The DeepXube Software Package for Solving Pathfinding Problems with Learned Heuristic Functions and Search

Forest Agostinelli

Department of Computer Science and Engineering
University of South Carolina, USA
foresta@cse.sc.edu

Abstract

DeepXube is a free and open-source Python package and command-line tool that seeks to automate the solution of pathfinding problems by using machine learning to learn heuristic functions that guide heuristic search algorithms tailored to deep neural networks (DNNs). DeepXube is comprised of the latest advances in deep reinforcement learning, heuristic search, and formal logic for solving pathfinding problems. This includes limited-horizon Bellman-based learning, hindsight experience replay, batched heuristic search, and specifying goals with answer-set programming. A robust multiple-inheritance structure simplifies the definition of pathfinding domains and the generation of training data. Training heuristic functions is made efficient through the automatic parallelization of the generation of training data across central processing units (CPUs) and reinforcement learning updates across graphics processing units (GPUs). Pathfinding algorithms that take advantage of the parallelism of GPUs and DNN architectures, such as batch weighted A* and Q* search and beam search are easily employed to solve pathfinding problems through command-line arguments. Finally, several convenient features for visualization, code profiling, and progress monitoring during training and solving are available.

Introduction

Pathfinding problems are found throughout mathematics, chemistry, robotics, and computing. Pathfinding algorithms that solve such problems include those based on human-designed algorithms, pattern databases (Korf 1997; Culberson and Schaeffer 1998), and heuristics derived from formal representations of pathfinding domains (Helmert 2006). In the past decade, it has been shown that representing heuristic functions with deep neural networks (DNNs) (Schmidhuber 2015; LeCun, Bengio, and Hinton 2015), training them with machine learning, and combining them with heuristic search can rival or outperform the aforementioned methods while assuming a black-box representation of the pathfinding domain and without relying on pre-existing solvers (Agostinelli et al. 2019; Chen et al. 2020; Muppasani et al. 2024; Agostinelli, Panta, and Khandelwal 2024; Chervov et al. 2025). These techniques have been applied to puzzle solving (Agostinelli et al. 2019; Chervov et al. 2025), chemical reaction mechanism pathfinding (Panta et al. 2024), quantum algorithm compiling (Zhang et al. 2020; Bao and Hartnett 2024; Qiuhao et al. 2024; Turner, Agostinelli, and Fu 2025), cryptography (Jin and Kim 2020), and parking lot optimization (Siddique, Gue, and Usher 2021).

DeepXube (pronounced "Deep Cube") is a free and open-source Python package that seeks to automate the solution of pathfinding problems given only a black-box implementation of a pathfinding domain and a heuristic function represented as a DNN. DeepXube is comprised of the latest advances in the combination of deep reinforcement learning, supervised learning, and heuristic search for pathfinding. These include batch weighted A* and Q* search (Agostinelli et al. 2024), limited-horizon Bellman-based learning (LHBL) (Hadar, Agostinelli, and Shperberg 2026), hindsight experience replay (Andrychowicz et al. 2017), supervised learning based on random walk path costs (Chervov et al. 2025), and specifying goals with answer set programming (Agostinelli, Panta, and Khandelwal 2024; Agostinelli 2025). Pathfinding domains are implemented with arbitrary Python code and heuristic functions are implemented with arbitrary PyTorch code (Paszke et al. 2019). Given an implementation of a pathfinding domain and heuristic function, DeepXube samples problem instances from the domain and uses pathfinding to generate training data for the heuristic function, as shown in Figure 1. Training is automatically parallelized across a user-specified number of central processing units (CPUs) and the DNN is parallelized across all visible graphics processing units (GPUs). Several methods to monitor progress are provided, including TensorBoard (Abadi et al. 2015), a pathfinding-specific training summary, periodic testing of solution quality on a test set, and a log file. After training, given pathfinding problems, DeepXube uses the trained heuristic function with heuristic search to solve problems and produces a file summarizing the solutions.

Preliminaries

Pathfinding

A **pathfinding domain** is defined as a weighted directed graph (Pohl 1970), where nodes represent states, edges represent actions that transition between states, and weights on

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

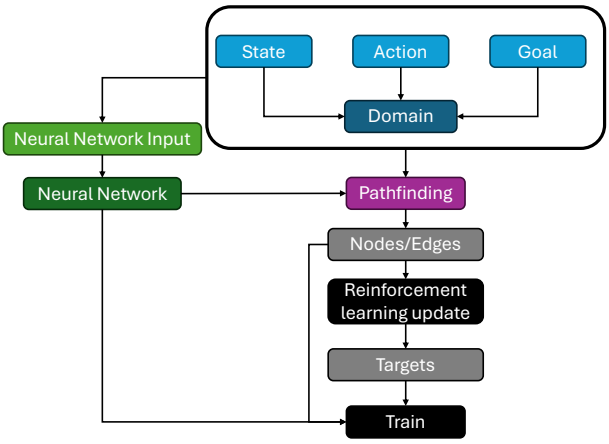

Figure 1: An overview of how DeepXube trains heuristic functions. The user implements the domain, along with its states, actions, and goals, as well as a DNN architecture that processes states, goals, and possibly actions. DeepXube then samples problem instances from the domain and uses the DNN to attempt to solve them with pathfinding, generating nodes and edges from the search tree. A reinforcement learning update is then applied to either the nodes or edges to compute targets. The DNN is then trained to match the targets.

the edges represent transition costs. The set of all possible outgoing edges of a state, s, which corresponds to the actions that can be taken in s, is denoted $E_o(s)$. The transition function is represented by T, where $s' = T(s, a)$ for some $a \in E_o(s)$. The transition cost function is represented by c, where $c(s, a)$ is the transition cost when taking action a in state s. The set of all possible actions is denoted $\mathcal{A}$. A **pathfinding problem instance** is defined by a tuple (D, s_0, g), where D is the pathfinding domain, s_0 is the start state, and g is the set of states considered goal states. Given a pathfinding problem, the objective is to find a path, which is a sequence of actions, that transforms the start state into a goal state while attempting to minimize the path cost, where the path cost is the sum of transition costs. The cost-to-go is the cost of a shortest path from a state, s, to a closest goal state in a goal, g, which we will denote $h^*(s, g)$.

Heuristic Search

Heuristic search is a widely used approach for solving pathfinding problems and is guided by a heuristic function that estimates the cost-to-go. Heuristic functions which operate on the nodes of the pathfinding domain graph map a state, s, and a goal, g, to an estimate $h(s, g) \approx h^*(s, g)$. Heuristic functions which operate on the edges of the pathfinding domain graph map a state, s, goal, g, and action, a, to an estimate $q(s, g, a) \approx c(s, a) + h^*(T(s, a), g)$. We will refer to heuristic functions which operate on nodes as heuristic-v functions, since they are often trained with value iteration (Bellman 1957), and heuristic functions which operate on edges as heuristic-q functions, since they are often trained with q-learning (Watkins and Dayan 1992). Heuristic search algorithms create a search tree, where nodes represent states and edges represent actions.

Graph Search Graph search algorithms in the context of heuristic search maintain a closed dictionary that maps states to their cheapest path cost observed so far, in that search. The most notable graph search algorithm that uses a heuristic-v function, A* search (Hart, Nilsson, and Raphael 1968), iteratively selects nodes from the search tree for expansion from a priority queue prioritized by the cost of the node, which is the sum of its path cost from the start node (i.e. cost-to-come computed by the sum of transition costs) and its heuristic value (i.e. cost-to-go computed by h). A node is expanded by applying every possible action to the state associated with that node and creating child nodes from the resulting states. The cost of each child node is computed and the child node is added to the priority queue if the state associated with the node is not in the closed dictionary or has been reached via a cheaper path. A* search terminates when a node associated with a goal state is selected for expansion and returns the path to that node.

Q* search (Agostinelli et al. 2024) is a variant of A* search that uses a heuristic-q function. Q* search iteratively selects edges for expansion and, in doing so, only generates nodes when removing an edge from the priority queue. As a result, the number of nodes generated at each iteration is independent of the number of available actions. Furthermore, some deep Q-network architectures can map a state, s, and a goal, g, to $q(s, g, a)$ for all possible actions, a (Mnih et al. 2015), thus computing the heuristic value for all edges with a single forward pass through a DNN.

Both A* and Q* search have batched and weighted variants referred to as batch weighted A* search (BWAS) and batch weighted Q* search (BWQS), respectively. For weighted search (Pohl 1970), the path cost is multiplied by a scalar, $\lambda \in [0, 1)$. This decreases the influence the path cost has on the overall cost and exchanges potentially faster search times for potentially greater solution costs. For batched search, $B > 1$ nodes or edges are removed from the priority queue each iteration. When heuristic functions are represented as DNNs, this exploits the parallelism provided by GPUs for DNNs.

Beam Search As opposed to graph search, beam search neither maintains a closed dictionary nor uses a priority queue. Beam search maintains a "beam" of nodes of a maximum size, B, starting with just the start node in the beam. At each iteration, beam search selects the top B edges for each node in the beam and transitions along those edges to produce the nodes in the beam for the next iteration. The metric to evaluate the edges could come from computing $-(c(s, a) + h(T(s, a), g))$ with a heuristic-v function, computing $-q(s, g, a)$ with a heuristic-q function, or computing the probability density for each action with a policy function, $\pi(a|s, g)$. Stochasticity can be added to the beam search by selecting edges according to a Boltzmann distribution with some given temperature, τ, and/or randomly selecting edges with some probability ϵ. One notable special case is the ϵ-greedy policy, often used in reinforcement learning, where $B = 1$ and the action taken is random with

probability ϵ and greedy, otherwise (i.e. $\tau = 0$).

Learning Heuristic Functions

Reinforcement learning can be used to learn heuristic functions based only on a definition of the domain and a way to generate problem instances. A DNN used to represent a heuristic-v function, h_θ, with parameters, θ, can be trained using approximate value iteration (Bellman 1957; Bertsekas and Tsitsiklis 1996; Sutton and Barto 2018). The DNN is trained with gradient descent to minimize the loss function in Equation 1, where N denotes the batch size and θ denotes the parameters of the DNN. The value iteration update in the context of pathfinding is shown in Equation 2, where θ^- are the parameters of the target network (Mnih et al. 2015) which are periodically updated to θ.

$$L(\theta) = \frac{1}{N} \sum_{i}^{N} (h'(s_i, g_i) - h_\theta(s_i, g_i))^2 \tag{1}$$

$$h'(s, g) = \begin{cases} 0, & \text{if } s \in g, \\ \min\limits_{a \in E_o(s)} c(s, a) + h_{\theta^-}(T(s, a), g), & \text{ow.} \end{cases} \tag{2}$$

A DNN used to represent a heuristic-q function, q_θ, with parameters, θ, can be trained using approximate Q-learning (Watkins and Dayan 1992; Mnih et al. 2015). The DNN is trained with gradient descent to minimize the loss function in Equation 3. The q-learning update in the context of pathfinding is shown in Equation 4, where $s' = T(s, a)$.

$$L(\theta) = \frac{1}{N} \sum_{i}^{N} (q'(s_i, g_i, a_i) - q_\theta(s_i, g_i, a_i))^2 \tag{3}$$

$$q'(s, g, a) = \begin{cases} 0, & \text{if } s \in g, \\ c(s, a) + \min\limits_{a' \in E_o(s')} q_{\theta^-}(s', g, a'), & \text{ow.} \end{cases} \tag{4}$$

Defining Domains

To define a domain with DeepXube, the following methods must be implemented: sampling problem instances (`samp_prob_insts`), sampling a valid action in a given state (`samp_state_act`), the state transition and transition cost when taking a given action in a given state (`next_state`), and a goal test given a state and goal (`is_solved`). The benefit of assuming the domain definition is a black box is that its implementation can depend on arbitrary Python packages, such as RDKit for chemistry (Landrum et al. 2013), and goals can represent a set of states without having to know of any state that is a member of that set. For example, for chemical synthesis, states can be molecules and goals can be properties that a molecule should have, implicitly defining the set of molecules that have those properties. For quantum circuit synthesis, states could be quantum circuits and goals could be matrices representing quantum algorithms, implicitly defining the set of circuits that implement those algorithms (Turner, Agostinelli, and Fu 2025).

Mixins

DeepXube contains multiple subclasses of the `Domain`, which we refer to as "mixin" classes, that define additional functionality. Depending on the pathfinding or training algorithm used, the domain may need to subclass certain mixins. For example, A* search in an enumerable action space requires the domain to subclass the `ActsEnum` mixin from which the result of applying all possible actions to a state can be obtained with the `expand` method. Generating problem instances with random walks requires the domain to subclass the `GoalSampleableFromState` mixin from which goals can be obtained through the `samp_goal_from_state` method. The mixin classes also simplify domain definition by automatically implementing methods based on the implementation of simpler methods. For example, if the action space is enumerable and fixed, the child class can implement a single method that gets all possible actions and the mixin class implements `samp_state_act` as well as `expand`. A domain can inherit from multiple mixins and any automatically implemented method can be overridden, if the user deems it necessary.

While sampling problem instances may be implemented in any manner, DeepXube adds mixin classes that define ways of sampling problem instances that can work well in practice. For actions that can be reversed, such as the Rubik's cube, sliding tile puzzles, and certain quantum circuit synthesis tasks, sampling a goal state and corresponding goal and taking a random walk in reverse from the goal state has worked well in practice (Agostinelli et al. 2019; Zhang et al. 2020; Chervov et al. 2025). A reverse walk can also be done for irreversible action spaces given an implementation of a reverse walk, such as Sokoban where boxes are pulled instead of pushed. For action spaces that are not easily reversed, such as many robotics tasks and chemical reaction mechanisms, or domains where sampling a reachable goal is difficult, sampling a start state, taking a random walk, and sampling a goal from the terminal state has worked well in practice (Agostinelli, Panta, and Khandelwal 2024; Panta et al. 2024; Agostinelli and Soltani 2024). The mixin classes are summarized in Figure 2. Visualizations of problem instance generation mixins are shown in Figure 3.

Two mixins, `StateGoalVizable` and `StringToAct` are provided to visualize problem instances as well as use the terminal to apply actions to the states and visualize the resulting next states. The visualization is done via a `matplotlib Figure` object, so visualizations can be interactive, if need be. For example, a visualization for the Rubik's cube could allow the user to click and drag to rotate the cube.

Defining Neural Networks

DeepXube has strong integration with both heuristic-v and heuristic-q functions and preliminary integration with policy functions for continuous action spaces, such as those seen in robotics, or very large action spaces where enumeration is impractical, such as multi-agent path finding. To ensure the implemented DNN accepts the appropriate input,

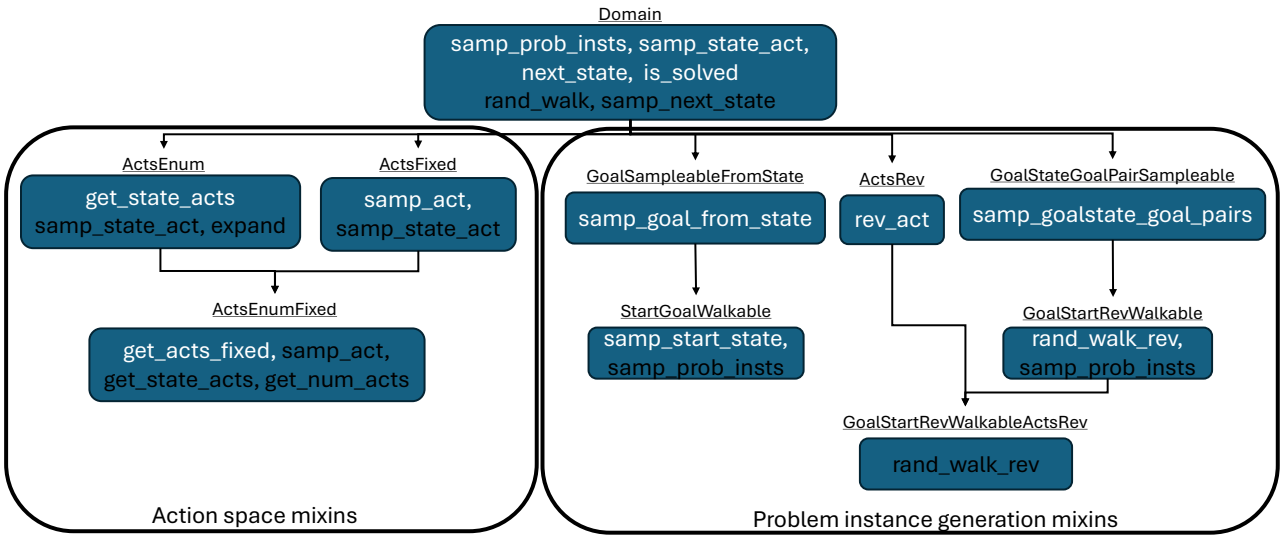

Figure 2: The primary mixin classes used to construct domains. Methods in white are abstract and methods in black are implemented by the class. Mixins inherit all functionality of their ancestors.

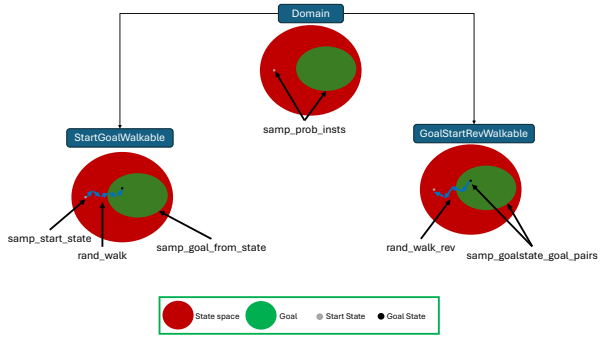

Figure 3: A visualization of the problem instance generation function along with two mixin classes for generating problem instances by either 1) sampling a start state, taking a random walk, and sampling a goal from the terminal state of that random walk; or 2) sampling a goal and a corresponding goal state, taking a random walk in reverse, and using the terminal state of that random walk as the start state.

the DNN class specifies what `NNetInput` class it requires. The `NNetInput` class converts the anticipated Python objects to a representation suitable for the DNN.

Heuristic-v functions take a state and goal as an input. Heuristic-q functions can either take a state, goal, and action as an input, or take a state and goal as an input and output the heuristic values for all possible edges with a single forward pass. Currently, this is only supported for a fixed action space; however, future updates plan to implement this for domains whose action spaces are dynamic and can be represented with a structured output, such as graphs (You et al. 2018). For policy functions, during training, the DNN is trained to match the distribution of edges seen during heuristic search, with the intuition that successful searches will be biased towards edges that lead to the goal (Mark et al. 2024). During inference, the policy network samples actions from its modeled distribution. However, the approach to learning a policy for pathfinding is preliminary and may be significantly updated in the future. A visualization of the neural networks that can be defined along with their inputs and outputs is shown in Figure 4.

There can be multiple `NNetInput` classes for different kinds of DNN architectures for a single domain and single kind of heuristic function. For example, a domain could have a `NNetInput` for a heuristic-v function represented as a convolutional neural network and a `NNetInput` for a heuristic-v function represented as a transformer (Vaswani et al. 2017). This allows architecture search for a single domain over multiple different kinds of architectures as well as multiple domains to use the same DNN architecture provided the appropriate `NNetInput` class is defined.

Pathfinding Algorithms

DeepXube has pathfinding algorithms that search over both nodes and edges, that operate on enumerable and infinite action spaces, and that perform beam search or graph search. Pathfinding algorithms specify what mixin class a domain must subclass as well as what functions it uses. For pathfinding algorithms that operate on enumerable action spaces, the domain must subclass `ActsEnum`. Otherwise, the pathfinding algorithms accept any subclass of `Domain` while also requiring a policy function to sample actions. For beam search, a temperature of 0 means that nodes are selected greedily unless ϵ is greater than 0. Graph search is also given an ϵ parameter where a random node or edge is popped from the priority queue with probability ϵ. This is primarily used for exploration during training. A summary of the pathfinding algorithms is shown in Figure 5. Note that, while the methods for training policy functions may change, the

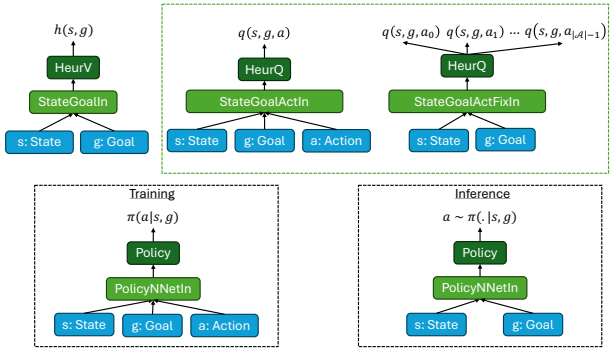

Figure 4: The interaction between states, goals, actions, neural network inputs, and heuristic/policy functions. Heuristic-q functions take an action as an input or output the q-values for all possible actions. A policy function takes an action as an input during training and samples actions during inference.

pathfinding algorithms that expect policy functions only assume the ability to sample actions from the policy function and, therefore, are agnostic to how the policy function is trained.

Learning Heuristic Functions

When learning heuristic functions, the generation of training data is parallelized across C CPUs and G GPUs and the heuristic function training is parallelized across G GPUs. CPUs sample problem instances, perform pathfinding, and compute the target for the DNN using either value iteration on the nodes or q-learning on the edges produced by pathfinding. The target network can also be used to guide the pathfinding algorithm; however, while this may work for some domains, this can result in a shift in the distribution of states seen when performing pathfinding with the trained network versus the target network. Therefore, there is also the option to use the network currently being trained to guide pathfinding. However, this requires synchronizing between the parallel processes and the main processes, which slows both down. Furthermore, this requires double the DNN applications since, when guiding pathfinding with a target network, the nodes and edges required for the reinforcement learning update have already been evaluated with the target network during pathfinding. An overview of the parallelization of training is shown in Figure 6. There is a special case of training where, instead of using reinforcement learning to compute targets, supervised learning is performed on the path costs of random walks. This has been shown to be effective at learning heuristic functions for puzzles such as the Rubik's cube (Chervov et al. 2025).

Reinforcement Learning

Problem instances are generated with a given parameter, K, where each problem instance is generated with a value, k, where k is uniformly distributed between 0 and K and k denotes the length of the random walk used to generate the problem. However, user-defined implementations of sampling problem instances can ignore k if deemed necessary. Every U iterations, there is a check to see if the parameters of the target network, θ^{-}, should be updated to the current parameters, θ. This check can always return true or be based on a threshold for the training loss. During the first U training iterations, the target network is a function that returns zero for all inputs.

Given a training batch size, N, $U \cdot N$ training data instances are generated per update check. Search with a given pathfinding algorithm is then performed for a maximum of I iterations[1]. All nodes expanded (for heuristic-v functions) or edges traversed (for heuristic-q functions) during search are used for training. Therefore, $\frac{U \cdot N}{I}$ problem instances are generated and, if a problem instance, i, generated with random walk length, k_i, is solved, then a new problem instance is generated in its place using the same k_i. Since a good value for K may not be known beforehand, DeepXube has the option to automatically adapt K during training by starting with $K = 1$ and doubling it when 50% of instances are solved. This is done until K reaches some given $K_{\max}$.

After search, the corresponding reinforcement learning update is performed on the expanded nodes (value iteration), if training a heuristic-v function or traversed edges (q-learning), if training a heuristic-q function. The user may choose to use LHBL (Hadar, Agostinelli, and Shperberg 2026), which recursively backs up the entire search tree before computing the target. LHBL has been shown to help learn heuristic functions that solve problems with fewer search iterations and better avoid depression regions (Aine et al. 2016). DeepXube has the ability to learn policy functions by matching the distribution of edges traversed during pathfinding. However, this is still in the preliminary stages and may change significantly in the future.

The user may choose to use hindsight experience replay (HER) (Andrychowicz et al. 2017) where, if the pathfinding algorithm fails to solve the problem in I iterations, HER selects a deepest node in the search tree to generate a goal. This ensures that every search produces an example of a reached goal, which can facilitate learning when the generated problem instances are all difficult to solve. This training method also requires the domain to subclass `GoalSampleableFromState`. A visualization of training without and with HER is shown in Figure 7. Lastly, a replay buffer may be used during training (Mnih et al. 2015), where the replay buffer stores the most recent nodes or edges along with their training targets from the past R update checks and samples from the replay buffer to generate training data.

Supervised Learning from Random Walks

Supervised learning from the path costs of random walks has also been shown capable of learning heuristic functions that perform well with large beam searches (Chervov et al. 2025). DeepXube can learn heuristic-v or heuristic-q functions with either random forward or reverse walks where the target is set to be the path cost of the random walk. One ben-

[1]Currently, the batch or beam size of the pathfinding algorithm used must be 1 during training.

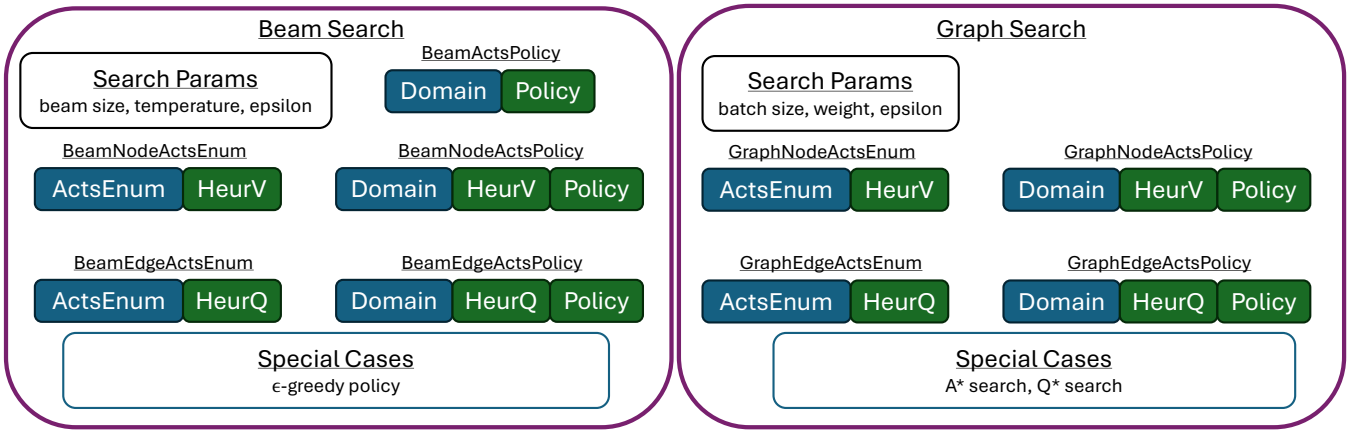

Figure 5: The available pathfinding algorithms. The mixin class that the domain must subclass is in blue and the functions used are in green. If both a heuristic and policy function are used, then the pathfinding algorithm is guided by the heuristic function and the policy function is used to sample actions.

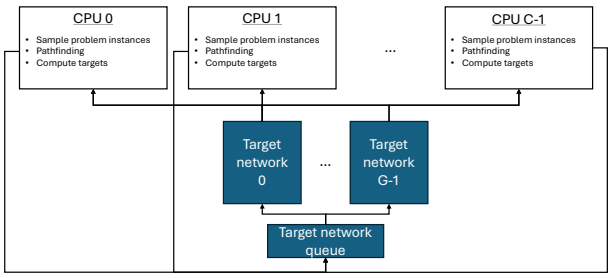

Figure 6: Training is parallelized across C CPUs and G GPUs. CPUs sample problem instances, perform heuristic search, and compute targets for training. When computing targets, the CPU sends the corresponding input data and its process ID to the target network queue, which sends the data to the first available target network. The output of the target network is obtained and sent to the correct CPU via the given process ID.

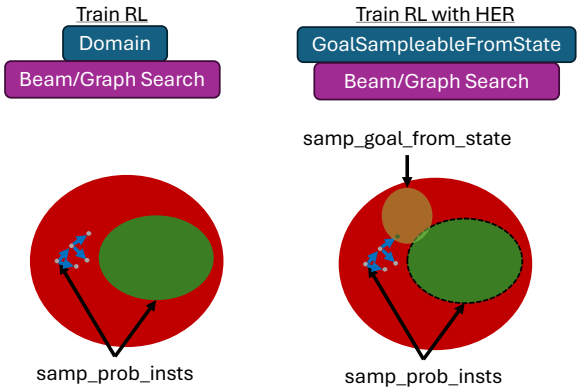

Figure 7: The two RL training approaches. For both approaches, problem instances are generated and the selected pathfinding algorithm is used to try to reach the goal from the start state. When using HER, if a path to the goal is not found, a goal is sampled from the state associated with a deepest node in the search tree and is used in place of the original goal.

efit to this approach is that no DNN is needed to compute targets and, therefore, targets can be obtained much faster. For heuristic-q functions learned with reverse walks, the domain must have reversible actions so, at the end of the reverse walk, the final edge on the reverse random walk that produces the start state can be reversed and used to update the heuristic-q function. DeepXube has the ability to learn policy functions by matching the distribution of edges traversed during the random walk. To function with the overall DeepXube architecture, these supervised learning approaches are implemented as pathfinding algorithms. A summary of these algorithms and their required mixin is shown in Figure 8.

Monitoring Progress

To monitor progress, TensorBoard can be used to plot the mean, minimum, and maximum values of the training targets, the training loss, the path cost and search iterations for solved problem instances and percentage of problem instances solved when generating training data using pathfinding as a function of training iteration. This pathfinding information is also plotted for test data, if a test dataset is given. DeepXube runs the given test pathfinding algorithms with the trained DNNs on the test dataset every given number of update checks.

Furthermore, a custom visualization can be accessed from the command line during and after training. This visualization shows the percentage of states solved, path costs, search iterations, the target cost-to-go, and the number of instances generated as a function of the random walk length used to generate problem instances. The visualization also shows a plot of the DNN predictions as a function of the target values. A slider can be used to see how these plots change as a function of training iteration, as shown in Figure 9.

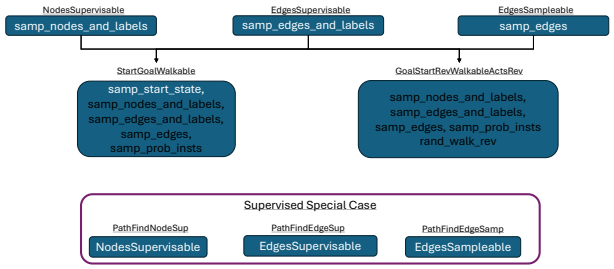

Figure 8: The mixins for supervised learning and the pathfinding algorithms along with their required mixin. Mixins from Figure 2 can be used to automatically implement the functionality of the mixins for supervised learning.

An output file is updated each update check that summarizes data generation, pathfinding performance, training performance, testing performance, and the time elapsed during execution of components. A verbose option is available that shows more detailed timing information for problem instance generation and pathfinding.

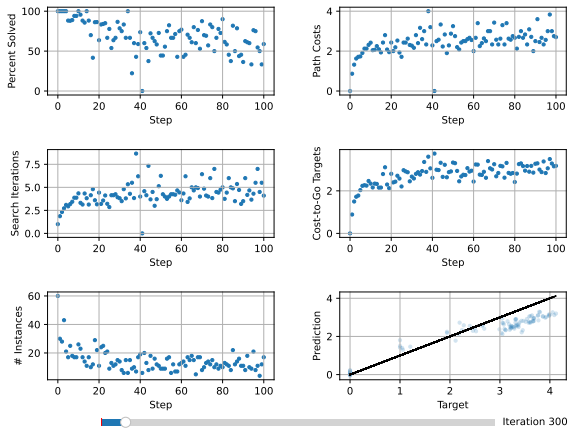

Figure 9: Visualization to monitor training progress.

Solving Pathfinding Problems

Given a trained heuristic and/or policy function, any of the aforementioned pathfinding algorithms in Figure 5 can be employed to solve problem instances. There is a verbose option for all pathfinding algorithms that prints the statistics of the path costs and heuristic values seen each iteration along with the frontier size (i.e. number of nodes in the beam or priority queue). DeepXube logs solutions found as well as important statistics such as the number of nodes generated, search iterations, and total search time in a Python pickle file. If the domain is a subclass of the `StateGoalVizable` mixin, then the pickle file can be used to visualize solutions found during pathfinding.

If the pathfinding algorithm expects a heuristic function and is not given one, then it creates a heuristic function that always returns zero. This can be a convenient way to obtain a baseline using uniform cost search. If a policy function is expected and not given, then it creates a policy function that randomly samples actions with the `samp_state_act` method.

Other Mixins

NumPy Mixins

For domains for which the next state method can be implemented in NumPy, DeepXube provides a `NextStateNP` mixin where the user defines how to convert states to NumPy, compute the next states in NumPy, and convert NumPy to states. Based on these methods, the mixin implements the next state method and overrides the random walk method so that this conversion problem only happens once instead of every time the next state function is called. If the action space is enumerable, then the `NextStateNPActsEnum` mixin overrides the expand function so that the conversion process only happens once.

Answer Set Programming Mixin

When representing states as a complete assignment of values to a set of variables, a goal can then be represented as a partial assignment of values to variables. Building on this, if an assignment is represented as a ground atom in first-order logic, then a goal can be represented with further abstraction by deriving logical predicates from these atoms. DeepXube provides the `GoalGrndAtoms` mixin where states are represented as complete assignments and goals are represented as partial assignments. Derived predicates can then be defined with this mixin to be used to create answer set programs (Brewka, Eiter, and Truszczyński 2011) that represent more abstract goals. Answer set solvers (Gebser et al. 2014) can then be used to find partial assignments that represent a set of goal states. This has been shown to allow for expressive goal specification (Agostinelli, Panta, and Khandelwal 2024) and allow for efficient goal reaching using answer set solvers and exploiting negation as failure through methods such as conflict-driven goal reaching (Agostinelli 2025).

PDDL Mixin

One may wish to compare solvers from DeepXube with other solvers based on the planning domain definition language (PDDL) (McDermott 2000), such as Fast Downward (Helmert 2006). The `SupportsPDDL` mixin allows the user to define how to convert the DeepXube domain to a PDDL domain, a problem instance to a PDDL problem instance, as well as convert a string representation of a PDDL action to the DeepXube Python action representation.

Convenience Features

Timing

When implementing a domain, one may wish to time the methods in the domain as well as set breakpoints for debugging. DeepXube provides a tool that, given a domain, runs basic functionality such as generating problem instances, sampling actions, getting next states, and goal tests. It also runs more specific functionality, such as expand, based on what mixins the domain subclasses. If given a particular

heuristic or policy DNN, it generates data to give to the DNN and times the execution. The user can set breakpoints anywhere in the domain or DNN code to help with debugging.

Problem Instance Generation

Given a domain, DeepXube can generate a given number of problem instances using a given minimum and maximum random walk length and save them to a file. If the domain allows for visualization, one can visualize the generated problem instances. One can also run uniform cost search, which requires no training, on the generated problem instances.

Command-Line Tool

Installing DeepXube also installs the `deepxube` command-line tool. Given an implementation of a domain and DNN, all the aforementioned functionality can be accessed using the command line. DeepXube uses a registry system of domains, neural network inputs, and DNNs. Neural network inputs are associated with a domain when registered whereas DNNs are not, allowing a single DNN architecture to be used across multiple domains. Using the registered name, domains and DNNs can be referenced using the command line. Furthermore, a parser can be registered for domains and DNNs to allow one to specify domain arguments or DNN hyperparameters via the command line. This can make for convenient hyperparameter search. Multiple neural network inputs can be defined and registered for a single domain. When given a domain and DNN architecture, DeepXube searches for its corresponding neural network input for that domain using the registry and uses the first one it finds. Pathfinding algorithms and their arguments can also be given via the command line. For example, `graph_q.10B_0.5W` performs BWQS with a batch size of 10 and weight of 0.5.

The `deepxube` command-line tool expects a positional argument that determines the kind of functionality it will execute. For example, `deepxube viz --domain cube3 --steps 5` visualizes a Rubik's cube that has been scrambled 5 times. All positional arguments can be accessed via `deepxube --help` and all arguments for a particular positional argument can be accessed with `--help` (e.g. `deepxube viz --help`). The `domain_info` and `heuristic_info` show the information for registered domains and heuristic functions, respectively. `time` times basic functionality and `problem_inst` generates problem instances. `train` trains DNNs, `train_summary` shows the custom training progress visualization in Figure 9, and `solve` solves problem instances.

Future Work

Policy Networks

DeepXube currently supports the use of policy functions from which actions can be sampled in its pathfinding algorithms and has limited support for the training of such functions. Future work will build on algorithms in reinforcement learning for learning policies that can sample from arbitrary distributions (Haarnoja et al. 2017; Tessler, Tennenholtz, and Mannor 2019). The ability to do so will make DeepXube more applicable to domains with continuous action spaces, such as those seen in robotics, as well as those with very large action spaces, such as when modeling multi-agent path finding as a joint action space (Tang et al. 2025).

Learned Models

Hand-coding a transition function for pathfinding problems can be impractical for real-world domains, such as robotic manipulation. Model-based reinforcement learning (Sutton 1991) with deep learning has been shown to be capable of learning transition functions based on observed transitions (Oh et al. 2015; Tian et al. 2021). Furthermore, the integration of learned models with planning has been shown to result in better performance than just using reinforcement learning alone (Asai et al. 2022; Agostinelli and Soltani 2024). Future work will allow for training of a transition function from observed transitions and using the trained transition function in the definition of the domain.

Artificial Curiosity

Besides user-defined methods to sample problem instances, DeepXube offers two mixins that both sample problem instances from random walks. While this has been observed to work well in practice (Agostinelli et al. 2019; Tian et al. 2021; Chervov et al. 2025), there are many domains for which this will not produce useful problem instances. For example, for domains with many dead-ends, most random walks may end up in dead-ends and, therefore, not adequately explore the state space. Methods based on artificial curiosity or intrinsic motivation (Barto et al. 2004) explore the state space based on some metric of novelty, such as next state prediction or inverse dynamics error (Pathak et al. 2017). Related methods, such as learning to create a curriculum could be used to better explore the state space (Narvekar et al. 2020).

Conclusion

The integration of deep learning, reinforcement learning, and heuristic search has proven capable of solving challenging problems, such as combinatorial puzzles and quantum algorithm compilation, and offers the potential of learning powerful domain-specific heuristic functions in a largely domain-independent fashion. DeepXube provides a package that applies state-of-the-art heuristic learning and heuristic search algorithms based on a user-provided definition of a pathfinding domain and DNN architecture. DeepXube provides convenient visualization and progress monitoring tools for implementing domains and DNNs, training DNNs, solving problems, and visualizing solutions. Pathfinding algorithms integrate seamlessly with DNNs and can take advantage of GPU parallelism with a large beam or batch size. Time-consuming aspects of training are parallelized across both CPUs and GPUs. DeepXube also provides functionality to speed up domain methods with NumPy as well as for expressive goal specification with answer set programming. Finally, this functionality is conveniently accessible via the `deepxube` command-line tool.

References

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; et al. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems.

Agostinelli, F. 2025. A Conflict-Driven Approach for Reaching Goals Specified with Negation as Failure. In *SoCS*, volume 18, 2–10.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8): 356–363.

Agostinelli, F.; Panta, R.; and Khandelwal, V. 2024. Specifying goals to deep neural networks with answer set programming. In *34th International Conference on Automated Planning and Scheduling*.

Agostinelli, F.; Shperberg, S. S.; Shmakov, A.; McAleer, S.; Fox, R.; and Baldi, P. 2024. Q* search: Heuristic search with deep Q-networks. In *ICAPS PRL Workshop*.

Agostinelli, F.; and Soltani, M. 2024. Learning discrete world models for heuristic search. In *Reinforcement Learning Conference*.

Aine, S.; Swaminathan, S.; Narayanan, V.; Hwang, V.; and Likhachev, M. 2016. Multi-heuristic A*. *The International Journal of Robotics Research*, 35(1-3): 224–243.

Andrychowicz, M.; Wolski, F.; Ray, A.; Schneider, J.; Fong, R.; Welinder, P.; McGrew, B.; Tobin, J.; Abbeel, O. P.; and Zaremba, W. 2017. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, 5048–5058.

Asai, M.; Kajino, H.; Fukunaga, A.; and Muise, C. 2022. Classical planning in deep latent space. *Journal of Artificial Intelligence Research*, 74: 1599–1686.

Bao, N.; and Hartnett, G. S. 2024. Twisty-puzzle-inspired approach to Clifford synthesis. *Physical Review A*, 109(3): 032409.

Barto, A. G.; Singh, S.; Chentanez, N.; et al. 2004. Intrinsically motivated learning of hierarchical collections of skills. In *Proceedings of the 3rd International Conference on Development and Learning*, volume 112, 19. Citeseer.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bertsekas, D. P.; and Tsitsiklis, J. N. 1996. *Neuro-dynamic programming*. Athena Scientific. ISBN 1-886529-10-8.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM*, 54(12): 92–103.

Chen, B.; Li, C.; Dai, H.; and Song, L. 2020. Retro*: learning retrosynthetic planning with neural guided A* search. In *International Conference on Machine Learning*, 1608–1616. PMLR.

Chervov, A.; Khoruzhii, K.; Bukhal, N.; Naghiyev, J.; Zamkovoy, V.; Koltsov, I.; Cheldieva, L.; Sychev, A.; Lenin, A.; Obozov, M.; et al. 2025. A machine learning approach that beats Rubik's cubes. In *NeurIPS*.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence*, 14(3): 318–334.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo= ASP+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*.

Haarnoja, T.; Tang, H.; Abbeel, P.; and Levine, S. 2017. Reinforcement learning with deep energy-based policies. In *International conference on machine learning*, 1352–1361. PMLR.

Hadar, G.; Agostinelli, F.; and Shperberg, S. S. 2026. Beyond Single-Step Updates: Reinforcement Learning of Heuristics with Limited-Horizon Search. In *AAAI*.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Jin, J.; and Kim, K. 2020. 3D CUBE Algorithm for the Key Generation Method: Applying Deep Neural Network Learning-Based. *IEEE Access*, 8: 33689–33702.

Korf, R. E. 1997. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, 700–705. AAAI Press. ISBN 0-262-51095-2.

Landrum, G.; et al. 2013. Rdkit documentation. *Release*, 1(1-79): 4.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *nature*, 521(7553): 436.

Mark, M. S.; Gao, T.; Sampaio, G. G.; Srirama, M. K.; Sharma, A.; Finn, C.; and Kumar, A. 2024. Policy agnostic RL: Offline RL and online RL fine-tuning of any class and backbone. *arXiv preprint arXiv:2412.06685*.

McDermott, D. M. 2000. The 1998 AI planning systems competition. *AI magazine*, 21(2): 35–35.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533.

Muppasani, B.; Pallagani, V.; Srivastava, B.; and Agostinelli, F. 2024. Comparing Rubik's Cube Solvability in Domain-Independent Planners Using Standard Planning Representations for Insights and Synergy with Upcoming Learning Methods. In *International Conference on Automated Planning and Scheduling - Heuristics and Search for Domain-Independent Planning Workshop*.

Narvekar, S.; Peng, B.; Leonetti, M.; Sinapov, J.; Taylor, M. E.; and Stone, P. 2020. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21(181): 1–50.

Oh, J.; Guo, X.; Lee, H.; Lewis, R. L.; and Singh, S. 2015. Action-conditional video prediction using deep networks in

atari games. *Advances in neural information processing systems*, 28.

Panta, R.; Tavakoli, M.; Geils, C.; Baldi, P.; and Agostinelli, F. 2024. Finding Reaction Mechanism Pathways with Deep Reinforcement Learning and Heuristic Search. In *ICAPS PRL Workshop*. ICAPS.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Pathak, D.; Agrawal, P.; Efros, A. A.; and Darrell, T. 2017. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, 2778–2787. PMLR.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4): 193–204.

Qiuhao, C.; Du, Y.; Zhao, Q.; Jiao, Y.; Lu, X.; and Wu, X. 2024. Efficient and practical quantum compiler towards multi-qubit systems with deep reinforcement learning. *Quantum Science and Technology*.

Schmidhuber, J. 2015. Deep learning in neural networks: An overview. *Neural networks*, 61: 85–117.

Siddique, P. J.; Gue, K. R.; and Usher, J. S. 2021. Puzzle-based parking. *Transportation Research Part C: Emerging Technologies*, 127: 103112.

Sutton, R. S. 1991. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4): 160–163.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Tang, Y.; Xiong, X.; Xi, J.; Li, J.; Bıyık, E.; and Koenig, S. 2025. RAILGUN: A Unified Convolutional Policy for Multi-Agent Path Finding Across Different Environments and Tasks. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 8645–8651. IEEE.

Tessler, C.; Tennenholtz, G.; and Mannor, S. 2019. Distributional policy optimization: An alternative approach for continuous control. *Advances in Neural Information Processing Systems*, 32.

Tian, S.; Nair, S.; Ebert, F.; Dasari, S.; Eysenbach, B.; Finn, C.; and Levine, S. 2021. Model-Based Visual Planning with Self-Supervised Functional Distances. In *International Conference on Learning Representations*.

Turner, I.; Agostinelli, F.; and Fu, P. 2025. Quantum Circuit Synthesis with Deep Reinforcement Learning and Heuristic Search. In *ICAPS PRL Workshop*. ICAPS.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Watkins, C. J.; and Dayan, P. 1992. Q-learning. *Machine learning*, 8(3-4): 279–292.

You, J.; Liu, B.; Ying, Z.; Pande, V.; and Leskovec, J. 2018. Graph convolutional policy network for goal-directed molecular graph generation. *Advances in neural information processing systems*, 31.

Zhang, Y.-H.; Zheng, P.-L.; Zhang, Y.; and Deng, D.-L. 2020. Topological Quantum Compiling with Reinforcement Learning. *Physical Review Letters*, 125(17): 170501.