded_cell,
                         filled_density=filled_density, void_density=void_density, dim=dim, vec=vec)


env = TopOptEnv(size_x=Nx, size_y=Ny, render_mode="human", jax_model=simulator)
init(autoreset=True)

state_size = (6,6,3)
action_size = 36
batch_size = 128
output_dir = 'model_output/'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

avg_q_values = []
class StateTransformationLayer(Layer):
    def __init__(self):
        super().__init__()

    def call(self, inputs):
        inputs = tf.array(inputs[-1, :, :, -1])
        return tf.reshape(tf.rot90(inputs, k=1, axes=(1, 0)), (-1))

class _build_model(tf.keras.Model):

    def __init__(self):
        super().__init__()
        self.conv = Sequential()
        self.conv.add(Conv2D(16,(3,3),padding='same' ,activation='relu', input_shape=state_size))
        self.conv.add(Conv2D(8,(3,3),padding='same',activation='relu'))
        self.conv.add(Conv2D(4,(3,3),padding='same',activation='relu'))
        self.conv.add(Conv2D(1,(3,3),padding='same', activation='relu'))

    def call(self, x):
```

```
76          x = self.conv(x)
77          x = x[:,::-1,:,:].reshape((x.shape[0], -1), order='F')
78          return x
79
80  class DQNAgent:
81      def __init__(self, state_size, action_size, env:TopOptEnv,
        ↪  load_=False):
82          self.state_size = state_size
83          self.action_size = action_size
84          self.memory = deque(maxlen=30000)
85          self.gamma = 0.1
86
87          self.epsilon_decay = 3.5e-4
88          self.epsilon_min = 0.01
89          self.learning_rate = 2.5e-3
90          self.e_start = 0
91          self.epsilon = 0.9
92
93          self.model = _build_model()
94          self.model_target = _build_model()
95
96          if load_:
97              self.model.built = True
98              # Specify which model to load here
99              self.load(output_dir+'weights_2200.hdf5')
100             print(f'Model Succesfully Loaded')
101
102         self.model.compile(loss='mean_squared_error',
            ↪  optimizer=Adam(lr=self.learning_rate))
103         self.model_target.compile(loss='mean_squared_error',
            ↪  optimizer=Adam(lr=self.learning_rate))
104
105         self.registering_memory_step = 0
106         self.env = env
107
108
109
110     def remember(self, state, action, reward, next_state, done):
111         self.registering_memory_step += 1
112         self.memory.append((state, action, reward, next_state, done))
113
114     def train(self, batch_size, episode_num):
115
116         if len(self.memory) < batch_size:
117             return
118
```

```
119         minibatch = random.sample(self.memory, batch_size)
120
121         state_shape = (batch_size, self.state_size[0],
            ↪  self.state_size[1], self.state_size[2])
122         states = np.zeros(state_shape)
123         states_nxt = np.zeros(state_shape)
124         actions, rewards, dones = [],[],[]
125
126         for i in range(batch_size):
127             states[i] = minibatch[i][0]
128             states_nxt[i] = minibatch[i][3]
129             actions.append(minibatch[i][1])
130             rewards.append(minibatch[i][2])
131             dones.append(minibatch[i][4])
132
133
134         targets = onp.array(self.model(states))
135         targets_nxt = self.model(states_nxt)
136
137         targets_val = self.model_target(states_nxt)
138
139         for i in range(batch_size):
140             if dones[i]:
141                 targets[i][actions[i]] = rewards[i]
142             else:
143                 a_max = np.argmax(targets_nxt[i])
144                 targets[i][actions[i]] = rewards[i] + self.gamma *
                    ↪  targets_val[i][a_max]
145
146         self.model.fit(states, targets, epochs=1)
147
148         self.new_eps = 1 - episode_num * self.epsilon_decay
149
150         if self.new_eps > self.epsilon_min:
151             self.epsilon = self.new_eps
152         else:
153             self.epsilon = self.epsilon_min
154
155
156     def act(self, state, DQN_q_vals):
157         # Filtering the action_space s.t. we only randomize from legal
            ↪  actions
158         new_action_space = range(36)
159         new_action_space = [ele for ele in new_action_space if ele not in
            ↪  self.env.bounded_cells]
```

```
160         new_action_space = [ele for ele in new_action_space if ele not in
            ↪  self.env.loaded_cells]
161         if onp.random.rand() <=0.4:
162             new_action_space = [ele for ele in new_action_space if ele
                ↪  not in self.env.removed_cells]
163         if onp.random.rand() <= self.epsilon:
164             print(f"\n############### RANDOM ACTION ###############")
165             return random.choice(new_action_space), DQN_q_vals
166         print(f"\n############### DQN ACTION ###############")
167 
168         state_inp = onp.array(state[onp.newaxis, :, :, :])
169         act_values = self.model(state_inp)
170 
171         new_act_values= onp.array(act_values).squeeze(0)
172         print(f'New act Values : {new_act_values}')
173 
174         DQN_q_vals.append(onp.argmax(new_act_values))
175 
176         return onp.argmax(new_act_values), DQN_q_vals
177 
178     def save(self, name):
179         self.model.save_weights(name)
180 
181     def load(self, name):
182         self.model.load_weights(name)
183 
184 load_ = False
185 agent = DQNAgent(state_size, action_size, env, load_=load_)
186 
187 n_episodes = 5000
188 DQN_avg_q_vals = []
189 
190 for e in range(agent.e_start, n_episodes):
191     state = env.reset()
192     done = False
193     time = 0
194     DQN_q_vals = []
195     while not done:
196         action, DQN_q_vals = agent.act(state, DQN_q_vals)
197         state, action, reward, next_state, done = env.step(action, e)
198 
199         agent.remember(state, action, reward, next_state, done)
200         state = next_state
201         print(f'REWARD : {reward}')
202         if done:
```

```
203             print("episode: {}/{}, score: {}, eps: {:.2}".format(e,
                ↪  n_episodes-1, reward, agent.epsilon))
204
                ↪  DQN_avg_q_vals.append(onp.sum(onp.array(DQN_q_vals))/len(DQN_q_vals))
205         time += 1
206     # Save q vals in csv for easy reading and plotting
207     data = onp.asarray(DQN_avg_q_vals)
208     onp.savetxt('Avg_q_vals_per_episode.csv', data, delimiter=',')
209
210     agent.train(batch_size, e)
211
212     if e % 100 ==0 and e>1:
213         agent.model_target.set_weights(agent.model.get_weights())
214
215     if e % 50 == 0 or e == n_episodes-1:
216         print(f"\nLen of memory = {len(agent.memory)}")
217         if load_:
218             output_dir = 'model_output_loaded/'
219             if not os.path.exists(output_dir):
220                 os.makedirs(output_dir)
221         agent.save(output_dir + "weights_" + '{:04d}'.format(e) +
            ↪  ".hdf5")
222
223
```

A.5 Illustration of an entire training episode

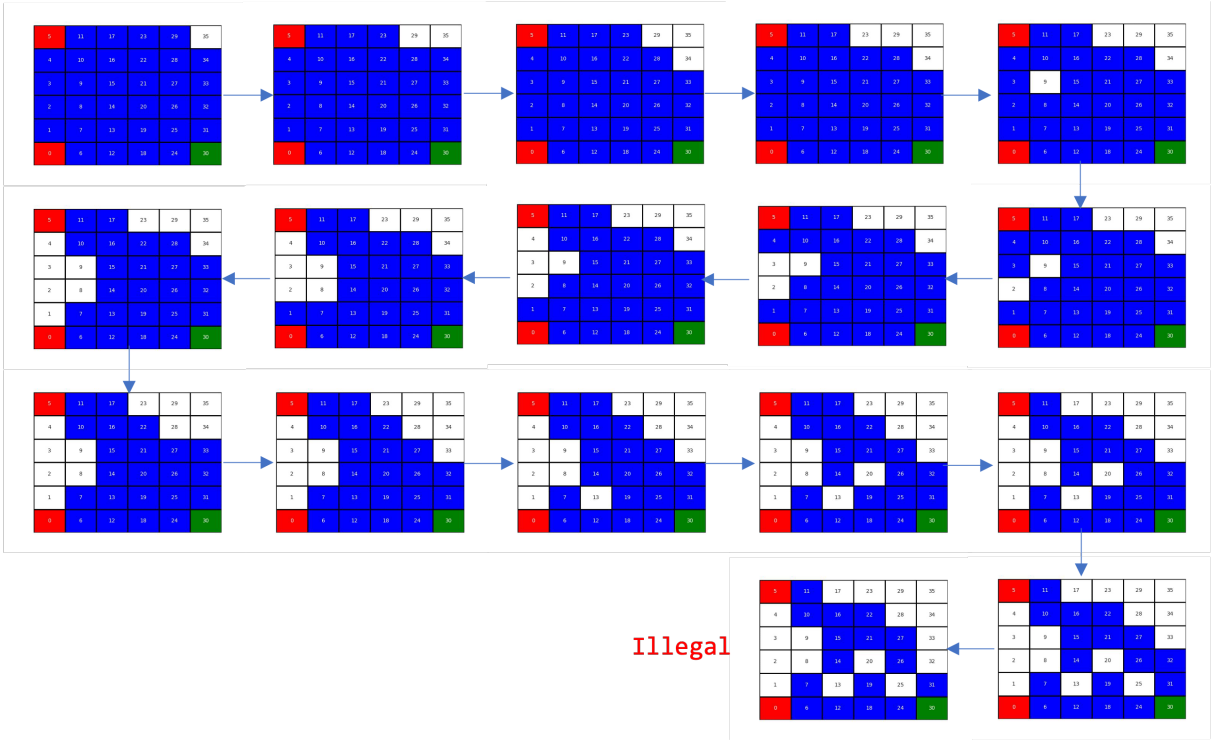


Figure 13: Training episode 2245