Learning to Design Without Prior Data: Discovering Generalizable Design Strategies Using Deep Learning and Tree Search

Ayush Raina[1]
Department of Mechanical Engineering,
Carnegie Mellon University,
Pittsburgh, PA 15213
e-mail: araina@andrew.cmu.edu

Jonathan Cagan
Department of Mechanical Engineering,
Carnegie Mellon University,
Pittsburgh, PA 15213
e-mail: cagan@cmu.edu

Christopher McComb
Department of Mechanical Engineering,
Carnegie Mellon University,
Pittsburgh, PA 15213
e-mail: ccm@cmu.edu

Building an Artificial Intelligence (AI) agent that can design on its own has been a goal since the 1980s. Recently, deep learning has shown the ability to learn from large-scale data, enabling significant advances in data-driven design. However, learning over prior data limits us only to solve problems that have been solved before and biases data-driven learning toward existing solutions. The ultimate goal for a design agent is the ability to learn generalizable design behavior in a problem space without having seen it before. We introduce a self-learning agent framework in this work that achieves this goal. This framework integrates a deep policy network with a novel tree search algorithm, where the tree search explores the problem space, and the deep policy network leverages self-generated experience to guide the search further. This framework first demonstrates an ability to discover high-performing generative strategies without any prior data, and second, it illustrates a zero-shot generalization of generative strategies across various unseen boundary conditions. This work evaluates the effectiveness and versatility of the framework by solving multiple versions of two engineering design problems without retraining. Overall, this paper presents a methodology to self-learn high-performing and generalizable problem-solving behavior in an arbitrary problem space, circumventing the need for expert data, existing solutions, and problem-specific learning. [DOI: 10.1115/1.4056221]

Keywords: design automation, artificial intelligence, deep learning, generalizability, agent-based design, data-driven design, design representation, generative design, machine learning, Monte Carlo Tree Search

1 Introduction

Solving design problems is one of the most ubiquitous processes in engineering and arguably the most challenging [1,2]. The design automation research paradigm aims to augment the continually evolving design-solving process by enabling machines to engage in design. Despite decades of research in the area, modern-day automated design synthesis is still heavily guided by handcrafted rules and prior expert data, making it susceptible to non-generalizability and errors resulting from human bias [3,4]. Developing a design agent that can learn from scratch is still a long-standing challenge. This paper addresses this challenge by introducing a generalizable design agent framework that integrates newly developed tree search and deep learning methods. The tree search enables exploration and information gathering, while the deep learning representation helps the agent leverage self-generated experience. Together, these methods provide a symbiotic integration of decision-making methods to effectively explore and learn in unknown design problem spaces.

Learning problem-solving strategies from scratch has been achieved in multiple domains [5–8]. Some of these methods use a dual-process decision-making framework which is often compared to the *slow* and *fast thinking* ideology [9]s. That framework consists of a deep learning model that simulates fast and intuitive thinking,[2] while the tree search component corresponds to slow and rational thinking. This method is well suited for discretized spaces as

found in standard games [5–7] and combinatorial problems [10,–13]. For instance, games like Chess, Go, poker, and Atari have a given set of discrete rules and actions that are universally standardized. In contrast, design problems often have more open-ended and loosely defined formulations which can vary substantially across engineering domains. The action space of a design process is challenging to generically represent as it constrains the problem space and dictates the exploration capability of a designer. Computationally, the action space may include a combination of continuous and discrete parameters; they may be constrained, state-dependent, or even temporally divergent, as the number of actions keeps on increasing as the process progresses [14]. An action definition that can represent all such variability is referred to as a complex action space [14,15].

The concept of a complex action space is the central piece in extending deep learning and tree search methods to design problems. While a complex action space enables standardization of state-action definitions across engineering domains, it is challenging to represent with existing data-driven methods that use deep learning and tree search [15–17]. This paper solves this challenge by first using a complex action-based formalism for defining and *standardizing* design processes. Second, the paper proposes novel methods to enable the application of deep learning and tree search algorithms to complex action spaces that enable *representation* and *discovery* of design strategies. These three methods (*standardizing*, *discovering*, and *representing*) provide the basis for the Self-Learning Design Agent (SLDA) framework introduced in this work.

Figure 1 summarizes the framework and the overall methodology of the paper. The methodology begins by standardizing the problem space followed by employing the deep learning and tree search-based decision-making. Here, the deep learning network leverages past experiences in terms of preference for actions and the tree

---

[1]Corresponding author.

[2]Thinking is defined as the process before any actionable change taken by an agent. The actionable change is referred to as decision making.
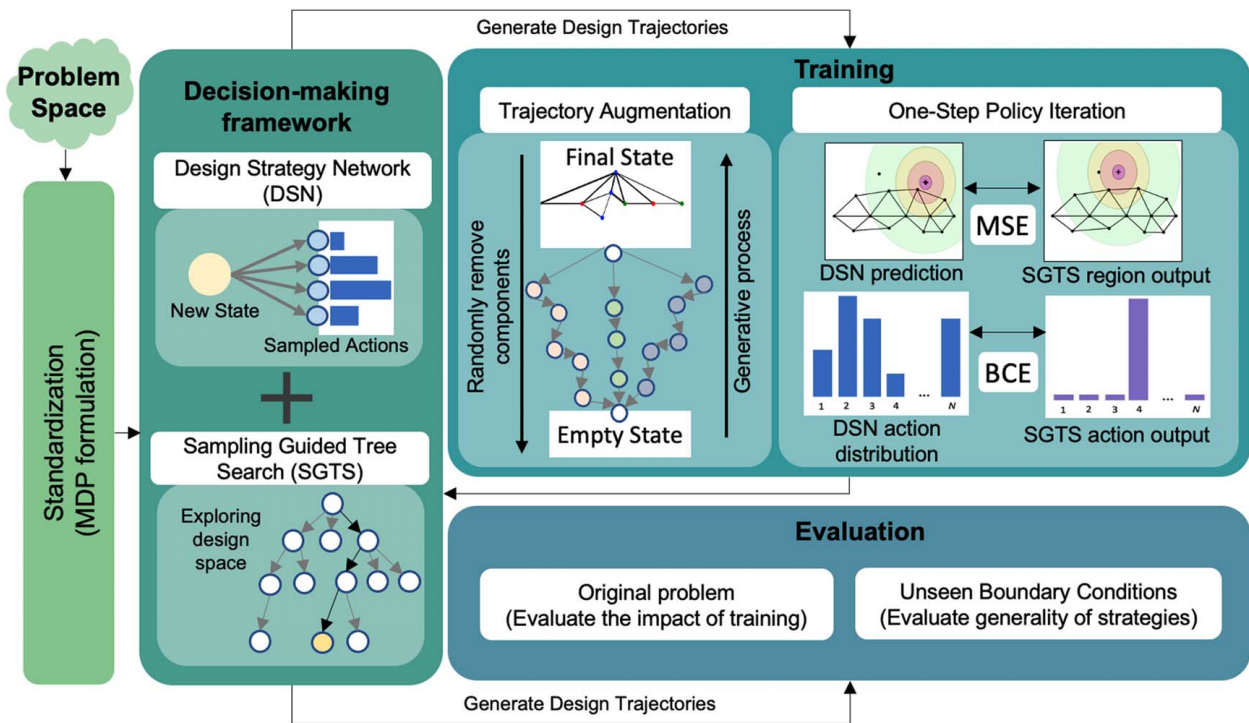
Fig. 1 Schematic representing the Self-Learning Design Agent (SLDA) framework

search further improves the decision-making by evaluating design states emanating from the preferred actions. Starting from random initializations of the network, the SLDA generates several design trajectories, discovering high-performing generative design behavior. These self-generated trajectories are used to train the deep learning network to emulate high-performing generative design behavior. No prior or expert data are required in this process as the agent learns with self-generated data. Finally, the methodology culminates with the evaluation of the SLDA framework.

The ability to solve a design problem from scratch is a multi-faceted skill. So far, we have discussed the importance of enabling self-learning using computational methods. However, the evaluation of such agents also requires careful consideration. An expert designer is expected to solve several problems within a domain. For example, a bridge designer can create diverse structures over varying terrain conditions since they are all based on similar underlying structural design principles. Hence, an expert design agent is not just expected to solve one problem but also learn the generative process itself, which includes the underlying principles of generating high-performing solutions. Several methods in design agent literature like simulated annealing [18], genetic algorithms [19], and tree search methods [20–23] solve an isolated optimization task offering no transfer of strategies or experience across analogous problems. More recently, generative models [24–26] and reinforcement learning [27–29] methods have been developed to leverage experience; however, these works find the highest performing design in particular problems with no focus on transferability or generalizability.

Our paper presents the first agent framework capable of learning generic design strategies from scratch. The agent can discover a common generative design strategy given only possible actions and illustrate its capability across more challenging variations of the problem. The contributions made by the paper are two-fold and answer the following research question: *Can a deep learning-guided search agent enable learning of design strategies from scratch (without prior data) and illustrate an effective transfer across problems?* The agent is evaluated on two diverse problems: truss design and circuit routing. Both truss design and circuit routing are formulated as sequentially generative design problems

with several variations. The variations within the problems are analogous to tasks and support assessment of generalizability of the SLDA framework.

The paper is divided into five sections. Section 2 contains a background of some of the methods used in the framework. The overall SLDA framework and the individual methods that compose the agent framework are introduced in Sec. 3. Section 4 introduces the truss design problem, experimental setup, and results. Section 5 introduces the circuit routing problem, experimental setup, and results. Finally, Sec. 6 concludes the paper by highlighting the main findings of the research.

2 Literature Review

This section includes the literature review about the existing methods leveraged in the agent framework: Monte Carlo Tree Search (MCTS) and agent frameworks that leverage deep learning with a tree search algorithm.

2.1 Monte Carlo Tree Search. The MCTS algorithm [30–32] is a best-first search algorithm that does not require any prior data and can iteratively gather information to enhance decision-making. Although computationally expensive, MCTS has been very popular in the literature [30,33], especially in long-horizon planning applications where no immediate reward is available. Several methods aim to improve the speed of the search such that decisions can be made quicker. They often use neural networks [34] and distributive computing [35,36] to achieve a linear speedup with a minimal performance drop. Liu et al. [37] provide one such distributed approach called Watch the Unobserved Upper Confidence Trees (WU-UCT). It leverages a unique statistic for counting incomplete simulations, which improves its decision-making capability motivating its usage in our work.

Monte Carlo Tree Search algorithms were primarily designed for discrete actions [32], where all feasible actions can be listed. Specific extensions are required for application to continuous domains and other action spaces [17,38–40]. Hubert et al. [41] provide an alternative methodology for dealing with arbitrary

complex action spaces. They use a sampling-based action-independent formulation and enable a new policy iteration method. Our current research provides a much simpler approach that extends the WU-UCT formulation [37] by employing an order-invariant policy network [14] to sample feasible actions.

2.2 Deep Learning and Tree Search. MCTS has been increasingly used in several problem-solving applications due to its ease of integration with data-driven deep learning models. Successful agents [5,42,43] have leveraged deep learning-based models to guide the decision-making of MCTS, enabling superhuman-level performance. These agents integrate fast, intuitive thinking from policy networks with a deep search using MCTS. The key idea is that MCTS can generate novel experiences by interacting with the environment and planning for long-term rewards, while the deep learning aspect can learn from that data to generalize plans and leverage it to augment future decision-making [11,44]. These approaches have recently been applied to several design problems, including robot design [45], chip placement [46], and circuit routing [47], amongst others [13]. However, these methods do not consider complex action spaces and consider the problems as a one-time search and optimization method.

The idea of self-play was developed by Silver et al. [42], where the agent plays Go with itself while iteratively improving its performance. Such frameworks involve alternating between a data generation phase where the MCTS explores and a learning phase where deep learning models are trained [11,12]. The basic idea for such methodology is that the search agent acts as an expert who can utilize computation to make meaningful actions, and then, the policy model regresses over the decisions made by the teacher MCTS agent [48,49]. That method treats the problem of learning as a classification problem where the network imitates the expert action [11,43,50]. Those methods are much more sample efficient than randomly interacting with the environment and learning by reinforcement. Our method follows one iteration of the self-play (self-learning) algorithm by Silver et al. [42] and only trains the policy network once over the self-generated data. Our method does not use a value network due to its propensity to require diverse data for proper training and attribution of correct values based on optimal policies and not rough estimates [10,11].

3 Self-learning Design Agent Framework

This section presents the SLDA framework and its five constituent methods. SLDA framework presents a methodology to discover and learn design strategies given nothing but the design actions and physics-based evaluation of the design state. Figure 1 represents the overall agent framework, showing the flow of the agent. The SLDA framework includes the three colored boxes in the figure, which correspond to decision-making, training, and evaluation mechanisms.

We first convert an abstract problem into a *standardized* state-action representation before applying the SLDA framework on it. A unique complex-action Markov Decision Process (MDP) [51] is developed (in Sec. 3.1) that enables a task-independent decomposition of design processes into a state, action, probability, and reward definition. This provides a methodology to formulate a generative design problem into a sequential decision process with complex actions.

Next, the SLDA involves a decision-making mechanism, composed of a dual-process methodology including deep learning and tree search. Section 3.2 details the deep learning part which *represents* generative design strategies in complex action spaces. This deep learning network is built upon Design Strategy Network (DSN) [14] and enables SLDA to learn over self-generated experience and extract meaningful relationships between the state and selected actions. However, in this work, this framework is extended from a descriptive model to a generative model as it is used to predict actions on sequential design states. The tree search part of the decision-making mechanism is introduced in Sec. 3.3. A

novel tree search method called Sampling Guided Tree Search (SGTS) is proposed in this work to *discover* strategies in complex action spaces. It leverages the deep learning network from Sec. 3.2 to filter out preferred actions and then fine-tunes the final action selection using tree search. SGTS is the planning component of SLDA which iteratively gathers information in the unseen problem space by exploring design states.

The combined decision-making mechanism of SLDA is used to generate several design trajectories which are then used as the dataset for training the deep learning model. The training mechanism here includes a novel augmentation process and a one-step policy iteration algorithm. The augmentation process is detailed in Sec. 3.4 and creates diverse trajectories that lead to the same final design, enhancing the quality of the dataset. Finally, Sec. 3.5 introduces the one-step policy iteration used to train the deep learning model of SLDA. Overall, this section introduces the novel SLDA framework and provides the working details for the involved methods that SLDA uses to achieve learning from scratch in design.

3.1 Design as a Complex Action Markov Decision Process. This section introduces a unique complex action MDP that can define a generative design process as a sequential process [14,52,53]. This definition provides a standardized manner of representing design processes and guiding environment development for learning-based systems. This work extends an MDP to include state-dependent action formulation to represent arbitrarily complex actions. A standard MDP is represented as a four-tuple $\langle S, A, P, R \rangle$. Here, S corresponds to the state space and s_t represents the design state at time t. Design state can be composed of one of the many representation methods like images, graphs, voxels, or point clouds amongst others [4] that can comprehensively include all information about the design state. A refers to the action space composed of all possible actions. Every design process is composed of a finite set of action classes or generative grammar types, $A_{class} = \{a_1, a_2, \ldots, a_K\}$, that modify the design state. Each action may have associated parameters that define its properties, represented as $a_k = \langle a_k^d, a_k^c \rangle$, where a_k^d refers to the discrete component and a_k^c refers to the continuous component of the action class. The properties of the actions including their feasibility are conditioned on the current state. Hence, the state conditional action space can be defined as a union of all action tuples a_k in A_{class}

$$A = \bigcup_{k \in |A_{class}|} \langle a_k^d | s_t, a_k^c | s_t \rangle \tag{1}$$

The continuous component of the action a_k^c can lead to infinite feasible actions in a given state. We propose a sampling-based approach to discretize the continuous domains and limit the feasible actions [14]. A prior probability p_k is defined to sample in the continuous parameter domain a_k^c for every action type a_k. Previous work models p_k as a mixture of bivariate Gaussian distributions to represent a spatial region [14] in a design space; however, arbitrary prior distribution definitions can be used depending on the design task. The prior distributions can further be learned using a deep learning model that can be conditioned on the current state. This prior-based sampling method enables an adaptive discretization of complex action spaces.

The transition probability defined by P is always 1 for deterministic environments like generative design which means, given state s_t and action a the resulting state is always a fixed state s_{t+1}. The reward function is denoted by R that takes the input of the current state and an arbitrary action and returns a reward value, $r = R(s_t, \langle a_k^d, a_k^c \rangle)$. This formulation can model most generative design problems as a complex-action MDP and extends the application of deep learning and computational methods to design.

3.2 Design Strategy Representation. This section introduces a method for representing decision-making strategies in complex action spaces. A previously developed hierarchical neural

network architecture called a DSN [14] is leveraged to represent design strategies. A strategy or a policy is represented by a function π

$$\text{Prob}(A) = \pi_{DSN}(s) \tag{2}$$

where Prob(A) represents the probability distribution of feasible actions A in state s and π_{DSN} represents the data-driven architecture that can be trained based on design trajectories of designers to mimic their decision-making.

A DSN is composed of three deep-learning networks that split the learning problem. First, an encoder network takes in the design state as an image and generates a low-dimensional latent representation. Second, a linear action network generates a prior that governs feasible action sampling. Finally, the feasible actions and latent state representation are input into an order-invariant selection network which generates a probability distribution over the set of feasible actions. More details on the architecture are provided in the original paper [14]. This hierarchical policy network generates a combined latent representation of state and action and transforms it into an initial prior for action selection.

3.3 Sampling Guided Tree Search.
This subsection details the novel SGTS algorithm that enables search and exploration in complex action spaces. The code for this algorithm has been provided for public use.[3] SGTS combines the DSN with a tree search to compute the best actions in each state. SGTS algorithm takes an input s and iteratively builds a search tree to return the best action a based on the gathered information. Each node of the tree represents a state and has two main search statistics N_{s_t} and V_{s_t}, which record the number of visits and cumulative reward of state s_t, respectively.

SGTS extends the application of the distributed WU-UCT [37] algorithm to complex action spaces of design problems by modifying the decision-making. An order-invariant policy network (DSN) is introduced to guide the decision-making toward a finite sampling of preferred actions along with an action distribution. The overall tree search algorithm undergoes three sequential processes: *tree traversal*, *expand and rollout*, and *backpropagate*. This is similar to most tree search algorithms. However, SGTS introduces changes in the *expand* and *backpropagate* steps. The three processes are defined as follows:

(1) *Tree traversal* step is identical to the original algorithm [37]. It starts at state s and traverses the explored part of the tree using the following formula from the WU-UCT algorithm [37] until a leaf node (unexplored state) is reached

$$a_s = argmax_{s' \in \mathcal{C}(s)} \left\{ V_{s'} + \beta \sqrt{\frac{2 \log(N_s + O_s)}{N_{s'} + O_{s'}}} \right\} \tag{3}$$

This formula controls the balance between exploration and exploitation in deciding which action to pick. Here, N_s represents the number of visits to the state s. $\mathcal{C}(s)$ refers to the set of children nodes of an explored state s. O_s refers to the number of incomplete *rollout* processes at state s. The term inside the square root guides exploration as its value is high when state s' is unexplored. $V_{s'}$ corresponds to the accumulated value of state s' after multiple iterations of tree search. β balances the tradeoff between the exploitation (accumulated value $V_{s'}$) and exploration (component within the square root) by enabling relative weights between them.

(2) *Expand* process is executed when an unexplored state (leaf node) is encountered. DSN is used to sample a set of feasible actions $A(s)$ which determines the children states $\mathcal{C}(s)$

$$\mathcal{C}(s) \leftarrow environment(s, A(s)); A(s) \leftarrow \pi_{DSN}(s) \tag{4}$$

---

[3]https://github.com/ayushraina10/SGTS

This step differentiates SGTS from standard MCTS since a policy network (DSN) is used to sample a finite set of actions. This step uses the DSN to guide the search from infinitely possible actions toward specific design spaces by limiting the action set. This sampling of actions introduces stochasticity in the design decision-making process which is otherwise deterministic. The resulting new states are then evaluated using a rollout process, which again leverages DSN for selecting actions at every state. The rollout process again leverages DSN to sequentially take actions until either a terminal state or max search depth is reached. It returns an estimated value $\hat{V}$ of the new state. The rollout essentially acts as an information retrieval step that explores multiple design trajectories and identifies any meaningful or high-performing states. A properly trained DSN will guide the search toward better design spaces and enable a more accurate rollout evaluation of new states. Hence, DSN can control the effectiveness of this information retrieval process as it guides and controls the search using action sampling.

(3) The *backpropagate* step then updates the statistics of all intermediate nodes on the path as follows:

$$N_{s_t} \leftarrow N_{s_t} + 1 \tag{5}$$

$$\hat{V}_{s_t} \leftarrow \max\left(R(s_t, a_t) + \gamma \hat{V}_{t+1}, \hat{V}_{s_t}\right) \tag{6}$$

$$V_{s_t} \leftarrow \max(V_{s_t}, \hat{V}_{s_t}) \tag{7}$$

Here, γ refers to the reduction factor in rewards and lies between [0,1]. R corresponds to the rewards obtained with s_t and a_t. This formulation is different from the WU-UCT algorithm and most other MCTS formulations since it is based on a modified Maximum Upper Confidence Trees (Max-UCT) [54,55] instead of the average UCT. Average UCT is traditionally used for zero-sum games like Chess and Go, where actions must be robust across all possibilities, and a greedy search can lead to bad performance. However, in design applications, averaging provides a poor estimate of the state especially when a high-performing state is neighboring several low-performing states. Hence, a Max-UCT mitigates these issues and helps the search process pursue high-performing states.

The whole three-step process corresponds to one simulation (*Sim*) of SGTS. Increasing the number of simulations, depth, and width of the tree search enhances the performance of MCTS algorithms as it gathers more information about the problem space while increasing the computation time. After multiple simulations, the final values of V_{s_t} (cumulative reward for the particular state) are used to compute the final action based on the WU-UCT formula Eq. (3), leading to a new state s'. SGTS uses an identical formulation to WU-UCT for the algorithm apart from the details highlighted in this Section. More information about the original algorithm can be found in the study by Liu et al. [37].

3.4 Trajectory Noise: Augmentation Method for Generative Processes.
This subsection presents a novel approach to augment generative design trajectories. Data augmentation methods are often used to diversify data during reinforcement learning training [10]. However, design states have essential spatial and physical information (like gravity) that may be obfuscated with standard augmentation methods. A unique augmentation method, Trajectory Noise, is developed for generative design applications. This method applies to processes where different action sequences can lead to the same resulting state. The algorithm (represented graphically in Fig. 1) takes the final design state as the input and then sequentially removes components until an empty state (initial state) is reached. It selects the component to remove

```
Algorithm : Self Learning Design Agent - One Step Policy Iteration
π_DSN_0 = Randomly Initialized Policy
SGTS = Tree Search Algorithm
param = Search Parameters
D = Empty Dataset
for Sample = 1 : N do
    s ← Initial State
    trajectory = Empty List
    R = Empty List
    while s not terminal do
        a ← SGTS(s, π_DSN_0, param)
        s', reward ← Env(s,a)
        trajectory.append(s, a, s')
        R.append(reward)
        s ← s'
    end
    if isFeasible(R) then
        D.append(TrajectoryNoise(trajectory, TN))
end
π_DSN_1 ← Train(π_DSN_0, D)
```

Fig. 2   Algorithm for self-learning design agent framework

randomly and hence generates several diverse trajectories. The reverse of the removal process corresponds to a generative design trajectory. Since the final state is the same, the trajectory rewards are unchanged unless a high-performing intermediate state is reached. The process can be repeated multiple times to generate several trajectories for a given design state. In a design process, multiple actions may be equally preferred as they reach the same final state and must have an equal probability of selection; this algorithm helps uncover such multiplicity. Although the generated trajectories have different sequences of actions, they are all functionally equivalent since they lead to the same final design. As a result, this augmentation introduces diversity in action selection, in turn allowing a richer learning of state–actions relationships from trajectory data.

3.5   Training Process. This section introduces details of the training process of the SLDA framework. Figure 2 presents the agent's algorithm for one-step policy iteration on self-generated data. SLDA Algorithm in Fig. 2 can be understood as follows. The training process begins with a randomly initialized DSN policy π_{DSN_0}. N trajectories are generated using the agent. The SGTS algorithm explores the state space with random priors from π_{DSN_0} and appropriate search parameters (*param*). At every step, the SGTS undergoes multiple simulations of tree search to determine the best action a in the given state. A trajectory is generated by iteratively applying the SGTS algorithm and applying the best action a in the state until a terminal state is achieved. SGTS with a high compute budget acts as a policy improvement operator [56], which improves the action predictions of the random policy π_{DSN_0} by multi-step lookahead search. The generated trajectory is augmented using the TrajectoryNoise function (Sec. 3.4) if the trajectory corresponds to a feasible design. TrajectoryNoise takes an argument TN which determines the number of augmented
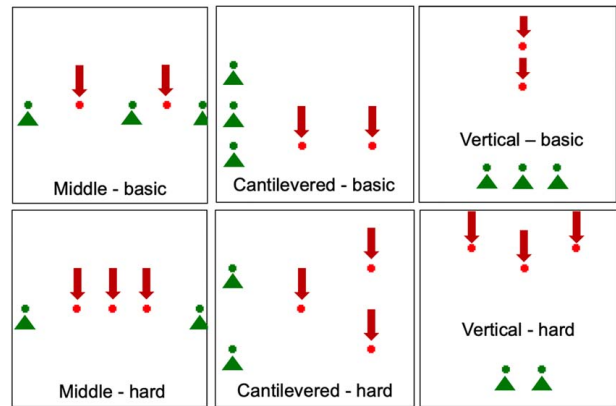
Fig. 3   Six different boundary conditions of the truss design problem are illustrated. The red arrows represent the loading nodes, and the green triangles refer to the support nodes. The top row represents a basic version of the problem, while the lower row corresponds to a more complex version with three loading nodes.

trajectories to generate for a given trajectory. The collected data D are then used to train the policy network to create an updated network π_{DSN_1}. The policy π_{DSN_1} is trained to predict the action a given state s. The goal of the learning problem is for the DSN to imitate the SGTS algorithm's behavior in dataset D.

4   Case Study: Truss Design Problem

4.1   Problem Definition. Truss design is a classic engineering design problem [52,53,57,58] where the goal is to build a bridge that can support a given load. The bridge is a structure of carefully placed components like members (beams) and nodes (joints). The objective is to achieve a target strength of the structure by utilizing minimal mass. Generically this problem can be defined as a parameterized configuration design problem [59] where the final configuration of members and nodes needs to be determined. For the scope of this research, the whole process of generation, iteratively placing, and tuning the components is considered. This formulation is similar to previous work [14]; however, the generation process is limited to 100 steps. The state, action, probability, and reward definitions of the problem are provided in Table 1. This work develops six Boundary Conditions based on standard truss design problems.

Figure 3 shows the initial states of the boundary conditions, which have a set of support nodes (green triangles) and loading nodes (red arrows). New nodes are blue circles, and new members are shown in Black (shown later in Fig. 7). The triangles and arrows are for indication purposes only. Three boundary conditions, inspired by standard loading cases [60–62], are developed: Middle, Cantilevered, and Vertical loadings. These boundary conditions are designed in pairs, a basic one, and its more complicated version. These pairs allow the problems to be split into training and testing sets to evaluate the generalization capability of the agent's

Table 1   MDP formulation for the truss design problem

State	A colored image representation of the problem ($128 \times 128 \times 3$ pixels). It captures the design space in an RGB image, as shown in Fig. 3	
Action	Action type	Parameters
	Add a node (*continuous*)	$x, y \in$ Continuous 2D space
	Add a member (*state-dependent discrete*)	$(x_1, y_1, x_2, y_2) \in$ Node Set
	Increase thickness (*state-dependent discrete*)	$(x_1, y_1, x_2, y_2) \in$ Member Set
Probability	$P = 1$, a deterministic process	
Reward	$R = \dfrac{\text{targetMass} \times \text{FOS}}{\text{mass}} + \dfrac{\max(\text{FOS} - \text{targetFOS}, 0)}{\text{FOS} - \text{targetFOS}}$	

Table 2 Variants of the SLDA agent

SLDA Variants	Search parameters			Boundary condition	Number of samples
	Simulations	Depth	Width		
1-TASK TRAINED (T)	128	5	10	Middle-Basic	48
MULTI-TASK TRAINED (MT)	128	5	10	All basic	3×48

strategies; this is analogous to multi-task learning environments where an agent is expected to solve diverse challenges [63].

The reward function in Table 1 is based on a physics-based finite element evaluation of the truss structure. Factor of Safety (FOS) corresponds to the stresses developed due to loads and represents the structure's strength. *targetFOS* corresponds to the required strength level (here, *targetFOS* = 1.0). *mass* corresponds to the material weight of the designed structure. *targetMass* is a normalizing factor that maintains the range of $R \in [0, 2]$, by setting *targetMass* to 100.

4.2 Policy Network Architecture. The policy network represents the relationship between the state and the actions of a generative design problem. Given an image-based representation of the design state, a policy network generates an action distribution across all feasible actions, illustrating the actions' importance. The policy network for truss design is similar to DSN used to imitate human strategies in previous work [14]. Two modifications to the DSN network are made. First, this formulation uses a *coordconv* layer [64] as the first layer of the encoder network to extract spatial information more accurately from image representations [64]. Second, the selection network uses a feasible action mask, which equates the probabilities of non-feasible actions to infinitely small values. This masking methodology is more effective for training over a large action space with numerous infeasible actions [65].

4.3 Training the SLDA Agent for Truss Design. Two different variants of design agents are developed. The first variant is the Trained SLDA; it is trained on only Middle-basic. The second variant is termed Multi-task Trained (MT). This variant is trained over three conditions: Middle-basic, Cantilevered-basic, and Vertical-basic. In contrast to Trained SLDA, MT Trained agents learn potentially a more generalized generative design strategy by learning over three different boundary conditions. Table 2 lists the differences between the two variants. Other search parameters are kept consistent across the runs, balance between exploration and exploitation (β) is set at 2, and the discount factor (γ) is fixed at 0.95 across all runs. The discount factor was adjusted from 0.99 (WU-UCT [37]) to 0.95 to weigh immediate rewards higher and guide the design toward feasible regions. Trajectory noise augmentation is used for training both agents. This augmentation is used to sample 10 different trajectories for every final state, effectively augmenting the data size ten times.

The training process is identical to DSN [14]. Two loss functions are used, Mean Square Error (MSE) for spatial region prediction and Binary Cross-Entropy (BCE) loss for the final action selection. An illustrative representation of the training process for truss design is shown in Fig. 1. The DSN is trained to emulate the decision-making of the SGTS algorithm. Hence, the policy network imitates the decision-making achieved by an in-depth computation and captures the underlying state-action relationships generating designs. A 90:10 train–test split is used for evaluating the trained DSN. The network of Trained SLDA (one boundary condition) achieves 84% test accuracy with final action prediction while MT SLDA (three boundary conditions) network achieves 78%. For comparison, a random policy leads to 6.67% and 11.7% accuracies, respectively.

4.4 Experimental Setup. The experimental setup for truss design involves different design boundary conditions and multiple versions of the SLDA algorithm with different search parameters. Three different agent variants are compared: Untrained, Trained, and MT. Two experiments are developed to evaluate their effectiveness on truss design problems. Experiment 1 evaluates the advantage of using trained policy networks to guide the search process and gauges the impact of varying the search budget of the agent. The idea is to compare trained and untrained agents to observe how training and search budget affect performance. Experiment 2 extends the evaluation of the three agent variants to unseen problems. The agents are evaluated over the other five Boundary conditions to observe the effectiveness of the learned strategies. This experiment evaluates the generality of the generative behavior of the agent by evaluating its performance over unseen boundary conditions.

4.5 Results 1: The Trained SLDA Agent. Figure 4 compares the performance efficiency of SLDA variants on the "Middle-basic" truss design problem. The *y*-axis represents the Reward value, and the *x*-axis represents the time taken by the agent to generate a single action on a logarithmic scale. The left end of the graph can be viewed as evaluating the deep learning priors since the compute level is low, and the agent depends on the decision-making of the policy network. The right end of the graph corresponds to a high compute budget (deep search); at that point, the impact of priors is diminished. However, the middle portion of the graph illustrates the combination of deep learning and search-based strategies. Every point is a unique SLDA agent; the connecting lines indicate the same agent variant used to generate the sample. Each point represents the Inter Quartile Mean[4] (IQM) of reward over 48 runs of the agent. IQM helps to normalize any noise in the results to outliers. A reward value greater than 1 represents a feasible design. The legend in Fig. 4 details the labeling scheme. The shape corresponds to the number of simulations, all samples have depth 5 except for Sim 256, which has a depth of 10 (corresponding to very high search budget). From left to right, the increasing time represents that the agent has access to a higher search budget. The dashed horizontal lines represent the IQM performance levels by humans [60], Deep Learning Agents (DLA) [52], and Goal-Directed DLAgents (GD-DLA) [53] as labeled in Fig. 4. It must be noted that those agents were trained with human data and aim to replicate their behavior on this problem. The human performance is based on a study with undergraduate engineering design students who have experience in solid mechanics. More information about how human-level performance was measured is provided in previous work [60].

Several observations can be made from this plot. First, untrained agent illustrates high performance without training. It can outperform mean-human-level performance and DLAgents with a search budget of 128 Simulations and 5 Depth. This high-level performance illustrates the effectiveness of the SGTS algorithm. The agent can explore the design space and identify meaningful solutions even with random priors. Second, it can be observed that both trained agents (Trained and MT) perform much better than

---

[4]Inter Quartile Mean (IQM) finds the mean of the data between the 25% and 75% of the data samples. This method excludes any outliers by discarding the best and worst values from the samples and is a robust metric to evaluate stochastic data collection [66].
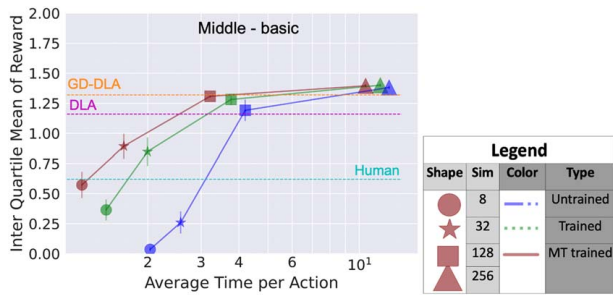
Fig. 4 Performance efficiency for SLDA variants for middle-basic boundary condition

Fig. 5 Performance efficiency for SLDA variants for the unseen mid-hard boundary condition. All samples have search depth 5.

the untrained version, especially with low search budgets showing the usefulness of the priors, which enable the search to focus on essential regions in the design space and emulate high-level performance with only a fraction of the search budget. Further, it must be noted that the trained versions perform faster in absolute time for the same set of parameters, showing that priors help the search converge quickly, reducing the exploration of new states. Third, it can be observed that even though Trained agents were trained on specifically this boundary condition, MT agents outperform them at low search budgets. Since MT agents learn over more diverse

boundary conditions, they generalize the process of design generation much better. Fourth, the Untrained agent can perform at the level of trained variants at very high search budgets, and the impact of priors is diminished when extensive computational resources are used.

These results illustrate two main findings. First, since training on self-generated data enables the agent to outperform the untrained agent, it illustrates a policy improvement behavior meaning that the search agent can leverage old strategies to discover new and
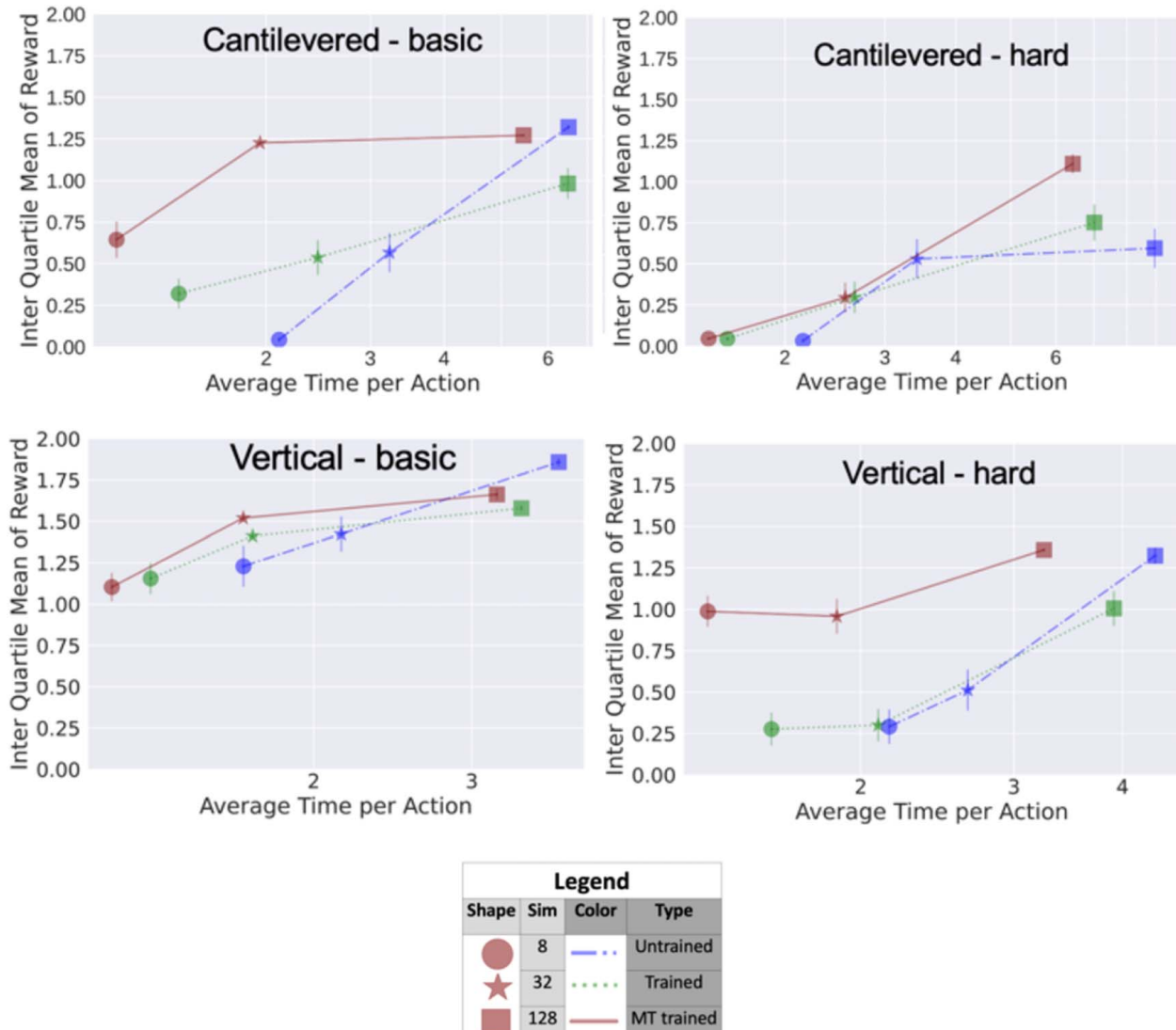

Fig. 6 Performance efficiency plots across different boundary conditions. All samples have a search depth of 5.

improved ones. This effect is more pronounced at low compute budgets and diminishes at high compute. Second, the deep learning-guided search combination provides a versatile framework for high-performance design generation. Without any human guidance, it can achieve superhuman performance and enable variations by controlling the search budget. Finding the global optimal is challenging. However, the best design performance for humans was at a reward value of 1.45; in comparison, the Trained agents reached a maximum of 1.81, which is about 25% more than the best human performance.

4.6 Results 2: Performance Across Unseen Boundary Conditions. In this section, the three agent variants are compared across unseen boundary conditions. These involve some "basic" boundary conditions used to train the MT version; however, the "hard" conditions are unseen for every variant. The y- and x-axis, and the legend are identical to the corresponding values in Fig. 4, and only three values of search parameters corresponding to different compute budgets are compared. Each data point on the plot represents the IQM value of 48 runs and standard errors are shown for every data sample.

Figure 5 shows the performance comparison on the "Middle-hard" condition. First, the trained variants perform significantly better than untrained variants at lower compute levels. MT variant can outperform both Trained and Untrained at all levels of performance. Second, it must be noted that the untrained agent outperforms the Trained variant at high compute. Since Trained variant was only trained on "Middle-basic," this result illustrates overfitting or ineffective transfer of strategies to this unseen problem. Since MT agents are trained on three "basic" conditions, they generalize better and perform much higher even on an unseen task. Figure 5 also shows the performance of human-data trained agents, GD-DLA and DLA were trained on "Middle—basic" and evaluated on the "Middle—hard." GD-DLA can generalize better to this boundary condition as it has a one-step search enabling the agent to adjust decision-making based on new goals and boundary conditions; however, they are still outperformed by the SLDA framework.

The rest of the boundary conditions are shown in Fig. 6. Overall, the agents maintain their performance trends across these problems;

at low compute the trained agents perform much better, however, with high compute their performances converge. The untrained agent's performance curve can be used as a benchmark to determine how challenging the problem is. A major observation here is that the training is especially helpful with "hard" conditions, showing that it is challenging to solve those problems with random priors or no training. For example, MT agents can solve the "Cantilevered-Hard" problem with high compute while the Untrained variant is unable to reach high-performance values even with high compute. This shows the complexity of the boundary condition and also illustrates the usefulness of training for a complex problem as the effect is more pronounced. This effectiveness of training is seen across all "hard" problems as the MT variant performs better or equal to trained across almost all compute levels. On the other hand, even a low search budget Untrained agent can solve "Vertical-Basic" considerably well, showing that this condition is easier to solve. In this scenario, the Untrained agent performs better than the Trained agents at a high-compute level which is contrary to other observations. This hints that training may lead to creating biased designs where the generated designs are much more complex than they need to be, especially for a simpler boundary condition. The human-data-trained agents are not compared in these conditions due to their low performance in these unseen new boundary conditions. These previous agents were not trained to accommodate different loading and support conditions and could only be applied to middle-loading boundary conditions.

Understanding the *why* and *how* of the dominant performance of the MT variant is a potentially exciting direction for future research. The agent has implicitly learned how to "grow" a design across a variety of support and loading points. The learned strategy to iteratively grow the truss structure may include several insights about truss design as they perform better than human data; however, a detailed behavioral analysis may be carried out in the future to explore that.

4.7 Discussion. This section presents some visual results to enable a qualitative analysis and discussion of the agent's performance. Figure 7 provides several examples of the final design achieved by MT and Untrained agents. The numerical values in
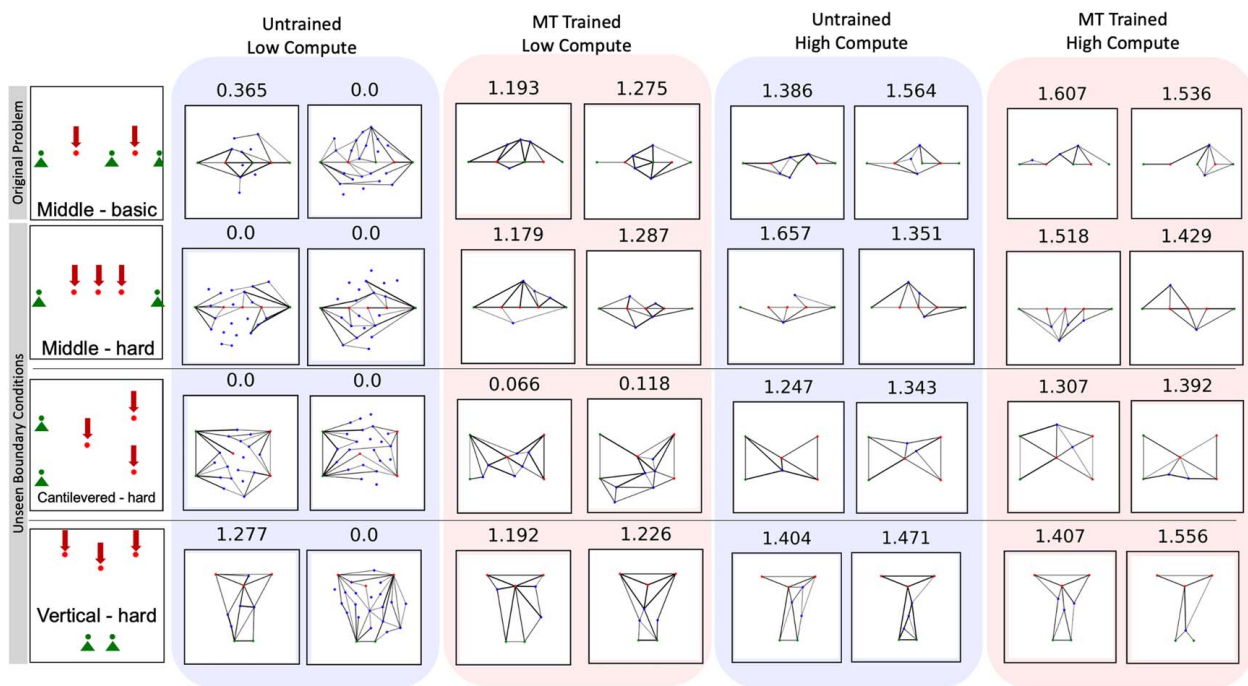

Fig. 7 Examples of final designs generated by the agent with different training and compute parameters. Here, red circles represent loading nodes, green represent support nodes, and blue represent added nodes.

the images are the reward value associated with the designs in the range [0, 2], with the higher value corresponding to better quality. A value less than 1.0 corresponds to an infeasible design and hence a failure of the design process. A 0.0 value corresponds to an unsupported design for which the quality metric does not exist. Intermediate values (0–1.0) refer to correctly supported structures that are incomplete or infeasible.

The figure compares the agent's high and low compute versions for the original problem and the unseen Boundary Conditions. High-compute refers to Sim 128 Depth 5 and Low-Compute refers to Sim 8 and Depth 5. It first illustrates the ability of the agent to create diverse designs. It must be noted that the MT agent has a common set of weights for its DSN, which are used across these boundary conditions to generate appropriate high-performing truss designs.

Three significant observations can be made. First, the high compute variants of both Untrained and MT agents perform very similarly. The truss structures designed are well triangulated and have similar reward values. This similarity illustrates the effective performance of the SGTS algorithm with and without DSN priors across multiple boundary conditions. Second, a significant difference can be observed in structures when comparing the low-compute variants. Untrained SLDA structures have unsupported nodes and unconnected members. The unsupported nodes are expected with low compute as the agent behaves randomly. In comparison, the MT agent performs significantly well. It designs fully connected and well-supported structures across different conditions. The complete connectedness provides evidence that the agent has learned certain underlying design principles for truss structures such that it can apply them across multiple problems without a deep search. The third observation even more strongly supports this claim. Comparing the low-compute designs of the Cantilevered-hard problem, both Untrained and MT agent variants cannot create a feasible structure; however, their failure manner is different. All the nodes generated by the MT agent variant are properly supported, and the members are also well triangulated, whereas the Untrained variant appears to be generating randomly.

From visual observation, it can be concluded that the MT agent has learned the principle of triangulation which is one of the essential generative grammars for the development of trusses [67,68]. Triangulation enables stable and well-supported designs and visually illustrates high-performance. This analysis motivates further exploration of the generative strategies leveraged by these agents. These agents can potentially help uncover new strategies or best practices in different design problems, mainly because they perform better than humans. These agents can be utilized as knowledge generators that continuously explore a given problem space and gather information. Combined with the representation methods presented in this work, the SLDAgents can work as forever learning design agents, iteratively improving upon their understanding of design knowledge indefinitely.

The SLDA method applies generally to any problem that can be described as a Markov Decision Process with information about state, action, and rewards. The approach was used to solve another challenging problem in engineering design, circuit routing
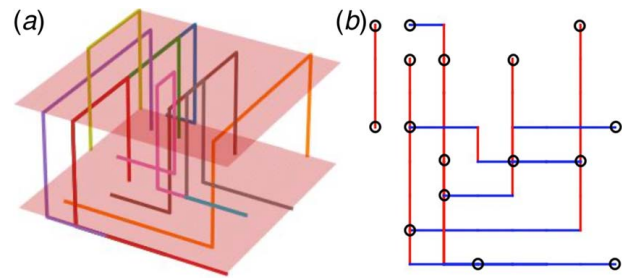

Fig. 8 Visual representations of a completed circuit layout problem with ten-pins: (a) an isometric view and (b) top view with pin-pairs shown as circles

[28] with equally effective results [69]. The self-trained agents were able to significantly outperform a standard heuristic-based optimizer across multiple scenarios. The trained SLDA demonstrate a general routing strategy that could effectively solve diverse problems without any retraining. This result highlights the effectiveness of the dual-process framework and its ability to transfer strategies across scenarios, generating designs in a generalizable manner.

5 Case Study 2: Circuit Routing Problem

5.1 Problem Definition. Circuit routing is a standard problem in electronics that is part of designing an Integrated Circuit (IC) chip. An IC is composed of multiple electronic components connected using a wiring pattern. The components are placed on a 2D gridspace while the wires can be routed through the gridspace as well as a pass-over plane as shown in Fig. 8(a). Figure 8(b) shows the top-view of the same pattern where components are marked as circles. This gridspace is discretized into blocks or cells and every cell has an associated capacity that determines the maximum number of wire connections that can pass through that space. The goal of this problem is to connect the components using wires such that the total *wire length* (WL) of routing is minimized and no cell has a capacity overflow (OF), caused by multiple wires passing through the same gridspace. The problem's more detailed definition is provided in previous work that decomposes it as a Markov decision process to implement a reinforcement learning-based agent [28]. In both that work and the current work, circuit routing is idealized as the connection of components in a pre-defined order. We do modify the State, Action, and Reward formulation for our proposed agent, a summary of which is provided in Table 3. Future work can also address the biasness of sequentiality by either learning to order the components to route or altering the action set to enable concurrent routing.

The state space combines global capacity, location vector, and a progress fraction to provide holistic information for the agent to make decisions. The global capacity is represented as a three-dimensional vector $g \times g \times 6$, where g is the size of gridspace and six corresponds to the spatial directions the agent can move in ($\pm x$, $\pm y$, and $\pm z$). The global capacity matrix represents the

Table 3 MDP Formulation for the routing problem

| State	Combination of:
	(1) Global capacity image (grid size $\times$ grid size $\times$ 6 pixels)
	(2) Vector for agent location (x, y, z), distance from goal (x, y, z), local capacity in six spatial dimensions
	(3) Progress ratio (routed connections/total connections)
Action	Action type Parameters
	Spatial direction (0–6) Distance (0—grid size)
Probability	$P = 1$, a deterministic process
Reward	$\text{Reward}(a, s'_t) = \begin{cases} 100 - t & \text{if } s'_t \text{ is the target pin} \\ -t & \text{if } t \text{ is max iteration} \\ 0 & \text{otherwise} \end{cases}$
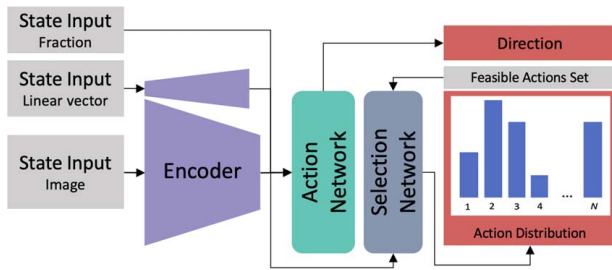
Fig. 9 Policy network architecture for circuit routing

number of wires that can be added to any cell on the gridspace. The location vector is 12-dimensional and constitutes agent location (x, y, z) the distance to the goal (x, y, z), and immediate capacity in the six spatial directions. This vector is identical to the state representation used by Liao et al. [28]. Finally, to represent the progress of the overall routing operation, a progress fraction is introduced. The ratio of connections made to the total number of connections to be routed is used as state input. We include this additional information in the neural networks such that the agent can make more informed long-term decisions. This work also updates the action space representation from one-step actions in the six-spatial directions as initially defined in Ref. [28] to a compound action composed of *direction* and *distance,* as shown in Table 3. With this formulation, the agent can make long strides in a particular direction when routing, reducing the number of actions required to complete routing.

The Boundary Conditions in routing are the different components that need to be routed and their locations on the gridspace. The training environment generates random locations of these components for a given problem. This research considers two sets of problems: a simple ten-pin problem and a more complex 50-pin problem, where pins refer to the components that need to be connected. Both problems have grid size (g), 8; max number of pins in each connection, 2; capacity, 5. Given both problems have the same cell capacities, a higher number of components require careful routing of wires to avoid overflow and to ensure optimal utilization of capacity.

5.2 Policy Network Architecture. The policy network is based on the original DSN architecture, and minor adjustments are made to suit the dimensioning of the problem. The overall schematic of the network is shown in Fig. 9. Encoder network has three inputs: global capacity, state vector, and progress ratio. Since global capacity is a 2D image with six channels, it is transformed using a convolutional encoder. It has a *coordconv* layer followed by two layers of convolutions, all with kernel size three and stride 1. These layers reduce the image to a 512-unit latent vector. The state-vector is a 1D (12 unit) vector fed into a linear layer that transforms it into 64 units. Finally, the 512-unit latent vector, 64-unit state vector, and progress ratio are concatenated and passed through the final linear layer which transforms it into a 512-dimensional latent vector. The action

network employs this combined representation to output the direction of the action. However, this output is not used for decision-making as the action space is finite and discretized so a prior-based sampling approach is not critical. The order-invariant selection network takes the latent representation of the State Input and the set of feasible actions as input to generate the final probability distribution over actions. The selection network transforms the actions into a latent representation by breaking them into two components, direction and distance. The direction input passes through three layers of transformation (64, 128, 256 units), and the distance input passes through a single layer with 64 units. The output is combined before another linear layer and is finally concatenated with the State's latent representation. Finally, the combined state-action representation passes through two linear layers (128, 32 units) before generating a SoftMax probability over the feasible actions. Overall, the input to the DSN for routing is a combination of global and local capacity, location, goal, and progress, and the output is a probability distribution over a feasible complex action set.

5.3 Training Process. The 10-pin problem is used as the primary problem to generate routing strategies and use that to train the policy network of the agent. Four random boundary conditions are generated for the 10-pin problem, and an untrained SLDA with Simulations: 512, Depth: 25 is used to generate data. For each boundary condition, three runs are carried out, generating 12 trajectories of sequentially generating routing patterns for the 10-pin problem. Other search parameters are determined after several experiments and kept consistent across the runs. The balance between exploration and exploitation (β) is set at 50 as the reward values can reach high values (much higher than standard 0–1 range). The discount factor (γ) is fixed at 0.5 to balance between short-term rewards and long-term planning, favoring short-term rewards. Specific to the routing problem formulation, there are intermediate rewards that the agent may ignore with high discount factors.

The training process is similar to DSN with trusses where two target outputs are used to train the network. Here, the Action network predicts the direction (or class) of the final action and the Selection network generates the final action distribution. Both these outputs are discrete, and only BCE loss is used for training. The data from 4 10-pin boundary conditions are split into 90–10 training and testing data. The DSN is then trained to predict the final action selection of the SLDA agent as well as the direction of action from the Action network. The training on direction improves the testing accuracy as it acts as a skip connection providing more information to the encoder. Testing accuracy of 49% was achieved for predicting the final action on the small dataset. For comparison, random guessing can only achieve 8.5% accuracy.

5.4 Experimental Setup. Following the structure from truss design experiments, two experiments are defined to evaluate SLDA performance in routing. Two different problem sets are developed. A primary problem is developed with 10-pins on an

Table 4 Comparing wire length (WL) performance for routing solution on 10-pin boundary conditions (BC)

Algorithm	10 Pin BC							
	BC1		BC2		BC3		BC4	
A*(WL | OF)	86	0	73	0	77	0	55	0
DQN with A*	**86**		**73**		**77**		**55**	
	(0%)		(0%)		(0%)		(0%)	
Untrained SLDA	91		76		90		62	
(Sim: 512, Depth: 25)	(+5.81%)		(+4.11%)		(+16.88%)		(+12.73%)	
Trained SLDA	**86**		77		**77**		57	
(Sim: 512, Depth: 5)	(0%)		(+5.48%)		(0%)		(+3.64%)	

Note: Bold numbers indicate values better or equal to A* performance.

Table 5   Comparing wire length (WL) performance for routing solution on 50-pin boundary conditions (BC)

Algorithm	50 Pin BCs (unseen)							
	BC1	BC2	BC3	BC4	BC5	BC6	BC7	BC8
A* (WL | O.F.)	– 1	359 0	– 5	373 0	352 0	353 0	370 0	421 0
DQN w A*	**319**	**329**	**368**	**373**	**334**	**347**	**354**	**380**
(Trained for each BC)	(−1.85%)	(−8.36%)	(−4.17%)	(0%)	(−5.11%)	− (1.7%)	(−4.32%)	(−9.74%)
Untrained SLDA	364	407	–	–	–	–	–	–
(Sim: 512, Depth: 5)	(+12%)	(+13.37%)						
Untrained SLDA	364	398	–	444	–	387	412	437
(Sim: 512, Depth: 10)	(+12%)	(+10.86%)		(+19.03%)		(+9.63%)	(+11.35%)	(+3.8%)
Trained SLDA	**335**	**341**	**386**	**367**	**350**	**350**	**354**	**399**
(Sim: 512, Depth:5)	(+3.08%)	(−5.01%)	(+0.52%)	(−1.6%)	(−0.57%)	(−0.85%)	(−4.32%)	(−5.22)

Note: Bold numbers indicate values better or equal to A* performance.

8,8 grid, with a maximum capacity of 5. Another set with the same parameters but 50-pins is developed as a more challenging version of the routing problem. Experiment 1 compares the performance of the Trained with the Untrained agent on the original problem used for training. Experiment 2 evaluates the generalizability of these agents on the routing problem by evaluating it across different boundary conditions and when extending to a more challenging unseen problem.

Moreover, to benchmark the SLDA's performance, it is compared with A* [70] and a Deep Q-Network (DQN) formulation that combined A* with reinforcement learning previously used by Liao et al. [28]. A* is a heuristic-based search algorithm for path-planning problems. For routing, it decomposes the problem into multiple two-pin problems and then connects the components sequentially. Since the A* is a heuristic search-based process, it can generally be applied to the different conditions in the routing problem without any modification and is the main benchmarking comparison for SLDA. The goal of the DQN agent in previous work [28] was to enhance the performance of the A* algorithm by enabling learning by reinforcement. The agent is trained to route these problems by interacting with the environment and training over the generated experience. These agents are trained for 10,000 episodes for every boundary condition and were previously seen to improve over the A* algorithm significantly after training. Although there are newer reinforcement learning approaches that may outperform DQN, it acts as a strong baseline and illustrates the effectiveness of training for every single boundary condition. DQN's performance builds upon A* and acts as an upper bound for every boundary condition as it has been trained specifically for them.

5.5 Results: Trained SLDA Agent on Routing Problem.

In this section, two variants of the SLDA agent (trained and untrained) are compared on the original four boundary conditions of the 10-pin scenario. This scenario is an example of a Type-1 (no-edge depletion) problem where the grid space has large capacities, and instances of overflow (OF) are unlikely. Table 4 compares the WL of the final routing solution between SLDA variants, A* solver, and DQN with A* guidance [28]. A lower WL value corresponds to better performance. Table 4 includes the percentage difference as compared to A* benchmark results. Also, the instances where the agent is being evaluated on an unseen boundary condition are shaded gray. It must be noted that the DQN is trained as per previous work for every single boundary condition individually while the trained SLDA has a common set of weights across all comparisons and no retraining is required.

It can be observed that A* and DQN reach the best solutions for the 10-pin problem. The untrained agent with high search depth (Sim: 512, Depth: 25) is within a 20% range of these values. After training, SLDA self-discovers good routing strategies outperforming the untrained version with much less computing (Depth 5 versus 25). Trained SLDA matched the A* performance 50% of the

time and are within 5% WL the other two times. It must be noted that the DQN is trained for each condition individually while SLDA trains across these problems with only one set of policy network weights. Moreover, the results represent the best performance in 10k episodes for DQN, while for SLDA, we compare the best WL across three runs. Liao et al. [59] noted that the advantage of reinforcement learning-based approaches could not be observed in Type-1 problems since A* most certainly reaches high-performing results in simpler low pin number layouts. Section 5.6 evaluates the more challenging problem with more pins and denser layouts.

5.6 Results 2: Performance Across Multiple Boundary Conditions.

This section evaluates the generalizability of the trained SLDA agent across unseen problems. The same trained SLDA agent from the previous experiment is used to test its effectiveness on the more challenging 50-pin scenarios. Table 5 presents the comparison of the 50-pin boundary conditions. Bolded values represent performance greater than the A* benchmark. The following observations can be made: First, A* fails to generate a feasible solution for Scenarios 1 and 3, which both have a non-zero overflow (OF). Second, the trained SLDA outperforms A* in 100% of boundary conditions considering OF, presenting an improved general solver algorithm for circuit routing in these conditions. However, the individually trained DQN agent has the best performance. The Trained SLDA agent can reach within 5% of the DQN performance without any retraining. Given that these are all unseen problems, it shows high-performance generalizing behavior by SLDA. Third, the Untrained agents illustrate the difficulty of solving these problems from scratch, and without heuristics, most of them cannot create feasible solutions (marked as a dash). These results show high-performance behavior by the trained agent across multiple boundary conditions, demonstrating the generalizable behavior learned by the dual-process process agent framework.

6   Conclusions

A generalizable agent framework that can extract and learn design generation strategies from scratch is presented in this paper. The agent framework is inspired by fast and slow thinking processes in humans and applies it to design applications. A deep learning-based deep hierarchical policy network achieves fast and intuitive decision-making that can represent and emulate strategies in complex action spaces. A novel Sampling Guided Tree Search (SGTS) algorithm models the slow decision-making algorithm that can explore complex action spaces effectively. The combined dual-process framework is termed SLDA, which can work with *standardized* design problems by *discovering*, *representing*, and *transferring* generative strategies from scratch. The agent is evaluated on multiple variants of the truss design and circuit routing problems. It illustrates an ability to discover novel design strategies from scratch given only the feasible actions in the design task and

rewards for the state. It also demonstrates a high-performing generalizable behavior over multiple variants within a problem. Finally, applying this framework over trusses and circuit routing illustrates its applicability to diverse problem domains to extract and learn design decision-making behavior by reinforcement.

The main contributions of the work are three-fold. First, it creates a novel framework that can effectively discover and learn design strategies without prior data. Second, the work illustrates the impact of using data-driven design strategies to reduce the computation time while achieving high-level performance in a design problem. Third, the agent demonstrates an ability to solve a myriad of unseen design problems by generalizing the sequential design generation process itself, instead of learning final design solutions. Overall, the agent framework presents a novel contribution to the design automation literature since it achieves learning generalizable strategies from scratch in challenging engineering design problems.

Even though SLDA agents illustrate high-performing behavior with low search budgets, MCTS is inherently compute-intensive. The DL-based filtering of search spaces promotes scalability by reducing the compute requirements for effective tree search. Further, the SLDA framework is easily extended to arbitrarily complex action space representations due to an order-invariant network architecture. Future work may be aimed at exploring the limits of this method with scaled-up search spaces. Moreover, the search algorithm can be augmented to simplify continuous actions and avoid loops in sequential actions. Further work toward continually iterating and learning improved policies can also enable more advanced performance levels. Finally, these agents can be used in real-time guidance for human-machine teaming activities since these perform at significantly higher-performance levels than humans with similar compute times.

Acknowledgment

Conflict of Interest

There are no conflicts of interest.

Data Availability Statement

The data and information that support the findings of this article are freely available online.[6]

References

[1] Brown, D. C., and Chandrasekaran, B., 2014, *Design Problem Solving: Knowledge Structures and Control Strategies*, Morgan Kaufmann, Elsevier Science, Burlington, MA.

[2] Simon, H. A., 2019, *The Sciences of the Artificial*, MIT Press, Cambridge, MA.

[3] Chakrabarti, A., Shea, K., Stone, R., Cagan, J., Campbell, M., Hernandez, N. V., and Wood, K. L., 2011, "Computer-Based Design Synthesis Research: An Overview," ASME J. Comput. Inf. Sci. Eng., 11(2), p. 021003.

[4] Regenwetter, L., Nobari, A. H., and Ahmed, F., 2022, "Deep Generative Models in Engineering Design: A Review," ASME J. Mech. Des., 144(7), p. 071704.

[5] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., et al., 2020, "Mastering Atari, Go, Chess and Shogi by Planning With a Learned Model," Nature, 588(7839), pp. 604–609.

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., et al., 2015, "Human-Level Control Through Deep Reinforcement Learning," Nature, 518(7540), pp. 529–533.

[7] Brown, N., and Sandholm, T., 2018, "Superhuman AI for Heads-Up No-Limit Poker: Libratus Beats Top Professionals," Science, 359(6374), pp. 418–424.

[8] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., et al., 2019, "Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning," Nature, 575(7782), pp. 350–354.

[9] Kahneman, Daniel, 2011, *Thinking Fast and Slow*, Farrar, Straus and Giroux, New York.

[10] Graepel, T., 2016, "AlphaGo—Mastering the Game of Go With Deep Neural Networks and Tree Search," Lecture Notes Computer Science, 9852 LNAI, p. XXI.

[11] Anthony, T., Tian, Z., and Barber, D., 2017, "Thinking Fast and Slow with Deep Learning and Tree Search," Adv. Neural Inf. Process. Syst., 2017, pp. 5361–5371.

[12] Li, Z., Chen, Q., and Koltun, V., 2018, "Combinatorial Optimization With Graph Convolutional Networks and Guided Tree Search," Advances in Neural Information Processing Systems, Montreal, Canada, Sept. 8, pp. 539–548.

[13] Gaymann, A., and Montomoli, F., 2019, "Deep Neural Network and Monte Carlo Tree Search Applied to Fluid-Structure Topology Optimization," Sci. Rep., 9(1), p. 15916.

[14] Raina, A., Cagan, J., and McComb, C., 2022, "Design Strategy Network: A Deep Hierarchical Framework to Represent Generative Design Strategies in Complex Action Spaces," ASME J. Mech. Des., 144(2), p. 021404.

[15] Hubert, T., Schrittwieser, J., Antonoglou, I., Barekatain, M., Schmitt, S., and Silver, D., 2021, "Learning and planning in complex action spaces," International Conference on Machine Learning, Virtual online, July 18.

[16] Hausknecht, M., and Stone, P., 2016, "Deep Reinforcement Learning in Parameterized Action Space," The 4th International Conference on Learning Representations, ICLR 2016—Conference Track Proceedings

[17] Lee, K., Kim, S. A., Choi, J., and Lee, S. W., 2018, "Deep Reinforcement Learning in Continuous Action Spaces: A Case Study in the Game of Simulated Curling," Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10, J. Dy and A. Krause, eds., PMLR, pp. 4587–4596.

[18] Cagan, J., and Mitchell, W. J., 1993, "Optimally Directed Shape Generation by Shape Annealing," Environ. Plan. B Plan. Des., 20(1), pp. 5–12.

[19] Renner, G., and Ekárt, A., 2003, "Genetic Algorithms in Computer Aided Design," CAD Comput. Aided Des., 35(8 SPEC.), pp. 709–726.

[20] Campbell, M. I., Rai, R., and Kurtoglu, T., 2012, "A Stochastic Tree-Search Algorithm for Generative Grammars," ASME J. Comput. Inf. Sci. Eng., 12(3), p. 031006.

[21] Kumar, M., Campbell, M. I., Königseder, C., and Shea, K., 2012, "Rule Based Stochastic Tree Search," Design Computing and Cognition, Houston, TX, June 5, Springer Netherlands, Dordrecht, pp. 571–587.

[22] Short, A.-R., DuPont, B. L., and Campbell, M. I., 2018, "A comparison of tree search methods for graph topology design problems," Design Computing and Cognition, Milan, Italy, July 2, Springer International Publishing, Cham, pp. 75–94.

[23] Short, A. R., Dutta, P., Gorr, B., Bedrosian, L., and Selva, D., 2022, "Representing and Analyzing Sequential Satellite Mission Design Decisions Through Anisomorphic Trees and Directed Graphs," AIAA Science and Technology Forum and Exposition, AIAA SciTech Forum 2022, San Diego, CA, Jan. 3.

[24] Chen, W., and Ahmed, F., 2021, "MO-PaDGAN: Reparameterizing Engineering Designs for Augmented Multi-Objective Optimization," Appl. Soft Comput., 113(A), p. 107909.

[25] Chen, W., Chiu, K., and Fuge, M. D., 2019, "Aerodynamic Design Optimization and Shape Exploration Using Generative Adversarial Networks," AIAA Scitech 2019 Forum, San Diego, CA, Jan. 7.

[26] Zhang, W., Yang, Z., Jiang, H., Nigam, S., Yamakawa, S., Furuhata, T., Shimada, K., and Kara, L. B., 2019, "3D Shape Synthesis for Conceptual Design and Optimization Using Variational Autoencoders," Proceedings of the ASME Design Engineering Technical Conference, 2A-2019, Anaheim, CA, Aug. 18.

[27] Ruiz-Montiel, M., Boned, J., Gavilanes, J., Jiménez, E., Mandow, L., and Pérez-De-La-Cruz, J. L., 2013, "Design With Shape Grammars and Reinforcement Learning," Adv. Eng. Informatics, 27(2), pp. 230–245.

[28] Liao, H., Zhang, W., Dong, X., Poczos, B., Shimada, K., and Burak Kara, L., 2020, "A Deep Reinforcement Learning Approach for Global Routing," ASME J. Mech. Des., 142(6), p. 061701.

[29] Lee, X. Y., Balu, A., Stoecklein, D., Ganapathysubramanian, B., and Sarkar, S., 2019, "A Case Study of Deep Reinforcement Learning for Engineering Design: Application to Microfluidic Devices for Flow Sculpting," ASME J. Mech. Des., 141(11), p. 111401.

[30] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S., 2012, "A Survey of Monte Carlo Tree Search Methods," IEEE Trans. Comput. Intell. AI Games, 4(1), pp. 1–43.

[31] Coulom, R., 2007, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), H. J. van den Herik, P. Ciancarini, and H. H. L. M. Jeroen Donkers, eds., Springer, Berlin/Heidelberg, pp. 72–83.

---

[5]https://github.com/haiguanl/DQN_GlobalRouting
[6]https://github.com/ayushraina10/SGTS

[32] Kocsis, L., and Szepesvári, C., 2006, "Bandit Based Monte-Carlo Planning," *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, eds., Springer, Berlin/Heidelberg, pp. 282–293.

[33] Świechowski, M., Godlewski, K., Sawicki, B., and Mańdziuk, J., 2022, "Monte Carlo Tree Search: A Review of Recent Modifications and Applications," Artif. Intell. Rev.

[34] Rosin, C. D., 2011, "Multi-Armed Bandits With Episode Context," Ann. Math. Artif. Intell., **61**(3), pp. 203–230.

[35] Chaslot, G., Bakkes, S., Szitai, I., and Spronck, P., 2008, "Monte-Carlo Tree Search: A New Framework for Game AI1," Proceedings of the Belgian/Netherlands Artificial Intelligence Conference, Belgium, Oct. 15, pp. 389–390.

[36] Zhu, J., Johnson, B., Bangalore, P., Huddleston, D., and Skjellum, A., 1998, "On the Parallelization of CH3D," International Water Resources Engineering Conference—Proceedings, Memphis, TN, Aug. 3, pp. 1108–1113.

[37] Liu, A., Chen, J., Yu, M., Zhai, Y., Zhou, X., and Liu, J., 2018, "Watch the Unobserved: A Simple Approach to Parallelizing Monte Carlo Tree Search," International Conference on Learning Representations, Addis Ababa, Ethiopia (virtual), Apr. 26, pp. 1–21.

[38] Cou, A., Hoock, J., Sokolovska, N., Teytaud, O., Cou, A., Hoock, J., Sokolovska, N., al., 2011, "Continuous Upper Confidence Trees To Cite This Version : HAL Id : Hal-00542673 Continuous Upper Confidence Trees," Proceedings of the 5th International Conference on Learning and Intelligent Optimization, Italy, January, pp. 433–445.

[39] Moerland, T. M., Broekens, J., Plaat, A., and Jonker, C. M., 2018, "A0C: Alpha Zero in Continuous Action Space,"

[40] Rybkin, O., Pertsch, K., Derpanis, K. G., Daniilidis, K., and Jaegle, A., 2019, "Learning What You Can Do Before Doing Anything," Proceedings of the 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, May.

[41] Hubert, T., Schrittwieser, J., Antonoglou, I., Barekatain, M., Schmitt, S., and Silver, D., 2021, "Learning and Planning in Complex Action Spaces," ArXiv, abs/2104.0.

[42] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., et al., 2017, "Mastering the Game of Go Without Human Knowledge," Nature, **550**(7676), pp. 354–359.

[43] Guo, X., Singh, S., Lee, H., Lewis, R., and Wang, X., 2014, "Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning," *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, eds., Curran Associates, Inc., Red Hook, NY, pp. 3338–3346.

[44] Schwarzschild, A., Borgnia, E., Gupta, A., Huang, F., Vishkin, U., Goldblum, M., and Goldstein, T., 2021, "Can You Learn an Algorithm? Generalizing from Easy to Hard Problems with Recurrent Networks," 35th Conference on Neural Information Processing Systems, Virtual only, December.

[45] Zhao, A., Xu, J., Konaković-Luković, M., Hughes, J., Spielberg, A., Rus, D., and Matusik, W., 2020, "RoboGrammar: Graph Grammar for Terrain-Optimized Robot Design," ACM Trans. Graph., **39**(6), pp. 1–16.

[46] Mirhoseini, A., Goldie, A., Yazgan, M., Jiang, J. W., Songhori, E., Wang, S., Lee, Y. J., et al., 2021, "A Graph Placement Methodology for Fast Chip Design," Nature, **594**(7862), pp. 207–212.

[47] He, Y., and Bao, F. S., 2020, "Circuit Routing Using Monte Carlo Tree Search and Deep Neural Networks," International Symposium on VLSI Design, Automation and Test (VLSI-DAT), Taiwan, April.

[48] Chang, K. W., Krishnamurthy, A., Agarwal, A., Daumé, H., and Langford, J., 2015, "Learning to Search Better Than Your Teacher," Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, JMLR.org, Lille, France, July, pp. 2058–2066.

[49] Grill, J. B., Altché, F., Tang, Y., Hubert, T., Valko, M., Antonoglou, I., and Munos, R., 2020, "Monte-Carlo Tree Search as Regularized Policy Optimization," Proceedings of the 37th International Conference on Machine Learning, ICML 2020, Vienna, Austria, July, PMLR, pp. 3727–3736.

[50] Ross, S., Gordon, G. J., and Bagnell, J. A., 2011, "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning," J. Mach. Learn. Res., **15**(1), pp. 627–635.

[51] Sutton, R. S., and Barto, A. G., 1998, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA.

[52] Raina, A., McComb, C., and Cagan, J., 2019, "Learning to Design From Humans: Imitating Human Designers Through Deep Learning," ASME J. Mech. Des., **141**(11), p. 111102.

[53] Raina, A., Puentes, L., Cagan, J., and McComb, C., 2021, "Goal-Directed Design Agents: Integrating Visual Imitation With One-Step Lookahead Optimization for Generative Design," ASME J. Mech. Des., **143**(12), p. 124501.

[54] Jacobsen, E. J., Greve, R., and Togelius, J., 2014, "Monte Mario: Platforming With MCTS," GECCO 2014—Proceedings of 2014 Genetic and Evolutionary Computation Conference, Vancouver, Canada, July, pp. 293–300.

[55] Keller, T., and Helmert, M., 2013, "Trial-Based Heuristic Tree Search for Finite Horizon MDPs," ICAPS 2013—Proceedings of the 23rd International Conference on Automobile Plan. Schedule, Rome, Italy, June, pp. 135–143.

[56] Sutton, R. S., and Barto, A. G., 2018, *Reinforcement Learning: An Introduction*, A Bradford Book, Cambridge, MA.

[57] Puentes, L., Raina, A., Cagan, J., and McComb, C., 2020, "Modeling a Strategic Human Engineering Design Process: Human-Inspired Heuristic Guidance Through Learned Visual Design Agents," Proceedings of the Design Society: DESIGN Conference, Croatia (online), June, pp. 355–364.

[58] McComb, C., Cagan, J., and Kotovsky, K., 2015, "Lifting the Veil: Drawing Insights About Design Teams From a Cognitively-Inspired Computational Model," Des. Stud., **40**(1), pp. 119–142.

[59] Wielinga, B., and Schreiber, G., 1997, "Configuration-Design Problem Solving," IEEE Expert. Syst. their Appl., **12**(2), pp. 49–56.

[60] McComb, C., Cagan, J., and Kotovsky, K., 2015, "Rolling With the Punches: An Examination of Team Performance in a Design Task Subject to Drastic Changes," Des. Stud., **36**(C), pp. 99–121.

[61] Spillers, W. R., and MacBain, K. M., 2009, "Some Basic Optimization Problems," *Structural Optimization*, K. M. MacBain, and W. R. Spillers, eds., Springer US, Boston, MA, pp. 103–137.

[62] Querin, O. M., Victoria, M., Alonso, C., Ansola, R., and Martí, P., 2017, "Topology Optimization as a Digital Design Tool," *Topology Design Methods for Structural Optimization*, O. M. Querin, M. Victoria, C. Alonso, R. Ansola, and P. Martí, eds., Academic Press, Oxford, pp. 93–111.

[63] Yu, T., Quillen, D., He, Z., Julian, R., Narayan, A., Shively, H., Bellathur, A., Hausman, K., Finn, C., and Levine, S., 2019, "Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning," CoRR, abs/1910.1.

[64] Liu, R., Lehman, J., Molino, P., Such, F. P., Frank, E., Sergeev, A., and Yosinski, J., 2018, "An Intriguing Failing of Convolutional Neural Networks and the CoordConv Solution," Adv. Neural Inf. Process. Syst., **2018**, pp. 9605–9616.

[65] Huang, S., and Ontañón, S., 2020, "A Closer Look at Invalid Action Masking inPolicy Gradient Algorithms," arXiv.

[66] Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A., and Bellemare, M. G., 2021, "Deep Reinforcement Learning at the Edge of the Statistical Precipice," Proceedings of the 35th Conference on Neural Information Processing Systems, Virtual only, December, pp. 1–27.

[67] Reddy, G., and Cagan, J., 1994, "An Improved Shape Annealing Method for Truss Topology Generation," Proceedings of ASME Design Engineering Technical Conference, Part F167892-1, Minneapolis, MN, September, pp. 331–341.

[68] Königseder, C., and Shea, K., 2014, "Systematic Rule Analysis of Generative Design Grammars," Artif. Intell. Eng. Des. Anal. Manuf. AIEDAM, **28**(3), pp. 227–238.

[69] Raina, A., 2022, *Towards Deep Learning Guided Search Agents for Sequentially Generative Design Problems*, Carnegie Mellon University ProQuest Dissertations Publishing, Pittsburgh, PA.

[70] Hart, P. E., Nilsson, N. J., and Raphael, B., 1968, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Trans. Syst. Sci. Cybern., **4**(2), pp. 100–107.

[71] Raina, A., Cagan, J., and McComb, C., 2022, "Self-Learning Design Agent (SLDA): Enabling Deep Learning and Tree Search in Complex Action Spaces," International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, St. Louis, MO, August.