National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Computing Systems Laboratory

Horizontal Scalability in cloud environments orchestrated by Kubernetes

Diploma Thesis

of

NEFELI PANAGIOTA TZAVARA

Supervisor: Nectarios Koziris
Professor, NTUA

Athens, November 2023

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Computing Systems Laboratory

Horizontal Scalability in cloud environments orchestrated by Kubernetes

DIPLOMA THESIS

of

NEFELI PANAGIOTA TZAVARA

Supervisor: Nectarios Koziris
Professor, NTUA

Approved by the examination committee on 2nd of November 2023.

(Signature)	*(Signature)*	*(Signature)*
....	....	....
Nectarios Koziris	Dimitrios Tsoumakos	Ioannis Konstantinou
Professor, NTUA	Associate Professor, NTUA	Assistant Professor, UTh

Athens, November 2023

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Computing Systems Laboratory

(Signature)

. . . .
Nefeli Panagiota Tzavara

Electrical and Computer
Engineer, M.Eng. NTUA

Περίληψη

Μελετώντας σε βάθος τα χαρακτηριστικά του 21ου αιώνα, μπορούμε να παρατηρήσουμε το γεγονός ότι ο όγκος των δεδομένων που παράγονται αυξάνεται συνεχώς, περισσότερο από ποτέ. Οι πρόσφατες τεχνολογικές εξελίξεις στην επιστήμη της πληροφορικής και στις υπηρεσίες της έχουν καταστήσει πλέον δυνατό τον εξ αποστάσεως υπολογισμό μεγάλων δεδομένων σε κατανεμημένα συστήματα και/ή σε υπηρεσίες νέφους. Ο εξ αποστάσεως υπολογισμός, σε αυτά τα πλαίσια, καθίσταται δυνατός με τη χρήση microservices που μπορούν να εκτελούνται ανεξάρτητα σε δομές περιεκτών (containers), οι οποίες έχουν επίσης γίνει πολύ δημοφιλείς. Η οργάνωση ή αλλιώς ¨ενορχήστρωση' των περιεκτών γίνεται από τους λεγόμενους ενορχηστρωτές, με πιο δημοφιλή το Kubernetes , το οποίο είναι λογισμικό ανοικτού κώδικα. Το Kubernetes είναι υπεύθυνο για τον χρονοπρογραμματισμό των pods σε διαφορετικούς κόμβους μιας συστάδας (cluster) και επίσης για τη λήψη αποφάσεων σχετικά με την κλίμακα του cluster. Τα pods είναι μονάδες που ενσωματώνουν συλλογές containers και εκτελούνται μέσα σε κόμβους. Η κλίμακα της συστάδας παίζει καθοριστικό ρόλο, καθώς οι απαιτήσεις των υπηρεσιών μεταβάλλονται και η συστάδα πρέπει να είναι σε θέση να αντέξει τις διαφορετικές συνθήκες χρήσης (λ.χ. δημιουργία πολλών pods από αυξημένο φόρτο κίνησης σε έναν server). Έτσι, η αποτελεσματική κλιμάκωση έγκειται στην αποτελεσματική και έγκαιρη πρόβλεψη σχετικά με το πόσοι και ποιοι πόροι θα διατεθούν σε ορισμένες διεργασίες, ώστε να αποφεύγεται η αποτυχία του συστήματος και να ελαχιστοποιείται η αδράνεια των πόρων. Στην παρούσα διπλωματική, ξεκινάμε με τη μελέτη της αρχιτεκτονικής του Kubernetes και των διαφόρων εννοιών κλιμάκωσης με έμφαση στην οριζόντια κλιμάκωση, δηλαδή την κλιμάκωση όπου τροποποιείται ο αριθμός των κόμβων συστάδας. Για τις δοκιμές μας, μελετάμε τη χρήση της εξεργαστικής ισχύος CPU σε μια συστάδα που ενορχηστρώνεται από το Kubernetes. Συγκεκριμένα, μελετάμε ανώνυμα δεδομένα που συλλέχθηκαν επί 15 ημέρες από έναν διακομιστή παραγωγής στο ινστιτούτο CERN και περιέχουν δεδομένα χρήσης της CPU με την πάροδο του χρόνου. Πραγματοποιούμε διερευνητική ανάλυση δεδομένων για την περαιτέρω κατανόηση των δεδομένων αυτών και έπειτα, προχωράμε στην ανάπτυξη μοντέλων πρόβλεψης για την πρόβλεψη της χρήσης της CPU . Τέλος, αναπτύσσουμε έναν πράκτορα ενισχυτικής μάθησης με πολιτική ελαχιστοποίησης της χρήσης CPU με βάση την οριζόντια κλιμάκωση της συστάδας. Απώτερος στόχος μας είναι να προτείνουμε έναν προορατικό custom resource allocator που μπορεί να ενσωματωθεί σε μια συστάδα του Kubernetes , έτσι ώστε οι αποφάσεις κλιμάκωσης να βασίζονται στην πρόβλεψη της χρήσης της CPU και συνεπώς στην χρήση του απαραίτητου αριθμού κόμβων.

Λέξεις Κλειδιά
Βαθιά Μάθηση, Kubernetes, Deep Q Learning, Reinforcement Learning, LSTM, Οριζόντια κλιμακωσιμότητα, πρόβλεψη χρονοσειράς

Abstract

Taking a look into the 21st century's distinctive features, it is prevalent that the volume of data produced is constantly rising, more than ever. Recent technological innovations in computer science and services have now made possible the remote computation of large data in distributed systems and/or cloud services. The remote computation in these frameworks is made possible by the use of microservices that can run independently in containers who have also become very popular. The organising, or more commonly, the "orchestration" of containers is done by the so called orchestrators, the most popular one being Kubernetes, which is an open-source software. Kubernetes is responsible for scheduling pods in the different nodes of a cluster and also for making decisions about the scale of the cluster. Pods are units that encapsulate collections of containers and are excetuted inside nodes. The scale of the cluster plays a crucial role, as the computational needs of different services are variant and the cluster must be able to withstand the different usage scenarios (e.g. large traffic in a server) . Thus, efficient scaling lies in making efficient and timely predictions about how many and which resources will be allocated to certain processes, so that system failure is avoided and idleness of the resources is minimized. In this thesis, we begin by studying the Kubernetes architecture and the different scaling concepts with a focus on horizontal scaling, meaning the scaling where the number cluster nodes is modified. For our tests, we study the CPU usage in a cluster orchestrated by Kubernetes. Namely, we study anonymized data collected over 15 days from a production server in CERN that contain CPU usage data over time. We perform exploratory data analysis to further understand the data and proceed to develop forecasting models for CPU usage predictions. At last, we develop a reinforcement learning agent with a policy of minimizing CPU usage based on horizontal scaling of the cluster. Our ultimate goal is to propose a proactive custom resource allocator that can be integrated into a Kubernetes cluster, so that the scaling decisions are based on CPU usage prediction and thus a provision of the necessary number of nodes.

Keywords

Deep Learning, Kubernetes, Deep Q Learning, Reinforcement Learning, LSTM, Horizontal Scaling, Time series forecasting

To My Parents

Ευχαριστίες

Η παρούσα διπλωματική αποτελεί κατόρθωμα προσωπικό και συλλογικό καθώς η ολοκλήρωση της παρούσας δεν θα μπορούσε να είχε επιχτευχθεί χωρίς την συνδρομή πολλαπλών προσώπων.

Θα ήθελα να ευχαριστήσω τον υπεύθυνο καθηγητή Νεκτάριο Κοζύρη για την προθυμία και την αρωγή του προς την εκπόνηση της παρούσας διπλωματικής, καθώς και τα μέλη του CSLab που παρείχαν βοήθεια και τους απαραίτητους πόρους για την εξοικείωση με τα εργαλεία. Επιπλέον θα ήθελα να ευχαριστήσω τον διδακτορικό φοιτητή Κωνσταντίνο Μπιτσάκο, η συνδρομή του οποίου ήταν καθοριστική για την πορεία αυτής της εργασίας με τις γνώσεις και τις συμβουλές που προσέφερε.

Ακόμη, θα ήθελα να ευχαριστήσω τους φίλους μου, τους εξ αποστάσεως και δια ζώσης που σταθερά με υποστήριζαν σε όλες μου τις σπουδές και τον αδερφό μου τον Γιάννο που αποτελεί στήριγμα για μένα σε κάθε κατάσταση. Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου, Θεόδωρο και Μαργαρίτα που χωρίς την έμπνευση αλλά και τη στήριξη που μου έδωσαν δεν θα μπορούσα να είχα καταφέρει να ξεκινήσω αλλα και να ολοκληρώσω αυτές τις σπουδές.

Νεφέλη Παναγιώτα Τζαβάρα

Αθήνα, Νοέμβρης 2023

Contents

List of Figures

Chapter 0

Εκτεταμένη Ελληνική Περίληψη

Το παρόν κεφάλαιο έχει ως σκοπό την ανάπτυξη των εννοιών και μεθόδων της παρούσας διπλωματικής στα ελληνικά με περιληπτικό τρόπο, διατηρώντας όμως τη κεντρική ουσία.

0.1 Περίληψη

Η παρούσα διπλωματική αφορούσε μελέτη του Kubernetes και της κλιμακωσής του. Συγκεκριμένα, μελετήσαμε τους τρόπους με τους οποίους πραγματοποιείται η κλιμάκωση των πόρων ώστε το σύστημα να διατηρήσει κάποιες σταθερές επιθυμητές συνθήκες. Στόχος ήταν η μελέτη των βημάτων, ώστε να σχεδιαστεί ένας resource allocator με χρήση Deep Q Learning για τη βελτίωση των αποφάσεων κλιμάκωσης.

Το κίνητρο για τη παρούσα διπλωματική ξεκίνησε από το γεγονός ότι τα τελευταία χρόνια οι υπολογιστικές ανάγκες έχουν αυξηθεί και αυτό έχει συντελέσει στη δημοτικότητα της κατανεμημένης επεξεργασίας, καθώς πλέον πάρα πολλές υπηρεσίες μεταφέρονται στα υπολογιστικά νέφη. Πλέον, οι εφαρμογές τρέχουν σε μορφή microservices στα υπολογιστικά νέφη, τα οποία με τη σειρά τους καταμερίζουν τα microservices σε περιέκτες ή αλλιώς containers, μια δομή που δίνει αυτονομία στην εκτέλεση και την σωστή λειτουργία των εν λόγω εφαρμογών. Το πιο δημοφιλές εργαλείο για την ενορχήστρωση, δηλαδή την αυτοματοποίηση της διαδικασίας του deployment, της κλιμάκωσης και της επιτήρησης της σωστής λειτουργίας αυτών των πόρων είναι το Kubernetes, το οποίο, αν και ξεκίνησε ως εσωτερικό πρότζεκτ της Google, είναι πλέον ανοιχτού κώδικα και πολύ πρόσθετο λογισμικό αναπτύσσεται επ' αυτού.

Ωστόσο, το δικό μας κίνητρο έγκειται στη παρατήρηση ότι η προεπιλογή του Kubernetes περιλαμβάνει μη αποδοτική κλιμάκωση, καθώς βασίζεται σε συνθήκες κατωφλιοποίησης. Η μη αποδοτική κλιμακωσιμότητα έχει μεγάλο αντίκτυπο στη χρήση επεξεργαστικής ισχύος και μνήμης, το οποίο εκτός από κακή απόδοση έχει και περιβαλλοντικό κόστος.

Το πρόβλημα, λοιπόν, διατυπωμένο είναι η μελέτη και σχεδίαση ενός resource allocator, ο οποίος θα μπορούσε να ενσωματωθεί σε μια συστάδα ενορχηστρωμένη από το Kubernetes για να βελτιώσει τη χρήση επεξεργαστικής ισχύος, κρατώντας τη γύρω από μια επιθυμητή τιμή. Πιο συγκεκριμένα, βάσει μιας χρονοσειράς χρήσης επεξεργαστικών πόρων θα πρέπει να προβλέπουμε τον αριθμό των pods που πρέπει να διατεθούν στο σύστημα κρατώντας κάποιους περιορισμούς.

Τα δεδομένα που χρησιμοποιήσαμε είναι μια καταγραφή δεδομένων επεξεργαστικής ισχύος από πραγματική χρήση εφαρμογών στο ινστιτούτο CERN, τα οποία είναι διαθέσιμα για εξωτερική χρήση. Οι μετρήσεις έγιναν σε διάρκεια 15 ημερών ανά 5 λεπτά.

Η δικιά μας προτεινόμενη σχεδίαση περιλαμβάνει την ανάπτυξη ενός Πράκτορα ενισχυτικής μάθησης. Όπως είναι γνωστό, πλέον η Μηχανική Μάθηση έχει εφαρμογές σε πολλούς κλάδους της επιστήμης υπολογιστών, και δεν αποτελεί καμία εξαίρεση ο κλάδος της πρόβλεψης χρονοσειρών και η λήψη αποφάσεων σε πολυπαραμετρικά περιβάλλοντα όπως το δικό μας. Ο πράκτορας, όπως θα αναλύσουμε και παρακάτω, θα στηρίζεται σε Deep Q Learning, μια τεχνική βασισμένη στον δυναμικό προγραμματισμό, καθώς και σε μοντέλα LSTM που είναι γνωστά στη βιβλιογραφία για την καταλληλότητα σε πρόβλεψη χρονοσειρών.

0.2 Θεωρητικό Υπόβαθρο

Στην παρούσα υποενότητα θα παρουσιάσουμε τις απαραίτητες θεωρητικές γνώσεις για να θεμελιώσουμε το υπόβαθρο της διπλωματικής. Ξεκινώντας από έννοιες όπως είναι οι περιέκτες και το Kubernetes, θα προχωρήσουμε εξηγώντας βασικές έννοιες της ενισχυτικής μάθησης και τις εφαρμογές τους.

Kubernetes

Προχωρώντας στο απαραίτητο υπόβαθρο θα αναλύσουμε συνοπτικά τη δομή και λειτουργία μιας συστάδας Kubernetes και συγκεκριμένα τα πεδία του που μας ενδιαφέρουν περισσότερο.

Κάθε συστάδα έχει έναν (ή και περισσότερους) master nodes με τον ρόλο του control plane και συνήθως $n > 2$ worker nodes , ώστε να τρέχουν το φορτίο.

Στο control plane υπάρχουν τα εξής μέρη:

- Ο apiserver που ειναι ουσιαστικά το front end που επικοινωνεί με το Kubernetes API, το οποίο διαχειρίζεται το configuration των pods, services και άλλων μερών του συστήματος. Εκεί ανήκουν και οι scalers που θα δούμε παρακάτω, όπως ο HPA.
- Ο controller manager, που ουσιαστικά αποτελεί έναν ατέρμονο βρόγχο υπεύθυνο για την συνεχή επιτήρηση της κατάστασης της συστάδας και πιο συγκεκριμένα του kube-apiserver.
- Ο Kubernetes Scheduler αποτελεί σημαντικό μέρος του συστήματος. Μέσω του δρομολογητή αυτού χρονοπρογραμματίζονται οι διεργασίες στα κατάλληλα pods. Ο τρόπος λειτουργίας του Scheduler ειναι περίπλοκος και βασίζεται σε μια ορισμένη ακολουθιακή διαδικασία στην οποία αρχικά γίνεται ένα φιλτράρισμα των διαθέσιμων πόρων ως προς την καταλληλότητά τους και ύστερα βαθμολογούνται και επιλέγονται για δρομολόγηση.
- Η μνήμη etcd, όπου το d προέρχεται από το distributed, όπου δηλαδή αποθηκεύονται κεντρικά οι πληροφορίες για το κατανεμημένο cluster ώστε να υπαρχει ένα κοινό αρχείο με συνέπεια.

Από την άλλη, οι worker nodes απαρτίζονται από :

- το kubelet, το οποίο ευθύνεται για την επικοινωνία με τον master και υπάρχει ένας σε κάθε κόμβο
- το kubeproxy, το οποίο επίσης υπάρχει σε κάθε κόμβο και επιμελείται τα endpoints των εφαρμογών που είναι σε λειτουργία εντός της συστάδας.

Τα παραπάνω φαίνονται στην Εικόνα 1.

Προχωράμε σε μια από τις βασικότερες ιδιότητες του Kubernetes, τη κλιμακωσιμότητα.

Συγκεκριμένα, οι μηχανισμοί κλιμακωσιμότητας χωρίζονται σε 3 βασικές κατηγορίες , την οριζόντια, την κάθετη και την κλιμάκωση σε επίπεδο συστάδας.

- Στην οριζόντια κλιμακωσιμότητα πρακτικά αλλάζουν οι κόμβοι και το πόσα pods διαμοιράζονται εντός της συστάδας, κάτι που επηρεάζει άμεσα την χρήση επεξεργαστή και μνήμης.
- Στην κάθετη κλιμακωσιμότητα αλλάζουν τα χαρακτηριστικά κάθε κόμβου, δηλαδή κάθε μηχανήματος, όπως είναι οι μνήμες και ο επεξεργαστής, με αποτέλεσμα να προσαρμόζονται μεταβλητές, όπως το πόση μνήμη ή μέχρι τι επεξεργαστική ισχύ μπορεί να διατεθεί για την εκτέλεση της

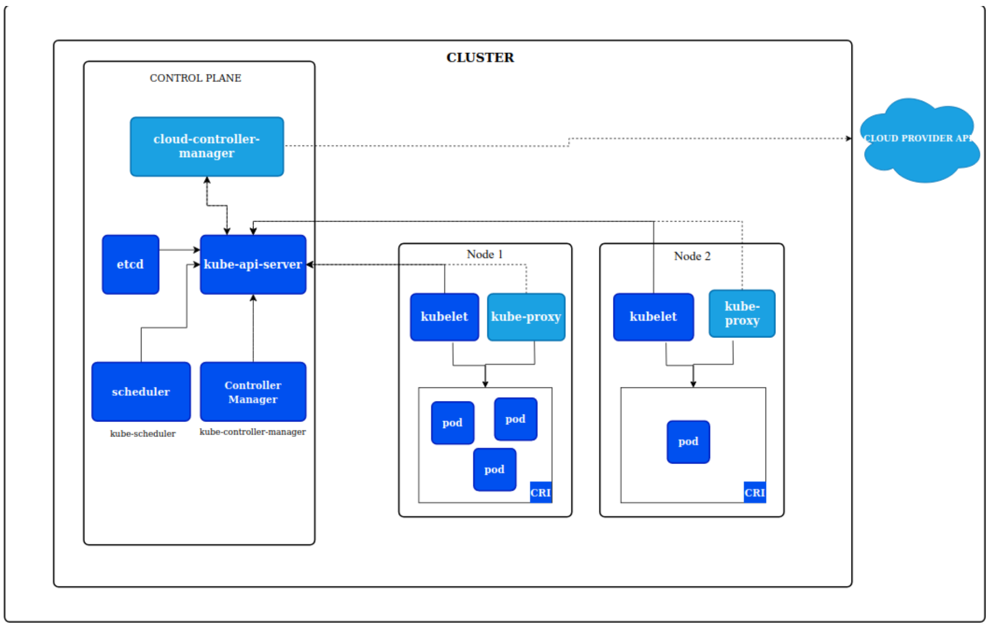


Figure 1. *Η εσωτερική αρχιτεκτονική του Kubernetes*

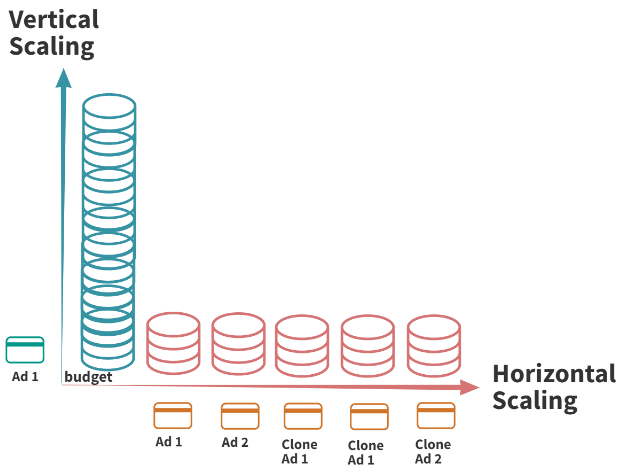


Figure 2. *Στρατηγική οριζόντιας και κάθετης κλιμάκωσης*

εκάστοτε εφαρμογής. Στις περισσότερες περιπτώσεις αυτό συνεπάγεται την ολική επανεκκίνηση της συστάδας και νέα ανακατανομή των πόρων. Είναι ένας δύσχρηστος μηχανισμός, διότι μέχρι στιγμής υλοποιείται μέσω του τερματισμού και επανεκκίνησης των pods.

- Στη κλιμάκωση σε επίπεδο συστάδας προσαρμόζεται το μέγεθος της ίδιας της συστάδας δυναμικά, συνήθως μέσω κάποιας υπηρεσίας νέφους που διευκολύνει τη δέσμευση και αποδέσμευση μηχανημάτων.

Θα εστιάσουμε στην οριζόντια κλιμάκωση και συγκεκριμένα στον Horizontal Pod Autoscaler, την οποία πραγματοποιεί αυτόματα ενσωματωμένος μέσα στον metrics server του Kubernetes. Οι μετρικές που παρακολουθεί ο HPA είναι συνήθως η χρήση επεξεργαστή και μνήμης, ωστόσο μπορεί να επεκταθεί και σε custom μετρικές. Οι αποφάσεις του, όμως, περιγράφονται από κατωφλιοποίηση. Καθώς οι επιδόσεις του δεν είναι επαρκείς για πολλούς χρήστες, και αφού υπάρχει μεγάλη open source κοινότητα, πολλοί χρήστες υλοποιούν custom scaling policies. Τέτοιες μπορεί να είναι βάσει κατωφλίου, με διαφορετικές επιλογές κατωφλίου, βασισμένα σε queing [1] [2] [3], control theory[4],

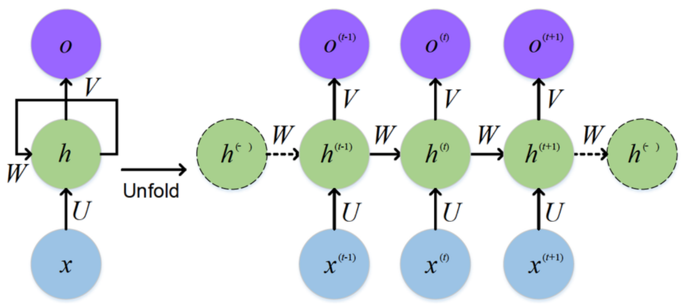


Figure 3. *Η βασική δομή του RNN και η αναλυτική ¨ξεδιπλωμένη' περιγραφή του*

historical performace [5], αλλά και reinforcement learning [6] [7] [8], όπως ο δικός μας.

Εδώ βασικό είναι να αναφέρουμε ότι οι custom πολιτικές μπορούν να ενσωματωθούν σε Kubernetes cluster μέσω σχεδίασης custom controller. Οι custom controllers είναι ανεξάρτητα μέρη που αλλάζουν την συμπεριφορά του control plane κατά το δοκούν, με τις επιθυμητές προδιαγραφές και πολιτικές. Είναι ανεξάρτητες διεργασίες που επικοινωνούν με το Kubernetes API. Στην δικιά μας περίπτωση, ο custom controller θα μεσολαβήσει μέσω του control plane και του πράκτορα ενισχυτικής μάθησης, γεφυρώνοντας τις περίπλοκες αποφάσεις για την κλιμακωσιμότητα σε πράξεις του Kubernetes.

Μηχανική Μάθηση

Η Μηχανική Μάθηση περιλαμβάνει ένα τεράστιο εύρος μοντέλων, στόχος των οποίων είναι η αυτόματη εκμάθηση των μοτίβων και πρόβλεψη των λύσεων ενός προβλήματος. Έχει αποκτήσει μεγάλη δημοτικότητα στο πεδίο της επιστήμης των υπολογιστών για πλείστες εφαρμογές σε προβλήματα κατηγοριοποίησης, πρόβλεψης κ.α.

Χωρίζεται σε 3 βασικές κατηγορίες: την Επιβλεπόμενη Μάθηση, την Μη Επιβλεπόμενη Μάθηση και την Ενισχυτική Μάθηση. Στη παρούσα εργασία θα χρησιμοποιήσουμε επιβλεπόμενη μάθηση για τη πρόβλεψη χρήσης επεξεργαστικής ισχύος και ενισχυτική μάθηση για τις τελικές προβλέψεις πάνω στην συστάδα κόμβων. Το μοντέλο επιβλεπόμενης μάθησης που θα χρησιμοποιήσουμε είναι το Long Short Term Memory Model (LSTM).

Τα μοντέλα LSTM είναι μια μεταγενέστερη εκδοχή των μοντέλων ανατροφοδότησης RNN. Εν συντομία, τα RNN εφαρμόζουν ανατροφοδότηση του ενδιάμεσου στρώματος hidden layer υπολογισμού των βαρών, δημιουργώντας μια εξάρτηση μεταξύ εισόδου και εξόδου του κάθε κελιού (Βλ. Εικόνα 3), δηλαδή ιδιότητα μνήμης. Ωστόσο, παρουσιάζουν βασικά προβλήματα υπολογισμού, καθώς κάποιοι παράμετροι είτε εξαφανίζοταν στον χρόνο είτε παρουσίαζαν τεράστιες τιμές (vanishing gradients, exploding gradients). Τα μοντέλα LSTM έλυσαν τα παραπάνω προβλήματα εισάγοντας μια δομή που φαίνεται στην Εικόνα 4. Το νέο κελί υπολογισμού διαθέτει 3 πύλες, μια υπεύθυνη για την είσοδο, μια για την έξοδο και την πιο σημαντική: την πύλη που ορίζει το εάν και πόσο το μοντέλο θα θυμάται τις προηγούμενες εξόδους.

Λόγω αυτής της δομής, μνημονεύουν κοντινές τιμές αλλά και βαθύτερα μοτίβα μέσα στον χρόνο, και έτσι έχουν χρησιμοποιηθεί σε πλείστες εφαρμογές[9][10], όπως είναι η πρόβλεψη κατανάλωσης ενέργειας [11][12] και το χρηματιστήριο[13][14]

Ενισχυτική Μάθηση

Η Ενισχυτική μάθηση (Reinforcement Learning) είναι πεδίο της Μηχανικής Μάθησης πέραν της κλασσικής Εποπτευόμενης και Μη Εποπτευόμενης Μάθησης.

Η βασική ιδέα της ενισχυτικής μάθησης συμπυκνώνεται στο εξής σενάριο: ένας Πράκτορας (Agent) πραγματοποιεί Δράσεις (Actions) σε ένα Περιβάλλον, (Environment) το οποίο του επιστρέφει Κέρδη

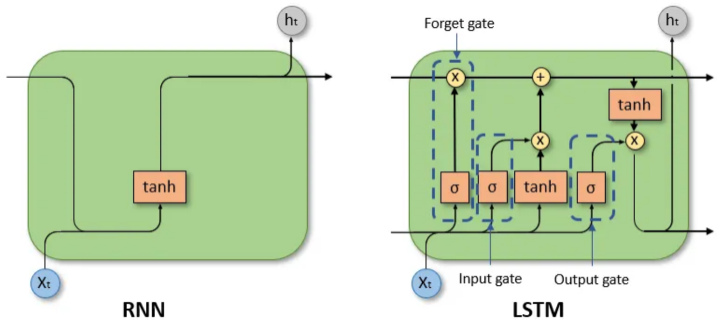


Figure 4. *Οι δομικές διαφορές μεταξύ RNN & LSTM*

(Rewards). Στόχος του Πράκτορα είναι η μεγιστοποίηση των Κερδών που του επιστρέφονται για τις Δράσεις του. Ορίζουμε:

- s την παρούσα κατάσταση
- s' την επόμενη κατάσταση
- a την πράξη μετάβασης
- $Q(s, a)$ τον πίνακα με τα ανεμενόμενα συγκεντρωτικά κέρδη για κάθε ζεύγος s, a
- γ τον παράγοντα που ορίζει το ποσοστό στο οποίο θα έχει ρόλο η μέλλουσα επιβράβευση, όπου $\gamma = 0$ καθόλου, $\gamma = 1$ πολύ
- $R(s, a, s')$ τη συνάρτηση κέρδους μετάβασης από μια κατάσταση σε μια άλλη

Ο Πράκτορας μαθαίνει την πιθανότητα μετάβασης στην επόμενη κατάσταση s' από την αλληλεπίδραση με το Περιβάλλον. Το $Q - Learning$ είναι μια από τις βασικότερες τεχνικές της ενισχυτικής μάθησης, που χρησιμοποιείται ώστε ένας πράκτορας να μάθει μια βέλτιστη πολιτική $\pi*$.Η συνάρτηση περιγράφει την Πράξη a που πραγματοποιεί ο Πράκτορας στη κατάσταση s, η οποία τον φέρνει στην κατάσταση s' όπου διαλέγει την βέλτιστη πράξη a' που μεγιστοποιεί το $Q - Value$.Αυτό μπορεί να περιγραφεί από τις θεμελιώδεις συναρτήσεις:

$$Q^*_{k+1}(s, a) = \sum_{s'} P(s'|s, a)(R(s, a, s') + \gamma \max a' Q_k(s', a')) \tag{1}$$

$$target(s') = R(s, a, s') + \gamma \max a' Q_k(s', a') \tag{2}$$

Για να είναι εφικτή η σύγκλιση του πράκτορα προς μια βέλτιστη πολιτική, αρχικά κβαντοποιούμε τις πιθανές καταστάσεις μετάβασης σε ένα διακριτό χώρο καταστάσεων και επιπλέον εισάγουμε την έννοια του $Q - Network$.Έτσι, ο χώρος των πιθανών καταστάσεων μειώνεται σημαντικά σε μέγεθος και με τη χρήση νευρωνικών δικτύων λαμβάνουμε προσεγγιστικές τιμές στις τιμές του πίνακα $Q(s, a)$, ώστε να μην υπάρχουν ανεξερεύνητες καταστάσεις που να επηρεάσουν σημαντικά τον Πράκτορα.

Η υλοποίηση των παραπάνω σε αυτό που είναι σήμερα γνωστό ως ενισχυτική μάθηση ήρθε με την εισαγωγή του Deep Q Network (DQN), ενός αλγορίθμου που επαναστάτησε στον χώρο της Ενισχυτικής Μάθησης υλοποιώντας το “playing Atari with Deep RL” [15]. Εκεί εισάγονται δύο νέες ενότητες : το Target Network και ο Experience Replay Buffer.

Το Target Network αποτελεί ένα νέο νευρωνικό Q', ακριβές αντίγραφο του Q. Το Q' έχει παραμέτρους θ' οι οποίες προσαρμόζονται ανά κάποια ορισμένη συχνότητα C με τις θ. Αυτό επιλύει το πρόβλημα του κινούμενου στόχου που παρουσίαζαν μέχρι τότε οι αλγόριθμοι $Q - Learning$.

Ο Experience Replay Buffer αποτελεί μια μνήμη των προηγούμενων καταστάσεων, μαζί με τις πράξεις και την επιβράβευση που έφεραν την νέα κατάσταση. Τυχαία δείγματα επιλέγονται από τον Replay Buffer για την εκπαίδευση του Πράκτορα.

0.3 Πειραματικό Μέρος

Στην πρόσφατη βιβλιογραφία παρατηρούμε αύξηση της έρευνας σχετικά με σχεδιασμό παραμετροποιημένων scalers. Αυτό προκύπτει από την φύση των διαφορετικών εφαρμογών που μπορούν να έχουν ποικίλες απαιτήσεις σε μνήμη, επεξεργαστική δύναμη και άλλους πόρους. Ωστόσο, η κύρια δυσκολία στους σημερινούς autoscalers είναι ότι βασίζονται σε δεδομένες τιμές και δεν προσαρμόζονται δυναμικά, ώστε να αποδώσουν στην παρούσα κατάσταση απαιτήσεων. Συνεπώς, υπάρχει ανάγκη για ανάπτυξη προνοητικών πρακτόρων (Proactive Agents), δηλαδή πράκτορες που παίρνουν υπόψη ιστορικά δεδομένα της συστάδας για να πάρουν αποφάσεις κλιμακωσιμότητας. Μια Proactive προσέγγιση στα παραπάνω αποτελεί και ένας Deep Q Network Πράκτορας.

Συνεχίζουμε με μια σύντομη περιγραφή της μεθοδολογίας στην οποία στηρίχθηκε η μελέτη μας. Παρακάτω θα ξεκινήσουμε εξηγώντας τα δεδομένα που χρησιμοποιήσαμε και τη διερευνητική ανάλυση που πραγματοποίησαμε πάνω σε αυτά, τα μοντέλα LSTM που αναπτύξαμε για τη πρόβλεψη χρήσης επεξεργαστικής ισχύος και τέλος, τους πειραματισμούς μας με τον πράκτορα βαθειάς ενισχυτικής μάθησης και τα αποτελέσματά του στη κλιμάκωση.

Διερευνητική Ανάλυση Δεδομένων

Σκοπός της διερευνητικής ανάλυσης δεδομένων είναι η καλύτερη κατανόηση των δεδομένων μέσω μεθόδων παρατήρησης και μελέτης τους. Μία τέτοια μέθοδος είναι ο καθαρισμός των κακών καταχωρήσεων π.χ. κενά πεδία που δημιουργούν ασυνέπειες στην επεξεργασία. Έπειτα, η μελέτη επεκτείνεται στην φανέρωση των μοτίβων των δεδομένων. Στην παρούσα περίπτωση που διαθέτουμε χρονοσειρές, μπορούμε να παρατηρήσουμε τις τάσεις και την εποχικότητα που παρουσιάζουν. Μέσα στην ΔΕΑ περιλαμβάνεται και η οπτικοποίηση, όπου θα παρατηρηθούν μοτίβα που ίσως να μη φαίνονται στην μαθηματική χρονοσειρά. Τα δεδομένα που αξιοποιήσαμε είναι δημοσίως προσβάσιμα, προερχόμενα από μια συστάδα ενορχηστρωμένη από το Kubernetes, η οποία έτρεχε σε διακομιστή παραγωγής στο ινστιτούτο CERN. Δεκάδες χιλιάδες χρήστες καλούν καθημερινά την υπηρεσία Single Sign On, οπότε τα API calls είναι πλείστα και αναμένουμε μοτίβα σε ώρες, εργάσιμες ημέρες και ώρες αιχμής. Η χρονοσειρά περιελάμβανε συνολικά 4435 σημεία που καταγράφησαν με παρεμβολές 5 λεπτών. [16][17]

Προχωρώντας στις οπτικοποιήσεις των δεδομένων, βλέπουμε την πλήρη χρονοσειρά και τη κατανομή της. Από τις συνήθεις συναρτήσεις της βιβλιοθήκης Pandas βλέπουμε, επίσης, τα βασικά στατιστικά στοιχεία όπως η μέση, μέγιστη και ελάχιστη τιμή και τα ποσοστιαία διαστήματα. Παρατηρούμε ότι η μέση τιμή είναι 0.3486, πολύ πιο κάτω από τη μέγιστη τιμή, καθώς και ότι το 75% των τιμών είναι κοντά σε αυτήν τη τιμή και όχι κοντά στο 2.056. Αυτό μας υποδεικνύει πως υπάρχουν απότομες μεταβολές κατά τις οποίες η επεξεργαστική ισχύς παίρνει μεγάλες τιμές. Αυτό μπορούμε να το επαληθεύσουμε και από τη κατανομή, όπου οι περισσότερες τιμές είναι στο διάστημα 0.02 - 0.5. Από 0.5 και πάνω μπορούμε με ασφάλεια να συμπεράνουμε ότι είναι οι τιμές με υψηλή χρήση. Από την τιμή 1.25 και πάνω μπορούμε να θεωρήσουμε πλέον ότι η τιμή είναι απροσδόκητα υψηλή.

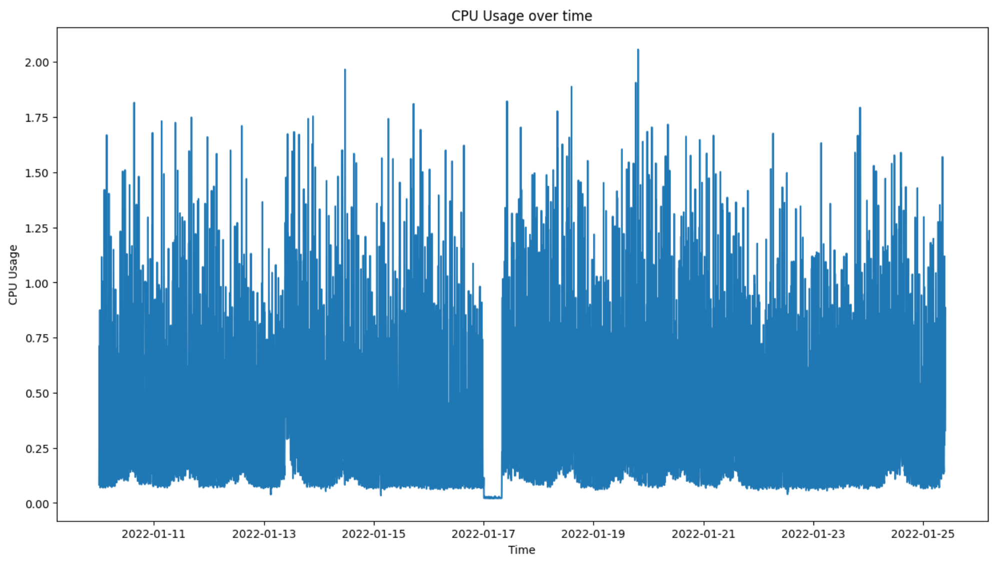


Figure 5. *Η χρονοσειρά χρήσης επεξεργαστικής ισχύος*

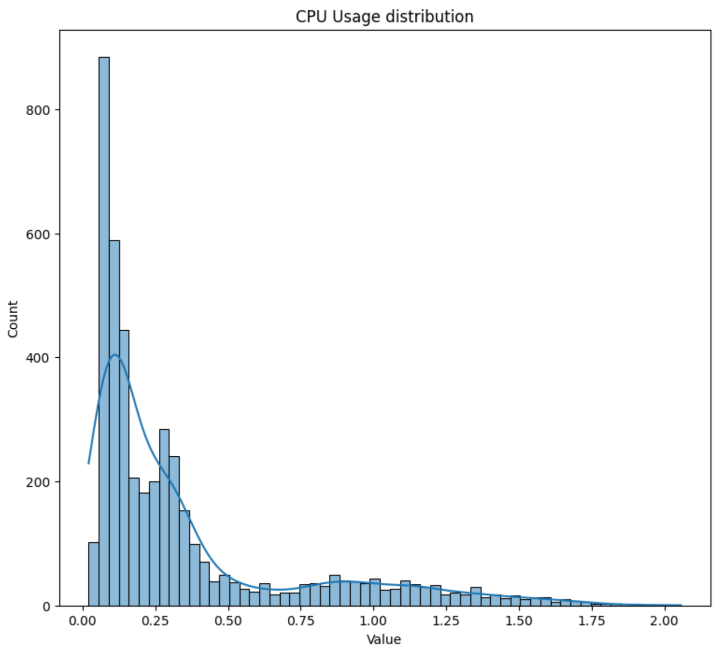


Figure 6. *Η κατανομή χρήσης επεξεργαστικής ισχύος*

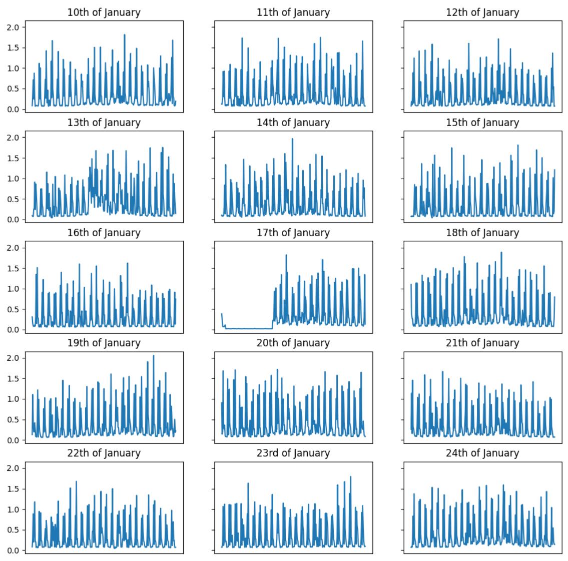


Figure 7. *Η ημερήσια χρήση επεξεργαστικής ισχύος*

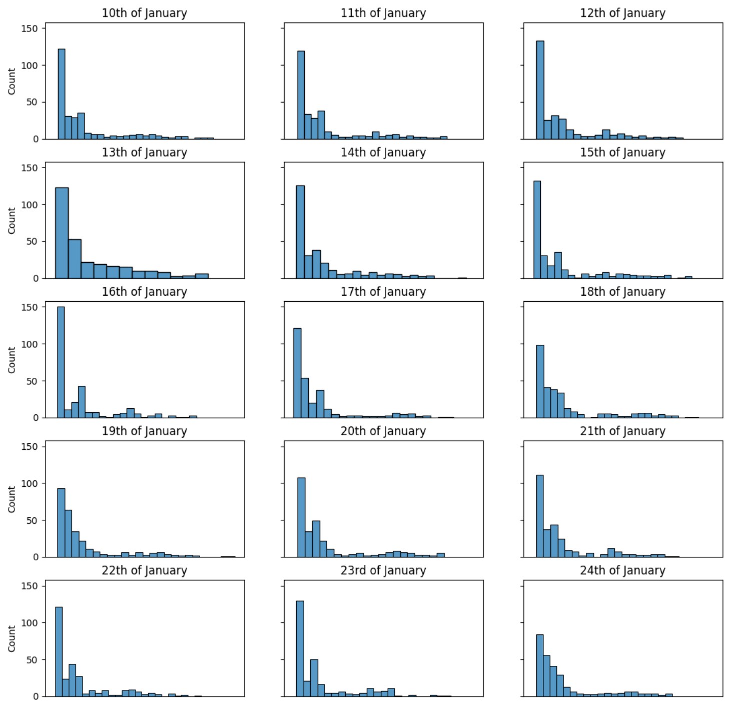


Figure 8. *Η ημερήσια κατανομή χρήσης επεξεργαστικής ισχύος*

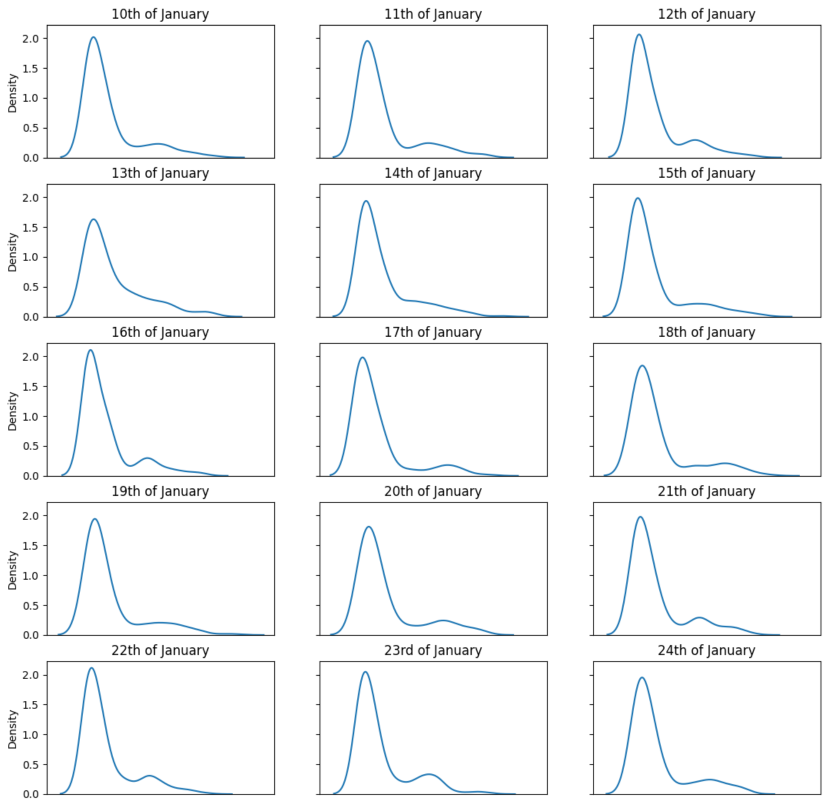


Figure 9. *Η ημερήσια κατανομή χρήσης επεξεργαστικής ισχύος*

Για τη καλύτερη κατανόηση των δεδομένων προχωράμε σε ανάλυση ανά ημέρα, όπου μπορούμε να παρατηρήσουμε διαφορές στις κατανομές τα Σαββατοκύριακα. Εκεί, οι κατανομές περίπου μετά την τιμή 1.25 που θέσαμε πριν είναι πιο επίπεδες από άλλες μέρες.

Οπτικοποιώντας τη χρονοσειρά παρατηρούμε, επίσης, μια ανωμαλία την 17η Ιανουαρίου, όπου υπήρξε διακοπή ρεύματος για κάποιο διάστημα, οπότε δεν έγιναν κλήσεις στο API.

Επιπλέον, υπολογίζουμε τη συσχέτιση μεταξύ των ημερών για να δούμε τυχόν μοτίβα μεταξύ τους. Όλα τα ζεύγη με μεγάλη συσχέτιση είναι διαφορετικές ημέρες της εβδομάδας, αλλά βλέπουμε πως οι ημέρες 22,23,24, δηλαδή οι 3 τελευταίες είναι ισχυρά συσχετιζόμενες με τις 12,10 και 11 αντίστοιχα. Αυτό μας δίνει ένα έναυσμα να μη δοκιμάσουμε το μοντέλο με δεδομένα επικύρωσης μόνο από τις τελευταίες 3 μέρες, καθώς θα έχει ήδη δει το ίδιο μοτίβο στις 3 πρώτες μέρες.

Για την ανάλυση της τάσης και της εποχικότητας διαλέγουμε μια εργάσιμη και μια μη εργάσιμη μέρα. Παρατηρούμε ότι η τάση αυξάνεται τις εργάσιμες ώρες στην εργάσιμη μέρα και μετά μειώνεται, ενώ στην μη εργάσιμη διατηρεί μια σχετικά σταθερή χαμηλή τιμή. Για την εποχικότητα βλέπουμε εμφανώς 2 κορυφώσεις ανά την ώρα. Στις έκτοπες τιμές, επίσης, παρατηρούμε πολύ περισσότερες τιμές στην εργάσιμη μέρα.

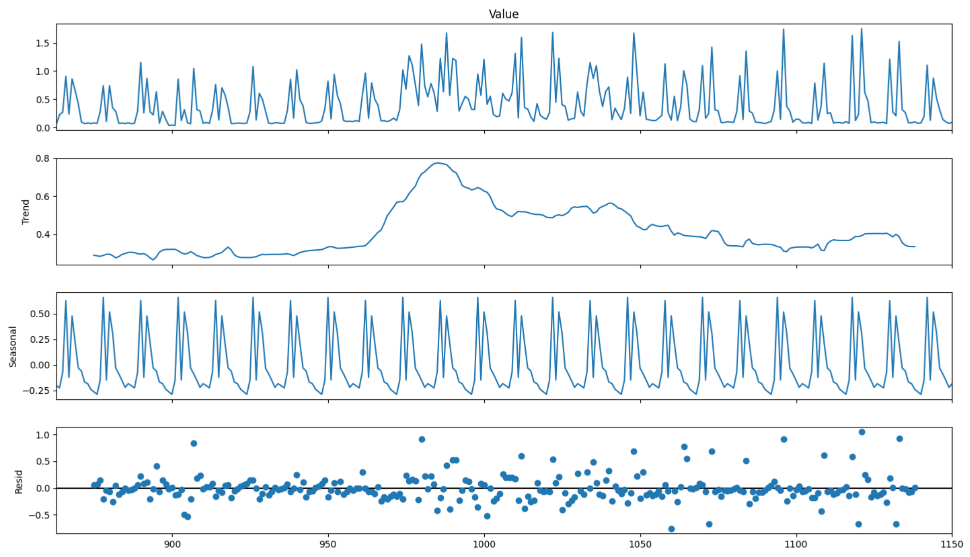


Figure 10. *Η εποχικότητα, η τάση και οι έκτοπες τιμές για μια καθημερινή*

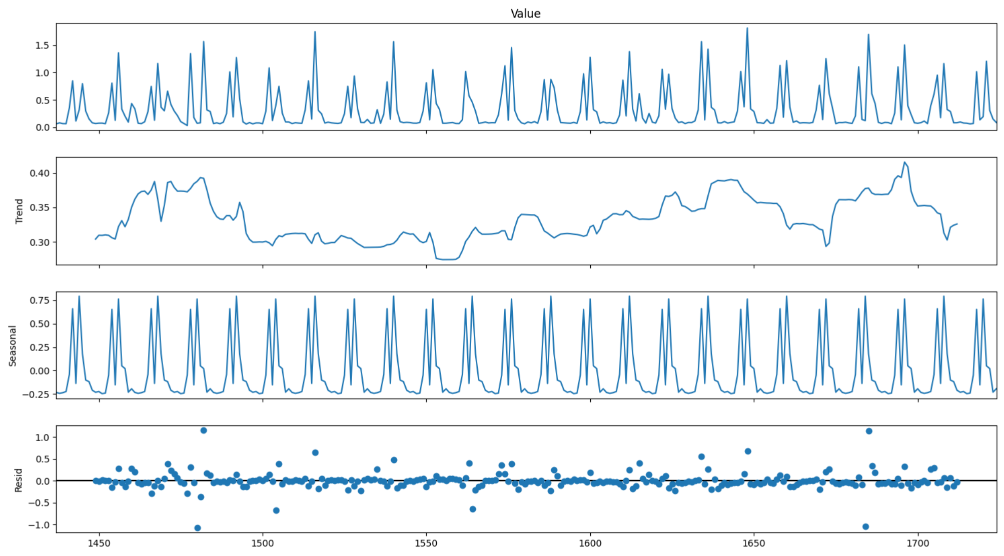


Figure 11. *Η εποχικότητα, η τάση και οι έκτοπες τιμές για το Σαββατοκύριακο*

Συνολικά, λοιπόν, από την ΔΕΑ εξάγουμε ότι παρατηρούνται μοτίβα χρήσης επεξεργαστικής ισχύος. Παρατηρούμε πολλές ξαφνικές κορυφώσεις που προκαλούν την ανάγκη για κλιμάκωση. Επίσης, παρατηρούμε ωριαία εποχικότητα, με τις κορυφώσεις να συμβαίνουν παρόμοιες ώρες κάθε ημέρα. Αυτές ερμηνεύονται από τον αυτόματο συγχρονισμό του CERN (π.χ. για permissions σε authorization groups).

Μοντέλα Πρόβλεψης Χρονοσειράς

Έχοντας πλέον γνώση των δεδομένων που θα αναλύσουμε, προχωράμε στην ανάλυση και κατασκευή μοντέλων για τη πρόβλεψη της χρήσης της επεξεργαστικής ισχύος. Χρησιμοποιήσαμε τη γλώσσα Python, στο περιβάλλον Kaggle και τις βιβλιοθήκες TensorFlow και Keras [18] σε όλες τις παρακάτω δοκιμές.

Δοκιμάσαμε ένα ευρύ σετ παραμέτρων για την αρχιτεκτονική του μοντέλου και την εκπαίδευση του, συγκεκριμένα:

```
LSTM units = [16, 32, 64, 128]
LSTM Layers = [1, 2]
sequence_length = [4, 6, 12, 144, 287]

epochs: [20,40,60,75,100]
learning rate: [0.00001, 0.0001 , 0.001]
loss= ['mean_squared_error', 'huber_loss_function']
optimizer='adam'
```

Βλέπουμε παρακάτω τη δομή και τις συναρτήσεις εκπαίδευσης του καλύτερου μοντέλου πρόβλεψης.

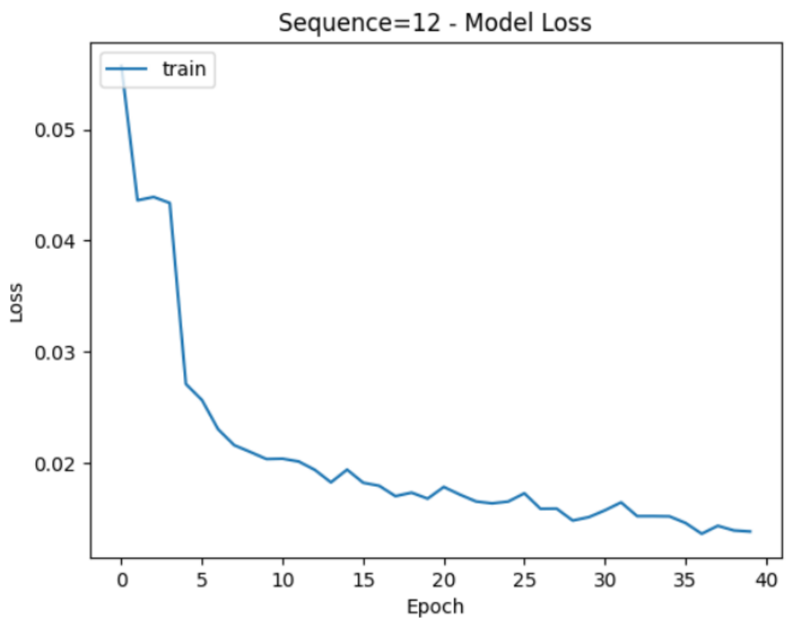
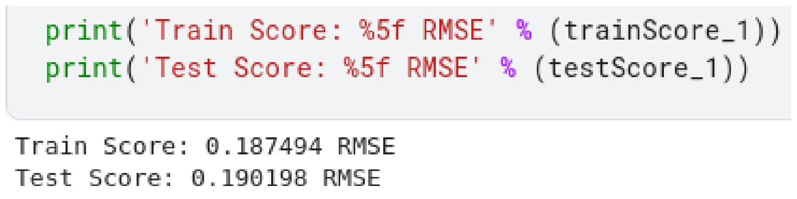


Figure 12. *Η συνάρτηση απωλειών κατά την εκπαίδευση*



```
print('Train Score: %5f RMSE' % (trainScore_1))
print('Test Score: %5f RMSE' % (testScore_1))
```

```
Train Score: 0.187494 RMSE
Test Score: 0.190198 RMSE
```

Figure 13. *Τα σφάλματα στα δεδομένα εκπαίδευσης και επικύρωσης*

Όπως ήταν αναμενόμενο, οι επιδόσεις ήταν καλύτερες στα δεδομένα εκπαίδευσης. Παρατηρούμε παρακάτω μια ικανοποιητική προσέγγιση της χρονοσειράς.

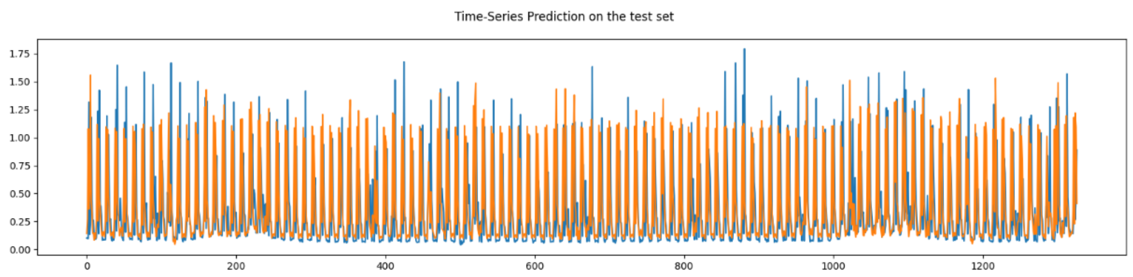


Figure 14. *Οι προβλέψεις στα δεδομένα επικύρωσης*

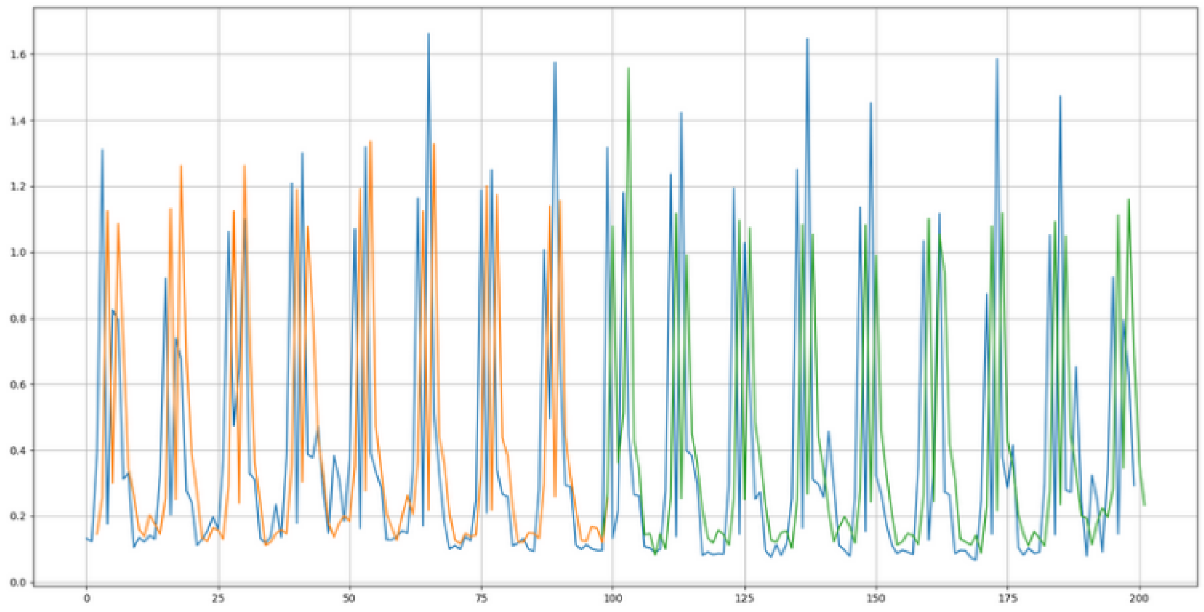


Figure 15. *Πορτοκαλί: Οι προβέψεις στα δεδομένα εκπαίδευσης. Πράσινο: οι προβλέψεις στα δεδομένα επικύρωσης*

Πράκτορας Βαθειάς Ενισχυτικής Μάθησης

Υλοποιώντας, τελικά, τον Πράκτορα έχουμε ικανοποιητικές επιδόσεις στις προβλέψεις. Παρατηρούμε παρακάτω πως η συνάρτηση απωλειών μειώνεται καθώς προχωράει η εκπαίδευση και γίνεται η εκμάθηση των μοτίβων κλιμακωσιμότητας, καθώς επίσης και οτι η συνάρτηση κερδών έχει αύξουσα πορεία, διότι τα κέρδη μεγιστοποιούνται.

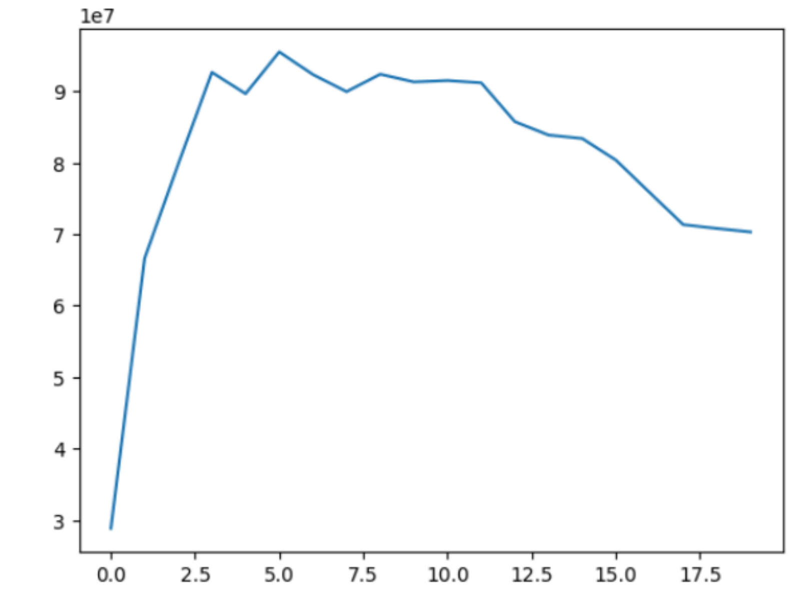


Figure 16. *Η συνάρτηση απωλειών*

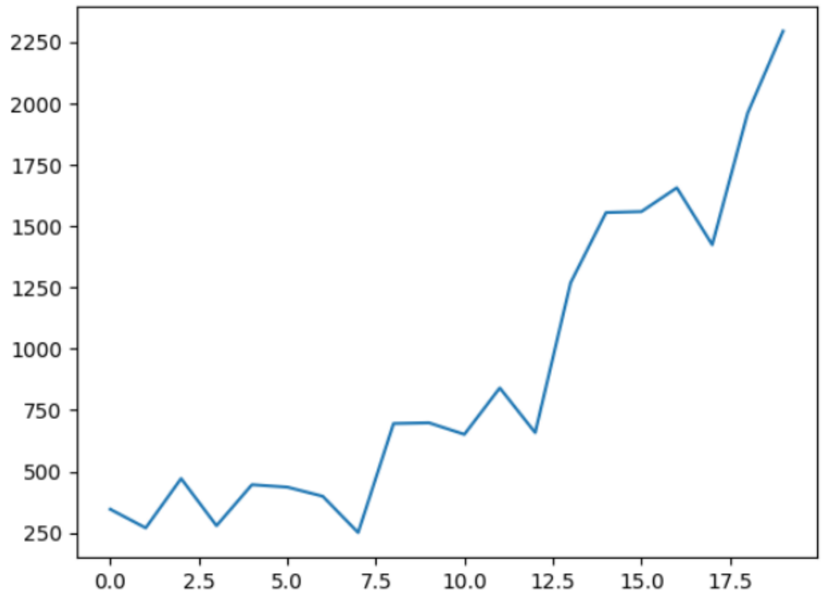


Figure 17. *Η συνάρτηση κέρδους*

Μελετώντας τις επιδόσεις συγκρίνουμε μεταξύ δύο βασικών διαφορετικών καταστάσεων. Στην Εικόνα 18 είχαμε ορίσει $history\_window = 60$, ενώ στην Εικόνα 19 $history\_window = 12$. Εξάγουμε πως στην πρώτη περίπτωση ο πράκτορας κάνει πιο συντηρητικές προβλέψεις, κρατώντας σχετικά σταθερό τον αριθμό των κόμβων, γνωρίζοντας από την εμπειρία πως η αυξημένη χρήση θα επαναληφθεί. Αντιθέτως, στη δεύτερη περίπτωση που η δοθείσα εμπειρία είναι μικρότερη, η κλιμάκωση είναι απότομη και ενδεχομένως ακόμα και μη εφικτή σε πραγματικά σενάρια χρονοπρογραμματισμού.

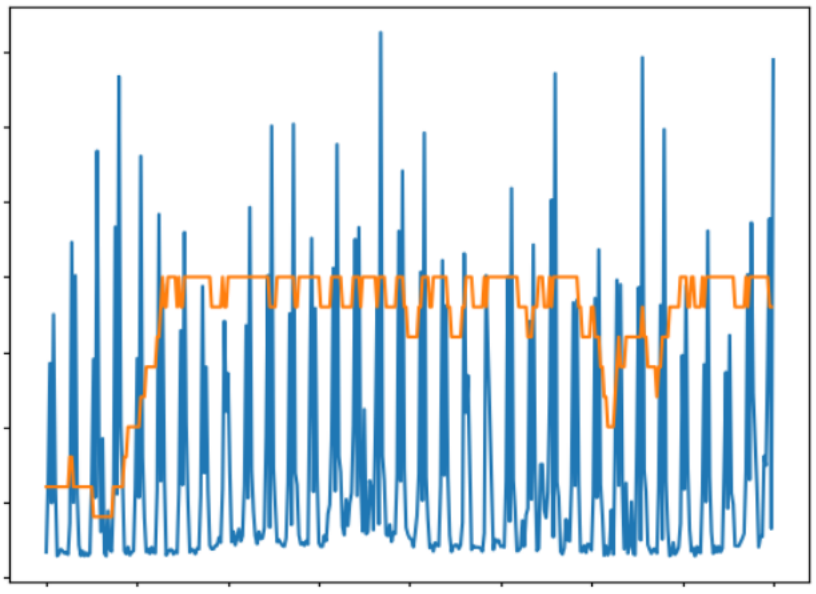


Figure 18. *Οι προβλέψεις με History Window = 60*

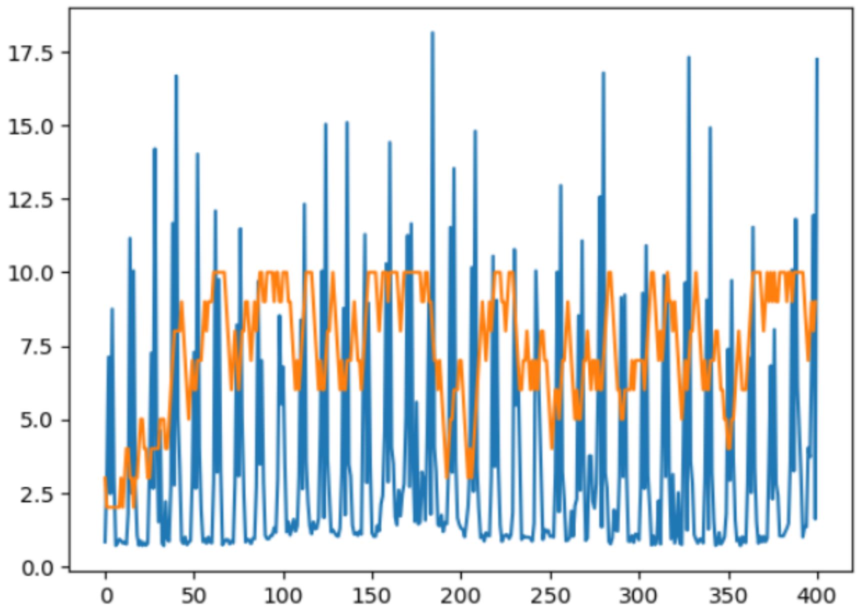


Figure 19. *Οι προβλέψεις με History Window = 12*

0.4 Συζήτηση και Μελλοντική Δουλειά

Σχολιάζοντας τα άνωθι, πρέπει να γίνουν κάποιες επισημάνσεις για τους περιορισμούς της παρούσας εργασίας, καθώς και για τις πιθανές μέλλουσες βελτιώσεις.

Αρχικά, πρέπει να σημειωθεί πως η πηγή των δεδομένων δεν είναι ξεκάθαρη σχετικά με όλα τα χαρακτηριστικά της συστάδας και συνεπώς δεν γνωρίζουμε τον πραγματικό αρχικό αριθμό των κόμβων της συστάδας ούτε άλλες μετρικές που την περιγράφουν. Επομένως, περιοριζόμαστε σε προβλέψεις μόνο βάσει χρήσης του επεξεργαστή. Επιπλέον, πρέπει να αναφερθεί πως, παρά τις προσπάθειες συλλογής δεδομένων σε πραγματική συστάδα κόμβων στο CSLab, δεν μπορέσαμε να πραγματοποιήσουμε σωστό internal network και ανάκτηση των μετρικών από τεστ αντοχής του συστήματος. Ομοίως, οι προσπάθειες δεν ήταν ικανοποιητικές ούτε σε συστάδα που υλοποιήσαμε στο νέφος Okeanos Knossos. Τέλος, τα μοντέλα LSTM και ο Πράκτορας ήταν επιρρεπείς σε overfitting και είναι πιθανό να πέσαμε σε τοπικό ελάχιστο και όχι ολικό, οπότε δεν έχουμε σιγουρευτεί για τη πραγματική απόδοση και λειτουργία του Πράκτορα. Περισσότερα layers και αρχιτεκτονικές πρέπει να δοκιμαστούν τόσο στα LSTM όσο και στην εκπαίδευση του Πράκτορα.

Στη μελλοντική δουλειά πρέπει να συμπεριληφθεί συλλογή δεδομένων από πραγματικά τεστ αντοχής σε δικιά μας συστάδα, η οποία να συνδυάζει πολλαπλά είδη δεδομένων από την ευρεία γκάμα που υποστηρίζουν τα εργαλεία συλλογής [19]. Επιπλέον, πρέπει να πραγματοποιηθεί περαιτέρω ανάπτυξη του κώδικα του Πράκτορα και ενσωμάτωσή του σε Custom Controller. Καταληκτικά, θα πρέπει να δοκιμαστούν νέα Q Networks για τον Πράκτορα π.χ. Bidirectional LSTMs και να γίνει εκ νέου επικύρωση για την ευαισθησία του Πράκτορα στις διαφορετικές παραμέτρους και αρχιτεκτονικές.

Chapter 1

Thesis Outline

Chapter 2: Theoretical Background, provides the necessary background knowledge for the technical background of the thesis. Starting from elementary technologies such as Containers and Docker we proceed to elaborate on the Kubernetes architecture and further into monitoring tools for metrics collection and most importantly the default scaling policies.

Chapter 3: Machine Learning & Reinforcement Learning, gives an introduction Machine Learning and dives deeper into Reinforcement Learning which is used in this thesis. Also, an overview of the machine learning methods that are being used in scheduling is given. A focus on Deep Q Learning is noted.

Chapter 4: Exploratory Data Analysis, is dedicated to the exploratory data analysis on the CPU usage data acquired. The data is analysed and notes are made on the patterns of the usage and the metrics produced. Then, the time series prediction models are developed with the use of LSTM models. The results of training and validating these models are predicted and compared.

Chapter 5: Model Development, is dedicated to the reinforcement agent that is developed on the previous results. The choices for the agent's environment, rewards, actions and parameters are explained. The training and validation are presented as well.

Chapter 6: Future Work and Extensions, contains our conclusion, summarizing our findings and providing an outlook into the future work.

Chapter 2

Theoretical Background

2.1 Containers

A container image is defined as “a ready-to-run software package containing everything needed to run an application: the code and any run-time it requires, application and system libraries, and default values for any essential settings” [20]. It has been developed and popularized in the latest years to serve the needs of modern applications that are being based on big data centers and severs. The significant change in needs of computation, resilience and efficiency has mitigated the focus of engineers towards Cloud Computing [21], which requires a type of virtualization that container images satisfy . Container images are lightweight and isolate the software that runs inside from its surroundings. They contain their code, libraries, system tools and settings separately and can be run as a stand-alone software. These characteristics have popularized containers as they minimize conflicts and can be efficient for software development on the same infrastructure.

2.2 Docker

Docker [22] is a specific containerization platform that popularized and standardized the use of containers in the software industry. Docker was popularized because it provided an easy platform to create, deploy and run applications. It introduced a standard format for container images and a set of tools for building, distributing, and running containers, making it an industry standard. Through the Docker Engine component, one can run and manage containers on the host system, utilising the same available hardware. By efficiently utilizing available hardware,the performance is significantly boosted and the application size is reduced. While Docker played a pivotal role in the containerization revolution, the technology has evolved, and many more platforms are now available.

2.3 Kubernetes

Kubernetes [23], also known as K8s, is a portable, extensible, open source platform for automating deployment, scaling, and management of containerized applications. The most widely known run-time in these applications is Docker, however more are supported such as cri-o [24] and containerd [25]. Kubernetes originated as an internal project by Google but announced in 2014 and released in 2015 as an open source project. It was easily popularized as it offered the decoupling the application containers from the details of the computing systems on which they are executed, through orchestration and management of containers. K8s also became an industry-standard as it had many benefits in large scale production environments such as

- Orchestration of containers among multiple hosts

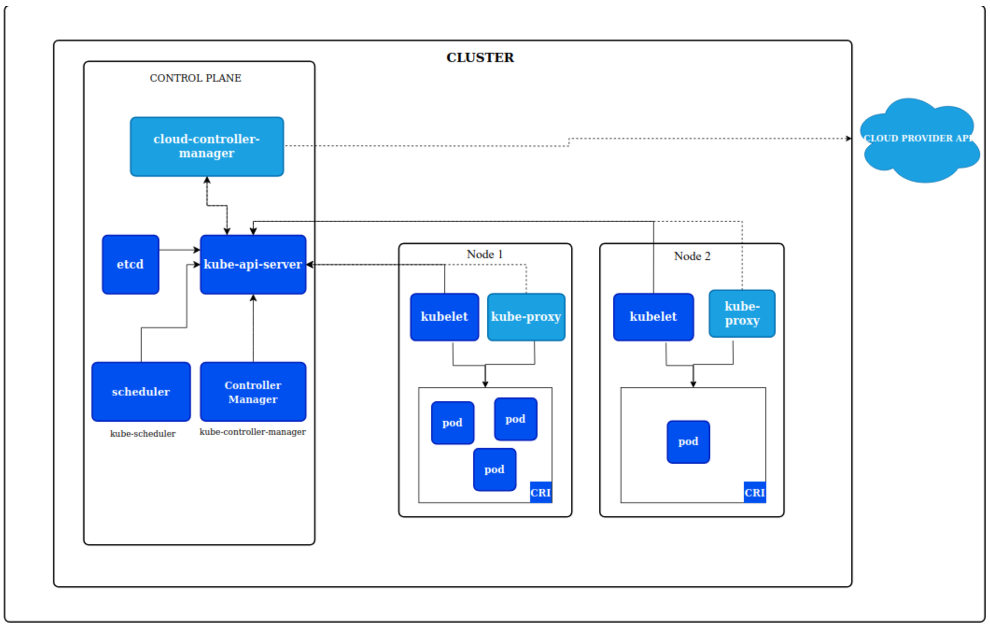


Figure 2.1. *The Kubernetes Cluster Architecture*

- Resource utilization and load balancing
- Automation of operation tasks and application deployments
- Scalability of services, applications and clusters
- Self-healing via health checks, auto-start, auto replication and auto-placement
- Declarative Configuration
- Abstraction of infrastructure

Upon deploying Kubernetes, a cluster is formed which consists of worker machines, called Nodes, that run containerized applications. The application workload is distributed between the worker nodes in "building blocks" called Pods. The control plane manages the worker nodes and the Pods in the cluster. The Kubernetes architecture outline can be seen in Figure 2.1 [26] and will be explained in detail below.

We can examine the Kubernetes architecture in the following structural sections:

2.3.1 Cluster Architecture

1. Nodes

   All servers that are participating in the K8s cluster can be referred to as "Nodes". Nodes can be generally categorized into one or multiple "master" nodes, also known as "control plane" and usually multiple "worker" nodes. Worker nodes run the actual workloads that are appointed to them and are networked to communicate with the master components. They require few configurations for networking, only joining the cluster in a private network manner and they run independently from one another. In each node there are 2 important components: the kubelet, an on-machine agent and the cadvisor that analyzes the resource usage of each node and monitors different performance characteristics.

2. Pods

A pod serves as the fundamental component of Kubernetes, constituting the smallest and most basic unit within the Kubernetes object model that can be instantiated or launched. It signifies an active process within the cluster, encompassing an application container (or, in specific scenarios, multiple containers), storage assets, a distinct network IP, and configurations that dictate the behavior of these containers. Essentially, a Pod represents a deployment entity that can be a solitary instance of an application in Kubernetes. This instance can comprise either a lone container or a compact group of containers closely interconnected and jointly utilizing resources.

3. Services

   In a cluster environment, Pods can be terminated or restarted unexpectedly, their IPs are constantly changing and therefore not reliable. Services provide a stable IP address to connect to Pods. Each Service is associated with a group of Pods. When traffic reaches the Service, it is redirected to the back-end Pods accordingly.

   All in all, Services serve as an abstraction layer that provides a stable network endpoint for a set of pods, allowing them to be accessed consistently despite their dynamic nature. Kubernetes services are categorized into various types, each tailored to specific use cases: ClusterIP, NodePort, LoadBalancer and more.

   - ClusterIP that enable internal network communication within the cluster
   - NodePort that expose applications externally by binding a port on each node's IP
   - LoadBalancer that integrate with cloud providers to distribute external traffic with public IP

   These service types collectively empower Kubernetes to efficiently manage networking and ensure applications are accessible and resilient within the cluster.

4. Deployments

   Deployments are the standard way of running applications in a cluster, and they represent the final stage where the application is running on the cluster. In a certain Deployment any Pod is replacable and thus can be changed at any moment, ensuring the good management of the application.

2.3.2 Kubernetes Components

Control Plane

1. API Server

   The API Server or kube-apiserver [27] is the front end of the control plane node and is the component that communicates with the Kubernetes API. The Kubernetes (REST) API manages the configuration of pods, services and other components and validates the data.

2. Controller Manager

   The Controller Manager or kube-controller-manager is a daemon responsible for the controller's smooth operation. The controller is overlooking the state of the cluster and making the necessary actions to reach a stable wanted state through a control loop that is non-terminating and monitors the apiserver.

3. Scheduler

The default Scheduler in K8s is responsible for selecting the target Node for a Pod. It examines the available Nodes and filters them on a multi-criteria basis such as the available memory and then assigns a score value calculated on a set of rules [28][29][30]. The filtering and scoring modules are defining the suitability for each Node in order to run the desirable Pod. Going into more detail about the filtering and scoring we can examine each variable:

- queueSort → sorts all the Pods in the queued state. A queue sort plugin essentially provides a Less(Pod1, Pod2) function.
- preFilter → pre-processes the information Pods and checks certain conditions that the Pod must satisfy
- filter → filters the Nodes that are unsuitable to run the Pod. The filters will be looped across all Nodes and if a Node throws an infeasibility error then is aborted from the rest of the loop.
- postFilter → is called if only none Node was feasible for the Pod. A typical PostFilter implementation is preemption, which tries to make the pod schedulable by preempting other Pods.
- preScore → generates a shareable state for scoring
- score → ranks the Nodes that have been deemed suitable after filtering. A well defined range of integers representing the minimum and maximum scores is returned which passed through a normalising function before the scheduler combines node scores from all plugins according to the configured plugin weights
- reserve → notifies when resources on a node are being reserved and unreserved for a given Pod. The reserve is split into two methods, called Reserve and Unreserve.
- permit → does one of three things: "approve", "deny", "wait". The "approve" method approves the Pod and sends it for binding to a Node. The "deny" denies the Pods and returns it to the queue. The "wait" keeps the Pod in an internal "waiting" Pods list, and the binding cycle of this Pod starts but directly blocks until it gets approved. If a timeout occurs, wait becomes deny and the Pod is returned to the scheduling queue, triggering the Unreserve phase.
- preBind → pre-processes the networks requirements for Pod binding
- bind → binds the Pod to the Node after all preBind plugins are completed
- postBind → is called after a successful bind to signal the end of the binding cycle
- multiPoint → sets the plugin preferation based on the user needs

In the scheduling terms above, the reason we refer to these containers as Pods is that containers, although they run on the nodes, they are not themselves the schedulable entities. The smallest schedulable entity is a Pod, which provides an environment where one or more containers run. However, as is usually the case, Pods are used as a wrapper for single containers. Therefore, we consider the Pod to be of the most common type, the singleton Pod [31]

4. etcd

The etcd space [32], [33] is the storage where all cluster data is stored. The "d" in etcd stands for "distributed" and is meant to showcase that the distributed manner of the K8s needs to a data storage system that provides a consistent and trustworthy data source with all the information about the clusters. To be more precise, etcd contains all the configuration data, state data, and metadata for all the clusters, pods, services etc.

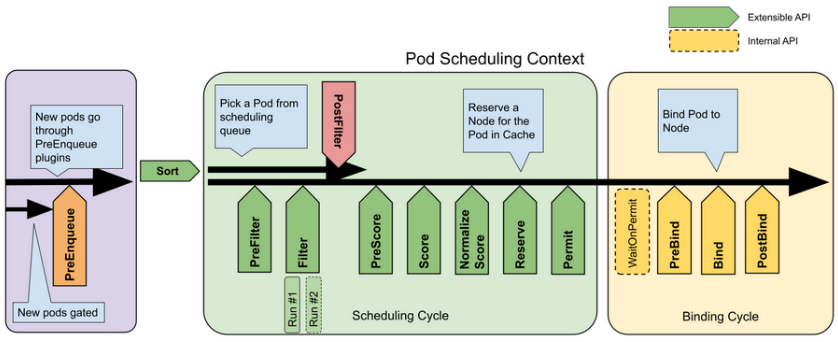


Figure 2.2. *Scheduling Framework*

Worker Nodes

1. kubelet

   The kubelet component is a part of every node in a cluster. In node-level it manages the communication between the control plane and the individual node, ensuring the effective deployment and execution of containerized applications throughout the entire cluster.

2. kubeproxy

   The kube-proxy component is also integrated in node-level. It is responsible for monitoring the changes in the deployed Services and their endpoints.

2.3.3 Kubernetes Networking

The Kubernetes Networking [34] is a complex multi-level network that ensures consistency between Pods, Nodes, Services and endpoints in a given cluster. Networking in a cluster is a challenge since many components change constantly, Services are deployed, Pods are failing, and Scaling is often needed so the IPs constantly change. The basic communication levels are showcased below:

1. Container-to-Container & Pod-to-Pod communication

   In any K8s cluster, containers communicate with each other within a Pod through a Pod network "namespace." This namespace gives each Pod its own network environment that's separate from the rest of the system. Inside a Pod, all the containers share the same IP address and ports. They can communicate with each other using "localhost" because they're all part of this same network environment. The Pod-to-Pod communication is showcased in Figure 2.3

   For Pod communication it is crucial to note that every node has a designated CIDR range of IPs for Pods. This ensures that every Pod receives a unique IP address that other Pods in the cluster can see. When a new Pod is created, the IP addresses never overlap. Unlike container-to-container networking, Pod-to-Pod communication happens using real IPs, whether you deploy the Pod on the same node or a different node in the cluster. As seen in Figure 2.3 for the exact communication a connection between the Pod namespace and the Root namespace is needed, which is possible through a virtual network bridge connects these virtual interfaces, allowing traffic to flow between them using the Address Resolution Protocol (ARP).

2. Pod to Service

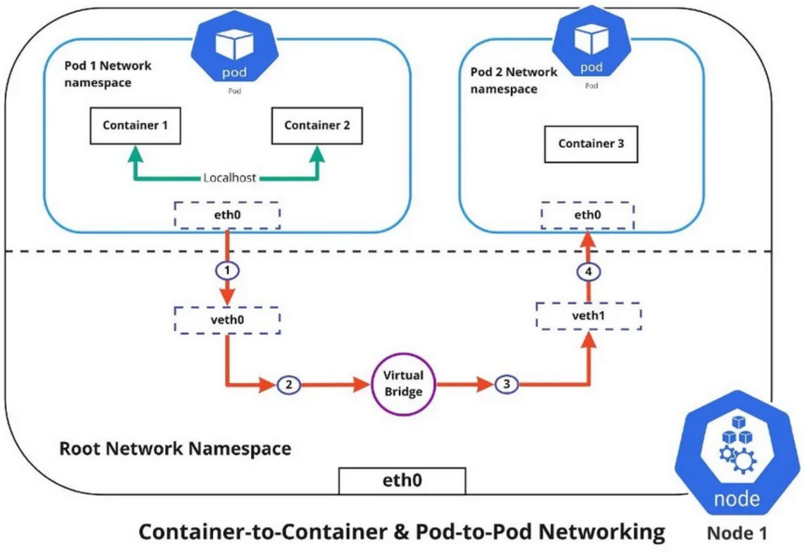


Figure 2.3. *Container-to-container & Pod-to-Pod communication*

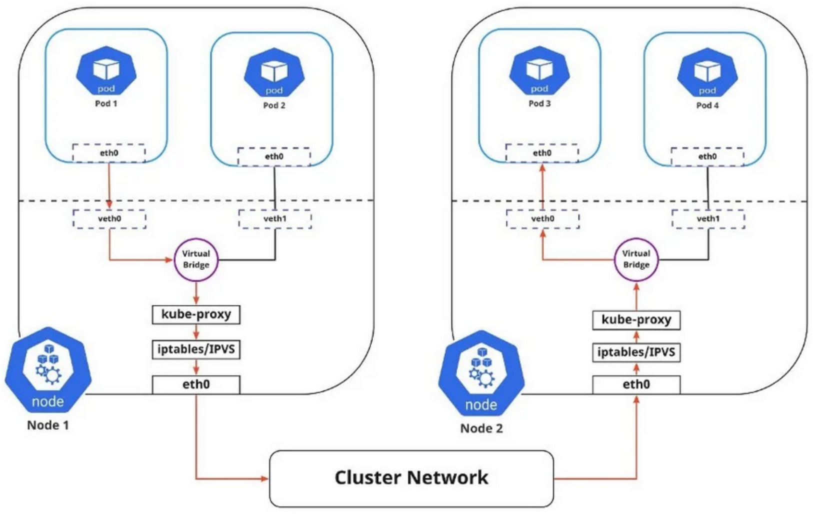


Figure 2.4. *Pod-to-Service communication*

The Pod-to-Service communication is In Kubernetes, the Service function addresses communication challenges in the following ways:

- Static Virtual IP: Services assign a fixed virtual IP address that acts as a consistent entry point for connecting to any associated back-end Pods. This virtual IP simplifies the process of reaching the right destination.
- Load-Balancing: Traffic directed to the virtual IP of a Service is efficiently load-balanced across the group of back-end Pods. This ensures even distribution of requests and prevents overloading of a specific Pod.
- IP Tracking: Services keep track of the IP addresses of Pods. Even if a Pod's IP address changes (which can happen due to various reasons), clients can still reliably connect to the Pod because they interact with the stable virtual IP address provided by the Service.

In-cluster load balancing is handled in two ways:

- IPTABLES:
  In this mode, kube-proxy continuously monitors the API Server for changes. For each new Service, it configures iptables rules to capture traffic to the Service's clusterIP and port, then redirects it to one of the back-end Pods. Pod selection is random. This method is reliable and has lower system overhead as it leverages Linux Netfilter for efficient traffic management.
- IPVS: IPVS, built on Netfilter, implements transport-layer load balancing. It operates in the kernel space and offers lower-latency, higher-throughput, and better performance compared to iptables. Kube-proxy in IPVS mode efficiently directs traffic within the cluster.

This system ensures that requests to Services are efficiently managed and load-balanced across the relevant Pods, maintaining the reliability and performance of the cluster as seen in Figure 2.4

2.3.4 Kubernetes Scaling

One of the most important features of Kubernetes is the scaling mechanisms. Kubernetes scaling helps manage and optimize the allocation of computing resources. In the dynamic realm of modern cloud-native environments, the ability to scale applications both horizontally and vertically has become crucial. The scaling mechanisms can be categorised into the following concepts and strategies:

Scaling Concepts

1. Horizontal Scaling, is the scaling behind which you can adjust metrics like CPU usage or memory usage. Scaling horizontally means meaning modifying the resources of an existing cluster by adding new nodes or pods to it (Horizontal Pod Autoscaler).

2. Vertical Scaling, is the scaling for adjusting metrics such as resource requests and limits per pod. In most cases, this means creating an entirely new node pool using machines that have different hardware configurations (Vertical Pod Autoscaler).

3. Cluster Scaling, is the scaling concept where the cluster itself is resized to accomodate the varying workloads. A common application of cluster scaling is through an interface with the chosen cloud provider so that it can request and deallocate nodes seamlessly as needed.

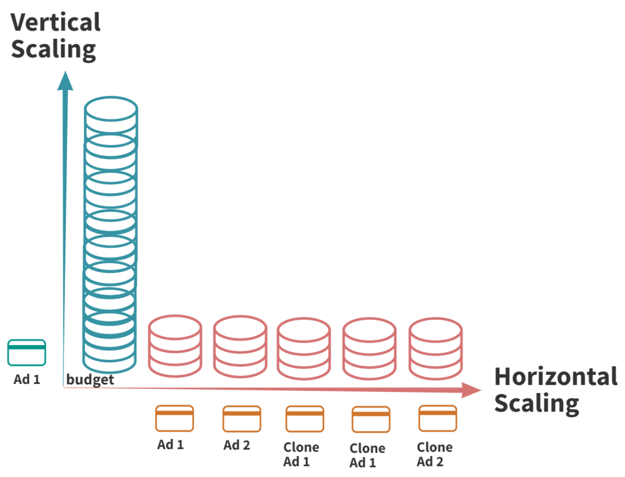


Figure 2.5. *Vertical vs Horizontal Scaling Strategies*

Scaling Strategies

1. Horizontal Pod Autoscaler

   The Horizontal Pod Autoscaler (HPA) [35] [36] uses and autoscaling algorithm to scale the number of pods a replication controller, deployment or replica set based on observed CPU utilization. The metrics, as mentioned are usually CPU and memory usage but it must be noted that upon customisation it can also observe and make decisions based on application-provided metrics. The HPA is implemented as a resource of the Kubernetes API and as a controller, where the resource constantly checks the metrics collected from the metrics-server and provides them to the controller who periodically makes decisions. These decisions are made to match the observed CPU utilization to the desired defined by the user and are by definition threshold-based. In case of custom metrics, the cluster needs to be linked to a time-series database holding the desired metrics from where the information will be used for the scaling purposes.

   As this diploma thesis is focused on autoscaling by adding or substracting pods, we will go into more detail about how the HPA works and its' limitations.

   As we mentioned, by default the HPA takes into account CPU utilization measurements. The HPA periodically fetches monitoring data from the system, and takes a decision on how many Pods the cluster should have. This period is called a scaling interval.

   We define :

   (a) c_{min} and c_{max} the minimum and maximum allowed Pods
   (b) $\hat{u}\epsilon[0,1]$is the user desired CPU over all Pods
   (c) scaling interval is a R^{+}-long time window, the end of which signals the scalind decision
   (d) $s\epsilon[0,1]$ is the scaling tolerance
   (e) d is the downscale stabilization which can represent the interval between two downscales

   And we get the limits:

   $$\left|\frac{u_i}{\hat{u}} - 1\right| > s \tag{2.1}$$

   which represents the scaling decision in an interval i and the number of Pods in an interval $i+1$ is recursively yielded as:

   $$c^{*}_{i+1} = \left\lceil c_i \frac{u_i}{\hat{u}} \right\rceil \tag{2.2}$$

With the limits, we can describe the pod count as

$$c^{'}_{i+1} = \begin{cases} c^{*}_{i+1} \text{ if } c_{min} \leq c^{*}_{i+1} \leq c_{max} \\ c_{min} \text{ if } c^{*}_{i+1} < c_{min} \\ c_{max} \text{ if } c^{*}_{i+1} > c_{max} \end{cases} \tag{2.3}$$

However, as mentioned, HPA will not perform downscale operation if there was another one in the previous time window of length d.

$$c_{i+1} = \begin{cases} c_i \text{ if } c^{'}_{i+1} < c_i \text{ and } d_{i+1} = 1 \\ c^{'}_{i+1} \text{ else} \end{cases} \tag{2.4}$$

The higher the d value, the more resources the application uses, because it scales out without such stabilization, but scaling in is slower compared to applying a low d

2. Vertical Pod Autoscaler

   The Vertical Pod Autoscaler (VPA) [37] is different from the HPA in many ways as its' goal is to assign more resources, usually memory or CPU internally, namely in the Pods that are already running the workload. The challenge of this mechanism has put the VPA in the beta version of Kubernetes for years, as the main issue it can currently only scale by terminating and restarting the application Pods [38], which is inefficient and many times unwanted by the user. In the last years, attempts to solve these challenges have been made as extensions to Kubernetes, but have not been officially integrated. As the interest is much as the development is still active, an official version of the VPA is expected in the future

3. Custom Scaling Policies and Controllers

   Since Kubernetes is an open-source framework, customisation and most definitely scaling policies are being customised commonly. Some of the common strategies used for custom schedulers are: Threshold-based [39] [40], Queuing model-based [1] [2] [3], Control-theory based [4], Reinforcement Learning based [6] [7] [8] , Performance prediction based [5] and more.

   In this thesis we are focused on Reinforcement Learning methods. More on this will be elaborated in Section 3.4.

   In order to extend and customize the behavior of the Kubernetes control plane to to manage resources according to specific requirements or policies custom controllers can be implemented. Custom controllers are implemented as independent processes that interact with the Kubernetes API server. The custom controller in our case can serve as an intermediary between our Reinforcement Learning agent and the Kubernetes control plane, in the sense that it can bridge the gap between the agent's high-level scaling decisions and the actual Kubernetes operations required to modify the cluster.

   In that sense, we are proposing a Node autoscaler, a custom controller that may manage the scaling of nodes based on the RL agent's decisions. It can create or delete nodes in the cluster to accommodate workload changes.

2.4 Monitoring Tools

When managing a K8s cluster, it is vitally important to have monitoring tools deployed inside the cluster with the purpose of exporting metrics, alerts and monitoring the load, behavior and usage of the cluster. One of the most common monitoring tools is Prometheus [19].

2.4.1 Prometheus

The Prometheus monitoring tool is a powerful open source monitoring and alerting platform released in 2016. It has become widely popular as it provides a multi-dimension data model by collecting time-series data at a parameterised set of intervals. The time series data are then stored locally in key-value pairs that make processing easier. Prometheus provides a Dashboard in an HTTP endpoint from inside the K8s cluster to visualize and to collect the real-time metrics in a time series database in an understandable way. Furthermore, it utilizes PromQL [41], a easy SQL-like language that enables flexible queries and real-time alerting. The fundamental components of Prometheus are listed below:

- Prometheus Server $\rightarrow$ scrapes the metrics and record the numeric time series
- Client Libraries $\rightarrow$ matches the application language
- Alertmanager $\rightarrow$ manages the alert notifications and forwards them (email,notifications,etc)
- Exporters $\rightarrow$ for disseminating existing metrics from the third-party systems
- Grafana $\rightarrow$ connects to the Prometheus endpoint for graphics visualisations

Chapter 3

Machine Learning & Reinforcement Learning

3.1 Machine Learning

Machine Learning (ML) is a now widely known field of Computer Science, vastly popularized in the latest years due its significant breakthroughs in multiple applications and research topics. The ML field consists of different kinds of models that are being developed with a variety of algorithms that automate predictions, decision making, labeling and more. To be more presice, the machine learning algorithms create a mathematical model that is based on data - either real or synthetic - that are commonly called "training data". The algorithm then "trains" on the data in various ways, such that the underlying patters and mathematical curves that describe them are estimated. The convergence process, the evaluation and the way that each model learns can vary a lot on the type of the model and the type of the data. The three basic categories of Machine Learning are:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

Each of them is described below:

3.1.1 Supervised Learning

Supervised learning is a machine learning paradigm where algorithms learn from labeled data. It involves a clear relationship between input data and corresponding output, allowing the model to make predictions or classifications. In this approach, the algorithm is trained on a dataset where each example is paired with its correct answer. It learns to generalize patterns and relationships within the data, making it capable of making accurate predictions on new, unseen data. Common Supervised Learning tasks are classification tasks, Regression Tasks, Bayes Classifier, Linear Regression, Support Vector Machines, Desicion Trees

3.1.2 Unsupervised Learning

Unsupervised Learning: Unsupervised learning, in contrast, operates without labeled data or explicit output. It focuses on discovering patterns, structures, or relationships within a dataset. Algorithms in unsupervised learning aim to group or cluster data based on inherent similarities or to reduce the dimensionality of data. Common techniques include clustering and dimensionality reduction. Unsupervised learning is often used for tasks like customer segmentation, anomaly detection, and data compression.

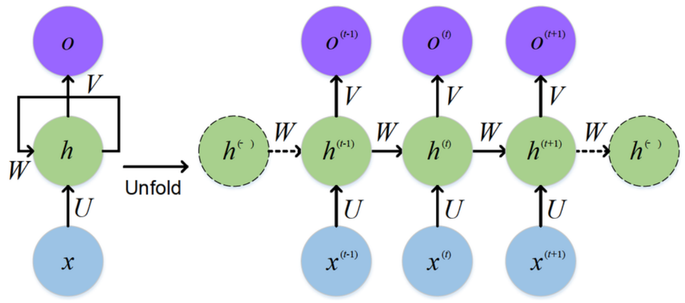


Figure 3.1. *The standard RNN and the unfolded version*

3.1.3 Reinforcement Learning

Reinforcement learning is a learning paradigm where agents interact with an environment and learn to make sequential decisions to maximize a cumulative reward. It is akin to training a virtual agent to navigate and make choices in an environment. The agent receives feedback in the form of rewards or penalties based on its actions, allowing it to adapt and improve its decision-making over time. Reinforcement learning is used in applications such as game playing, autonomous robotics, and recommendation systems, where an agent learns to optimize its actions based on feedback from the environment. Reinforcement Learning will be explained in more detail below.

3.2 Recurrent Neural Networks

In order to get a deep understanding of Long-Short Term Memory Models, their use for time series prediction and their integration into a reinforcement learning agent, we have to introduce the Recurrent Neural Networks (RNNs). RNNs are a greater family of models, including LSTMS and GRUs (Gated Recurrent Unit) that are based on the same basic concept: the reccurency of the hidden layer. They are feed-forward neural networks that have integrated memory, designed to handle sequential data such as natural language texts or time series.

The RNN architecture can been seen in Figure 3.1, in both raveled and unraveled form that showcases their novelty. The input sequence is represented as $x^{(t)}$ and the output sequence as $o^{(t)}$. The novelty relies on the information flow, where in contrast to standard to neural network, the inputs and outputs are not independent of one another thus creating a memory of the previous items in the sequence. The component that remembers the previous sequences is the internal state of the RNN which is called the hidden state of the unit. This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past. The equations that describe the above relations can be summed as:

$$a^{(t)} = b_1 + Wh^{(t-1)} + Ux^{(t)} \tag{3.1}$$

$$h^{(t)} = \sigma(a^{(t)}) \tag{3.2}$$

$$o^{(t)} = b_2 + Vh^(t) \tag{3.3}$$

where b_1 and b_2 are bias vectors, U, V, W are the weighting matrices of the input-to-hidden connection, hidden-to-output connection and hidden-to-hidden connection respectively. As σ we denote the activation function that can usually be an non linear function, for example a sigmoid $\sigma(x) = (1 + e^{-x})^{-1}$

3.3 Long-Short Term Memory Models

The Long-Short Term Memory Models, as said are a part of the RNN family of models, and where introduced by Hochreiter and Schmidhuber [42] to tackle the gradient vanishing problem and exploding problem in RNNs [43]. These problems are solved by upgrading the RNN cell, in particular replacing the hidden layer with a memory cell c, and introducing three gates, the forget gate f, the input gate i and the output gate o as seen in Figure 3.3. The forget gate is used to forget the current cell, the input gate to read input input and the output gate to output the new cell value. Each of these gates effects to one layer. If the gate is 1, LSTM keeps the value of corresponding layer, and if the gate is 0, it sets this value to zero. The definitions of all gates, cell update and output at time t are given as follows:

$$i^{(t)} = \sigma(b_i + U_i x^{(t)} + W_i h^{(t-1)}) \tag{3.4}$$

$$f^{(t)} = \sigma(b_f + U_f x^{(t)} + W_f h^{(t-1)} \tag{3.5}$$

$$o^{(t)} = \sigma(b_o + U_o x^{(t)} + W_o h^{(t-1)}) \tag{3.6}$$

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \sigma(b + U x^{(t)} + W h^{(t-1)}) \tag{3.7}$$

$$h^{(t)} = o^{(t)} \odot tanh(c^{(t)}) \tag{3.8}$$

The aforementioned mechanisms allow LSTMs (and GRUs [44]) to selectively update and forget information from the past. LSTMs, beside the reccurency of their nodes also hold an internal state that is used as a long term memory space. This feature allows for information retrieval over many time steps. The results of the calculations of the input, output and internal state are not only used to produce the output of the node but also to update the internal cell state that represents the long-term memory.

As seen in Figure 3.3, and through the equations we can deduct that the operations and activations in the relationships between the gates are used to selectively recall or discard information. In particular, the forget gate computes the ratio of the information that should be retained from the previous state, the input gate allows the influx of new information and the output gate determines how much impact the current cell and hidden states have on giving an output on the current time step.

As is evident, the LSTM models are designed to train on sequences and learn patterns that are in the short realm of history but also on deeper patterns that get preserved through time. For that reason exactly they have been used extensively in applications of time series prediction.

3.3.1 Applications in Time Series Prediction

Multiple surveys [9] [10] have studied the applications of LSTMs in a wide range of domains focusing on time series forecasting. Applications include energy consumption forecasting [11] [12], stock prices forecasting [13] [14] and more. The field of LSTM for forecasting was spawned by mainly the finance field, where accurate predictions of stock prices can yield a large sum of money to traders. Since then, the applications have broadened as recent advances in deep learning technologies have led to a significant increase in the accuracy and performance of forecasting all kinds of time series. As these models can capture complex temporal patterns and dependencies, we explore the forecasting of CPU usage in order to efficiently scale a computing system with heavy usage.

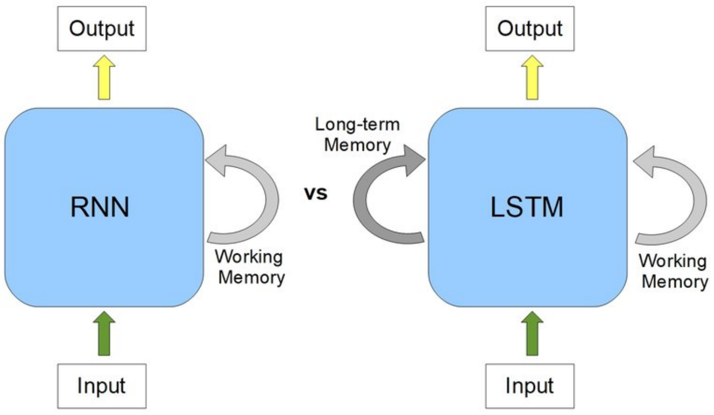


Figure 3.2. *Abstract differences between RNN & LSTM*

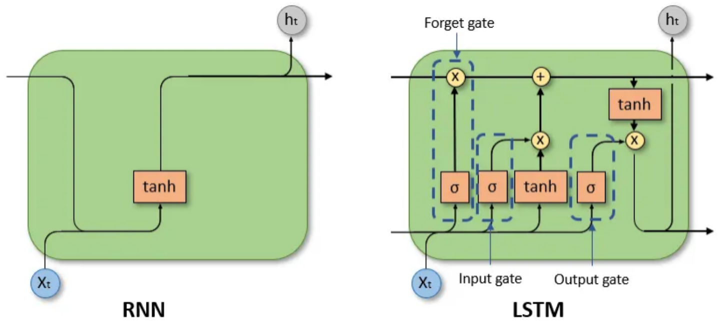


Figure 3.3. *Structual differences between RNN & LSTM*

3.4 Reinforcement Learning

As previously mentioned, Reinforcement Learning is a field of Machine Learning. The idea behind it is that an Agent is taking Actions in an Environment which returns rewards, with the ultimate goal of maximizing the sum of total rewards that are returned for it's actions. The Environment is modeled as a Markov Decision Process (MDP), whose solutions are often being met in dynamic programming. The fundamental difference between dynamic programming and Reinforcement Learning is that the latter is utilising large MDPs, where classing dynamic programming methods could not apply. Also, RL is used where the P-function or the Reward function are not known.

3.4.1 Q-Learning

Q-Learning is one of the most fundamental Reinforcement Learning Techniques, which can be used to learn an optimal policy $\pi*$ where dynamic programming cannot apply.

We first define a new function $Q(s,a)$, also known as Q Value function, in which the Agent starts from an initial state s, selects and action a and the choice is considered optimal. Thus, the function $Q(s,a)$ is computes the quality of the (s,a) pair as follows:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma \max a' Q^*(s',a')) \tag{3.9}$$

According to the previous equation, when an Agent is on the state s and selects the action a, that leads him to the state s' and the subsequent action a' that maximizes the Q-Value.

Iterating through the values we get the Q-Value Iteration so that we can compute the array of Q-Values as follows:

$$Q^*_{k+1}(s,a) = \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma \max a' Q_k(s',a')) \tag{3.10}$$

The solution to the above is crucial so that all the possible transitions are known from one state s to another state s', as well as the possibilities of the respective occurrences. However, in many situations these values are not known ad-hoc, but are computed using the expected value as follows:

$$Q^*_{k+1}(s,a) = E_{s'\sim P(s'|s,a)}[(R(s,a,s') + \gamma \max a' Q_k(s',a')] \tag{3.11}$$

Subsequently, because the Agent does not know the possibility of the transition to s', he will attain knowledge from the constant interaction with the Environment. In that way, the Q-Values will be computed. The target is estimated after every interaction as

$$target(s') = R(s,a,s') + \gamma \max a' Q_k(s',a') \tag{3.12}$$

where the $R(s,a,s')$ value is the reward for the estimation, which is then considered optimal and proceeds to compute the Q-Values as

$$Q^*_{k+1}(s,a) = (1-a)Q_k(s,a) + a[target(s')] \tag{3.13}$$

Summarizing the above, the algorithm can be summarized as:

Algorithm 3.1: *Q-Learning Algorithm*

```
Start with $Q_0(s,a)$ for all $s, a$
Get initial state $s$
for k in 1,2,... until convergence do:
    Sample action a: get next state $s'$
    if $s'$ is terminal then
        $target = R(s,a,s')$
        Sample new initial state s'
    else
        $target = R(s,a,s') + \gamma \max a' Q_k(s',a')$
    end if
    $Q^*_{k+1}(s,a) \leftarrow (1-a)Q_k(s,a) + a[target(s')]$
    $s' \leftarrow s$
end for
```

As the algorithm shows, the agent needs to store the Q-Values in a table to be used in next iterations. That means that the state space is vast and needs a large memory space to be store which also makes the algorithm significantly less efficient and impractical, often times impossible to use. Furthermore, the table needs to be initialized correctly as the non explored fields would confuse the agent.

The solutions to the above impracticalities rely on the use of neural networks and the use of quantization. As the state space needs to be significantly reduced in size, quantization helps to store the Q-Values in a practical way in order for the algorithm to converge more easily. The use of neural networks on the other hand, helps on approximating the $Q(s,a)$ function for every (s,a) pair, so that there are no unexpected values in the unexplored space. The network that approximates the above will be explained below

Q Network

Instead of iteratively changing the Q table with new Q-Values we can define the $Q(s, a)$ function as:

$$target(s') = R(s, a, s') + \gamma maxa' Q_{\theta k}(s', a') \tag{3.14}$$

Now, using Gradient Descent the θ values can be iteratively updated instead of the table using the equation:

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta E_{s' \sim P(s'|s,a)}[(Q_{\theta(s,a)} - target(s'))^2]|_{\theta=\theta k}] \tag{3.15}$$

The algorithm is now shaped as:

Algorithm 3.2: *Q-Network Algorithm*

```
Start with $Q_0(s, a)$ for all $s, a$
Get initial state $s$
**for** k in 1,2,... until convergence **do**:
    Sample action a: get next state $s'$
    **if** $s'$ is terminal **then**
        $target = R(s, a, s')$
        Sample new initial state s'
    **else**
        $target = R(s, a, s') + \gamma \max a' Q_k(s', a')$
    **end if**
    $\theta_{k+1} = \theta_k - \alpha \nabla_\theta E_{s' \sim P(s'|s,a)}[(Q_{\theta(s,a)} - target(s'))^2]|_{\theta=\theta k}]$
    $s' \leftarrow s$
**end for**
```

The agent chooses the action a with the biggest possible value of $Q(s, a)$. Every action a in a current state s is considered optimal.

However, there are problems to be faced with the above algorithm, namely:

- As the θ parameters are updated for the target approximation, the Q-Values for the state s, where the action a was taken, are updated in accordance to the other actions in the same state. That means that the target is changing. That can lead to problems in the convergence, as traditional neural networks have a well defined target
- As the agent's actions are inputs to the computation of the next state, the inputs of the neural networks are directly correlated. That can also lead to convergence problems as the network may converge in local maxima

In order to overcome the above and create a Q Learning agent with Deep Learning we use Deep Q-Learning as explained below.

3.4.2 Deep Q-Learning

The Deep Q-Network (DQN) was first introduced by Mnih et al. [15] and proposed a groundbreaking algorithm that combined Q-Learning with deep neural networks for reinforcement learning tasks. It became widely popular for its impressive ability to learn optimal policies in complex environments and many papers have been published since. In a Deep Q Network two more components are introduced: a second neural network called $TargetNetworkQ'$ and new inputs in form of a shuffle memory called $ExprerienceReplay$

Target Network

The Target Network tackles the moving target problem that traditional Q-Learning poses. A new neural Network Q' is used, an exact copy of the prediction network Q with parameters θ'. The target is now computed by using the Target Network and applying the Gradient Descent algorithm with parameters θ between the prediction of the prediction network Q with parameters θ and the target acquired by the target network Q' with parameters θ'. The equations can now be described as:

$$target(s') = R(s, a, s') + \gamma \max a' Q'(s', a', \theta') \tag{3.16}$$

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta [(Q_{\theta(s,a)} - target(s'))^2]|_{\theta=\theta k}] \tag{3.17}$$

We also define a frequency C which describes a time period after which the θ' parameters of the Target Network are updated from the θ parameters of the Prediction Network.

Experience Replay Buffer

Since the moving target problem is now solved, we proceed to explain the solution to the input correlation for the neural network. This is solved by introducing new experiences for the agent, independent from one another, which will include the outcomes from each action of the agent. Every experience will be comprised from a state s, the taken action a from that state, the new state s' which the agent was lead to and the reward r that was given for that action to the new state. Thus, the experiences will be pairs of $< s, a, r, s' >$.

There experiences will be stored in an array and in each iteration will be randomly sampled in batches to train the neural network. In parallel, since new experiences will be added to the replay buffer in every iteration, the old ones will be removed.

Integrating the above, the DQN algorithm is shaped as follows:

Algorithm 3.3: *Deep Q-Network Algorithm with Experience Replay*

```
Initialize replay memory D to capacity N
Initialize action-value Q with random weights $\theta$
Initialize target-action-value function $Q'$ with weights $\theta' = \theta$
**for** episode = 1, M **do**:
    Initialize sequence $s_1 = \{x_1\}$ and pre-processed sequence $\phi_1 = \phi(s_1)$
    **for** $t = 1, T$ **do**
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = aQ(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r$ and image $x_{t_1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and pre-process $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random mini-batch of transitions $\phi_j, a_j, r_j, \phi_{j+1}$ from $D$
        **if** episode terminates at step $j + 1$ **then**
            Set $y_i = r_j$
        **else**
            Set $y_i = r_j + \gamma \max a' Q'(\phi_{j+1}, a', \theta')$
        **end if**
        Perform a gradient descent step on $((y_j - Q(\phi_j, a_j, \theta))^2$ with respect to the network
parameters $\theta$
        Every C steps reset $Q' = Q$
    **end for**
**end for**
```

Exploration vs Exploitation

On terms of optimizing even more the Agent's performance and ensuring better convergence we give the Agent an ϵ probability which dictates whether the Agent takes a completely random action. That strategy, also referred to as "Exploration" gives the Agent the ability to experience states that may have never been accessible through the standard knowledge path. The ϵ quantity is usually chosen so that it is of great significance in the first iterations and less significant (close to 0) as the episodes go by. [45]

3.4.3 Applications in Scaling

In recent times, significant attention has been directed toward the concept of automatic scaling. This has garnered interest due to the appeal of having a flexible system that can effectively manage peak traffic loads. Balla et al. [46] have highlighted a major drawback in contemporary Kubernetes auto-scalers, which stems from their reliance on fixed measurements that fail to adjust to the current usage patterns of the system. On the end of Chapter 2 we touched on different methods that are commonly used for custom scaling policies. Toka et al. [36] stress the need of a more dynamic approach to scaling, as the scaling provided in Kubernetes is often reactive rather than proactive. In their study, they propose an LSTM solution, however on data that were not collected from a real production server.

In that essence and to be more clear, a proactive Agent is different from a reactive Agent in a very crucial feature. The reactive Agents make the decision depending on the current state while a proactive Agent is also based on historical data. In the next chapters, we propose an LSTM to learn the patterns from the historical sequential data and then develop the DQN Agent that will be proactive.

Chapter 4

Exploratory Data Analysis

We proceed our study with an Exploratory Data Analysis (EDA). The EDA is a standard practice amongst data scientists in order to have a better understanding of the dataset. It involves practices of statistical computation, observation and visualization amongst others, as it differs for each type of data

The dataset [16] used in this thesis is publicly available [17] on Github and consists of CPU usage metrics collected over 15 days, between the 10th of January 2022 and the 25th of January 2022.

The cluster was running in a production server and served a running API service being called by the European Laboratory for Nuclear Physics (CERN) Single Sign On service and other clients. The API is called by tens of thousand of users daily and thus we can expect patterns in the context of hours, days, rush hours etc.

The data was queried with the use of PromQL, from the Prometheus Monitoring tool as seen below:

Listing 4.1: The PromQL query used for data collection

```
sum(rate(
  process_cpu_seconds_total{
  kubernetes_namespace = ~
      "api-process"}[5m]
  )
)
```

The set interval was set at 5 minutes, yielding a sequence of total length 4435 points over 15 days. The query returned data in a json file, an example is given below:

Listing 4.2: Example of data returned by the query

```
{  "resultType": "matrix" ,
   "result": [
      (    "metric": (),
           "values": [
           [1641773100,"0.08205714228572106"]
           ])]
 }
```

4.1 Outline of the EDA

For the EDA we used the Kaggle notebook service and the languages/libraries Python, Pandas, Numpy that are commonly used for such purposes.

Data Cleaning

The first step in the EDA process is to clean the data, pinpoint inconsistencies, missing data and in general technical difficulties that could make our computation faulty.

We load the data using the $prom - parser.py$ code provided in [17] and saving the sequence in a Pandas DataFrame. We study the data using built-in Pandas function and original scripts to find that there is no missing data, and a general description of the data is shown in **??** **??**.

The data is collected with 5 minute intervals which equals to 12 measurements each hour, or 287 measurements per day.

We proceed to better analyse the data.

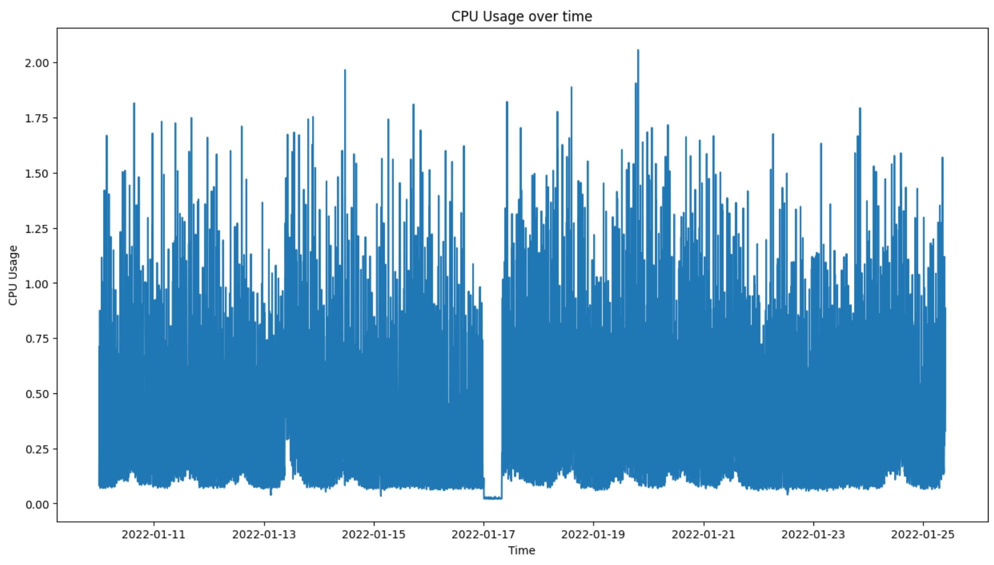
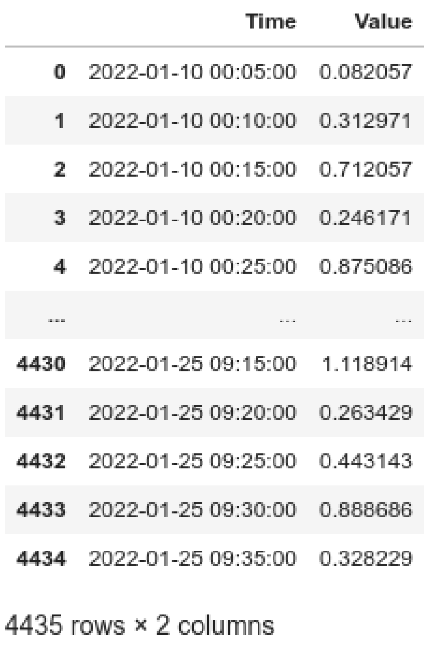


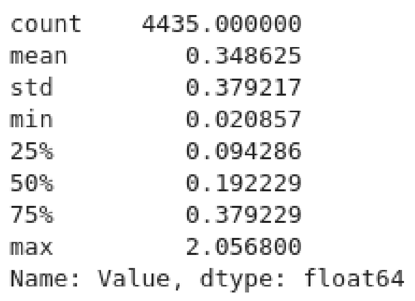
Figure 4.1. *CPU Usage over time*



	Time	Value
0	2022-01-10 00:05:00	0.082057
1	2022-01-10 00:10:00	0.312971
2	2022-01-10 00:15:00	0.712057
3	2022-01-10 00:20:00	0.246171
4	2022-01-10 00:25:00	0.875086
...	...	...
4430	2022-01-25 09:15:00	1.118914
4431	2022-01-25 09:20:00	0.263429
4432	2022-01-25 09:25:00	0.443143
4433	2022-01-25 09:30:00	0.888686
4434	2022-01-25 09:35:00	0.328229

4435 rows × 2 columns

Figure 4.2. *Pandas DataFrame*



```
count    4435.000000
mean        0.348625
std         0.379217
min         0.020857
25%         0.094286
50%         0.192229
75%         0.379229
max         2.056800
Name: Value, dtype: float64
```

Figure 4.3. *CPU Usage over time - Overall Statistics*

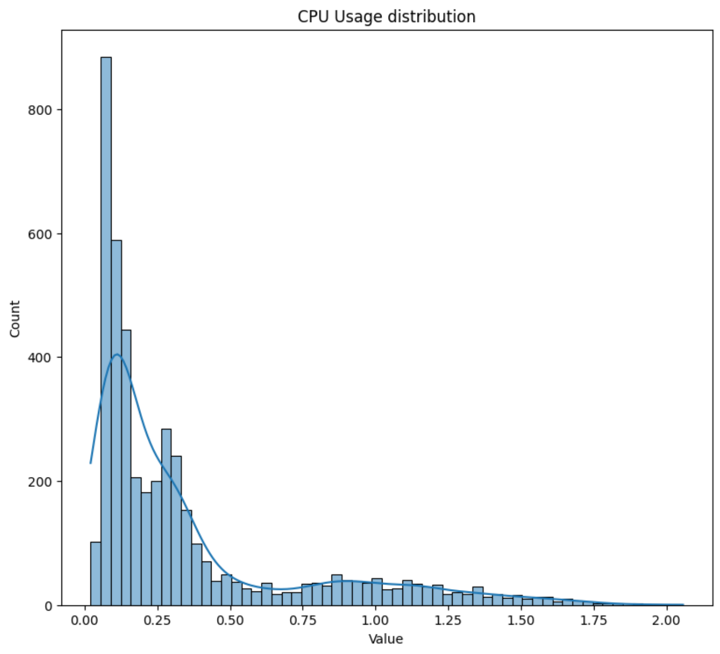


Figure 4.4. *CPU Usage over time - Distribution*

Visualisations

By visualising the data we can see the drop in usage in 4.6 and 4.7 over the weekends (18th-19th of January, 22nd-23rd of January). We also notice the power outage on the 17th of January that leads to idleness.

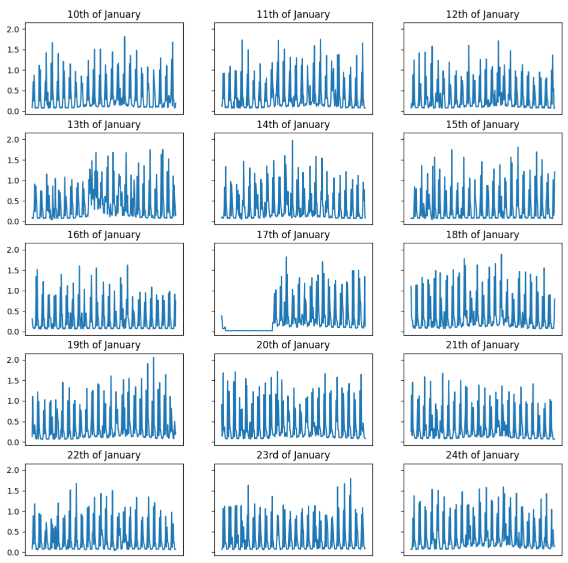


Figure 4.5. *CPU Usage for each day*

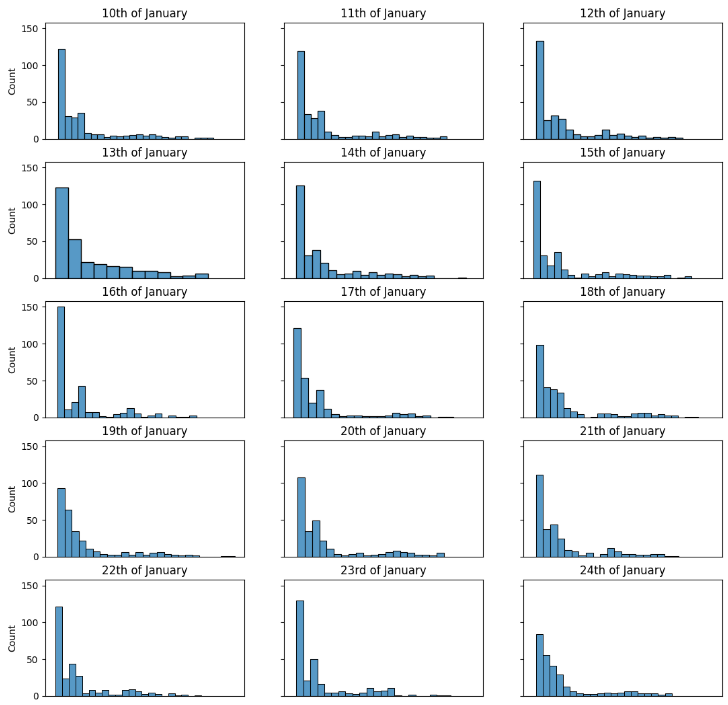


Figure 4.6. *Histogram of daily CPU Usage*

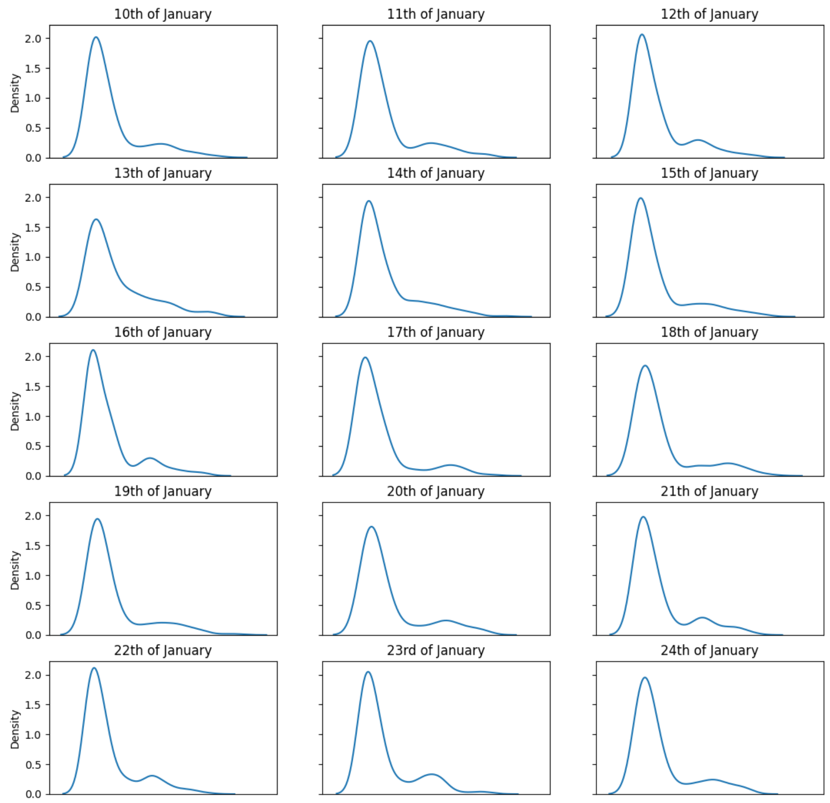


Figure 4.7. *Histogram Curve of daily CPU Usage*

Trends, Seasonality, Residuals

Using the Statsmodels Python library we can see more in depth the daily patterns. We randomly select a weekday and a weekend day and decompose the data into Trend, Seasonality and Residuals.

- Trend shows whether the data are increasing or decreasing in the long term
- Seasonality shows if certain periodic changes occur in the data and can be spotted by removing the other pieces of data
- Residuals show the noise in the data which can be represented in this case by ad-hoc users of the API which cause some load on the system or other anomalies

Comparing the results from Figure 4.8 and Figure 4.9 we can extract many observations. In terms of Trend, we can see an evident higher Trend on the weekdays between 9:00 and 17:00 and an obvious drop after 17:00 which leads only to automated processes. In terms of seasonality, we can observe 2 spikes every hour in a very repetitive and identical manner. These spikes are caused by automated processes that periodically process data e.g. synchronising data. For example, one account at CERN may belong to plenty authorisation groups and thus there must be consistency.

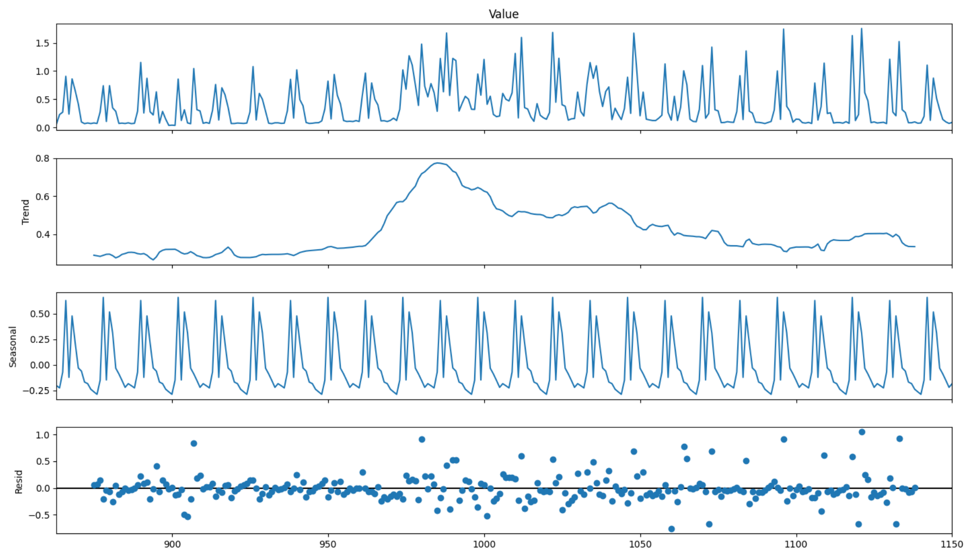


Figure 4.8. *Seasonality, Trend and Residuals for a weekday*

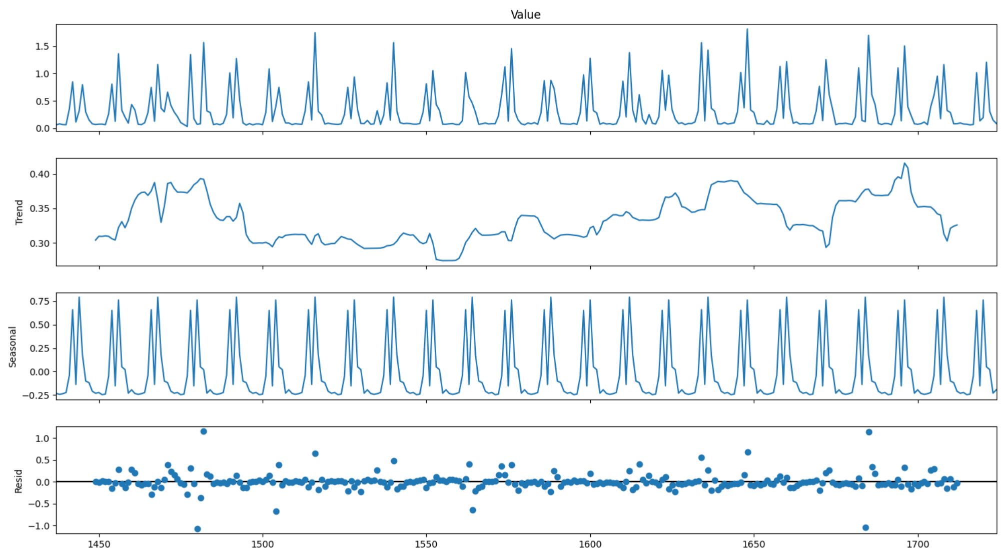
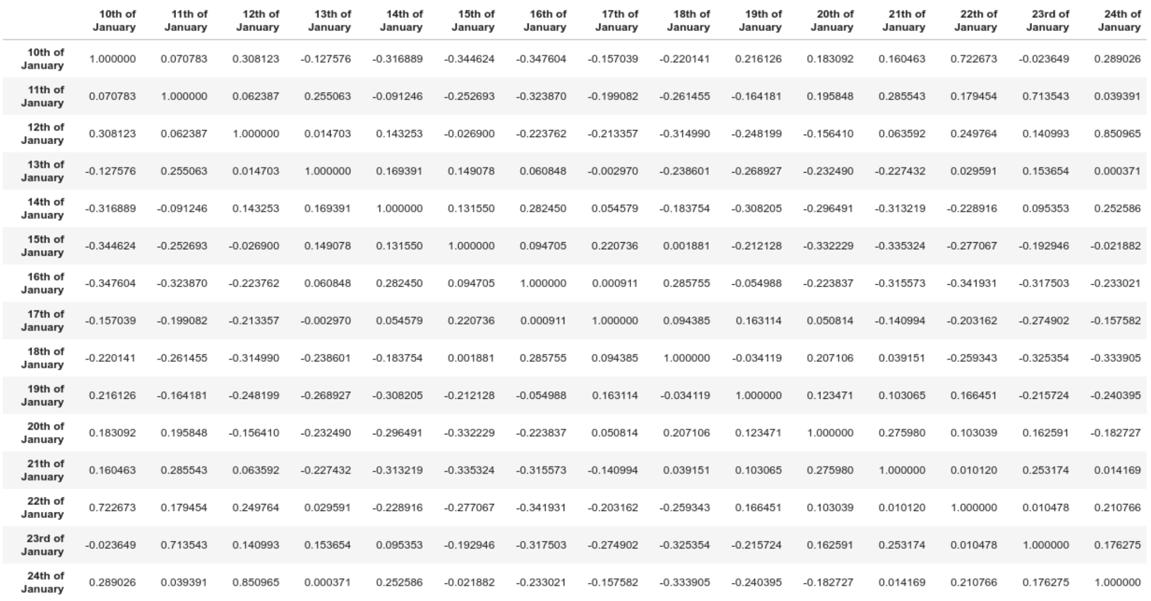


Figure 4.9. *Seasonality, Trend and Residuals for a weekend*

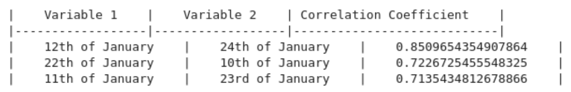
Correlation and Other Statistics

We study the correlation between the different days. The correlation table can be seen below in Figures 4.10 4.11as well as the highest correlated days with a correlation coefficient over 0.4.



	10th of January	11th of January	12th of January	13th of January	14th of January	15th of January	16th of January	17th of January	18th of January	19th of January	20th of January	21th of January	22th of January	23rd of January	24th of January
10th of January	1.000000	0.070783	0.308123	-0.127576	-0.316889	-0.344624	-0.347604	-0.157039	-0.220141	0.216126	0.183092	0.160463	0.722673	-0.023649	0.289026
11th of January	0.070783	1.000000	0.062387	0.255063	-0.091246	-0.252693	-0.323870	-0.199082	-0.261455	-0.164181	0.195848	0.285543	0.179454	0.713543	0.039391
12th of January	0.308123	0.062387	1.000000	0.014703	0.143253	-0.026900	-0.223762	-0.213357	-0.314990	-0.248199	-0.156410	0.063592	0.249764	0.140993	0.850965
13th of January	-0.127576	0.255063	0.014703	1.000000	0.169391	0.149078	0.060848	-0.002970	-0.238601	-0.268927	-0.232490	-0.227432	0.029591	0.153654	0.000371
14th of January	-0.316889	-0.091246	0.143253	0.169391	1.000000	0.131550	0.282450	0.054579	-0.183754	-0.308205	-0.296491	-0.313219	-0.228916	0.095353	0.252586
15th of January	-0.344624	-0.252693	-0.026900	0.149078	0.131550	1.000000	0.094705	0.220736	0.001881	-0.212128	-0.332229	-0.335324	-0.277067	-0.192946	-0.021882
16th of January	-0.347604	-0.323870	-0.223762	0.060848	0.282450	0.094705	1.000000	0.000911	0.285755	-0.054988	-0.223837	-0.315573	-0.341931	-0.317503	-0.233021
17th of January	-0.157039	-0.199082	-0.213357	-0.002970	0.054579	0.220736	0.000911	1.000000	0.094385	0.163114	0.050814	-0.140994	-0.203162	-0.274902	-0.157582
18th of January	-0.220141	-0.261455	-0.314990	-0.238601	-0.183754	0.001881	0.285755	0.094385	1.000000	-0.034119	0.207106	0.039151	-0.259343	-0.325354	-0.333905
19th of January	0.216126	-0.164181	-0.248199	-0.268927	-0.308205	-0.212128	-0.054988	0.163114	-0.034119	1.000000	0.123471	0.103065	0.166451	-0.215724	-0.240395
20th of January	0.183092	0.195848	-0.156410	-0.232490	-0.296491	-0.332229	-0.223837	0.050814	0.207106	0.123471	1.000000	0.275980	0.103039	0.162591	-0.182727
21th of January	0.160463	0.285543	0.063592	-0.227432	-0.313219	-0.335324	-0.315573	-0.140994	0.039151	0.103065	0.275980	1.000000	0.010120	0.253174	0.014169
22th of January	0.722673	0.179454	0.249764	0.029591	-0.228916	-0.277067	-0.341931	-0.203162	-0.259343	0.166451	0.103039	0.010120	1.000000	0.010478	0.210766
23rd of January	-0.023649	0.713543	0.140993	0.153654	0.095353	-0.192946	-0.317503	-0.274902	-0.325354	-0.215724	0.162591	0.253174	0.010478	1.000000	0.176275
24th of January	0.289026	0.039391	0.850965	0.000371	0.252586	-0.021882	-0.233021	-0.157582	-0.333905	-0.240395	-0.182727	0.014169	0.210766	0.176275	1.000000

Figure 4.10. *Correlations over days*



```
|    Variable 1    |    Variable 2    | Correlation Coefficient    |
|-----------------|-----------------|---------------------------|
|    12th of January    |    24th of January    |    0.8509654354907864    |
|    22th of January    |    10th of January    |    0.7226725455548325    |
|    11th of January    |    23rd of January    |    0.7135434812678866    |
```

Figure 4.11. *Top correlations*

4.2 Observations

We can notice plenty of patterns from the EDA. Overall,

- The CPU Usage is spiking in many cases, calling for need of proper scaling
- On the 17th of January half the day is idle due to a power outage
- Hourly seasonality shows spikes that happen in similar hours each day
- Weekday traffic is significantly higher that weekends
- After 5 o' clock - the standard shift hours - the usage drops
- Every hour 2 identical spikes happen

Chapter 5

Model Development

5.1 LSTM for Time series prediction

Data preparation

In order to prepare the data series we introduce the functions "data_generator" and "get_-window" that produce a label y for a data sequence x_i with length $i = seq\_length$ as follows:

Listing 5.1: Data generator fucntions

```
def get_windows(data, seq_length, pred_length=2):
    x = []
    y = []

    for i in range(len(data)-seq_length-pred_length+1):
        _x = data[i:(i+seq_length)]
        _y = np.stack(data[i+seq_length:i+seq_length+pred_length], axis=1)[0]
        x.append(_x)
        y.append(_y)

    return np.array(x),np.array(y)

def data_generator(raw_values, seq_length=4, pred_seq_len=1, normalize=True):

    if (normalize==True):
        sc = MinMaxScaler()
        training_data = sc.fit_transform(raw_values)
    else:
        sc = None
        training_data = raw_values

    x, y = get_windows(training_data, seq_length, pred_seq_len)

    return x, y , sc
```

Using the above functions, we can experiment with "different" dataset modalities and explore which sequence length is better for predicting the next value.

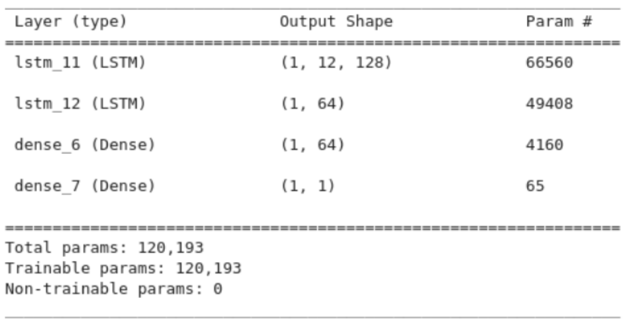
Training and Validation

For the LSTM model we used the language Python alongside the TensorFlow [18] library and Keras back-end, in a Kaggle notebook.

We experimented with various parameters, namely:

```
LSTM units = [16, 32, 64, 128]
LSTM Layers = [1, 2]
sequence_length = [4, 6, 12, 144, 287]

epochs: [20,40,60,75,100]
learning rate: [0.00001, 0.0001 , 0.001]
loss= ['mean_squared_error', 'huber_loss_function']
optimizer='adam'
```

Our best model's summary is printed in Figure 5.1

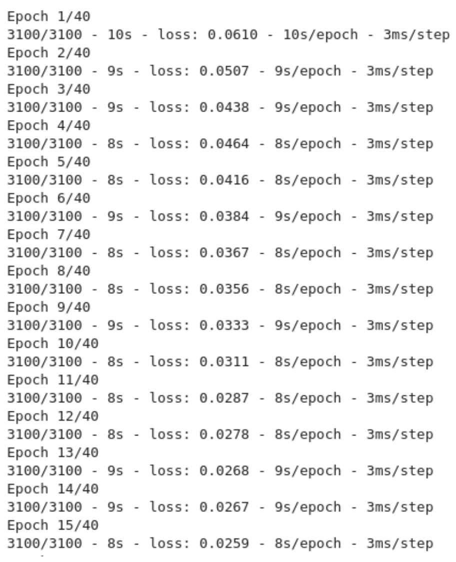


```
Layer (type)                 Output Shape              Param #
=================================================================
lstm_11 (LSTM)               (1, 12, 128)              66560

lstm_12 (LSTM)               (1, 64)                   49408

dense_6 (Dense)              (1, 64)                   4160

dense_7 (Dense)              (1, 1)                    65

=================================================================
Total params: 120,193
Trainable params: 120,193
Non-trainable params: 0
_________________________________________________________________
```

Figure 5.1. *Our LSTM Architecture for CPU forecasting*

We trained the model over 40 epochs in order to avoid overfitting. The phenomenon of overfitting refers to the model learning the data by heart and not memorizing the patterns behind it. Underfitting on the other hand is the phenomenon where the model is not trained enough to learn any patterns.

Results



```
Epoch 1/40
3100/3100 - 10s - loss: 0.0610 - 10s/epoch - 3ms/step
Epoch 2/40
3100/3100 - 9s - loss: 0.0507 - 9s/epoch - 3ms/step
Epoch 3/40
3100/3100 - 9s - loss: 0.0438 - 9s/epoch - 3ms/step
Epoch 4/40
3100/3100 - 8s - loss: 0.0464 - 8s/epoch - 3ms/step
Epoch 5/40
3100/3100 - 8s - loss: 0.0416 - 8s/epoch - 3ms/step
Epoch 6/40
3100/3100 - 9s - loss: 0.0384 - 9s/epoch - 3ms/step
Epoch 7/40
3100/3100 - 8s - loss: 0.0367 - 8s/epoch - 3ms/step
Epoch 8/40
3100/3100 - 8s - loss: 0.0356 - 8s/epoch - 3ms/step
Epoch 9/40
3100/3100 - 9s - loss: 0.0333 - 9s/epoch - 3ms/step
Epoch 10/40
3100/3100 - 8s - loss: 0.0311 - 8s/epoch - 3ms/step
Epoch 11/40
3100/3100 - 8s - loss: 0.0287 - 8s/epoch - 3ms/step
Epoch 12/40
3100/3100 - 8s - loss: 0.0278 - 8s/epoch - 3ms/step
Epoch 13/40
3100/3100 - 9s - loss: 0.0268 - 9s/epoch - 3ms/step
Epoch 14/40
3100/3100 - 9s - loss: 0.0267 - 9s/epoch - 3ms/step
Epoch 15/40
3100/3100 - 8s - loss: 0.0259 - 8s/epoch - 3ms/step
```

Figure 5.2. *Train Logs from the LSTM Model training*

The results of our model can be seen below in Figure 5.3 and Figure 5.4

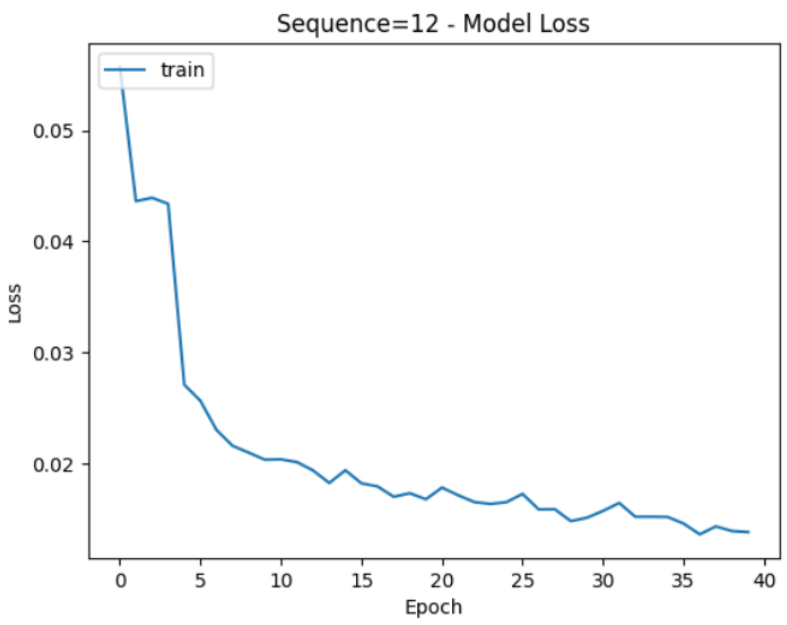
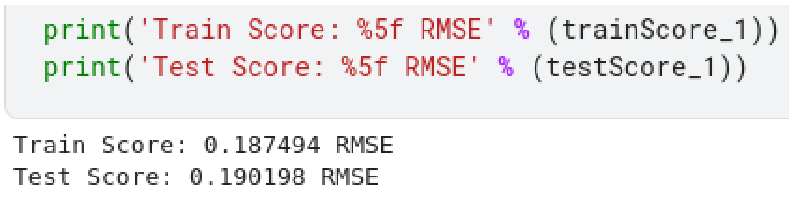


Figure 5.3. *Loss function during training*



```
print('Train Score: %5f RMSE' % (trainScore_1))
print('Test Score: %5f RMSE' % (testScore_1))
```

```
Train Score: 0.187494 RMSE
Test Score: 0.190198 RMSE
```

Figure 5.4. *Root Mean Square Error on the train and test set*

As expected, our model had better performance on the train set and slightly worse on the test set. When we visualise the predictions we can see a good pattern forecasting.

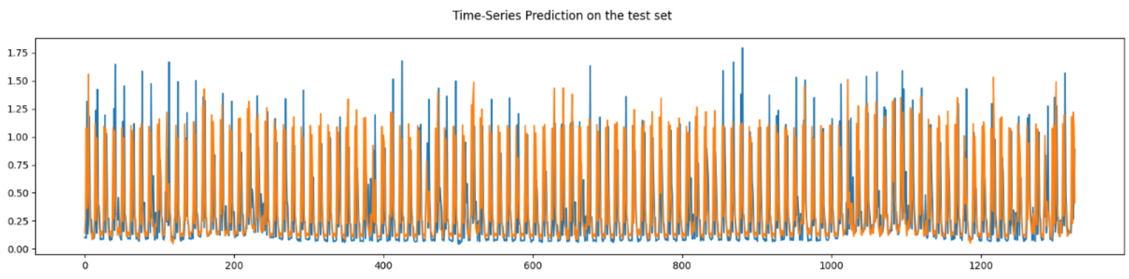


Figure 5.5. *Prediction Sequence on the test set*

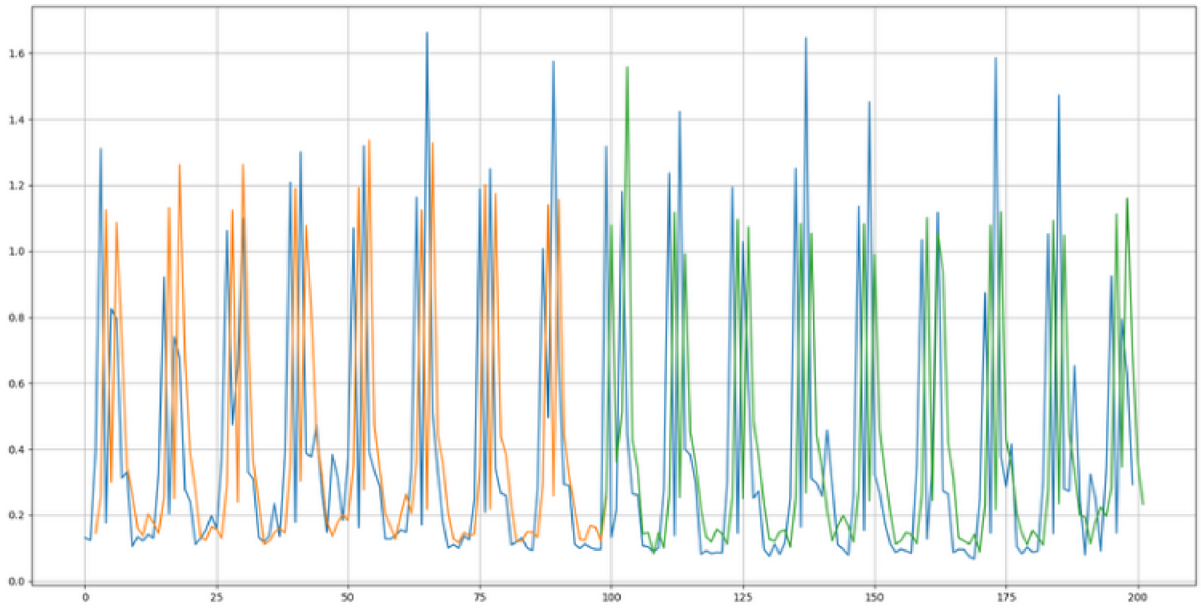


Figure 5.6. *Orange: Prediction on the train set. Green: Predictions on the test set*

5.2 Reinforcement Learning Agent

We can now proceed to introducing a Reinforcement Learning agent that takes into consideration the above forecasting model predictions in the Q-Network.

5.2.1 Environment and Q Network

We define the agent's environment called the *KubernetesEnv* as follows. The environment contains information about the cluster state, computes the reward and returns the observations of the agent. The Q Network follows the architecture explained in Section 5.1.

```
class KubernetesEnv:
    def __init__(self,data,num_actions,history_window):
        super(ClusterEnvironment, self).__init__()

        ###STATE
        self.cluster_state #initial Cpu usage - State representation
        self.max_cluster_state # maximum cpu usage we want to reach
        self.min_cluster_state # idle state

        ###ACTIONS
        self.nodes #current number of nodes in the cluster
        self.max_nodes #maximum nodes we can have
        self.action_set #scaling actions, e.g. ["None","Up","Down"]
        self.action_space = np.arange(num_actions)


        self.observation_space = []
        self.history_window=history_window
        self.history = [0 for i in range(self.history_window)]
        self.data = data
        self.current_t = 0
        self.done=False
        self.profits=0
        self.reset()


    def reset(self):
        #resets the environment to it's original values

    def step(self, action):

        reward = 0

        if action == 0:
            #NO action

        elif action == 1:
            #Scale Up
            self.nodes = min(self.nodes+1,self.max_nodes)
            reward += int(self.max_cluster_state-
            (self.data.iloc[self.current_t-1]["Value"]
            -self.data.iloc[self.current_t]["Value"])*100)
            if(self.current_t==0):
                reward += int(self.max_cluster_state
                -( self.data.iloc[self.current_t]["Value"])*100)
            self.profits += reward

        elif action ==2:
            #Scale Down
            self.nodes = max(self.nodes-1,1)
            reward += int(self.max_cluster_state
            - (self.data.iloc[self.current_t-1]["Value"]
            - self.data.iloc[self.current_t]["Value"])*100)
            if(self.current_t==0):
                reward += int(self.max_cluster_state
                -( self.data.iloc[self.current_t]["Value"])*100)
            self.profits += reward
        self.current_t += 1
        self.position_value = self.data.iloc[self.current_t]["Value"]
        self.history.pop(0)
        self.history.append(self.data.iloc[self.current_t - 1]["Value"])
        if self.current_t == len(self.data) - 1:
            done = True
        return self.history, reward, self.done , self.nodes

class Q_Network(nn.Module):
    def __init__(self,input_size=6 ,hidden_size=32, actions=3):
        super(Q_Network, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size=input_size,
                            hidden_size=hidden_size,
                            num_layers=2,
                            batch_first=True)
        self.fc = nn.Linear(128, 64)
        self.fc2 = nn.Linear(64, actions)

    def forward(self, x):
        #Pass through the model
        o, _ = self.lstm(a)
        f = self.fc(o)
        out = self.fc2(f)
        return out
```

The Target Network is created by simply copying the Q Network with the fuction *deepcopy* to avoid any correlations of the variables.

```
Q_b = copy.deepcopy(Q)
```

5.2.2 Training Parameters

We define the criterion for the loss function of our Q-Network as well as the optimizer. For the loss function we experimented both with the *nn.MSE*() function that represents the Mean Squared Error. However, the MSE function did not lead to proper convergence and the loss was not sufficiently minimized. That phenomenon is caused due to the fact that the computation of the MSE accumulates big values caused by the outliers, which we have plenty of. A better approach to the loss minimization is the Huber Loss function[47]. Concerning the optimizer, we used the *Adam*() optimizer [48] which has proven suitable for most optimization in Machine Learning models.

```
criterion = nn.HuberLoss()
optimizer = optim.Adam(list(Q.parameters()),learning_rate=0.00001)
```

When selecting the specific parameters we defined the ones stated below, experimenting mostly with the first set. The number of epochs greatly impacts the total training time, but it is not always beneficial to train for a long time. We experimented with a set of 20,50,100 epochs, ultimately settling for the 20. Another critical set of parameters was the size of the Memory Replay Buffer, defined by *mem_size* as well as the *batch_size* used for sampling and training.

```
epochs = ...
step_max = len(env.data) - 1
mem_size = ...
batch_size = ...
gamma = ...

memory = []
total_step = 0
total_rewards = []
total_losses = []
```

```
#exploration
epsilon = 1.0
epsilon_min = 0.1
epsilon_decrease = 1e-3
start_reduce_epsilon = 200

#network updates
update_q_freq = ...
```

Learning Rate

The learning rate defines in which degree the acquired knowledge for the $Q(s, a)$ will overpower older information. A value of zero will lead the Agent to discarding the new knowledge and only using the old one, while on the other hand a value of one will not take into consideration the old information at all. In a deterministic environment the learning rate would always be one. However in a stochastic problem, convergence requires prior information otherwise it is not possible. For that reason, the learning rate is set close to zero. We experimented with values 0.001, 0.0001 and 0.00001, ultimately deciding that a slow learning rate, 0.00001, is more beneficial for the training of our model.

The γ factor

The γ factor defines the value of the future rewards. If we select $\gamma = 0$, then the Agent is not taking into consideration any value of future rewards. In contrast, a factor of $\gamma = 1$ will lead the Agent to look forward to bigger future rewards continuously which will make convergance impossible. In practice, the values used are close to zero but not equal to, such that the algorithm converges [49]. In our case we selected $\gamma = 0.97$

Experience Replay Buffer and Random Sampling

As explained in Section 3.4.2 the Replay Buffer stores the experiences of the agent in a tuple of $< s, a, r, s' >$, with the purpose of random sampling and training. In our code, two more variables are stored in each tuple, the *done* parameter that signals the end of the episode and the n parameter that stores the number of nodes in the state. By extending the memory we can store more experiences and sample a more statistically random sample from the buffer. We experimented with sizes ranging from 10 to 1000 for the memory and 2 to 50 for the size of the random batch. Before selecting the random batch, the memory is shuffled.

```
memory.append([pobs, pact, reward, obs, done, n])
    if len(memory) > mem_size:
        memory.pop(0)
[...]

if len(memory) == mem_size:
    if (total_step+1) % train_freq == 0:
        shuffled_memory = np.random.permutation(memory)
    memory_idx = range(len(shuffled_memory))
```

Computation of Q-Values

When iterating the batch, we extract the tuple elements $< s, a, r, s', done, n >$ in different variables and compute the Q-Values for each state s and the target values for each s'

```
for i in memory_idx[::batch_size]:
    batch = np.array(shuffled_memory[i:i + batch_size])
    [...]
q = Q(...)
q_ = Q_b(...)
maxq = np.max(q_.data.numpy(),axis=1) #Get the max value from the Q table
target = copy.deepcopy(q.data)
```

Training

Following the equation explained in Section 3.4.2 we compute the new cumulative expected reward for each state. Backpropagation computes the weights for each value and adds to the total loss. Afterwards we check the frequency and when reached, we update the Target Network as well.

```
for j in range(batch_size):
    target[j] = b_reward[j] + gamma * maxq[j] * (not b_done[j])
    Q.zero_grad()
    optimizer.zero_grad()
    loss = criterion(q,target)
    #Backpropagation
    loss.backward(retain_graph=True)
    total_loss += loss.item()
    optimizer.step()

[...]

if total_step % update_q_freq == 0:
        Q_b = copy.deepcopy(Q)
```

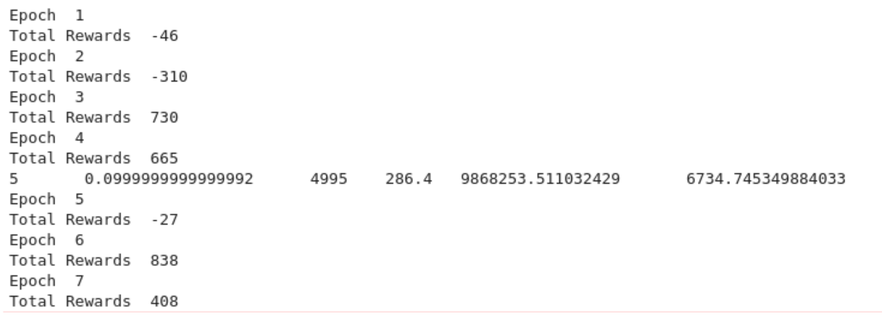
Exploration vs Exploitation

We also utilise the feature of Exploration by selecting a random action more frequently for the smaller ϵ values. At the end of each epoch we update to a higher ϵ value.

```
pact = np.random.randint(3) #random action

if np.random.rand() > epsilon:
    #Select random action
```

5.3 Results

Concluding, we evaluated our Agents performance, keeping the best ones.



```
Epoch  1
Total Rewards  -46
Epoch  2
Total Rewards  -310
Epoch  3
Total Rewards  730
Epoch  4
Total Rewards  665
5      0.0999999999999992      4995    286.4   9868253.511032429       6734.745349884033
Epoch  5
Total Rewards  -27
Epoch  6
Total Rewards  838
Epoch  7
Total Rewards  408
```

Figure 5.7. *Train Logs from the Agent's training*

We can observe the loss function declining as the Agent learns the patterns for scaling as well as the reward function having a rising curve as the Agent maximizes the rewards.

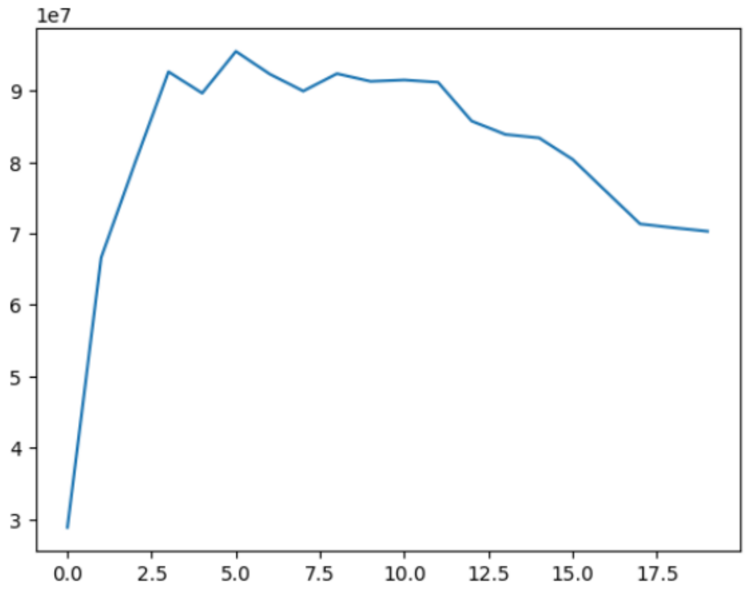


Figure 5.8. *Loss Function of the Agent*

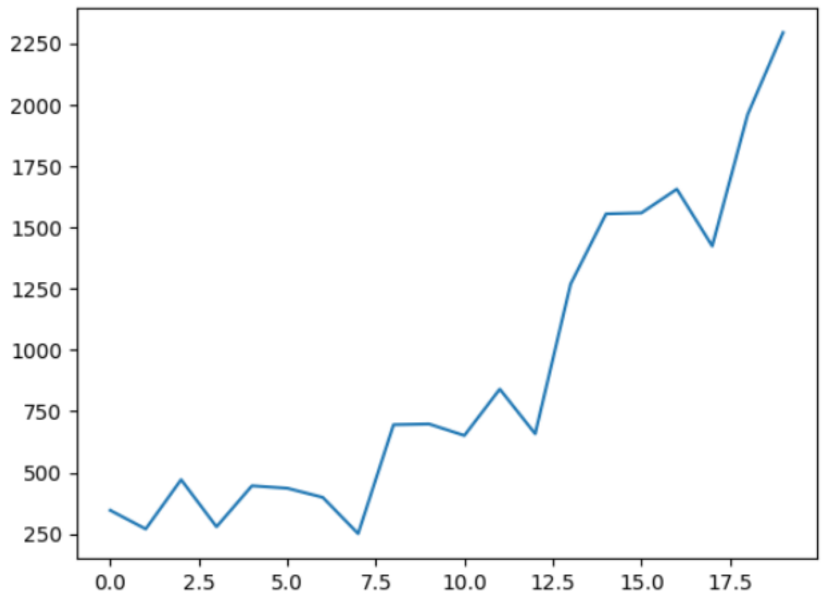


Figure 5.9. *Reward Function of the Agent*

When we study the Agents performance we can see and compare between two different cases. In Figure 5.10, we defined *history_window* = 60 and in Figure 5.11 we defined $historywindow = 12$. We can notice that on the first case the Agent makes more conservative scaling decisions, by not decreasing the node number significantly as the experience calls for a recurring requirement around 9 nodes. On the second case, the Agent takes into account a shorter sequence in the experiences, thus makes more reactive scaling decisions that may even be non practical in the real scheduling world.

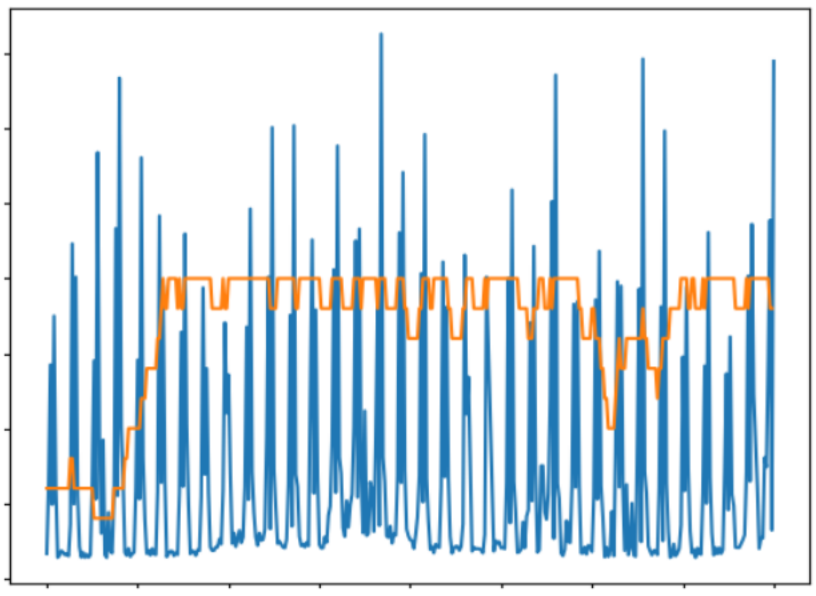


Figure 5.10. *Predictions of the Agent, History Window = 60*

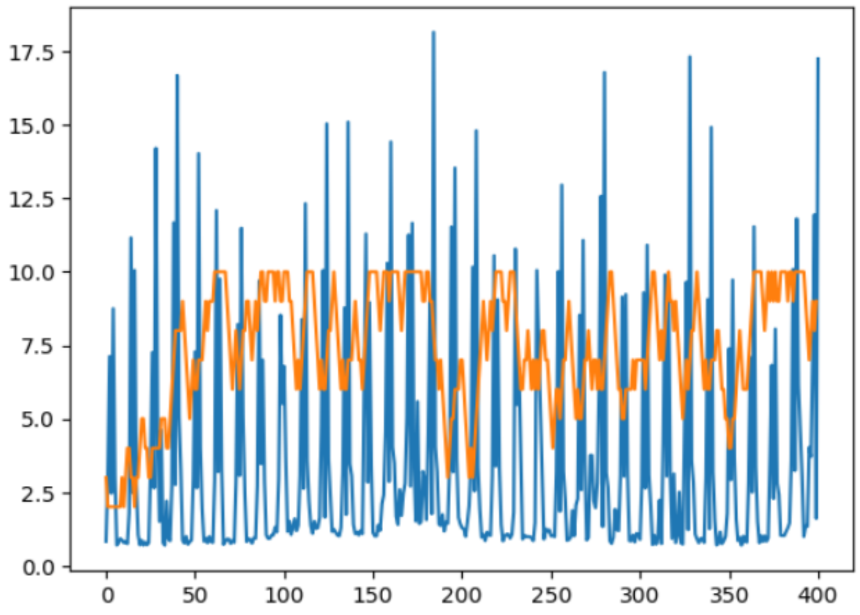


Figure 5.11. *Predictions of the Agent, History Window = 12*

Chapter 6

Future Work and Extensions

6.1 Discussion

Through these experiments we acquired a sense of scaling and how these high level components can be designed. However, there is need for further discussion on our resources, our methodology and, ultimately, our decisions.

Data Acquisition

Concerning our data, there are a lot of improvements to be made. Our data source is very restricted on every aspect as it does not include basic information about the cluster such as the number of nodes in the said cluster. Without knowing the number of nodes, we can only arbitrarily assume a number for the nodes, the minimum and the maximum. Most importantly, however, this gap affects our analysis in the sense that we cannot know the effect that our speculative scaling has on the future CPU usage. This fact alone can defy the pro-activeness of the Agent. If the number of nodes, and its variance, was known alongside the usage we could approximate a ratio of the scaling effect on the cluster.

Moreover, we cannot overlook the fact that the CPU usage is just one metric and many more can be acquired through PromQL queries in order to have the bigger picture of the cluster. Metrics such as the memory usage, the SLA violations and more could be of great usage.

Extending this conversation, there can even be doubt for the usability of the data we proposed data. There have been works that sufficiently predict the scalabitily of a cluster based solely on seasonality and trend.

Methodology

On the discussion about our methodology there are various points to be made. Our original motivation included the creation of our own cluster. The Computer Systems Laboratory (CSLab) generously provided resources on their servers. However, there were plenty of networking problems mainly in the internal Kubernetes network that exposed the endpoints. On our first mitigation we utilized the available resources on the Okeanos Knossos cloud in GRNET's cloud service [50], where the creation of the internal network is more user friendly and fail proof. Unfortunately, in this case the efforts to load-test the cluster in order to obtain the metrics were futile and resulted in empty files or missing values across all tests. Our methodology in this effort included creating various scenarios with the Kube-burner tool [51] which offered little detail in the documentation and few usage examples in local clusters.

Continuing with our plan C, we utilized the only available open source CPU usage dataset online. However, our current methodology still needs to be discussed. Throughout our tests, even though many parameters were tested, the LSTM models and the Agent were prone to overfitting and it is probable that we have reached a local minima and the values are not optimal. As a result, we cannot be sure about the Agent's performance and more parameters need to be tested and aggregated into a complete review.

6.2 Future Work

Extending the above into possibilities for feature work we can suggest the following.

Data Acquisition

Data collection is very important for this task, as our point stands: Every application is different and designing an Agent for an open source dataset with little to none information on the cluster can be vastly different from one that can perform and be tested on our specific needs. Future work needs to be implemented on a local or directly customised cluster that can run load testing scenarios with many parameters and well defined limitations. Moreover, there need to be included many more metrics, and the exploratory data analysis can then study their significance. The PromQL queries can expose a huge number of complicated monitoring metrics.

Methodology

Besides the methodology that needs to be followed in order to acquire a working cluster that can be tested, we need to stress the need for further development of the Python code for both the Agent and the LSTM models. That includes defining a more complicated reward system in the Agent's environment as well as different LSTM models in the Q Network, such as bidirectional LSTMs that can capture the information from both sides of the time sequence.

Last but not least, the Agent needs to be integrated into a custom controller and evaluated in a real usage scenario and be compared to different approaches such as naive scheduling methods based on mean value but also more complicated ones from the relevant literature available.

Bibliography

[1] Andrei Gurtov και Vladimir Mazalov. *Queueing System with On-Demand Number of Servers. Mathematica Applicanda*, 40(2), 2012.

[2] M.A. Kaboudan. *A dynamic-server queuing simulation. Computers &amp Operations Research*, 25(6):431–439, 1998.

[3] Yanhe Jia, Lixin Tang, Zhe George Zhang και Xiaofeng Chen. *MMPP/M/C queue with congestion-based staffing policy and applications in operations of steel industry. Journal of Iron and Steel Research International*, 26(7):659–668, 2018.

[4] Qian Zhu και Gagan Agrawal. *Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments. IEEE Transactions on Services Computing*, 5(4):497–511, 2012.

[5] Muhammad Wajahat, Anshul Gandhi, Alexei Karve και Andrzej Kochut. *Using machine learning for black-box autoscaling. 2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, σελίδες 1–8, 2016.

[6] Hamid Arabnejad, Claus Pahl, Pooyan Jamshidi και Giovani Estrada. *A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling. 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, σελίδες 64–73, 2017.

[7] Shay Horovitz και Yair Arian. *Efficient Cloud Auto-Scaling with SLA Objective Using Q-Learning. 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, σελίδες 85–92, 2018.

[8] Fabiana Rossi, Matteo Nardelli και Valeria Cardellini. *Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, σελίδες 329–338, 2019.

[9] Bryan Lim και Stefan Zohren. *Time-series forecasting with deep learning: a survey. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 379(2194):20200209, 2021.

[10] Sardar M N Islam, Narina Thakur, Kanishka Garg και Akash Gupta. *A Recent Survey on LSTM Techniques for Time-Series Data Forecasting. Applications of Artificial Intelligence, Big Data and Internet of Things in Sustainable Development*, σελίδες 123–132. CRC Press, 2022.

[11] Jian Zheng, Cencen Xu, Ziang Zhang και Xiaohua Li. *Electric load forecasting in smart grids using Long-Short-Term-Memory based Recurrent Neural Network. 2017 51st Annual Conference on Information Sciences and Systems (CISS)*. IEEE, 2017.

[12] Nédra Mellouli, Mahdjouba Akerma, Minh Hoang, Denis Leducq και Anthony Delahaye. *Multivariate Time Series Forecasting with Deep Learning Proceedings in Energy Consumption. Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management.* SCITEPRESS - Science and Technology Publications, 2019.

[13] Saurav Kumar και Dhruba Ningombam. *Short-Term Forecasting of Stock Prices Using Long Short Term Memory. 2018 International Conference on Information Technology (ICIT).* IEEE, 2018.

[14] Shashank Tripathi. *Can stock prices be predicted? Comparative study of LSTM and SVR for financial market forecast.* 2022.

[15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra και Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning*, 2013.

[16] Marius Cioca και Ioan Cristian Schuszter. *A System for Sustainable Usage of Computing Resources Leveraging Deep Learning Predictions. Applied Sciences*, 12(17):8411, 2022.

[17] saibot94. *Anonymized CPU usage dataset - Github [Online]. Available: https://github.com/saibot94/cpu-dataset-prometheus.*

[18] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu και Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2015. Software available from tensorflow.org.

[19] *Prometheus - Monitoring System and Time Series Database [Online]. Available: https://prometheus.io/.*

[20] *Kubernetes Documentation: Containers https://kubernetes.io/docs/concepts/containers/.*

[21] David Bernstein. *Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Computing*, 1(3):81–84, 2014.

[22] *Docker [Online]. Available: https://www.docker.com/.*

[23] The Kubernetes authors. *Kubernetes: Production-Grade Container Orchestration [Online]. Available: http://kubernetes.io/*, 2022.

[24] *cri-o Lightweight Container Runtime for Kubernetes [Online]. Available: https://cri-o.io/.*

[25] *Containerd An industry-standard container runtime with an emphasis on simplicity, robustness and portability [Online]. Available: https://containerd.io/.*

[26] *Cluster Architecture - Kubernetes [Online]. Available:https://kubernetes.io/docs/concepts/architecture/.*

[27] *The Kubernetes API - Kubernetes [Online]. Available: https://kubernetes.io/docs/concepts/overview/kubernetes-api/.*

[28] *Scheduling Framework - Kubernetes [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.*

[29] *Kubernetes Scheduler - Kubernetes [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler.*

[30] *Scheduling Framework - Kubernetes [Online]. Available: https://kubernetes.io/docs/reference/scheduling/config/.*

[31] *Pods - Kubernetes [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/.*

[32] *etcd [Online]. Available: Documentation https://github.com/etcd- io/etcd/blob/master/Documentation/docs.md.*

[33] *What is etcd? - IBM [Online]. Available: https://www.ibm.com/topics/etcd.*

[34] *Cluster Networking - Kubernetes [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/networking/.*

[35] *Horizontal Pod Autoscaling- Kubernetes [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.*

[36] Laszlo Toka, Gergely Dobreff, Balazs Fodor και Balazs Sonkoly. *Machine Learning-Based Scaling Management for Kubernetes Edge Clusters. IEEE Transactions on Network and Service Management*, 18(1):958–972, 2021.

[37] *Vertical Pod Autoscaling, Google Kubernetes Engine [Online]. Available: https://cloud.google.com/ kubernetes-engine/docs/concepts/verticalpodautoscaler*, 2021.

[38] Huawei-Cloudnative/Kubernetes. *Vertical-Scaling. [Online]. Available: https://github.com/huawei-cloudnative/kubernetes/tree/vertical-scaling.*

[39] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu και Devesh Tiwari. *Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019.

[40] Hamzeh Khazaei, Rajsimman Ravichandiran, Byungchul Park, Hadi Bannazadeh, Ali Tizghadam και Alberto Leon-Garcia. *Elascale: Autoscaling and Monitoring as a Service*, 2017.

[41] *Prometheus Query Language - PromQL [Online]. Available: https://prometheus.io/docs/prometheus/latest/querying/basics/.*

[42] Sepp Hochreiter και Jürgen Schmidhuber. *Long Short-Term Memory. Neural Computation*, 9(8):1735–1780, 1997.

[43] Y. Bengio, P. Simard και P. Frasconi. *Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[44] Kyunghyun Cho, Bartvan Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk και Yoshua Bengio. *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014.

[45] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage και Anil Anthony Bharath. *Deep Reinforcement Learning: A Brief Survey. IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[46] David Balla, Csaba Simon και Markosz Maliosz. *Adaptive scaling of Kubernetes pods. NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium.* IEEE, 2020.

[47] Peter J. Huber. *Robust Estimation of a Location Parameter. The Annals of Mathematical Statistics*, 35(1):73–101, 1964.

[48] Diederik P. Kingma και Jimmy Ba. *Adam: A Method for Stochastic Optimization*, 2017.

[49] Vincent Franç ois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare και Joelle Pineau. *An Introduction to Deep Reinforcement Learning. Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

[50] *Home | okeanos-knossos [Online]. Available: https://okeanos-knossos.grnet.gr/home/.*

[51] *Kube-burner Documentation [Online]. Available: https://cloud-bulldozer.github.io/kube-burner/v1.7.10/.*

List of Abbreviations

AI	Artificial Intelligence
ML	Machine Learning
CPU	Central Processing Unit
MLP	Multilayer Perceptron
K8s	Kubernetes
HPA	Horizontal Pod Autoscaler
VPA	Vertical Pod Autoscaler
PromQL	Prometheus Query Language
API	Application Program Interface
EDA	Exploratory Data Analysis
DL	Deep Learning
LSTM	Long-Short Term Memory
MDP	Markov Desicion Process
RL	Reinforcement Learning
DQN	Deep Q Network