

Computer Games Development

Project Report

Year IV

Ben Millar

C00236772

26/04/2022

[Declaration form to be attached]

Contents

[INTRODUCTION 1](#_Toc101540546)

[Why should we solve this problem? 1](#_Toc101540547)

[Why deep reinforcement learning? 1](#_Toc101540548)

[Scope 2](#_Toc101540549)

[METHODS 2](#_Toc101540550)

[Evaluating performance 2](#_Toc101540551)

[Modelling the problem 3](#_Toc101540552)

[RESEARCH 4](#_Toc101540553)

[Algorithm selection 4](#_Toc101540554)

[RESULTS 5](#_Toc101540555)

[CONCLUSIONS 5](#_Toc101540556)

[FURTHER WORK 5](#_Toc101540557)

[References 6](#_Toc101540558)

Efficient Provisioning of Virtual Machine Sets with Placement Constraints in IaaS Clouds with Deep Reinforcement Learning

***Abstract***–Today, cloud service providers like Amazon Web Services (AWS), Google Cloud, Microsoft Azure and more provide computing capacity on demand. A fundamental problem that arises when providing these services is that of mapping requests to physical machines in such a way as to minimise wasted space and maximise the number of requests that can be served given fixed physical infrastructure. This project explores the viability of using deep reinforcement learning to determine a solution for VM mapping problems to minimize waste, and therefore maximise revenue for the cloud service provider. Following some consideration, a proximal policy optimization (PPO) algorithm was used in conjunction with a multi-layer perceptron (MLP) network comprising 2 hidden layers of 64 nodes with input and output layers correlating with the environment size.

# INTRODUCTION

## Why should we solve this problem?

Infrastructure-as-a-service is undeniably a huge industry, and it is estimated that it will continue to grow by 23.2% year-on-year until 2027 (Gaul, 2022). As demand for these services increases, so too will the amount of work required to allocate requests in a timely fashion in terms of both time and processing power. Traditionally, these problems have been solved using complex integer linear programming techniques, or one-size-fits-all heuristics like first-fit decreasing (FFD) – I posit that by harnessing recent advances in machine learning, particularly concerning the efficiency of deep Q networks, we can reduce both the complexity of mapping these requests and the amount of space wasted. Harnessing this knowledge could not only improve efficiency at the point of allocation but also enable cloud service providers to tailor their pricing model based on their current capacity to encourage consumer behaviour in line with the optimal policy, which in turn would allow providers to offer more competitive rates.

## Why deep reinforcement learning?

To answer this question, we should first examine all of our options. Machine learning approaches can be broken down into three main categories:

1. Supervised learning.
2. Unsupervised learning.
3. Reinforcement learning.

Supervised learning requires a labelled dataset for training, for example, pictures of dogs that are tagged “dog”, and its goal is to learn the association between data and its label. This allows the agent to look at new data and determine what labels should be applied to it with varying degrees of certainty. Training this type of machine learning agent requires people to manually review data and apply the appropriate tags, which is in no way practical for the type of problem we are trying to solve.

Unsupervised learning does not require a labelled dataset; its goal is to find patterns in data which it can then use to decide how similar two pieces of data are. One application of this type of agent is anomaly detection in cases where we might want to flag data or events that differ from the norm. Again, this does not fit our use case.

Finally, reinforcement learning uses a trial-and-error approach whereby an agent takes actions in some environment; this agent is then either rewarded or penalised for these actions to reinforce the behaviour we want the agent to learn. Over time, a value mapping can be determined between states and actions. You could imagine, for example, that in a game of rock paper scissors we may have a state wherein our opponent has just played paper; in this state, we would expect a well-trained RL agent to associate a high expected reward with the action ‘scissors’, and a low reward with the action ‘rock’ (Of course, this sort of asymmetric gameplay is not traditionally encouraged in rock paper scissors…).

In complex environments where it may not be practical to map every state to the optimal action, we can introduce an artificial neural network (ANN)to estimate the value of states based on the values of similar states we have seen in the past. The addition of an ANN is what distinguishes deep reinforcement learning from reinforcement learning.

## Scope

In this paper, we are going to focus specifically on the provision of Virtual Machines (VMs), and all real-world data will be taken from Amazon’s Elastic Compute Cloud (EC2). This choice was made to both keep things consistent, and to limit the research scope to manageable levels. It is expected that other market-leading cloud service providers would operate similar pricing models to AWS to remain competitive and so the learnings from this project should be transferrable.

# METHODS

## Evaluating performance

It is challenging to benchmark the success of our approach without having some data to compare it against. As detailed revenue information from EC2 is not publicly available, I propose determining some baseline for the possible revenue generated for a given infrastructure and scoring our attempts against this. If we determine the average value per unit by dividing the cost of an instance by the sum of its dimensions, then we can multiply this value by the number of units available in our environment to estimate the value we could expect from a perfect fit of a uniform range of instances.

Using this, we may see that a heuristic-based approach achieves 80% of the benchmark or exceeds it by some percentage; this can give us a quantifiable target against which we can measure.

## Modelling the problem

There are three principal areas to consider with this problem:

1. Modelling some environment that is both analogous to the real-world problem, and conducive to machine learning approaches in the sense that we can extrude some clear value signal to optimize.
2. Finding data that is representative of the real world.
3. Determining some machine learning model which can produce meaningful insights.

I believe that we can treat this as a multi-dimensional specialization of the online knapsack problem in which our agent is given some number of servers each with fixed storage, memory, and CPU capacity. At each timestep, our agent will then receive a request for VM capacity which it must then either allocate to a server or refuse.

The modelling of the environment will be covered in more depth in the technical design document but, in short, I first reduced the problem to a 1-dimensional bin packing environment whereby we had some number of bins, or containers, each with a non-negative integer capacity. At each time step, we would generate a new item with a size between 1 and the maximum bin capacity and try to place this into one of the bins. From here, I iteratively increased the complexity of the environment first by adding a value to each item that did not necessarily correlate to its size (turning this into a knapsack-style problem), and then by increasing the dimensionality of the items.

I pulled real-world pricing data from Amazon’s EC2 service; this was taken both from the on-demand pricing[[1]](#footnote-1) and the spot pricing[[2]](#footnote-2). I collected data on the price, the number of vCPUs, memory, storage, and network performance of 392instance types (instance types were ignored where pricing information for both on-demand and spot instances was not available). While we cannot know the true distribution of these instances in the real world, we can make certain assumptions. If we compare the pricing for both instance types, we can determine some discount factor that we assume to be inversely proportional to that instance type’s popularity (David Naori, 2020).

# RESEARCH

Previous research has looked at solving this problem through approaches like integer linear programming and heuristic-based approaches like first-fit descending.

## Algorithm selection

The foundational method in the deep reinforcement learning space is DQN, or Deep Q-Network, which was proposed in the paper “Playing Atari with Deep Reinforcement Learning” (Mnih, et al., 2013). DQN uses Q-learning, whereby we seek to continually improve some reward or Q-value, with a neural network to approximate some state-value function which allows us to estimate a value for a particular action in some state. DQN also employs experience replay by storing random samples of learning steps which are later recalled and trained on; this improves data efficiency by getting the most value out of a given data set and, by breaking up the pattern with respect to time, it also aims to reduce autocorrelation which may occur in online learning environments such as ours and cause overfitting of the model.

The drawback of DQN is that it is quite unstable; that is to say, it is sensitive to changes in input data, reducing its ability to generalise solutions to problems. In well-defined environments, like the Atari games it was designed to solve, this is acceptable as each instance tends to play out in a similar way when the game is played optimally. I do not feel that this is the case in our environment. Given the stochastic nature of our inputs, no two episodes are likely to play out in the same way and so a standard DQN would find it difficult to converge on a general solution.

Improvements were made to DQN algorithms with the advent of policy gradients. The paper “Trust Region Policy Optimization” (Schulman, et al., 2015) proposed stability improvements over DQN while making minimal sacrifices to the data efficiency by utilizing a policy optimisation function that considers the change between policies at each step. This policy optimisation function considers the advantage at each timestep – that is, how much better a certain new action is as compared to the previous on-policy action – and scales this advantage by some delta coefficient which is a measure of how much this new policy differs from the previous. Once we determine which change provides the maximum advantage, we use a KL-divergence policy to mediate the reward signal by how drastically it changes the policy, helping to modulate wildly changing reward values and increase the stability of our model.

The drawback of TRPO however was its complexity. A lot of calculations needed to happen at each step, and this made it both expensive to run and challenging to implement.

Finally, Proximal Policy Optimization (Schulman, et al., 2017) was proposed as a simplification of TRPO. PPO cuts down on the number of steps and uses a hyperparameter, epsilon, to constrain the distribution change (typically between 0.8 - 1.2, using an epsilon value of 0.2). By removing the incentive for the policy to move too far at a given timestep, we increase the stability of the function without having to calculate a KL divergence[[3]](#footnote-3). This, of course, only gives us an approximation of the results we would see in TRPO, but at a reduced cost. To further reduce instability, PPO uses a minimization function taking the lower bound of the change which will produce results, therefore making the smallest effective change.

For the reasons covered above, in addition to PPO being a popular algorithm in the DRL space today meaning it has a lot of up-to-date documentation, PPO was selected as the algorithm of choice for this project.

# RESULTS

## Definitions

|  |  |
| --- | --- |
| **Placed** | How many items were placed into bins correctly. |
| **Misplaced** | How many times was an item incorrectly allocated. |
| **Discarded** | How many items were discarded without being placed. |
| **Steps taken** | How many timesteps/actions (place, misplace, or discard) were taken total. |
| **Utilization** | How efficiently our bins were packed (I.e., 9 out of 10 slots filled = 90%) |
| **Accuracy** | How many items were correctly placed first try. |

## 1-dimensional bin packing

This environment provided the foundation for my experimentation, and allowed me to investigate the effects of various input parameters on the performance of the RL agent. At each timestep in this environment, the agent could attempt to allocate the current item to one of our bins or discard it; as such, the following 3 results were possible:

1. **Placed**: The object was successfully placed in a bin with sufficient capacity to hold it.
2. **Misplaced**: The agent attempted to place the object in a bin without sufficient capacity, meaning it was carried over to the next timestep to try again.
3. **Discarded**: The agent chose to discard the item without attempting to place it.

Each of these outcomes carried their own reward value which I tuned as follows:

1. **Constant reward values**: The agent receives a reward of +1 for placing an item, and a penalty of -1 for misplacing an item.
2. **Linear reward values**: The agent receives a reward equal to +(item size) for placing an item, and a penalty of –(item size) for misplacing an item.
3. **Asymmetric reward values**: The agent receives a reward of +1 for placing an item, and a penalty of -10 for misplacing an item.

Each of these methods were attempted with:

1. **No discard penalty**.
2. **Small discard penalty**: Penalty equal to ½ of the misplaced item penalty.
3. **Large discard penalty**: Penalty equal to the misplaced item penalty.

The goal of this environment was to increase the accuracy with which items were allocated – that is, minimise the number of misplacements – to demonstrate that a DRL agent was capable of extracting some reward signal from our custom environment and acting on it in such a way as to improve its performance.

This environment showed excellent results as compared to our control environment, with some highlights being:

### Results highlights

* Accuracy increased from **11.19%** to up to **99.73%[[4]](#footnote-4)**.
* Average steps per episode reduced from **546.2** to between **93.9** and **271.6**.
* Average number of items misplaced reduced from **351.6** to between **0.2** and **13.4**.

### Detailed results

While our DRL agent saw improvements across the board, there were interesting variations across the reward methods we used. Our asymmetric reward method saw the highest accuracy of all methods, but this was at the expect of episode length wherein this method scored lowest.

### Constant Reward

(fig. 1a)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Environment** | **Placed** | **Misplaced** | **Discarded** | **Steps** | **Utilization** | **Accuracy** |
| Constant reward, no penalty | 68.3 | 1.1 | 167.2 | 236.6 | 97.9 | 98.41 |
| Constant reward, with discard penalty | 47.6 | 4.5 | 41.8 | 93.9 | 97.73 | 91.719579 |

(fig. 1b)

### Linear Reward

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Environment** | **Placed** | **Misplaced** | **Discarded** | **Steps** | **Utilization** | **Accuracy** |
| Linear reward, no penalty | 52.6 | 2.2 | 74 | 128.8 | 96.36 | 94.88 |
| Linear reward, small discard penalty | 48.1 | 3 | 45.7 | 96.8 | 98.6 | 94.237487 |
| Linear reward, large discard penalty | 46.9 | 13.4 | 41 | 101.3 | 98.1 | 79.64606 |

### Asymmetric Reward

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Environment** | **Placed** | **Misplaced** | **Discarded** | **Steps** | **Utilization** | **Accuracy** |
| Asymmetric reward, no penalty | 75 | 0.2 | 196.4 | 271.6 | 96.65 | 99.73 |
| Asymmetric reward, small discard penalty | 57.1 | 2 | 92.3 | 151.4 | 97.8 | 96.582375 |
| Asymmetric rewards, large discard penalty | 47.8 | 8.4 | 41.8 | 98 | 97.7 | 85.430196 |

### No Penalty

(fig. 1a)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Environment** | **Placed** | **Misplaced** | **Discarded** | **Steps** | **Utilization** | **Accuracy** |
| Constant reward | 68.3 | 1.1 | 167.2 | 236.6 | 97.9 | 98.41 |
| Linear reward | 52.6 | 2.2 | 74 | 128.8 | 96.36 | 94.88 |
| Asymmetric reward | 75 | 0.2 | 196.4 | 271.6 | 96.65 | 99.73 |

(fig. 1b)

### Small Penalty

# CONCLUSIONS

# FURTHER WORK

# References

Amazon, 2022. *Amazon EC2 On-Demand Pricing.* [Online]   
Available at: https://aws.amazon.com/ec2/pricing/on-demand/  
[Accessed 12 03 2022].

Amazon, 2022. *Amazon EC2 Spot Instances Pricing.* [Online]   
Available at: https://aws.amazon.com/ec2/spot/pricing/  
[Accessed 12 03 2022].

David Naori, D. R., 2020. Online Placement of Virtual Machines with Prior Data. *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications,* p. 10.

Engstrom, L. et al., 2020. *Implementation matters in deep policy gradients: a case study on PPO and TRPO.* s.l., ICLR.

Gaul, V., 2022. *Infrastructure as a Service Market Statistics, Drivers and Forecast.* [Online]   
Available at: https://www.alliedmarketresearch.com/infrastructure-as-a-service-IAAS-market  
[Accessed 2022].

Mnih, V. et al., 2013. *Playing Atari with Deep Reinforcement Learning,* s.l.: Google DeepMind.

Schulman, J. et al., 2015. *Trust Region Policy Optimization.* Proceedings of the 32nd International Conference on Machine Learning, PMLR 37:1889-1897, s.n.

Schulman, J., Wolski, F., Dhariwal, P. & Radford, A., 2017. Proximal Policy Optimization Algorithms.

1. This is the normal pricing offered for all instances. (https://aws.amazon.com/ec2/pricing/on-demand/) [↑](#footnote-ref-1)
2. This is a discounted price based on current demand, meaning that lesser-used instance types are offered at a discount. (https://aws.amazon.com/ec2/spot/pricing/) [↑](#footnote-ref-2)
3. This describes the clipping method, but it is also possible to use the KL-divergence penalty method, or indeed a hybrid method using a KL-divergence constraint wrapping our clipping method. An ablation study by OpenAI found the clipping method to work better (Engstrom, et al., 2020). [↑](#footnote-ref-3)
4. Average accuracy seen in the ‘Asymmetric reward, no penalty’ environment. [↑](#footnote-ref-4)