Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Mariana Trench. Using Deep Reinforcement learning for elastic resource management in cloud based environments

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΩΝΣΤΑΝΤΙΝΟΣ ΜΠΙΤΣΑΚΟΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Οκτώβριος 2017

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Mariana Trench. Using Deep Reinforcement learning for elastic resource management in cloud based environments

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΩΝΣΤΑΝΤΙΝΟΣ ΜΠΙΤΣΑΚΟΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 24η Οκτωβρίου 2017.

....	....	....
Νεκτάριος Κοζύρης Καθηγητής ΕΜΠ	Νικόλαος Παπασπύρου Αν. καθηγητής ΕΜΠ	Γεώργιος Στάμου Αν. καθηγητής ΕΜΠ

Αθήνα, Οκτώβριος 2017

....

Κωνσταντίνος Μπιτσάκος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Κωνσταντίνος Μπιτσάκος, 2017.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Τα τελευταία χρόνια βρισκόμαστε μπροστά σε δύο μεγάλες αναταράξεις πάνω στον τομέα της επιστήμης υπολογιστών.
Από τη μία η ανάγκη για επεξεργασία μεγάλου όγκου δεδομένων σε συνδυασμό με τις τεχνολογίες που αναπτύχθηκαν σχετικά πρόσφατα, έφεραν στο πεδίο τις μη σχεσιακές βάσεις δεδομένων .
Πλέον μπορούμε να αποθηκεύουμε και να επεξεργαζόμαστε μεγάλους όγκους δεδομένων, διαμοιρασμένων σε υπολογιστικά νέφη, χωρίς τις παραδοσιακές σχεσιακές βάσεις που θα αποτύγχαναν στο σκοπό.
Η δυναμική κατανομή πόρων σε αυτά τα υπολογιστικά νέφη ονομάζεται ελαστικότητα.

Από την άλλη η τεχνολογική εξέλιξη μας επέτρεψε τη δημιουργία των βαθειών νευρωνικών δικτύων.
Το 2013 συνδυάστηκαν τα βαθειά νευρωνικά δίκτυα με την ιδέα της ενισχυτικής μάθησης.
Ένα πάντρεμα γνωστό και ως Deep Reinforcement Learning που έμελε να ταράξει τα νερά στο πεδίο της μάθησης παγκοσμίως.

Σε αυτή την εργασία προσπαθούμε να συνδυάσουμε αυτούς τους δύο σχετικά νεοσύστατους και ραγδαία αναπτυσσόμενους κλάδους, της ελαστικότητας στα υπολογιστικά νέφη και του Deep reinforcement learning.
Αποτέλεσμα αυτού του παντρέματος είναι η Mariana Trench, ένα σύστημα που ''μαθαίνει'' τις ανάγκες του χρήστη ενός υπολογιστικού νέφους, ύστερα από άμεση διάδραση με το περιβάλλον, προσαρμόζεται ταχύτατα και πετυχαίνει να διαμοιράζει τους πόρους του χρήστη ανάλογα με τις συγκεκριμένες ανάγκες του, χρησιμοποιώντας το Deep RL αλλά και δύο εξαιρετικά αποδοτικές επεκτάσεις του.
Το Full Deep RL και το Double Deep RL.

Τεστάρουμε την υλοποίησή μας πάνω σε απαιτητικές προσομοιώσεις με ευρύ όγκο εισερχόμενων δεδομένων και παραμέτρων για το δίκτυο μας, όπως και με πειράματα πάνω σε πραγματικά υπολογιστικά νέφη (υπηρεσία Okeanos), τα εξαιρετικά αποτελέσματα των οποίων σας παρουσιάζουμε.
Βλέπουμε πως πετυχαίνουμε σημαντική βελτίωση της τάξεως του 60 % στο κέρδος που συλλέγουμε και ταχύτερη σύγκλιση στη βέλτιστη συμπεριφορά από προηγούμενες υλοποιήσεις. Επίσης πετυχαίνουμε να μετατρέψουμε το μεγάλο όγκο εισερχόμενων δεδομένων από μειονέκτημα σε πλεονέκτημα του πράκτορά μας, καθώς όσο μεγαλύτερος χώρος καταστάσεων τόσο αποτελεσματικότερη η λειτουργία του.

Στη συνέχεια δείχνουμε τις προεκτάσεις μιας τέτοιας υλοποίησης, που φέρνει το Deep RL έξω από τα στενά όρια της επεξεργασίας δεδομένων εικόνων ή ήχων και αναζητούμε τα όρια που τίθονται στα περιβάλλοντα στα οποία μπορεί να λειτουργήσει και να ξεπεράσει τον άνθρωπο ένας πράκτορας του Deep RL, εάν αυτά υπάρχουν.

Λέξεις κλειδιά

Ελαστικότητα, Διαχείριση Πόρων, Υπολογιστικό Νέφος, Βαθειά ενισχυτική μάθηση, Βαθειά μηχανική μάθηση, Διπλή βαθειά ενισχυτική μάθηση, Μη σχεσιακές βάσεις, Mariana Trench

Abstract

Over the last years we have witnessed two significant breakthroughs in computer science. On one hand the need to elaborate with Big Data in combination with the newly formed technologies, brought to spotlight NoSQL databases.
We now can store and manipulate Big Data spread across cloud services, without using the traditional Relational Databases which have failed in the field.
In order to manage to dynamically allocate resources for these cloud application IaaS use the so called elasticity in cloud computing.

On the other hand, the rapid development of computational power has allowed us to leverage ideas created in the past and efficiently construct and train Deep Neural Networks.
In 2013 a combination of Deep Neural Networks and Reinforcement learning occurred. This combination became known as Deep Reinforcement learning (Deep RL) shook the world in the field of machine learning.

In this paper we are trying to combine these two newly formed but exponentially growing fields, the field of elasticity in cloud computing for Big Data manipulation and the field of Deep Reinforcement learning.
The result of this effort is Mariana Trench, an agent that ”learns” a user's behaviour and needs in a cloud service and manages to dynamically allocate his resources based on his needs, by using Deep RL and two really promising extensions of Deep RL, Full Deep RL and Double Deep RL.

We test our agent in a variety of demanding simulations with large incoming data and factors for our network and also with experiments in a real cloud service (Okeanod IaaS). We present our significant results. We show that we manage to succeed better profit up to 60 % and fastest convergence than previous approaches. Also we manage to turn the disadvantage of previous version in handling large amount of data into our advantage as the biggest the space of our incoming data, the better Mariana Trench behaviour is.

We then show the extensions of our implementation that push Deep RL beyond the narrow limits of image or sound processing where it is usually applied and we search out the limits in environments where a Deep RL agent can perform better than a human being, if there are any.

Key words

Elasticity, Resource management, Cloud computing, Deep Reinforecement learning, Deep Q learning, Double deep Q learning, NoSQL databases, Mariana Trench

Ευχαριστίες

Με την εκπόνηση αυτής της εργασίας θα ήθελα να ευχαριστήσω τον κ. Κοζύρη που μου έδωσε την ευκαιρία να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα που ξεφεύγει από τα όρια μιας φορμαλιστικής διπλωματικής.
Θα ήθελα να ευχαριστήσω τον μεταδιδακτορικό ερευνητή Ιωάννη Κωνσταντίνου που όχι μόνο μου έδωσε το ελεύθερο αλλά με προέτρεψε να δοκιμάσουμε και να ξαναδοκιμάσουμε συνεχώς καινούρια πράματα πάνω στο Deep Reinforcement Learning, μέχρι να φτάσουμε στο επιθυμητό αποτέλεσμα.
Θα ήθελα να ευχαριστήσω τον διδακτορικό ερευνητή Παναγιώτη Φιλντίση, χωρίς τη βοήθεια του οποίου (πάνω στην κατανόηση των νευρωνικών δικτύων) αυτή η εργασία δε θα είχε ολοκληρωθεί ή θα είχε ολοκληρωθεί τελείως διαφορετική τρία χρόνια πριν.
Τέλος θα ήθελα να ευχαριστήσω τους γονείς και τη γιαγιά μου για τη στήριξη που μου παρείχαν κατά τη διάρκεια των σπουδών μου, ψυχολογική και πρακτική.

Κωνσταντίνος Μπιτσάκος,

Αθήνα, 24η Οκτωβρίου 2017

Contents

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Contents 11

1. Εισαγωγή 15
- 1.1 Κίνητρο εργασίας 15
- 1.2 Σχετικές εργασίες 16
- 1.3 Η δομή του κειμένου 18

2. Πειραματικά αποτελέσματα και συμπεράσματα 19
- 2.1 Υποδομή 19
- 2.2 Εκπαίδευση 20
- 2.3 Τεστ 21
- 2.4 Συμπεράσματα από τα πειραματικά αποτελέσματα 22
  - 2.4.1 Υιοθέτηση της ιδανικής συμπεριφοράς 22
  - 2.4.2 Βέλτιστη εκμετάλλευση της εισόδου 22
  - 2.4.3 Χωρική πολυπλοκότητα 22

3. Introduction 23
- 3.1 Motivation 23
- 3.2 Related Work 24
- 3.3 Thesis Structure 25

4. Necessary theoretical background 27
- 4.1 Tiramola 27
  - 4.1.1 Tiramola's Architecture 27
- 4.2 Cassandra 29
  - 4.2.1 Cassandra's Architecture 31
- 4.3 Ganglia 33
  - 4.3.1 Ganglia Monitoring Daemon (gmond) 33
  - 4.3.2 Ganglia Meta Daemon (gmetad) 34
  - 4.3.3 Using Ganglia to monitor Tiramola 34
- 4.4 Yahoo Cloud Serving Benchmark 34
  - 4.4.1 Using ycsb with Tiramola 35

5. Machine learning- Reinforcment learning 37
- 5.1 Machine learning through the years 37
  - 5.1.1 Markov chains and processes 37
- 5.2 Reinforecment learning 40

5.2.1 Basic RL Algorithms . . . 43
5.3 Q Learning . . . 43
5.3.1 Q learning algorithm . . . 43

6. Neural Networks, Deep Q learning . . . 47
6.1 Neural Networks origins . . . 47
6.1.1 The Perceptron . . . 47
6.1.2 The discovery of hidden layers . . . 50
6.2 The backpropagation breakthrough . . . 51
6.2.1 Gradient Descent . . . 51
6.2.2 backpropagation . . . 52
6.3 Deep Q learning . . . 55
6.3.1 DeepMind's Deep RL . . . 55

7. Mariana Trench . . . 59
7.1 Background . . . 60
7.1.1 Our network . . . 61
7.2 Simple Deep Rl on Mariana Trenchamola . . . 62
7.2.1 The algorithm . . . 63
7.3 Full Deep Rl on Mariana Trenchamola . . . 63
7.3.1 The algorithm . . . 64
7.4 Double Deep Rl on Mariana Trenchamola . . . 64
7.4.1 The algorithm . . . 67

8. Simulation results . . . 69
8.1 Parameterization . . . 69
8.1.1 Testing the Batch size . . . 71
8.1.2 Batch size=20 . . . 71
8.1.3 Batch size=80 . . . 73
8.1.4 Batch size=360 . . . 75
8.1.5 Testing the annealing steps . . . 77
8.1.6 Annealing steps=1000 . . . 77
8.1.7 Annealing steps=5000 . . . 79
8.1.8 Annealing steps=20000 . . . 81
8.1.9 Testing the learning rate . . . 83
8.1.10 Learning rate=0.0025 . . . 83
8.1.11 Learning rate=0.00025 . . . 86
8.1.12 Learning rate=0.000025 . . . 88
8.2 Comparison between different Algorithms in a simple scenario . . . 90
8.2.1 Cluster setup . . . 90
8.3 Comparison between different Algorithms in a complex senario . . . 95
8.3.1 Cluster setup . . . 95
8.4 Conlcusions . . . 99

9. Experimental Results and Conclusion . . . 101
9.1 Experimental results . . . 101
9.1.1 Infrastructure . . . 101
9.1.2 Training . . . 102
9.1.3 Testing . . . 102
9.1.4 Conclusions of experimental Results . . . 107
9.2 Conclusions . . . 108
9.3 Future work . . . 108

Bibliography 111

Chapter 1

Εισαγωγή

Σε αυτή την εργασία παρουσιάζουμε έναν Deep Reinforcment Learning (DRL) πράκτορα που χρησιμοποιείται για να αυξομειώνει το μέγεθος μιας συστάδας υπολογιστών. Ο πράκτορας πετυχαίνει να βρίσκει την ιδανική συμπεριφορά μέσα σε ένα περιβάλλον, εάν του έχει δοθεί μια κατάλληλη συνάρτηση ανταμοιβής και ένα σετ από παραμέτρους που αφορούν τη λειτουργία της συστάδας.

Αυτός ο πράκτορας, τον οποίος αποκαλούμε Mariana Trench [18] αποτελεί έναν επιτυχημένο συνδυασμό των πιο εξελιγμένων αλγοριθμικών προσεγγίσεων πάνω στη βαθειά μηχανική μάθηση (deep machine learning) και του τομέα για δυναμική αναδιαρύθμιση πόρων σε ένα υπολογιστικό νέφος (elasticity). Σε κάθε βήμα ζωής του πράκτορα η ιδανική συμπεριφορά αποφασίζεται ανάλογα με το μέγεθος της συστάδας (πρόσθεσε/αφαίρεσε πόρους/ μην κάνεις τίποτα), με σκοπό να κερδίσει τις μεγαλύτερες ανταμοιβές προϊόντος του χρόνου.

Σαν προέκταση, οι ανταμοιβές μπορούν να προσαρμοστούν στις ανάγκες του εκάστοτε χρήστη, όπως εκείνος κρίνει. Εάν ο χρήστης θέλει να διατηρήσει μικρό κόστος στις παροχές του, μπορεί να δώσει μεγαλύτερο βάρος στο να κρατιέται το μέγεθος της συστάδας μικρό, στη συνάρτηση ανταμοιβής. Εάν ο χρήστης θέλει να έχει μεγαλύτερη και αποτελεσματικότερη διακίνηση δεδομένων μέσα στη συστάδα του, μπορεί να δώσει μεγαλύτερο βάρος σχετικά, στη συνάρτηση ανταμοιβής.

1.1 Κίνητρο εργασίας

Τα τελευταία δέκα χρόνια βιώνουμε μια εκρηκτική αύξηση στις υπηρεσίες υπολογιστικών νεφών. Η εξέλιξη στις τεχνολογίες που αφορούν τα υπολογιστικά νέφη σε συνδυασμό με τη μεγάλη αύξηση στον όγκο πληροφοριών που χρειάζονται να αποθηκευτούν και να επεξεργαστούν, δημιούργησε την ανάγκη για νέες τεχνολογίες που θα μπορούσαν να επεξεργαστούν αυτό τον μεγάλο όγκο δεδομένων. Σαν αποτέλεσμα αυτού, οι παραδοσιακές σχεσιακές βάσεις δεδομένων (SQL) έδωσαν τη θέση τους στις μη σχεσιακές βάσεις δεδομένων (NOSql databases) [23]. Οι μη σχεσιακές βάσεις δεδομένων αναπτύσσονται πάνω σε μια πλειάδα ξεχωριστών κέντρων δεδομένων, όπου αποθηκεύουν τα δεδομένα τους, με διαφορετικά αρχιτεκτονικά μοντέλα από αυτά που οι σχεσιακές βάσεις χρησιμοποιούσαν για την αποθήκευση.

Τα μη σχεσιακά συστήματα ενσωματώνονται πάνω στα υπολογιστικά νέφη με σκοπό να επιτύχουν καλύτερο χειρισμό των πληροφοριών που χρησιμοποιούνται από τις υπηρεσίες νεφών. Η Cassandra, η Hbase, η Mongo DB είναι μόνο μερικά παραδείγματα μη σχεσιακών βάσεων. Σε αυτή την εργασία χρησιμοποιούμε ένα περιβάλλον υπολογιστικού νέφους δομημένο πάνω σε μια βάση δεδομένων, Cassandra. Ο όγκος των δεδομένων που χρειάζονται αποθήκευση σήμερα, στα υπολογιστικά νέφη φτάνει σε μεγέθη τρισάκις εκατομυρίων σε gigabyes ή ακόμα και zetabytes.

Εκτός από το χειρισμό τον μεγάλο όγκο πληροφοριών, το οποίο λύθηκε από τις μη σχεσιακές βάσεις, οι υπηρεσίες υπολογιστικών νεφών είχαν ακόμα ένα πρόβλημα να αντιμετωπίσουν. Πώς να προσεγγίσουν τις συγκεκριμένες ανάγκες ενός συγκεκριμένου χρήστη, ώστε να σιγουρευτούν ότι πολλοί πόροι

δεν πάνε χαμένοι και δε μένουν αχρησιμοποίητοι εξαιτίας μιας λάθος απόφασης; Η απάντηση σε αυτό το πρόβλημα δόθηκε από τον κλάδο της **ελαστικότητας** στα υπολογιστικά νέφη (elasticity). Η ελαστικότητα στα υπολογιστικά νέφη είναι ένας κλάδος μελέτης που αφορά το πώς να μεγαλώνεις ή να μικραίνεις το μέγεθος μιας συστάδας υπολογιστών αυτόματα ανάλογα με τις ανάγκες του χρήστη.

Όπως θα δείξουμε στην επόμενη ενότητα 3.2, υπάρχουν πολλοί τρόποι για να αντιμετωπίσεις το θέμα της ελαστικότητας στα υπολογιστικά νέφη.
To deep reinforcement learning έχει χρησιμοποιηθεί αποτελεσματικά σε περιβάλλοντα όπου η είσοδος είναι εικόνες ή ήχητικά δεδομένα. Ο στόχος αυτής της εργασίας είναι να ερευνήσουμε το αν οι αλγόριθμοι βαθειάς ενισχυτικής μάθησης (Deep Reinforcement Learning), μπορούν να χρησιμοποιηθούν στο περιβάλλον της ελαστικότητας όπου χειρίζονται διαφορετικό τύπο δεδομένων από εικόνες ή ηχητικά δεδομένα.
Όπως θα σας δείξουμε, πετυχαίνουμε ο πράκτορας DRL που κατασκευάζουμε να πετυχαίνει την βέλτιστη συμπεριφορά στο περιβάλλον της ελαστικότητας. Γεγονός που καταδεικνύει ότι οι DRL πράκτορες μπορούν να χρησιμοποιηθούν σε ένα ευρύ πεδίο παρόμοιων καταστάσεων-περιβαλλόντων στην επιστήμη εν γένει, στον τομέα των μηχανικών προβλημάτων, στη θεωρία πληροφορίας, στα παιχνίδια, στην πολιτική, ακόμα και στα προβλήματα κοινωνικών επιστημών κτλ κτλ.

1.2 Σχετικές εργασίες

Σε αυτό το κεφάλαιο θα παρουσιάσουμε εργασίες σχετικές με τη δική μας, κατηγοριοποιημένες σε υποκατηγορίες. Στην πρώτη υποκατηγορία θα μιλήσουμε για σχετικές εργασίες στον τομέα της ελαστικότητας στα υπολογιστικά νέφη και στη δεύτερη υποκατηγορία θα μιλήσουμε για σχετικές εργασίσες στο DRL.

Σχετικές εργασίσες πάνω στην ελαστικότητα στα υπολογιστικά νέφη

O Tiramola [13] είναι μια υπηρεσία που δουλεύει πάνω σε υπολογιστικά νέφη και χρησιμοποιείται για να αυξομειώνει αυτόματα το μέγεθος υπολογιστικών συστάδων ανάλογα με την επιθυμητή πολιτική του εκάστοτε χρήστη. Ο Tiramola αυξάνει ή μειώνει το μέγεθος μιας συστάδας με το να προσθέτει ή να αφαιρεί ένα ένα VM σε κάθε φάση εκτέλεσης, με στόχο να κερδίσει το μεγαλύτερο κέρδος αναλόγως με τις ανάγκες του χρήστη. Οι παράμετροι που εξετάζει το μοντέλο του Tiranola είναι η διακίνηση πληροφοριών μέσα στη συστάδα (throughput), η καθυστέρηση (latency) και ο αριθμός των VMs. O Tiramola μοντελοποιεί τη συστάδα ως μια μορκοβιανή διαδικασία αποφάσεων όπου διαφορετικά μεγέθη υπολογιστικών συστάδων εκπροσωπούν διαφορετικές μαρκοβιανές καταστάσεις. Οι επιτρεπόμενες ενέργειες είναι η αφαίρεση ή η πρόσθεση ενός VM. Για να καταφέρουμε να απομονώσουμε τις πιο πρόσφατες εμπειρίες του πράκτορα, ο Tiramola χρησιμοποιεί μια K-ομαδοποίηση) των εισερόχμενων δεδομένων ώστε το αναμενόμενο κέρδος να υπολογιστεί βάσει της κεντροειδούς αυτής της ομαδοποίησης.

Μια πιο σύγχρονη μορφή του Tiramola από τον Κωνσταντίνο Λόλο [14], είναι μια υπηρεσία που δουλεύει πάνω σε υπολογιστικά νέφη που χρησιμοποιείται ακριβώς όπως ο Tiramola, αλλά διαφοροποιείται ουσιαστικά πάνω στο πως το σύστημα μοντελοποιεί την μονάδα αποφάσεών του. Σε αυτή την εκδοχή του Tiramola χρησιμοποιούνται δέντρα αποφάσεων [7] με σκοπό να επιτύχουν δυναμικό διαμοιρασμό του χώρου καταστάσεων σε μια μαρκοβιανή διαδικασία. Αυτή η προσέγγιση προτείνει έναν αλγόριθμο που μοντελοποιεί το χώρο σαν μια μαρκοβιανή διαδικασία και χρησιμοποιεί δέντρα αποφάσεων για να γενικεύσει επί της εισόδου του.

Η Nefeli [11] είναι μια υπηρεσία δομημένη πάνω σε υπολογιστικά νέφη όπου οι χρήστες παρέχουν στοιχεία για το είδος της εφαρμογής, επιτρέποντας στο πρόγραμμα να τροποποιήσει τις τακτικές προγραμματισμού για να βελτιώσει την απόδοση της εφαρμογής. Οι δημιουργοί της Nefeli ισχυρίζονται πως κατάφεραν να πετύχουν εντυπωσιακή βελτίωση στο συνολικό χρόνο και την ενέργεια που χρειάζεται

για την εκτέληση των φορτιων εργασίας και στις προσοιώσεις αλλά και στα πειράματα σε πραγματικό υπολογιστικό νέφος. Το μειονέκτημα της Nefeli είναι πως είναι αναγκαίο να είναι εγκατεστημένο ένα ενδιάμεσο λογισμικό στο στρώμα διαχείρισης του νέφους, για να λειτουργήσει ομαλά, κάτι που δε συμβαίνει με τον Tiramola

Σχετικές εργασίσες πάνω στο Deep Reinforcement Learning

Η πρώτη προσπάθεια να συνδυαστεί η ενισχυτική μάθηση με τα βαθειά νευρωνικά δίκτυα (BNN) προτάθηκε στη σχετική εργασία της Deepmind [21] όπου BNN χρησιμοποιήθηκαν για να προβλέψουν τη συμπεριφορά σε ένα περιβάλλον και να αποφασίσουν την καλύτερη δυνατή απόφαση που μπορούσε να πάρει ένας πράκτορας αναφορικά με την κατάσταση στην οποία βρισκόταν κάθε φορά το περιβάλλον. Η ονομασία του αλγορίθμου ήταν Deep Q Learning, μιας και για να υπολογίζει τους στόχους του BMN χρησιμοποιούσε τις εξισώσεις του Bellman. Ο πράκτορας χρησιμοποιεί πίνακες με pixels εικόνων στην είσοδό του, προκειμένου να αποφασίσει την επόμενη κίνησή τους.

Εδώ [6] η BEM χρησιμοποιείται προκειμένου ο πράκτορας να μπορέσει να παίξει το γνωστό παιχνίδι Go και πάλι χρησιμοποιώντας πίνακες με pixels εικόνων σαν είσοδο, όπου μοντελοποιούμενος σαν BNN προσπαθεί να βρει την ιδανική συμπεριφορά στο περιβάλλον του παιχνιδιού. Αυτός ο πράκτορας ονομαζόταν AlphaGo και έγινε αρκετά γνωστός καταφέρνοντας να κερδίσει τον δεύτερο καλύτερο παίκτη στον κόσμο στο GO με τελικό σκορ 4-1.

Τέλος στη σχετική εργασία [34], η BEM χρησιμοποιήθηκε προκειμένου να δημιουργηθεί ένας πράκτορας που μπορεί να λειτουργήσει και να βρει τη βέλτιστη πολιτική μέσα σε ένα περιβάλλον φυσικών επιστημών. Οι συντελεστές δημιούργησαν έναν αλγόριτθμο που δε χρησιμοποιεί μοντέλα του περιβάλλοντος και βασίζεται σε μια ντετερμινιστική πολιτική που μπορεί να λειτουργήσει σε περιβάλλοντας με συνεχή κατανομή επιτρεπτών ενεργειών. Η προσέγγισή τους καταφέρνει να λύσει αποτελεσματικά πάνω από 20 φυσικά προβλήματα, όπως το πρόβλημα

In [34], DRL is used in order to create an agent that can manage itself in physics based environments. The authors created an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces. Their approach robustly solves more than 20 simulated physics tasks, including classic problems such as ισορροπίας ενός ρομπότ σε δοκό, το πρόβλημα χειρονακτικής επιδεξιότητας ενός ρομπότ, το πρόβλημα κίνησης με περπάτημα ενός ρομπός και το πρόβλημα οδήγησης αμαξιού από ρομπότ. Και αυτή η προσέγγιση χρησιμοποιεί πίνακες εικόνων σαν είσοδο.

1.3 Η δομή του κειμένου

Η δομή του κειμένου είναι η εξής:

Στο κεφάλαιο 4 αναλύουμε το θεωρητικό υπόβαθρο της κατασκευής μας. Παρουσιάζουμε την αρχιτεκτονική του Tiramola, πάνω στην οποία χτίζουμε την εκτέλεσή μας. Επίσης παρουσιάζουμε τη μη σχεσιακή βάση δεδομένων, Cassandra, πάνω στην οποία τρέχει ο Tiramola, την Ganglia την πλατφόρμα μέσω της οποίας παίρνουμε μετρικές για το σύστημά μας και το YCSB μια δομή του yahoo, στα πλαίσια του ελεύθερου λογισμικού που χρησιμοποιείται για να παράγουμε μια ποικιλία διαφορετικών φόρτων εργασίας στην κατασκευή μας με σκοπό να τεστάρουμε την επιτυχή λειτουργία της.

Στο κεφάλαιο 5 κάνουμε μια παρουσίαση των βασικών θεμελίων πάνω στη θεωρία της μηχανικής μάθησης και της ενισχυτικής μάθησης και επικεντρώνουμε το ενδιαφέρον μας πάνω στο Q learning, την μορφή ενισχυτικής μάθησης που θα χρησιμοποιήσουμε πάνω στα βαθειά νευρωνικά δίκτυα προκειμένου να χτίσουμε τη Mariana Trench.

Τα νευρωνικά δίκτυα και η βαθειά ενισχυτικής μάθηση παρουσιάζονται στο κεφάλαιο 6, όπως και ένας πρότυπος αλγόριθμος του Deep Q Leargning που θα χρησιμοποιηθεί για να δομήσουμε πάνω του τη μονάδα αποφάσεων της Mariana Trench.

Στο κεφάλαιο 7 παρουσιάζουμε τη Mariana Trench. Κάνουμε μια εμβάθυνση στην αρχιτεκτονική και τη δομή του δικτύου μας και παρουσιάζουμε τις τρεις διαφορετικές αλγοριθμικές εκδοχές που θα χρησιμοποιήσουμε, το Deep Q learning,το Full Deep Q learning και το Double Deep Q learning.

Στο κεφάλαιο 8 παρουσιάζουμε τα αποτελέσματα του πράκτορά μας μέσα σε ένα κύκλο από αρκετά απαιτητικές προσομειώσεις οι οποίες χρησιμεύουν επίσης και για το καλιμπράρισμα του δικτύου μας. Τεστάρουμε τον πράκτορά μας σε ένα απλό και σε ένα πιο σύνθετο σενάριο.

Στο κεφάλαιο 9 τεστάρουμε το σύστημά μας σε ένα πραγματικό περιβάλλον υπολογιστικού νέφους, συλλέγουμε και κρίνουμε τα πειραματικά αποτελέσματά μας. Παράλληλα βγάζουμε συμπεράσματα για το σύστημά μας και προσφέρουμε υλικό για μελλοντικές σχετικές εργασίες πάνω στον κλάδο.

Chapter 2

Πειραματικά αποτελέσματα και συμπεράσματα

Σε αυτό το κεφάλαιο παρουσιάζουμε τα πειραματικά αποτελέσματα που συλλέξαμε τεστάροντας τη Mariana trench σε πραγματικό χρόνο σε πραγματικά περιβάλλοντα. Χρησιμοποιούμε τον Okeano για το υπολογιστικό νέφος που χρειαζόμαστε. Έχουμε φτιάξει μια συστάδα υπολογιστών πάνω στους οποίους τρέχουμε την Cassandra, και περιοδικά πυροδοτούμε τη βάση μας με αιτήματα χρησιμοποιώντας την υπηρεσία του ycsb. Συλλέγουμε τις μετρήσεις μας χρησιμοποιώντας το XML API της Ganglia και τέλος χρησιμοποιούμε τη μονάδα αποφάσεων της Mariana Trench για να πάρουμε μια απόφαση μέσα στο περιβάλλον, αφού πρώτα την έχουμε εκπαιδεύσει. Το προγραμματιστικό περιβάλλον που χρησιμοποιούμε είναι η Anaconda [2] και η βιβλιοθήκη που μας παρέχει τα κατάλληλα εργαλεία για τα νευρωνικά δίκτυα και την εκπαίδευσή τους είναι το Tensorflow της Google [1].

Αρχικά ας παρουσιάσουμε τα κύρια κομμάτια της υλοποίησής μας.

2.1 Υποδομή

Έχουμε μια συστάδα 16 εικονικών υπολογιστών στο υπολογιστικό νέφος του Okeanos [12]. Πυροδοτούμε φορτία με αιτήματα στη βάση μας με το ycsb [40], τα οποία ακολουθούν μια ημιτονική κατανομή από αιτήματα για αποθήκευση ή ανάσυρση δεδομένων από τη βάση μας. Το ποσοστό αποθηκεύσεων και ανασύρσεων είναι τυχαίο. Χρησιμοποιούμε κάθε κόμβο της συστάδας μας σαν αποδοχέα των αιτημάτων μας, μιας και στην Cassandra κάθε κόμβος μπορεί να εξυπηρετήσει αιτήματα και δεν υπάρχει κάποιος κεντρικός κόμβος όπως σε άλλες μη σχεδιακές βάσεις, όπως η Hbase. Στέλνουμε τα αιτήματά μας χρησιμοποιώντας το kamaki API [10], από έναν υπολογιστή. Ο υπολογιστής ”σπάει” τον αριθμό των αιτημάτων σε ίσα κομμάτια ισάριθμα με τον αριθμό των κόμβων στην υπολογιστική μας συστάδα. Η Mariana trench δημιουργεί ένα νήμα για κάθε κόμβο και ύστερα στέλνει το δικό της μερίδιο από αιτήματα στον αντίστοιχο κόμβο. Κάθε δέκα δευτερόλεπτα συλλέγουμε μετρικές χρησιμοποιώντας το telnet για να επικοινωνήσουμε με το XML API της Ganglia [9]. Οι μετρικές που χρησιμοποιούμε για να παραστήσουμε μια κατάσταση (S) της συστάδας μας είναι οι εξής:

- Ο αριθμός των εικονικών υπολογιστών (VMs) στη συστάδα μας.
- Η καθυστέρηση στην εξυπηρέτηση των αιτημάτων μας.
- Η αποτελεσματική διακίνηση δεδομένων μέσα στη συστάδα μας.
- Το ποσό καταναλισκώμενης κρυφής μνήμης στη συστάδα μας.
- Ο αριθμός από λειτουργίες/αιτήματα που μπορεί να εξυπηρετήσει η συστάδα μας στην παρούσα στιγμή.
- Ο αριθμός από λειτουργίες/αιτήματα που εξυπηρέτησε η συστάδα μας στην προηγούμενη κατάσταση. Χρειαζόμαστε αυτή την πληροφορία για να αποφασίσουμε αν ο αριθμός αιτημάτων που θα

εξυπηρετήσει μελλοντικά η συστάδα μας είναι πιο πιθανό να αυξηθεί ή να μειωθεί, να αποφασίσουμε δηλαδή την κλίση της καμπύλης των αιτημάτων.

- Το ποσό ελεύθερης μνήμης στη συστάδα μας.
- Το ποσοστό της κεντρικής μονάδας επεξεργασίας που μένει αχρησιμοποίητη.
- Το ποσό της μνήμης των buffers της συστάδας μας.
- Το ποσό της διαθέσιμης μνήμης στη συστάδα μας.
- Το ποσό της διαμοιραζόμενης μνήμης στη συστάδα μας.
- Το ποσό ελεύθερου χώρου στο δίσκο της συστάδας μας.
- Το ποσό των bytes που εισέρχονται σε κάθε κόμβο την παρούσα στιγμή στη συστάδα μας.
- Το ποσό των bytes που εξέρχονται σε κάθε κόμβο την παρούσα φάση στη συστάδα μας.

Η συνάρτηση ανταμοιβής είναι

$$Reward = 0.01 * throughput - 0.00001 * latency - 2 * VMs \tag{2.1}$$

μιας και θέλουμε να κρατήσουμε χαμηλά το κόστος των εικονικών μηχανών ενώ παράλληλα να πετυχαίνουμε μεγάλη διακίνηση δεδομένων στη συστάδα μας και μικρή καθυστέρηση στην εξυπηρέτηση των αιτημάτων μας. Σε κάθε βήμα εκτέλεσης ο πράκτοράς μας παίρνει μια απόφαση, συλλέγει μετρικές, υπολογίζει τη συνάρτηση ανταμοιβής, παίρνει μια νέα απόφαση και εκτελεί μια νέα ενέργεια.

2.2 Εκπαίδευση

Αρχικά εκπαιδεύουμε τον Double Deep Q learning, Mariana trench πράκτορά μας για 20000 βήματα. Τα βήματα μείωσης της e-greedy πολιτικής είναι 2000. Αυτό σημαίνει ότι για τα πρώτα 2000 βήματα ο αλγόριθμός μας κάνει τελείως τυχαίες ενέργειες, προκειμένου να εξερευνήσει καλύτερα και σε μεγαλύτερο βάθος το περιβάλλον και μόνο στο τέλος τέλος της εκπαίδευσης όλες οι ενέργειες που κάνει αποφασίζονται εξ ολοκλήρου βάσει της μονάδας αποφάσεων της Mariana trench (δηλαδή της καλύτερης απόφασης για ύψιστο κέρδος)

Χρησιμοποιούμε 620 βήματα προ εκπαίδευσης, δηλαδή τα βήματα που αρχικά παίρνει ο πράκτοράς μας τελείως τυχαία, κινούμενος τυχαία στο περιβάλλον μέχρι να γεμίσει τον buffer για την τεχνική της επανάληψης εμπειριών (experience replay). Όπως υπολογίζουμε κατά το καλιμπράρισμα του δικτύου μας στο κεφάλαιο 8 θέτουμε τους κάτωθι παράγοντες:

- Το μέγεθος του buffer μας για την επανάληψη εμπειριών είναι 360 παρελθοντικές εμπειρίες.
- Το learning rate μας είναι 0.00025.

2.3 Τεστ

Αφού τελειώσει το στάδιο της εκπαίδευσης, τεστάρουμε τις επιδόσεις της Mariana trench για 2000 βήματα. Τεστάρουμε και τις τρεις διαφορετικές εκδοχές της Mariana trench, το Simple DQN, το Full DQN και το Double DQN. Παρατηρούμε ότι ο πράκτοράς μας μαθαίνει γρήγορα το περιβάλλον τους και συγκλίνει προς την βέλτιστη συμπεριφορά, συλλέγοντας μεγάλα κέρδη. Όταν ο πράκτοράς μας είναι ο Simple DQN, τότε σπαταλάει λίγο περισσότερο χρόνο μέχρι να συγκλίνει προς τη βέλτιστη λύση, μιας και δε χρησιμοποιεί διαφορετικό δίκτυο για τους στόχους και διαφορετικό για τις αποφάσεις όπως δείχνουμε στο κεφάλαιο 6 7. Βλέπουμε τα αποτελέσματα στις εικόνες 9.1, 9.2, 9.3 .

Όταν χρησιμοποιούμε τον καλύτερο πράκτορά μας, τον Double DQN με μεγαλύτερο σετ εμπειριών για εκπαίδευση, ήτοι 60000 παρελθοντικές εμπειρίες, βλέπουμε τα βελτιωμένα αποτελέσματα στην εικόνα 9.4

2.4 Συμπεράσματα από τα πειραματικά αποτελέσματα

Τα συμπεράσματα που βγάζουμε από τα πειραματικά αποτελέσματα δε διαφέρουν από τα συμπεράσματα που βγάλαμε στο κεφάλαιο με τις προσομοιώσεις 8.

2.4.1 Υιοθέτηση της ιδανικής συμπεριφοράς

Η Mariana trench πετυχαίνει γρήγορα και αποτελεσματικά να υιοθετήσει τη τη βέλτιστη συμπεριφορά, ασχέτως του μεγέθους του σετ με τις εμπειρίες εκπαίδευσης. Αν εξαιρέσουμε την προσέγγιση με το Simple Deep Q leagning, ο πράκτοράς μας μπορεί να συγκλίνει προς τη βέλτιστη λύση με αρκετά μικρά σετ εκπαίδευσης.

2.4.2 Βέλτιστη εκμετάλλευση της εισόδου

Η Mariana trench πετυχαίνει να συγκεκριμενοποιείται επί της εισόδου της, εννοώντας ότι καταφέρνει γρήγορα να μαθαίνει ποιες από τις εισόδους της έχουν όντως κάποιο αντίκτυπο στη βέλτιστη συμπεριφορά της και ποιες όχι. Στα πειράματά μας είχαμε καταστάσεις που διακρίνοντας από 15 διαφορετικές εισόδους και ο πράκτοράς μας πετύχαινε να αποφασίσει το ποιες από αυτές τις εισόδους ήταν σημαντικές και ποιες όχι στην πορεία της ζωής του ακόμα και με μικρό ποσό από βήματα εκπαίδευσης.

2.4.3 Χωρική πολυπλοκότητα

Η προσέγγιση της Mariana trench επειδή χρησιμοποιεί νευρωνικά δίκτυα μας δίνει τεράστιο πλεονέκτημα όσον αφορά τη χωρική πολυπλοκότητα, μιας και τα ΝΔ δε χρησιμοποιούν υπολογιστικό χώρο για να αποθηκεύσουν πληροφορία. Ό,τι πληροφορία χρειαζόμαστε αποθηκεύεται στα βάρη του δικτύου μας. Στη δική μας προσέγγιση, όπου καταφέραμε να πετυχαίνουμε βέλτιστα αποτελέσματα με μόλις 3-επίπεδα, ο χώρος μνήμης που χρειαζόμαστε είναι εξαιρετικά μικρός. Αυτό μας δίνει την ελαστικότητα να μπορούμε να διαχειριστούμε όσο περισσότερα δεδομένα, όσο περισσότερη εμπειρία, όσο περισσότερη πληροφορία θέλουμε, μιας και δε χρειάζεται να ανησυχούμε για τον αποθηκευτικό χώρο. Ένα από τα πλεονεκτήματα αυτής της προσέγγισης, είναι ότι μπορούμε εύκολα να κλιμακώσουμε τον πράκτοράς μας σε μεγαλύτερα περιβάλλοντα, με μεγαλύτερα σετ εμπειριών και εισόδων, χωρίς να του προκληθεί δυσκολία στο να βρει τη βέλτιστη λύση, αλλά (και εδώ είναι το κύριο πλεονέκτημά μας) να μπορεί να την εντοπίσει πιο εύκολα και πιο αποτελεσματικά τότε.

Chapter 3

Introduction

In this diploma thesis we present a Deep Reinforcement Learning (DRL) agent for managing the cluster size in cloud applications. The agent is able to find the optimal behaviour in an environment, given a reward function and a set of cluster parameters.

This agent which we call Mariana Trench [18] constitutes a successful combination of cutting edge algorithmic approaches in deep learning and resource management in cloud based environments. At any given time the optimal decision is determined in regards to the size of the cluster (increase/decrease/do nothing), in order to accumulate the best rewards over time.

Furthermore, the rewards can be adapted according to the needs of each cloud user. If the user wants to keep a low cost then the reward function gives a larger weight on keeping the cluster's size small. If the user is in a higher need of producing better throughput results in his cluster, then the agent assigns a larger weight on the throughput parameter.

3.1 Motivation

The last ten years an explosive growth of cloud computing services has taken place. The evolution of cloud technologies and also the large growth of information that needs to be stored and managed created the need of new technologies that could handle these large amounts of data. As a result, traditional SQL databases gave their place to the NOSQL [23] databases. NoSQL databases spread across many data points where they store their data, with different models than the traditional relation model that SQL databases commonly use.

NoSQL systems have been integrated onto cloud computing systems providing better handling of the information being used by cloud services. Cassandra, HBase, and MongoDB are some examples of NoSQL databases. In this diploma thesis we are using a cloud based environment built on a Cassandra database. The volume of data stored nowadays within the cloud is counted in trillions of gigabytes (or Zetabytes).

Apart from the handling of the information which has been solved by the use of NoSQL systems, large IaaS (Infrastructure as a Service) providers had another problem to face: How to manage the specific needs of a user, in order to make sure that these large resources are not being wasted? The answer to this problem was provided by **elasticity**. Elasticity is the area of study on how to scale up or down a cluster automatically based on the user requirements.

As it will be shown in the next section 3.2, there are many ways that have been used to manage elasticity on cloud computing. DRL has already been employed successfully in environments where the inputs are waveforms/images. Our goal in this thesis was to explore whether DRL algorithms can improve current state-of-the-art elasticity methods and handle other sources of information. Our results show that we achieve optimal behaviour using DRL, and also suggest that DRL can be employed in a

vast area of similar situations in science, engineering, information technology, gaming, politics, social relationships etc.

3.2 Related Work

In this section we will present related works, separated in two different subsections. The first subsection presents works in elasticity in cloud computing and the second Deep Reinforcement works.

Related works in Cloud computing Elasticity

Tiramola [13] is a cloud-enabled open-source framework used to perform auto scaling in clusters based on user's defined policies. Tiramola increases or decreases the size of a cluster by adding or removing one VM at a time in order to gain a biggest reward based on the user's needs. The factors that tiramola's decision module examines are the throughput of the cluster, the latency and the number of the VMS. Tiramola models the cluster as a Markov Decision Process where different cluster sizes represent different states. The available actions are adding or removing a VM. In order to isolate the most relevant experiences to the expected resulting state of each action, tiramola uses K-means clustering so that the expected reward is being calculated using the centroid of the cluster.

Tiramola's extended version [14] is a cloud-enabled open-source framework that is used exactly as tiramola, but differentiates on how the system is modeled and how the decision module works. In this version of tiramola decision trees [7] are used in order to perform dynamic partitioning of the state space in a Markov Decision Process. This approach proposes a full-model Markov Decision Process based algorithm that uses a Decision Tree to generalize over its input.

Nefeli [11] is a cloud-enabled infrastructure gateway that offers mechanisms to migrate virtual machines as needed, in order to adapt to the changing performance needs of each user. Each user needs to provide Nefeli with information regarding the handling of their jobs. The authors suggest that using Nefeli, they managed to get significant improvements in overall time needed and energy consumed for the workload's execution in simulated and also real cloud computing environments. The downside of Nefeli is that it needs internal information about the cloud platform infrastructure in order to work properly, whereas tiramola does not.

Related works in Deep Reinforcement Learning

A first attempt to Reinforcement Learning using Deep Neural Networks (DNN) (a.k.a. Deep Reinforcement Learning - DRL) was proposed in [21] where DNNs were used to forecast the behaviour of an environment and determine the best decision that an agent should make in a specific environment state. This approach was called "Deep Q learning", as it employed neural networks in order to simulate the Bellman equation. The agent receives image vectors as inputs in order to determine its next move.

In [6] Deep Reinforcement Learning is employed to play the famous "Go" game again using images of the "Go" board as input and employing DNNs to find the optimal policies. This DRL agent which was called AlphaGo, enjoyed a lot of publicity after beating the second best player Go player in the world with a score of 4 to 1.

In [34], DRL is used in order to create an agent that can manage itself in physics based environments. The authors created an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces. Their approach robustly solves more than 20 simulated physics tasks, including classic problems such as cartpole swing-up, dexterous manipulation, legged locomotion and car driving. This approach also uses as input images.

3.3 Thesis Structure

The structure of this thesis is as follows:

In Chapter 4 we analyse the theoretical background behind our infrastructure. We introduce the Tiramola architecture upon which we built our implementation. We also present the NoSQL database Cassandra where tiramola is integrated on, Ganglia, a metric system that we use to collect our cluster metrics. and YCSB, which is an open source framework developed by Yahoo to test clusters with a variety of different workloads that imitate a real life cluster workload.

In Chapter 5 we present basic machine learning and reinforcement learning concepts, and especially Q learning which is the reinforcement learning technique which is improved by employing Deep Neural Networks.

Neural Networks and Deep Reinforcement Learning is presented in Chapter 6, a version of which we are using in our approach, as an agent on our decision module of Marianna Trench.

In chapter 7 we present our agent Mariana Trench. We elaborate in its architecture and its network infrastructure and we present the three different algorithmic approaches that we are gonna use, Deep Q learning, Full Deep Q learning and Double Deep Q learning.

In Chapter 8, we present the results of our approaches in some demanding simulations scenarios. We first test and calibrate our network in order to get best results upon our needs. We test our agent in a simple and a more complex scenario.

In chapter 9 we test our network on a real environment of a cloud computing application, similar with the one shown in chapter 2. We collect and discuss about our experimental data. Then we talk about our approach effectiveness (spoiler alert: it was extremely effective!) and make a conclusion about our thesis. Finally we present possible future work that could be inspired by this thesis, and the assumptions that one can make for the future of machine learning in general.

Chapter 4

Necessary theoretical background

In this chapter we will discuss about the different services that we use to test our implementation on. Let us introduce you the system that is our goal to improve by using our new approach, Tiramola. We are using Tiramola with the same way we used to do but with one major difference, its decision making system. We are presenting Tiramola in Section 4.1

The cluster we use is a Cassandra cluster. We will discuss about Cassandra on Section 4.2. We test our cluster by using ycsb benchmark service, which we present in Section 4.4 and we collect metrics by using Ganglia, as shown in Section 4.3

4.1 Tiramola

The platform in which we are going to implement our work on is Tiramola[13]. Tiramola is a cloud-enabled, open-source framework for automatic resizing of NoSQL clusters. In the older versions of Tiramola, the decision for adding or removing resources was modeled as a Markov Decision Process. In his Diploma Thesis, Konstadinos Lolos [14] tested a different approach and used Adaptive Space Partitioning Markov Decision Processes. In this Thesis, by using deep neural networks, we are going to approach the problem with the decisions that Tiramola makes. But first let us introduce you Tiramola.

Tiramola offers the following features:

- A generic VM-based module which monitors cloud-based NoSQL clusters. This module, offering multi-grained, scalable monitoring, is further modified in order to report real-time, client-side statistics.
- An implementation of the decision-making module as a Markov Decision Process or RL q learning algorithms later using Adaptive State Space Partitioning of Markov Decision Processes, enabling optimal policy generation relative to both changes in the environment and different cost functions.
- A real-time system that integrates these modules; utilizing popular open-source implementations for NoSQL, Cloud APIs and benchmarking tools, our system decides on the appropriate add/remove VM action according to the chosen optimization function and relative to cluster performance.

4.1.1 Tiramola's Architecture

The architecture of Tiramola is displayed on this figure 4.1. The Decision Making module incorporates both on the user-policy defined through an optimization function and on cluster- and client- side monitored metrics and periodically decides on cluster resize actions. In order to release or acquire more

virtual machines it outputs resize actions to the Cloud Management module that interacts with the cloud vendor. At this point, the Cluster Coordinator is responsible for orchestrating the addition and removal commands relative to the particular NoSQL cluster in hand. The Monitoring module maintains up-to-date performance metrics which are collected from both client nodes and cluster nodes. Let's describe in detail each module.

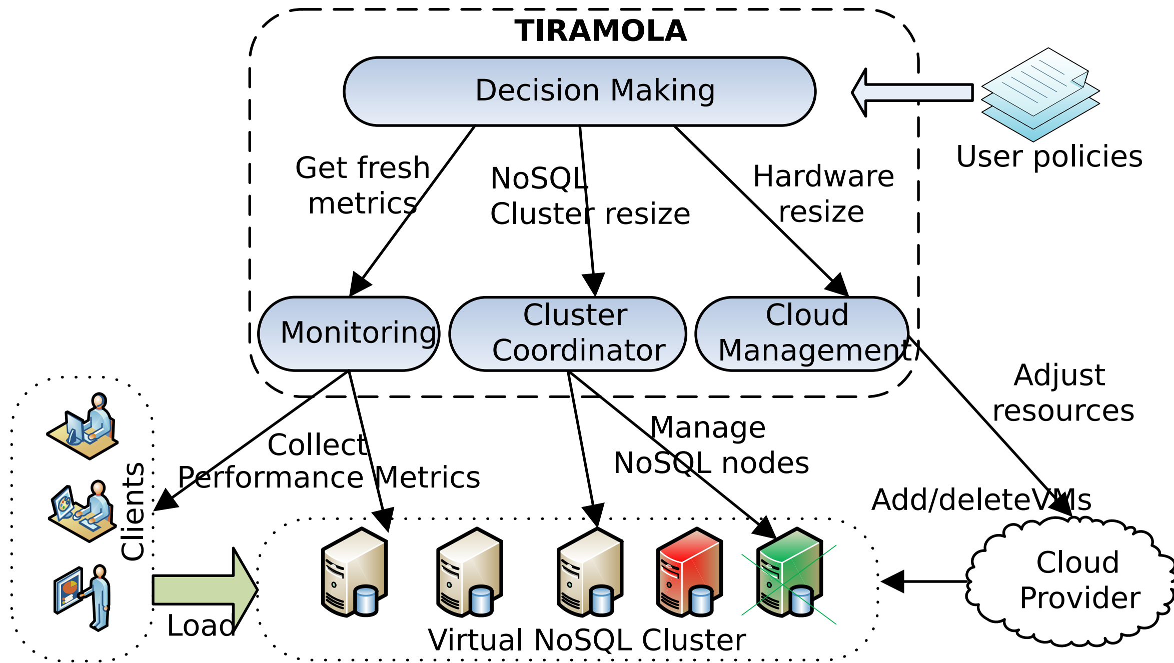

Figure 4.1: Tiramola's architecture

Decision Making Module

This module is responsible to decide the appropriate cluster resize action according to the applied load, the cluster and user-perceived performance and the optimization policy. Older versions of Tiramola formulated this particular process as a Markov Decision Process (MDP). We approach the subject by using deep neural networks (REF) as predictors that constantly identify the most beneficial action relative to the current system state. The user goals are defined through a reward function that translates the optimization on each application wishes to adhere to. When a resize module is reached, it then forwards this command to the Cloud Management module.

Monitoring

Tiramola uses Ganglia, a scalable 4.3 distributed monitoring tool which allows the remote collection of live or historical cluster statistics (such as CPU load averages, network, disk or memory space utilization, number of open client threads and many more) through its XML API. Apart from the server-side metrics, Ganglia is capable of collecting user-related metrics due to modifications that have been performed. That was necessary, because the system state may also depend on user-related information such as mean query latency. In order to achieve this, we modified our clients so that each one of them will report its own metrics by utilizing a well-known Ganglia operation called gmetric spoofing. With this mechanism, the monitoring module feeds the decision making module with an up-to-date system stake,taking into account both client and server side metrics.

Cloud management

Using the well-known kamaki, an API for communicating with Synnefo iaas, our system interacts with the cloud vendor. Our iaas service is Okeanos[24]. This module received while input commands for a NoSQL cluster resize (in the number of running VMs). The communication through kamaki's API with our cluster helps us manage that.

Cluster coordinator

The orchestration of freed or newly commissioned resources from the NoSQL cluster is being performed with a remote execution of shell scripts and the injection of automatically created NoSQL-specific configuration files to each VM. A high-level "start cluster", "add NoSQL node(s)" and "remove NoSQL node(s)" command is thus translated to a workflow of the aforementioned primitives. With the use of applicable time-outs, our implementation ensures that each step has succeeded before moving to the next one. Our framework has successfully incorporated three popular NoSQL systems that exhibit elastic behaviour: HBase (see experimental evaluation), Cassandra and Riak. With the implementation of the system's abstract primitives in the Cluster Coordinator module and with the inclusion of the system's binaries to the existing AMI virtual machine image the system is extensible enough to include more engines that support elastic operations. In the project's website the precooked virtual image is available for download. TIRAMOLA also strives to be robust: It sometimes checkpoints and it may restart after a failure; required state is maintained through the monitoring module as well as the underlying IaaS platform.

4.2 Cassandra

The powerful database system on whose shoulders we are going to build our implementation on is the Apache Cassandra, a free and open-source distributed NoSQL database. Its highly scalable and high-performance distributed database allows it to handle large amounts of data across many commodity servers providing high availability. Cassandra isn't familiar with the word failure. Assuming that the reader has basic knowledge with NoSQL databases, we present to you some features and information about Cassandra that led us to choose her among the other NoSQL databases.

Cassandra:

- is scalable, fault-tolerant, and consistent.
- is a column-oriented database.
- has a distribution design which is based on Amazon's Dynamo and its data model on Google's Bigtable.
- implements a Dynamo-style replication model with no single point of failure, but adds a more powerful "column family" data model.

We present some special features of Cassandra below

- *Fast linear-scale performance* - Cassandra is linearly scalable. It increases your throughput as you increase the number of nodes in the cluster resulting to maintain a quick response time.
- *no single point of failure* - Cassandra has no single point of failure, fact that makes it an ideal choice for business-critical applications that cannot afford a failure.
- *Fast linear-scale performance* - Cassandra is linearly scalable. It increases your throughput as you increase the number of nodes in the cluster. Therefore it maintains a quick response time.
- *Elastic scalability* - Cassandra is highly scalable; it permits us to add more hardware to accommodate more customers and data as per requirement.
- *Fast linear-scale performance* - Cassandra is linearly scalable. It increases your throughput as you increase the number of nodes in the cluster. Therefore it maintains a quick response time.
- *Easy data distribution* - Cassandra provides the flexibility to distribute data to where you need them by replicating data across multiple data centres.
- *Fast writes* - Cassandra is designed to run on cheap commodity hardware. It performs blazingly fast, writes and can store hundreds of terabytes of data without sacrificing the read efficiency.

4.2.1 Cassandra's Architecture

Cassandra's design goal is to handle big data workloads across multiple nodes without having any single point of failure. Cassandra has peer-to-peer distributed system across its nodes, and in a cluster data is distributed among all the nodes.

- In a cluster all the nodes play the same role. Each one is independent and at the same time interconnected to other nodes.
- When a node goes down, read/write requests can be served from other nodes in the network.
- Every node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster.

Data Replication

In Cassandra, for a given piece of data, some of the cluster's nodes act as replicas. If some of the nodes respond with an out-of-date value, Cassandra will then return the most recent value to the client. Cassandra's next step will then be to perform a read repair in the background to update the stale values. The image 4.2 presents a schematic view of how Cassandra uses data replication among the nodes in a cluster in a way to ensure that it does not have a single point of failure.

Cassandra's components

- *Data center* − A collection of related nodes.
- *Node* − The place where data is stored.
- *Cluster* − A cluster is a component that contains one or more data centres.
- *Commit log* − The commit log is a crash-recovery mechanism in Cassandra. Every write operation is written to the commit log.
- *Mem-table* − Mem-table is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.
- *SSTable* − A disk file to which the data is flushed from the mem-table when its contents reach a threshold value.
- *Bloom filter* − These are nothing but quick, non-deterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query.

Cluster

Cassandra is distributed through several machines that operate together. A Cluster is the outmost container of this system. In order to handle a failure every node contains a replica, and when such time comes the replica takes charge. The nodes at a cluster are arranged on a ring format and data assign to them.

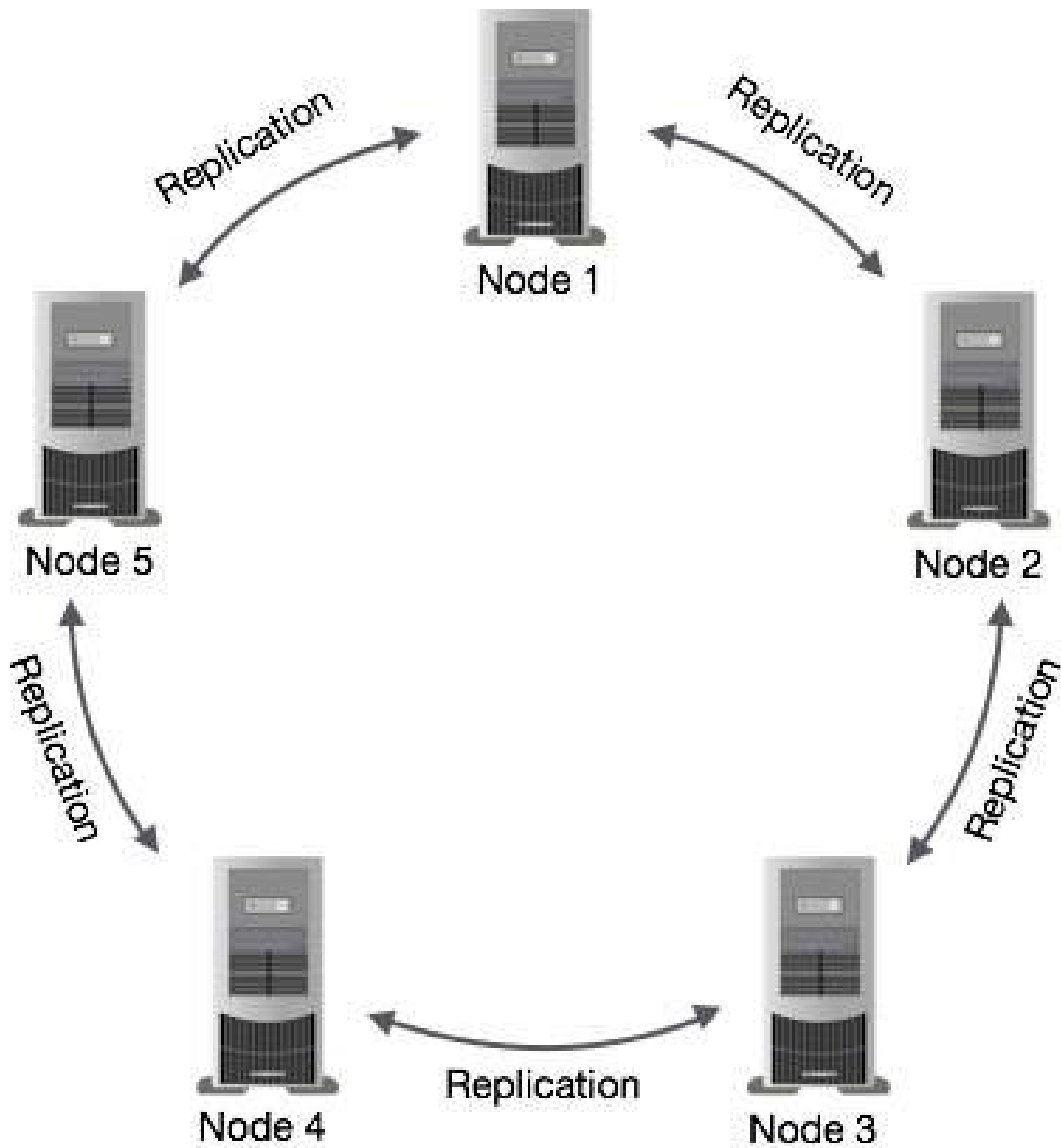

Figure 4.2: Data replication in Cassandra

Keyspace

The outermost container for data in Cassandra is a Keyspace. The basic attributes of a Keyspace are −

- *Replication factor* − It is the number of machines in the cluster that will receive copies of the same data.
- *Replica placement strategy* It is the strategy to place replicas in the ring. We have strategies such as simple strategy (rack-aware strategy), network topology strategy (datacenter-shared strategy) and old network topology strategy (rack-aware strategy.
- *Column families* − Keyspace is a container for a list of one or more column families. A column family, in turn, is a container of a collection of rows. Each row contains ordered columns. Column families represent the structure of your data. Each keyspace has at least one and often many column families. An example of Keyspace is shown here 4.3

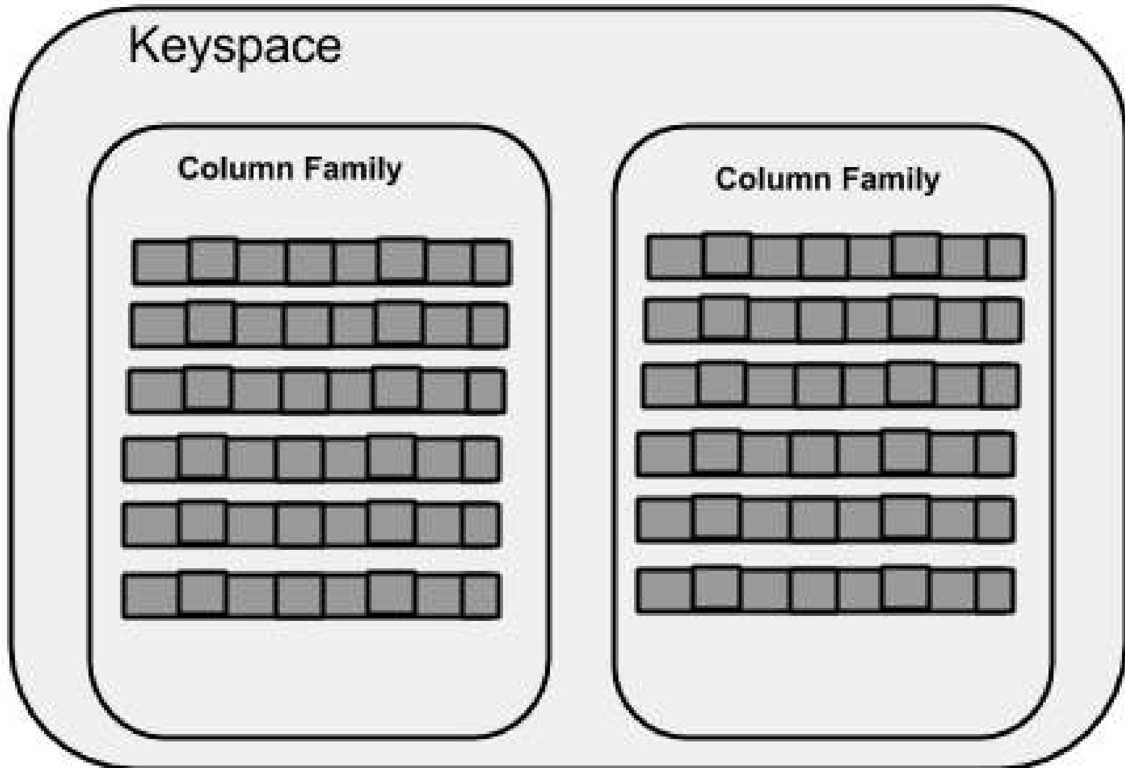

Figure 4.3: Cassanda's Keyspace

4.3 Ganglia

Ganglia [9] is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters. It leverages widely used technologies such as XML for data representation, XDR for compact, portable data transport, and RRDtool for data storage and visualization. It uses carefully engineered data structures and algorithms in order to achieve very low per-node overheads and high concurrency. The implementation is robust, has been ported to an extensive set of operating systems and processor architectures, and is currently being used on thousands of clusters around the world. It is used to link clusters across university campuses and around the world and is capable of scaling up to handle clusters with 2000 nodes. Ganglia consist of two system daemons, gmond and gmetad.

4.3.1 Ganglia Monitoring Daemon (gmond)

Gmond is a multi-threaded daemon which runs on each cluster node you want to monitor. Installation does not require having a common NFS filesystem or a database back-end, neither install special accounts or maintaining configuration files.

Gmond has four main responsibilities:

- Monitor changes in host state.
- Announce relevant changes.
- Listen to the state of all other ganglia nodes via a unicast or multicast channel.
- Answer requests for an XML description of the cluster state.

4.3.2 Ganglia Meta Daemon (gmetad)

Federation in Ganglia is achieved by using a tree of point-to-point connections amongst representative cluster nodes to aggregate the state of multiple clusters. At each node in the tree, a Ganglia Meta Daemon (gmetad) periodically polls a collection of child data sources, parses the collected XML, saves all numeric, volatile metrics to round-robin databases and exports the aggregated XML over a TCP socket to clients. Data sources may be either gmond daemons which represent specific clusters, or other gmetad daemons which represent sets of clusters. Data sources use source IP addresses for access control and can be specified using multiple IP addresses for failover. The latter capability is natural for aggregating data from clusters since each gmond daemon contains the entire state of its cluster.

4.3.3 Using Ganglia to monitor Tiramola

The Ganglia gmetad component listens on ports 8651 and 8652 by default and replies with XML metric data. Gmetad needs to be configured to allow XML replies to be sent to specific hosts or all hosts. By default only localhost is allowed. Connecting to port 8651 will get you as a response a default XML report of all metrics. Port 8652 is the interactive port which allows customized queries. Gmetad will recognize raw text queries sent to this port, i.e. not HTTP requests. This way we are receiving constantly metrics for our system, including metrics for:

- memory usage
- disk usage
- bytes coming in
- bytes coming out
- memory buffers
- shared memory
- cached memory

Cached memory, memory buffers, free memory and shared memory are combined to calculate the total memory.

4.4 Yahoo Cloud Serving Benchmark

With all these new serving databases available including Sherpa, BigTable, Azure and many more, the decision on which system is right for your application might be difficult, due to the fact that the features differ between systems, and also because there is not an easy way to compare the performance of one system versus another. The goal of the Yahoo Cloud Serving Benchmark (YCSB)[40] project is to develop a framework and common set of workloads for evaluating the performance of different ”key-value” and ”cloud” serving stores. The project comprises two areas:

The YCSB Client, an extensible workload generator The Core workloads, a set of workload scenarios to be executed by the generator Although the core workloads provide a well-rounded picture of a system’s performance, the Client is extensible so that you can define new and different workloads to examine system aspects, or application scenarios, not adequately covered by the core workload.

Similarly, the Client is extensible to support benchmarking different databases. Despite that we include sample code for benchmarking HBase and Cassandra, it is straightforward to write a new interface layer to benchmark your favorite database. A common use of the tool is to benchmark multiple systems and compare them. For example, you can install multiple systems on the same hardware configuration, and run the same workloads against each system. Then you can plot the performance of each system (for example, as latency versus throughput curves) to see when one system does better than another.

4.4.1 Using ycsb with Tiramola

We are using ycsb to load periodically our Cassandra cluster and get results as for the throughput and the latency we observe. The incoming load is determined after each step of our experiments.

Chapter 5

Machine learning- Reinforcment learning

In this chapter we will shortly discuss the history of machine learning before we meet MDPs, where we are going to elaborate. Then we are going to discuss about different approaches used to solve MDP problems, until ending up to the origins of Reinforcement Learning. The user should be aware of the fact that this little journey will help me obtain a bigger understanding of the machine learning philosophy and the foundations of the first MDP mathematical models. These models will then lightly travel us to the idea behind the Reinforcement learning and especially Q learning. We have to fully understand Q learning and the maths behind Q learning before we see the Deep Q learning algorithms, that this Diploma Thesis uses in order to construct Mariana Trench 7.

5.1 Machine learning through the years

It was in the Ancient Greek tragedies that the concept of deus ex machina emerged. Writers used this technique when they had painted themselves into a corner as a way to progress the plot. It means ”god by the machine”. The Antikythera mechanism [3] was the first analog computer to be used to comfort knowledge and data and extract finite conclusions. Aristotle was the first philosopher who tried to formalize the concept of knowledge and invented syllogistic logic, the first formal deductive reasoning system. In the newest history Leibniz tried to carry on Aristotle's vision, by creating a language that could describe and solve every problem that exists. We all know how that story went on, from 1700 to the early 90's, when Godel proved that such a language could never exist.

Back to machines that can think and act like humans, inventions like talking heads[32], Da Vinci's walking lion [38] and Pascal's calculator [25] achieved to imitate certain human's behaviour.

The first machine which was believed to have truly accomplished to ”think and act” like a human, meaning to be capable to be left alone in an environment and take decisions for its own good and benefit while it also interacts with another being in the very same environment was the Turk[36]. The Turk was able to beat some of the best chess players of each era. Nowadays the Turk is believed to have been nothing more than a scam.

In the beginning of the 19th century Andrey Markov, a Russian mathematician, in his attempt to predict whether the next letter of Alexander's Pushkin's poem, Eugene Onegin, was going to be a vowel or not, created the famous Markov chains [19], most widely known as Markov chains for discrete time problems and Markov processes for continuous time problems.

5.1.1 Markov chains and processes

Markov processes obey to the Markov property. In probability theory and statistics, the term Markov property refers to the memoryless property of a stochastic process, which is defined as

$$Pr(Xn+1 = x|X1 = x1, X2 = x2, ..., Xn = xn) = Pr(Xn+1 = x|Xn = xn) \\ \text{where} Pr(X1 = x1, ..., Xn = Xn) > 0 \tag{5.1}$$

A stochastic process has the Markov property if the conditional probability distribution of future states of the process (conditional on both past and present values) depends only upon the present state; that is, given the present, the future does not depend on the past. A process with this property is said to be Markovian or a Markov process.

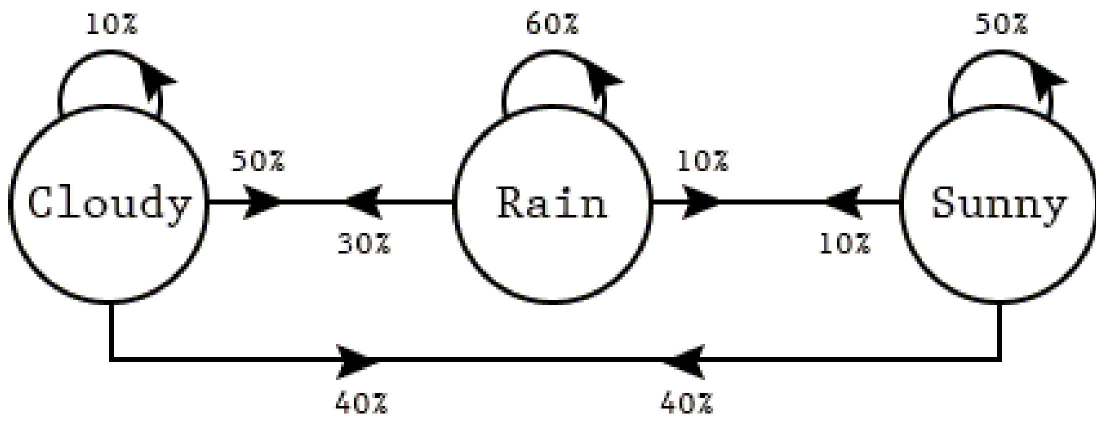

Figure 5.1: Markov Chain

Markov decision processes [22] ,in situations where outcomes are partly random and partly under the control of a decision maker,provide a mathematical framework for modeling decision making.

A Markov decision process is a 5-tuple $(S, A, P.(\cdot,\cdot), R.(\cdot,\cdot), \gamma)$, where

- S, is a finite set of states
- A, is a finite set of actions (alternatively, A is the finite set of actions available from state s),
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$, is the probability that action a in state s at time t will lead to state s' at time $t+1$
- $R_a(s, s')$, is the immediate reward (or expected immediate reward) received after transitioning from state s' to state s' ', due to action a
- $\gamma \in [0, 1]$, is the discount factor, which represents the difference in importance between future rewards and present rewards.

The goal of MDPs is to choose a policy π which will maximize some cumulative function of the random rewards, typically the expected discounted sum over a potentially infinite horizon:
$\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})$ (where we choose $a_t = (s_t) a_t = (s_t)$)

The above problem can be solved with dynamic programming with **Value Iteration** or **Policy Iteration** Older versions of Tiramola used MDPs and that's why we are going to briefly introduce to you these two basic algorithms.

Value Iteration

The first algorithm we will look at is value iteration. The essential idea behind Let's look at the value iteration of the first algorithm. The essential idea behind value iteration is: if we knew the true value of each state, our decision would be simple: always choose the action that maximizes expected utility. Only that we don't know a state's true value from the start; we only know its immediate reward. But, for example, a state might have low initial reward but a potential for a high-reward state. The true value (or utility) of a state is the immediate reward for that state, plus an expected discounted reward if the agent acted optimally from that point on.

- Algorithm
  - Start with $V_0^*(s) = 0$ for all s.
  - For $i = 1, \ldots, H$ Given $Vi*$, calculate for all states $s \in S$:
    $V_i + 1^*(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + V_i^*(s')]$
  - This is called a value update or Bellman update/back-up
- $V_i^*(s')$= the expected sum of rewards accumulated when starting from state s and acting optimally for a horizon of i steps

Theorem 1. Value Iteration Convergence

Value iteration converges. At convergence, we have found the optimal value function V for the discounted infinite horizon problem, which satisfies the Bellman equations*

$\forall S \in S : V_i + 1^*(s) \leftarrow \max_A \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_i^*(s')]$

- What should we do then when we have infinite horizon with discounted rewards:
  - Run value iteration till convergence.
  - This produces V*, which in turn tells us how to act, namely following: $^*(s) = \text{argmax}_{a \in A} \sum_{s'} T(s, a, s')[R(s, a,$
    $\gamma V^*(s')]$
- Note: the infinite horizon optimal policy is stationary, i.e., the optimal action at a state s is the same action at all times. (Efficient to store!)

Policy Iteration

Value iteration works fine apart from two weaknesses: the first one is that it can take a long time to converge in some situations, even when the underlying policy is not changing, and the second one is that it doesn't actually do what we really need. We don't really care what the value of each state is; that's just a tool to help us find the optimal policy. Why then can we not have the policy right away? We actually can, by modifying value iteration to iterate over policies. We start with a random policy, compute each state's utility given that policy, and then select a new optimal policy.

- Recall value iteration iterates: $V_i + 1^*(s) \leftarrow \max_A \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_i^*(s')]$
- Policy evaluation: $V_{i+1}\,(s) \leftarrow \sum_{s'} T(s, (s), s')[R(s, (s), s') + \gamma V(s')]$
- At convergence: $\forall s V(s) \leftarrow \sum_{s'} T(s, (s), s')[R(s, (s), s') + \gamma V(s')]$

Theorem 2. *Policy iteration is guaranteed to converge and at convergence, the current policy and its value function are the optimal policy and the optimal value function!*

5.2 Reinforecment learning

The journey to the Machine learning field continues and in 1950 Alan Turing developed the Turing test [35]. A test which was able to determine if a machine could exhibit intelligent behaviour equivalent to, or indistinguishable from, that of a human. This very same year, the Three Laws of Robotics by Isaac Asimov was also published.

- A robot may not harm a human being, or, through inaction, allow a human being to come to harm.
- A robot must obey the orders received by humans except where such orders would conflict with the First Law.
- A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

In 1951 the first Neural Network is constructed. We are going to discuss about Neural Networks, in the next chapter. In 1956 John McCarthy coined the term ”artificial intelligence” as the topic of the Dartmouth Conference, the first conference devoted to the subject. In 1989 the concept of Reinforcement Learning [27] emerges. Reinforcement learning is an area of machine learning inspired by behaviourist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward.

In machine learning, the environment is typically formulated as a Markov decision process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical techniques and reinforcement learning algorithms is that the latter do not need knowledge about the MDP and they target large MDPs where exact methods become infeasible. Reinforcement learning differs from standard supervised learning because correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected. Apart from that, there is a focus on on-line performance, which involves exploitation (of current knowledge) and finding a balance between exploration (of uncharted territory). The exploration vs. exploitation trade-off in reinforcement learning has been most thoroughly studied through the multi-armed bandit [4] problem and in finite MDPs.

RL played a major role in the bond between artificial intelligence and other engineering disciplines. Not so long ago, AI was viewed as almost entirely separate from control theory and statistics. It had to do with logic and symbols, not numbers. AI was large LISP programs, not linear algebra, differential equations or statistics. Over the last decades this view has gradually eroded. Modern AI researches accept statistical and control algorithms, for example as relevant competing methods or simply as tools of their trade. The previously ignored areas lying between AI and conventional engineering are now

among the most active, including new fields such as neural networks, intelligent control and RL. In RL we extend ideas from optimal control theory and stochastic approximation to address the broader and more ambitious goals of AI.

The top part of Figure shows that several academic disciplines that have contributed to RL.

Figure 5.2: Reinforecment learning

One can identify four main sub elements of a reinforcement learning system. A ***policy***, a ***reward function***, a ***value function*** and optionally a ***model*** of the environment.

A ***policy*** defines the way of behaving of a learning agent at a given time. Roughly speaking, policy is a mapping of perceived states of the environment to actions to be taken when in those states. It corresponds to, what psychology names, a set of stimulus-response rules or associations. In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as process. The policy is the core of an RL agent in the sense that it alone is sufficient to determine behaviour. Generally speaking, policies might be stochastic.

A ***reward function*** defines the goal in an RL learning problem. Roughly speaking, it maps each perceived state (or state-action pair) of the environment to a single number, a reward, indicating the intrinsic desirability of that state. An RL agent's sole objective is to maximize the total reward it receives in the long run. The reward function defines what are good and what are bad events for the agent. In a biological system, it would not be inappropriate to identify rewards with pleasure and pain. They are the immediate and defining features of the problem faced by the agent. As such, the reward function must necessarily be unalterable by the agent. However, it may serve as a basis fot altering the policy. For example, if an action selected by the policy is followed by low reward, then the policy may change to select some other action in that situation in the future. In general, reward functions may be stochastic.

Whereas reward function indicates what is good in an immediate sense, a ***value*** function is what specifies as good in the long run. Roughly speaking, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards determine the immediate, intrinsic desirability of environment states, values indicate the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states. For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards. The reverse could be also true. To make a parallelism, in humans rewards are like pleasure (if high) and pain (if low), whereas values correspond to a more refined and far-sighted judgement of how pleased or displeased we are that our environment is in a particular state. Expressed this way, we help it clear that value functions formalize a basic and familiar data.

Rewards are in a sense primary, whereas values, as predictions of rewards, are secondary. Without rewards there would be no values and the only purpose of estimating values would be to achieve more reward. Nevertheless, it is values that concerns as the most when making and evaluating decisions. Action choices are made based on value judgements. We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward for us over the long run. In decision-making and planning, the derived quantity called value is the one which concerns us the most. Unfortunately, it is much harder to determine values than it is to determine rewards. Rewards are basically given directly by the environment, but values must be estimated and re-estimated from the sequences of observations an agent makes over its entire lifetime. In fact, the most important component of almost all reinforcement learning algorithms is the method for estimating values efficiently. Over the last decades, the most important thing we have learned about reinforcement learning is arguably the central role of value estimation.

The fourth and last element of some reinforcement learning systems is a ***model*** of the environment. It is something that mimics the behaviour of the environment. For example, when given a state an action, the model might predict the resultant next state and rewards. Models are used for planning, and by that we mean any kind of decision on a course of action, by considering possible future situations before they actually happen. The incorporation of models and planning into reinforcement learning systems is a relatively new development. Early RL systems were explicitly trial-and-error learners. What they did was viewed as almost the opposite of planning.

5.2.1 Basic RL Algorithms

A wide variety of algorithms exist to date with which RL problems can be addressed.

At the core of most RL algorithm lies the method of Temporal Differences [31], a prediction method proved by Richard S. Sutton on converge in 1987. We consider a sequence of states followed by rewards $s_t, r_{t+1}, s_{t+1}, r_{t+2}, \ldots, r_T, s_T$.

The complete return R_t to be expected in the future from state s_t is

$$R_t = r_{t+1} + \gamma^1 r_{t+2} + \ldots + \gamma^{T-t-1} r_T,$$

where γ<1 is a discount factor (distant rewards are less important). Reinforcement learning assumes that the value of a state V(s) is directly equivalent to the expected return $V(s) = E_\pi(R_t|s_t = s)$

where π is here an unspecified action policy. Thus, the value of state s_t can be iteratively updated with

$$V(s) \rightarrow V(s_t) + [R_t - V(s_t)]$$

where α is a step-size (often =1). Note, if $V(s_t)$ correctly predicts the expected complete return R_t , the update will be zero in average and we have found the final value for V . This method requires waiting until a sequence has reached its terminal state before the value-update can commence. For long sequences this may be problematic. However, given that $E(R_t) = E(r_t + 1) + V(s_t + 1)$ we can also update iteratively by

$$V(s) \rightarrow V(s_t) + [r_{t+1} + V(s_t + 1) - V(s_t)]$$

Properties of TD-learning

: This will converge to the final value function assigning to each state its final value, if all states have been visited ”often enough”. However, this can, lead to very slow convergence if the state space is large. The expectation value of the δ-error denoted by Ξ(δ) will converge to zero, while δ itself can - for example - also alternate between positive and negative values. For large state spaces and/or sparse rewards convergence may require many steps and can be very slow.

5.3 Q Learning

Q-learning [39] is a model-free reinforcement learning algorithm. It can be viewed as a method of asynchronous dynamic programming (DP). Q-learning provides agents with the capability of learning how to act optimally in Markovian domains by experiencing the consequences of actions, while Q-learning [39] is a model-free reinforcement learning algorithm. It can be viewed as a method of asynchronous dynamic programming (DP). Q-learning provides agents with the capability of learning how to act optimally in Markovian domains by experiencing the consequences of actions, without requiring them to build maps of the domains. Learning proceeds similarly to the method of temporal differences which we discussed above. An agent tries an action at a particular state, and evaluates its consequences in terms of the immediate reward or penalty it receives and its estimate of the value of the state to which it is taken. By trying all actions in all states repeatedly, it learns which is best overall, judged by long-term discounted reward.

5.3.1 Q learning algorithm

To begin with, let’s introduce two very important variables of the q learning approach.

Learning rate

α is the learning rate of the algorithm. It determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent learn nothing, while a factor of 1 would make the agent consider only the most recent information. In fully deterministic environments, a learning rate of $\alpha_t = 1$ is optimal. The algorithm, when the problem is stochastic, still converges under some technical conditions on the learning rate that require it to decrease to zero. In practice, often a constant learning rate is used, such as $\alpha_t = 0.1 forallt$

Discount factor

γ is the discount factor. It determines the importance of future rewards. A factor of 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward. If the discount factor meets or exceeds 1, the action values may diverge. For $\gamma = 1$, without a terminal state or, if the agent never reaches one, all environment histories will be infinitely long, and utilities with additive, undiscounted rewards will generally be infinite. As we know from previous chapters there is at least one optimal stationary policy π which is such that

$$V^*(x) = V^{\pi *}(x) = max_a(R_x(a) + \gamma \sum_y P_{xy}[\alpha]V^{\pi *}(y))$$

is as well as an agent can do from state x. Although this might look circular, it is actually well defined and we are provided with a number of methods for calculating V^* and one π^*, assuming that $R_x(a)$ and $P_{xy}[a]$ are known. The task facing a Q learner is that of determining a π^* without initially knowing these values. For a policy π^* , define Q values (or action-values) as:
$Q^{\pi}(x, a) = R_x(\alpha) + \gamma \sum_y P_{xy}[\pi(x)]V^{\pi}(y)$ In other words, the Q value is the expected discounted reward for executing action α at state x and following policy π thereafter. The object in Q learning is to estimate the Q values for an optimal policy. It is straightforward to show that $V*(x) = \max_a Q^*(x, a)$ and that if a* is an action at which the maximum is attained, then an optimal policy can be formed as $\pi^*(x) = \alpha^*$. Herein lies the utility of the Q values: if an agent can learn them, it can easily decide what it is optimal to do. Although there may be more than one optimal policy or a*, the Q* values are unique. In Q-learning, the agent's experience consists of a sequence of distinct stages or episodes. In the nth episode, the agent:

- observes its current state x_n,
- selects and performs an action a_n,
- observes the subsequent state Y_n,
- receives an immediate payoff r_n and
- adjusts its Q_{n-1} values using a learning factor a_n, according to:

$$Q_n(x, a) = \begin{cases} 1, & 1 - a_n Q_{n-1}(x, a) + \alpha_n[r_n + \gamma V_{n-1}(y_n) \; if x = x_n and \alpha = \alpha_n and \\ 0, & Q_{n-1}(x, a) otherwise \end{cases} \quad (5.2)$$

where $V_{n-1}(y) == \max_b(Q_{n-1}(y, b))$

is the best the agent thinks it can do from state y. Of course, in the early stages of learning, the Q values may not accurately reflect the policy they implicitly define (the maximizing actions in the second equation). The initial Q values, $Q_0(X, a)$, for all states and actions are assumed given.

This description assumes a look-up table representation for the $Q_n(x, a)$. Watkins (1989) shows that Q-learning may not converge correctly for other representations.

Chapter 6

Neural Networks, Deep Q learning

In this chapter we are going to discuss about Neural Networks, an approach on machine learning problems. Firstly let's talk about linear regression [16]. Let's have a look at the Figure 6.1 Take some points on a 2D graph, and draw a line that fits them as well as possible. What you have just done is generalized from a few examples of pairs of input values (x) and output values (y) to a general function that can map any input value to an output value. This is known as linear regression, and it is a wonderful, little 200 years old technique, for extrapolating a general function from some set of input-output pairs. And here is why having such a technique is wonderful: there is an incalculable number of functions that are hard to develop equations for directly, but are easy to collect examples of input and output pairs for in the real world - for instance, the function mapping an input of recorded audio of a spoken word to an output of what that spoken word is. Linear regression is a technique to solve the problem of speech recognition, though a bit too wimpy, but what makes it essential is what supervised Machine Learning is all about: given a training set of examples, where each example is a pair of an input and output from the function (we shall touch on the unsupervised flavour in a little while), the machine can 'learn' a function. To be more specific, machine learning methods should produce a function that can generalize well to inputs not in the training set, since then we can actually apply it to inputs for which we do not have an output. For example, the speech recognition technology that is currently used by Google is powered by Machine Learning with a massive training set, but not nearly as big a training set as all the possible speech inputs you might task your phone with understanding.

This generalization principle is so important that there is almost always a test set of data (more examples of inputs and outputs) that is not part of the training set. The separate set can be used to evaluate the effectiveness of the machine learning technique by seeing how many of the examples the method correctly computes outputs for given the inputs. The nemesis of generalization is overfitting - learning a function that works really well for the training set but badly on the test set. Since machine learning researchers had the need to state means to compare the effectiveness of their methods, over time there appeared standard datasets of training and testing sets that could be used to evaluate machine learning algorithms.

6.1 Neural Networks origins

6.1.1 The Perceptron

Linear regression resembles the first idea conceived specifically as a method to make machines learn: Frank Rosenblatt's Perceptron Figure 6.2 Rosenblat, a psychologist, conceived of the Perceptron as a simplified mathematical model of how the neurons in our brains work: it takes a set of binary inputs (nearby neurons), multiplies each input by a continuous valued weight (the synapse strength to each nearby neuron), and thresholds the sum of these weighted inputs to output a 1 if the sum is big enough and otherwise a 0 (in the same way neurons either fire or do not). Most of the inputs that a Perceptron

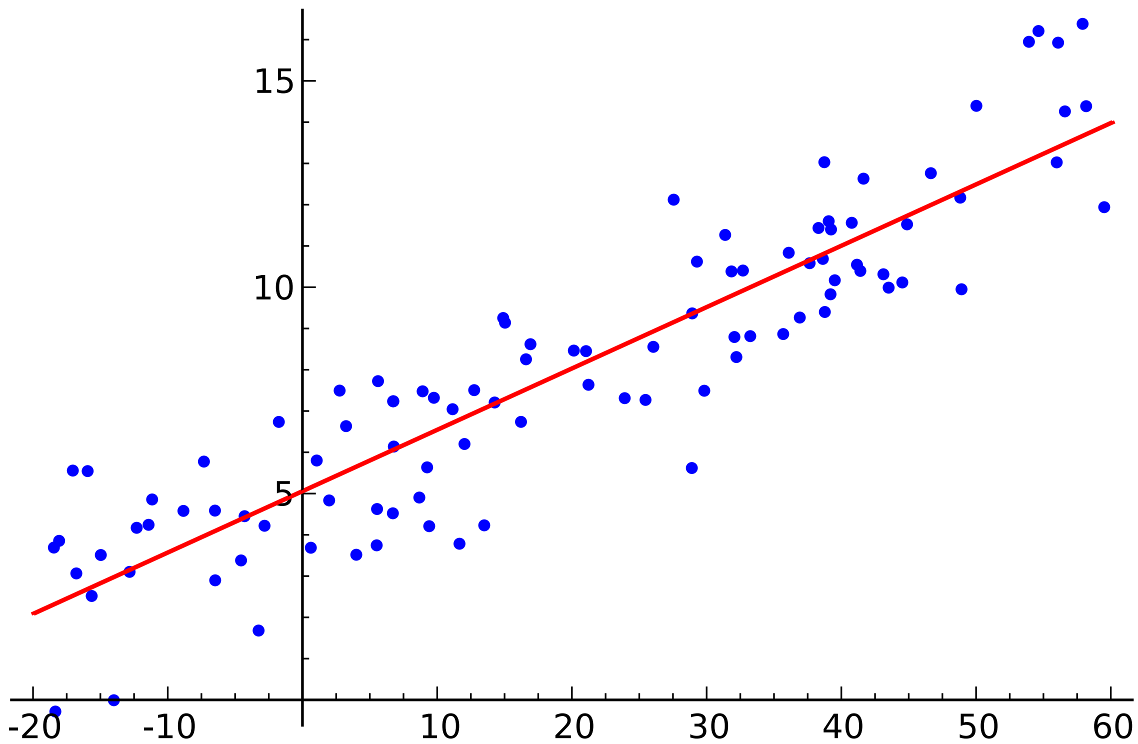

Figure 6.1: Linear Regression

takes can be either the output of another Perceptron or some data, but an extra detail is that Perceptrons also have one special 'bias' input, which just has a value of 1 and basically ensures that more functions are computable with the same input by being able to offset the summed value. This model of the neuron built by Warren McCulloch and Walter Pitts Mcculoch-Pitts[20] , who showed that a neuron model that sums binary inputs and outputs a 1 if the sum exceeds a certain threshold value, and otherwise outputs a 0, can model the basic OR/AND/NOT functions. In the first days of AI, this was a significantly important issue - the predominant thought at the time was that making computers able to perform formal logical reasoning would essentially solve AI. However, the Mcculoch-Pitts

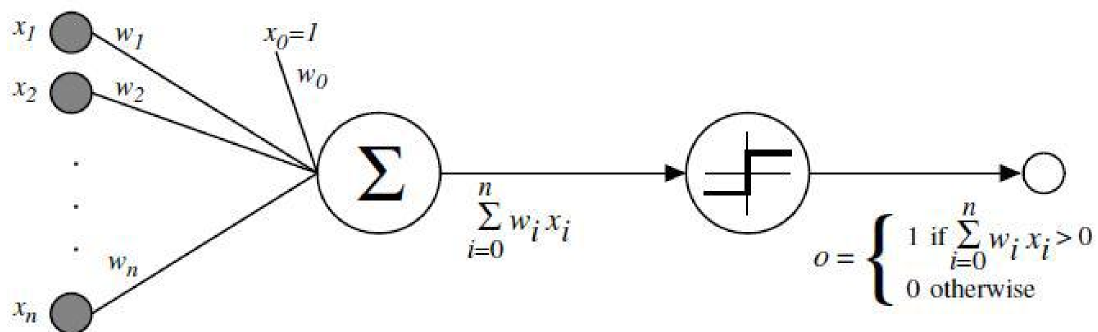

Figure 6.2: Perceptron

model lacked the crucial mechanism for learning, which was highly needed in order to be usable for AI. This is where the Perceptron excelled - Rosenblatt came up with a way to make such artificial neurons learn, finding inspiration from the foundational work of Donald Hebb. Hebb put forth the unexpected and hugely influential idea that knowledge and learning occurs in the brain primarily through the formation and change of synapses between neurons - concisely stated as Hebb's Rule:

> “When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing B, is increased.”

This idea was not exactly the one that the Perceptron followed, but having weights on the inputs, allowed for a very simple and intuitive learning scheme: given a training set of input-output examples the Perceptron should ‘learn’ a function from, for each example increase the weights if the Perceptron output for that example’s input is too low compared to the example, and otherwise decrease the weights if the output is too high. To state the algorithm ever so slightly more formally, the basic steps are given below:

1. accommodate the growth of load in a manner that does not impact the quality of the service and

2. utilize the addition of new resources to their full extend, in order to improve its performance.

There are two methods of scaling, horizontal (scaling out) and vertical (scaling up), which are explained below:

1. Start off with a Perceptron having random weights and a training set

2. For the inputs of an example in the training set, compute the Perceptron’s output

3.
    - If the output should have been 0 but was 1, decrease the weights that had an input of 1.
    - If the output should have been 1 but was 0, increase the weights that had an input of 1.

4. Go to the next example in the training set and repeat steps 2-4 until the Perceptron makes no more mistakes

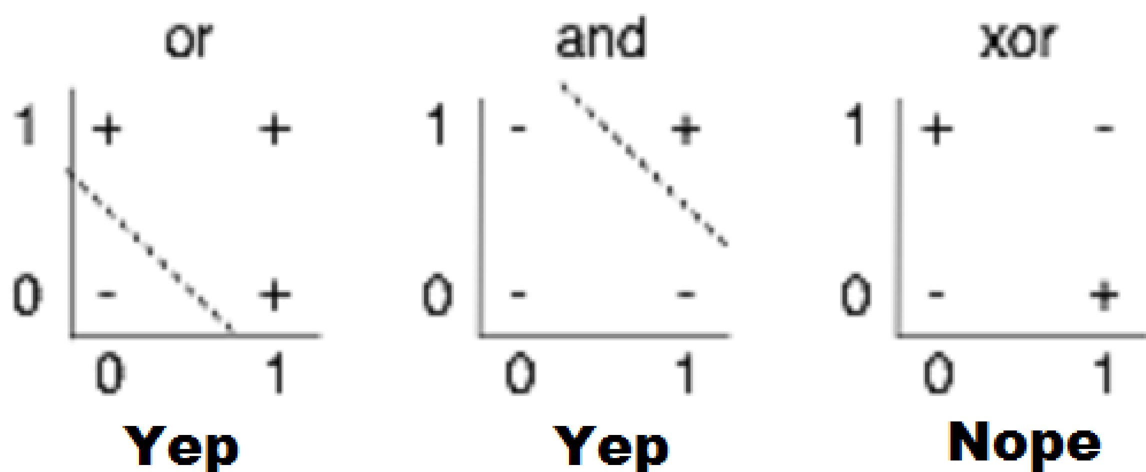

Figure 6.3: limitations of Perceptrons. Finding a linear function on the inputs X,Y to correctly output + or - is equivalent to drawing a line on this 2D graph separating all + cases from - cases; clearly, for the third case this is impossible

6.1.2 The discovery of hidden layers

The single layer neural networks were quickly seemed to not be effective enough. But what was the reason? The idea, after all, was to combine a bunch of simple mathematical neurons to do complicated things, not to use a single one. In other words, we don't only use one output layer to send an input to arbitrarily many neurons, which are called a hidden layer 6.4 because their output acts as input to another hidden layer or the output layer of neurons. All the intermediate computations done by the hidden layer(s) can confront vastly more complicated problems than just a single layer, but only the output layer's output is 'seen', which is the answer of the neural net. In basic terms, hidden layers can

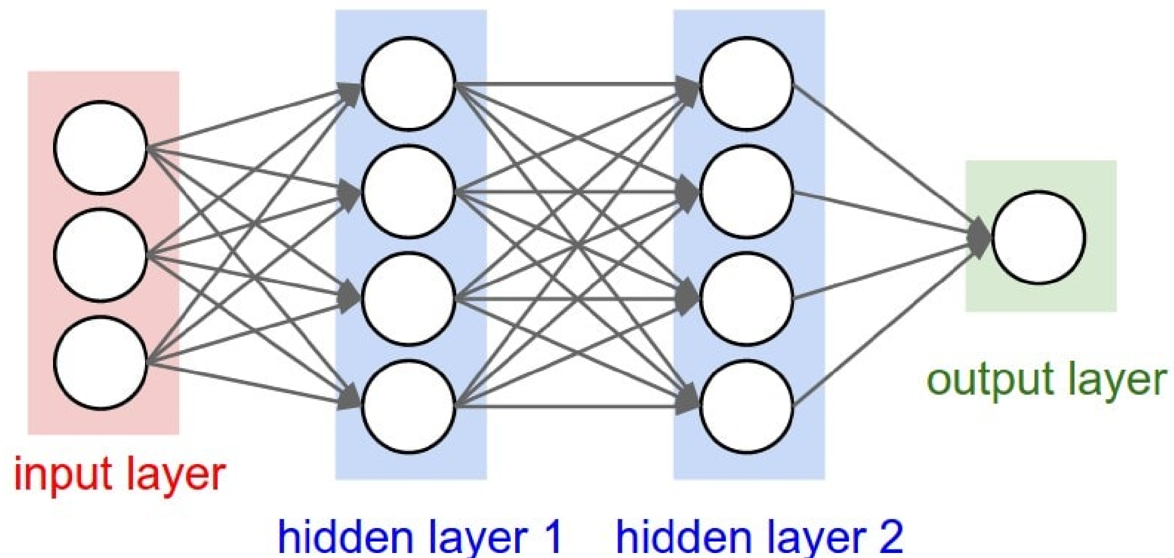

Figure 6.4: Neural net with two hidden layers

find features within the data and allow following layers to operate on those features rather than the noisy and large raw data, which is very beneficial. For example, in the very common neural net task of finding human faces in an image, the first hidden layer could take in the raw pixel values and find lines, circles, ovals, and so on within the image. The next layer would receive the position of these lines, circles, ovals, and so on within the image and use those to find the location of human faces - much easier! And people, basically, understood this. In fact, until recently, machine learning techniques were commonly not applied directly to raw data inputs such as images or audio. Instead, machine learning was done on data after it had passed through feature extraction - that is, to make machine learning, learning easier was done on preprocessed data from which had been already extracted more useful features such as angles or shapes. It is highly important to mark that Minsky and Papert's analysis of Perceptrons did not merely show the impossibility of computing XOR with a single Perceptron, but specifically argued that it had to be done with multiple layers of Perceptrons - what we now call multilayer neural nets - and that Rosenblatt's learning algorithm did not work for multiple layers. And that was the real issue: the simple learning rule, that was previously outlined for the Perceptron, is not functional for multiple layers. In order see the reason behind, let's reiterate how a single layer of Perceptrons would learn to compute some function:

1. A number of Perceptrons equal to the number of the function's outputs would be started off with small initial weights
2. For the inputs of an example in the training set, compute the Perceptrons' output
3. For each Perceptron, if the output does not match the example's output, adjust the weights accordingly
4. Go to the next example in the training set and repeat steps 2-4 until the Perceptrons no longer make mistakes

The fact why this does not operate for multiple layers should be clear: the example only specifies the correct output for the final output layer, so how in the world should we know how to adjust the weights of Perceptrons in layers before that? The answer, despite taking some time to derive, proved to be once again based on age-old calculus: the chain rule. The key realization was that if the neural net neurons were not quite Perceptrons, but were made to compute the output with an activation function that was still non-linear but also differentiable, as with Adaline, the chain rule could be used to compute the derivative for all the neurons in a prior layer and thus the way to adjust their weights would also be known and also the derivative could be used to adjust the weight to minimize error. To make it more clear: we can use calculus to assign some of the blame for any training set mistakes in the output layer to each neuron in the previous hidden layer, and then we can further split up blame if there is another hidden layer, and so on - we backpropagate the error. And so, we can find how much the error changes if we change any weight in the neural net, including those in the hidden layers, and use an optimization technique (for a long time, typically stochastic gradient descent) to find the optimal weights to minimize the error.

6.2 The backpropagation breakthrough

In 1985, David Rumelhart, Geoff Hinton and Ronald J. Williams [30], described the process of backpropagation, as a neural networks technique. Backpropagation was well known form the 60s but it wasn't till 1985 that Rumelhart, Hinton and Williams used the basic idea of backpropagation and introduced to the world a new method of solving neural networks with hidden layers. To talk about backpropagation we shall be first be familiar with the concept of gradient descent.

6.2.1 Gradient Descent

Gradient descent [5] is used for finding the minimum of a function and it is a first-order iterative optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point. On the other hand, gradient ascent is the procedure in which one takes steps proportional to the positive of the gradient and one approaches a local maximum of that function, which distinguishes it from the gradient descent. Gradient descent is based on the observation that if the multi-variable function $\mathbf{F}(\mathbf{x})$ is defined and differentiable in a neighbourhood of a point **a** , then $\mathbf{F}(\mathbf{x})$ decreases fastest if one goes from **a** in the direction of the negative gradient of **F** at $\boldsymbol{a}, -\boldsymbol{\nabla F(a)}$. It follows that, if $\mathbf{a^{n+1}} = \mathbf{a^n} - \gamma\nabla\mathbf{F}(\mathbf{a^n})$ for γ small enough, then $\mathbf{F}(\mathbf{a^n}) \geq \mathbf{F}(\mathbf{a^{n+1}})$. In other words, the term $\gamma\nabla\mathbf{F}(\mathbf{a})$ is subtracted from **a** because we want to move against the gradient, namely down toward the minimum. With this observation in mind, one starts with a guess $\mathbf{x_0}$ for a local minimum of F, and considers the sequence $\mathbf{x_0}, \mathbf{x_1}, \mathbf{x_2}, \ldots$ such that $\mathbf{x_{n+1}} = \mathbf{x_n} - \gamma_\mathbf{n}\nabla\mathbf{F}(\mathbf{x_n}), \mathbf{n} \geq 0$ We have $\mathbf{F}(\mathbf{x_0}) \geq \mathbf{F}(\mathbf{x_1}) \geq \mathbf{F}(\mathbf{x_2}) \geq \cdots$, so hopefully the sequence $\mathbf{x_n}$ converges to the desired local minimum. Note that the value of the step size γ is allowed to change at every iteration. So what gradient descent

does is moving us to minimums of our functions. That is a powerful tool, that tells us that we can re-evaluate our function variables, using gradient descent which takes us to a family of our variables space where we can minimise our function. The way to use gradient descent on neural networks is backpropagation. So what gradient descent does is moving us to minimums of our functions. That is a powerful tool which tells us that we can re-evaluate our function variables using gradient descent, fact which leads us to a family of our variables space where we can minimize our function. The way to use gradient descent on neural networks is backpropagation.

6.2.2 backpropagation

Here is an introductive example about backpropagation. We are about to use a neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias. Here's the basic structure: We put some random initial weights and biases. The goal

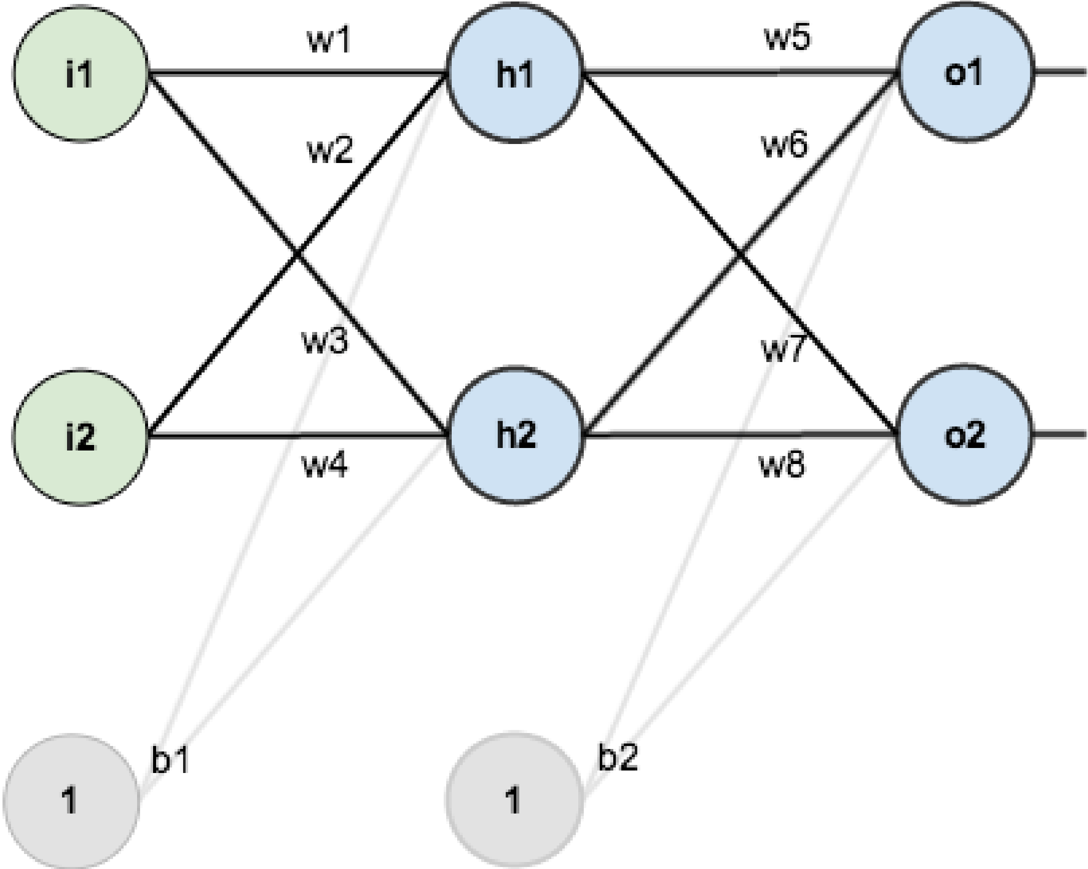

Figure 6.5

of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.

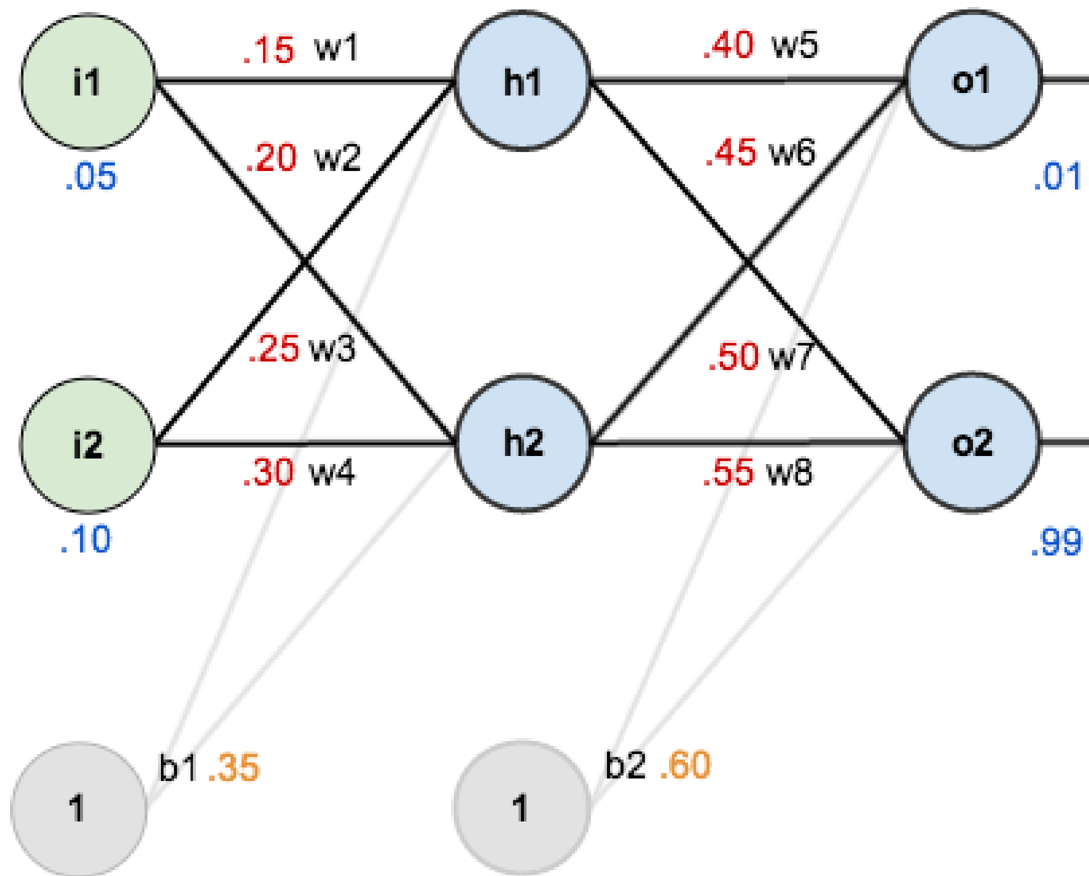

Figure 6.6

The Forward Pass

To begin with, let's see what the neural network currently predicts, given the weights and biases above and inputs of 0.05 and 0.10. To do this we'll feed those inputs forward though the network. We figure out the total net input to each hidden layer neuron, squash the total net input using an activation function (here we use the logistic function), then repeat the process with the output layer neurons. Here's how we calculate the total net input for h_1: We then squash it using the logistic function to get the output of h_1: $out_{h1} = \frac{1}{1+e_{-net_{h1}}} = \frac{1}{1+e_{-0.3775}}$ Carrying out the same process for h_2 we get: $out_{h2} = 0.59688$ We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs. Here's the output for o_1: $net_{o1} = w_5 * out_h 1 + w_6 * out_h 2 + b_2 * 1$ $net_{o1} = 0.4*0.59326+0.45*0.59688+0.6*1 = 1.10590$ $out_{o1} = \frac{1}{1+e_{-net_{o1}}} = \frac{1}{1+e_{-1.10590}} =$ 075136

Calculating the Total Error

We are now able to calculate the error for each output neuron using the squared error function and sum them up to get the total error: $E_{total} = \sum \frac{1}{2}(target - output)^2$ The $\frac{1}{2}$ is included so that exponent is cancelled when we differentiate later on. The result is eventually multiplied by a learning rate anyway

so it doesn't matter that we introduce a constant here. For example, the target output for o_1 is 0.01 but the neural network output 0.75136507, therefore its error is: $E_{o1} = \sum \frac{1}{2}(target_{o1} - output_{o1})^2 = \sum \frac{1}{2}(0.01 - 0.75135)^2 = 0.27841$ and $E_{o2} = 0.02356$ The total error for the neural network is the sum of these errors: $E_{total} = 0.29837$

The Backwards Pass

Consider w_5. We want to know how much a change in w_5 affects the total error, aka $\frac{\partial E_{total}}{\partial w_5}$. $\frac{\partial E_{total}}{\partial w_5}$ is is read as "the partial derivative of E_{total} with respect to w_5". You can also say "the gradient with respect to w_5". By applying the chain rule we know that: $\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$ Visually, here's what we're doing: We need to figure out each piece in this equation. First, how much

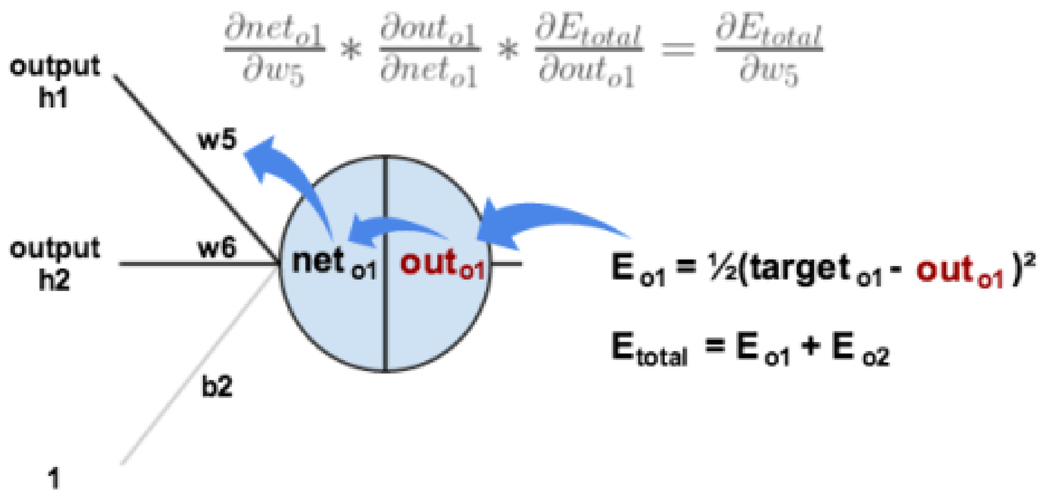

Figure 6.7

does the total error change with respect to the output? $E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$ $\frac{\partial E_{total}}{\partial out_{o1}} = 2\frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$ $\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = 0.74136$ When we take the partial derivative of the total error with respect to out_{o1}, the quantity $\frac{1}{2}(target_{o2} - out_{o2})^2$ becomes zero because out_{o1} does not affect it which means we're taking the derivative of a constant which is zero. Next, how much does the output of o_1 change with respect to its total net input? The partial derivative of the logistic function is the output multiplied by 1 minus the output: $out_{o1} = \frac{1}{1 + e_{-net_{o1}}}$ $\frac{out_{o1}}{net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75316 * (1 - 0.75316) = 0.18618$ Finally, how much does the total net input of o_1 change with respect to w_5 ? $net_{o1} = w_5 * out_h 1 + w_6 * out_h 2 + b_2 * 1$ $\frac{out_{o1}}{w_5} = 1 * out_{h1} * w_5^(1 - 1) + 0 + 0 = out_{h1} = 0.59326$ Putting it all together: $\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$ $\frac{\partial E_{total}}{\partial w_5} = 0.74136 * 0.18681 * 0.59326 = 0.08216$ To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5): $w_5^(+) = w_5 * \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.08126 = 0.35891$

6.3 Deep Q learning

Deep Learning

We have come to an understanding of what deep neural networks are all about when we referred to backpropagation on the previous chapter. But before we proceed on deep reinforcement learning let us define it accurately The application to learning tasks of artificial neural networks (ANNs) that contain more than one hidden layer is called deep learning (also known as deep structured learning or hierarchical learning). Deep learning is part of a broader family of machine learning methods based on learning data representations, as opposed to task specific algorithms. Learning can be supervised, partially supervised or unsupervised. Deep learning is a class of machine learning algorithms that

- use a cascade of many layers of non-linear processing units for feature extraction and transformation. Each successive layer uses the output from the previous layer as input. The algorithms may be supervised or unsupervised and applications include pattern analysis (unsupervised) and classification (supervised).
- are based on the (unsupervised) learning of multiple levels of features or representations of the data. Higher level features are derived from lower level features to form a hierarchical representation.
- are part of the broader machine learning field of learning representations of data.
- learn multiple levels of representations that correspond to different levels of abstraction; the levels form a hierarchy of concepts.

Classification-Reinforcement learning

Before we talk about Deep Q Learning, let us first understand Classification as an aspect of problems solved with neural networks. Classification problems are a subcategory of supervised learning algorithms. In classification problems we have a set of data. As we previously saw we divide our data to a training and a test set. When we train our Neural Network we let it predict a value for a given input. The difference between the real value, which we call target and the predicted value is the error that we want to minimize. In classification problems the target is district. Meaning yes or no answers are the right prediction. For example, ”does this image depicts a cat?”. But to train our algorithm we must first know the right answer for the target. In reinforcement learning, as we saw on chapter 5, we don't know the right answer, the target from the beginning. What we do is letting the agent find the best practice after feeding him with rewards. So how could the above two strategies combine?

6.3.1 DeepMind's Deep RL

In 2013, Deepmind publicized a paper [21] that was about to change the course of learning evolution. The paper's name was Playing Atari with Deep Reinforcement Learning and that was what they basically did. They found a way of combining deep neural networks techniques for classification problems, with reinforcement learning, especially Q Learning. In this paper they demonstrated how a computer learned to play Atari 2600 video games by observing just the screen pixels and receiving a reward when the game score increased. The result was remarkable, because the games and the goals in every game were very different and designed to be challenging for humans. The same model architecture, without any change, was used to learn seven different games, and in three of them the algorithm performed even better than a human! Deepmind's research took place in order to diminish

the problems that occurred when trying to combine RL algorithms with Deep learning. Reinforcement learning presents several challenges from a deep learning perspective. Firstly, most successful deep learning applications to date required large amounts of hand labelled training data. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting rewards, which can be thousands of timesteps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning. Another issue is that most deep learning algorithms assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviours, which can be problematic for deep learning methods that assume a fixed underlying distribution. Deepmind's goal was to create a neural network agent that could learn to play as many games as possible. Atari 2600 games implemented in The Arcade Learning Environment (example in Figure 6.8 . Atari 2600 is a challenging RL testbed that presents agents with a high dimensional visual input (210 × 160 RGB video at 60Hz) and a diverse and interesting set of tasks that were designed to be difficult for humans players.

Background

How did Deepmind managed set up its experiments? First of all they had to find a depiction of the environment, so they used the raw pixels of the Atari screen. So first we have a $x_t =$ vector containing all screen pixels. At each state s_t our agent performs an action a_t with $a_t \in A = 1, .., K$ the legal set of actions. When performing an action a_t from state s_t the agent receives a reward r_t. As it is pretty obvious it is impossible to understand the current situation from only the current screen state x_t. Therefore they considered a sequence of actions and observations $s_t = x_1, a_1, x_2, a_2, ..., x_T$. All sequences on the emulator are assumed to terminate in a finite number of time steps. This formalism gives rise to a large but finite Markov decision process (MDP) in which each sequence is a distinct state. As a result, we can apply standard reinforcement learning methods for MDPs, simply by using the complete sequence s_t as the state representation at time t. They make the standard assumption that future rewards are discounted by a factor of γ per time-step. The future discounted return at time t is defined as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r'_t$. T is the time-step at which the game terminates. The optimal value $Q*(s, a)$ classically is the maximum expected value after executing an action a when in a state s. $Q*(s, a) = max_\pi E[R_t | s_t = s, a_t = a, \pi]$. The optimal action-value function obeys an important identity known as the Bellman equation which we discussed in chapter 5.1.1. $Q*(s, a) = E[r + \gamma \max_{a}^{'} Q*(s', a'|s, a)]$ The basic idea behind many reinforcement learning algorithms is to estimate the action value function, by using the Bellman equation as an iterative update. In practice, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalisation. Instead, it is common to use a function approximator to estimate the action-value function, $Q(s, a,) \approx Q*(s, a)$ (s, a). In the reinforcement learning community this is typically a linear function approximator, but sometimes a non-linear function approximator is used instead, such as a neural network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i, $L_i(\theta_i) = E[(y_i - Q(s, a, \theta_i))^2]$

where, **and this is the most important part**, $y_i = E[r + \gamma max_a' Q(s', a', \theta_{i-1} | s, a]$. So our targets are determined by the Bellman's equation. In classification problems, we had fixed targets. Now we obtain our targets by simply performing the Bellman's equation on our network. We use our networks weights to compute the $Q*$ in one time-step on the future, so we can go then back and using backpropagation, compute our new network weights. This is the hole key. Where we were trying to get from the beginning of this paper, so that we then be able to understand the thinking behind this diploma thesis.

Figure 6.8: Atari game example

Note that this algorithm is model-free: it solves the reinforcement learning task directly using samples from the emulator E, without explicitly constructing an estimate of E. It is also off-policy: it learns about the greedy strategy $a = max_a Q(s, a;)$, while following a behaviour distribution that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an e-greedy strategy that follows the greedy strategy with probability $1 - $ e and selects a random action with probability e.

The trick of experience replay

So Deepmind wanted to connect a RL algorithm with a deep neural network operating on RGB images and being able to efficiently been trained by performing stochastic gradient descent updates. Tesauro's TD-Gammon [33] architecture provided a starting point for such an approach. Teasuro's architecture updates the parameters of the network which estimates the value function from on-policy samples of experience $s_t, a_t, r_t, s_{t+1}, a_{t+1}$ drawn from the algorithm's interactions with the environment (or by self-play, in the case of backgammon). This approach was able to outperform the best human backgammon players 20 years ago! So it is natural to wonder whether two decades of hardware improvements, coupled with modern deep neural network architectures and scalable RL algorithms might produce significant progress. In contrast to TD-Gammon and similar online approaches, Deepmind utilized a technique known as **experience replay** where they: store the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data-set $D = e_1, ..., e_N$, pooled over many episodes into a **replay memory**. During the inner loop of the algorithm, they apply Q-learning updates, or mini batch updates, to samples of experience, $e \in D$, drawn at random from the pool of stored samples. After performing experience replay, the agent selects and executes an action according to an e-greedy policy. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically. By using experience replay the behaviour distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. Since using histories of arbitrary length as inputs to a neural network can be difficult, Deepmind's Q-function instead works on fixed length representation of histories produced by a function . The full algorithm, called as deep Q-learning, is presented below.

The Algorithm

Initialize replay memory D to capacity N Initialize action-value function Q with random weights
for episode = 1, M **do**
  Initialise sequence $s_1 = x_1$ and preprocessed sequenced $_1 = (s_1)$
  for t=1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \max_a Q * ((s_t), a; \theta)$
    Execute action a_t and observe reward r_t and image $x_(t+1)$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $_{t+1} = (s_{t+1})$
    Store transition $(_t, a_t, r_t, _(t+1))$ in D
    Sample minibatch of transitions $(_j, a_j, r_j, _(j+1))$ from D

$$Sety_j = \begin{cases} r_j & forterminal_{j+1} \\ r_j + \gamma max'_a Q(_{j+1}, a'; \theta) & fornonterminal_{j+1} \end{cases} \quad (6.1)$$

    perform a gradient descent step on $(y_j - Q(_j, a_j; \theta))^2$
  end
end

Algorithm 1: Deep Q learning

Chapter 7

Mariana Trench

In the previous three chapters we discussed and explained all the things we must know. At this point, we are going to demonstrate our approach. We are using Deepmind's algorithm. But deep RL is applied on neural networks that are used for computer vision. Deep RL takes as input image vectors. So how is it possible for us to use this algorithm for a problem as specified in section 4.1; It is needless to say that we are going to use Tiramola, after applying changes on its decision module. Apart from that it is worth mentioning that in Mariana Trench we added control at the unite of cluster coordinator. From here and now, Mariana Trenchamola will never fail on adding or removing VMs, because it makes sure that it has accomplished its orders before proceeding to a next action. Thus the number of VMs after taking an action does not depend on a probability function.

Let's return to Mariana Trenchamola. The idea is that if Deepmind's RL can find patterns behind its inputs, after being provided with a reward function, then it does not matter if the input is an image vector or in our case a metrics vector. Of course, due to the fact that we are not playing with image vectors, we don't use convolution layers but fully connected layers as. We apply different processing on the input data, from what we would do if we have image vectors. We use deep RL for **hree main reasons**.

Experience replay

Using experience replay, we can be assure that our agent will not fall into local minimums and that it will eventually find the optimal policy given the reward function. Therefore we have a system that never fails. We break correlation between data, we are able to learn from all past policies and we are basically using of-policy Q learning. As we previously discussed, the basic idea is that by storing an agent's experiences and then randomly drawing batches of them to train the network, we can learn more robustly to perform well in the task. By keeping random the experiences we draw, we prevent the network from learning only what it is immediately doing in the environment, and allow it to learn from a more varied array of past experiences. Each of these experiences are stored as a tuple of <state,action,reward,next state>. The Experience Replay buffer stores a fixed number of recent memories, and as new ones come in, old ones are removed. When the time comes to train, we simply draw a uniform batch of random memories from the buffer, and train our network with them.

Clustering

Deep RL allows the agent to find out on its own which input data play an important role on the behaviour of the agent and which are not. Given these circumstances, we do not need to cluster our input. We just feed the agent with any information we have and let it decide which input are important and which one are insignificant for its future behaviour. That approach was first used in Tiramola, in this paper [17]. Using decision trees and Adaptive State Space Partitioning the authors manage to

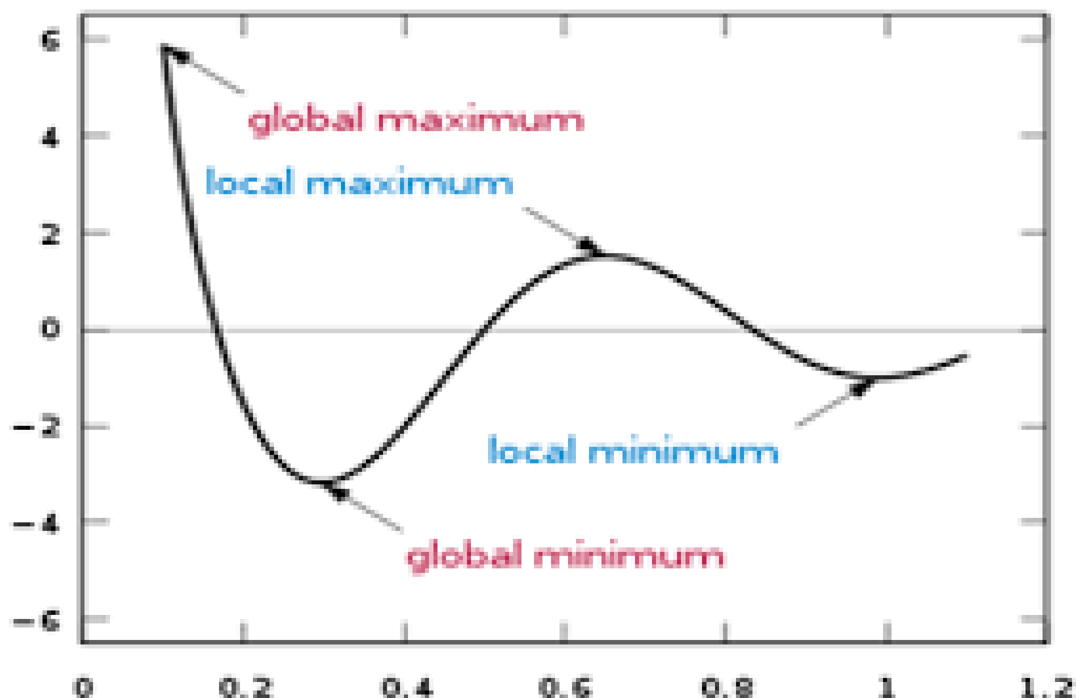

Figure 7.1: Local Minimum

create an agent that could decide on its own which input data were important and which were not. With our neural network approach, our networks weights approach zero for inputs which are insignificant for its decision.

Data Size

The problem with the above method was that due to the obsolete RL approach the data provided to the algorithm, it could not surpass some certain amounts. This is where Neural Networks outmatch classic MDPs and RL algorithms. They do not need a table or any other storage form. Al the information they need is being ”stored” in the network weights. We are able feed it with as many information as possible and never have a storage or latency problem. On the contrary, the more data we can provide it, the better for us and for the agent.

We are going to introduce to you below the three different approaches we tried before ending up to the best algorithmic approach for Mariana Trenchamola. Our agent can easily use one of these three approaches, depending on our user’s demands.

7.1 Background

Let us first state our implementation in the technical level. We have the Tiramola architecture with its modules, as presented in section 4.1. We have three modules:

- Monitoring module. This module provides us with input data from the environment. At the simulations, in replace of the monitoring module we just construct out data and feed our system. Our system input parameters which also **indicate our state** are:

- the percentage of free ram
  - our cluster storage capacity
  - the percentage of the read loads
  - the number of VMs
  - the number of CPUs
  - the I/O of the cluster per second
  - the percentage of CPU usage
  - the total load
  - the total capacity

- Cluster coordinator module. This module adds or removes VMs from our cluster. In our case it can only add or replace one VM.

- Decision making module. In our case, at each section of the below sections we are going to use a different algorithmic approach, all based in the idea behind Deep RL. The part of the reward function which we set, plays significant role to the outcome of the algorithm. Our reward function varies from if it is a simulation or it is a real life experiment. The main objective behind every reward function is to maximize our throughput while minimize the network latency and the number of VMs.

7.1.1 Our network

Our network is a 3-layer fully connected network. It is important to mention that we had to test different type of networks, number of layers and number of neurons at each layer before we end up to this approach. This approach surely can be modified to get even better results. The think with Neural Networks is that one can only go with trial and error approach on network's architecture before finding the optimal solution for his specific problem. The first layer consists of 64 neurons, the second 128 neurons and the third 256 neurons. Our activation function at each layer is a relu function [28].

As a trainer we use tesnorflow's RMSPropOptimizer [29]. In all of our cases we use mini batches of for learning. RMSProp algorithm is an algorithm for handling this kind of problems. What it does is that it divides the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight. This is the mini-batch version of just using the sign of the gradient. RMSProp has shown excellent adaptation of learning rate in different applications. RMSProp can be seen as a generalization of Rprop and is capable to work with mini-batches as well opposed to only full-batches. There is also a memory. As memory we call the buffer where we store passed experiences of our agent, meaning (s_t, a_t, s_{t+1}, r_t). The function sample, of the object memory, selects every time a random sample of our buffer data so that we can feed our networks and train it from the beginning.

- **Input size** is the number of our network parameters
- **batch size** is the number episodes that we are going to take from our memory everytime we make a training step to our network
- **Pretrain steps** is the number of steps our algorithm is going to take until we start using mini-batches from our memory.
- **Experience replay** finally is up to us whether or not, we are going to use experience replay to our algorithm.

So now I think that we are ready, to fully understand the three below versions of our algorithmic approach.

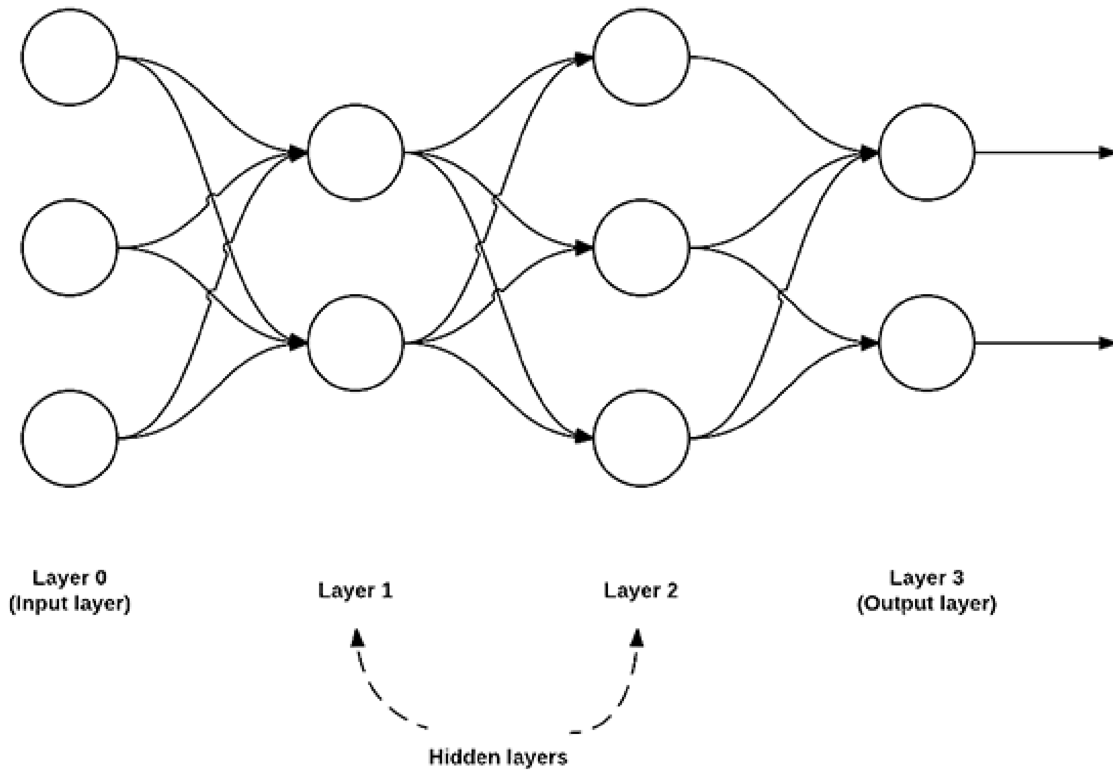

Figure 7.2: Example of 3-layer network

7.2 Simple Deep Rl on Mariana Trenchamola

In this version we implement the Deepmind's approach as published in 2013. We only have one deep neural network which we use both for taking a decision and for computing the targets using the Bellman's update, described in subsection **??** .

7.2.1 The algorithm

Initialize replay memory D to capacity N Initialize action-value function Q with random weights θ
for episode = 1, M **do**
Initialise sequence $s_1 = x_1$ and preprocessed sequenced $_1 = (s_1)$
for t=1,T **do**
With probability ε select a random action a_t
otherwise select $a_t = \max_a Q * ((s_t), a; \theta)$
Execute action a_t and observe reward r_t and image $x_(t+1)$
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $_{t+1} = (s_{t+1})$
Store transition $(_t, a_t, r_t, _(t+1))$ in D
Sample minibatch of transitions $(_j, a_j, r_j, _(j+1))$ from D

$$Sety_j = \begin{cases} r_j & forterminal_{j+1} \\ r_j + \gamma max'_a Q(_{j+1}, a'; \theta) & fornonterminal_{j+1} \end{cases} \quad (7.1)$$

perform a gradient descent step on $(y_j - Q(_j, a_j; \theta))^2$
end
end

Algorithm 2: Simple Deep Q learning

7.3 Full Deep Rl on Mariana Trenchamola

The second network is a clone of our main network and is being used to compute the target values. It is used to generate the target-Q values that will be used to compute the loss for every action during training. Why can we just use one network for both estimations? The reason is that at every step of training, the Q-network's values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated Q-values. In order to mitigate that risk, the target network's weights are fixed, and only periodically or slowly updated to the primary Q-networks values. In this way training can proceed in a more stable manner. Instead of updating the target network periodically and all at once, we will be updating it frequently, but slowly. An approach of this technique was introduced in DeepMind's 2015 paper [15], where they found that it stabilized the training process.

Every C steps we are re-evaluate our target networks with our main network values.

7.3.1 The algorithm

Initialize replay memory D to capacity N Initialize action-value function Q with random weights θ
Initialize target action-value function Q' with weights $' =$
for episode = 1, M **do**
Initialise sequence $s_1 = x_1$ and preprocessed sequenced $_1 = (s_1)$
for t=1,T **do**
With probability ε select a random action a_t
otherwise select $a_t = \max_a Q * ((s_t), a; \theta)$
Execute action a_t and observe reward r_t and image $x_(t + 1)$
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $_{t+1} = (s_{t+1})$
Store transition $(_t, a_t, r_t, _(t + 1))$ in D
Sample minibatch of transitions $(_j, a_j, r_j, _(j + 1))$ from D

$$Sety_j = \begin{cases} r_j & forterminal_{j+1} \\ r_j + \gamma max'_a Q(_{j+1}, a'; \theta) & fornonterminal_{j+1} \end{cases} \quad (7.2)$$

perform a gradient descent step on $(y_j - Q(_j, a_j; \theta))^2$
Every C-steps reset $Q' = Q$
end
end

Algorithm 3: Full Deep Q learning

7.4 Double Deep Rl on Mariana Trenchamola

The main intuition behind Double DQN [37] is that the regular DQN often overestimates the Q-values of the potential actions to take in a given state. While this would be fine if all actions were always overestimated equally, there was reason to believe this wasn't happening. You can easily easily imagine that if specific suboptimal actions were regularly given higher Q-values than optimal actions, the agent would have a hard time ever learning the ideal policy. So, in order to correct this, the authors of DDQN paper propose a simple trick: instead of taking the max over Q-values when computing the target-Q value for our training step, we use our primary network to choose an action, and our target network to generate the target Q-value for that action. By decoupling the action choice from the target Q-value generation, we are able to substantially reduce the overestimation, and train faster and more reliably.

The new DDQN equation for updating the target value is:

$Q - Target = r + \gamma Q(s', argmax(Q(s', a,), '))$

Theorem 3. *Consider a state s in which all the true optimal action values are equal at $Q * (s, a) = V * (s)$ for some $V * (s)$. Let Q_t be arbitrary value estimates that are on the whole unbiased in the sense that $\sum_a (Q_t t(s, a) - V * (s)) = 0$, but that are not all correct, such that $\frac{1}{m} \sum_a (Q_t(s, a) - V * (s))2 = C$ for some $C > 0$, where $m \geq 2$ is the number of actions in s. Under these conditions, $max_a Q_t(s, a) \geq V * (s) + \sqrt{\frac{C}{m-1}}$. This lower bound is tight. Under the same conditions, the lower bound on the absolute error of the Double Q-learning estimate is zero.*

The lower bound in Theorem 1 decreases with the number of actions. This is an artifact of considering the lower bound, which requires very specific values to be attained. More typically, the overoptimism

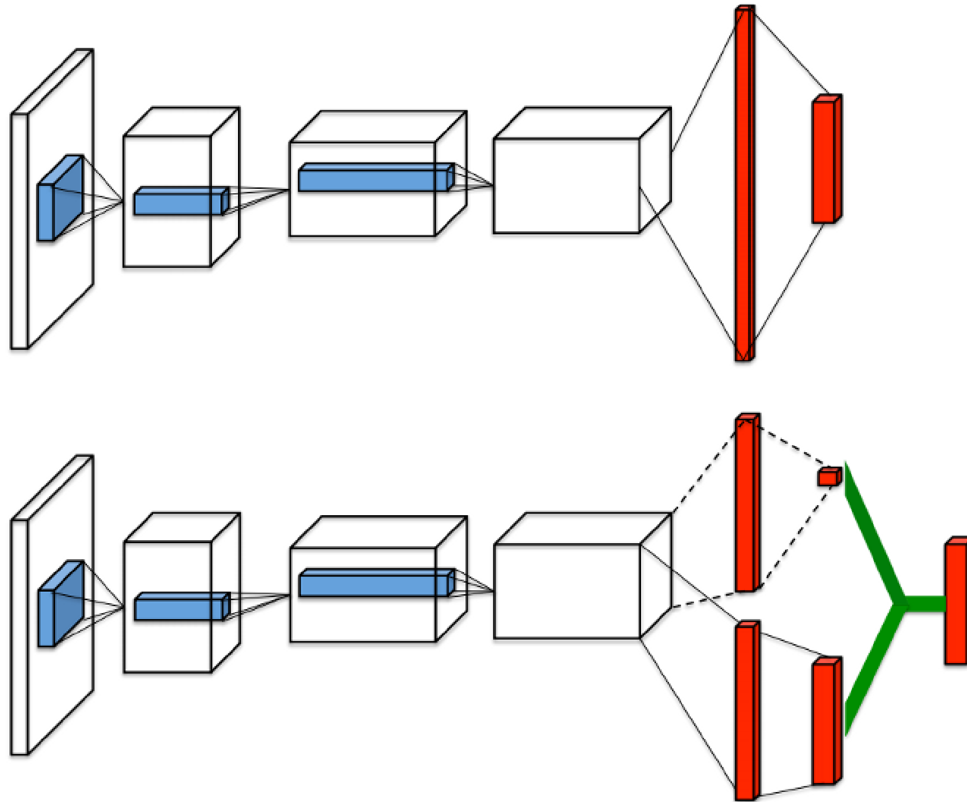

Figure 7.3: At the first image we see a simple DQN and at the second we see a Double DQN

increases with the number of actions as shown in Figure 7.4. Q-learning's overestimations there increase indeed with the number of actions, while Double Q-learning is unbiased.

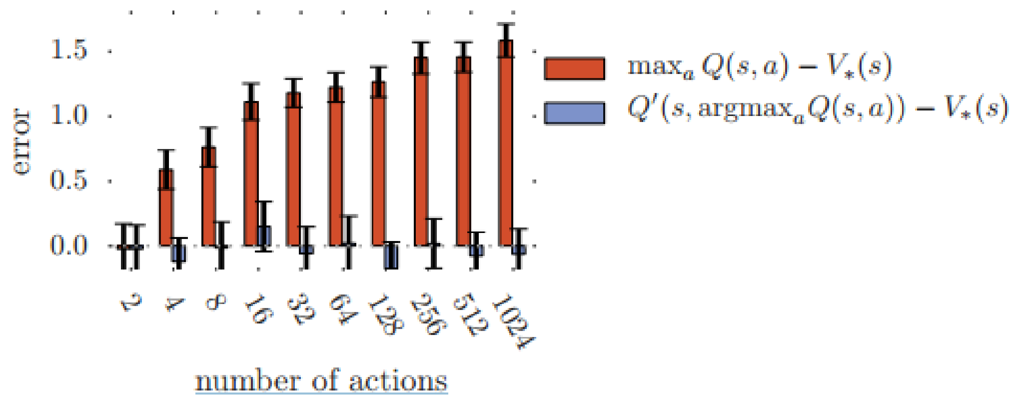

Figure 7.4: The orange bars show the bias in a single Qlearning update. The second set of action values Q, used for the blue bars, was generated identically and independently. All bars are the average of 100 repetitions

The goal of Double Q-learning is to reduce overestimations. This is getting done by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. So it is better to evaluate the greedy policy according

to the online network, but using the target network to estimate its value. In reference to both Double Q-learning and DQN, the algorithm name was Double DQL. Its update is the same as for DQN, but replacing the target Y^DQN with

$$Y^DQN = r_{t+1} + \gamma Q(s_{t+1}, argmax_a(Q(st+1, a; \theta_t)\theta'))$$

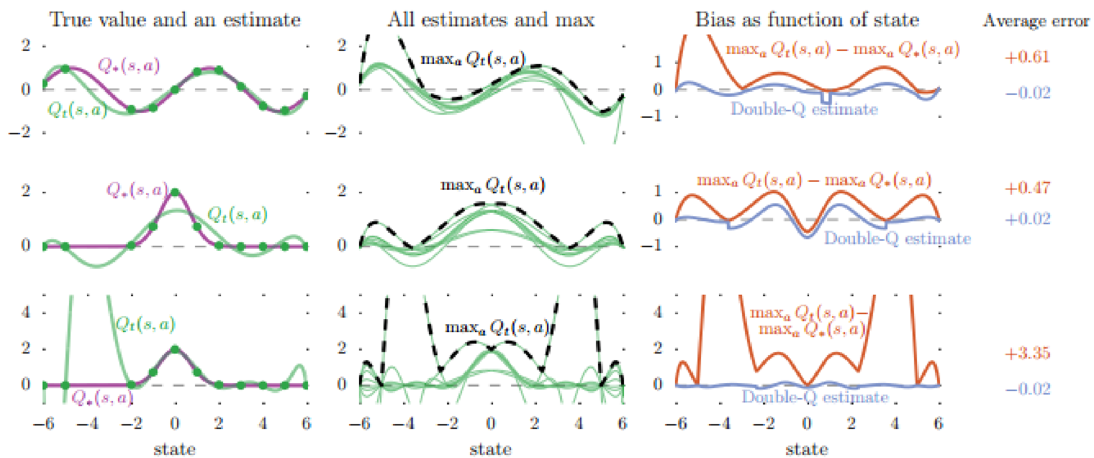

Figure 7.5: Illustration of overestimations during learning. In each state (x-axis), there are 10 actions. The left column shows the true values V□(s) (purple line). All true action values are defined by Q□(s, a) = V□(s). The green line shows estimated values Q(s, a) for one action as a function of state, fitted to the true value at several sampled states (green dots). The middle column plots show all the estimated values (green), and the maximum of these values (dashed black). The maximum is higher than the true value (purple, left plot) almost everywhere. The right column plots shows the difference in orange. The blue line in the right plots is the estimate used by Double Q-learning with a second set of samples for each state. The blue line is much closer to zero, indicating less bias. The three rows correspond to different true functions (left, purple) or capacities of the fitted function (left, green).

7.4.1 The algorithm

Initialize replay memory D to capacity N Initialize action-value function Q with random weights θ
Initialize target action-value function Q' with weights $' =$
for episode = 1, M **do**
Initialise sequence $s_1 = x_1$ and preprocessed sequenced $_1 = (s_1)$
for t=1,T **do**
With probability ε select a random action a_t
otherwise select $a_t = \max_a Q * ((s_t), a; \theta)$
Execute action a_t and observe reward r_t and image $x_(t + 1)$
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $_{t+1} = (s_{t+1})$
Store transition $(_t, a_t, r_t, _(t + 1))$ in D
Sample minibatch of transitions $(_j, a_j, r_j, _(j + 1))$ from D

$$Sety_j = \begin{cases} r_j & forterminal_{j+1} \\ r_j + \gamma Q(_{j+1}, argmax(Q(_{j+1}, a, \theta), \theta')) & fornonterminal_{j+1} \end{cases} \tag{7.3}$$

perform a gradient descent step on $(y_j - Q(_j, a_j; \theta))^2$
Every C-steps reset $Q' = Q$
end
end

Algorithm 4: Full Deep Q learning

Chapter 8

Simulation results

Now we are going to present results from a number of simulations. The goal of the simulations is to better understand the behaviour of the reinforcement learning models described, in the context of resource allocation problems in a cloud computing environment. Our goal is to better understand the behavior of Deep RL models that we described in chapter 7, in problems where our agent has to perform elasticity decisions in a cloud computing environment. For the simulations we are using the setup that Konstadinos Lolos described in his deploma Thesis [17], so we can test the performance of our approach in oppose to Lolos approach as well as former versions of Tiramola approaches. So we firstly creating Lolos environment, then testing our approach and changing some of our network parameters and lastly testing our algorithms versus former Tiramola's algorithms.

The following algorithms will be tested throughout this chapter:

- MDP The full-model based Markov Decision Process approach, having a fixed number of states and maintaining transition and reward information in its Q-states
- Q-learning The model-free reinforcement learning approach, also having a fixed number of states but not maintaining transition and reward information
- MDDPT A full-model based decision tree implementation
- QDT The Q-learning decision tree algorithm
- Simple Deep Q learning The deep RL algorithm designed for mariana trenc covered in section 7.2
- Full Deep Q learning The full deep rl algorithm designed for mariana trench covered in section 7.3
- Double Deep Q learning The double deep rl algorithm designed for mariana trench covered in section 7.4

All simulations were implemented in Python, using google's tensorflow [1] and numpy library. The environment was create with anaconda package-environment manager[2].

8.1 Parameterization

To correctly setup our network we firstly experiment with a number of different options that affect our agent performance. We will be using a simulation scenario from the field of cloud computing. In our scenario, the agent is asked to make elasticity decisions that resize a cluster running a database under a varying incoming load. The load consists of read and write requests, and the capacity of the cluster depends on its size as well as the percentage of the incoming requests that are reads. Specifically:

- The cluster size can vary between 1 and 20 virtual machines
- The available actions to the agent in each step are to increase the size of the cluster by one, decrease the size of the cluster by one, or do nothing.
- The incoming load is a sinusoidal function of time: $load(t) : 50 + 50\sin(\frac{2\pi t}{250})$
- The percentage of incoming requests that are reads is a sinusoidal function of time with a different period: $r(t) : 0.75 + 0.25\sin(\frac{2\pi t}{340})$
- If $vms(t)$ is the number of virtual machines currently in the cluster, the capacity of the cluster is given by: $capacity(t) = 10vms(t)r(t)$
- The reward for each action depends on the state of the cluster after executing the action and is given by: $R_t = min(capacity(t+1), load(t+1)) - 3vms(t+1)$.

As we can see the reward function encourages the agent to increase the size of the cluster to the point where it can fully serve the incoming load, but punishes it for going further than that. In order for the agent to behave optimally, it needs to not only identify the way its actions affect the cluster's capacity and the dependence on the level of the incoming load, but also the dependence on the types of the incoming requests. We shall not forget that one of our algorithms goals is to be able to recognise which input parameters matter for the outcome and which are not. So we feed our agent with 7 more randomly valued parameters. MDDPD and QDT managed to recognise and ignore those parameters as Simple DQN, Full DQN and Double DQN did.

All tests included a training phase and an evaluation phase. During the training phase, the selected action in each step was a random action with probability e, or the optimal action with probability 1-e (e-greedy strategy). During the evaluation phase only optimal actions were selected, as proposed by the algorithm. The metric through which different options are compared is the sum of rewards the agent managed to accumulate during the evaluation phase.

Simulation

Setup:

- Training steps: 5000
- Evaluation steps: 2000
- Algorithms: Simple DQN, Full DQN, Double DQN
- Statistical test max error 10^{-6}

We are going to test three different parameters and compare our results,

- The batch size
- The annealing steps
- The learning rate

Discount Factor

Surprisingly enough **we do not have to test different values for our discount factor** as it turns out that it plays no role at the outcome of the algorithm. That may seem strange at the beginning but if we give it a closer look it makes sense. As we can see Deep Rl uses experience replay buffers with a major number of different experience that train the algorithm at every step of its execution. In a simple RL problem it is useful to look at the future rewards, but in Deep Rl we train our agent with a batch of random experience which might be past, present, or future experienced rewards at every step. Given that we have a very small size of input parameters our agent does not need to more extract informations than it already does. So our snick pic on the future does not affects our agent neither positive or negative.

8.1.1 Testing the Batch size

Firstly we test different values for our batch size, which is the memory buffer that we use to train our network with at every step. For that we obtain our learning rate at 0.00025 and our annealing steps equal to our training steps, 5000. We are using 3 different batch sizes, in sizes of 20, 80 and 360 experiences. We are testing them in both Simple DQN and Double DQN.

8.1.2 Batch size=20

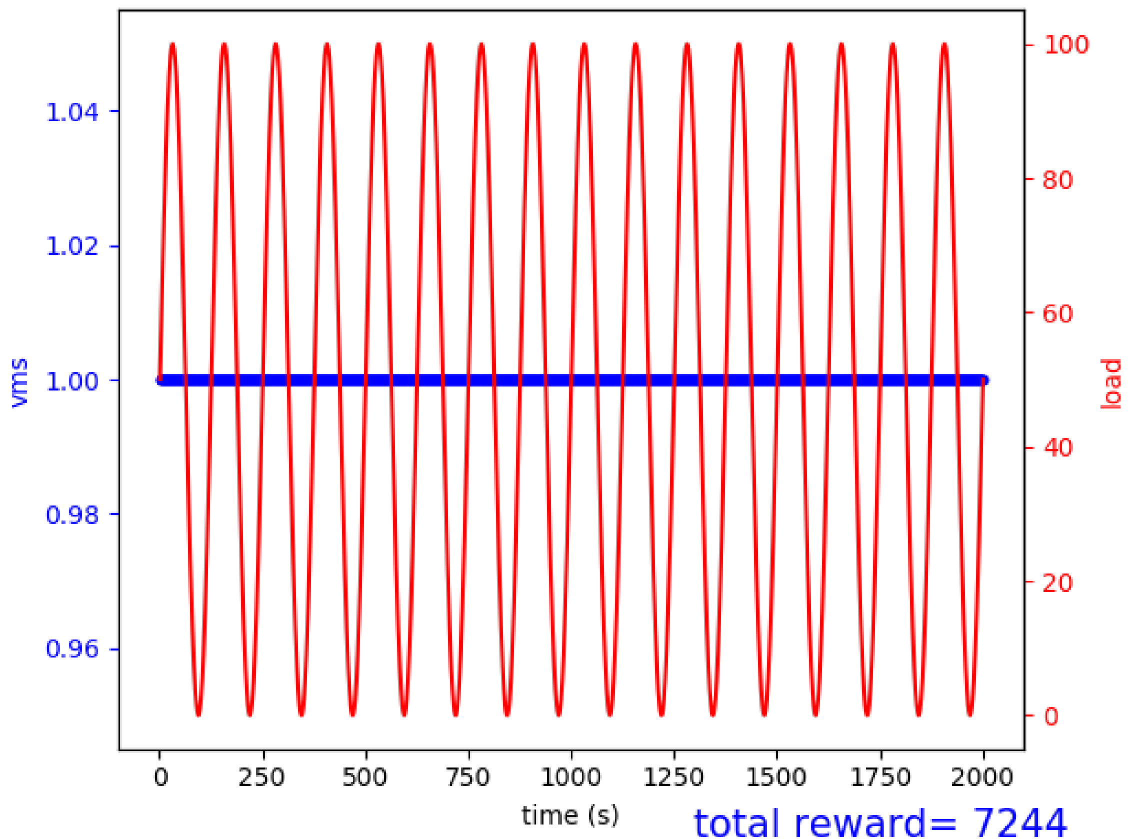

Figure 8.1: Simple DQN

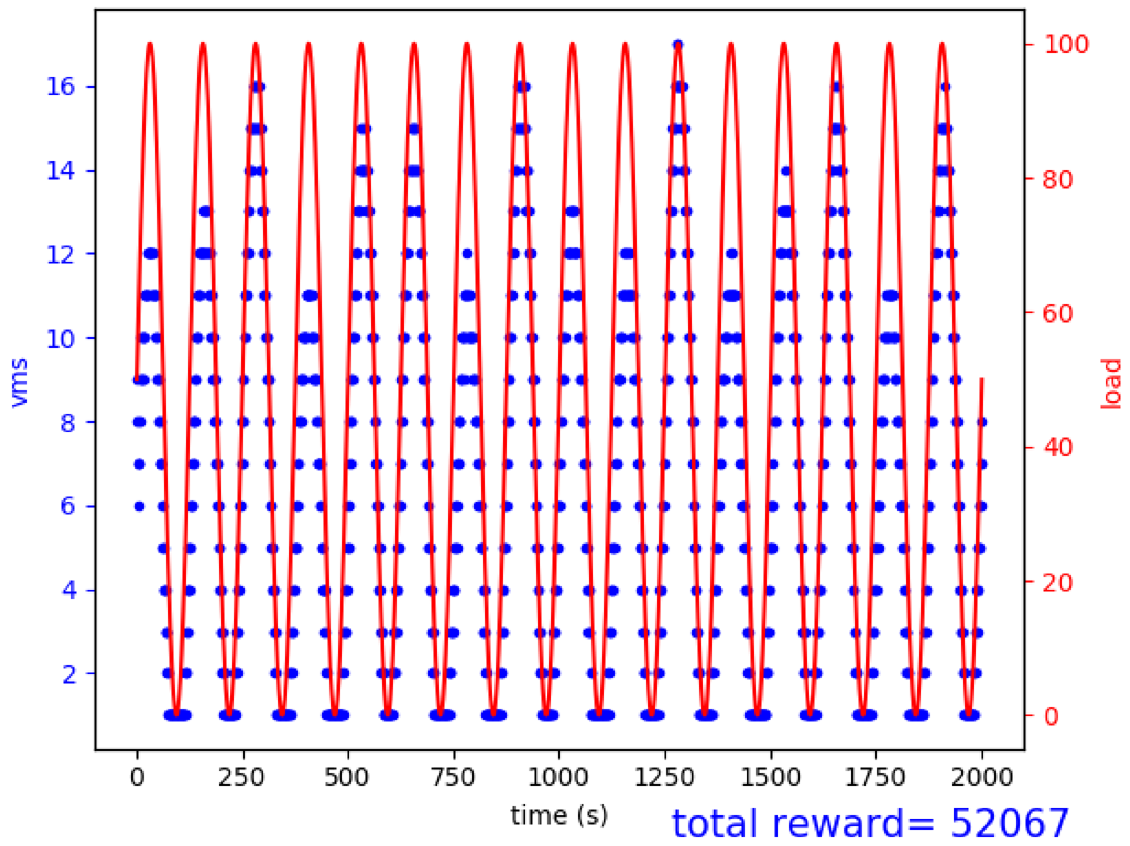

Figure 8.2: Double DQN

8.1.3 Batch size=80

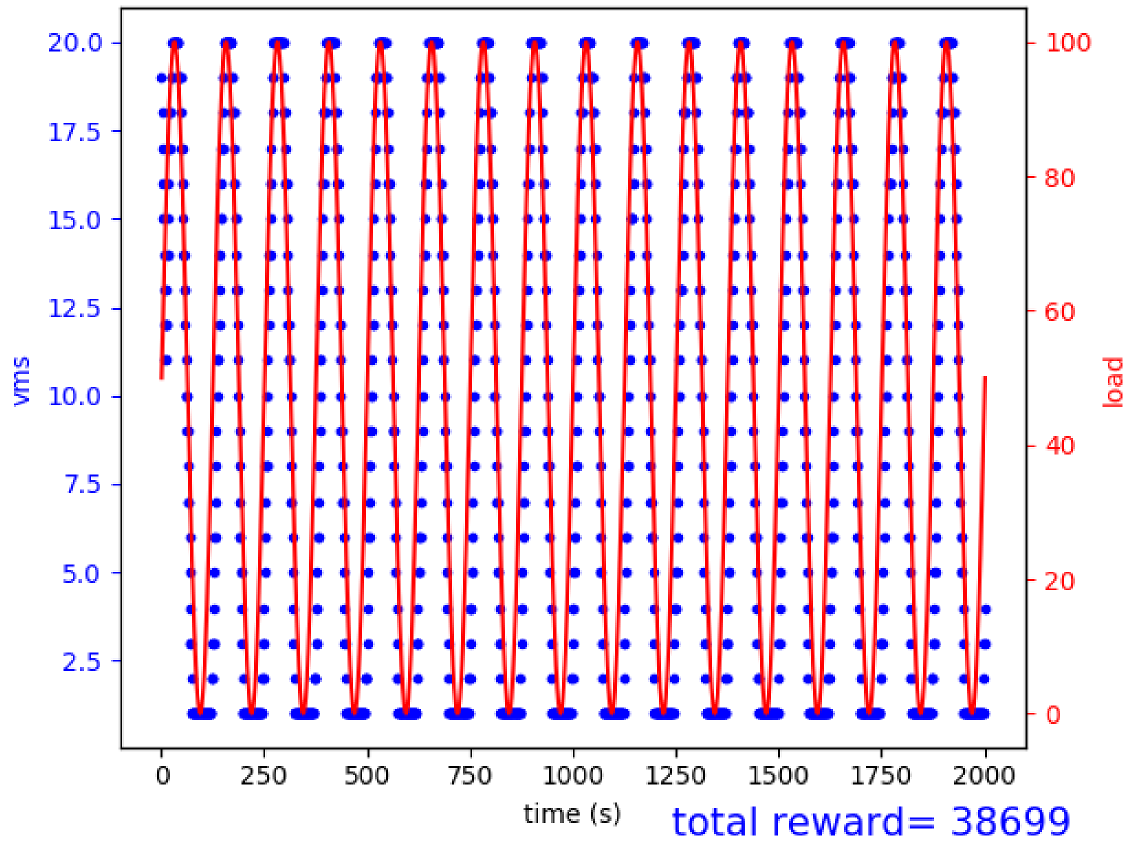

Figure 8.3: Simple DQN

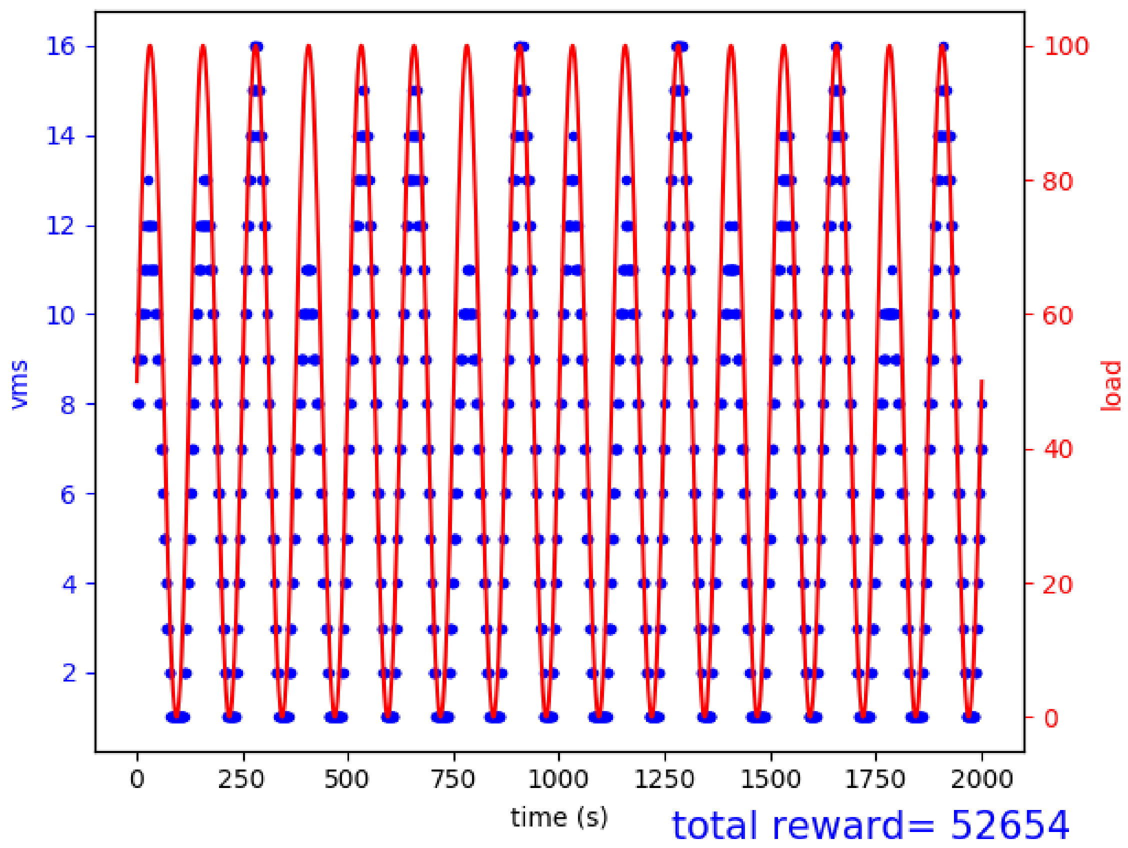

Figure 8.4: Double DQN

8.1.4 Batch size=360

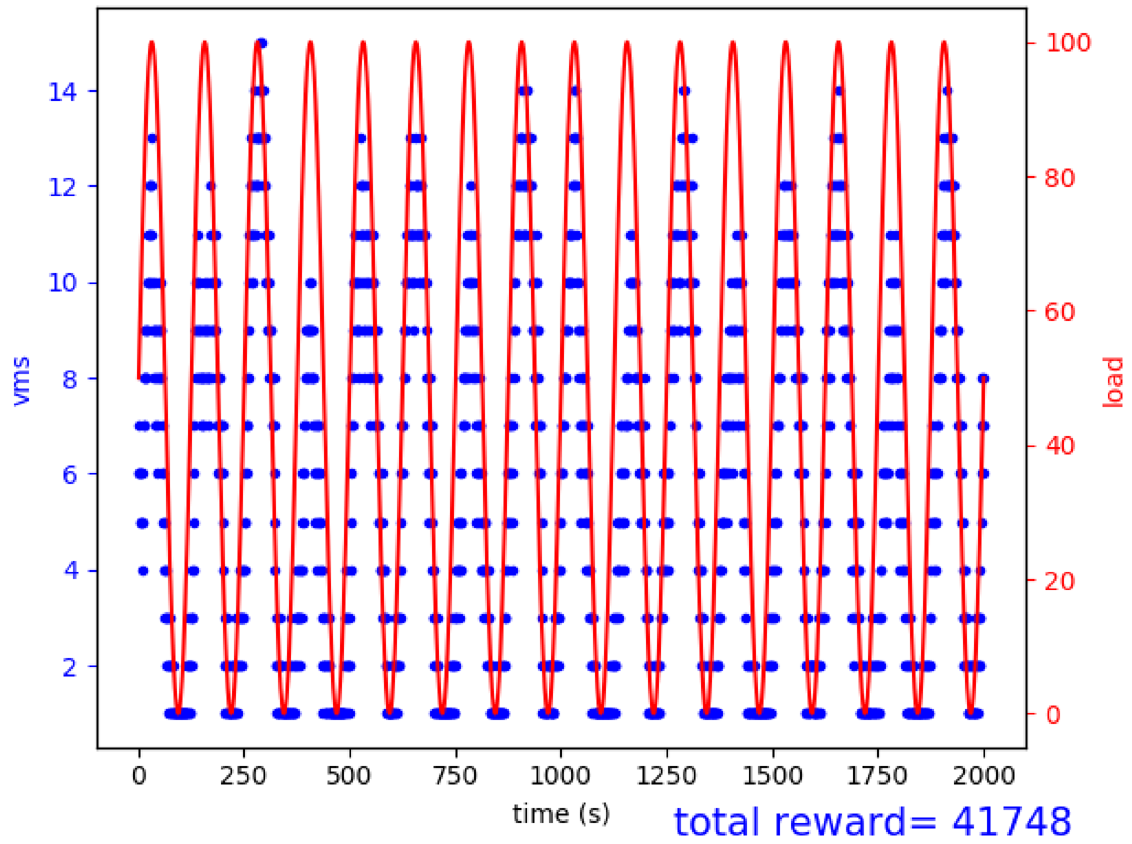

Figure 8.5: Simple DQN

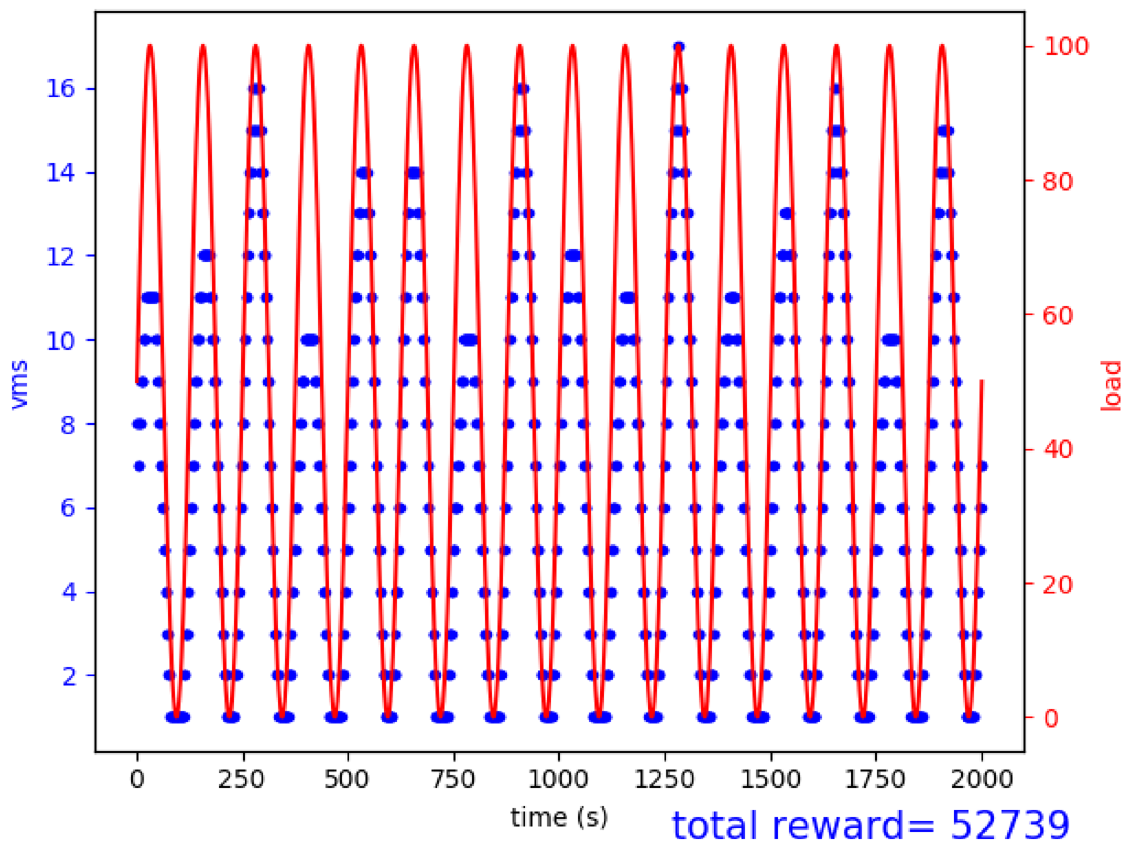

Figure 8.6: Double DQN

As we can see when using Simple DQN bigger batch sizes provides us with better results, where smaller batch sizes give us questionable results. In this particular example depicted our algorithm fails to converge to optimal behaviour. Double DQN on the other hand give us the same results for all batch sizes.

8.1.5 Testing the annealing steps

As we saw in the algorithms section every $annealing_steps/10$ we tell our agent to take the optimal decision and not a random one. Thus is, that every $annealing_steps/10$ we reduce the e, which starts at value 1. So when we have a small value of annealing steps our agent explores more optimal decisions form the begging, meaning we give more attention to the **exploitation**, whereas when we have a bigger value of annealing steps, our agent takes more random action at the beginning, meaning we give more attention to the **exploration**. For that we obtain our learning rate at 0.00025 and our batch size at 360. We are using 3 different annealing steps values, 1000, 5000, 20000. We are testing them in both Simple DQN and Double DQN.

8.1.6 Annealing steps=1000

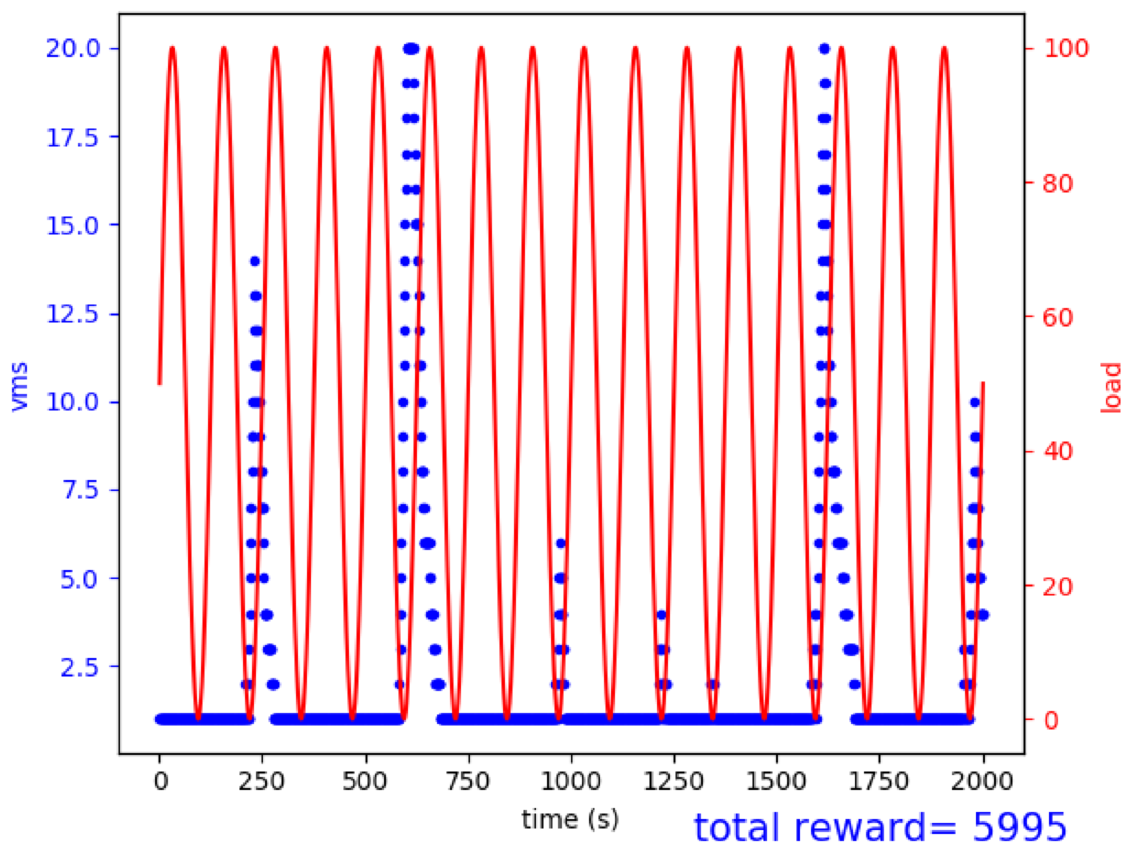

Figure 8.7: Simple DQN

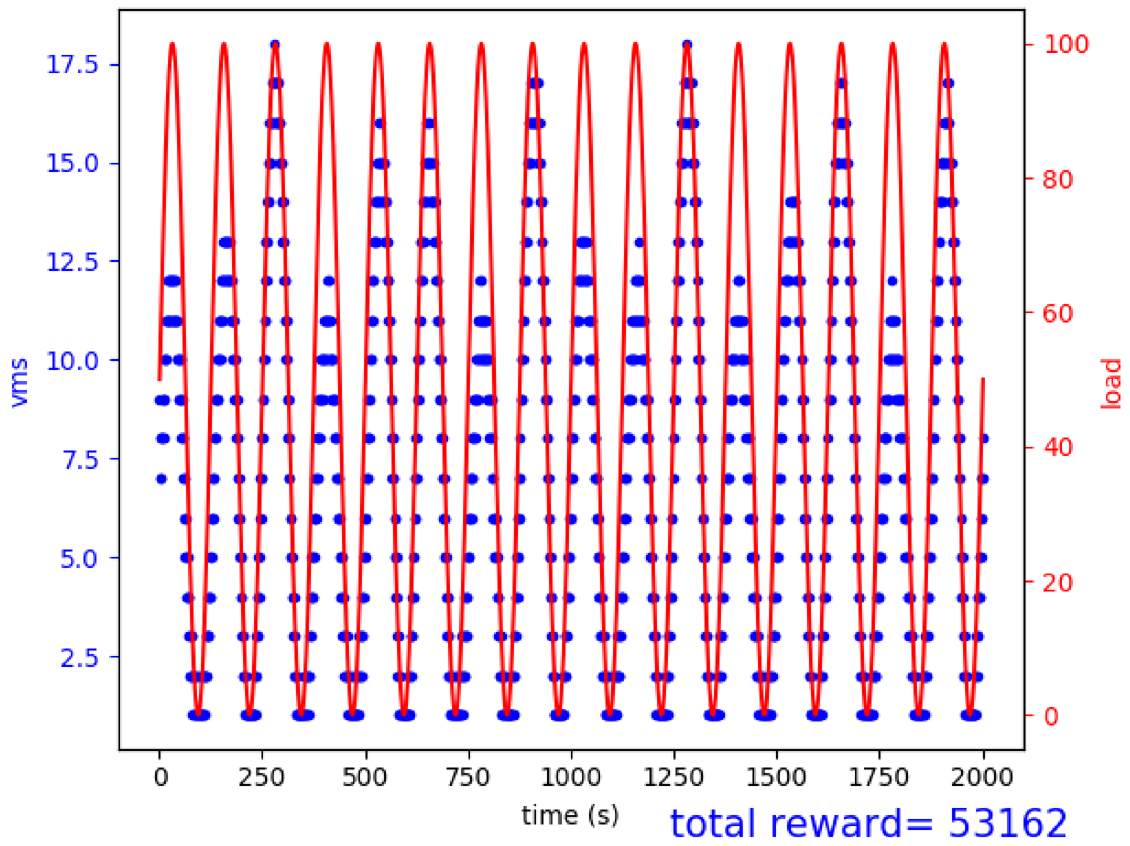

Figure 8.8: Double DQN

8.1.7 Annealing steps=5000

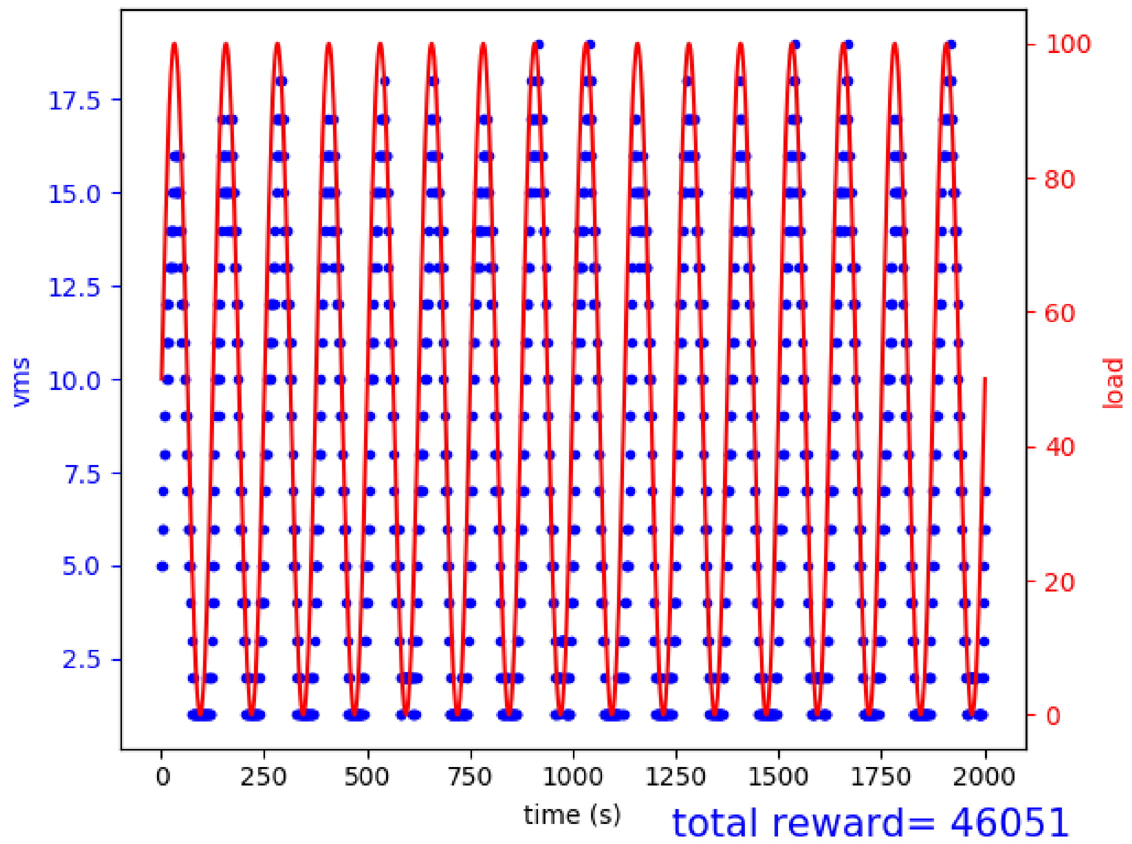

Figure 8.9: Simple DQN

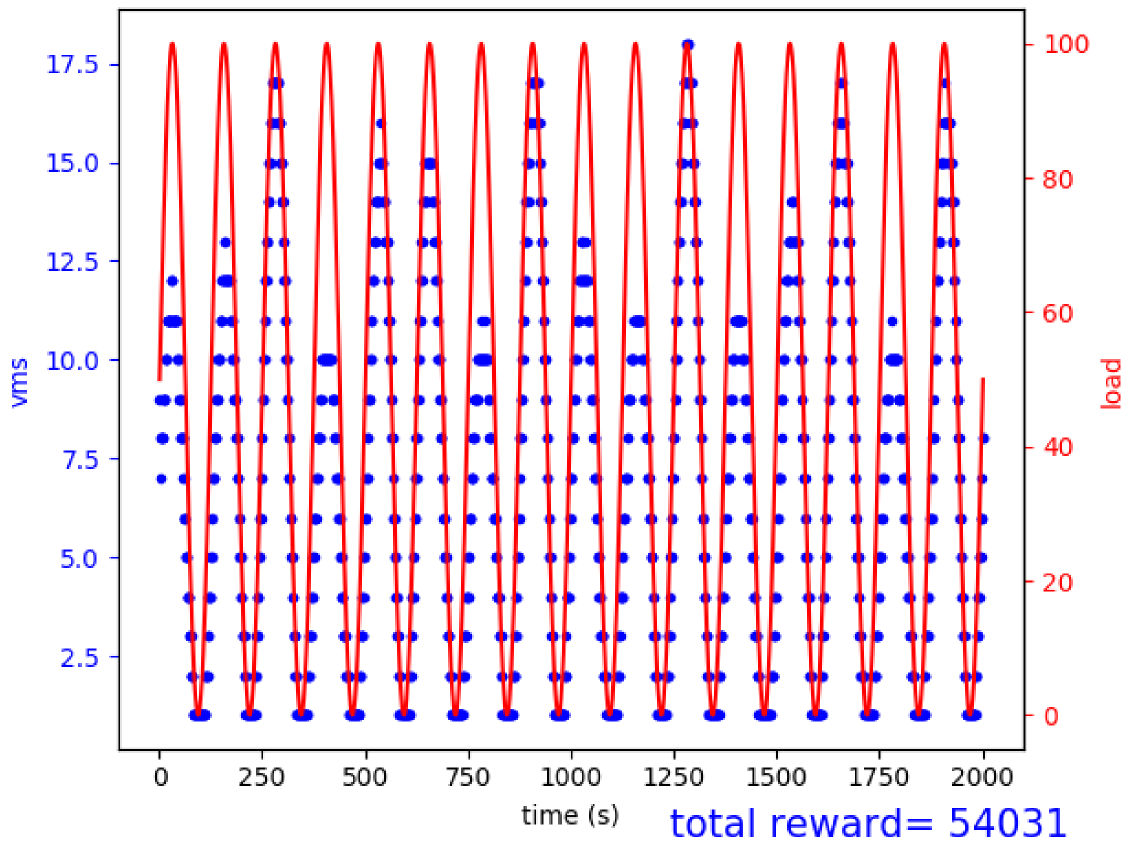

Figure 8.10: Double DQN

8.1.8 Annealing steps=20000

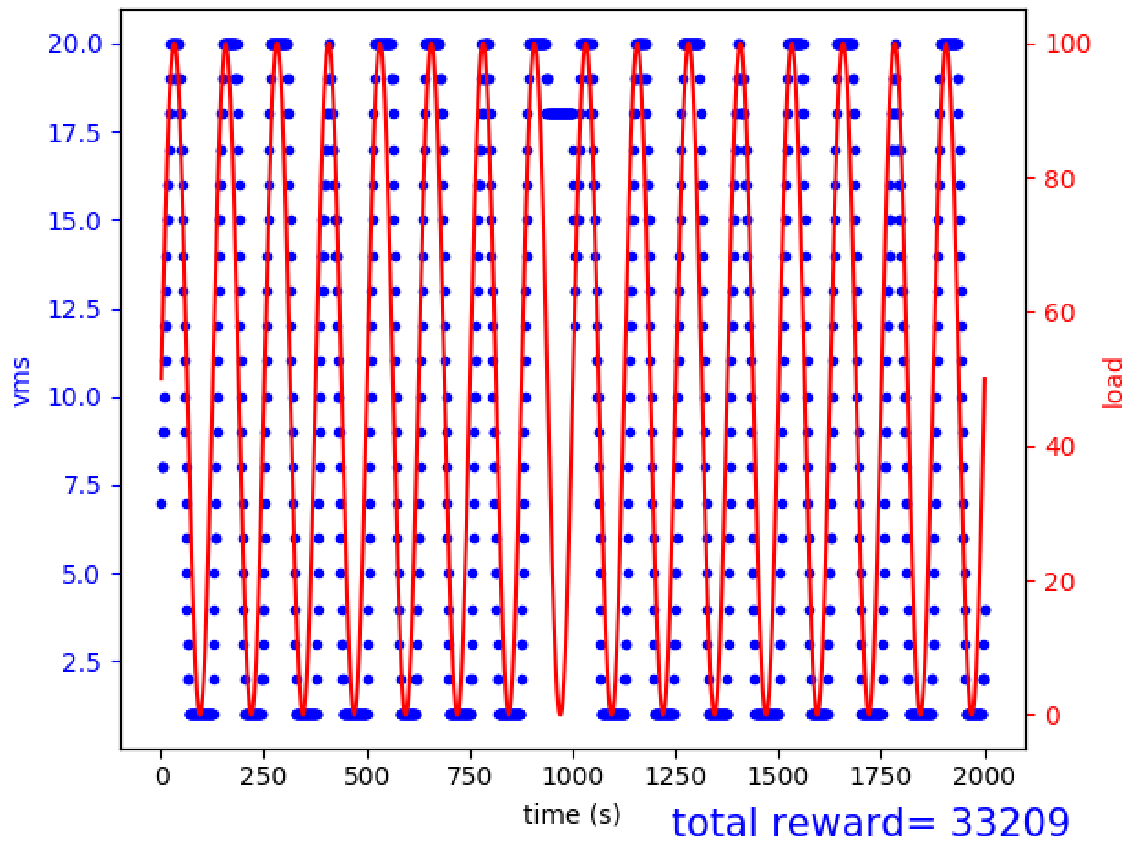

Figure 8.11: Simple DQN

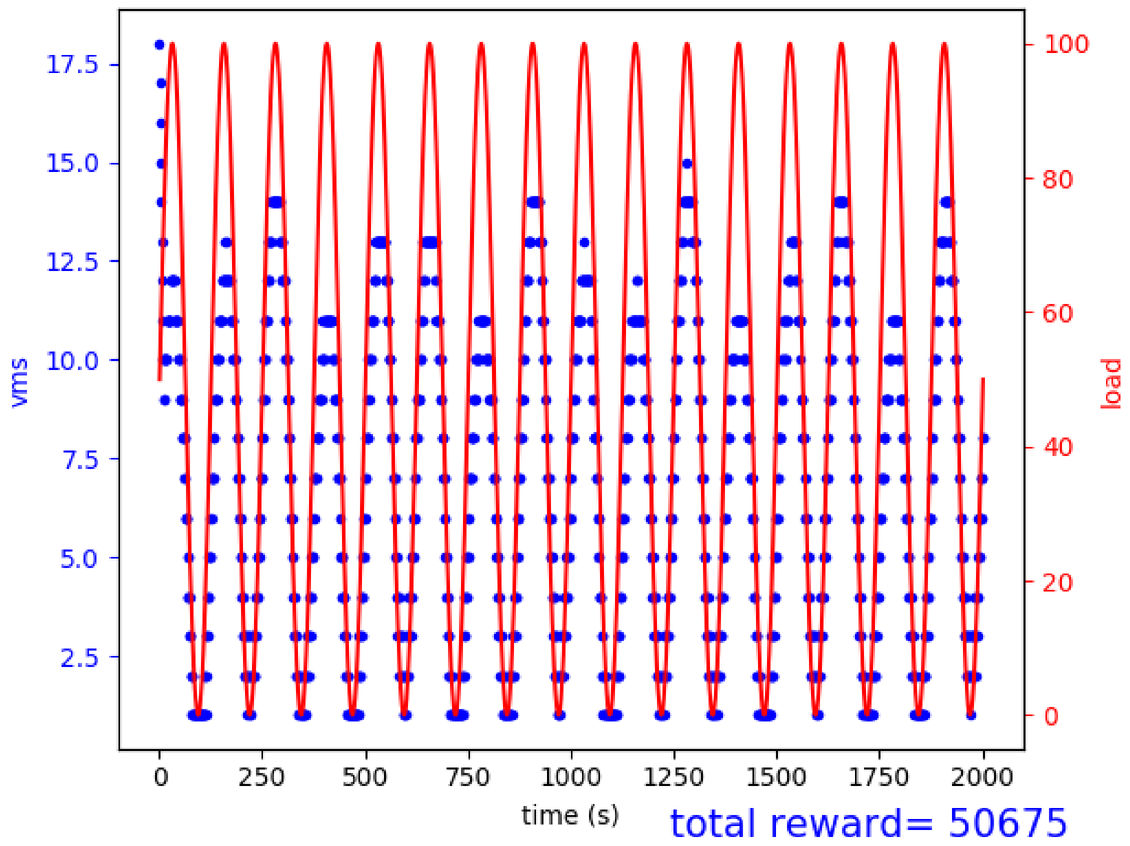

Figure 8.12: Double DQN

As we can see again Double DQN does not have any problem with every different value that we give to our annealing steps, proving once again how powerful tool it is. Simple DQN from the other hand experiences problems when we give it low or high values at annealing steps.

8.1.9 Testing the learning rate

Lastly we are going to test different values for our learning rate. We are going to be messing with the learning rate of the gradient descent method that we use for backpropagation, especially the RM-SpropOptimizer. In order for Gradient Descent to work we must set the learning rate to an appropriate value. This parameter determines how fast or slow we will move towards the optimal weights. If the learning rate is very large we will skip the optimal solution. If it is too small we will need too many iterations to converge to the best values. So using a good learning rate is crucial. For that we obtain our annealing steps at 5000 and our batch size at 360. We are using 3 different learning rate values, 0.0000025, 0.00025, 0.0025. We are testing them in both Simple DQN and Double DQN.

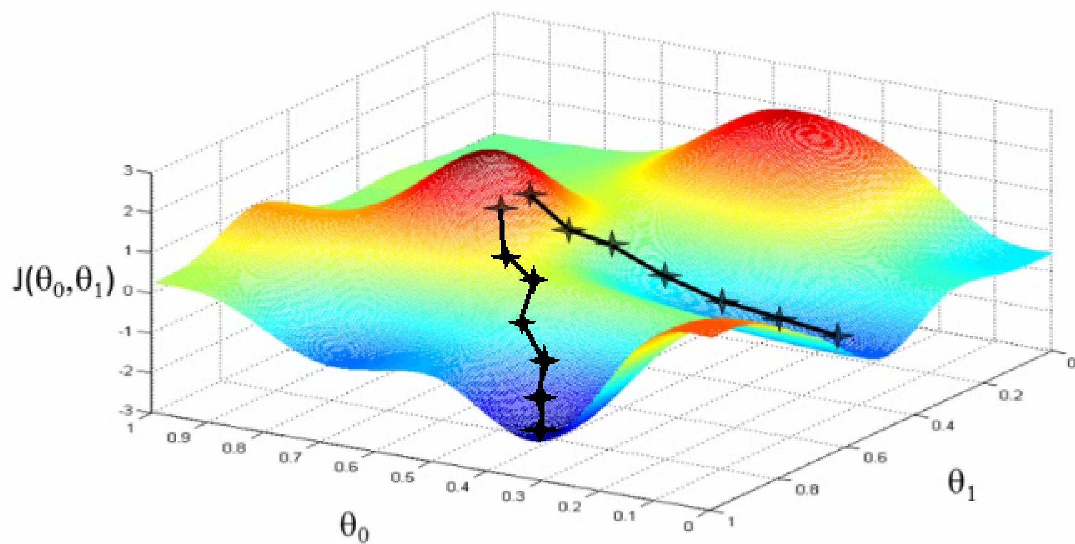

Figure 8.13

8.1.10 Learning rate=0.0025

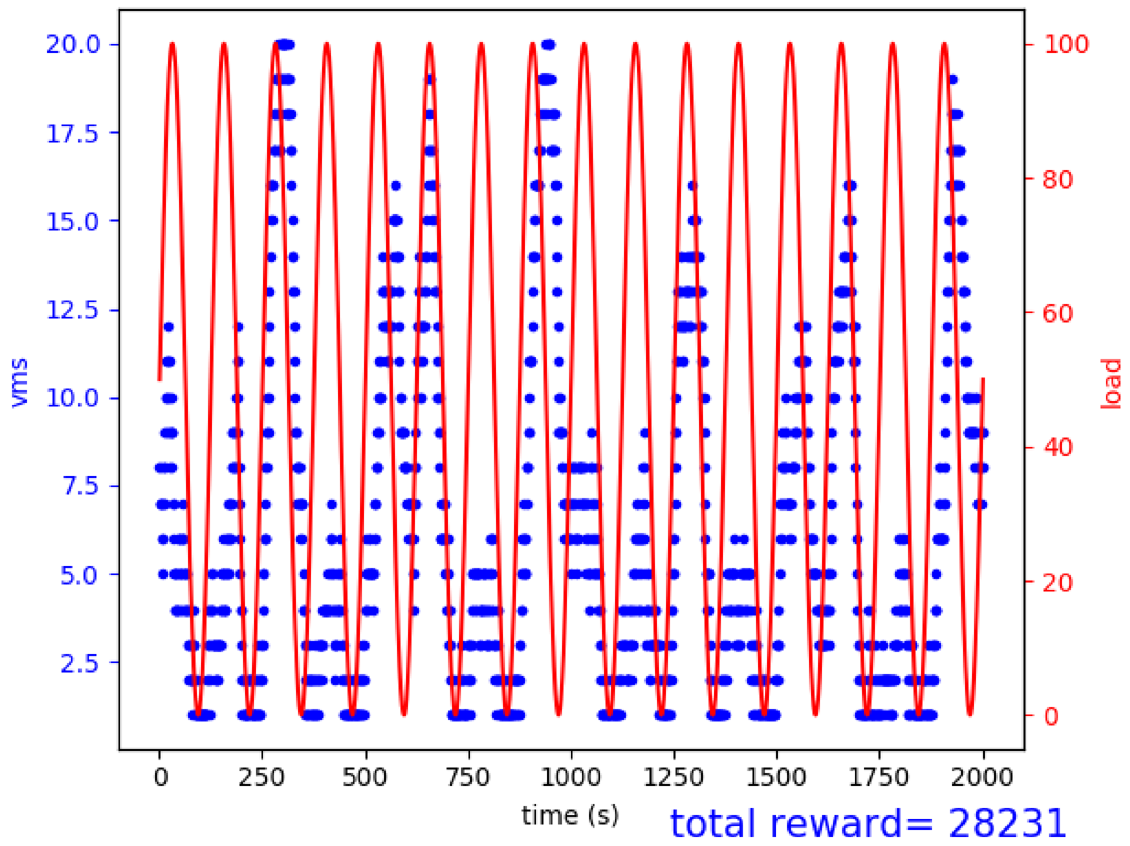

Figure 8.14: Simple DQN

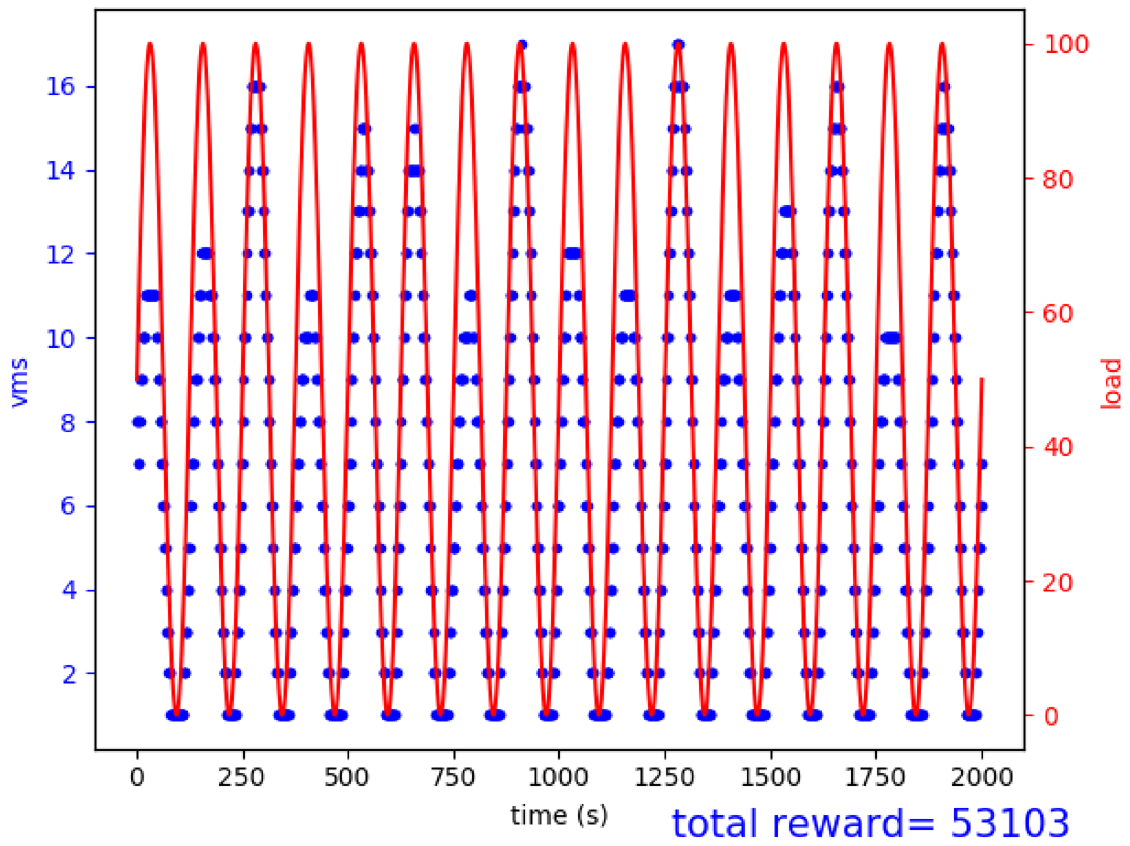

Figure 8.15: Double DQN

8.1.11 Learning rate=0.00025

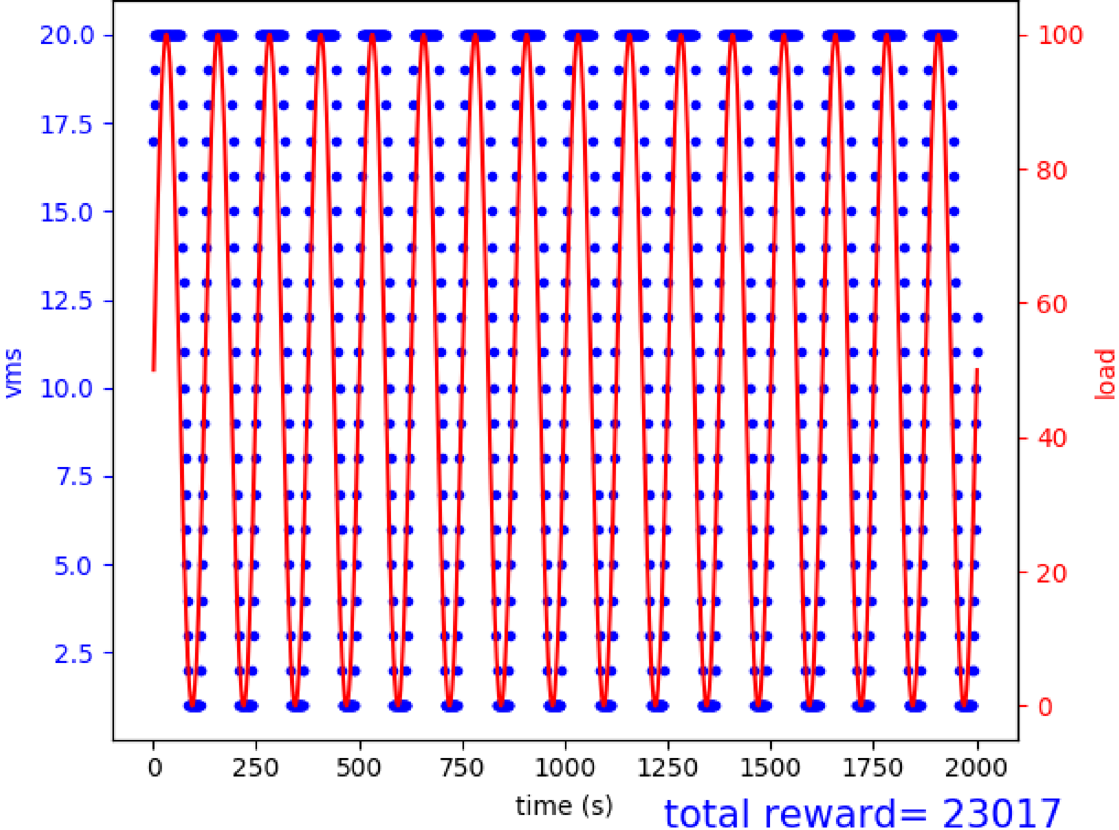

Figure 8.16: Simple DQN

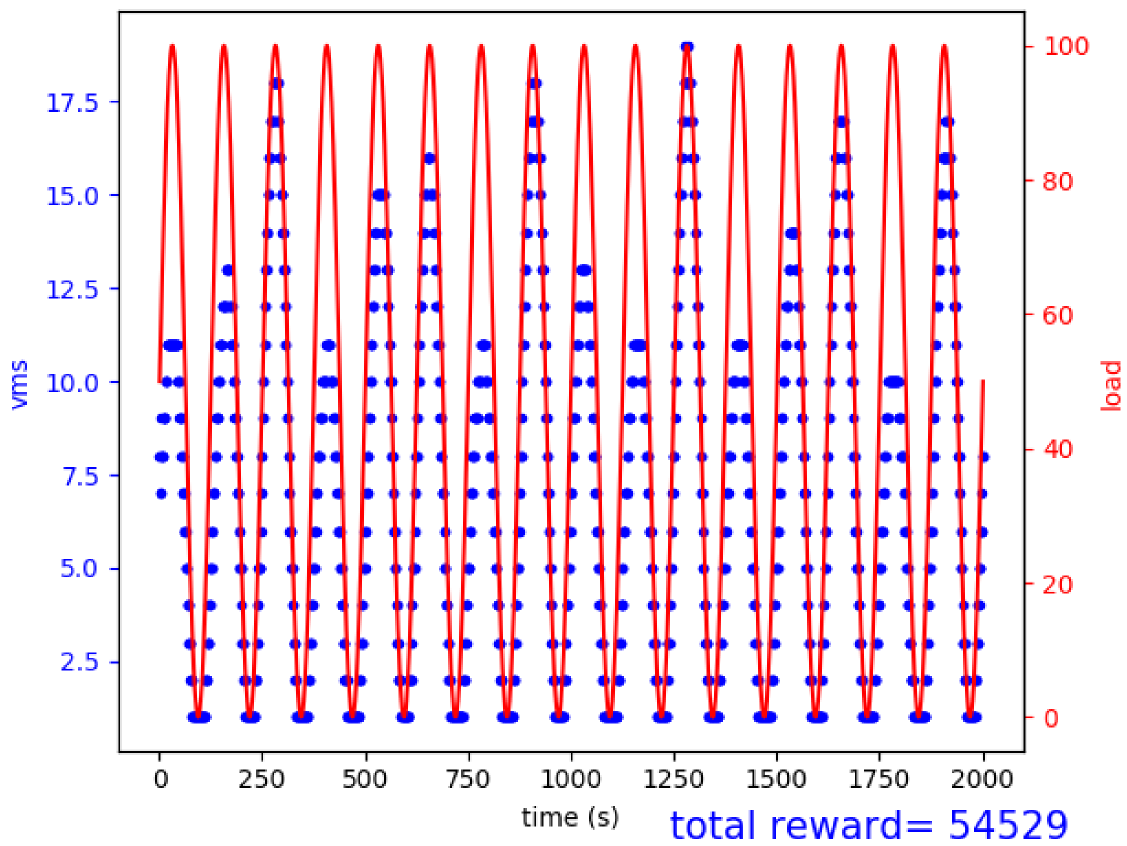

Figure 8.17: Double DQN

8.1.12 Learning rate=0.000025

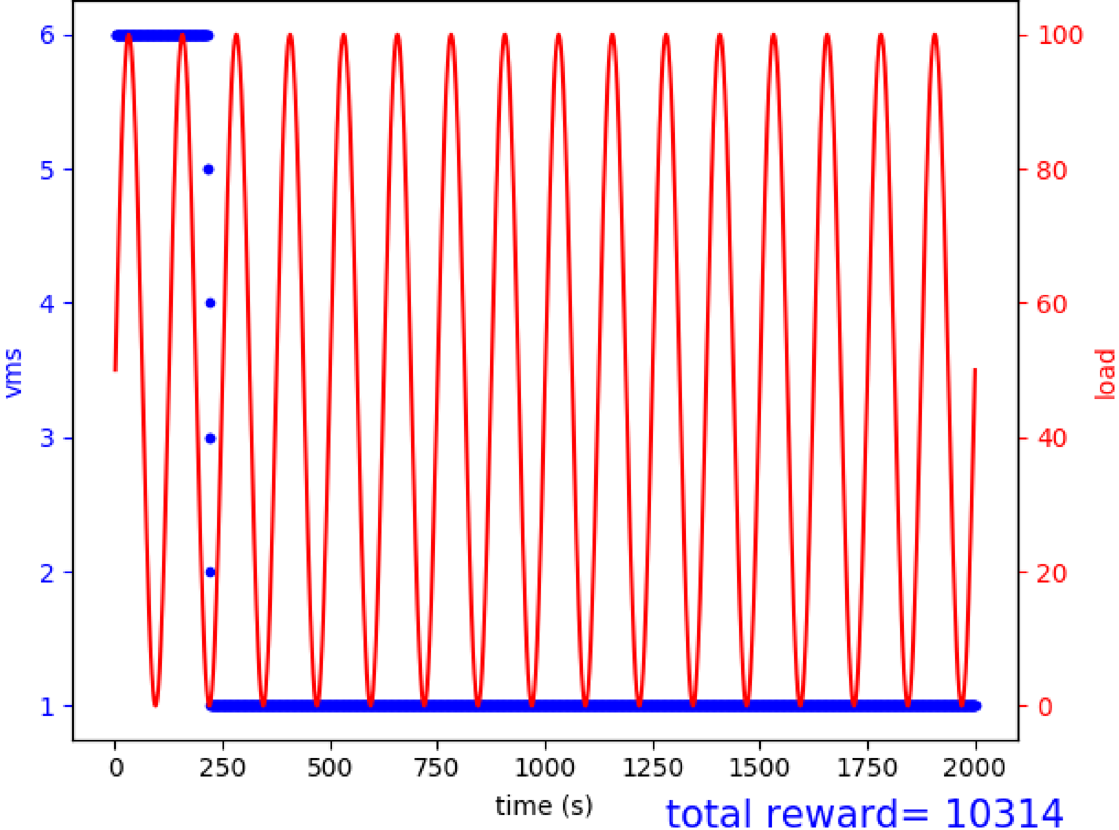

Figure 8.18: Simple DQN

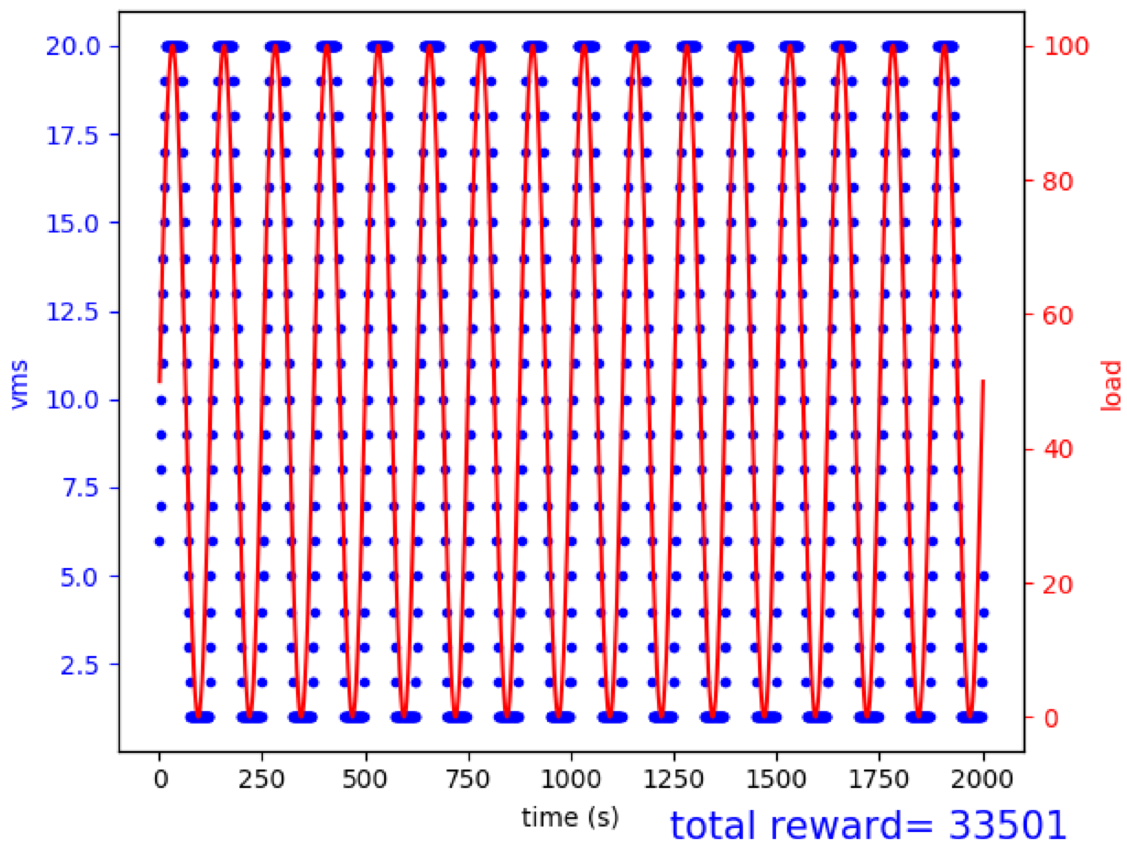

Figure 8.19: Double DQN

We see that a high or a low learning rate value damages our agent's decisions both in Simple DQN and Double DQN. This is the first time that a parameter messes up our Double DQN outputs, so that it means that the leraning rate is one of the most crucial parameters when it comes to neural networks optimizations.

So the best parametrization given the simulations is to pick a value of about 360 experience batch size for our memory buffer, a value of about 0.00025 for our learning rate and a value of annealing steps equal to our training steps every time, so that our e value decreases linearly to our input. One can also play with the number of neurons our networks has, or the number of layers. Also with the sequence that the target network update occurs in both Full DQN and Double DQN. We did, but we demonstrate here some examples, just to see the procedure. By playing with the networks and algorithm parameters you can optimize your solutions further and further.

8.2 Comparison between different Algorithms in a simple scenario

We will attempt to compare and evaluate the overall performance of the algorithms discussed in this work, namely Simple DQN, Full DQN, and Double DQN and algorithms used formerly by tiramola such as MDP, Q-learning, MDDPT, QDT . For that purpose we will use a scenario from the field of cloud computing. We use the simple cluster used for the parametrization of the algorithms in section 8.1.

8.2.1 Cluster setup

- The cluster size can vary between 1 and 20 virtual machines
- The available actions to the agent in each step are to increase the size of the cluster by one, decrease the size of the cluster by one, or do nothing.
- The incoming load is a sinusoidal function of time: $load(t) : 50 + 50\sin(\frac{2\pi t}{250})$
- The percentage of incoming requests that are reads is a sinusoidal function of time with a different period: $r(t) : 0.75 + 0.25\sin(\frac{2\pi t}{340})$
- If $vms(t)$ is the number of virtual machines currently in the cluster, the capacity of the cluster is given by: $capacity(t) = 10vms(t)r(t)$
- The reward for each action depends on the state of the cluster after executing the action and is given by: $R_t = min(capacity(t+1), load(t+1)) - 3vms(t+1)$.
- Training steps $\in [2000, 5000, 10000, 20000]$
- Evaluation steps: 2000
- max error: 10^{-6}
- Learning rate: 0.00025
- batch size: 360

In the first figure 8.20 we see the results of using MDP, Q-learning, MDDPT and QDT algorithms in our cluster and each rewards. One limitation is that we have to define a number of states in these algorithms in opposed to our approach where we don't have "states". Neural Networks work by finding

correlations between agent's inputs. The more the number of inputs, the better, meaning that we do not have any limitations.

In the second figure 8.21 we see the results of using our three approaches in our cluster and observing the rewards. It is pretty obvious that Full DQN and Double DQN quickly adapt to the problem, find correlation between input parameters, manage to find which input parameters to ignore and obtain large rewards by choosing optimal decisions. Simple DQN on the other hand, shows some deficient compared to the above two approaches, but it is not such a bud solutions as a stand alone approach, compared to simple MDPs or Q-learning.

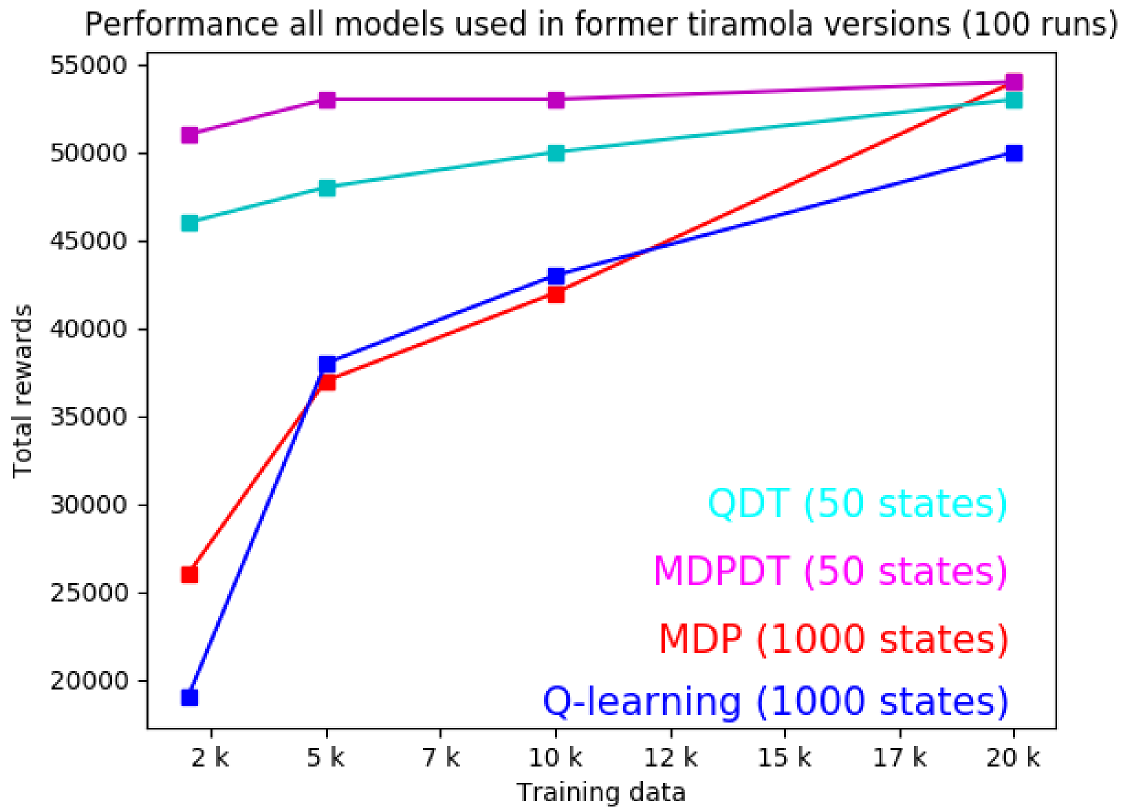

Figure 8.20

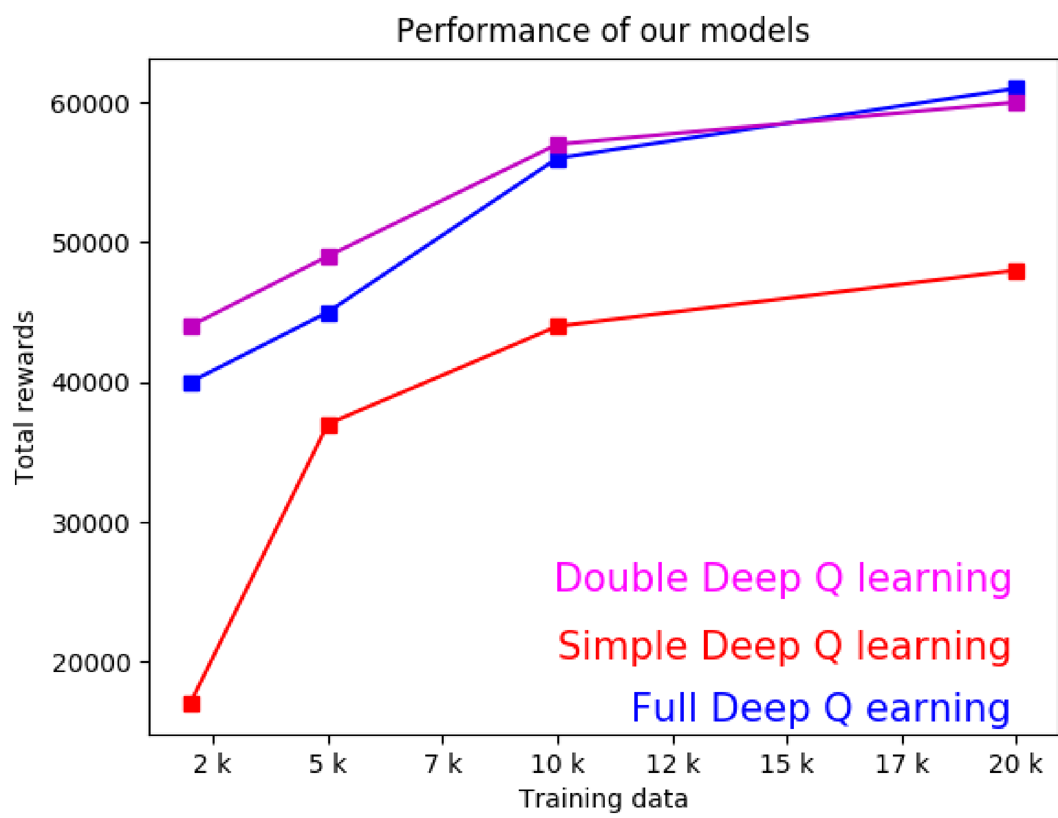

Figure 8.21

In figure 8.22 we compare our best player Double DQN with MDDPT and in figure 8.23 with QDT. Our approach give better results than the approaches based on decision trees models. We must take under consideration here that the simulation test that we are using, were formally used to better test these models, so the fact that our approach surpasses theirs, in these tests proves a better adaptation by our side.

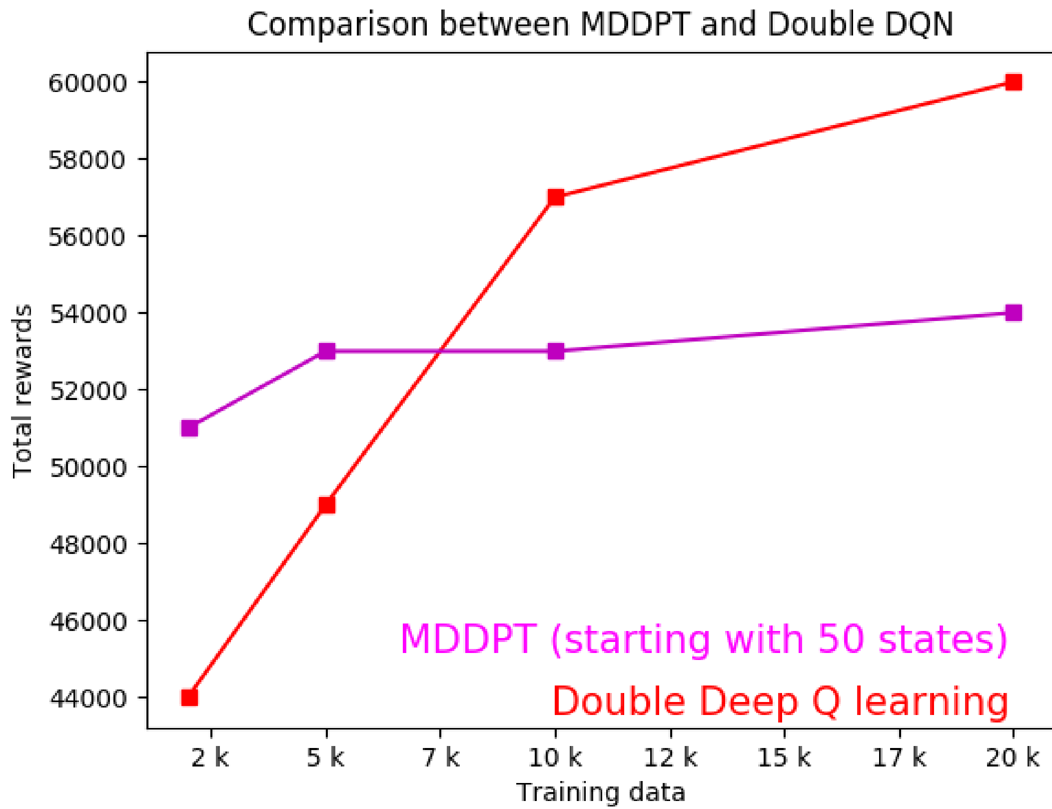

Figure 8.22

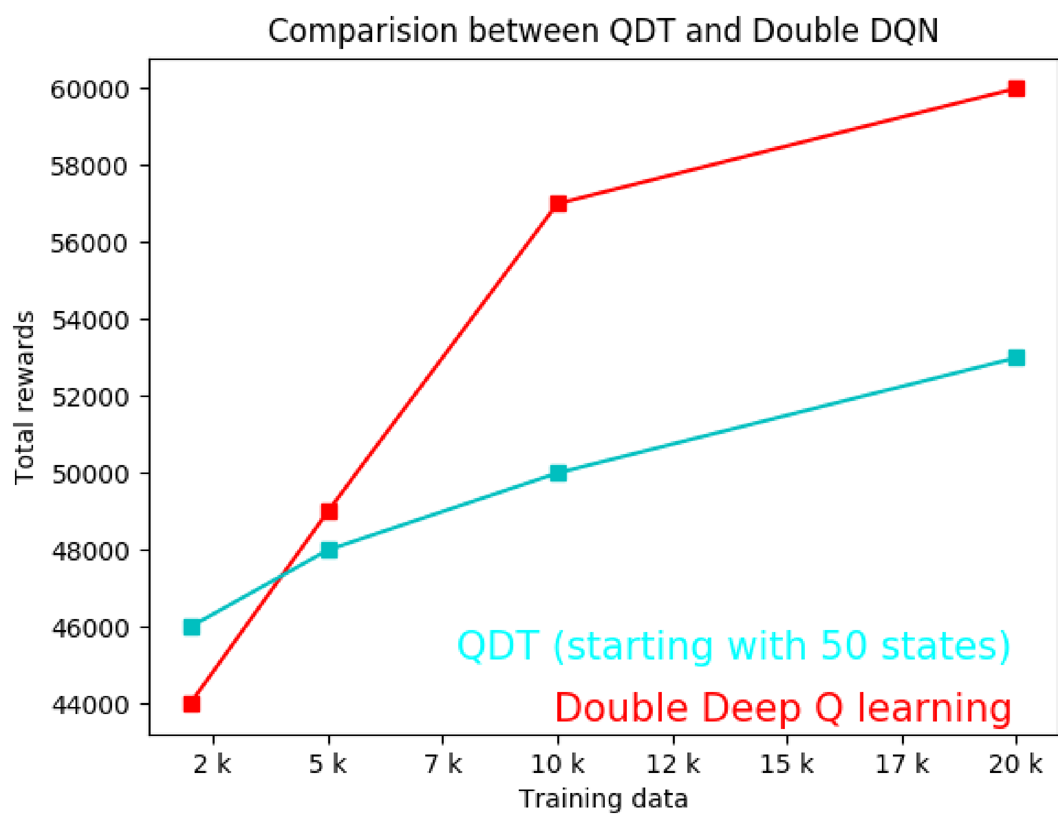

Figure 8.23

8.3 Comparison between different Algorithms in a complex senario

Here we are testing our agent's performance for each of our algorithms in a more complex cloud-computing based scenario. An example of our Double DQN agent handling the Difficult requirements and challenges of the complex scenario we can see in figure 8.24

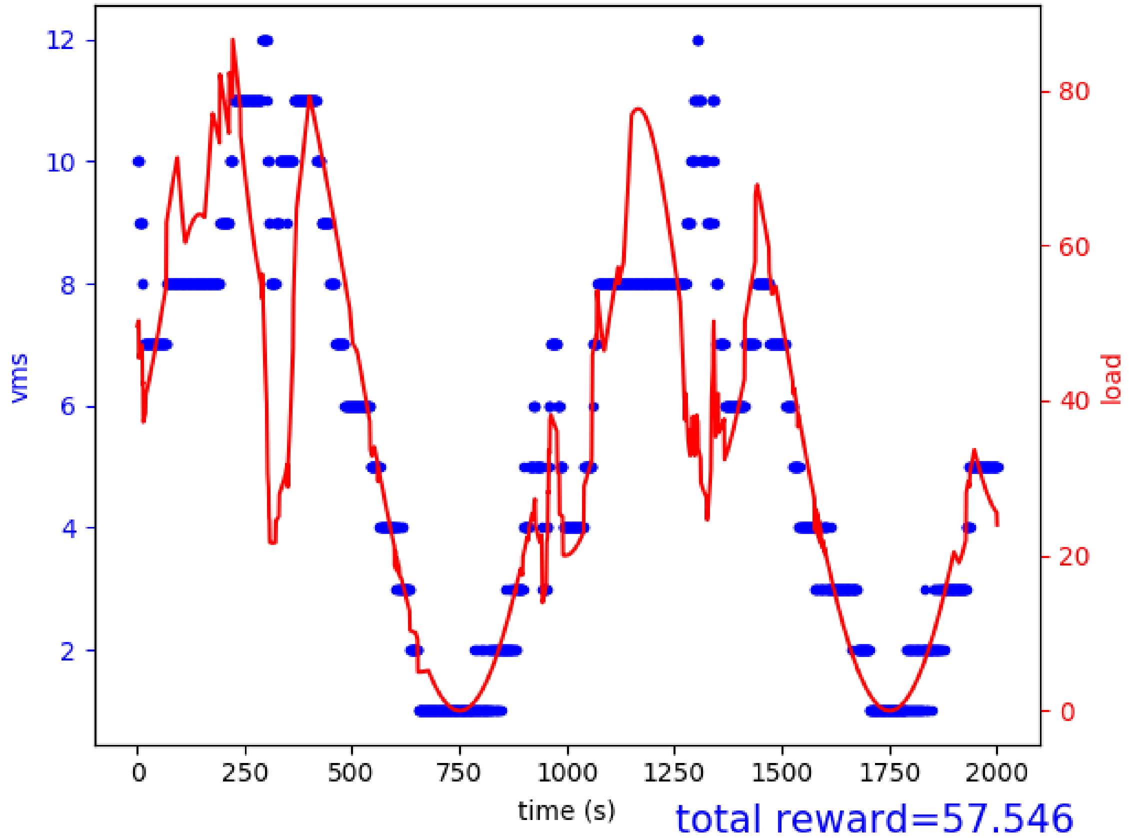

Figure 8.24: Double DQN in the complex scenario

8.3.1 Cluster setup

- The cluster size can vary between 1 and 20 virtual machines
- The available actions to the agent in each step are to increase the size of the cluster by one, decrease the size of the cluster by one, or do nothing.
- The incoming load is a sinusoidal function of time: $load(t) : 50 + 50\sin(\frac{2\pi t}{250})$
- The percentage of incoming requests that are reads is a sinusoidal function of time with a different period: $r(t) : 0.7 + 0.3\sin(\frac{2\pi t}{340})$
- I/O operations per second: $IO(t) : 0.6 + 0.4\sin(\frac{2\pi t}{195})$
- I/O penalty:

$$IO_pen(t) = \begin{cases} 0 \quad if 0.7 > IO(t) \\ IO(t) - 0,7 \quad if 0.7 < IO(t) < 0.9 \\ 0.2 \quad if IO(t) > 0.9 \end{cases} \tag{8.1}$$

- If $vms(t)$ is the number of virtual machines currently in the cluster, the capacity of the cluster is given by: $capacity(t) = 10vms(t)r(t)$
- The reward for each action depends on the state of the cluster after executing the action and is given by: $R_t = min(capacity(t+1), load(t+1)) - 3vms(t+1)$.
- Training steps $\in [2000, 5000, 10000, 20000, 500000]$
- Evaluation steps: 2000
- max error: 10^{-6}
- Learning rate: 0.00025
- batch size: 360

To increase the difficulty of this scenario, we have increased the effect of the types of the queries to the capacity of the cluster, and have also added one more parameter that affects the behavior of the system in a non-linear manner, namely the I/O operations per second. This parameter takes values between 0.2 and 1.0, but only affects the performance of the cluster if its value is higher that 0.7 by adding a penalty to the performance of each VM. Just like in the simple scenario we included 6 additional random input parameters. Three of them followed a uniform distribution within [0, 1], and another three took integer values within [0, 9] with equal probability.

In the first figure 8.25 we see the results of using MDP, Q-learning, MDDPT and QDT algorithms in our cluster and each rewards.

In the second figure 8.26 we see the results of using our three approaches in our cluster and observing the rewards.

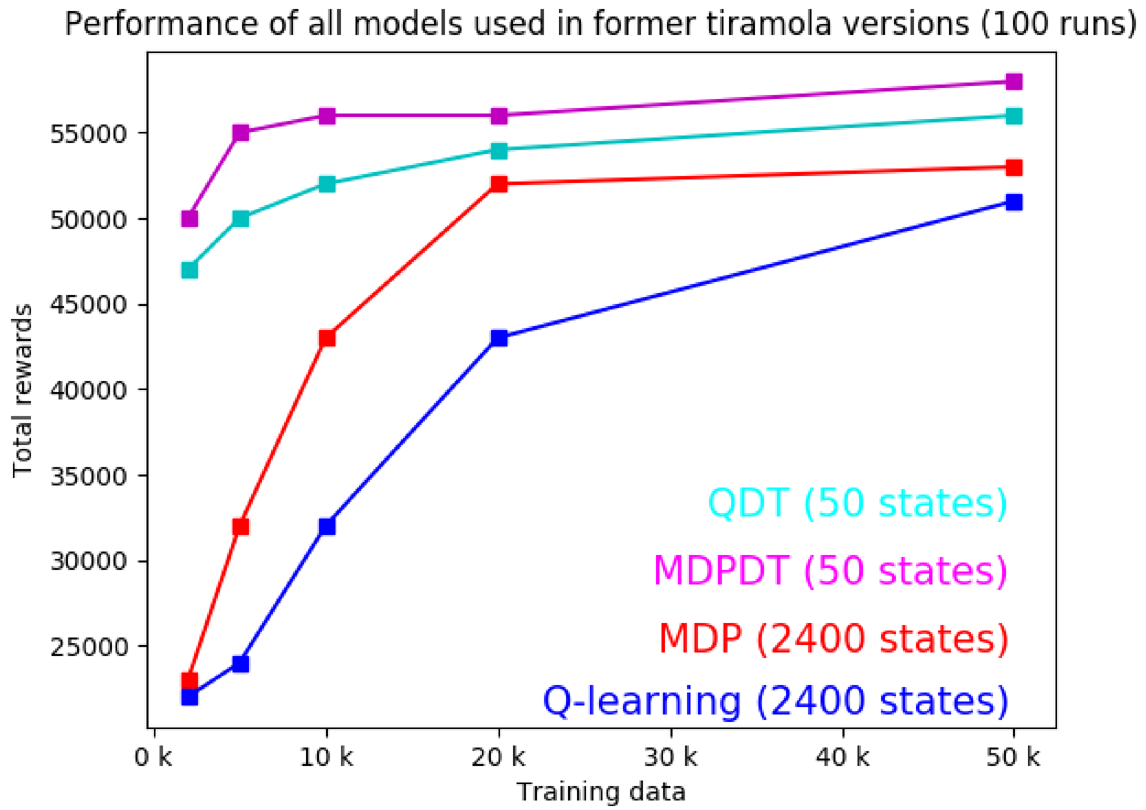

Figure 8.25

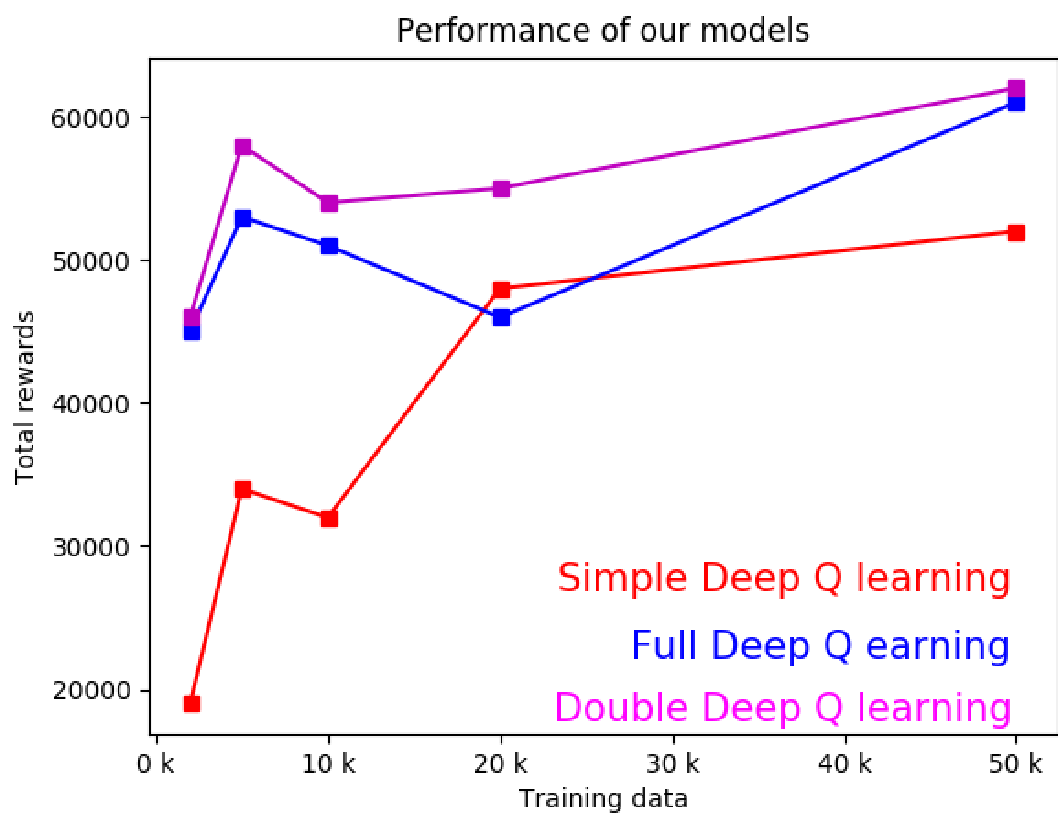

Figure 8.26

In figure 8.27 we compare our bet player Double DQN with MDDPT and in figure 8.28 with QDT.

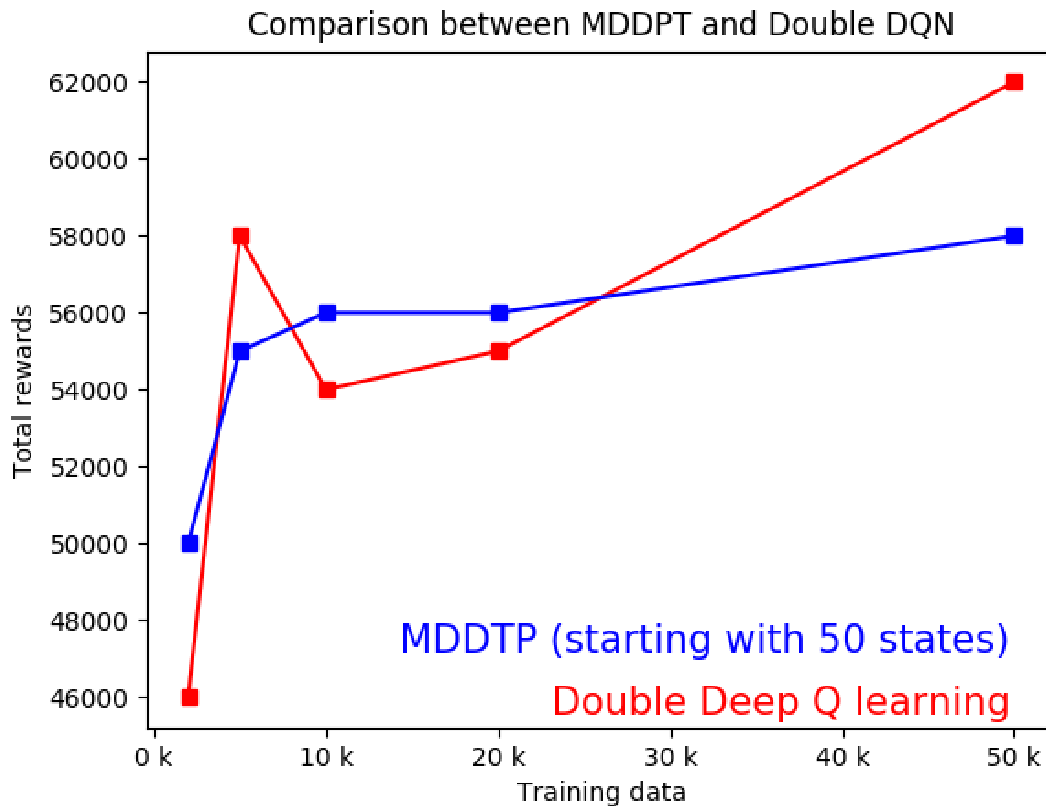

Figure 8.27

8.4 Conlcusions

As we see Deep Rl carrys a lot of advantages in opposed to former algorithms used on Tiramola. We will mention some of them, observed by the experimental results we have just provided you.

Adaptation

The adaptation of Deep Rl model is robust and and rapid, regardless the size of the training set. As we see in former algorithmic models used by Tiramola, the more complex models needed larger set of datas before fully adapting to the problem and finding the optimal solution, whereas more simple algorithmic approaches such as MDP, adapted more quickly but did not manage to find the optimal decision the most times and obtain a larger reward.

Clustering

Deep Rl algorithms manage to cluster the data and find which input data really matters to the outcome of the algorithm and which not. Deep Rl algorithms can do that at any set of data, regardless how big this is. They do not need any number of states provided and as much more data we have to feed our agents, including the number of input parameters, the better.

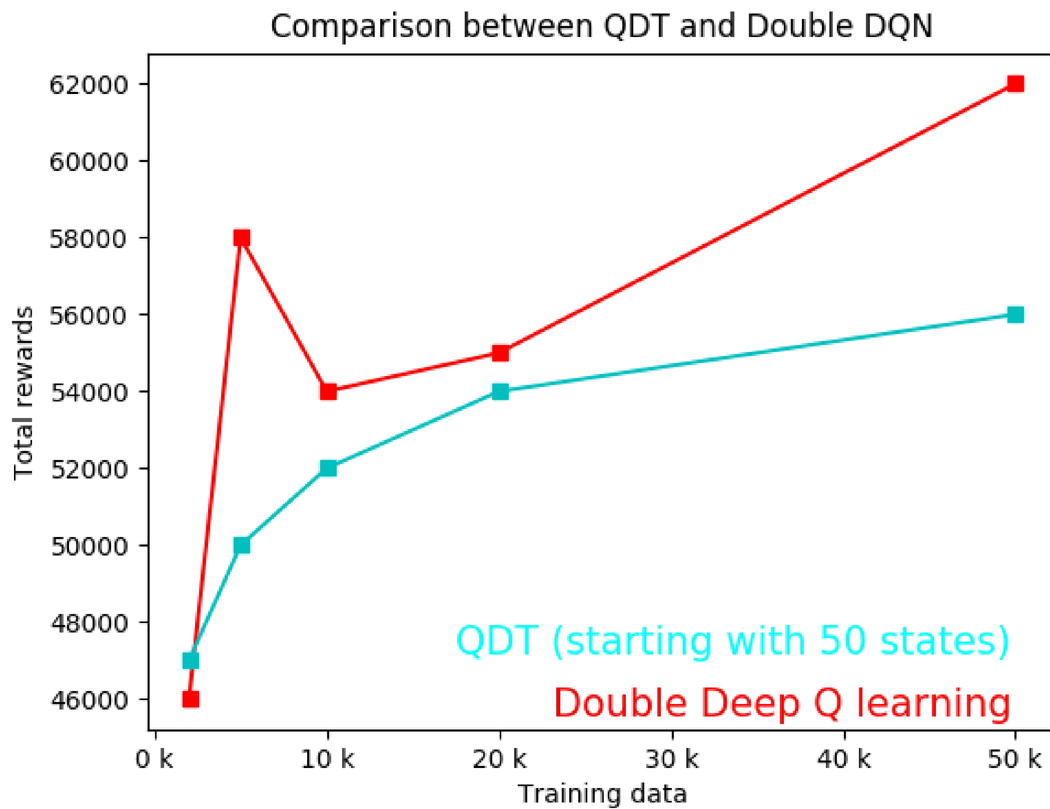

Figure 8.28

Space complexity

The space complexity of the Deep Rl approach is what gives it a large edge against other approaches. Thus because all other RL approaches need computer space for storaging past experiences. The attempt of the former version of Tiramola, using the decision trees approach was to find a way to manage the data without using to much computer space and at the other hand extract as much information as possible. Deep Rl uses Neural Networks which gives a tremendous advantage in the aspect, as NN do not have to use any of computer space to store information. Every information needed is simply stored in the network's weighs. That gives as the elasticity to handle as much data, as much experience, as much information as we want, cause we do not have to worry about storage space our about handling the information extracted from this tons of data. One of the main advantages of our approach is that it can scale to really large environments and datasets and not struggle, but become more efficient.

Total Rewards

Even if all the above sections were handled sufficiently by our agent, still that would be meaningless if it couldn't get sizeable enough rewards. But as we can see in our examples our Full DQN agent and our Double DQN agent manage to get large enough rewards, even better from all the previous versions of Tiramola.

At the end of the game Mariana Trench surpasses its ancestors in terms of sufficiency, efficiency, adaptation, storage economy and scaling.

Chapter 9

Experimental Results and Conclusion

9.1 Experimental results

In this section we are going to present you the experiments we did with Mariana Trench in real life environments. We are using Okeanos service as our cloud infrastructure. We have built a Cassandra cluster upon okeanos, which we trigger with different workloads produced by ycsn service. We collect metrics with ganglia XML backend and use our Mariana Trench agent to determine the best decision on every step of the procedure, after we have trained it. The programming environment that we use is Anaconda [2] and the library that provide us the neural network and training tools is google's Tensorflow [1].

Let us present the basic factors of our implementation.

9.1.1 Infrastructure

We have a cluster containing 16 VMs on okeanos [12] cloud infrastructure. We produce workloads with ycsb [40] framework. Our workloads are sinusoidal reads and writes on our cassandra database. The load of those reads and writes follows a sinusoidal distribution. The percentage of reads or writes requests is random. We use every single node of our cluster as a receiver of our requests as in Cassandra every node can serve requests and there is no central node as in other NOsql databases such as Hbase. We send the request using kamaki API [10], by one single computer. The computer splits the workloads in as many pieces as the cluster number of nodes. It then generates as many threads as the number of splits and each thread sends requests to each one of the cluster VMs. Every 10 seconds we obtain metrics using telnet to contact with ganlia's [9] XML API. These metrics represents the cluster current state and consist of the following parameters:

- The number of active VMs on the cluster.
- The latency of the cluster.
- The throughput of the cluster.
- The amount of cached memory on the cluster.
- The current number of operations/requests served by the cluster at the current point.
- The number of operations/requests served by the cluster on the last state. We use this information to determine if the size of operations is currently more possible to be on an increasing or decreasing slope.
- The amount of free memory on the cluster.

- The percentage of cluster CPU idle.
- The cluster amount of buffered memory.
- The cluster amount of available memory.
- The cluster amount of buffered memory.
- The cluster amount of shared memory.
- The cluster amount of free disc space.
- The amount of bytes in, that each node of the cluster experience in the current state
- The amount of bytes out, that each node of the cluster experience in the current state

Our reward function is

$$Reward = 0.01 * throughput - 0.00001 * latency - 2 * VMs \tag{9.1}$$

as we want to keep the cost of our cluster low (VMs) while achieving the highest throughput and the lowest latency we can. In each step the agent takes a decision, collects new metrics, calculates a new reward function makes a new decision and takes a new action.

9.1.2 Training

We let our Double Deep Q learning, Mariana Trench agent train for 20.000 steps. We set the annealing steps factor at 2000. The annealing steps arrange the amount of steps taken before we decrease the e factor for our e-greedy policy algorithm. Thus means that for the first 2000 steps the algorithm takes actions completely randomly in order to achieve better exploration of the environment and only at the very end of the experiment all of our agents actions are based on our networks best decision.

We use 620 pretrain steps, thus meaning the number of steps our algorithm makes completely random moves in the environment in order to fill our memory-batch. As we have calculated at the calibration of our system in the previous chapter we set the following factors as so:

- Our batch size is set at 360 experiences
- Our learning rate is set at 0.00025.

9.1.3 Testing

After the training part, we test our Marianna Trench agent for 2000 execution steps. We test it for our three different approaches, Simple DQN, Full DQN and Double DQN. We observe that our agent quickly converts its behaviour to the optimal, obtaining large rewards. When Simple DQN is the case, our agent, spends a little more time until it finds the optimal solution. We see the results in images 9.1, 9.2, 9.3

Then we test our best agent, Double DQN with a biggest dataset for training, containing 60000 different states. We see the result in image 9.4

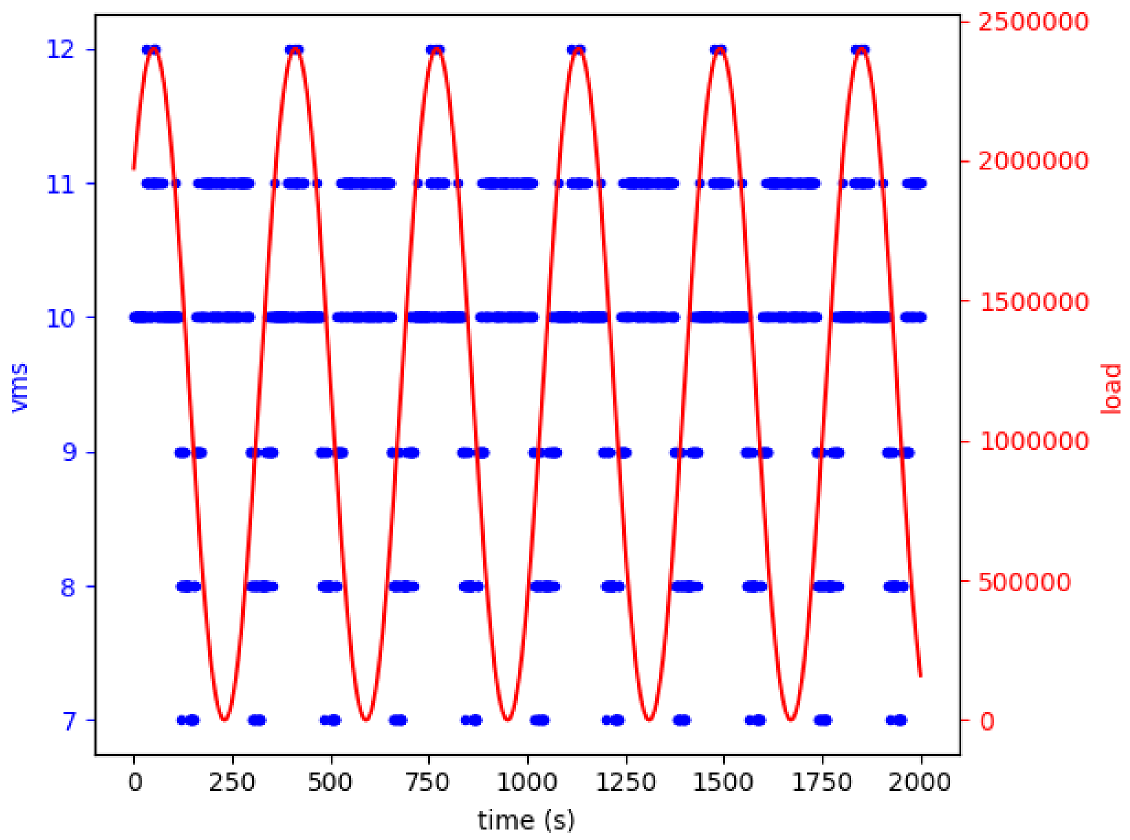

Figure 9.1: Performance of our agent when using the Simple Deep Q learning algorithm

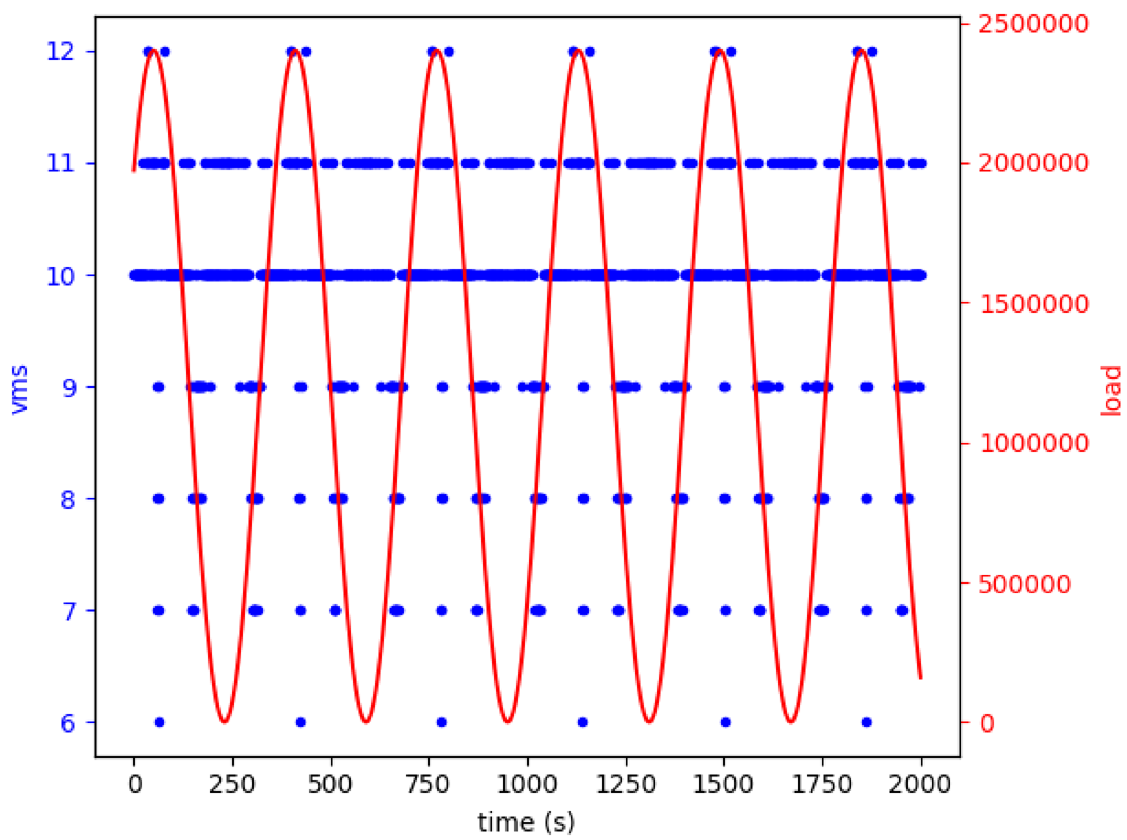

Figure 9.2: Performance of our agent when using the Full Deep Q learning algorithm

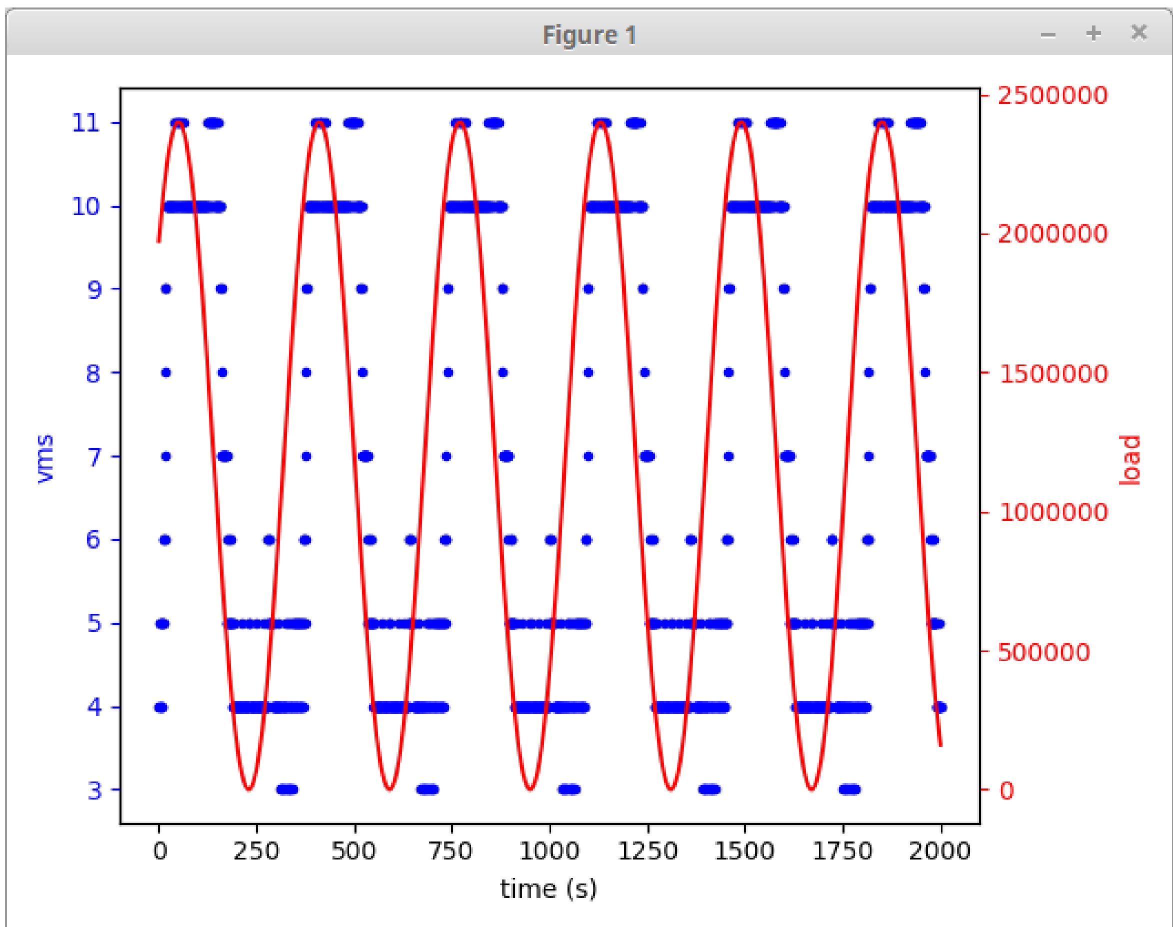

Figure 9.3: Performance of our agent when using the Double Deep Q learning algorithm

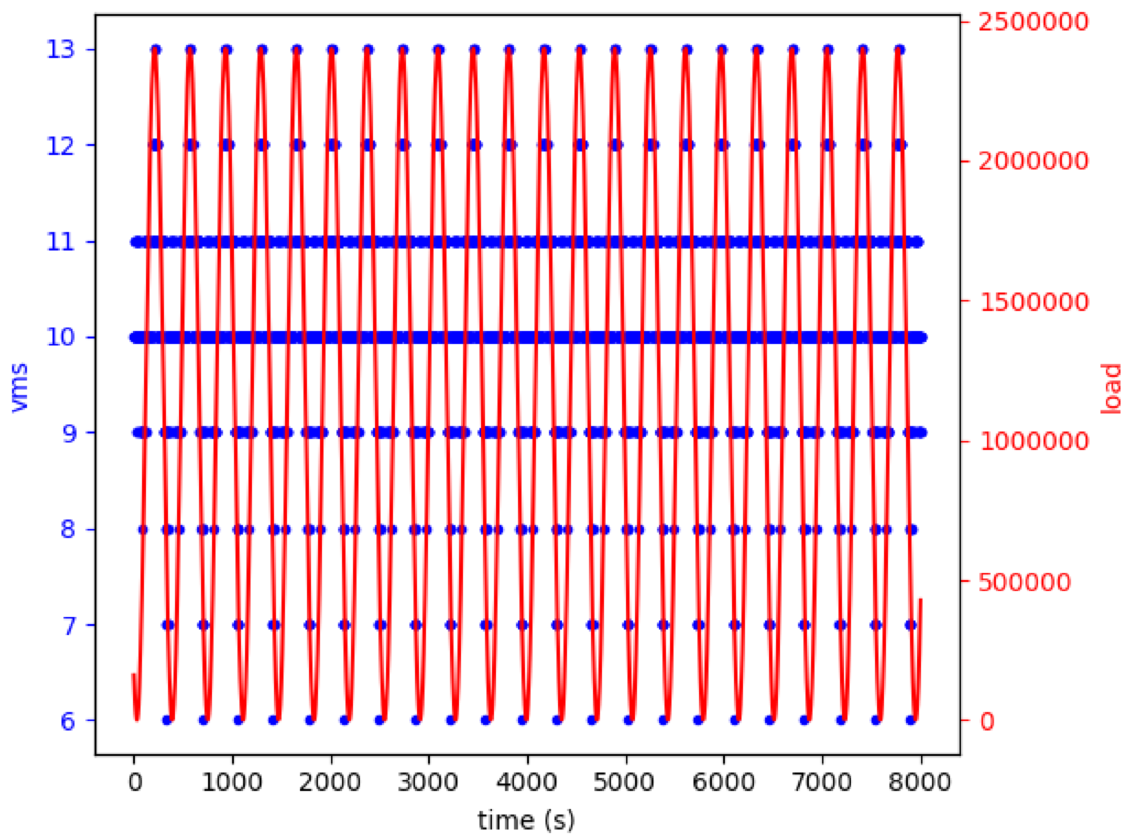

Figure 9.4: Performance of our agent when using the Double Deep Q learning algorithm with large dataset

9.1.4 Conclusions of experimental Results

The conclusions we get from the experimental results does not differ from the conclusion we got from our simulation in the previous chapter 8.

Adaptation

Mariana trench approach is robust and and rapid, regardless the size of the training set. If we except the Simple Deep RL approach, our agent converges to the optimal solution with small datasets sizes.

Clustering

Mariana Trench manages to cluster the data and find which input data really matters to the outcome of the algorithm and which not. In the experiments our input consist of 15 different parameters and yet our agent manages to determine which of them are important and which are not after experiencing a small amount of training steps.

Space complexity

Mariana's Trench Deep Rl approach because of using Neural Networks gives us a tremendous advantage in the aspect of space complexity, as NN do not have to use any of computer space to store information. Every information needed is simply stored in the network's weighs. That gives as the elasticity to handle as much data, as much experience, as much information as we want, cause we do not have to worry about storage space. One of the main advantages of our approach is that it can scale to really large environments and datasets and not struggle, but (and this is our main advantage) become more and more efficient as we have mentioned before.

9.2 Conclusions

In this work we presented a Deep Reinforcement Learning agent called Mariana Trench for cloud elasticity problems by combining cutting edge algorithmic techniques in both deep learning and cloud resource management areas. Our results show that our agent significantly outperforms all previous approaches used in previous versions of the Tiramola architecture. In our experimental settings the input consisted of 13 input parameters, however our agent can perform efficiently with much larger more complex environments.

To summarise, the advantages of our approach are the following:

- Mariana Trench can learn and perform tasks in large environments where each state depends on multiple parameters. In addition, Mariana Trench does not need any space partitioning or data clustering and does not experience problems when dealing with large input data. On the contrary the biggest the data pool, the best for Mariana Trench.

- Mariana Trench shows that we can efficiently use DRL outside of image problems where it is usually used, and achieves excellent results even in an environment where the state is controlled by a small number of parameters. Our agent behaved optimally in a cloud environment where it provides each user with the best resource previsioning based on his predefined needs.

- Our approach successfully updated Tiramola without using more computer space (memory) or resources than its previous versions. Its decision module does not lack speed although one of its versions (Simple Deep Rl) does take a little more train time before converging to the best decisions.

- Finally, Marianna Trench includes a lot of our network factors than can be recalculated and achieve better results for different environments. Some of these factors include the number of layers/neurons or the number of steps that pass before doing a target network update.

It is our belief that one now can see that Deep RL if carefully used (as it has many different versions that provide us with different benefits) can solve problems at every aspect of science. We showed that Deep Rl agent can perform optimally in an environment that is built upon a human user needs and manage to get the best out of it. As we shown at the introduction [34] Deep Rl has been shown to be successful in achieving great results in the area of physics (although in this area it is still only tested on image vector inputs). Many more aspects such as mathematical problems, physic problems, social or economy problems are worthy of giving a Deep Rl agent a shot on them. Perhaps we are in front of a new area, where machines can truly replace human minds in every field that does not require consciousness. It is certain that we have not yet scratched the surface of this new world, as it has been introduced to us 4 years ago by Deepmind. Let’s hope that thesis like this are helping on the better understanding of the Deep RL potentials and not helping in future birth of skynet.

9.3 Future work

In this diploma thesis we explored some of the aspects of Deep RL in an environment of cloud computing but there are a lot of things that we did not have the opportunity to tackle with. Although we believe that they are worth dealing with and being investigated in the future. Some of them are:

Recalibrating of the current network and Mariana Trench parameters

As mentioned above there are a lot of parameters in our current version of Mariana Trench that can be revised and maybe achieve better results. Such parameters, are the number of steps that we do the update of our second network in both Full DQN and Double DQN. The numbers of the layers or the neurons that we use in our network could also be tuned better. Someone might try to use convolutional neural networks instead of our fully connected network. Getting a lot more input parameters or testing Mariana Trench in a real life environment with a pool of real life users-testers would be a great idea too, to check how Mariana Trench scales and if it can sufficiently give the results it promises to give, in real life environments.

Using Recurrent Neural Networks

For those who are not familiar with Recurrent Neural Networks [26], they are neural networks known for their ability to deal with incomplete data. Meaning environments where the agent sometimes can only partially observe. Deep Rl can be combined with recurrent networks and achieve great results. As Mariana Trench deals with a cloud environment where the metrics might sometimes contain noise, or be not so accurate, or even have been failed to be collected, Deep RL with recurrent neural networks is possible to give better results in the long run than the current versions. An investigation on the subject would be sourly really interesting and revealing.

Using Dueling Deep Q learning

Dueling Deep Q learning [41] deals with Deep RL using a different approach than the q learning bellman's function. Let us first remember bellmans' equation in simple terms:

$$Q(s, a) = V(s) + A(a) \tag{9.2}$$

thus mean that our Q value is a combination of the value function V(s), which simply says how good it is to be in any given state and the advantage function A(a), which tells what our agent would earn by taking an action a at this particular point. In our approach out target networks computes the Q values as one. In Dueling deep q learning the agent's goal is to separately compute the advantage and value functions, and then combine them back into a single Q-function only at the final layer. Why doing that? In some points of our agent's life in our environment the best action might as well be to take no action. To stay still in its current state. In that case the advantage function is not needed and worthy of calculating. In cloud computing suppose we have a user whose preferred policy is to almost never change the number of VMs at his cluster, meaning the state of the cluster, except in exceptional cases. For this particular user it would be better to calculate the rewards he gets separately for being in this current state and for hypothetically moving to another state.

Experimenting with different approaches that examine the exploration-exploitation duel

In our approach we use **e-greedy** policy. In this approach the agent, when in training, takes at first random actions and gradually takes more beneficial than random actions until it gets to a point that it only takes the best action given by the decision module of the network at each step.
One can test different approaches, such as **boltzmann** approach, in which the agent instead of always taking the optimal action, or taking a random action, chooses an action with weighted probabilities. This is accomplished by using a softmax over the networks estimates of value for each action. The advantage over the e-greedy policy would be that information about likely value of the other actions

of our agent can also be taken into consideration.
One can also tests **bayesian** approaches. A bayeasian nueral network is known for maintaining a probability distribution over possible weights in oppose to traditional NN which are deterministic. In a RL setting, the distribution over weight values gives us the ability to obtain distributions over actions as well. Yarin Yal's phd thesis [8] researches the subject in depth, with some really interesting results.

Bibliography

[1] JJ Allaire, Dirk Eddelbuettel, Nick Golding, and Yuan Tang. tensorflow: R Interface to TensorFlow, 2016.

[2] Anaconda. https://conda.io/docs/user-guide/tasks/manage-environments.html.

[3] Antikithira. https://en.wikipedia.org/wiki/Antikythera_mechanism.

[4] Armed bandit. https://en.wikipedia.org/wiki/Multi-armed_bandit.

[5] Leon Bottou. Gradient descent.

[6] Chris J. Maddison Arthur Guez Laurent Sifre George van den Driessche Julian Schrittwieser Ioannis Antonoglou Veda Panneershelvam Marc Lanctot Sander Dieleman Dominik Grewe John Nham Nal Kalchbrenner Ilya Sutskever Timothy Lillicrap Madeleine Leach Koray Kavukcuoglu Thore Graepel Demis Hassabis David Silver, Aja Huang. Mastering the game of go with deep neural networks and tree search.

[7] Decision-trees. https://en.wikipedia.org/wiki/Decision_tree_learning.

[8] Yarin Gal. Uncertainty in deep learning.

[9] Ganglia monitoring system. http://ganglia.sourceforge.net/.

[10] Kamaki api. https://www.synnefo.org/docs/kamaki/latest/.

[11] Vangelis Floros Konstantinos Tsakalozos, Mema Roussopoulos. Nefeli.

[12] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. okeanos: Building a cloud, cluster by cluster. IEEE Internet Computing, 17(3):67–71, May 2013.

[13] Dimitrios Tsoumakos & Ioannis Konstantinou & Christina Boumpouka & Spyros Sioutas & Nectarios Koziris. Automated, elastic resource provisioning for nosql clusters using tiramola.

[14] Konstantinos Lolos & Ioannis Konstantinou & Verena Kantere & Nectarios Koziris. Elastic resource management with adaptive state space partitioning of markov decision processes.

[15] Timothy P. Lillicrap, Jonathan J. Hunt, , Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning.

[16] Linear regression. https://onlinecourses.science.psu.edu/stat501/node/251.

[17] Konstantinos Lolos, Ioannis Konstantinou, Verena Kantere, and Nectarios Koziris. Elastic resource management with adaptive state space partitioning of markov decision processes. CoRR, abs/1702.02978, 2017.

[18] Mariana trench. https://en.wikipedia.org/wiki/Mariana_Trench.

[19] Markov chains. https://en.wikipedia.org/wiki/Markov_chain.

[20] WARREN S. MCCULLOCH and WALTER PITTS. A logical calculus of the ideas immanent in nervous activityg. BULLETIN OF MATHEMATICAL BIOPHYSICS, 1943.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. CoRR, abs/1312.5602, 2013.

[22] Mpds. https://en.wikipedia.org/wiki/Markov_decision_process.

[23] Marianna trench. https://en.wikipedia.org/wiki/NoSQL.

[24] Okeanos. https://okeanos.grnet.gr/home/.

[25] Pascal's calculator. https://en.wikipedia.org/wiki/Pascal%27s_calculator.

[26] Reccurent neural networks. https://en.wikipedia.org/wiki/Recurrent_neural_network.

[27] Reinforecment learning. https://en.wikipedia.org/wiki/Reinforcement_learning.

[28] relu function. https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[29] Rmsprop. https://www.tensorflow.org/api_docs/python/tf/train/RMSPropOptimizer.

[30] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation.

[31] Richard S. Sutton. Learning to predict by the methods of temporal differences. Machine Learning, 3(1):9–44, 1988.

[32] Talking heads. http://www.haskins.yale.edu/featured/heads/simulacra.html.

[33] Gerald Tesauro. TD-Gammon: A Self-Teaching Backgammon Program, pages 267–285. Springer US, Boston, MA, 1995.

[34] Alexander Pritzel Nicolas Heess Tom Erez Yuval Tassa David Silver Daan Wierstra Timothy P. Lillicrap, Jonathan J. Hunt. Continuous control with deep reinforcement learning, year = 2016, url = https://arxiv.org/abs/1509.02971,.

[35] Turing test. https://en.wikipedia.org/wiki/Turing_test.

[36] The turk. https://en.wikipedia.org/wiki/The_Turk.

[37] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning.

[38] Walking lion. http://dangerousminds.net/comments/leonardo_da_vincis_incredible_mechanical_lion.

[39] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. Machine Learning, 8(3):279–292, 1992.

[40] Ycsb yahoo cloud system benchmark. https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/.

[41] Matteo Hessel Hado van Hasselt Marc Lanctot Nando de Freitas Ziyu Wang, Tom Schaul. Dueling network architectures for deep reinforcement learning.