[Re] Utilising Uncertainty for Efficient Learning of Likely-Admissible Heuristics

Jason M. Wille
School of Computer Science and Applied Mathematics
University of the Witwatersrand
Johannesburg, South Africa
1352200@wits.ac.za

Reproducibility Summary

Scope of Reproducibility

I will try and reproduce the result where the paper claims that their method can produce likely-admissible heuristics that produce optimal results more frequently than past work that trained their heuristics on optimal plans.

Methodology

In order to implement the experiments I used a hybrid approach of referring to the code repository from the paper and also by using the concepts discussed in the paper itself. I only attempted to reproduce the results for the 15 Puzzle environment and a library was used to represent the Puzzle objects.

Results

Results were not obtained from the experiments run and no conclusions were drawn about the validity of the results from the original paper. A discussion was had around the difficulty to implement the methods described.

What was easy

It was easy to understand the desirability of being able to learn likely-admissible heuristics from non-optimal plans. In terms of re-implementing the paper, it was not easy to follow the implementation from the C# code, but it was much easier to write code in Python.

What was difficult

Understanding the core concepts of uncertainty and the different neural networks and their purposes was difficult. Implementing them also became challenging because of this. The paper was difficult to understand without a strong mathematical background.

33rd Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

1 Introduction

Creating admissible heuristics is desirable as they lead to optimal plans. The challenge faced is that creating strong admissible heuristics often requires expert domain knowledge or high memory resources. In order to work around these issues machine learning techniques are used to develop likely-admissible heuristics. This means that they are admissible with some probability and therefore the plans produced are also likely-optimal plans with some probability. The machine learning algorithms train on datasets comprised of states and their corresponding cost-to-goal. This leads to a model that outputs an estimate for the cost-to-goal from any given state.

The development of likely-admissible heuristics originally required training data that consisted of optimal plans which were prohibitive to produce. This then led to a method that did not require optimal plans, however, it had two main shortcomings. One was that it took very long to train the heuristics and the other was that learning on non-optimal plans meant that errors compounded which lead to heuristics that had high suboptimality.

The paper being reproduced [MR20] aims to overcome both of the shortcoming mentioned in the previous paragraph by utilising different measures of uncertainty. The paper was able to produce optimality statistics that are competitive with previous approaches trained on optimal plans.

2 Scope of reproducibility

The paper being reproduced [MR20] had the main aim of being able to train likely-admissible heuristics on plans that are not always optimal and have the likely-admissible heuristics still perform well compared to heuristics trained on optimal plans. The other shortcoming it tried to alleviate was efficiency, the generation of tasks should perform better than random task generation.

- By modelling uncertainty the heuristics trained on non-optimal plans will be able to solve the 15-puzzle optimally at a percentage of time similar to previous work that trained on optimal plans.
- The approach used in *GenerateTaskPrac* should be more efficient at generating tasks than random task generation.

3 Background

3.1 Planning

A planning domain is defined as a tuple $\langle S, O, \varepsilon, C, S_0, S_g \rangle$. The values are representative of the following:

- S - The state space.
- O - $O(s)$ is an operator that returns legal operators that can be executed in state $s \in S$.
- ε - $\varepsilon(s, o)$ is an effect function that returns the next state $s' \in S$ when operator $o \in O(s)$ is applied in state s.
- C - $C(s, s')$ is a strictly positive and bounded function for the cost of moving from state s to state s'.
- S_0 - $S_0 \in S$ is the set of possible start states.
- S_g - $S_g \in S$ is the set of possible goal states.

For a planning task, the start state and goal state are set to one state each and the start state and goal state may not equal each other ($s_0 \neq s_g$). In [MR20] each domain has a unique goal state and therefore tasks in a domain only differ in start states.

Planning algorithms require a heuristic function $h(s)$ that estimates the optimal cost-to-goal from state s. We denote the optimal and unknown cost-to-goal from s with $h^*(s)$. Given a planning task T, a planning algorithm aims to find a plan $\pi = (s_0, s_1, s_2, ..., s_n = s_g)$. If h is an admissible heuristic so that $h(s) \leq h^*(s)$ for all $s \in S$ then certain planning algorithms guarantee that π is optimal. π is optimal if it a plan with minimal cost. We denote optimal plans with π^*.

Guaranteeing admissibility is difficult and a weaker property is for a heuristic to be likely-admissible. A likely-admissible heuristic, denoted h^α, is admissible with probability $\alpha \in (0, 1)$. i.e. $P(h^\alpha(s) \leq h^*(s)) = \alpha$ for all $s \in S$.

3.2 Learning Heuristics from Data

Given a plan π we can compute for each $s_j \in \pi$ where $j < n$ the cost to goal from s_j to $s_n = s_g$, $y_j = \sum_{i=j}^{n-1} C(s_i, s_{i+1})$. If we are given a dataset consisting of M plans from the planning domain we can construct a training dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$ of size $N = \sum_{i=1}^{M} |\pi_i| - M$. This would be done by calculating the cost-to-goal for the relevant states in each plan where $x_i = F(s_i)$ and F is a function that converts a state to a feature representation that can be used as input to a supervised machine learning algorithm.

We can then learn a heuristic from the dataset D. In general, such a heuristic is not admissible and therefore will not lead to optimal plans. The quality of the learned heuristic depends on the quality of the plans used for training. Training on optimal or near optimal-plans will produce a heuristic that is a closer approximation to h^* and therefore have lower sub-optimality.

3.3 Bayesian Neural Networks

Neural networks can be viewed as a probabilistic model $P(y|x, \mathbf{w})$ where x is the input to the network and $\mathbf{w}$ are the weights. For regression we generally assume that $y|x, \mathbf{w} \in \mathbb{R}$ follows a Gaussian distribution such that $y|x, \mathbf{w} \sim N(\hat{y}(x, \mathbf{w}), \sigma_a^2(x, \mathbf{w}))$.

Typically when doing regression with neural networks only a single output neuron is used to learn $\hat{y}$ while σ_a^2 is assumed to be a known constant, and this corresponds to the well-known squared loss objective $(\hat{y} - y)^2$. However, it is straightforward to adjust the network to have two output neurons and learn σ_a^2 as well. This leads to the following minimisation objective function:

$$L(D, \mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \frac{(\hat{y}(x_i, \mathbf{w}) - y_i)^2}{2\sigma_a^2(x_i, \mathbf{w})} + \frac{1}{2} \log \sigma_a^2(x_i, \mathbf{w}) \tag{1}$$

In practice, we learn a second output neuron $p \in \mathbb{R}$ and then apply a transformation to get $\sigma_a \in \mathbb{R}^+$ such as $\sigma_a = \log(1 + \exp(p))$. Then σ_a^2 captures the noise in the data and is thus a measure of aleatoric uncertainty. This is the uncertainty that the model cannot describe.

Epistemic uncertainty (which accounts for uncertainty in the model that can be explained away given enough data) is more difficult to model as it requires the posterior distribution $P(\mathbf{w}|D)$ which is generally intractable to compute.

One effective and computationally efficient method to approximate the posterior distribution is to use a weight uncertainty neural network (WUNN) and choose to model the weights not as fixed point values but as independent Gaussians, $w_i|D \sim N(\mu_{w_i|D}, \sigma_{w_i|D}^2)$.

Using a Gaussian prior for the weights with mean μ_0 and variance σ_0^2 so that $w_i \sim N(\mu_0, \sigma_0^2)$, the variational approximation to the posterior can be shown to be the following minimisation objective:

$$L(D, \theta) = \beta KL[P(\mathbf{w}|\theta)||P(\mathbf{w})] - \mathbb{E}_{P(\mathbf{w}|\theta)}[\log P(D|\mathbf{w})] \tag{2}$$

In this equation KL is the Kullback-Leibler divergence, $\theta = \{(\mu_{w_i|D}, \sigma_{w_i|D}^2)\}_i$ for each weight connection in the network and $\beta > 0$ controls how much weight to give the prior relative to the likelihood. A common strategy is to decay β over time so as to initially give more emphasis to the prior and gradually reduce this as the network is trained on more data.

In practice, the objective function is approximated with Monte Carlo during training by sampling $z_i^s \sim N(0, 1)$ for each weight connection in the network and then applying the transformations $w_i^s = z_i^s \sigma_{w_i|D} + \mu_{w_i|D}$ where $s \in [1, S]$ and S is the number of Monte Carlo samples averaged over.

Given a WUNN trained to minimise the objective, we can get a measure of epistemic uncertainty for an input x by sampling weights from the network and computing the variance of the network output

$$\sigma_e^2(x) = \frac{1}{K} \sum_{k=1}^{K} \hat{y}^2(x, \mathbf{w}_k) - \hat{y}^2(x) \tag{3}$$

where K is the number of samples from the network and $\hat{y}(x) = \frac{1}{K} \sum_{k=1}^{K} \hat{y}(x, \mathbf{w}_k)$.

Finally, given an input x we can approximate the posterior predictive distribution as

$$y|x \sim N(\hat{y}(x), \sigma_t^2(x)) \tag{4}$$

where $\sigma_t^2(x) = \sigma_a^2(x) + \sigma_e^2(x)$ and $\sigma_a^2(x) = \frac{1}{K}\sum_{k=1}^{K} \sigma_a^2(x, \mathbf{w}_k)$.

4 Methodology

The approach used was mixed between trying to directly implement the author's code and trying to implement directly from the description given in the paper. Initially when struggling to understand the concepts it seemed like trying to re-implement the paper almost exclusively by using the corresponding code repository seemed like a good option, this solution did not work well as the code was too complex to follow and that coupled with not being able to run the code led to many challenges.

This then led to the approach of coding from the understanding the paper had provided. This along with referring to the code and also getting the code to run meant that it was easier to understand more of the concepts and how to actually implement them. There were still many challenges faced.

4.1 Model descriptions

The two models that were implemented were the feed forward neural network (FFNN) and the weight uncertainty neural network (WUNN). The weight uncertainty neural network was implemented using a library called torchbnn [LKL22]. This library uses PyTorch under the hood and PyTorch was used to implement the FFNN.

The models both had one hidden layer with 20 neurons, and input layer with 16 neurons. Each input neuron was for one tile in the 15-puzzle. The WUNN had a single output neuron. The FFNN had two output neurons, one to predict the cost-to-goal from a given state, and another to predict the aleatoric uncertainty. In order to get a measure of the epistemic uncertainty, the WUNN was queried multiple times with the same input and the variance of the outputs would represent the epistemic uncertainty.

The WUNN was initialised with $\mu_0 = 0$ and $\sigma_0^2 = 10$.

4.2 Environments

4.2.1 15-Puzzle

The only environment that was used was the 15-puzzle. It was implemented using a library called 15-Puzzle-Solvers. The only function that needed to be hand coded relating to the environment was the implementation of IDA* that used the custom heuristic that the FFNN would help to create.

In order to create the optimal plans used to compare the performance of the trained heuristic, we would use the built in A* search of the library and use the default heuristic, Manhattan Distance. This heuristic is admissible and therefore the plans are optimal when found using A* search.

4.3 Hyperparameters

The hyperparameters used were those of the original paper. The setup tried to mimic the experiments as closely as possible. Some of the parameters used were:

- Feedforward neural network learning rate = 0.001
- Weight uncertainty neural network learning rate = 0.01
- All neural networks layers used relu activations, had a single hidden layer, and were trained with the Adam optimiser.

4.4 Experimental setup and code

The experiments were not replicated, the setup attempted was to try and train the heuristic using *LearnHeuristicPrac*. If we were able to train a heuristic we would have measured the performance of it on 100 15-Puzzle problems and compared it to optimal solutions obtained using A* search and Manhattan Distance. This would have given a measure of the optimality of the trained heuristic.

4.5 Computational requirements

In order to attempt the trials the following hardware was used:

Apple MacBook Pro 2021

- RAM: 16 GB
- CPU: M1 Pro (10 Cores)

In an initial attempt to solve 100 randomly generated 15-Puzzle problems using A* search with the Manhattan Distance heuristic to get optimal solutions, it took 3 hours to solve 5 puzzles. This result seemed like a blocker as I was unsure how to speed it up.

5 Results

No experiments were able to be run on the implementation completed due to the challenges faced in implementation. These will be discussed further in Section 6.5.

6 Discussion

6.1 Approach

The approach that was taken originally of trying to implement the solution based solely on the C# code was not a good use of time. This conclusion was drawn after a lot of time trying to understand the concepts purely from the code. The problem with using this approach was that I did not have a setup that was supporting me running the code and therefore debugging it and being able to navigate through the code effectively was not possible. This along with the fact that the code was written in a generic manner to support all of the different environments using generic types made it even more difficult to follow.

When the approach above eventually lead to a block, it was a good time to try and revise the approach. I then decided to take a hybrid approach and try to understand the concepts better and then implement them while just using the code as reference for when I was stuck. This option lead to a better implementation. The problems came about when it was time to implement the FFNN and the WUNN as these concepts were foreign to me and using a library such as PyTorch made easy to implement a basic model but did not help in understanding how it worked. This approach ultimately lead to the final product that is in the GitHub repository.

In order to refine the process further, with more time, one thing that could have been done is to debug the C# after setting up the environment successfully. This could then help to compare the steps taken in the code I implemented and the C# code.

6.2 Background

One blocker that lead to a lot of misunderstanding of the paper was the limited understanding of the Bayesian neural networks and the weight uncertainty neural network. These concepts and the idea of aleatoric and epistemic uncertainty could have been better explained in order to understand how they work and why they were used. The background knowledge given in the paper makes use of very formal definitions that did not lead to an intuitive understanding.

6.3 Supplementary Material

The use of algorithm 3 and algorithm 4 in the supplementary material was definitely the most helpful aspect of the paper when it came to reproducing it. If more of the paper could have been written in this was it would have helped a lot in the process of reproducing it. After implementing algorithm 3 and 4 to the best of my ability it become a challenge to then figure out if they were performing as they should be and therefore the algorithms did not solve all the problems. Debugging the code and trying to understand what should be happening proved very difficult.

6.4 What was easy

I did not find anything relating to reproducing this paper to be easy. I understood the 15-puzzle game and the fact that admissible heuristics produce optimal plans using the A* search.

6.5 What was difficult

Reproducing this paper was extremely challenging as there were many concepts that were unfamiliar. The ability to translate the concepts into code was also difficult.

Some of the challenges include, but are not limited to:

- The repository provided in the paper was difficult to setup as it was a .NET project which seemed to be better suited to Windows machines. The provided README also did not provide any steps on how to best setup the environment or compatible versions of the different software used.
- The model was developed in TensorFlowSharp, this should have been done using TensorFlowSharp.NET. This would have made the code much more readable. According to TensorFlowSharp's README:

  > I strongly recommend that you use TensorFlow.NET which takes a different approach than TensorFlowSharp, it uses the Python naming convention and has a much broader support for the higher level operations that you are likely to need - and is also actively maintained.

- The paper could have split the parameters and sections into more easily digestible sections. When trying to reproduce the implementation often it was difficult to keep track of which parameters referred to which experiment and between referring to the supplementary material and between the experiments, conceptual framework and practical implementation it became difficult to track.
- From the paper alone it was difficult to figure out how the first pass over LearnHeuristicPrac worked before the neural networks were trained.
- Understanding the code was extremely difficult because of generic it was due to the fact that it was developed for different environments/domains.

References

[LKL22] Sungyoon Lee, Hoki Kim, and Jaewook Lee. Graddiv: Adversarial robustness of randomized neural networks via gradient diversity regularization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[MR20] Ofir Marom and Benjamin Rosman. Utilising uncertainty for efficient learning of likely-admissible heuristics. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 560–568, 2020.