

Computer Games Development CW208

Technical Design Document

Year IV

|  |
| --- |
| Ben Millar  C00236772  28/04/2022 |

Contents

[Technical Design 1](#_Toc102040360)

[Background 1](#_Toc102040361)

[Environments 1](#_Toc102040362)

[1-Dimensional Bin Packing 1](#_Toc102040363)

[1-Dimensional Knapsack 3](#_Toc102040364)

[2-Dimensional Bin Packing 3](#_Toc102040365)

[2-Dimensional Knapsack 4](#_Toc102040366)

[Further Environments 4](#_Toc102040367)

[Driver Code 4](#_Toc102040368)

[Helper functions 4](#_Toc102040369)

[Import dependencies 4](#_Toc102040370)

[Create environment 5](#_Toc102040371)

[Run baseline test (No ML) 5](#_Toc102040372)

[Train RL model 6](#_Toc102040373)

[Save model 6](#_Toc102040374)

[Load model 6](#_Toc102040375)

[Test model 7](#_Toc102040376)

[Compare baseline to model 7](#_Toc102040377)

[Getting real-world data 7](#_Toc102040378)

[Instance Type 8](#_Toc102040379)

[On-Demand Price 8](#_Toc102040380)

[Spot Price 8](#_Toc102040381)

[vCPU 8](#_Toc102040382)

[Memory 8](#_Toc102040383)

[Storage 8](#_Toc102040384)

[Network Performance 8](#_Toc102040385)

[Discount Factor 8](#_Toc102040386)

[Processing data 9](#_Toc102040387)

# Technical Design

## Background

The purpose of this project is to determine both the viability and the efficacy of using deep reinforcement learning (DRL) to solve multi-dimensional bin packing problems. To achieve this, the codebase was split into TWO categories:

1. **Environment code**: This is the code that models the environment we are working with – a 1-dimensional bin packing environment for example – and will be contained in a *python (.py)* file.
2. **Driver code**:This is the code that instantiates the environments described above, and trains DRL models on the same. Each environment will have its own driver code, which will be contained in a *Jupyter Notebook (.ipynb)*.

Each of the environments examined is comprised of a pair of files as described above.

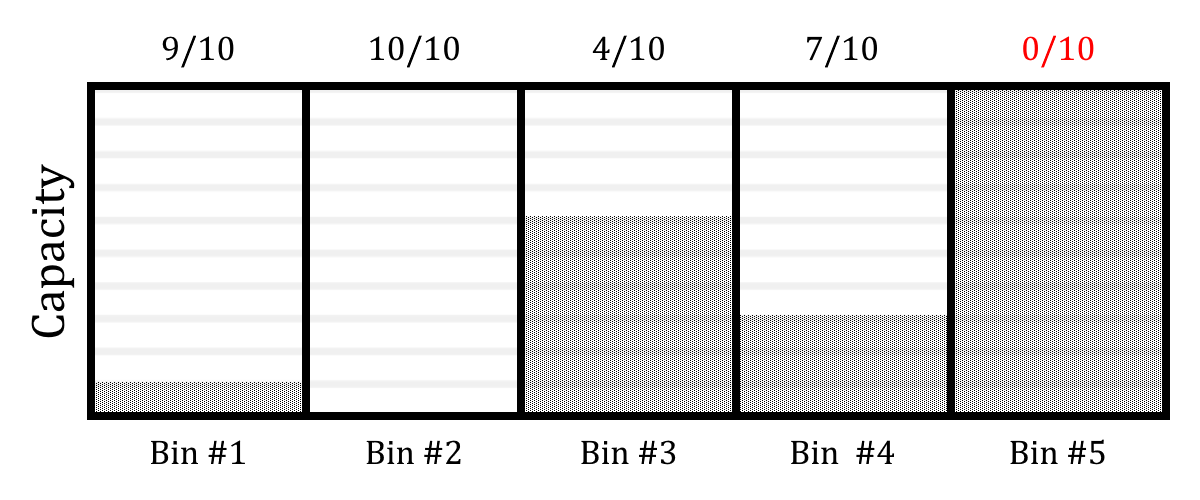
## Environments

I chose to approach this problem with an iterative solution, gradually increasing the complexity of the problem at hand and bringing it closer to the end goal. I will begin by describing the implementation of the simplest environment and from there I will describe only the changes from one environment to the next.

### 1-Dimensional Bin Packing

#### Overview

This environment models a 1-dimensional bin packing problem in which we have some number of bins, or containers, each with some non-negative integer capacity.

****

At each time step, an item is generated of some size between 0 andour maximum capacity(although typically, this was between 1 and ) and we may then take one of the following actions:

1. Discard the item without placing it.
2. Place the item in one of our bins, thus reducing its remaining capacity by the size of the item.

The environment will terminate once we have filled each of the bins to capacity (or to such a capacity that they cannot take any more items of the sizes available), or we have gone over some maximum number of steps (1000 for testing, with 10 bins of capacity 20).

#### Implementation

The environment was modelled as a NumPy array[[1]](#footnote-1) of size *n+1*, where *n* is the number of bins.

Indices 0 → *n-1* contained integers representing the remaining capacity of each bin, and index *n*contained an integer representing the next item to be placed.For example, if we had 5 bins of capacity 10, and the next item to place was of size 3, then our array would be as follows:

The action we take on this environment, then, is denoted by an integer value in the range of our array length. If the action value is in the range , then the item will be placed in the bin at that index. Otherwise, if the action value is equal to *n* (one beyond the last valid bin index)*,* then we will discard the value to be placed and choose a new one.

It should be noted that if we attempt to place an item in a bin that does not have sufficient capacity, then that action will have no effect.

Let us take the example above and say we take the action value *2*. This will place the item into the bin at index position 2 (third element), reducing its capacity accordingly.

A new item will then be created with a value in the range specified; in this case, .

This will continue until all bins have a capacity below the minimum item value (<1, in this case).

### 1-Dimensional Knapsack

#### Overview

#### Implementation

This environment is modelled as a 2-dimensional NumPy array of size *n+1*, taking the form:

where the top row represents the remaining capacities of each knapsack, and the bottom row represents the total value contained within each. As before, the final index of both arrays contains the next item to be placed.

### 2-Dimensional Bin Packing

#### Overview

#### Implementation

This environment is modelled as a 2-dimensional NumPy array of size *n+1*, like the 1D knapsack, but transposed such that each element of the outer array is itself an array of length 2: [x, y]. The final index, as before, holds the next item to be placed.

The behaviour of this environment is much like that of the 1-dimensional bin packing environment, except that each item has a dimensionality of 2 and thus must satisfy the condition

before it can be placed.

### 2-Dimensional Knapsack

#### Overview

#### Implementation

This environment is modelled as a 2-dimensional NumPy array of size *n+1*. It is implemented much like the 2-dimensional bin packing environment, but each subarray is of length 3 to allow a value to be stored alongside the dimensions, as below.

### Further Environments

All additional environments were modelled in the same way as the 2-dimensional knapsack, with additional indices being added to the internal arrays for each additional dimension considered.

## Driver Code

Each environment has its own Jupyter Notebook in which the environment and the deep reinforcement learning agent are initialized. Some additional functionality is also added to aid in logging useful data when testing DRL agents.

The driver code is broken into the following sections:

### Helper functions

Here, I have additional functionality which may be useful in the current environment. Typically, it is custom logging code setup to track certain environment variables. For example, the 1D knapsack environment has logging code which tracks:

* Average number of steps taken per episode.
* Average bin utilization (what percentage of the available space was used).
* Accuracy (percentage of items which were placed in a valid position on the first attempt).
* Minimum value of any bin.
* Maximum value of any bin.
* Average value of all bins.

### Import dependencies

Here we import our custom environment, and the dependencies required for our reinforcement learning:

* Tensorflow 2.7.0
* Gym (OpenAI gym)
* Keras
* Keras-rl2

### Create environment

This is where we instantiate and initialize our custom environment.

### Run baseline test (No ML)

Here we take a control measurement from our environment with no machine learning agent; we simply take random actions at each timestep and log the results.

We run a for loop for some number of episodes, for example 10, with each of these episodes representing a new instance of our environment running from the start. Inside this loop, we initialise our environment and then enter a while loop which will continue to take actions in our environment until we reach some terminal state (all bins are full), or we exceed some maximum threshold of steps.

Finally, at the end of each episode some key data is logged for future analysis.

Code snippet:

MAX\_STEPS = 1000

episodes = 10

**for** episode **in** range(1, episodes+1):

state = env.reset()

steps = 0

done = False

score = 0

**while** **not** done **and** steps < MAX\_STEPS:

action = env.action\_space.sample()

n\_state, reward, done, info = env.step(action)

score += reward

steps += 1

print('Episode: {} Score: {}'.format(episode, score))

print(env.logs)

control\_data.log(env)

env.logs = { 'placed':0, 'misplaced':0, 'discarded':0 }

### Train RL model

Now that we have run a baseline test with no machine learning, we instantiate our DRL model and train it on our custom environment.

We first import our required dependencies, and then set up a path for log files from our training. Then, we instantiate an environment which will be used for training. Finally, we create an instance of the PPO class which uses a multilayer perceptron network of 4 layers comprising:

* 11x input neurons (One for each bin, one for the next item)
* 64x hidden neurons
* 64x hidden neurons
* 11x output neurons (One to place in each bin, one to discard the item)

Code snippet:

import os

import gym

from stable\_baselines3 import PPO

from stable\_baselines3.common.vec\_env import DummyVecEnv

# Will throw an error if these don't exist

log\_path = os.path.join('Training', 'Logs')

env = KnapsackPacking(num\_knapsacks=10, capacity=20)

model = PPO('MlpPolicy', env, verbose=1, tensorboard\_log=log\_path)

model.learn(total\_timesteps=100000)

### Save model

Here, we save the model to file allowing us to use it again later without having to regenerate it from scratch.

Code snippet:

PPO\_Path = os.path.join('Training', 'Saved Models', 'Knapsack\_model')

model.save(PPO\_Path)

### Load model

A past model may then be loaded from the file path specified to avoid retraining a model each time.

Code snippet:

model = PPO.load(PPO\_Path, env=env)

### Test model

Like our baseline test, but here we’re calling ‘predict’ on our model passing in our observed state to determine which action we should take, rather than calling ‘action.sample()’ to choose a random action.

Code snippet:

real\_data = bin\_data(env)

env.logs = { 'placed':0, 'misplaced':0, 'discarded':0 }

MAX\_STEPS = 1000

episodes = 10

**for** episode **in** range(1, episodes+1):

obs = env.reset()

steps = 0

done = False

score = 0

**while** **not** done **and** steps < MAX\_STEPS:

action, \_ = model.predict(obs)

obs, reward, done, info = env.step(action)

score += reward

steps += 1

print('Episode:{} Score:{}'.format(episode,score))

print(env.logs)

real\_data.log(env)

env.logs = { 'placed':0, 'misplaced':0, 'discarded':0 }

### Compare baseline to model

Finally, we compare some of the results of our control and real evaluations to get a quick overview of the relative performance.

Code snippet:

control\_data.print\_data()

Average number of steps taken: 462.3

Average bin utilization: 98.5%

Accuracy: 10.24%

real\_data.print\_data()

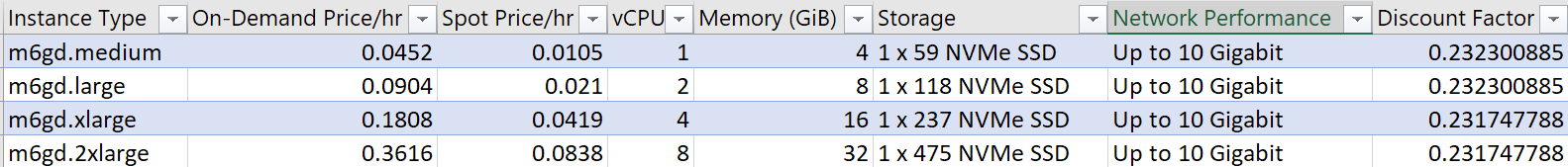
Average number of steps taken: 90.3

Average bin utilization: 97.8%

Accuracy: 85.85%

# Getting real-world data

To model a real-world environment more accurately, I wanted to use real-world data. This was challenging, as cloud service providers do not freely publish this information. To get this data, I transcribed the instance types from AWS EC2, along with pricing information for both on-demand[[2]](#footnote-2) and spot[[3]](#footnote-3) instances. (Note: This data can be found in data/EC2 Pricing.xlsx in the project directory; example provided below)



### Instance Type

This is the name of the instance taking the form [type].[size]. For example, the m6g instance type is powered by ARM-based Graviton2 processors, with the ‘d’ suffix indicating they use SSD storage[[4]](#footnote-4).

### On-Demand Price

This is the price per hour in USD for on-demand instances. That is, instances that are provisioned for the user as and when they are requested.

### Spot Price

This is the discounted price per hour in USD for instances which have been provisioned but are not currently active.

### vCPU

This is the number of virtual CPUs available on this instance.

### Memory

This is the working memory in gigabytes.

### Storage

This is the storage in gigabytes, along with the storage type (E.g., NVMe, SSD, HDD)

### Network Performance

This is the network performance guaranteed for this instance.

### Discount Factor

This is the ratio of the spot price to the on-demand price; this information is used to infer the popularity of a given instance type (with the assumption that the most popular instance types will be discounted less than the least popular instance types) and inform the distribution of instance types in our custom environment.

## Processing data

The data above could not be used in its original form for several reasons. The crucial issues were:

* Instance types were not required for training.
* Some instances did not have pricing information for both spot and on-demand, meaning a discount factor could not be calculated.
* Network performance was similar between all instances, and regardless was not something which was required for our model.
* Storage capacity was represented in forms like “1 x 118 NVMe SSD”, “4 x 900 NVMe SSD”.
* Each type of data had its own range of values; for example, vCPU was in the range 1-128 where storage was in the range 50-60,000. This data had to be normalised so that the neural network did not assume that values of higher magnitudes were more important; a process known as feature scaling[[5]](#footnote-5).

To remedy the above:

1. Unnecessary columns (Instance type, network performance) were removed.
2. Instances where there was not pricing information available for both on-demand and spot were removed (this was a minority).
3. Storage capacity was converted to numerical data (E.g., “1 x 118 NVMe SSD” → 118, and “4 x 900 NVMe SSD” → 3600.
4. All data ***except*** popularity was scaled to the range 0 – 1 using min-max normalisation.
5. Pricing data was kept +ve, but instance size data (vCPUs, Memory, Storage) was inverted to -ve to simplify calculations during training. As such, data is in the range -1 – 1.

The processed data was stored in a CSV and is read into the VM allocation environment at the beginning of training and used to generate new instance types for each time step.

Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Value | CPU | Memory | Storage | Popularity |
| 0.004431 | -0.02362 | -0.00397 | -0.02919 | 0.976822 |
| 0.022547 | -0.05512 | -0.01627 | -0.00417 | 0.97615 |
| 0.009021 | -0.05512 | -0.00807 | -0.02919 | 0.960879 |

Additionally, upon inspection, a handful of instance types had a much lower discount factor than the rest, indicating that they are discounted at a much lower rate than any other instance. This can be seen most clearly in the below graph:

Of the 392 instances remaining, only 6 of them have a discount factor of greater than 0.4. Typically, this may mean that they are exceedingly popular and as such are rarely discount, but upon further investigation these 6 instances are niche instance types providing up to 384GiB of RAM and 48 physical processor cores[[6]](#footnote-6)[[7]](#footnote-7) which are likely not discounted due to their uniqueness. As a result, I elected to remove these instance types from the dataset as I deemed their popularity factor to be erroneously high, and I felt it would throw off the distribution.

Following this, the graph looked as below:

1. https://numpy.org/doc/stable/reference/generated/numpy.array.html [↑](#footnote-ref-1)
2. https://aws.amazon.com/ec2/pricing/on-demand/ [↑](#footnote-ref-2)
3. https://aws.amazon.com/ec2/spot/pricing/ [↑](#footnote-ref-3)
4. https://aws.amazon.com/ec2/instance-types/m6g/ [↑](#footnote-ref-4)
5. https://en.wikipedia.org/wiki/Feature\_scaling [↑](#footnote-ref-5)
6. https://aws.amazon.com/ec2/instance-types/ [↑](#footnote-ref-6)
7. https://aws.amazon.com/about-aws/whats-new/2021/02/introducing-amazon-ec2-m5n-m5dn-r5n-and-r5dn-bare-metal-instances/ [↑](#footnote-ref-7)