

Computer Games Development CW208

Technical Design Document

Year IV

|  |
| --- |
| Ben Millar  C00236772  26/04/2022 |

[Declaration form to be attached]

Contents

[Ben Millar 0](#_Toc101127201)

[C00236772 0](#_Toc101127202)

[20/04/2022 0](#_Toc101127203)

[[Declaration form to be attached] 0](#_Toc101127204)

[Technical Design 2](#_Toc101127205)

[Background 2](#_Toc101127206)

[Environments 2](#_Toc101127207)

[1-Dimensional Bin Packing 2](#_Toc101127208)

[1-Dimensional Knapsack 3](#_Toc101127209)

[2-Dimensional Bin Packing 3](#_Toc101127210)

[2-Dimensional Knapsack 4](#_Toc101127211)

[3-Dimensional Knapsack w/ Real Data 4](#_Toc101127212)

[References 5](#_Toc101127213)

# 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.

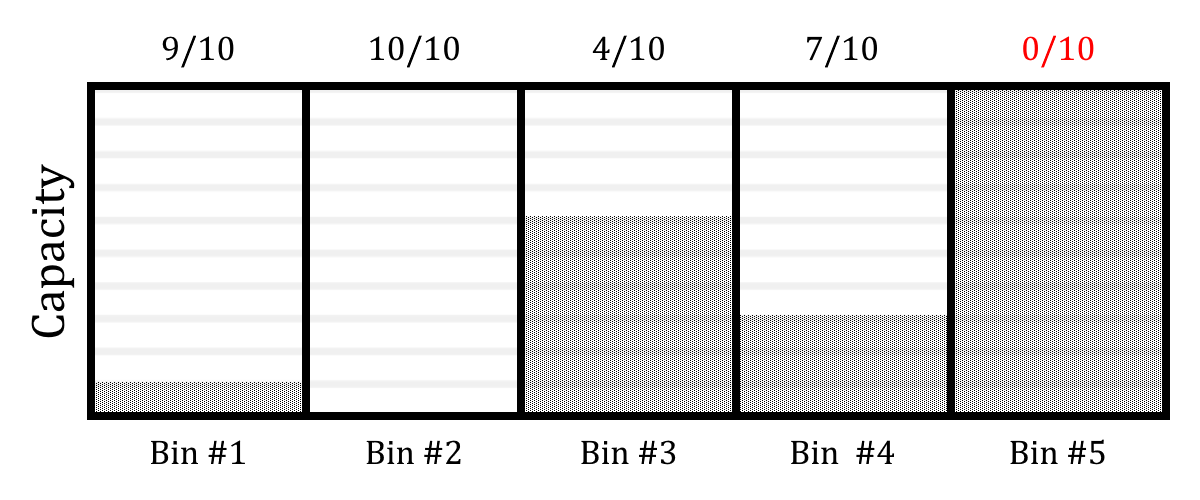
## Environments

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 aims to model 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 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,* 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.

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.

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

### Test model

### Compare baseline to model

1. https://numpy.org/doc/stable/reference/generated/numpy.array.html [↑](#footnote-ref-1)