Chapter 1

Introduction

Cloud computing is a rapidly growing technology with competition from Google, Amazon, Microsoft and others that aims to allow users to run computer programs that are too large, difficult or time consuming to run locally. These services provide the computational resources, e.g. CPU cores, RAM, hard drive space, bandwidth, etc to be able to run such programs. However, as these resources are limited, if users request an unbalance quantity of resources, bottlenecks can occur limiting the number of tasks [1] that can be run on servers simultaneously.

For Google Cloud Services (GCP), Microsoft Azure or Amazon Web Services, their cloud computing facilities contain huge server nodes limiting the probability that such a bottleneck occurs. Therefore this work considers a developing paradigm (Mao et al., 2017) called Mobile Edge Computing (Hu et al., 2015) referred to as MEC. MEC aims to provide users with the ability to run their tasks with greater proximity to them in the network, reducing latency, network congestion and providing better application performance.

Currently Disaster Response (Guerdan et al., 2017), Smart Cities (Alazawi et al., 2014) and the Internet-of-Things (Corcoran and Datta , 2016) are all areas that utilise MECs due to its ability to process computationally small tasks locally with low latency. For example, in Smart Cities, this allows smart intersection systems with road-side sensors or smart traffic lights to minimise cars waiting times at traffic lights and reduce overall congestion (Mustapha et al., 2018). Or for the police to analyse CCTV footage to spot suspicious

[1]Tasks, Programs and Jobs will be used interchangeable to refer to the same idea of a computer programs that has a fixed amount of resources required to compute.

behaviour and to track people between cameras (Sreenu and Saleem Durai, 2019). In the case of Disaster Response, maps can be produced using data from autonomous vehicles' sensors that can support in the search for potential victims and support responders in planning rescues (Alazawi et al., 2014).

With MECs, the problem of bottlenecks is of particular relevant as instead of large server farms that can be geographically distant from the users. Servers are significantly smaller, possibly just high powered desktop computers or single server nodes. This results in greater demand on individual server resources, meaning that efficient allocation of these resources is of growing importance, particularly as more technologies begin to utilise MECs.

However it is believed that there are shortcomings in existing research about resource allocation within MEC (Farhadi et al., 2019; Bi et al., 2019) due to the nature of how task resource usage is determined. Traditionally, a user would submit a request for a fixed amount of resources, i.e. 2 CPU cores, 8GB of RAM, 20GB of storage, that would be allocated to the user. As a result, these resources can't be redistributed until the user finishes with them. But there are good reasons for this form of resource allocation to be used and effective within cloud computing. Primarily, it is simple for the user to decide resource requirements, Cloud Computing companies can use simple linear pricing mechanisms and as it is rare for servers with large resource capacity to have bottlenecks. However it is believed that the problem of bottlenecks within MEC systems, warrant the investigation of an alternative resource allocation mechanisms.

In previous work a novel resource allocation mechanism was proposed (Towers et al., 2020) to allow for significantly more flexibility in determining resource usage with the aims of reducing possible bottlenecks. The mechanism is based on the principle that the time taken for an operation to complete is generally proportional to the resources provided for the operation. An example for this is downloading an image, the time taken is proportional to the bandwidth allocated. This sort of flexibility is similarly true for computing of most tasks [2] or sending back results to the user.
Based on this principle, a modified resource allocation mechanism can be reconstructed such that the users provide the task's total resource usage over its lifetime instead of the task's requested resource usage. This allows for each task's resource usage to be determined by the server rather than

[2]It is well known that some algorithms are not linearly scalable making this principle incompatible with those tasks. Therefore this work considers the case for algorithms that can be scalable linearly and leaves case of non-linearly scalable tasks to future research.

the user increasing a server's flexibility and control. Using this flexible resource allocation mechanism, algorithms proposed achieved 20% better social welfare than a fixed inflexible resource allocation mechanisms in one-shot cases investigated by Towers et al. (2020). This is due to the ability of the algorithms to properly balance task resources, preventing bottlenecks occurring as often, which in turn allowed for more tasks to run simultaneously and to reduce the price.

However that work only considered the proposed mechanism within a one-shot case where all tasks were presented at the first time step, where in all tasks would be auctioned and resource allocated. As a result, in practice the proposed algorithms would require tasks to be processed in batches, such that servers would bid on all tasks submitted every 5 minutes for example. This also meant that while resources could be dynamically allocated at the first time step, they would not change during future batches until the task was completed. This work aims to address these problems.

This was achieved by introducing time into the optimisation problem (outlined in Section 3.1). As a result, tasks now arrive over time and servers can redistribute resources at each time step. However, all previous mechanisms proposed in Towers et al. (2020) are incompatible with this modified online flexible optimisation problem. Therefore this work investigates Reinforcement Learning methods that train agents to optimally bid on tasks based on their resource requirements and efficiently allocate resources to tasks running on a server.

This report is set out in the following chapters. Chapter 2 investigates previous research that this project builds upon within both resource allocation in Cloud Computing and Reinforcement Learning. Chapter 3 proposes a solution to the problem outline in Chapter 1. The solution is implemented in Chapter 4 with testing and evaluation in Chapter 5. Chapter 6 presents the conclusion along with future work for the project.

In addition to this report, the paper referred to as Towers et al. (2020) was written within this academic year and thus considered part of this project's work. A copy of the paper can be found in Appendix A. The paper was also presented at SPIE Defense and Commercial Sensing 2020 as a recorded digital presentation. A copy of the slides can be found in Appendix B with a link to the recording.

Chapter 2

Literature Review

There is a considerable amount of research in the area of resource allocation and task pricing in cloud computing, where auction mechanisms are used to deal with competition. Section 2.1 presents the different approaches to resource allocation and pricing mechanisms in Cloud Computing.

The proposed solution of the project (presented in chapter 3) uses a form of machine learning, called Reinforcement Learning to train agents. Section 2.2 covers the current state-of-the-art algorithms in Q learning and policy gradient research.

2.1 Resource Allocation and Pricing Mechanisms in Cloud Computing

A majority of approaches taken for task pricing and resource allocation in Cloud Computing uses a fixed resource allocation mechanism, such that each user requests a fixed amount of resources for a task from a server. However this mechanism, as previously explained, provides no control for the server over the quantity of resource allocated to a task, only determining the task's price. As a result, a majority of approaches don't consider the server's management of resource allocation. Thus research has focused on designing efficient and incentive compatible auction mechanisms.

Work by Kumar et al. (2017) provides a systematic study of double auction mechanisms that are suitable for a range of distributed systems like Grid computing, Cloud computing, Inter-Cloud systems. The work reviewed 21 different proposed auction mechanisms over a range of important auction properties like Economic Efficiency, Incentive Compatibility and Budget-Balance. In a majority of the proposed auction mechanisms, truthfulness was only considered for the user, thus a Truthful Multi-Unit Double auction mechanism was presented as such that both users and server should act truthfully.

Deep Reinforcement Learning was implemented by Bingqian Du (2019) to learn resource placement and pricing in order to maximise cloud profits. Deep neural network models with Long/Short Term Memory units enabled state-of-the-art online cloud resource allocation and task pricing algorithms that had significantly better results than traditionally online mechanisms with that profit made and number of users accepted. The system considered both the pricing and placement of virtual machines in the system to maximise the profits of cloud providers through the use of Deep Deterministic Policy Gradient (Silver et al., 2014) to train agents. Users would request a type of virtual machine from the system that a server would allocate to a user. The price and placement of the virtual machine is determined by agents with neural networks. To train the agent, real-world Cloud Computing workloads were utilised and allow it to achieve significantly high profits even in worst-case scenarios compared to previous research.

Some approaches have been taken to increase flexibility within Fog Cloud Computing (Bi et al., 2019) through efficient distribution of data centers and connections to maximise social welfare. A truthful online mechanism was proposed that was incentive compatible and individually rational, to allow tasks to arrive over time by solving an integer programming optimisation problem. Similar research in Farhadi et al. (2019), considers the placement of code/data needed to run specific tasks over time where the demands change over time while also considering the operational costs and system stability. An approximation algorithm achieved 90% of the optimal social welfare by converting the problem to a set function optimisation problem.

Previous work proposed the novel resource allocation mechanism and optimisation problem that this project works to expand (Towers et al., 2020). The paper presents three mechanisms for the optimisation problem, one to maximise the social welfare and two auction mechanisms for self-interested users. The Greedy algorithm presented allows for quick approximation of

a solution through the use of several heuristics in order to maximise the social welfare. Results found that the algorithm achieved over 90% of the optimal solution given certain heuristics compared to a fixed resource allocation solution that achieved 70%. The algorithm has polynomial time complexity with a lower bound of $\frac{1}{n}$ however in practice achieves significantly better results.
The work also presented a novel decentralised iterative auction mechanism inspired by the VCG mechanism (Vickrey, 1961; Clarke, 1971; Groves, 1973) in order to iteratively increase a task's price. As a result, a task doesn't reveal its private task value that has particular interest within military tactical networks where countries do not need to reveal the importance of a task to another coalition country but can still allow them to run the task. The auction mechanism achieves over 90% of the optimal solution due to iteratively solving of a specialised server optimisation problem. The third algorithm is an implementation of a single parameter auction (Nisan et al., 2007) using the greedy algorithm to find the critical value for each task. Using the greedy algorithm with a monotonic value density function means the auction is incentive compatible and inherits the social welfare performance and polynomial time complexity of the greedy mechanism.

2.2 Reinforcement Learning

A key characteristic of humans is the ability to learn from experiences that for computers must be programmed in and so researchers have invented a variety of ways for computers to do this. These are broadly grouped into three categories: Supervised, Unsupervised and Reinforcement Learning. Supervised learning uses inputs that are mapped to known outputs, an example is image classification. Unsupervised learning in comparison doesn't have a known output for a set of inputs, instead algorithms try to find links between similar data, for example data clustering.

However both of these techniques are not applicable for cases where agents must interact with an environment making a series of actions that result in rewards over time. Algorithms designed for these problems fall into the category of Reinforcement Learning where agents select actions based on an environment state that generates the result state and reward from the action as shown in Figure 2.1.

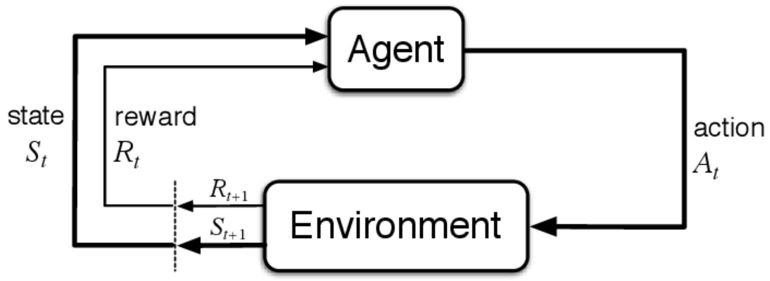

FIGURE 2.1: Reinforcement learning model (Source: Sutton and Barto (2018))

Q-learning (Watkins and Dayan, 1992) is a learning method used for estimating an action-value function, called the Q value, that forms the basis of modern Reinforcement Learning algorithms. The Q value represents the estimated discounted reward in the future given an action in a particular state. Equation (2.1) gives a mathematically description where rewards in the future are discounted. A recursive version of this equation can be formulated as equation (2.2) where the next state is the max Q value of the next state-action. Agents can be trained to approximate the Q value using equation (2.3) through the use of a table of state-action pairs by finding the difference of the best next action and the current Q value.

$$Q(s_t, a_t) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + \cdots\right] \tag{2.1}$$
$$Q(s_t, a_t) = r_t + \gamma \text{max}_a^{'} Q(s_{t+1}, a') \tag{2.2}$$
$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot \big(r_t + \gamma \cdot \text{max}_a Q(s_{t+1}, a) - Q(s_t, a_t)\big) \tag{2.3}$$

However the curse of dimensionally was found to be a major problem for using Q-learning as the number of states or actions increase, the table of state-action pairs grows exponentially. This makes the method impractical for problems with large state spaces due to both the table size and the required training time for agents.

Therefore function approximators are used to circumvent this problem, typically this is done using neural networks due to their ability to approximate any function (Csáji, 2001) and to be trained using gradient descent. Work by Mnih et al. (2013), implemented a deep Q network (DQN) to achieve state-of-the-art performance in six of seven games tested as part of the Atari game engine, with three of these scores being superhuman. This was done using two different neural networks, a model and target network in which the target

network is slowly updated by the model network to act as a slowly updating target Q value. An experience replay buffer was also implemented to enable the agent to learn from previous actions.
Follow up work by Mnih et al. (2015) found that with no modifications to the hyperparameters, neural network or training method; state-of-the-art results were achieved in almost all 49 Atari games with superhuman results in 29 of these games. The work showed that deep neural networks could be trained through observing just the raw game pixels and of the game score over time to achieve scores better than those humanly possibly.

Due to this research, a large number of heuristics have been proposed to the loss function (van Hasselt et al., 2015), network architecture (Wang et al., 2015), experience replay buffer (Schaul et al., 2015) and more to improve the algorithm. A combined agent (Hessel et al., 2017) applying a range of heuristics enabling it to achieved over 200% of the original DQN algorithm in score.

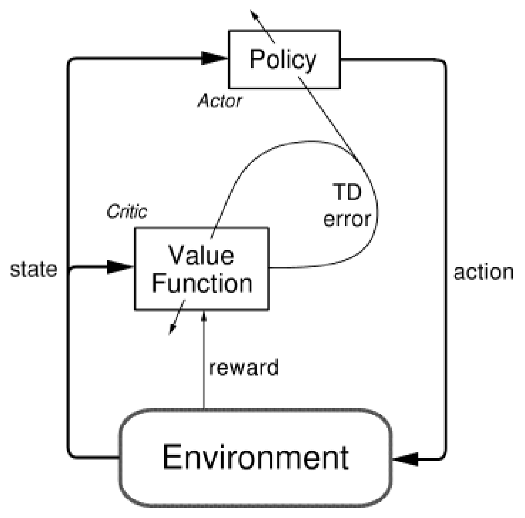

FIGURE 2.2: Actor Critic model (Source: Sutton and Barto (2018))

Policy gradient agents, Figure 2.2, using the base of Q-learning separate the action selection policy from the Q-value function known as the critic. In deep Q networks, actions are selected on the maximum Q-value for all of the actions, however this requires actions to be discretized. By splitting the actions from the Q values, an actor chooses an action based on the environment state allowing for both discrete and continuous action space to be utilised. The critic network is trained almost identically to the Dqn agent except for the use of a soft target update. While the actor network is trained through gradient ascent and the critic evaluation to increase the value. This has the advantage of the action policy being trained directly compare to Dqn agents that use an epsilon greedy action selection policy. However it is well known that Ddpg agents can also get stuck in local maxima's (Matheron et al., 2020). As a result policy gradient has been used to master the game of Go (Silver et al., 2017) and achieved the top 1% ranking in both Dota 2 (OpenAI, 2018) and Starcraft 2 (Vinyals et al., 2017) video games.

Chapter 3

Optimising Resource Allocation in MEC

In Chapter 1, the problem that this project aims to address was outlined along with a short description of the proposed solution. This chapter builds upon that, giving a formal mathematical model for the problem in Section 3.1. Section 3.2 proposes an auction mechanism in order to pay servers for their resources in order to deal with self-interested users and as servers are normally paid for use of their services.

Using the optimisation problem and auction mechanism from the previous sections, agents for both auction and resource allocation are proposed, in Section 3.3, that learn together to maximise a server's profits over time.

3.1 Resource Allocation Optimisation Problem

Using the flexible resource principle, the time taken for a operation to occur, e.g. loading of data, computing a program and sending back results, etc, is proportional to the amount of resources allocated to complete the operation. A modified version of a resource allocation optimisation model can be designed by building upon the formulation in Towers et al. (2020).

A sketch of the whole system is in Figure 3.1. The system is assumed to contain a set of $I = \{1, 2, \ldots, |I|\}$ servers that are heterogeneous in all characteristics. Each server has a fixed resource capacity: storage for the code/data needed to run a task, measured in GB, computation capacity in terms of CPU cycles per time interval, measured in GHz, and communication bandwidth to receive the data and to send back the results of the task after execution measured in Mbit/s. An example of a task could be the analysis of CCTV images with 2GB of data, 5GHz of CPU cycles and 5MB of results data. The resources for a server i are denoted: S_i for storage capacity, W_i for computation capacity, and R_i for communication capacity. The system occurs over discrete time steps that are defined as the set $T = \{1, 2, \ldots, |T|\}$.

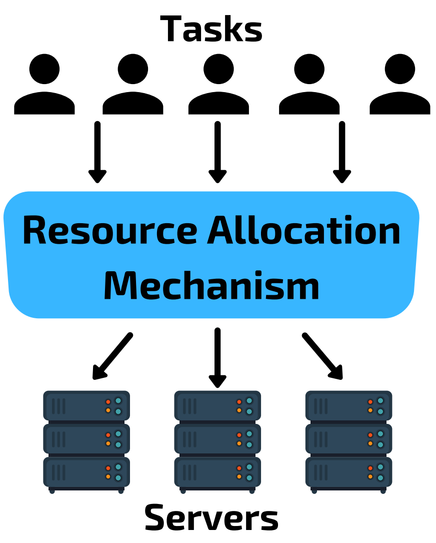

FIGURE 3.1: System model

The system is assumed to contain a set of $J = \{1, 2, \ldots, |J|\}$ heterogeneous tasks that require services from one of the servers in set I. To run any of these tasks on a server requires storing the appropriate code/data on the same server. This could be, for example, a set of images, videos or neural network layers used in identification tasks.
The storage size of task j is denoted as s_j with the rate at which the program is transferred to a server at time t being $s^{'}_{j,t}$. For a task to be computed successfully, it must fetch and execute instructions on a CPU. We consider the total number of CPU cycles required for the program to be w_j, where the number of CPU cycles assigned to the task at time t is $w^{'}_{j,t}$. Finally, after the task is run and the results obtained, the latter needs to be sent back to the user. The size of the results for task j is denoted by r_j, and the rate at which they are sent back to the user is $r^{'}_{j,t}$ on a server at time t.
The allocation of tasks to a server is denoted by $x_{i,j}$ for each task, $j \in J$, and server, $i \in I$. This is constrained by equation (3.1) meaning that a task can only be allocated to a single server at any point in time.

$$\sum_{i \in I} x_{i,j} \leq 1 \qquad \forall j \in J \tag{3.1}$$

$$x_{i,j} \in \{0, 1\} \qquad \forall i \in I, j \in J \tag{3.2}$$

As the task must complete each stage in series, additional variables are required to track the progress of each task stage. $\hat{s}_{j,t}$ denotes the loading progress (equation (3.3)), $\hat{w}_{j,t}$ denotes the compute progress (equation (3.4)) and $\hat{r}_{j,t}$ denotes the sending progress (equation (3.5)) of the task. Each of these variables are updated recursively depending on the progress at the previous time step with the resources allocated. For the compute and sending progress constraints, the resources allocated are clipped so that the previous stage finishes before the next stage starts. For each task stage, progress is limited to the total resource required to prevent unneeded resource allocation (constraints (3.6), (3.7) and (3.8)).

$$\hat{s}_{j,t+1} = \hat{s}_{j,t} + s'_{j,t} \qquad \forall j \in J, t \in T \tag{3.3}$$

$$\hat{w}_{j,t+1} = \hat{w}_{j,t} + w'_{j,t} \lfloor \frac{\hat{s}_{j,t}}{s_j} \rfloor \qquad \forall j \in J, t \in T \tag{3.4}$$

$$\hat{r}_{j,t+1} = \hat{r}_{j,t} + r'_{j,t} \lfloor \frac{\hat{w}_{j,t}}{w_j} \rfloor \qquad \forall j \in J, t \in T \tag{3.5}$$

$$\hat{s}_{j,t} \leq s_j \qquad \forall j \in J, t \in T \tag{3.6}$$

$$\hat{w}_{j,t} \leq w_j \qquad \forall j \in J, t \in T \tag{3.7}$$

$$\hat{r}_{j,t} \leq r_j \qquad \forall j \in J, t \in T \tag{3.8}$$

Every task has an auction time, denoted by a_j and a deadline, denoted by d_j. This is the time step when the task is auctioned and the last time for which the task can be completed successfully. Using this deadline constraint can be constructed such that the sending results progress is equal to the required results data at the deadline time step (equation (3.9)).

$$\hat{r}_{j,d_j} = r_j \qquad \forall j \in J \tag{3.9}$$

As servers have limited capacity, the total resource usage for all tasks running on a server must be capped. The storage constraint (equation (3.10)) is the sum of the loading progress for each task allocated to the server. While the computation capacity (equation (3.11)) is the sum of compute resources used by all of the tasks on a server i at time t and the bandwidth capacity (equation (3.12)) being less than the sum of resources used to load and send results back by all allocated tasks.

$$\sum_{j \in J} \hat{s}_{j,t} x_{i,j} \leq S_i \qquad \forall i \in I, t \in T \tag{3.10}$$

$$\sum_{j \in J} w^{'}_{j,t} x_{i,j} \leq W_i \qquad \forall i \in I, t \in T \tag{3.11}$$

$$\sum_{j \in J} (s^{'}_{j,t} + r^{'}_{j,t}) x_{i,j} \leq R_i \qquad \forall i \in I, t \in T \tag{3.12}$$

3.2 Auctioning of Tasks

While the mathematically description of the problem presented in the previous section doesn't consider any auctions occurring. In real life servers are normally paid for the use of their resources through auctions as discussed in Section 2.1. However the modifications that this project has made, all of the auction mechanisms discussed in the previous Chapter are incompatible due to users not requesting a fixed amount of resources. Nor can the available resources be easily computed as this is dynamic, depending on the different stages of tasks allocated to a server. The modifications also effect the algorithms presented in Towers et al. (2020) as they assume that all of the task stages can occur concurrently that this project does not. Because of this, an outline of the most common auctions is presented in Table 3.1 with their respective properties in Table 3.2 for selecting an auction mechanism to use.

Auction type	**Description**
English Auction	A traditional auction where all participants can bid on a single item with the price slowly ascending till only a single participant is left who pays the final bid price. Due to the number of rounds, this requires a large amount of communication.

Dutch Auction	The reverse of the English auction, where the starting price is higher than anyone is willing to pay with the price slowly dropping till the first participant ”jumps in”. This can result in sub-optimal pricing if the starting price is not high enough or possibly a large number of rounds until anyone bids. Plus due the auctions occurring over the internet, latency can have a large effect on the winner.
Japanese Auction	Similar to the English auction, the Japanese auction is instead over a set period of time lets bids increasing with the last bid being the winner. Because of this, there is no guarantee that the price will converge to the maximum price like the English but the auction has a fixed time length. However factors like latency can have a large effect on the winner and resulting price like the Dutch auction.
Blind Auction	Also known as a First-price sealed-bid auction, all participants submit a single secret bid for an item with the highest bid winning. As a result, no dominant strategy exist as an agent would wish to bid only a small amount more than the next highest price in order to not overpay for an item. But due to there being only a single round of biding, latency doesn’t affect the winner and can occur over a fixed period of time.
Vickrey Auction (Vickrey, 1961)	Also known as a second-price sealed-bid auction, participants each submit a single secret bid for an item with the highest bid winning like the blind auction. However the winner only pays the price of the second highest bid. Because of this, the dominant strategy (referred to as incentive compatibility) for an agent to bid its true value as even if the bid is much higher than all other participants its doesn’t matter as they pay the minimum required for them to win.

TABLE 3.1: Table of Auction mechanisms

The auction properties that this project considers most important are if the fixed time length, which enables fast auctions for processing large numbers of tasks quickly, and Incentive Compatibility such that an optimal strategy actually exists. Because of these two properties, the Vickrey Auction (Vickrey,

Auction	Incentive compatible	Fixed time length
English	False	False
Dutch	False	False
Japanese	False	True
Blind	False	True
Vickrey	True	True

TABLE 3.2: Table of Auction Properties

1961) is selected over the alternatives. An additional advantage of incentive compatibility, is that agents don't need to learn how to outbid another agents, they only need to learn to accurately evaluate each task allowing agents to learn purely through self-play.

However a modification must be made to the auction as servers generate the prices for tasks rather than the task suggesting a price to the servers. Because of this, the auction is reversed, such that the bid with the minimum price wins the task instead of the maximum price. The auction therefore works by allowing all servers to submit their bids for a task with the winner being the server with the lowest price, but as this is a Vickrey auction, the server actually gains second lowest price.

3.3 Server Agents

Using the optimisation formulation and auction mechanism from the previous two sections, the problem can be modelled as Markov Decision Process (Bellman, 1957) as shown in Figure 3.2. This has the advantage of separating out the auction and resource allocation parts of the problem with separate agents acting for each. Subsection 3.3.1 and 3.3.2 proposes agents for each of the auction and resource allocation environments respectively.

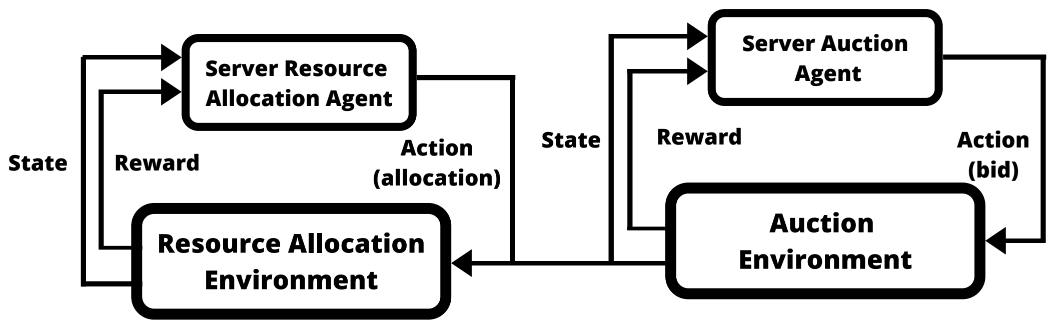

FIGURE 3.2: Markov Decision process system model

Due to the exploratory nature of the project, a range of reinforcement learning algorithms and neural network architectures will be implemented and compared to evaluate the effectiveness of each. These are outlined in Table 3.3 for proposed algorithms and Table 3.4 for networks architectures.

Algorithm	**Description**
Dqn (Mnih et al., 2015)	A standard deep Q network (Dqn) agent that discretizes the action space with a target neural network and experience replay buffer.
Double Dqn (van Hasselt et al., 2015)	A heuristic for the Dqn that modifies the loss function using the target network actions index but the model network value.
Dueling Dqn (Wang et al., 2015)	A heuristic for the Dqn that modifies the network to separates the state value and action advantages that can help understand the environment.
Categorical Dqn (Bellemare et al., 2017)	DQN represents the Q value as a scalar value, the Categorical DQN agent outputs probability distribution over values which is helpful for environments that are stochastic.
Deep Deterministic Policy Gradient (Silver et al., 2014)	Deep Deterministic Policy Gradient (DDPG) allows for continuous actions space, compared to DQN agents that must discretize the action space. This is done through the use of an actor and critic network with target networks for each that are updated with a soft target update method.

Twin Delay DDPG (Fujimoto et al., 2018)	Like the Double Dueling DQN agents, Twin delay DDPG (TD3) includes a couple heuristics for the DDPG algorithm. A critic twin is used to prevent the actor network from tricking the critic network in evaluating a state's Q value. Another heuristic is the delaying the updates for actor network compared to the critic network to allow the critic to out learn the actor to prevent being tricked.
Td3 Central critic	As there are multiple agents working together, the critic used for all of the agents can be same, this is inspired by Lowe et al. (2017) which the critic only evaluates the agents private observation not with complete information of all agents.

TABLE 3.3: Table of the Reinforcement Learning algorithms

Neural Network	**Description**
Artificial Neural Networks (McCulloch and Pitts, 1943)	Originally developed as a theoretically approximation for the brain, it was found that for networks with at least one hidden layer, networks could approximate any function (Csáji, 2001). This made neural networks extremely helpful for cases where it would normally be too difficult for a human to specify the exact function, neural networks can be used to find a close approximation to the true function through gradient descent.
Recurrent Neural Network (Elman, 1990)	A major weakness of artificial neural networks is in its use of a fixed number of inputs and outputs making them unusable with text, sound or video where previous inputs are important for understanding a current input. Therefore recurrent neural network's (RNN) extend artificial neural networks to allow for connections to previous neurons to ”pass on” information. However this was found to struggle from vanishing or exploding gradient during training such that gradients would tended either to zero or infinity over long sequences.

Long/Short Term Memory (Hochreiter and Schmidhuber, 1997)	Long/Short Term memory (LSTM) aims to remedy RNNs problem of vanishing and exploding gradient problems. This is done by using forget gates that determine how much information from the last state would get, allowing for more complex information to be remembered over time compared to RNNs.
Gated Recurrent unit (Chung et al., 2014)	Gated recurrent unit (GRU) are very similar to LSTMs, except for the use of different wiring mechanisms with one less gate, an update gate instead of two forget gates. These changes mean that GRUs run faster and are easier to code than LSTMs. However are not as expressive meaning that less complex functions can be encoded.
Bidirectional (Schuster and Paliwal, 1997)	With RNNs, LSTMs and GRUs, inputs are passed through forward however in understanding an input the subsequent inputs are also required. Bidirectional neural network fixed this by passes in an input twice, once forwards normally and then a second time in reverse. This allows networks to understand the context around an input from both before it and after it.
Neural Turing Machine (Graves et al., 2014)	Inspired by computers, Neural Turing Machines build on long/short term memory by using an external memory module instead of memory being inbuilt to the network. This allows for external observers to understand what is going on much better than other networks due to their black-box nature.
Differentiable Neural Computer (Graves et al., 2016)	An expansion to the Neural Turing Machine that allows the memory module to be scalable in size allowing for additional memory to be added if needed.
Sequence to Sequence Network (Sutskever et al., 2014)	All networks considered above allow for sequences to be passed in while outputting a single output vector. Sequence to sequence (Seq2Seq) networks utilise two sub-networks; an encoder and decoder. The encoder takes a sequence that is encoded which is outputted to the decoder that then outputs another sequence by continually passing in the decoders last input.

TABLE 3.4: Table of Neural network layers

3.3.1 Auction Agents

Traditionally pricing mechanisms (Al-Roomi et al., 2013) rely on mixture of metrics: resource availability, resource demand, quality of service, task resource allocation quantity, etc to determine a price. However these values are difficult to approximate during the auction for this project. So, due to the complexity of deriving this function, reinforcement learning will be used to learn an optimal policy for a server to maximise its profits over time.

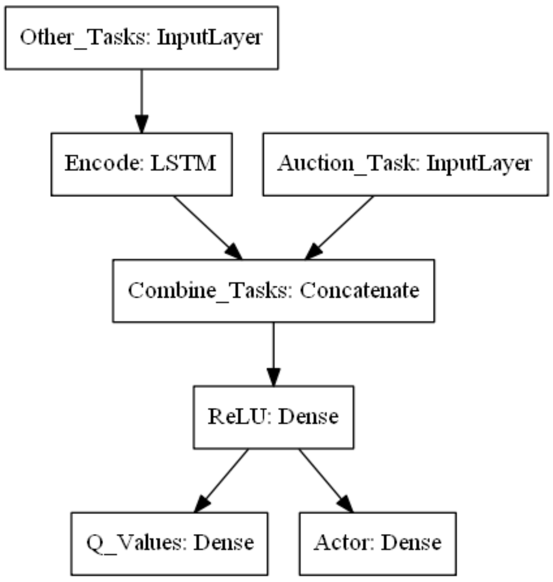

FIGURE 3.3: Task pricing network architecture

The network used for the auction must take into account a server's current tasks and the attributes of the task being auctioned. Because of this, a recurrent network must be used like the RNN, GRU, LSTM and Bidirectional as the number of tasks allocated to a server is unknown. For the DDPG, a single ReLU output will be used while a linear Q value output for DQN agent must be used as shown in Figure 3.3.

3.3.2 Resource Allocation Agents

When a new task is allocated to the server or a task completes a stage, the server will need to redistributed its resources to its tasks. But while the problem of how to allocate resources isn't as complex as the task pricing policy in Section 3.3.1, no obvious heuristics exist. This is because allocation of resources to a single task must take into account other tasks that it will affect in order to balance resources between allocated tasks. Because of this, reinforcement learning agents are proposed to learn this with two different network structures and a weighting heuristic to simplify the optimal function for agents.

The heuristic proposes that a weighted value for the task's resources is used instead of the raw resource usage of a task. While simpler, this approach still has the same expressiveness as the exact resource usage while avoiding the problem of either over or under allocating resources to tasks.

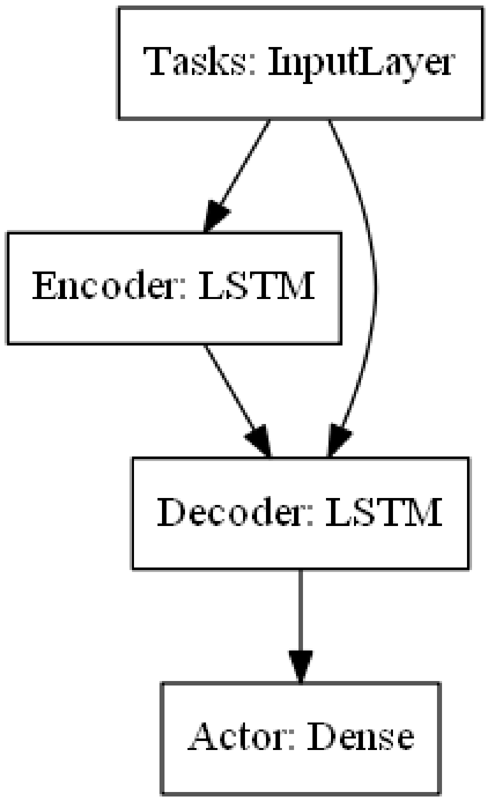

FIGURE 3.4: Multi-task Seq2Seq actor network architecture

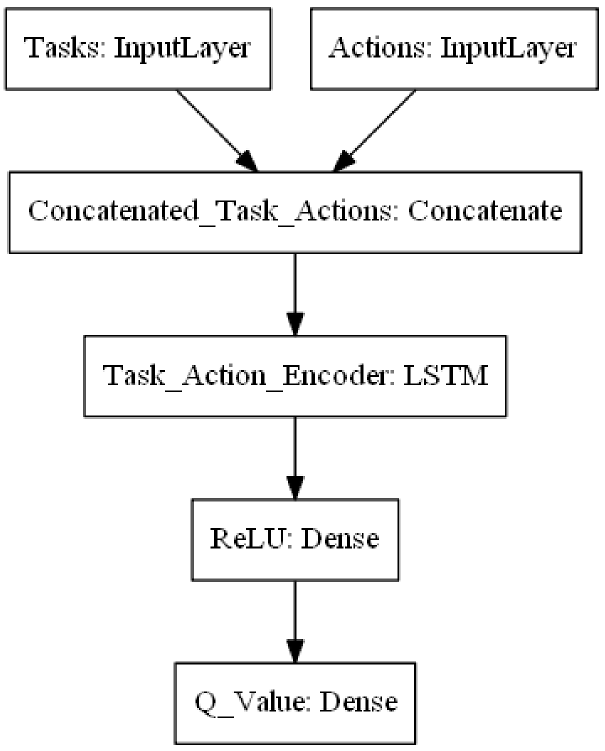
Tasks: InputLayer
Actions: InputLayer
Concatenated_Task_Actions: Concatenate
Task_Action_Encoder: LSTM
ReLU: Dense
Q_Value: Dense

FIGURE 3.5: Multi-task Seq2Seq critic network architecture

To allocate resources to a single task requires also being aware of the resource requirements of other tasks. Because of this, two different neural network structures are proposed, one using a Sequence to Sequence network to allow all tasks actions to be determined simultaneously and the other structure weights a single task at a time. For the Sequence to Sequence network, all tasks are passed in to the encoder that allows the task's attributes to be compressed before passing its output the decoder as shown in Figure 3.4. However Dqn

can't be used to train the network therefore a critic network, Figure 3.5, is required for policy gradient training.
The alternative network uses a simple recurrent neural networks but due to having to output a fixed shape, only a single task can be weighted at a time, as shown in Figure 3.6. Because of this, each task is required to pass through the network. However this network structure can be trained with both Dqn and policy gradient algorithms unlike the Sequence to Sequence network. The Neural Machine Machine and Differentiable Neural Computer were not implemented due to there complexity.

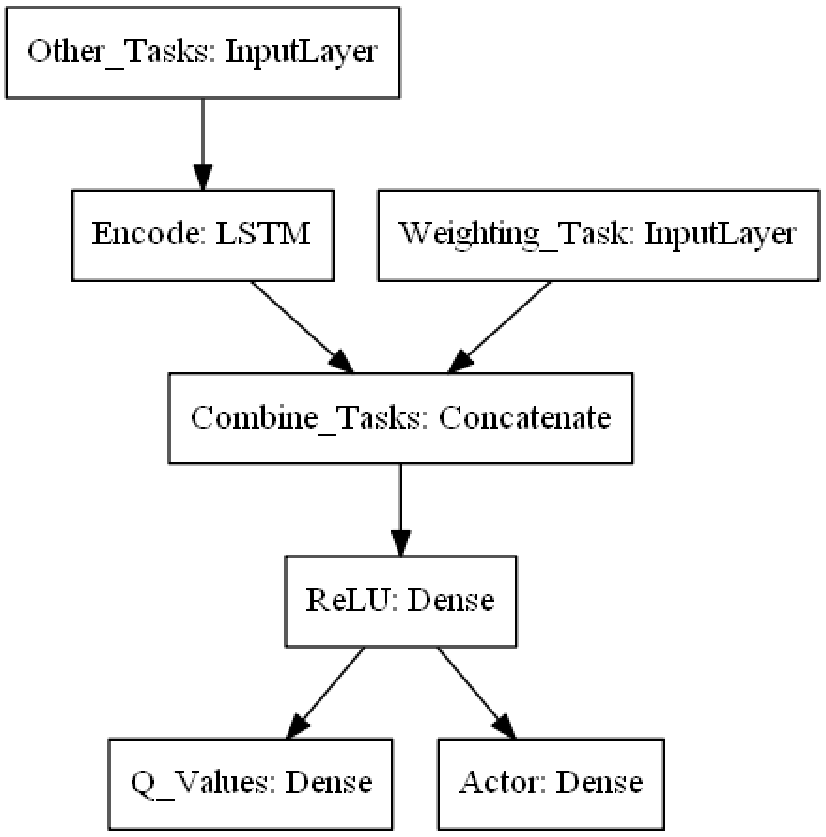

FIGURE 3.6: Single task resource weighting network architecture

Chapter 4

Implementing Flexible Resource Allocation Environment and Server Agents

To test the effectiveness of the proposed optimisation problem and agents from Chapter 3 a Mobile Edge Cloud (MEC) computing network must be simulated for both training and evaluation of agents. However as this is impractical to physically setup such a network to train agents in parallel and offline such a system must be simulated.
This chapter implements a simulation of MEC networks (Section 4.1), server auction and resource allocation agents (Section 4.2) and the training of agents (Section 4.3).

The implementation discussed below is written in Python and available to download from Github [1].

4.1 Simulating MEC Networks

While the aim of the environment is to accurately simulate MEC servers, the implementation of the environment must allow agents to interact and train on the environment efficiently. Therefore it has been implemented as an OpenAI gym (Brockman et al., 2016), the de facto standard for implementing reinforcement learning environments by researchers. However the standard

[1] `https://github.com/stringtheorys/Online-Flexible-Resource-Allocation`

specification must be modified due to the problem being multi-agent and multi-step as shown in Listing 4.1.

```
# Load the environment with a setting
env = OnlineFlexibleResourceAllocationEnv('settings.env')

# Generate the environment state
server_state = env.reset()

done = False
while not done:
    # Check if auction and resource allocation steps
    if server_state.auction_task:
        # Auction actions
        actions = {
            server: auction_agent.bid(state)
            for server, state in server_state
        }
    else:
        # Resource allocation actions
        actions = {
            server: resource_allocation_agent.weights(state)
            for server, state in server_state
        }

    # Take environment step
    server_state, reward, done, info = env.step(actions)
```

LISTING 4.1: Example running of the Online Flexible Resource allocation environment

4.1.1 Weighted Server Resource Allocation

A particular complication of the environment is to distribute server resources due to the fact that the resource allocation agents provide a resource weighting for a task rather than a task's actual resource usage. Originally, the weighting was converted to the actual resource usage by just allocating each task with the weighted resources from the server capacity. However this was found to be inefficient for when a task completed a stage, they would not use all of the resources that were allocated to them. As a result, a large amount of server resources were wasted using this method. Therefore an alternative novel algorithm was implemented to convert the weighting to the actual resources of each task that wasted zero resources.

This was done by checking if any of the tasks would finish a stage given the weighted resources that could be allocated. If this was true, the task was allocated just the resources required for the task to finish the current stage, with the resources removed from the servers available resources. This was then repeated, with the resource weightings being recalculated each time, until no task could finish its current stage with the weighted resources. For the remaining tasks, the standard algorithm was applied with their weighted resources.

For allocating storage and bandwidth resources, servers must be aware of simultaneously allocating resource to task either loading or sending results. Because of this, a tension exists between allocating bandwidth and storage resources for all of the tasks fairly. The algorithm thus chosen to give priority when allocating resources to the tasks sending results as these tasks are more likely to finish, not penalising the server by failing to complete the task within its deadline.

4.2 Server Auction and Resource Allocation Agents

For the server, two policies are required, one for bidding on auction tasks and the other for allocating resources. Subsections 3.3.1 and 3.3.2 propose server agents that use a variety of Reinforcement Learning algorithms (outlined in Table **??**) along with a range of neural networks to learn the agent policy.

The Reinforcement Learning algorithms were originally attempted to be implemented using Tf-Agent (Hafner et al., 2017) and a range of frameworks however due to numerous issues, due to modifying the OpenAI gym specification, that was not possible. Therefore all of the algorithms have been handcrafted, using tensorflow (Abadi et al., 2015) a Python module developed by Google designed for machine learning.

Both deep Q networks and policy gradient algorithms are based on the Q value function (explained in Section 2.2)) which tries to approximate the reward at the next time step. To do this requires a reward function and the agent's next observation to train these agents. The reward function and training observations are detailed in Subsections 4.2.1 and 4.2.2.

A particular problem that this project encountered was using recurrent neural networks with different lengths of data inputs during training. This is as the number of inputs to the networks are dependant on the number of tasks allocated to the server at the time for both the auction and resource allocation agents. However Tensorflow requires all inputs to have a fixed length, due to using Tensors in operations preventing multiple inputs of different lengths being passed to the network at the same time. Initially this was solved by calculating the loss for each input individually then finding the mean loss and gradient to update the networks. However this was found to be computationally impractical requiring over two days to train an single agent over 500 episodes. Eventually, a faster solution was found by padding all the inputs to be the same size using the Tensorflow preprocessing module with the sequence.pad_sequence function. As a result, training became significantly faster making large scale testing practical within a more reasonable time period of 16 hours for 600 episodes.

4.2.1 Agent Rewards Functions

As explained in the background review for Reinforcement Learning (Section 2.2), the Q value is the estimated discounted reward in the future for an action given a particular state. Therefore the rewards that an agent receives for taking an action is extremely important to enable the agent to learn a predictable reward function.

For the auction, the reward is based on the winning price of the task. If the task fails, the reward is instead multiplied by a negative constant in order to discourage the auction agent from bidding on tasks that it wouldn't be able to complete. As the price of zero is treated as a non-bid in the auction, the agent gets a reward of zero in order to not penalise the agent. However, if the agent does bid on a task and doesn't win, the agent's reward is set just below zero at -0.05 as a way of encouraging the agent to change their bid but not large enough to force it to.

$$R(j) = \begin{cases} p_j, & \text{if } \hat{r}_{j,d_j} = d_j \\ -p_j, & \text{else if } \hat{r}_{j,d_j} < d_j \\ 0, & \text{else if } p = 0 \\ -0.05, & \text{otherwise} \end{cases} \tag{4.1}$$

For resource allocation, the reward function is much simpler than the auction agent's reward function, as it only needs to consider the task being weighted at

the time and rewards from other tasks allocated at the same time step. This is because a task must consider its actions in conjunction with the resource requirements of other allocated tasks.
When the task is successfully finished, the reward is set to 1 while the reward is set to -1.5 if the task fails. This makes failing a task more costly than completing a task. But when a task's action is not under consideration, this reward is multiplied by 0.4, as while this rewards impact the task, their value is not as impactful as the reward for the action on a particular task.
These rewards don't consider the price paid for the task instead valuing each task equally for the aim of finishing all tasks not just the valuable ones. Using this information, the reward function is simply the sum of the rewards of the finished tasks in the next time step.

$$R(j') = \sum_{j \in j'} \begin{cases} 1, & \text{if } \hat{r}_{j,d_j} = d_j \\ -1.5 & \text{otherwise} \end{cases} \tag{4.2}$$

4.2.2 Agent Training Observations

In order to update both Dqn and Ddpg critic networks, the loss function used in training (Equation (4.3)) requires the next observation to evaluate the next Q value. However the next observation for an agent is not clear as shown in Figure 4.1. For both the auction and resource allocation agents, there is an unknown number of actions taken by the other agents before its next observation is known.

$$L_\theta = \mathbb{E}_{s,a \sim p(s)} \left[(r + \gamma \max_{a'} Q_{\theta'}(s', a') - Q_\theta(s, a))^2 \right] \tag{4.3}$$

For the resource allocation agent, a trick is implemented such that the next observation for the agent is not the actual next resource allocation observation as shown in Figure 4.1. Instead a generated observation from the resulting server state is used. This is identical to the last case in the figure where no auction occurs between resource allocation steps. The resource allocation Q value is therefore able to approximate the reward from its actions no matter the number of auctions that occur between resource allocation steps.

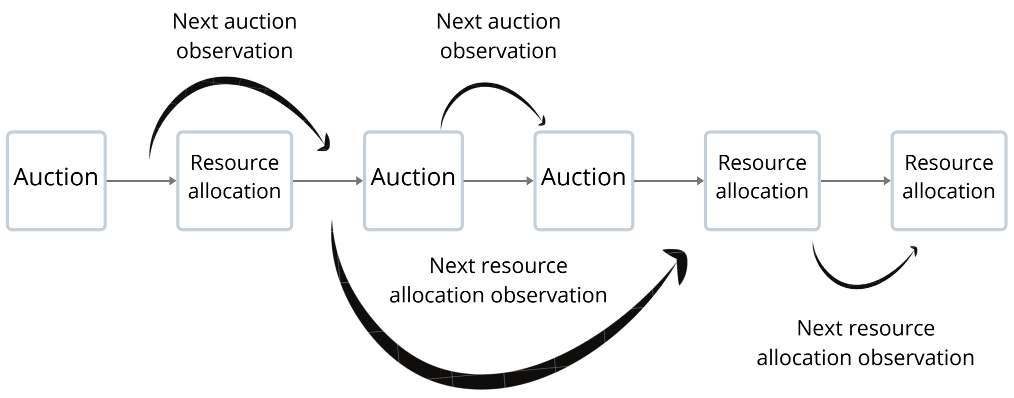

FIGURE 4.1: Environment server agents observations

For the auction agent, the agent's observations require an auction task to select an action (the task bidding price). Therefore a trick like the one implemented for the resource allocation agent can't be used in this case. Thus during training, each server's last observation is recorded, to be used with the next auction observation. But this is a suboptimal solution for the agent as the next observation has an unknown number of resource allocation steps between them. A possible solution that has not been implement in this project is n step prediction (Sutton, 1988). This heuristic modifies the discounted reward to be n steps in the future. This approach could help reduce the amount of randomness in the server's observations and improve bidding performance.

4.3 Training Agents

The first section of this chapter implementation simulations for MEC servers (section 4.1) as a Reinforcement Learning environment. While the second section implements auction and resource allocation agents that can interact with the proposed environment. This allows for the training of agents using a range of algorithms outlined in Table **??**.

Neural networks, the basis of the Reinforcement Learning agents implemented, often require huge amounts of data and high powered Graphical Processing Unit (GPU) to run efficiently. Because of this, Iridis 5, University of Southampton's supercomputer was utilised with GTX1050 GPUs to train these agents for long periods of time and en mass. During training, for each episode a random environment was generated from a list of possible environment settings.

After every 10 episodes, the agents would be evaluated using a set of environments that are pre-generated and saved at the beginning of training. This allowed the same environments to be used between training agents in order to have a consistent basis to compare the agents over time.

To ensure that agents explored the state space as much as possible, during training actions are taken randomly to allow the agents to "explore". This is a key problem within Reinforcement learning, the exploration vs exploitation dilemma (Sutton and Barto, 2018). For the Dqn agents, including the Double Dqn, Dueling Dqn and Categorical Dqn, epsilon greedy actions were taken such that a percentage of the actions are taken random. The exploration factor (epsilon) is initially set at 1 and ending at 0.1 with it linearly decreasing over 100,000 actions. For the Ddpg and Td3 agents, initially Gaussian distribution was used to add randomness to actions (Silver et al., 2014). However this was found to be ineffective, due to half of the actions added a negative value to the action which is unhelpful as actions must be positive. As a result, agents were found to only bid zero in all auctions. Therefore Gamma distribution (Stacy, 1962) was used with a constant shape parameter of 1 and the scale parameter linearly decreasing from 4.5 to 0.5 over 100,000 actions. However further research is required as agents are found to instead bid on every task.

A table of agent hyperparameters used in training can be found in Appendix C.

Chapter 5

Testing and Evaluation

To test and evaluate the effectiveness of the proposed solution and implementation, both functional unit tests and agent training evaluation have been used. To confirm that the environment, agents and training all work as intended, Unit testing has been implemented in Section 5.1.
Section 5.2 compares both Flexible and Fixed resource allocation problems in Subsection 5.2.1 and agents with different training hyperparameters (subsection 5.2.2), Reinforcement Learning algorithms (subsection 5.2.3) and Neural Network Architectures (subsection 5.2.4).

5.1 Functional Testing

To confirm that the implementation of the agents and environment correctly, PyTest, a Python module, has been used. These tests are split into three families: agent, environment and training that are explained in their respective Tables 5.1, 5.2 and 5.3.

Unit Test name	**Description**
Building agents	Constructs all of the agents with any arguments to confirm agents can accept of all its attributes due to multi-inheritance means that only certain arguments are valid for each superclass.
Saving agents	Confirms that agents can successfully save their neural networks and load the network again.

Agent actions	Confirms that all agents can generate valid actions for both bidding and weighting of tasks for both training and evaluation.
Gin config file	Gin is used to set agent and function arguments during training. To test that the file can be successfully parsed and arguments set.
Building networks	Constructs all of the neural networks to confirm that the network accept and return valid inputs and outputs.
Agent epsilon policy	While training agents randomly select actions in order to explore the state space. To tests that the random actions selected are valid and that the exploration factor (epsilon) reduces at a linear rate over time correctly.

TABLE 5.1: Table of Unit tests for Auction and Resource Allocation Agents

Unit Test name	**Description**
Saving and loading of environments	The online flexible resource allocation environment can be saved to a file in its current state: server allocations, future task auctions, time steps and total time steps. This tests that the environment can save and loaded again.
Loading environment settings	Tests that environment settings can be loaded correctly generating a new random environment.
Random action environment steps	Tests that inputs to the auction and resource allocation steps work, random actions are generated to check for unknown edge cases.
Auction step	To confirm the Vickrey auction mechanism is correctly implemented, a range of edges cases are tested to confirm that right price is set and the task is allocated to the correct server in all cases.
Resource allocation step	To confirm the resource allocation step is correct generate the updated state and reward tasks in all cases.
Allocation of computational resources	Checks that the servers correctly allocates computational resources to allocated tasks given resource weightings.

Allocation of storage and bandwidth resources	Checks that the servers correctly allocates storage and bandwidth resources to allocated tasks given resource weightings.
Allocation of all resources	Checks that resources are allocated by the servers correctly given a weighting input for storage, computation and bandwidth resources.

TABLE 5.2: Table of Unit tests for the Online Flexible Resource Allocation Environment

Unit Test name	**Description**
Task pricing training	Tests that the task pricing reinforcement learning agents correctly learning and train with different auction observations.
Resource allocation training	Tests that resource allocation reinforcement learning agents correctly learn and train with different resource allocation observations.
Agent evaluation	Tests that the agent evaluation function during training correctly captures information from the actions taken.
Agent training	Tests that agents correctly train over an environment with different actions and observations.
Random actions training	Tests random action agents using the environment training function to confirm that the function work as intended.

TABLE 5.3: Table of Unit tests for Agent training

5.2 Agent evaluation

Subsection 5.2.1, compares the proposed Online Flexible Resource Allocation environment to an Online Fixed Resource Allocation environment using a linear programming solver to evaluate the effectiveness of the proposed optimisation problem.

In order to compare the implemented agents from Chapter 4, a range of performance metrics are recorded each time the agents are evaluated during training. These metrics are: number of failed tasks, number of completed tasks, percentage of tasks attempted and a histogram of actions taken which together are used to compare the performance between different agents. Differences in Server agents are compared over training environment settings and number of agents, Reinforcement Learning algorithms and network architectures within Subsections 5.2.2 , 5.2.3 and 5.2.4 respectively.

5.2.1 Fixed Vs Flexible Resource Allocation Optimisation Problems

Due to tasks containing only the required resources for their lifetime instead of the proposed resource usage as in a fixed resource allocation scheme. It is difficult to convert the Flexible Resource Allocation Optimisation problem from Section 3.1. Appendix D describes the conversation process for tasks and the Fixed Resource Allocation Optimisation problem with solutions found using an integer programming solver. However due to the complexity a time limit of 2.5 minutes were used meaning that all solution are not optimal. For the flexible environment, a Dqn agent for both auctions and resource allocation were used.

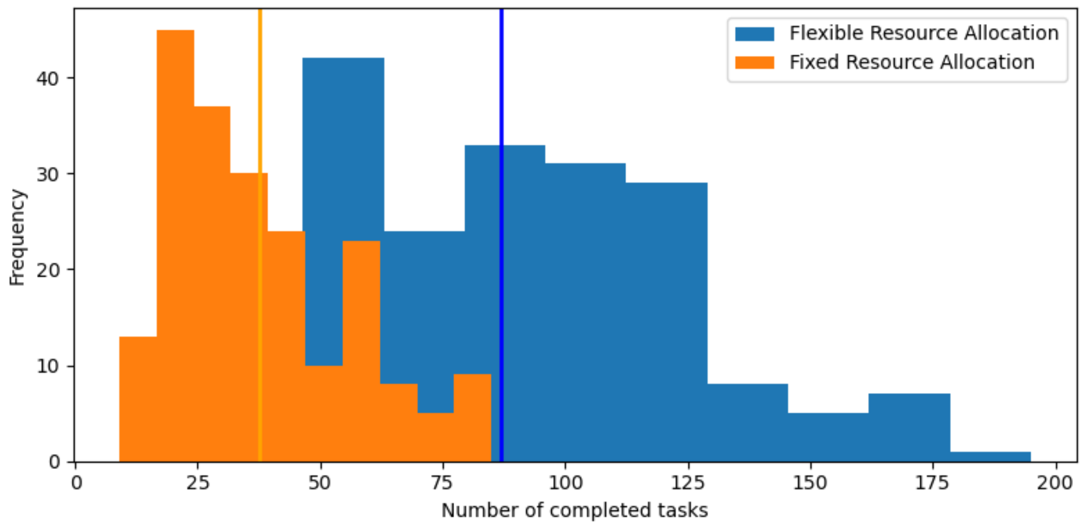

FIGURE 5.1: Number of Tasks completed with Fixed and Flexible resource allocation

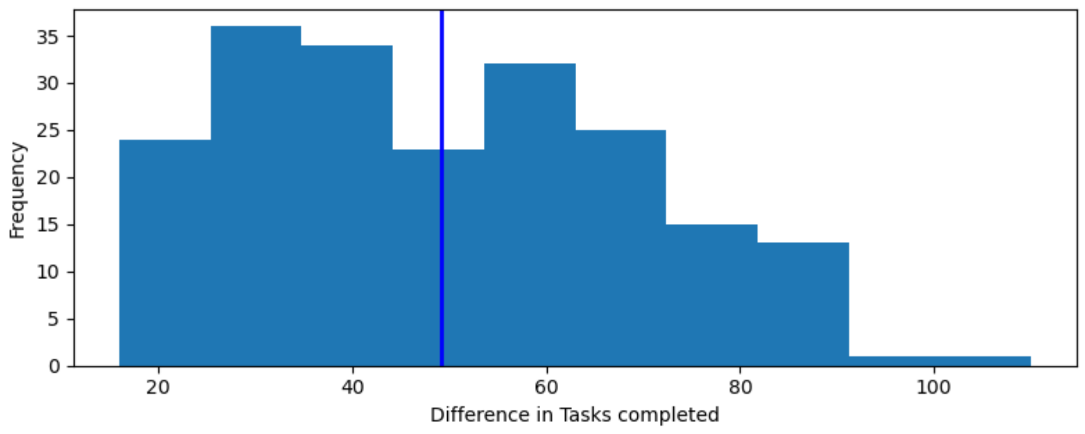

FIGURE 5.2: Difference in number of tasks completed in environment by the Fixed and the Flexible resource allocation agents

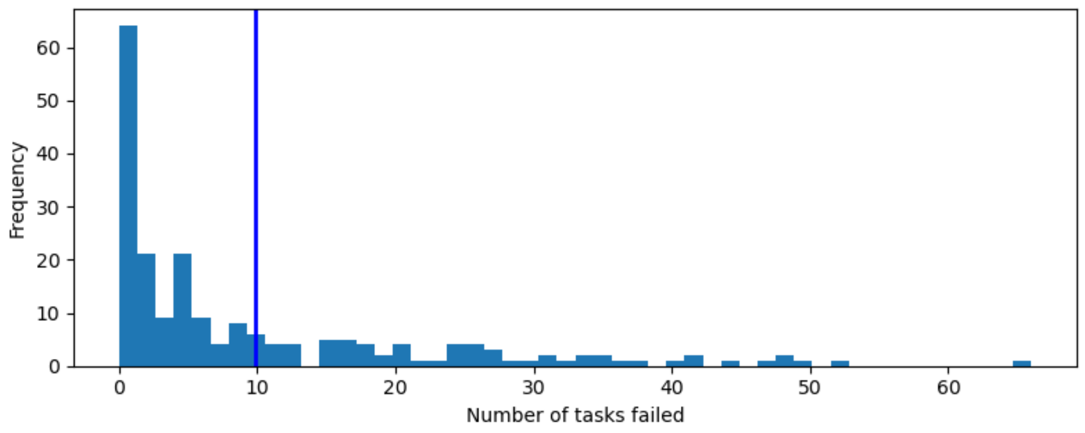

FIGURE 5.3: Number of tasks failed by the Flexible resource allocation agents

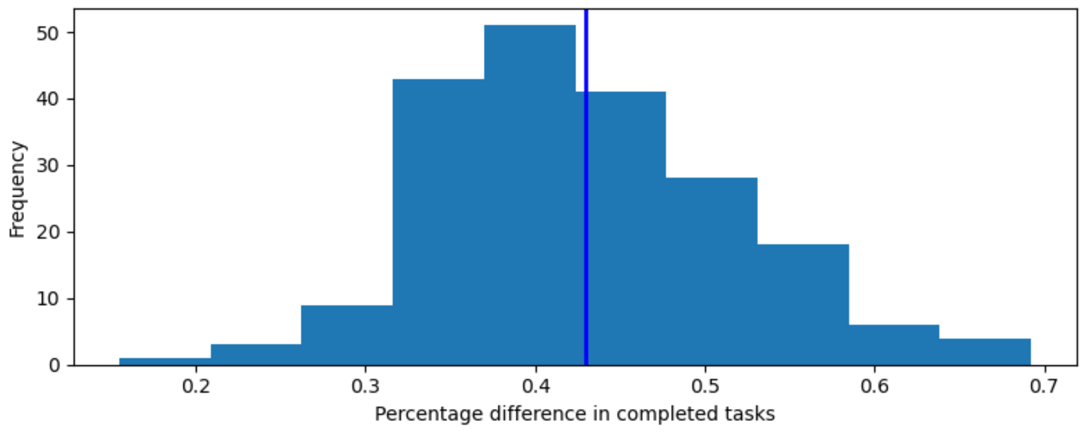

FIGURE 5.4: Percentage difference between the number of Completed tasks by the Fixed and Flexible resource allocation

A histogram of the number of tasks completed by the Fixed and Flexible resource allocation in Figure 5.1 shows that using flexible resource allocation on average over twice the number of tasks can be completed per environment, 35 to 81. Figure 5.2 shows the difference in the number of tasks completed per environment, with on average 51 more tasks being completed by the flexible resource allocation. However on average the flexible resource allocation also fails 10 tasks per environment, as shown in Figure 5.3 with in some cases significantly more.
While these results show that the Flexible resource allocation has significantly better results than the Fixed resource allocation. The results need further analyse to confirm as the fixed resource allocation is not a perfect conversation from the flexible environment and as the solution is found are not optimal.

5.2.2 Environment and Agent number training

This subsection analyses two qualities used by the following two subsections when training agents with different Reinforcement Learning algorithms or neural network architectures. These are the training and evaluation environments and number of agents trained concurrently.

Agents trained must able to adapt to a range of environment settings due to the unpredictability of real-life environments, agent generality is an important measure. In particular that agents do not overfit to particular environments used during training thus making them unreliable with unseen environments.
An advantage of the Vickrey auction, over alternative auctions is that it is incentive compatible, meaning that the dominant strategy for all agents is to bid truthfully. As a result, agents do not need to learn how to ”out bid” each other and can train through purely self-play. However single agents may struggle with a lack of ”genetic” diversity therefore making the number of agents an important hyperparameter to investigate.

This subsection compares the results of agents that are trained on a single environment setting and those trained on multi-environment settings as well as when multiple or single agents are trained together. These are compared using a set of pre-generated environments, 5 from the single environment setting in Figures 5.8, 5.9 and 5.10 and 20 from the multi-environment settings, in Figures 5.5, 5.6 and 5.7 to evaluate all agents.

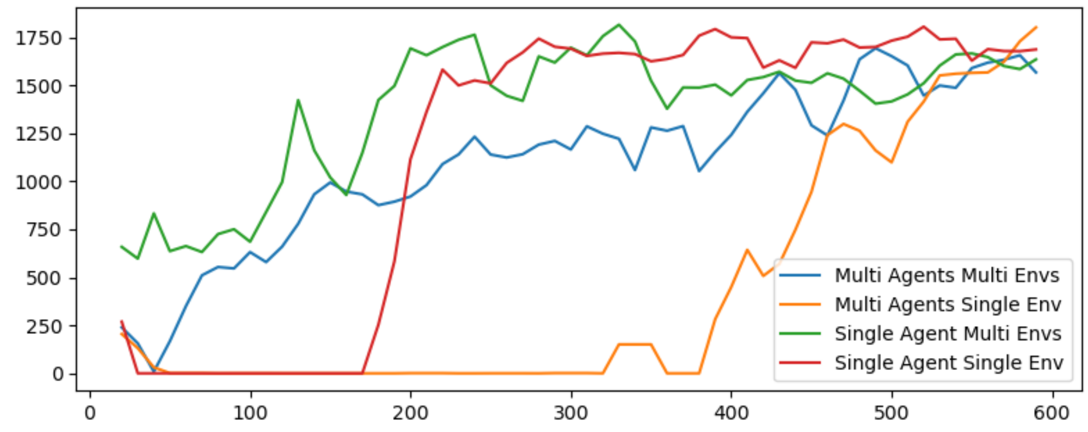

FIGURE 5.5: Environment and number of Agents - Number of completed tasks (Multi-env settings)

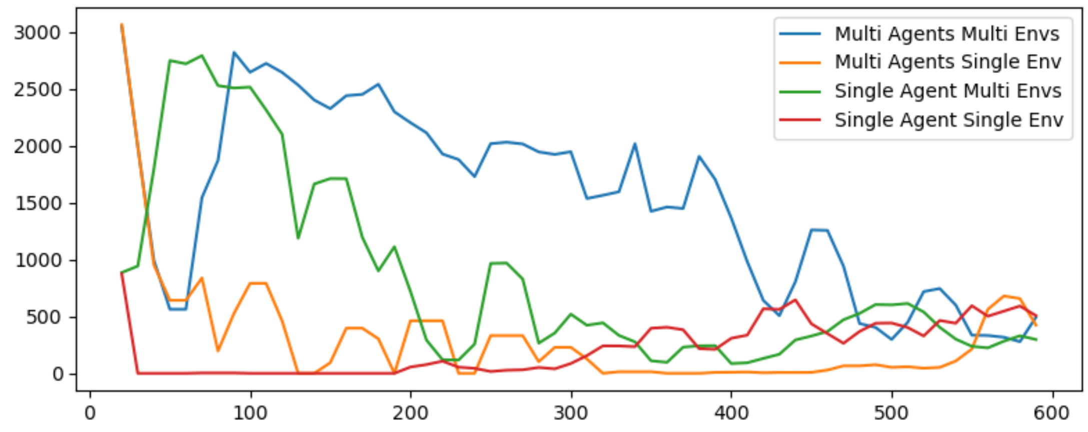

FIGURE 5.6: Environment and number of Agents - Number of failed tasks (Multi-env settings)

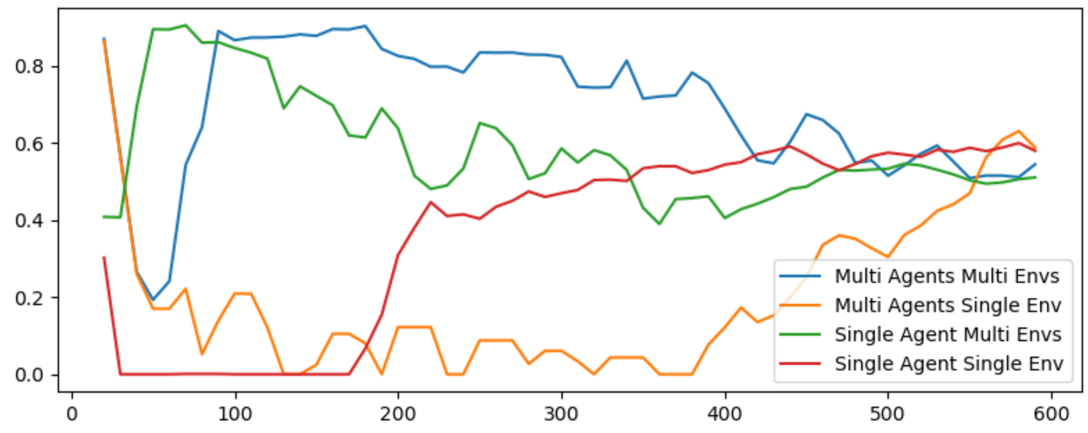

FIGURE 5.7: Environment and number of Agents - Percentage of tasks attempted (Multi-env settings)

In Figure 5.5, agents trained using only a single environment settings, both single and multiple agents variations, took a relatively long time (150 and 350 episodes) for the agents to achieve 25% of its final total tasks completed. In comparison, both multi-environment agents achieved similar results within 50 episodes.

Despite the training speed of the single environment agents, after 600 episodes of training time, all agents no matter the training environments or number of agents trained together achieve very similar results in all metrics (figure 5.5, 5.6 and 5.7. This is surprising that despite single environment agents not being trained on the multi-environment settings, they achieve similar results as multi-environment agents. This means agents are able to generalised to unknown environment effectively alas not as quickly. However more training is required to check if overfitting affects the performance as the agent over learns a particular setting.

This also shows that single agent trained to the same level as multiple agents training together achieve similar results. Meaning that showing that single agent can learn the optimal strategy without needing a diverse "genetic" pool of the multiple agents to train quickly.

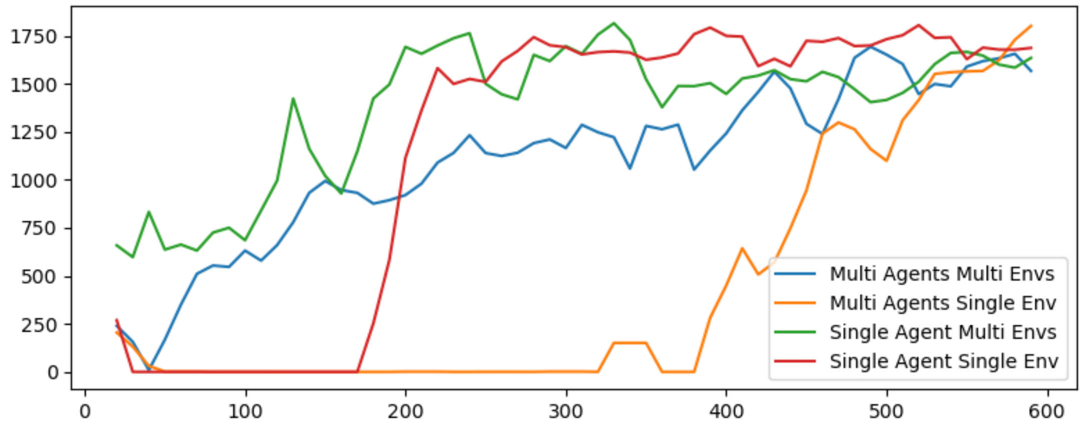

FIGURE 5.8: Environment and number of Agents - Number of completed tasks (Single env settings)

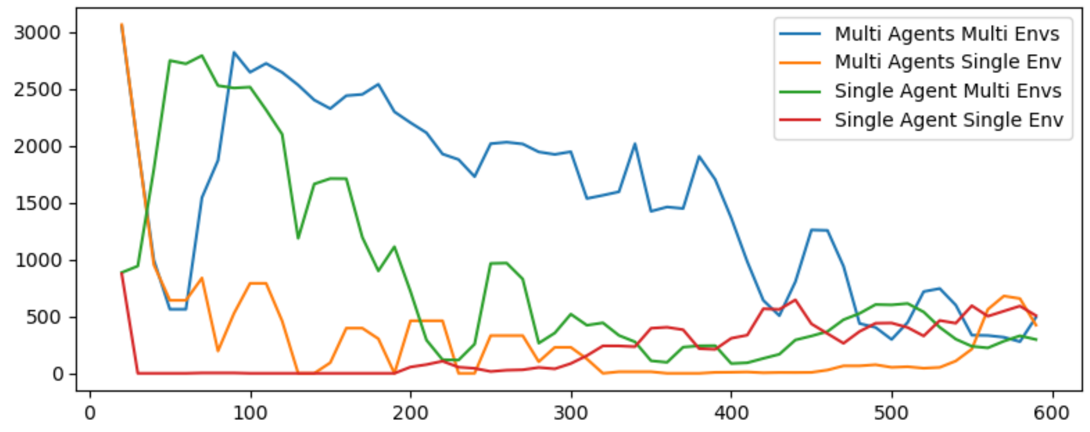

FIGURE 5.9: Environment and number of Agents - Number of failed tasks (Single env settings)

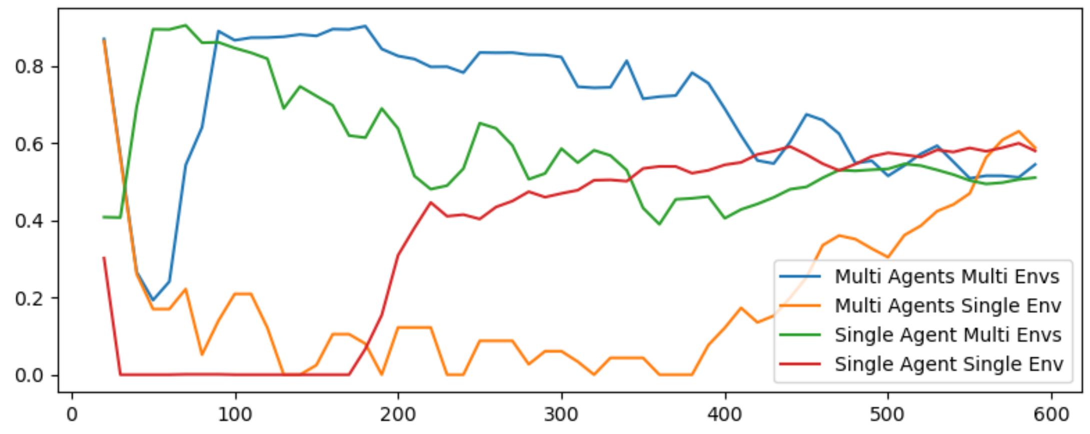

FIGURE 5.10: Environment and number of Agents - Percentage of tasks attempted (Single env settings)

During training, all agents were simultaneously evaluated on the single environment setting as well as the multi-environment settings, shown in Figures 5.8, 5.9 and 5.10. For the single environment agents, they achieve 10% more completed tasks than the multiple environment agents. This is understandable due to single environment being more ”specialised” for the single environment than multi-environment agents. As a result, this allows the single environment agents to maximise profits in specialised environments compared to the multi-environment agents that have learnt to maximise profit over more environments being less ”specialised” for a single environment.

5.2.3 Training Reinforcement Learning Algorithms

To compare Reinforcement Learning algorithms, set out in Table 3.3, each were trained using the algorithms for both auctioning and resource allocation. During training, multiple environments were used for both training and evaluation with three task pricing agents and a single resource allocation agents. This was done because of the analyse in in Subsection 5.2.2.

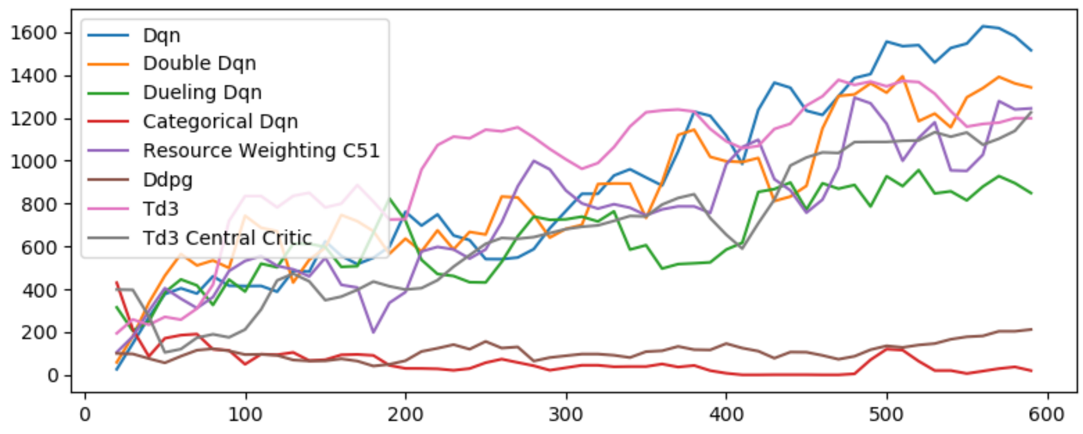

FIGURE 5.11: Reinforcement Learning algorithms - Number of completed tasks

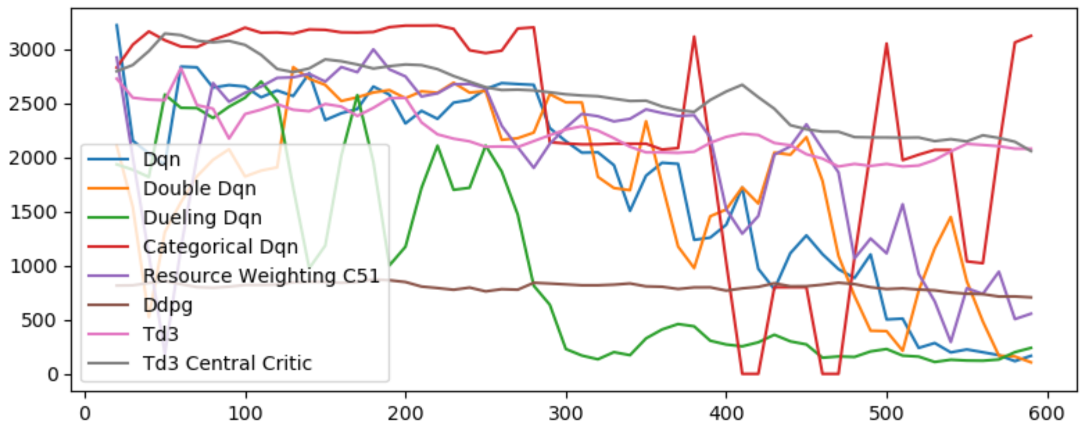

FIGURE 5.12: Reinforcement Learning algorithms - Number of failed tasks

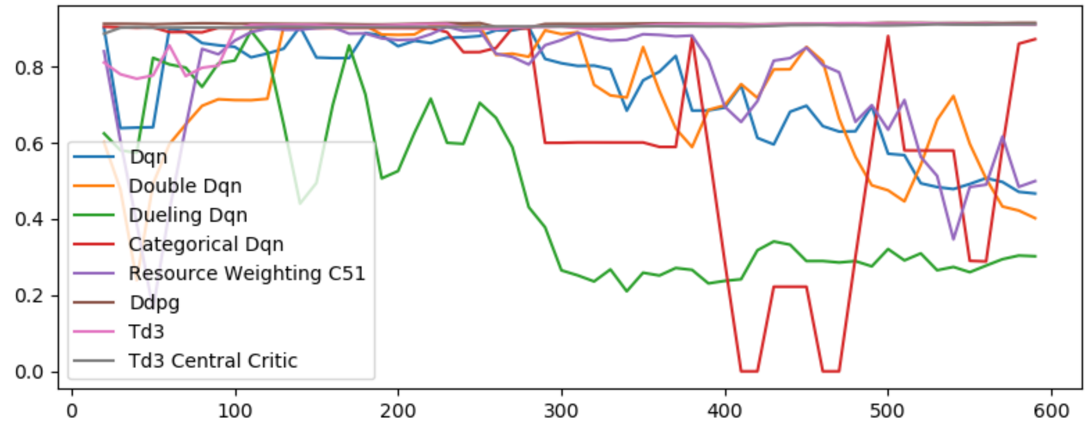

FIGURE 5.13: Reinforcement Learning algorithms - Percentage of tasks attempted

For the Dqn agents, the results did not match expectation as the heuristic Dqn agents achieved fewer completed tasks, Figure 5.11, compared to the standard Dqn agent. This is surprising as previous research (van Hasselt et al., 2015; Wang et al., 2015; Hessel et al., 2017) have found they to achieve better results than the standard Dqn, despite no changes being made to the agents hyperparameters. Possible reasons for this is due to an incorrect implementation, training time or initial network weights. While it is possible to be an incorrect implementation, this is unlikely due to the code matching other implementation. The initial network weights could be a problem, as when the best action is not the optimal actions can cause the network to be trained incorrect at the start but this will self-correct given enough time. Therefore training time is believed to be the primary reason for the heuristic Dqn agents achieving worse result. For future works, doubling the number of training episodes is proposed because of this.

The Categorical Dqn agent struggled to achieving over 50 completed tasks during evaluation after 600 episodes of training. However the reason for this is unknown as a separate training was done using a Dqn for the task pricing and a Categorical Dqn agent for the resource allocation, referred to as ’Resource Weighting C51’. This agent was able to achieve similar results to the other Dqn agent meaning that the task pricing agent fails to work with the Categorical Dqn agent. The reason for this is believed to be due to agent min and max value hyperparameters not due to the implementation. But further research is required to explore Categorical Dqn agents fully and understand why for task pricing this has not worked.

Figure 5.13 shows that the Ddpg agents (Ddpg, Td3, Td3 Central critic) attempted over 80% of auctioned tasks compared to the Dqn agents with only 60%. As a result, agent's can't distributed resource to each of its tasks meaning that the agents have a significantly higher number of failed tasks (Figure 5.12) in comparison to Dqn agent. In terms of completed tasks, the Td3 and Td3 central critic (where all task pricing agents use the same critic and twin critic) are able to achieve results as good as the Dqn agent. This is in comparison to the Ddpg agent which struggled to complete 25% of the amount achieved by the other agents showing that the twin critic heuristics are highly effective for the Ddpg agents.

5.2.4 Training Neural Network Architectures

There are a wide-range of compatible neural network architectures that agents can use, as outlined in Table 3.4. To compare these architectures, five different network architectures are trained: RNN, LSTM, GRU, Bidirectional using an LSTM network and a Seq2Seq network. These networks are trained using the Dqn algorithm due to its constant performance compared to the Ddpg agents as shown in the previous section. The Seq2Seq network used a Dqn Agent with a LSTM network for the task pricing agent and the Seq2Seq network with a Td3 agent for resource allocation. This is as the network architecture is only applicable for the resource weighting agent and requires a policy gradient algorithm to train.

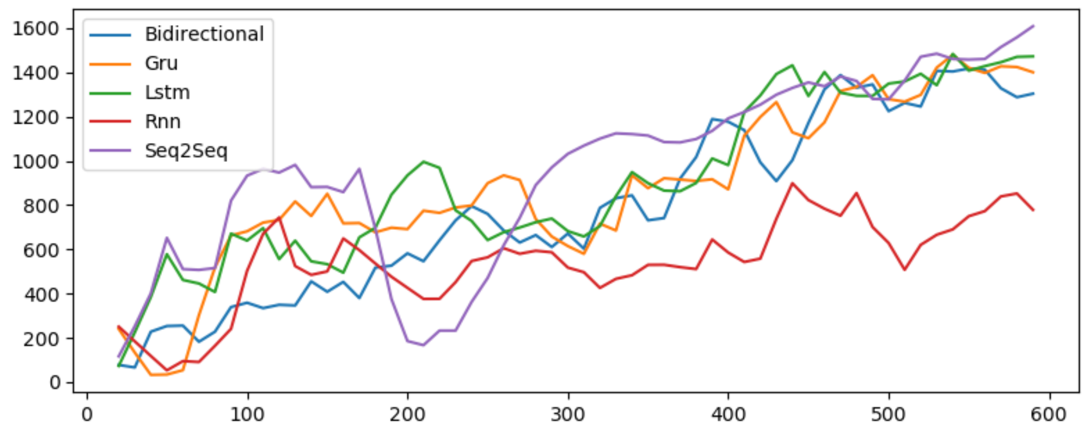

FIGURE 5.14: Network Architecture - Number of completed tasks

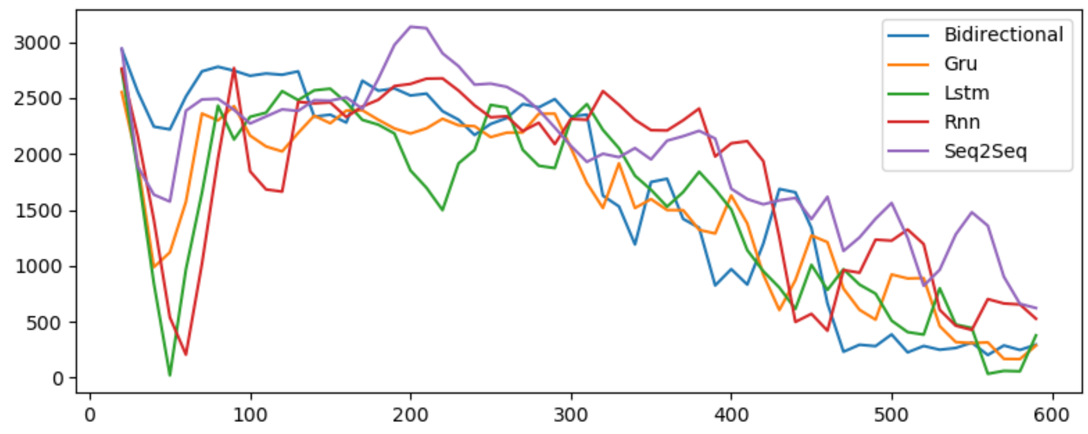

FIGURE 5.15: Network Architecture - Number of failed tasks

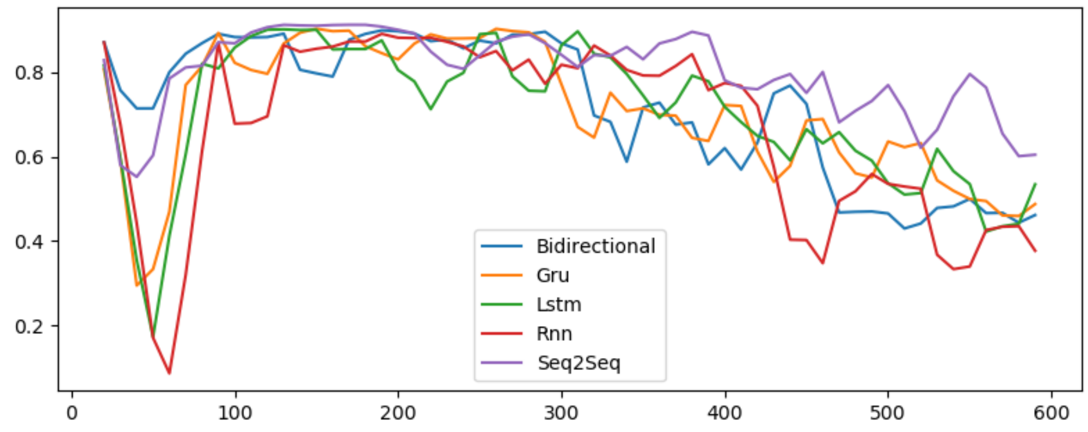

FIGURE 5.16: Network Architecture - Percent of tasks attempted

Figure 5.14 shows that the LSTM, GRU, Bidirectional and Seq2Seq networks all achieved a similar score in the number of tasks completed while the RNN network achieves 30% less. RNNs have a known problem of vanishing or exploding gradients as explained in Table 3.4 making it understandable why the network struggled.
These results are slightly surprising that the heuristic advantages of LSTM and Bidirectional over GRU are not required for agents to approximate the optimal strategy. While ability for the Seq2Seq to learn the task policies together is not a large advantage for the agents in comparison to the other network that weighted tasks individually. However more training time is required till all agents plateau in training which could allow the more complex networks to accel.

Chapter 6

Conclusion and future work

The aim of this project was to expand previous research to fix flaws in the elastic resource allocation optimisation problems for Mobile Edge Computing. This was achieved by introducing the notion of time into the optimisation model. As a result, a new optimisation problem was presented in Section 3.1 along with an auction mechanism to deal with self-interested users (Section 3.2).
Due to the complexity of the problem, Reinforcement learning was applied to learn how to bid on auctioned tasks and allocation the server's resources. By implementing an MEC environment and numerous Reinforcement Learning algorithms, agents were able to be train that learnt the optimal strategy for both task pricing and resource allocation.
The evaluation and testing of the implementation, in Chapter 5, found that agents could efficiently learn the policy achieving significantly more tasks completed than the Fixed Resource Allocation mechanisms. However around $\sim 5\%$ of all tasks were not completed within their time frame. As a result, this project has been viewed as a success however more research and analysis of agents are required before such systems can be implemented into real-life environments.

For future work into this project, this author believes that several additions to the proposed agents could greatly improve their performance like n-step rewards (Sutton, 1988) and distributional ddpg agents (Bellemare et al., 2017; Barth-Maron et al., 2018) that would improve agents due to the environment stochastic nature for agents. An additional heuristic for the policy gradient, would be to use a centralised critic (Lowe et al., 2017) that has been proposed in mix competitive-cooperative environment to help multiple agents working together.

The word count of the Project can be found in Appendix E.