

★ Member-only story

Reinforcement Learning for Combinatorial Optimization

Learning strategies to tackle difficult optimization problems using Deep Reinforcement Learning and Graph Neural Networks.



Or Rivlin · Follow

Published in Towards Data Science

9 min read · Apr 6, 2019

Listen

Share

More



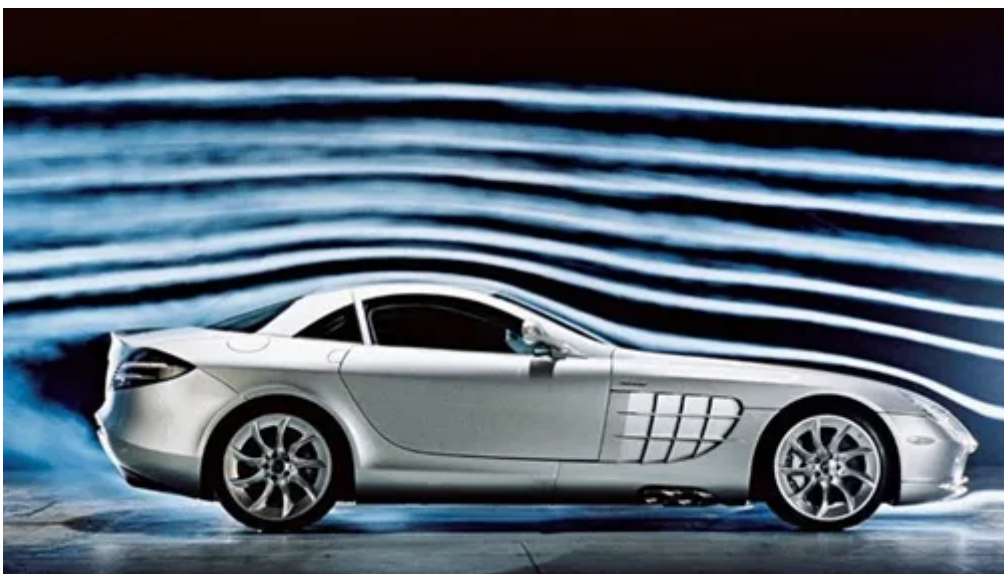
[source](#)

Why is Optimization Important?

From as early as humankind's beginning, millions of years ago, every innovation in technology and every invention that improved our lives and our ability to survive and thrive on earth, has been devised by the cunning minds of intelligent humans. From the fire to the wheel, and from electricity to quantum mechanics, our

understanding of the world and the complexity of things around us have increased to the point that we often have difficulty grasping them intuitively.

Today, designers of airplanes, cars, ships, satellites, complex structures many other endeavors are heavily relied on the ability of algorithms to make them better, often in subtle ways that humans could simply never achieve. In addition to design, optimization plays a crucial role in every-day things such as network routing (Internet and mobile), logistics, advertising, social networks and even medicine. In the future, as our technology continues to improve and complexify, the ability to solve difficult problems of immense scale is likely to be in much higher demand, and will require breakthroughs in optimization algorithms.



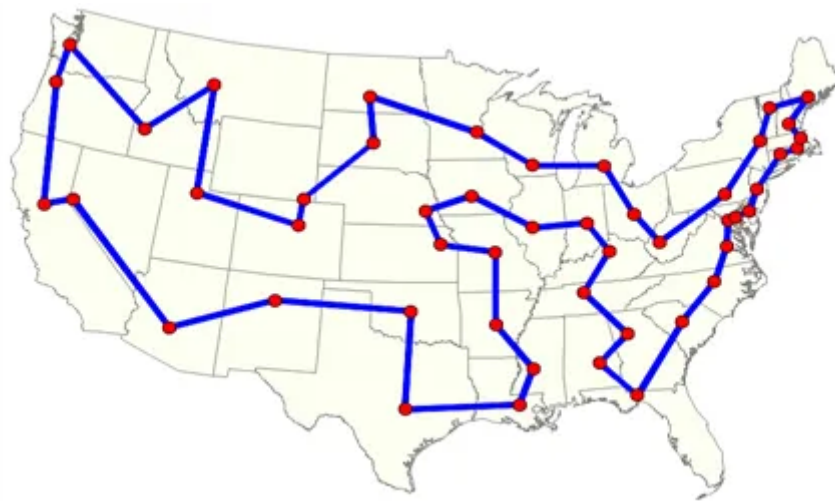
[source](#)

Combinatorial Optimization Problems

Broadly speaking, combinatorial optimization problems are problems that involve finding the “best” object from a finite set of objects. In this context, “best” is measured by a given evaluation function that maps objects to some score or cost, and the objective is to find the object that merits the lowest cost. Most practically interesting combinatorial optimization problems (COPs from now on) are also very hard, in the sense that the number of objects in the set increases extremely fast due to even small increases in the problem size, making exhaustive search impractical.

To make things clearer, we will focus on a specific problem, the well-known Traveling Salesman Problem (TSP). In this problem we have N cities, and our salesman must visit them all. However, travelling between cities incurs some cost and we must find a tour that minimizes the total accumulated cost while traveling to

all the cities and returning to the starting city. For example, the image below shows an optimal tour of all the capital cities in the US:



[source](#)

This problem naturally arises in many important applications such as planning, delivery services, manufacturing, DNA sequencing and many others. Finding better tours can sometimes have serious financial implications, prompting the scientific community and enterprise to invest a lot of effort in better methods for such problems.

While building a tour for a TSP instance with K cities, we eliminate a city at each stage of the tour construction process, until no more cities are left. At the first stage we have K cities to choose from to start the tour, at the second stage we have $K-1$ options and then $K-2$ options and so forth. The number of possible tours we can construct is the product of the number of options we have at each stage, and so the complexity of this problem behaves like $O(K!)$. For small numbers this may seem not so bad. Say we have a 5 city problem, the number of possible tours is $5!=120$. But for 7 cities it increases to 5040, for 10 cities it's already 362880 and for 100 cities it's a whopping $9.332622e+157$, which is many orders of magnitude more than the number of atoms in the universe. Practical instances of TSP that arise in the real world often have many thousands of cities, and require highly sophisticated search algorithms and heuristics that have been developed for decades in a vast literature in order to be solved in a reasonable time (which could be hours). Unfortunately, many COPs that arise in real-world applications have unique nuances and constraints that prevent us from just using state of the art solvers for known problems such as TSP, and require us to develop methods and heuristics specific to that problem. This process can be long and arduous, and might require the work of

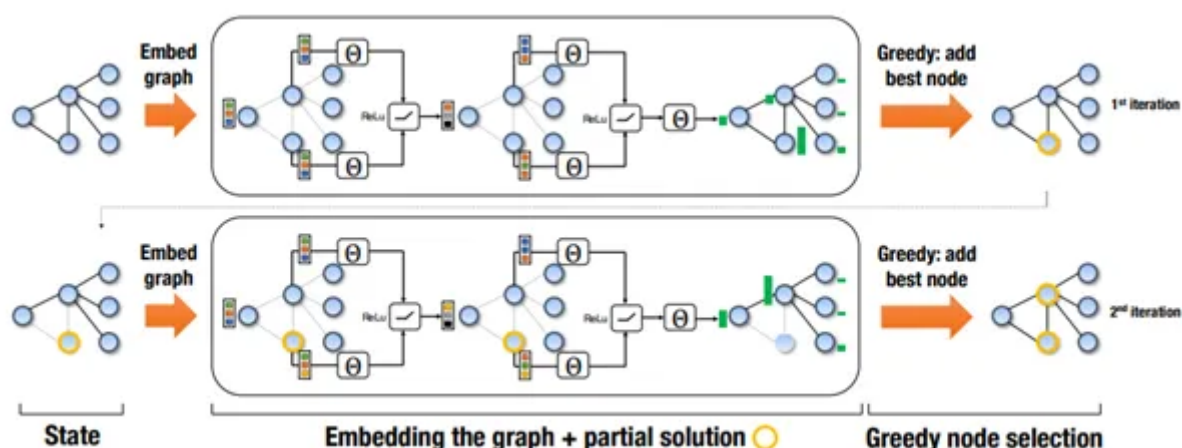
domain experts to detect some structure in the combinatorial search space of the specific problem.

Due to the dramatic successes of Deep Learning in many domains in recent years, the possibility of letting a machine learn how to solve our problem on its own sounds very promising. Automating the process of designing algorithms to difficult COPs could save a lot of money and time and could perhaps yield better solutions than human-designed methods could (as we have seen in achievements such as that of AlphaGo, which beat thousands of years of human experience).

Learning with Graph Representations

An early attempt at this problem came in 2016 with a paper called “[Learning Combinatorial Optimization Algorithms over Graphs](#)”. In this paper the authors trained a kind of Graph Neural Network ([I discuss graph neural networks in another article](#)) called **structure2vec** to greedily construct solutions to several hard COPs and achieved very nice approximation ratios (the ratio between the produced cost and the optimal cost).

The basic idea goes like this: the state of the problem can be expressed as a graph, on which the neural network builds the solution. At each iteration of the solution construction process our network observes the current graph, and chooses a node to add to the solution, after which the graph is updated according to that choice, and the process is repeated until a complete solution is obtained.

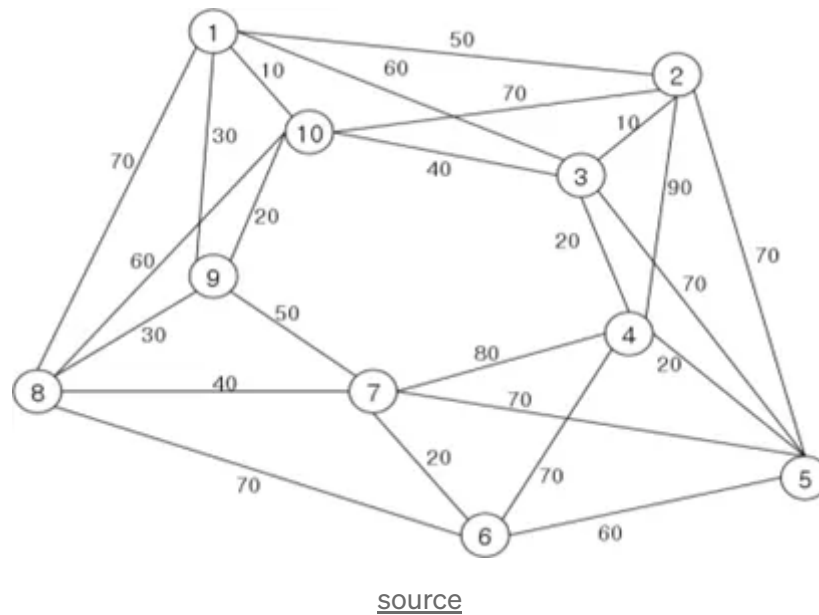


[source](#)

The authors trained their neural network using a DQN algorithm and demonstrated the learned model’s ability to generalize to much larger problem instances than it was trained on. Their model generalized well even to instances of 1200 nodes (while

being trained on instances of around 100 nodes), and could produce in 12 seconds solutions that were sometimes better than what a commercial solver could find in 1 hour. A big disadvantage of their method was that they used a “helper” function, to aid the neural network find better solutions. This helper function was a human designed one, and problem specific, which is what we would like to avoid.

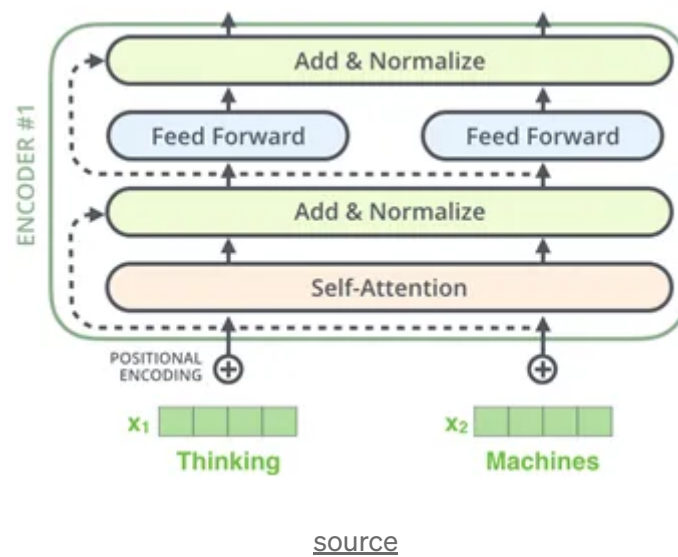
This use of a graph-based state representation makes a lot of sense, as many COPs can be very naturally expressed in this way, as in this example of a TSP graph:



The nodes represent the cities, and the edges contain the inter-city distances. A very similar graph can be constructed without the edge attributes (if we do not assume knowledge of the distances for some reason). Recent years have seen an incredible rise in the popularity of neural network models that operate on graphs (with or without assuming knowledge of the structure), most notably in the area of Natural Language Processing where **Transformer** style models have become state of the art on many tasks.

There are many excellent articles that explain the Transformer architecture in detail, so I will not delve too much into it, but give a very brief overview. The transformer architecture was introduced by Google researchers in a famous paper titled “**Attention Is All You Need**” and was used to tackle sequence problems that arise in NLP. The difference is that unlike in Recurrent Neural Networks such as LSTMs, which are explicitly fed a sequence of input vectors, the transformer is fed the input as a set of objects, and special means must be taken to help it see the order

in the “sequence”. The transformer uses several layers that consist of a multi-head self-attention sublayer followed by a fully connected sublayer.



The relationship to graphs becomes evident in the attention layers, which are actually a sort of message passing mechanism between the input “nodes”. Each node observes the other nodes and attends to those that seem more “meaningful” for it. This is very similar to the process that happens in Graph Attention Networks, and in fact, if we use a mask to block nodes passing messages to non-adjacent ones, we get an equivalent process.

Learning to Solve Problems Without Human Knowledge

In their paper “Attention! Learn to Solve Routing Problems”, the authors tackle several combinatorial optimization problems that involve routing agents on graphs, including our now familiar Traveling Salesman Problem. They treat the input as a graph and feed it to a modified Transformer architecture that embeds the nodes of the graph, and then sequentially chooses nodes to add to the tour until a full tour has been constructed. Treating the input as a graph is a more ‘correct’ approach than feeding it a sequence of nodes, since it eliminates the dependency on the order in which the cities are given in the input, as long as their coordinates do not change. This means that however we permute the cities, the output of a given graph neural network will remain the same, unlike in the sequence approach.

In the architecture presented in the paper, the graph is embedded by a transformer style Encoder, which produces embeddings for all the nodes, and a single embedding vector for the entire graph. To produce the solution, a separate Decoder network is given each time a special **context** vector, that consists of the graph

embedding and those of the last and first cities, and the embeddings of the unvisited cities, and it outputs a probability distribution on the unvisited cities, which is sampled to produce the next city to visit. The decoder sequentially produces cities until the tour is complete, and then a reward is given based on the length of the tour.

The authors train their model using a reinforcement learning algorithm called REINFORCE, which is a policy gradient based algorithm. The pseudo code for their version can be seen here:

Algorithm 1 REINFORCE with Rollout Baseline

```

1: Input: number of epochs  $E$ , steps per epoch  $T$ , batch size  $B$ ,
   significance  $\alpha$ 
2: Init  $\theta$ ,  $\theta^{\text{BL}} \leftarrow \theta$ 
3: for epoch = 1, ...,  $E$  do
4:   for step = 1, ...,  $T$  do
5:      $s_i \leftarrow \text{RandomInstance}() \ \forall i \in \{1, \dots, B\}$ 
6:      $\pi_i \leftarrow \text{SampleRollout}(s_i, p_\theta) \ \forall i \in \{1, \dots, B\}$ 
7:      $\pi_i^{\text{BL}} \leftarrow \text{GreedyRollout}(s_i, p_{\theta^{\text{BL}}}) \ \forall i \in \{1, \dots, B\}$ 
8:      $\nabla \mathcal{L} \leftarrow \sum_{i=1}^B (L(\pi_i) - L(\pi_i^{\text{BL}})) \nabla_\theta \log p_\theta(\pi_i)$ 
9:      $\theta \leftarrow \text{Adam}(\theta, \nabla \mathcal{L})$ 
10:   end for
11:   if OneSidedPairedTTest( $p_\theta, p_{\theta^{\text{BL}}}$ ) <  $\alpha$  then
12:      $\theta^{\text{BL}} \leftarrow \theta$ 
13:   end if
14: end for

```

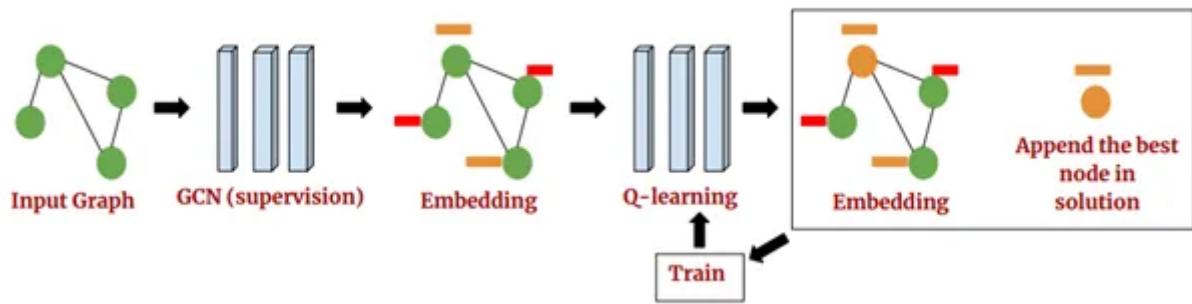
[source](#)

They use a roll-out network to deterministically evaluate the difficulty of the instance, and periodically update the roll-out network with the parameters of the policy network. Using this method, the authors achieve excellent results on several problems, surpassing the other methods that I mentioned in previous sections. However, they still train and evaluate their method on small instances, with up to 100 nodes. While these results are promising, such instances are minuscule compared to real-world ones.

Scaling to Very Large Problems

Very recently, an important step was taken towards real-world sized problem with the paper “[Learning Heuristics Over Large Graphs Via Deep Reinforcement Learning](#)”. In this paper the authors trained a Graph Convolutional Network to solve large instances of problems such as Minimum Vertex Cover (MVC) and Maximum Coverage Problem (MCP). They used a popular greedy algorithm for these problems

to train the neural network to embed the graph and predict the next node to choose at each stage, and then further trained it using a DQN algorithm.



[source](#)

They evaluated their method on graphs with **millions of nodes**, and achieved results that are both better and faster than current standard algorithms. While they did make use of a hand-crafted heuristic to help train their model, future works might do away with this constraint, and learn to solve huge problems Tabula Rasa.

Overall, I think that the quest to find structure in problems with vast search spaces is an important and practical research direction for Reinforcement Learning. Many critics of RL claim that so far it has only been used to tackle games and simple control problems, and that transferring it to real-world problems is still very far away. Though those claims might be true, I think that the methods I outlined in this article represent very real uses that could benefit RL in the very near-term future, and it's a shame that they don't attract as much attention as methods for video games.

I implemented a relatively simple algorithm for learning to solve instances of the Minimum Vertex Cover problem, using a Graph Convolutional Network. [Feel free to check it out.](#)

Machine Learning

Deep Learning

Reinforcement Learning

Artificial Intelligence

Towards Data Science

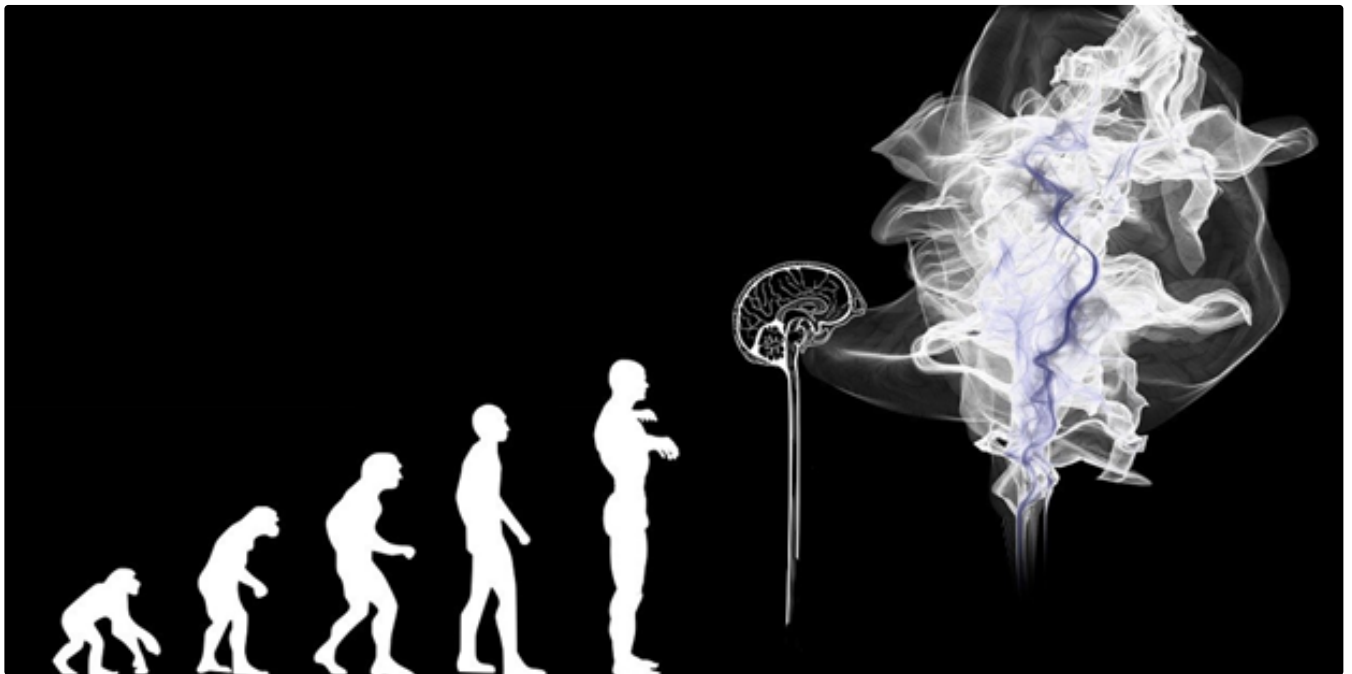


Follow

Written by Or Rivlin

799 Followers · Writer for Towards Data Science

More from Or Rivlin and Towards Data Science



 Or Rivlin in Towards Data Science

The Power of Offline Reinforcement Learning

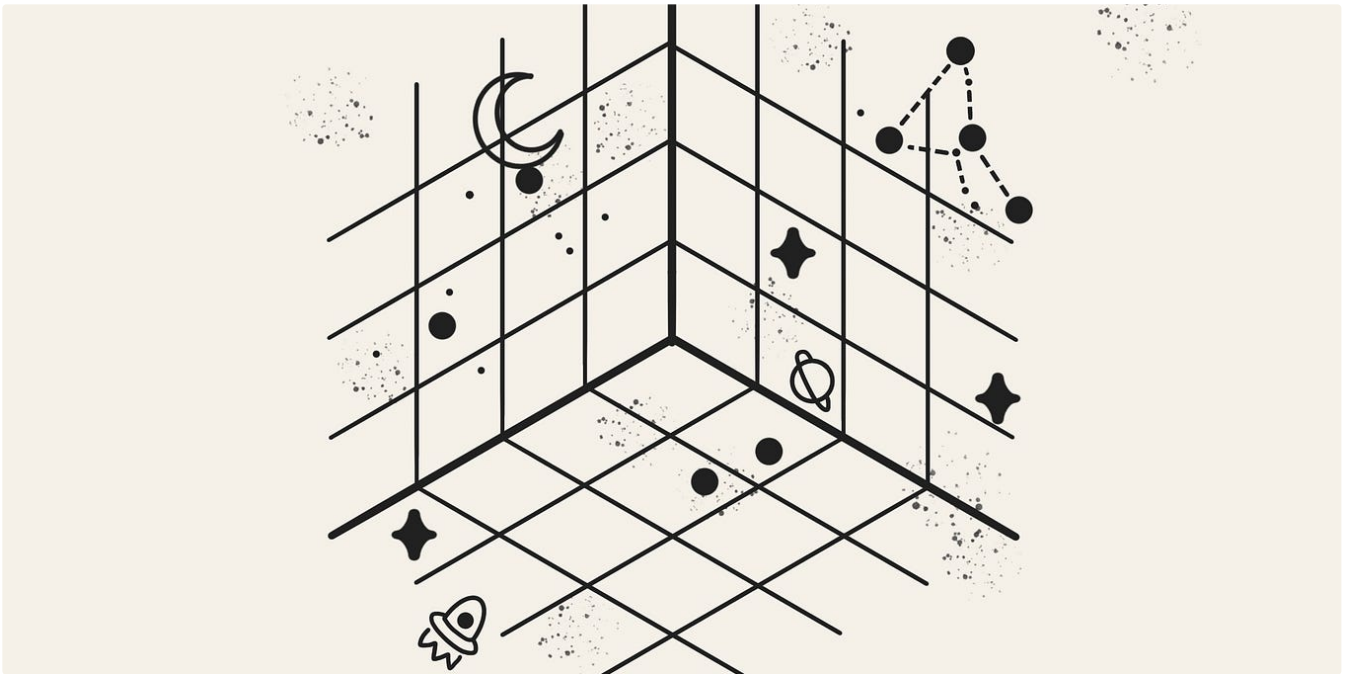
RL algorithms that could potentially scale to real-world problems

13 min read · Nov 1, 2020

 184

 1





 Leonie Monigatti in Towards Data Science

Explaining Vector Databases in 3 Levels of Difficulty


From noob to expert: Demystifying vector databases across different backgrounds

★ · 8 min read · Jul 4

 1.8K  18



 Kenneth Leung in Towards Data Science

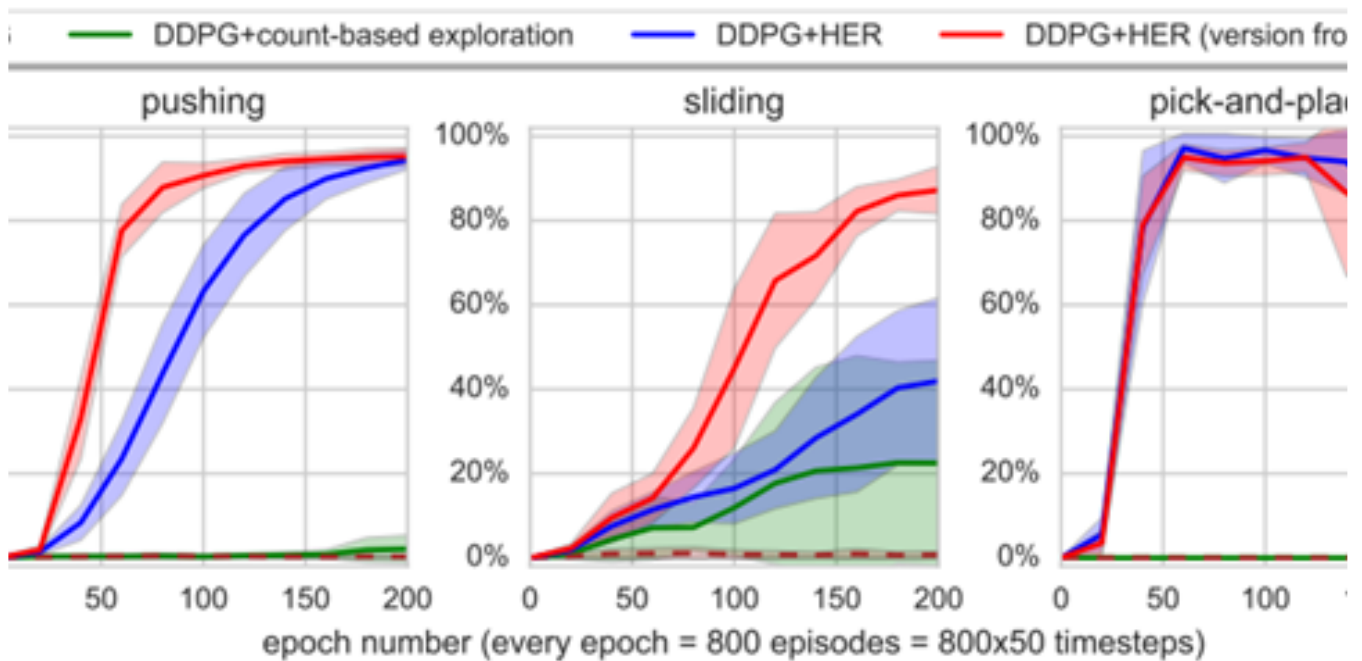
Running Llama 2 on CPU Inference Locally for Document Q&A


Clearly explained guide for running quantized open-source LLM applications on CPUs using LLama 2, C Transformers, GGML, and LangChain

★ · 11 min read · Jul 18

👏 1.4K 💬 21

🔖⁺ ⋮



 Or Rivlin in Towards Data Science

Reinforcement Learning with Hindsight Experience Replay

Sparse and Binary Rewards

10 min read · Jan 31, 2019

👏 777 💬 2

🔖⁺ ⋮

See all from Or Rivlin

See all from Towards Data Science