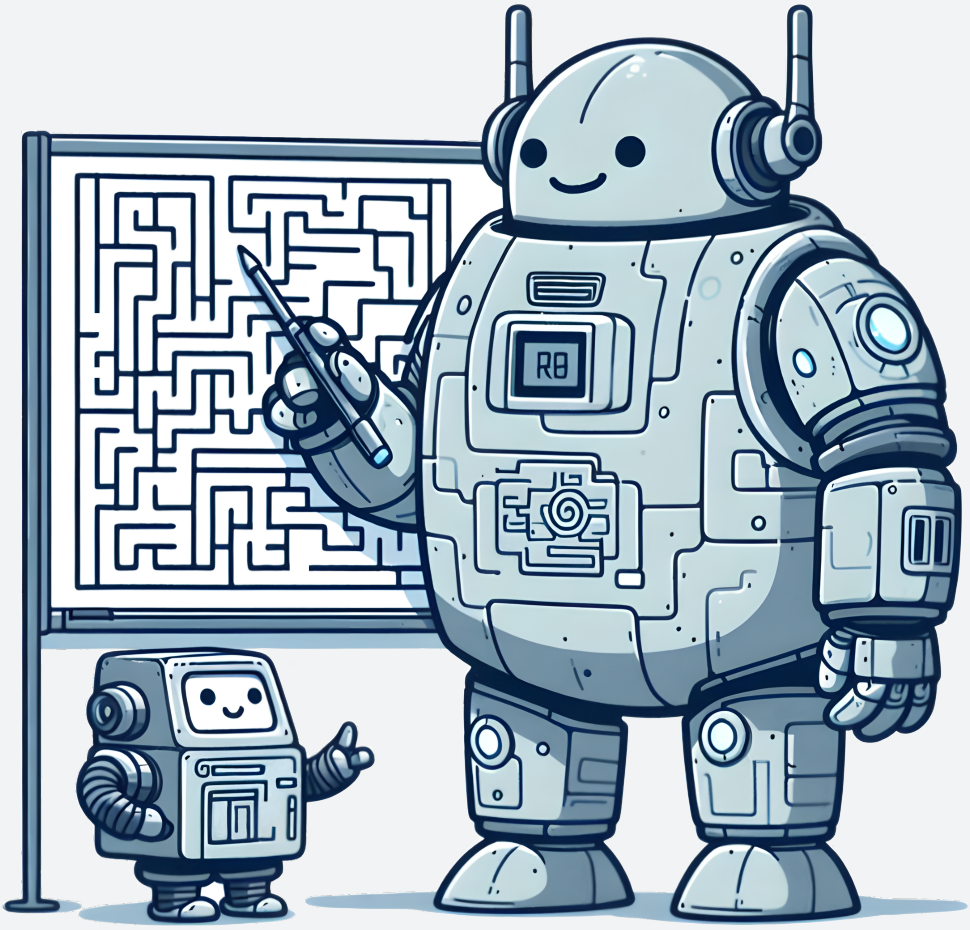
Towards Efficient Decision-Making

Policy Compression for Reinforcement Learning on Resource-Constrained Devices

Thomas Avé



Supervisors **prof. dr. ir. Kevin Mets** | **prof. dr. ir. Steven Latré**

Thesis submitted in fulfilment of the requirements for the degree of Doctor of Science: Computer Science
Faculty of Science | **Antwerpen, 2025**

University of Antwerp

Faculty of Science

Towards Efficient Decision-Making

Policy Compression for Reinforcement Learning on Resource-Constrained Devices

Thesis submitted in fulfilment of the requirements for the degree of
Doctor of Science: Computer Science
at the University of Antwerp

Thomas Avé

Antwerpen, 2025

Supervisors
prof. dr. ir. Kevin Mets
prof. dr. ir. Steven Latré

Jury

Chair
prof. dr. ir. José Antonio Oramas Mogrovejo, University of Antwerp, Belgium

Supervisors
prof. dr. ir. Kevin Mets, University of Antwerp, Belgium
prof. dr. ir. Steven Latré, University of Antwerp, Belgium

Members
prof. dr. Tim Verdonck, University of Antwerp, Belgium
prof. dr. Guillermo Alberto Pérez, University of Antwerp, Belgium
prof. dr. Pieter Libin, Vrije Universiteit Brussel, Belgium
prof. dr. Paul Harvey, University of Glasgow, United Kingdom

Funding
This research was funded by Research Foundation Flanders (FWO) [No. 1SD9523N].

Contact
Thomas Avé
University of Antwerp
Faculty of Science
Department of Computer Science
IDLab Research Group

`www.thomasave.be`

Acknowledgements

Most kids aspire to become astronauts, firefighters, or teachers when they grow up. I have always dreamed of becoming a scientist. I wanted to discover new things through experimentation and research and share that knowledge with the world. It is truly fulfilling to have realised that dream, and I have many people to thank for helping me along the way.

In secondary school, this narrowed down to computer science. Not through any classes I took, but rather when I accidentally broke into the school's administration system because they used the same login system as the teacher-only Wi-Fi, which I also wanted access to. Or by subsequently taking down that very Wi-Fi on purpose when I hadn't practised enough for a presentation. My first thanks therefore go out to my informatics teacher Jo Cleykens. Again, not for his actual teaching, but for putting up with my shenanigans and instead of discouraging me, guiding me through the fundamental basics of research ethics and responsible disclosure. I may have lost access to the Wi-Fi, but I'm proud to have helped fix this issue not just for my school, but for all schools in Flanders, where other students could have more nefarious intentions, which is substantially more valuable.

This naturally led me to study computer science at the University of Antwerp, where I met my good friends Tobia, Ewout, and Robin, who have later been invaluable in their support, encouragement and occasional welcome distractions. In particular, I want to thank Robin for teaching me by example to always strive for excellence, and for taking me along the ride with him. Whether it was simply the group projects we did together or the many national and international algorithmic programming competitions we participated in. I honestly don't think I would have even entertained the idea of doing a Ph.D. if it weren't for him.

We also joined the Honours Programme together, where I had my first real introduction to the IDLab research group and some people who would later become my colleagues. Among them was (now Prof. Dr.) Tom De Schepper, who was pursuing his own Ph.D. at the time, and warned me that doing so is a marathon, not a sprint; that it is as much about perseverance and endurance to work on the same topic for an extended period as it is about intelligence and competence. I feel that this discussion has helped me better prepare for the long journey, and got me through to its completion. So, I would like to thank him for his advice, and for later guiding me as a supervisor in the first steps of my Ph.D.

During my master's, I became intrigued by the idea of machine learning, and in particular, reinforcement learning. The concept of an AI agent learning how to perform a complex task entirely by itself, simply by telling it whether it was doing well or not, was wildly fascinating. The irony that I later spent most of my research working with distillation, which does work based on a stronger supervision signal, is not lost on me. Although we

didn't have any courses on the topic at the time, I was lucky to have Prof. Dr. Ir. Steven Latré, who encouraged me to pursue this interest in my master thesis as a supervisor, and later promotor of my Ph.D. He also gave me the opportunity to already work in a research environment at our group during my master, and begin preparing an FWO application for a Ph.D. position, which would prove to be successful. I am grateful for his faith in me from this early stage, the opportunities he helped enable, and the guidance he provided for shaping my Ph.D. topic.

Although he left the group shortly after to pursue his opportunities within imec, he passed the torch of my supervision to the capable hands of Prof. Dr. Ir. Kevin Mets as co-promotor. Kevin has been an invaluable source of high-quality feedback and guidance throughout my Ph.D., and I am grateful for his critical eye and the discussions we had. We may not have always immediately agreed on everything, but I feel that this has been instrumental in improving the quality of my work and of me as a researcher, and I am proud of the results we have achieved together. I am also thankful for giving me the occasional push when I needed to actually get things done, and for the trust he placed in me during the final stages to work independently and explore my own ideas.

There are many more people at IDLab that I would like to thank, including Louis, Fabian, Matthias, Mattias, Wei, Samer, and many others. I appreciate all the interesting discussions, collaborations, and simply making it an enjoyable place to work.

Then there are my friends Dries, Demi, Emily, Bern, Leen, Nick, and Valerii. Thank you for your patience when I was distracted during a D&D session because one of my experiments had just finished, and for still inviting me back the next time. The frequent role-playing with countless hours of fun together was essential for maintaining my mental well-being throughout these four years, and I am truly grateful for every second of it.

Finally, there's only the reader left to thank, including the Ph.D. jury. I hope you find this dissertation as interesting to read as it was to write, and that you might even be inspired by the future work and continue where I left off.

Antwerp, 2024
Thomas Avé

Summary

The widespread adoption of Deep Reinforcement Learning (DRL) has led to remarkable achievements in various domains, from mastering complex games to controlling robotic systems. However, deploying these powerful models on resource-constrained edge devices remains challenging due to their substantial computational requirements, memory footprint, and energy consumption. Traditional approaches to this problem, such as cloud offloading or relying on hardware improvements, have significant limitations. Cloud offloading introduces unacceptable latency for real-time decision-making tasks and requires constant network connectivity. Hardware improvements alone cannot keep pace with the growing complexity of DRL models. Additionally, directly training small models often fails to achieve the desired level of task performance, due to the inherent discrepancy between the capacity required for training and deployment.

To address these challenges, model compression techniques have been developed to reduce the size and computational cost of DRL models. Model compression typically first involves training large models that can learn complex tasks most effectively. Subsequently, these models undergo a compression process to reduce their size and computational complexity, thereby bypassing the difficulties of training smaller models directly, and making them suitable for deployment on edge devices. In policy distillation, a small student model is trained to mimic the behaviour of a larger teacher model. Pruning, on the other hand, involves removing redundant connections or neurons from a fully trained model, reducing the number of parameters. Finally, quantization reduces the numerical precision of the model's weights and activations, enabling the use of low-precision arithmetic during inference.

Model compression for DRL policy networks is still relatively understudied, however, with most research adapting supervised learning methods without fully leveraging DRL's unique characteristics. This gap limits the potential for smaller, more efficient DRL agents to be deployed at the edge. Moreover, not all DRL policies are compatible with existing compression methods, necessitating the development of new techniques tailored to their specific requirements.

This thesis hypothesises that by understanding and exploiting the unique properties of DRL models, novel compression methods can be developed that achieve higher compression rates and enable the compression of a wider range of models than previously possible. Through a series of methodological contributions, we demonstrate that compression ratios of more than a thousand times are achievable in extreme cases, while maintaining or improving performance in many scenarios. We explore several dimensions of policy compression to accomplish this, including the reduction of parameter count, precision, and the frequency of decision-making.

First, we introduce a novel distillation loss for actor-critic models that leverages the state-value predictions made by the teacher's critic as an auxiliary task. This approach enables

the student model to develop more informative internal representations, evidenced by distinct clusters that form in the student's network activations, corresponding to different ranges of expected cumulative rewards that can still be obtained from a given state. This improvement yielded up to 7% higher average returns compared to traditional distillation methods.

In addressing the challenges of low-precision deployment, we develop Quantization-aware Policy Distillation (QPD), which enables the stable training of policy networks with 8-bit integer parameters. We first show how training low-precision policies directly using randomly initialised weights is practically infeasible due to the steep and inaccurate optimization landscape. Instead, we propose a three-phase training process that first distils the teacher's policy into a full-precision student, which is then quantized using Post-Training Quantization (PTQ) to serve as initialization for a final Quantization-Aware Training (QAT) phase to recover from the quantization errors and the non-linear distribution shift. Along with additional improvements to the optimization stability, we demonstrate that QPD can be used to quantize models that perform comparably to their full-precision counterparts, as long as the quantization process is carefully managed.

We then demonstrate the importance of preserving policy entropy during compression in the context of continuous control tasks. By distilling both the mean action predictions and standard deviations of the teacher's policy, we achieve up to 36% higher returns compared to baseline methods that only transfer the mean actions. This contribution highlights how maintaining the stochastic nature of the original policy is crucial for retaining the most effective policies after distillation, especially for algorithms such as SAC that operate in continuous action spaces.

This thesis also introduces temporal distillation, a novel compression paradigm that reduces the computational cost of policy execution by enabling the student to predict sequences of repeated actions with a single decision, effectively learning when a different action will be required. Temporal distillation needs up to 462% fewer decisions to complete the same task, while maintaining or improving task performance, demonstrating that efficiency gains can be achieved not only through parameter reduction but also through intelligent action selection.

Our research further explores how compression affects a model's ability to adapt to new tasks, particularly in the context of pruning. We find that, while pruning can negatively impact adaptability, this effect can be mitigated through informed pre-training and task-relevant pruning methods that maintain the model's general features. This contribution extends beyond DRL to broader applications in machine learning deployment.

For partially observable environments, we demonstrate how to compress policies with Recurrent Neural Network (RNN) architectures by preserving the temporal dependencies between transitions in the replay memory used for training a student policy. This enables the student to learn an effective hidden state representation without directly emulating the teacher's memory mechanism, achieving a 40x reduction in size with only a 3% drop in average return.

The practical impact of these methods is demonstrated through comprehensive runtime performance benchmarks, showing potential decreases in average action execution time of up to 1487x. These improvements are achieved through a combination of reducing parameter count, architectural complexity, numerical precision, and decision frequency.

Real-world applications are showcased through the compression of a data-driven network function replica scaler and a photo-realistic embodied navigation task.

In conclusion, this thesis makes significant progress toward enabling the deployment of complex DRL models on highly resource-constrained edge devices. The developed methods demonstrate that through careful consideration of DRL-specific characteristics, it is possible to achieve substantial efficiency gains while maintaining or improving policy effectiveness. In addition, we enabled specialised compression of policies with actor-critic and recurrent network architectures, as well as for continuous action spaces, thereby expanding the range of models that can be effectively compressed. These contributions open new possibilities for the deployment of intelligent systems in scenarios where real-time sequential decision-making is crucial, but computational resources are limited, such as autonomous robotics and network management systems.

Samenvatting

De wijdverspreide adoptie van *Deep Reinforcement Learning (DRL)* heeft geleid tot opmerkelijke prestaties in verschillende domeinen, van het beheersen van complexe spellen tot het besturen van robotische systemen. Het inzetten van deze krachtige modellen op apparaten met beperkte middelen blijft echter een uitdaging vanwege hun aanzienlijke rekenkracht, geheugen voetafdruk en energieverbruik. Traditionele benaderingen van dit probleem, zoals het verplaatsen van de berekeningen naar de cloud of de afhankelijkheid van hardwareverbeteringen, kennen aanzienlijke beperkingen. Vertrouwen op de cloud voor het uitvoeren van de nodige berekeningen introduceert onaanvaardbare vertraging voor taken die afhankelijk zijn van een lage reactietijd en het vereist constante netwerkconnectiviteit. Verbeteringen aan de hardware alleen kunnen de toenemende complexiteit van DRL-modellen niet bijhouden. Bovendien lukt bij het direct trainen van kleine modellen vaak niet om het gewenste prestatieniveau te bereiken, vanwege het inherente verschil tussen de capaciteit die nodig is voor training en inzet van deze modellen.

Om deze uitdagingen aan te pakken, zijn er model compressietechnieken ontwikkeld om de grootte en rekenkosten van DRL-modellen te verminderen. Model compressie omvat meestal eerst het trainen van grote modellen die complexe taken het meest effectief kunnen leren. Vervolgens ondergaan deze modellen een compressie proces om hun grootte en rekencomplexiteit te verminderen, waardoor de moeilijkheden van directe training van kleinere modellen worden omzeild en ze geschikt worden gemaakt voor inzet op randapparaten. Bij policy distillatie wordt een klein student model getraind om het gedrag van een groter leraar model na te bootsen. Pruning, daarentegen, houdt in dat redundante verbindingen of neuronen uit een volledig getraind model worden verwijderd, waardoor het aantal parameters wordt verminderd. Ten slotte vermindert kwantisatie de numerieke precisie van de gewichten en activeringen van het model, waardoor het gebruik van rekenwerk in lage precisie tijdens inferentie mogelijk wordt.

Model compressie voor DRL netwerken is echter nog relatief weinig bestudeerd, waarbij de meeste onderzoeken zich richten op het aanpassen van methoden voor begeleid leren zonder de unieke kenmerken van DRL volledig te benutten. Deze kloof beperkt de mogelijkheden om kleinere, efficiëntere DRL-agenten aan de rand in te zetten. Bovendien zijn niet alle DRL netwerken compatibel met bestaande compressiemethoden, wat de ontwikkeling van nieuwe technieken vereist die zijn afgestemd op hun specifieke vereisten.

Deze thesis stelt de hypothese dat door het begrijpen en benutten van de unieke eigenschappen van DRL-modellen, nieuwe compressiemethoden kunnen worden ontwikkeld, die hogere compressieverhoudingen bereiken en het comprimeren van een breder scala aan modellen mogelijk maken dan voorheen mogelijk was. Door middel van een reeks methodologische bijdragen tonen we aan dat compressieverhoudingen van meer dan duizend keer haalbaar zijn in extreme gevallen, terwijl de prestaties in veel scenario's

behouden blijven of verbeteren. We verkennen verschillende dimensies van DRL netwerk compressie om dit te bereiken, waaronder het verminderen van het aantal parameters, precisie en de frequentie van besluitvorming.

Ten eerste introduceren we een nieuwe *loss* functie voor het distilleren van *actor-critic* modellen, die gebruik maakt van de schattingen van de waarde van een bepaalde staat in de omgeving, gemaakt door de *critic* van de leraar, als een aanvullende taak. Deze benadering stelt het student model in staat om meer informatieve interne representaties te ontwikkelen, wat blijkt uit de clusters die ontstaan in de netwerkactivaties van de student, die overeenkomen met verschillende bereiken van verwachte cumulatieve beloningen die nog kunnen worden behaald vanuit een bepaalde staat. Deze verbetering leverde gemiddeld tot 7% hogere cumulatieve beloningen op, vergeleken met traditionele distillatie methoden.

Bij het aanpakken van de uitdagingen rond de inzetting van lage precisie ontwikkelen we de *Quantization-aware Policy Distillation (QPD)* methode, waarmee het stabiel trainen van DRL netwerken met 8-bit parameters mogelijk wordt gemaakt. We laten eerst zien hoe het direct trainen van DRL netwerken in lage precisie met willekeurig geïnitialiseerde gewichten praktisch onhaalbaar is vanwege het steile en onnauwkeurige optimalisatie landschap. In plaats daarvan stellen we een driestaps trainingsproces voor waarbij eerst het gedrag van de leraar worden gedistilleerd in een student met volledige precisie, die vervolgens wordt gekwantiseerd met behulp van *Post-Training Quantization (PTQ)* om te dienen als basis voor een laatste *Quantization-Aware Training (QAT)*-fase om te herstellen van kwantisatie fouten en de niet-lineaire verschuiving in distributie. Met extra verbeteringen in de optimalisatie stabiliteit tonen we aan dat QPD kan worden gebruikt om modellen te kwantiseren, die vergelijkbare prestaties leveren als hun tegenhangers met volledige precisie, zolang het kwantisatie proces zorgvuldig wordt beheerd.

Vervolgens tonen we het belang aan van het behouden van de entropie in het gedrag van het netwerk tijdens compressie, in de context van continue beheerstaken. Door zowel de gemiddelde actie voorspellingen als standaarddeviaties van het gedrag van de leraar te distilleren, bereiken we tot 36% hogere cumulatieve beloningen vergeleken met basismethoden, die alleen de gemiddelde acties overbrengen. Deze bijdrage benadrukt hoe het behoud van de stochastische aard van het oorspronkelijke gedrag cruciaal is voor het behouden van de meest effectieve strategie na distillatie, vooral voor algoritmen zoals SAC die in continue actie ruimten werken.

Deze thesis introduceert ook temporele distillatie, een nieuw compressie paradigma dat de rekenkosten van het uitvoeren van een DRL model vermindert door de student in staat te stellen om reeksen van herhaalde acties te voorspellen met één beslissing, waardoor effectief wordt geleerd wanneer een andere actie nodig zal zijn. Temporele distillatie heeft tot 462% minder beslissingen nodig voor dezelfde taak, terwijl de prestaties behouden blijven of verbeteren. Hiermee wordt aangetoond dat efficiëntiewinsten kunnen worden behaald niet alleen door parameter reductie, maar ook door intelligente actiekeuze.

Ons onderzoek bekijkt verder hoe compressie de aanpasbaarheid van een model aan nieuwe taken beïnvloedt, met name in de context van pruning. We constateren dat hoewel pruning een negatieve invloed kan hebben op de aanpasbaarheid, dit effect kan worden beperkt door geïnformeerde *pre-training* en taakrelevante pruning-methoden die de algemene herkenningsmechanismen van het model behouden. Deze bijdrage gaat verder dan DRL en heeft bredere toepassingen binnen machinaal leren (ML).

Voor omgevingen die slechts gedeeltelijk observeerbaar zijn laten we zien hoe DRL netwerken met terugkerende architecturen (RNN) kunnen worden gecomprimeerd door de temporele afhankelijkheden tussen de overgangen in het terugkijk-geheugen te behouden, dat wordt gebruikt voor het trainen van een student netwerk. Dit stelt de student in staat om een effectieve verborgen staat-representatie te leren zonder direct het geheugensysteem van de leraar na te bootsen, wat leidt tot een reductie in grootte met een factor 40 en slechts 3% daling in gemiddelde cumulatieve beloningen.

De praktische impact van deze methoden wordt aangetoond door middel van uitgebreide benchmarks voor *runtime*-prestaties, die potentiële afnames in de gemiddelde uitvoertijd van een actie laten zien tot 1487x. Deze verbeteringen worden bereikt door een combinatie van het verminderen van het aantal parameters, de architecturale complexiteit, de numerieke precisie en de frequentie van beslissingen. Reële toepassingen worden gedemonstreerd door de compressie van een datagestuurde netwerkfunctie replica schaler en een foto-realistische navigatie taak.

Samengevat boekt deze thesis aanzienlijke vooruitgang in het mogelijk maken van de inzet van complexe DRL-modellen op randapparaten met sterk beperkte middelen. De ontwikkelde methoden tonen aan dat het, door zorgvuldig rekening te houden met de DRL-specifieke kenmerken, mogelijk is aanzienlijke efficiëntiewinsten te behalen terwijl de effectiviteit van het gedrag behouden blijft of verbetert. Daarnaast hebben we gespecialiseerde compressie mogelijk gemaakt van netwerken met *actor-critic* en terugkerende architecturen, evenals voor continue actie ruimten, waardoor het bereik van modellen, die effectief kunnen worden gecomprimeerd, wordt uitgebreid. Deze bijdragen openen nieuwe mogelijkheden voor de inzet van intelligente systemen in situaties waarin sequentiële beslissingen met hoge actiesnelheid cruciaal zijn, maar waar de rekencapaciteit beperkt is, zoals autonome robotica en netwerkbeheer.

Contents

List of Acronyms

A2C Advantage Actor-Critic xii, xxi, xxii, 25, 28, 29, 32, 43–53, 57, 60–62, 94, 144, 150

A3C Asynchronous Advantage Actor-Critic 94

Adam Adaptive Moment Estimation 20, 30, 32, 57, 118

AI Artificial Intelligence i, xxi, 17, 18, 123

AMR Autonomous Mobile Robot 66, 84

AN Autonomous Network 124

AP Access Point 65, 125

BERT Bidirectional Encoder Representations from Transformers 115

CNN Convolutional Neural Network 3, 19

CPU Central Processing Unit 52, 83, 126–128, 130–133

D4PG Distributed Distributional Deterministic Policy Gradients 29, 57

DAR Dynamic Action Repetition 94, 95

DD-PPO Decentralized Distributed Proximal Policy Optimization 138

DDPG Deep Deterministic Policy Gradient 57

DL Deep Learning xxi, 3, 4, 15, 17, 18, 26, 33, 58, 70, 113

DNN Deep Neural Network xxi, 1, 3, 5, 6, 17–20, 25–28, 30, 33, 34, 39–41, 56, 66, 84, 91, 110, 113, 128

DQN Deep Q-Network xxi, 7, 8, 26, 28–34, 37, 43–50, 52, 53, 57, 61–63, 66, 94, 115, 144–146, 149, 150

DRL Deep Reinforcement Learning iii–v, vii–ix, xxi, xxiv, 1–12, 15–18, 20, 21, 26–29, 34, 36, 43, 46, 53, 55–59, 62, 65–67, 72, 73, 84, 85, 91, 110, 111, 114, 115, 123–127, 133, 135, 136, 141, 143, 148–153

FiGAR Fine Grained Action Repetition 95

FPS Frames Per Second 51–53

GAE Generalized Advantage Estimation 32

GNN Graph Neural Networks 126

GPU Graphics Processing Unit xxiii, 3, 19, 52, 53, 55, 83, 101, 109, 137

GRU Gated Recurrent Unit 138

HRL Hierarchical Reinforcement Learning 150, 151

ICT Information and Communications Technology 124

IMP Iterative Magnitude Pruning xxiv, 114, 115, 117–121, 147

IoT Internet of Things 1, 55, 84, 91

KD Knowledge Distillation 8, 11, 17, 33, 34, 51, 57, 80, 103

KL Kullback-Leibler 10, 34, 37, 46, 53, 61, 68, 70, 76–82, 84, 85, 97, 150

KPI Key Performance Indicator 125

LR Learning Rate 20, 26, 47, 101, 128

LRP Layer-wise Relevance Propagation xxiv, 13, 41, 42, 114, 115, 117–121, 147

LSTM Long Short-Term Memory 138

MDP Markov Decision Process 9, 17, 21, 22, 24, 26, 124, 126

ML Machine Learning viii, xxi, 1, 3–5, 17, 18, 20, 124

MLP Multi-Layer Perceptron 18, 19, 97

MSE Mean Squared Error xxii, 19, 32, 33, 67–69, 76, 97

NF Network Function xxiv, 13, 16, 65, 123–125, 127, 153

NN Neural Network 18

NPU Neural Processing Unit 3, 19

NS Network Service 124–126

ONNX Open Neural Network Exchange 101, 132

PD Policy Distillation xxi–xxiii, 7, 8, 10, 12, 13, 34–37, 43, 44, 46, 47, 49–51, 53, 56–58, 60, 62, 66–70, 72, 80, 84, 85, 89, 93, 94, 96, 99, 102, 104, 110, 111, 115, 123, 128, 133, 135, 136, 139–141, 144–146, 148, 149

PLAID Progressive Learning and Integration via Distillation 68

POMDP Partially Observable Markov Decision Process 9, 13, 21, 28, 136

PPO Proximal Policy Optimization xxii, xxv, 7, 8, 25, 28, 29, 31, 32, 43, 44, 46–50, 52, 53, 57, 61, 62, 66, 73, 74, 79–81, 83, 85, 100, 115, 128, 129, 132, 133, 138, 144, 150

PTQ Post-Training Quantization iv, 10, 12, 37, 38, 55, 57–60, 62

QAT Quantization-Aware Training iv, xxii, 7, 11, 12, 38, 39, 55, 57–60, 62

QPD Quantization-Aware Policy Distillation iv, viii, xxii, 12, 56–63, 145, 146, 148, 151

QR-DQN Quantile Regression Deep Q-Network 29

ReLU Rectified Linear Unit 18, 97, 99

ResNet Residual Network 138

RL Reinforcement Learning xxi, 1, 4, 5, 8, 11, 13, 15, 17, 18, 20, 21, 24–26, 32, 36, 42, 58, 59, 65, 67, 91, 92, 94, 118, 124, 126, 129, 135, 137, 138

RMSprop Root Mean Square Propagation 20, 30, 101

RNN Recurrent Neural Network iv, ix, xxiv, xxvi, 9, 13, 16, 19, 29, 100, 123, 135–140, 144, 145, 150, 151

ROM Read-Only Memory 84

RSS Resident Set Size 132, 133

RTT Round-Trip Time 127

SAC Soft Actor-Critic iv, viii, xxii, xxv, 8, 11, 28, 29, 32, 33, 57, 66, 73, 74, 76–83, 85, 86, 150

SARSA State-Action-Reward-State-Action 25

SB3 Stable Baselines3 46–48, 50, 51, 73, 74, 100, 128

SDN Software-Defined Networking xxiv, 13, 65, 124, 135

SGD Stochastic Gradient Descent 20, 30

SL Supervised Learning 4–8, 11, 16, 20, 34, 36, 55, 57, 58, 80, 114

SLA Service Level Agreement 126–128, 130

STE Straight-Through Estimator 39, 58

TD Temporal Difference 24, 26, 30, 31

TD3 Twin Delayed Deep Deterministic Policy Gradient 28, 67

tinyML Tiny Machine Learning 5

UMAP Uniform Manifold Approximation and Projection xxi, 48, 49, 53

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Context

1.1.1 Reinforcement Learning at the Edge

Imagine a small autonomous drone tasked with monitoring crops in a vast agricultural field. The drone must navigate around obstacles, avoid birds, and adjust its flight pattern based on wind conditions. To make the best decisions in real-time, the drone could rely on RL (Reinforcement Learning) - a type of ML (Machine Learning) where an agent learns to take sequential actions by interacting with the environment [131]. However, for a device like this drone, which operates with limited battery life and computational power, running a complex RL model is challenging. While RL has demonstrated success in solving complex sequential decision-making problems, from game playing [94] to robotic control [71], deploying RL on resource-constrained devices like drones, robots, or IoT (Internet of Things) sensors requires overcoming significant hurdles [83].

One of the key challenges that limits us from applying RL to constrained environments is the high computational and energy cost associated with running large models. This is especially true for DRL (Deep Reinforcement Learning) algorithms, where DNNs (Deep Neural Networks) are used to learn and model behaviour, which tend to be highly resource intensive [115]. As we expect these DRL agents to grow in their capabilities, so do the complexity and size of the DNNs that power them. For instance, a typical DRL model might involve millions of parameters and require significant memory, processing power, and battery consumption [93]. In devices like autonomous robots or drones, where size, weight, and energy consumption are tightly constrained, deploying such a model could severely limit the system's operational capacity or render it entirely impractical.

Additionally, real-time decision-making is crucial in many RL applications. A robot navigating through a crowded warehouse, avoiding other robots and human workers, must process sensory data and take actions quickly to avoid accidents [44]. This requires the RL model to not only be accurate but also highly efficient in computation. Large, uncompressed models often introduce latency, which can lead to slow or delayed responses, an unacceptable risk in many practical settings.

1.1.2 Limitations of Cloud Offloading

The current most common solution for overcoming these limitations is to offload computationally intensive tasks to the cloud [106]. In this approach, each time the device needs to make a decision, it needs to transmit its entire observation of the environment, gathered through its sensors, to the cloud. There, a server equipped with more computational resources runs the complex DRL model, analysing the situation, and sends back the chosen action for the device to execute, as illustrated in Figure 1.1. While this may reduce the burden on resource-constrained devices, it introduces new challenges. The latency involved in transferring data to and from remote servers can hinder the performance of real-time applications even more than running the large model locally [140]. Privacy concerns also arise when sensitive data needs to be transmitted and processed off-device [154].

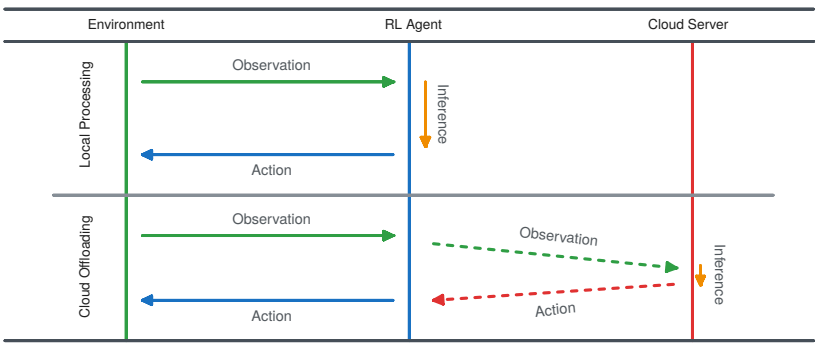


Figure 1.1: A diagram of the trade-off between local computation and cloud offloading.

Furthermore, the dependence on a constant stable and fast network connection poses practical challenges that severely limit the applicability of this approach. In scenarios where network coverage is unreliable, intermittent, or non-existent, such as in remote areas or during disasters, the entire system functionality can easily be disrupted. Even in areas with generally good network coverage, a momentary loss of connectivity can lead to critical delays or failures in real-time applications. This issue is particularly pronounced in mobile scenarios, where the device may transition between different network environments with varying quality and reliability. It also limits the potential scalability. As the number of devices deployed in an edge network increases, the network congestion grows and the latency rises. The quality of the network connection becomes the determining factor in the system's performance, making the operation unpredictable and unreliable.

Outside the practical limitations, there are also substantial financial and environmental concerns associated with cloud offloading. Cloud services, particularly for compute-intensive tasks like DRL, can be expensive to scale [3], making it a less sustainable option for long-term deployment. Moreover, shifting computation to the cloud doesn't fully address energy concerns, especially in the context of growing awareness around climate change. Data centres consume vast amounts of energy and contribute significantly to global carbon emissions [124].

While this approach can reduce the local energy consumption of the device itself, it simply shifts the burden to the cloud infrastructure and introduces additional overhead associated with data transmission and processing, exacerbating the overall impact. Therefore, while cloud offloading can temporarily address the computational demands, it is not a viable long-term solution for all scenarios, further emphasising the need for localised model optimisation techniques.

1.1.3 Reliance on Hardware Improvements

In the past, advances in hardware technology played a significant role in enabling more powerful and efficient computing systems. As processors became faster and more energy-efficient, software could grow in complexity without a proportionate increase in resource demand. This progress was largely driven by Moore's law, which predicted that the number of transistors on an integrated circuit would double approximately every two years with a minimal rise in cost [95], leading to exponential growth in computational power. However, we can no longer solely rely on these hardware improvements to sustain the growing computational needs of modern algorithms, particularly in fields like ML (Machine Learning) and DRL.

The physical limits of chip design have become a significant barrier to further advancements. As transistors shrink to the nanometre scale, issues like heat dissipation, leakage currents, and quantum effects [103] make it increasingly difficult to continue scaling hardware performance at the same rate. This has led to a slowdown in the once-reliable trend of Moore's law [139]. Chip manufacturers are struggling to pack more transistors onto a chip while keeping power consumption and heat manageable [31]. As a result, the gains in computational power that were once guaranteed by hardware evolution are diminishing, forcing researchers to look elsewhere for improvements.

To address this, hardware accelerators specifically designed for DNNs have been developed (NPUs (Neural Processing Units)) that encode the unique properties of common DL (Deep Learning) operations into the architecture of dedicated hardware, instead of relying on general-purpose processors [81]. Examples are the DNN inference coprocessors found in recent chips from AMD [4], Intel [60], or Qualcomm [107], and even specialised microcontrollers like the MAX 78000 from Analog Devices [5]. While the transition to specialised hardware architectures offers significant potential for enhancing the speed and energy efficiency of DNN computations [81] - much like the use of GPUs for computer graphics (and DL as well) - and we can anticipate substantial further progress in this field, these advancements are ultimately constrained by the same physical limitations.

The larger NPU coprocessors are equally unsuitable for deployment on low-power devices like drones or mobile robots, with a power consumption of up to 54W [4], without providing full-precision support for all DNN operations [60]. And while the specialised microcontrollers are more energy-efficient, they are still heavily constrained by the maximal precision and number of DNN parameters they can handle[1], limiting their applicability to small models. While this certainly increases the viability of deploying DRL models on the edge when combined with techniques for training smaller models, it is no silver bullet on its own.

[1]The ADI MAX 78000 supports up to 442,000 8-bit CNN parameters, with an input size of 1024x1024 [5].

Another limitation comes from the relative stagnation of battery technology. While low-cost embedded computing devices have still benefited from large advancements in their processing power, the energy storage capabilities of batteries, in terms of energy density and longevity, have only seen incremental improvements over the past years [156]. For resource-constrained devices like drones and robots, this means that the energy supply is often a bottleneck in system performance [138]. The increasing demand for computation, coupled with comparably smaller improvements in energy storage, presents a significant challenge for deploying complex ML algorithms in real-world settings.

Given these constraints, relying on hardware advancements alone to meet the demands of DRL applications is no longer feasible. This highlights the need for innovative approaches from the DL perspective to achieve efficiency gains within the existing physical and energy constraints.

1.1.4 The Need for Large Policy Networks

A potential naïve solution to address the challenges of deploying DRL models on resource-constrained devices would be to improve RL algorithms for directly training smaller models that do satisfy these constraints. Unfortunately, some intrinsic properties of the DRL training process prevent such small models from properly learning effective behaviour (or policy) through interactions with the environment, even if they theoretically have the capacity to model this learned behaviour after training [115, 26].

Due to the trial-and-error nature of the RL learning process, the model must explore many different decisions (or actions) in numerous states of the environment to find which strategies are the most effective based on a reward signal [131]. Most algorithms directly learn some notion of how valuable it is to be in these states of the environment, and then use this to guide their actions [93, 122, 92]. This includes exploring and learning about strategies that do not directly lead to the best possible outcome, but which are necessary to converge on the best policy in the long run (illustrated in Figure 1.2). To solve complex tasks, models need to be able to accurately represent small differences in the values between various states and actions during training, even if these will no longer be encountered when following the final policy. This causes an inherent discrepancy between the model capacity needed to learn during training and the capacity required to represent the learned policy after training.

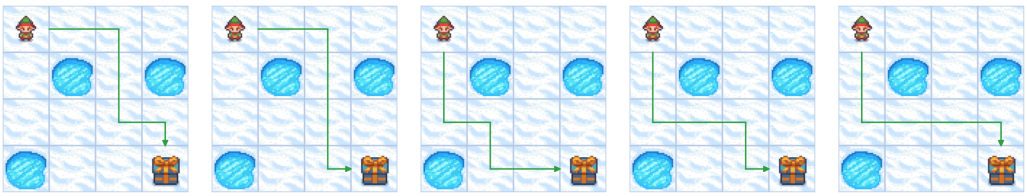


Figure 1.2: An example of alternative strategies in the FrozenLake environment.

Additionally, optimisation in DRL is notoriously less stable than in SL (Supervised Learning), and this instability is further amplified when training smaller models due to a more complex loss landscape with increased local minima [151]. To mitigate these challenges,

DRL models are often significantly overparameterized, at the cost of memory and computational demands [101, 98]. Furthermore, training DRL models is typically not very sample efficient, needing extensive environment interactions [82, 130]. Larger models, while resource-intensive during deployment, can be more sample efficient during training, learning effectively with fewer interactions [150]. This sample efficiency is especially crucial when training in real-world scenarios where each interaction can be costly.

The problem context can thus be summarised as follows:

1. We need RL models at the edge, but they are too large, slow and energy-intensive.
2. Offloading to the cloud is not a viable solution due to latency, privacy, environmental, and reliability concerns.
3. We can no longer rely solely on potential hardware improvements, which are slowing down for both battery density and computational efficiency. Hardware accelerators for DNNs are promising, but are still constrained by the same limitations, and need to be combined with other solutions to be effective.
4. Training small enough models directly is infeasible due to optimisation instability and the discrepancy between the capacity needed during training and inference.

1.1.5 Policy Compression as a Solution

Given the challenges outlined above, a promising approach to deploying efficient DRL models on resource-constrained devices is model compression. This technique allows us to leverage the power of large models for effective training while ensuring they are suitable for real-time deployment [22]. Instead of directly training small models, which as discussed earlier can be problematic, model compression initially focuses on training large, overparameterized models that are capable of learning complex tasks most effectively [23]. This way, the constraints imposed by the available resources at the edge are not limiting the learning capabilities of the model during training, and we can start with the best possible policy as a basis for further energy optimisation. It also enables the compression of existing models, making them more efficient without the need to train them from scratch [29].

Once these models have been trained, they undergo a compression process to reduce their size and computational complexity, bypassing the difficulties of training smaller models and making them suitable for deployment on edge devices [115]. This approach offers the best of both worlds: leveraging the power of large models for effective learning and the reduced size, memory footprint, and computational requirements of small models for real-time edge deployment, while maintaining their performance at solving the tasks they were trained on [26].

Model compression is not unique to DRL and has been extensively studied in the realm of SL [90, 69, 129, 146]. In fact, bringing the power of large DNNs to the edge, including through model compression, is the main goal in the domain of tinyML. The EDGE AI (formerly tinyML) Foundation [35] defines this as the subfield of ML that focuses on technologies and applications, including hardware, algorithms, and software, capable of

performing on-device sensor data analytics at extremely low power, typically in the mW range and below.

However, due to the disparity between the capacity required during training and inference in DRL, as noted previously, model compression techniques - which we will refer to as policy compression in this context - prove to be especially beneficial. Once a model has converged to a stable policy, much of the intricate environment exploration knowledge becomes redundant [115]. On the other hand, the unique properties of DRL models, such as the lack of a fixed dataset and the need for exploration, introduce new challenges that require tailored solutions, rather than simply applying existing solutions from SL in this context.

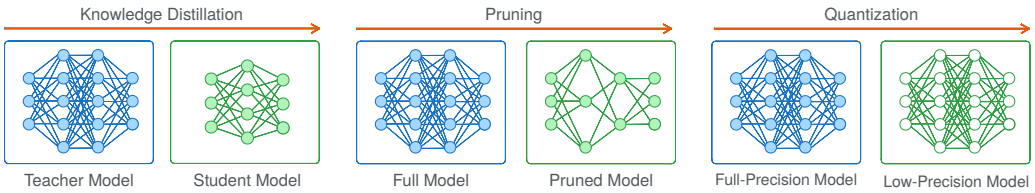


Figure 1.3: An illustration of the different model compression methods.

The challenge is to extract and retain only the essential parts of the policy, enabling it to make the same decisions with fewer parameters and consequently lower computational cost. Due to the black-box nature of DNNs and the complex interactions between the model's parameters, this is a non-trivial task, with several potential techniques that approach this problem from different perspectives. The three main model compression categories that address this challenge are as follows:

Knowledge Distillation involves training a smaller (student) model to mimic the behaviour of the larger (teacher) model after training by matching its output distribution when given the same inputs [52].

Pruning reduces the size of the large model by removing redundant connections or neurons, based on their weights or activations [79].

Quantization reduces the precision of the model's weights and activations, allowing for more efficient computation and reduced memory usage [73].

These methods are illustrated in Figure 1.3 and described in more detail in section 2.3. This concept enables a shift from relying solely on an increase in raw computational power to improve model performance, towards more intelligent and efficient use of existing resources. If applied successfully, policy compression has the potential to enable the efficient deployment of DRL models in highly resource-constrained settings, unlocking a wide range of applications that were previously infeasible.

1.2 Problem Statement & Hypothesis

Although model compression has shown extensive promise in the context of supervised classification tasks, its adaptation for DRL is still in its relative infancy. Existing research has primarily focused on translating model compression methods from SL to DRL, without fully exploiting the unique properties of DRL algorithms.

For example, PD (Policy Distillation) [115] has successfully shown how the same loss function originally proposed by Hinton et al. [52] in SL can be applied to the distillation of DQN (Deep Q-Network)-based DRL agents and this was later also confirmed to work for PPO (Proximal Policy Optimization) agents by Green et al. [43], which are two of the most popular DRL algorithms. But neither of these methods exploit the inherent differences between classification and DRL in the meaning of the network outputs, or between DQN and PPO in the type of output values and network architectures. The student is trained similarly to a classifier, by simply converting the outputs to a probability distribution over the possible actions, losing some additional information they contain.

An initial paper that applied pruning for DRL policies was presented by Livne et al. [84]. But here, the pruning step is not directly used for compression, but rather to estimate a good layer size for a new model, which is then trained using PD. In a sense, this is more a form of neural architecture search than policy compression, with the actual compression aspect still happening through distillation.

The authors of QuaRL [74] proposed the use of quantization for DRL policies as part of their ActorQ framework. This is a distributed actor-learner training system, in which the learner updates a full-precision model, and the actors use a quantized version of this model to more efficiently interact with the environment to obtain new training data. So while the quantization is used to speed up exploration during training, where some degree of inaccuracy is more tolerable, it is not used to optimise a final low-precision model for deployment.

We therefore observe a gap in the research on policy compression, where if the unique properties of DRL models would be more explicitly considered, even smaller and more efficient agents could become attainable. Additionally, not all types of policies are as compatible with existing methods that were originally proposed to compress classification models. This results in some DRL models simply not being compressible without developing new methods tailored to their different characteristics. Concretely, we highlight the following unique properties of DRL algorithms that present opportunities for improvement or even require novel compression methods to be developed:

Actor-Critic Architectures (1): Unlike supervised classification, where the network is optimised to predict a single target, actor-critic architectures in DRL (such as PPO [122]) consist of two distinct outputs: one for the policy (actor) and one for the state value function (critic). Only the actor is necessary during inference, but the critic still contains valuable information that could be exploited during compression, and would enable on-device fine-tuning if preserved during the compression phase. Some compression methods, such as QAT (Quantization-Aware Training), even operate during training [69], where the critic is still essential for the agent's learning process. These methods could be optimised by handling the two heads separately.

Value or Policy Based (2): DRL models can be divided into value-based (e.g., DQN [94]) and policy-based (e.g., PPO [122]) methods, each having different structures and output types. Value-based models focus on predicting the expected cumulative reward when taking different actions, while policy-based models directly model a probability distribution over actions [131]. These types of outputs have different distribution characteristics, and inherent properties and meanings that need to be accounted for and could be exploited during the compression process, resulting in a potential specialisation depending on whether a value function or probability distribution is the primary action selection mechanism. This is especially relevant for scenarios where the model needs to be fine-tuned or retrained after compression, in which case these inherent properties of the output values need to be preserved during compression.

Stochastic Policies (3): Some DRL policies, such as for DQNs, are inherently deterministic, meaning that they always output the same action for a given state [94]. But others, like for PPO [122] or SAC (Soft Actor-Critic) [47], output a probability distribution over the actions, allowing for stochastic behaviour. These stochastic policies can encourage exploration and be more robust to noise or changes in the environment [47]. Existing methods have typically focused on ensuring that the best action for a given state remains unchanged after compression [115], but not all policies perform best when applied deterministically. Moreover, stochastic policies are typically better suited for stochastic environments, where taking the same action in the same state might not always lead to the same outcome [131]. Therefore, compression techniques should be developed that preserve the stochasticity of these policies.

Continuous Actions (4): Many RL problems involve discrete action spaces, where the agent chooses between a finite set of actions by predicting a value or probability for each [94]. This is similar to classification tasks in SL, where the model typically outputs a probability for each class [52]. But some tasks, such as commonly used in robotics continuous control, are modelled with continuous action spaces [135]. There, the model must output a continuous value for each action dimension, which requires a different approach to compression. In PD, for example, the same loss function is used as in supervised KD (Knowledge Distillation) [52], but this is not directly applicable to continuous action spaces. This gap in the compression literature should be addressed by developing novel compression methods that can handle DRL models for continuous actions.

Exploration (5): In supervised classification, the dataset typically remains fixed throughout the training process, and the model is trained to minimise the error between its predictions and the ground truth labels. But in DRL, the training data is generated based on the interactions between the agent and the environment, and depends on the actions selected by a policy that evolves over time [131]. The trade-off between exploration and exploitation is a fundamental aspect of RL, where the agent must balance between trying out new actions to learn more about the environment and exploiting known strategies to maximise returns [131]. Since compression techniques are typically applied after training, the policy has converged to a relatively stable state that maximises exploitation. But by modifying the policy behaviour through compression, even if only slightly, we shift the data distribution that the model will be exposed to during deployment [26]. Compression methods should account for and minimise this distribution shift to ensure that the agent's effectiveness is not negatively impacted. Additionally, in the case of distillation, when transferring

knowledge from a teacher to a student policy, different strategies should be developed to provide the student with the most informative training data that ensures the generalisation capabilities of the teacher are maintained.

Sequential Nature (6): Another unique aspect of DRL is that agents make decisions sequentially, where current actions influence future states and rewards. For regular MDPs (Markov Decision Processs) this is usually not a problem, as the environment is fully observable and the Markov or "memoryless" property holds. This means that the current state contains all the information needed to make an optimal decision, and the history of previous states and actions is irrelevant [131]. But in partially observable environments (POMDP), the agent must often maintain some form of memory or internal state to keep track of past observations and actions [131]. Developing compression techniques that can handle this time-dependency, and potentially exploit it for increased efficiency, is another area where existing methods fall short. The ability to compress or distil to architectures that include RNN (Recurrent Neural Network) layers, for example, is essential for these types of environments. Additionally, it opens up the opportunity for another way to improve efficiency, by reducing the number of required decisions to be made for a given task.

Although it is not feasible to revolutionise the field of policy compression in a single thesis, and addressing the full range of challenges and opportunities outlined above for all types of model compression would be an ambitious goal, we do strive to make valuable contributions in each of these areas. Based on these observations, we formulate the following research hypothesis:

Hypothesis

By leveraging the six unique properties of DRL models identified above, more effective policy compression methods can be developed that:

1. Achieve higher compression rates by minimising the number and precision of parameters in the model and reducing how many decisions are needed to complete a task, while maintaining or even exceeding the agent's original effectiveness.

2. Enable the compression of a wider range of DRL models, including those with actor-critic and RNN-based architectures, continuous action spaces, and highly stochastic policies.

1.3 Research Questions

In order to validate the formulated hypothesis, we focus on the subset of previously outlined research opportunities with the highest potential, and define the following research questions:

1. **Can we extract auxiliary knowledge from DRL models that enables a student policy to learn a more informative representation of its environment?** In traditional PD, students learn to replicate the policy of the teacher based on random samples of (observation, action distribution) pairs. This is indeed the most intuitive way to transfer knowledge from one policy to another, as this is theoretically the only information that is required to make decisions and complete the task. However, as highlighted above, DRL models have some unique properties compared to models for supervised classification that make compression more challenging, but also result in additional information that could be utilised. Prime targets for this are the value function in actor-critic models and the sequential nature of the decisions made by the agent that only when combined forms a complete behavioural strategy. The question then becomes how to transform this additional information into a more informative training signal for the student policy, without introducing increased complexity that harms the compression potential.

2. **Are certain ways in which a policy can be represented inherently more compressible than others?** The same distillation loss based on the KL (Kullback-Leibler)-divergence has been applied for both value-based and policy-based models, but the distribution of the output values is fundamentally different. A closer look at how these differences impact the distillation loss would provide insights into whether the loss function should be adapted to achieve optimal performance. Additionally, empirical evidence suggests that learning an accurate value function requires more parameters than learning the policy, so more capacity is often allocated to the critic compared to the actor in actor-critic models [109]. It remains to be seen whether this also causes policies that rely on action-value estimates to be less compressible than those that directly output a probability distribution over actions, or if the transformation of these values by applying the softmax function in the distillation loss [115] mitigates this difference. Finally, in policies for continuous action spaces, where the output consists of a continuous value for each action dimension, an entirely distinct approach to distillation is required. Before the compression potential of these different types of policies can be accurately assessed however, new distillation methods tailored to these specific output types need to be developed.

3. **How detrimental is the reduced representational power of low-precision parameters to the effectiveness of a policy?** With the goal of deploying DRL models at the edge on low-power embedded devices, which often lack the capability to perform high-precision arithmetic [5], quantization becomes a crucial step in the compression process. The problem is that optimising low-precision parameters directly is relatively unstable [90], which only exacerbates the optimisation instability that is already present in DRL [151]. PTQ (Post-Training Quantization) can be used to convert a full-precision model to a quantized version after training, avoiding the unstable optimisation process, but this introduces inaccuracies for each operation that accumulate when propagating through the network [39]. A way to combine

training a DRL model with QAT where the quantization effects are taken into account during training [39] would result in the most effective low-precision model, but finding the best approach for this is an unsolved problem, especially when this should be combined with other compression methods to obtain models that are sufficiently small for deployment in terms of the number of parameters as well as their precision. To balance the trade-off between the reduced representational power from lower precision and from reduced capacity, the relation between these two factors should be investigated.

4. **Is the stochasticity of a policy accurately preserved by compression?** Especially in the context of continuous control tasks and algorithms such as SAC that optimise for an entropy-regularised return [47], the stochasticity of the policy is essential for maintaining the agent's behaviour. However, as noted by Stanton et al. [129] in an SL setting, KD does not typically work as commonly understood where the student learns to exactly match the teacher's behaviour, even if the student has the same capacity as the teacher and therefore should be able to match it precisely. It therefore remains to be seen to what extent the stochasticity of a policy can be preserved during distillation, whether this is necessary for the student to achieve the same level of performance, and if there is a direct relation between the accuracy of maintaining the stochasticity and the robustness of the compressed policy. This also relates to the exploration-exploitation trade-off, where a stochastic policy will likely perform more exploration [47], even after training. During distillation, the student should be able to learn from a wide distribution of states and actions to generalise well to unseen scenarios. Following a stochastic policy while gathering training data exposes the student to a more diverse set of states and actions, but it might also require more capacity to learn from than a deterministic policy. The impact of the degree of teacher stochasticity on the distillation process should therefore also be studied. Finally, given that stochastic behaviour can be essential for the agent's task performance, it is important to investigate how well these action probabilities can be represented in low-precision.

5. **How does pruning affect a model's ability to be fine-tuned and adapt to new tasks?** In contrast to distillation, where the network parameters and output values are transformed as part of the compression process [52], pruning aims to maintain the original internal representation of the model and only remove redundant parts without significantly affecting the behaviour [79]. This opens up the possibility of continuing to optimise the model using the original training method after compression. In fact, this is often done as part of an iterative process to recover accuracy after each pruning step, using the same training data [37]. Whether the model retains the flexibility to adapt to new tasks by fine-tuning after compression, and how to improve its generalisation to support this, remains an open question. This question is especially relevant when deploying at the edge where the models need to adapt to local conditions or continuously changing tasks on devices that lack the resources for training full-sized networks.

6. **Can we improve efficiency by reducing the number of active decisions required to complete a task?** Traditional model compression methods have focused on improving the efficiency of each decision made by the model, by reducing the number of parameters [115, 84] or the precision of the computations [74]. This makes sense in a classification setting, where each task only requires a single decision. But in RL, the agent must make a sequence of decisions to complete a task [131], with

each decision requiring additional time and energy. If an agent could learn to predict several actions with a single decision, it would reduce the overall energy consumption and latency of the model. But this would also require a more complex policy and architecture, which could be detrimental to the compression potential. On the other hand, many tasks only require the agent to change its action infrequently. In these cases, it would be sufficient to learn when a new action should be required.

1.4 Research Contributions

To address the research questions outlined above, the following contributions were made, with an overview depicted in Figure 1.4:

1. **A distillation loss optimised for actor-critic networks** *(Chapter 3)*.
    - While existing work on PD for actor-critic architectures has focused on distilling the actor head, we introduce a novel distillation loss that leverages both the actor and critic heads to improve the student's performance.
    - By visually representing the activations of the second-to-last layer of the student model, we show how the internal representation is improved by learning from the auxiliary critic loss.
    - Based on an investigation into how the distillation loss is affected by the output distributions in policy gradient networks compared to value-based teachers, we introduce actor smoothing to ensure more secondary knowledge is transferred.
    - Finally, we demonstrate how the regularisation effect from PD can recover the performance of models after they had started to regress due to overtraining.
2. **The QPD (Quantization-Aware Policy Distillation) algorithm for training tiny low-precision students based on full-precision teacher models** *(Chapter 4)*.
    - By combining quantization with PD, we provide a more stable optimization objective than traditional DRL algorithms, leading to state-of-the-art performance in low-precision DRL.
    - We present additional improvements to the stability of parameter updates by computing the gradient in full-precision based on low-precision activations, to gain the benefits of accurate optimization that still can account for the errors introduced by the quantization function.
    - A smoother transition from high to low-precision is provided to overcome the remaining optimization instabilities from training directly in low-precision, by leveraging both PTQ and QAT in a three-phase algorithm.
3. **A set of distillation losses designed to enable the compression of DRL models that operate over continuous action spaces** *(Chapter 5)*.
    - We designed three loss functions for distillation with an emphasis on preserving the stochastic nature of the original policies.
    - The effectiveness of these methods was shown to be highly correlated with accuracy in maintaining the action distribution entropy.

- We present an analysis of the benefits of using a stochastic student-driven control policy while gathering training data in PD.
- We explore the feasibility of deriving a more informative intrinsic reward signal from the teacher's actor and critic outputs to guide the student's learning process.

4. **The introduction of temporal distillation: an algorithm that combines distillation with a new form of compression in the temporal domain** *(Chapter 6)*.

   - An extension to the traditional RL loop is presented that supports performing an action over multiple time steps.
   - We define a new distillation loss with an additional term for learning a repeat value.
   - The replay memory of PD is extended to compute and store a safe empirical lower-bound on the number of consecutive repeats performed by the teacher.
   - An analysis is performed to compare two variants of how this value can be computed for a given trajectory.
   - By evaluating this method for a wide range of student sizes and devices, we show how the most efficient model size is a trade-off that depends on the deployment hardware.

5. **A training and compression procedure developed to enhance the adaptability of pruned networks** *(Chapter 7)*.

   - We compare how well pruned models can still adapt to (1) a distribution shift in the data with new subclasses and (2) continual learning for entirely new classes.
   - Propose a new method that increases generalisation by pre-training on a larger dataset with different classes or more diverse samples than useful for the core task and prune redundant features by leveraging LRP (Layer-wise Relevance Propagation) to identify task-relevant features.

6. **An application-focused set of guidelines for applying policy compression on real-world use cases** *(Chapter 8)*.

   - We present a four-phase training strategy for training a low-power NF (Network Function) replica scaler.
   - An iterative workflow for finding and training the optimal student architecture was outlined.
   - Several modifications were introduced to the SDN (Software-Defined Networking) simulator environment to make it more compatible with the distillation process.
   - An extension to the PD algorithm was proposed to allow for the training of RNN-based student architectures.
   - The importance of learning to maintain an informative recurrent hidden state as part of the distillation process to solve POMDPs was highlighted on a photo-realistic embodied point-goal navigation task.

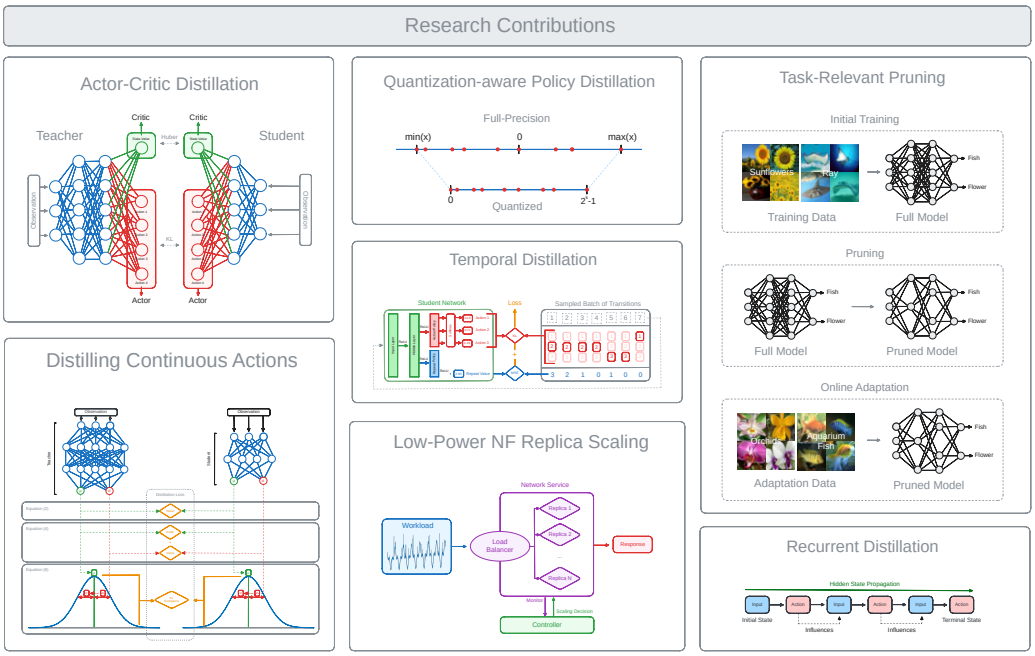


Figure 1.4: An overview of the research contributions.

1.5 List of Publications

The work presented in this thesis has resulted in the following peer-reviewed publications:

1. **Thomas Avé**, Kevin Mets, Tom De Schepper, and Steven Latré. *"Quantization-aware Policy Distillation (QPD)."* In Deep Reinforcement Learning Workshop, NeurIPS 2022, 9 December, 2022, pp. 1-15. 2022.
2. **Thomas Avé**, Paola Soto, Miguel Camelo, Tom De Schepper, and Kevin Mets. *"Policy Compression for Low-Power Intelligent Scaling in Software-Based Network Architectures."* In NOMS 2024-2024 IEEE Network Operations and Management Symposium, pp. 1-7. IEEE, 2024.
3. **Thomas Avé**, Tom De Schepper, and Kevin Mets. *"Policy Compression for Intelligent Continuous Control on Low-Power Edge Devices."* Sensors 24, no. 15 (2024): 4876.
4. **Thomas Avé**, Matthias Hutsebaut-Buysse, Wei Wei and Kevin Mets. *"Online Adaptation of Compressed Models by Pre-Training and Task-Relevant Pruning"* In The 32nd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN), Bruges, Belgium, 9-11 October 2024.

Additionally, the following publication is under review at the time of writing:

5. **Thomas Avé**, Matthias Hutsebaut-Buysse, and Kevin Mets. *"Temporal Distillation: Compressing a Policy in Space and Time."* Submitted to Springer Machine Learning.

And the following contributions were made outside the scope of this thesis:

6. Wei Wei, Matthias Hutsebaut-Buysse, **Thomas Avé**, Tom De Schepper and Kevin Mets. *"Feature-level fusion in wireless acoustic sensor networks with graph attention network for classification of domestic activities"* In Workshop on Data Fusion for Artificial Intelligence (DAFUSAI) at 27th European Conference on Artificial Intelligence (ECAI), Santiago de Compostela, Spain, 19-24 October 2024.
7. Wei Wei, Matthias Hutsebaut-Buysse, **Thomas Avé**, Tom De Schepper and Kevin Mets. *"Flexible and Efficient Feature-level Fusion with Wireless Acoustic Sensors using Graph Attention Networks"* Submitted to IEEE Sensors Journal.

1.6 Reproducibility

A generic software library[2] was created based on the contributions in Chapters 3, 4 and 5, to reproduce our original findings or apply the proposed compression methods for custom models. The code for the remaining contributions will be made available after publication, or upon request.

1.7 Thesis Outline

In the following chapters of this thesis, we first provide an overview of the necessary background information in Chapter 2, including the fundamentals of DL, RL, and DRL, as well as the different model compression methods we built upon. The research contributions are then presented in Chapters 3 to 8, as outlined in Table 1.1. This table provides an overview of which research questions are addressed by which contributions, how these utilise the research opportunities identified in Section 1.2, in which publication they are included, and where to find them in this thesis.

Table 1.1: An overview of the relation between research opportunities, questions, contributions, publications and chapters.

Chapter	**3**	**4**	**5**	**6**	**7**	**8**
Contribution	1	2	3	4	5	6
Question	1, 2, 4	2, 3, 4	1, 2, 4	1, 6	5	1, 2
Opportunity	1, 2, 3	2	3, 4, 5	6	/	6
Publication	1	1	3	5	4	2

[2] `https://links.thomasave.be/research`

We begin in Chapter 3 by introducing new distillation methods specifically designed for actor-critic networks, and compress these even further through quantization in Chapter 4 by proposing an algorithm to train low-precision agents. The work presented in Chapter 5 then addresses the challenges of distilling policies for continuous action spaces. In Chapter 6, we introduce a new form of compression in the temporal domain, achieving even greater efficiency than possible through traditional "spatial" methods.

By focusing on pruning in Chapter 7, we cover the full range of common model compression techniques. Here, we investigate how compressed models can adapt to new data on the edge. Note that this contribution was developed in a purely SL context, so it does not directly relate to any of the opportunities we identified for DRL-specific compression, but it does provide the preliminary insights required for bringing a similar pruning approach to DRL, as discussed in the future work (Section 9.2).

Finally, in Chapter 8, we explain how to effectively apply policy compression in practice. This includes a compression strategy for a low-power NF replica scaler, and how to distil RNN-based architectures for embodied point-goal navigation tasks. We conclude this thesis in Chapter 9 by summarising the main findings and revisiting the research questions posed in the introduction. As part of this, we also highlight several remaining open challenges and opportunities for future research in the field of policy compression.

Chapter 2

Background

The goal of this thesis is to study novel ways in which the DNNs used as part of DRL policies can be compressed. In order to do that, we first need to provide the necessary background for each component in this chapter.

Deep Learning (Section 2.1): We start by giving a quick introduction to the field of DL, including how DNNs work and how they can be trained.

Reinforcement Learning: (Section 2.2): Next, we will discuss the concept of RL, how the tasks it is designed to solve can be defined through MDPs, and some of the most common algorithms for training DRL policies.

Model Compression (Section 2.3): Finally, we explain the three most common methods used to compress DNNs: KD (Knowledge Distillation), quantization, and pruning.

2.1 Deep Learning

DL, a subfield of ML, which itself is a subfield of AI (Artificial Intelligence) (see Figure 2.1), revolves around the use of DNNs, capable of learning complex patterns and representations from large amounts of data, often outperforming traditional ML techniques in various tasks such as image recognition [75] and natural language processing [30]. As an ML technique, DL models are trained to make predictions based on input data by learning from examples instead of being explicitly programmed.

2.1.1 Network Architecture

These models are loosely inspired by the structure of the human brain [111] and consist of multiple layers (hence "deep") of interconnected neurons that process and transform input data into a desired output [1].

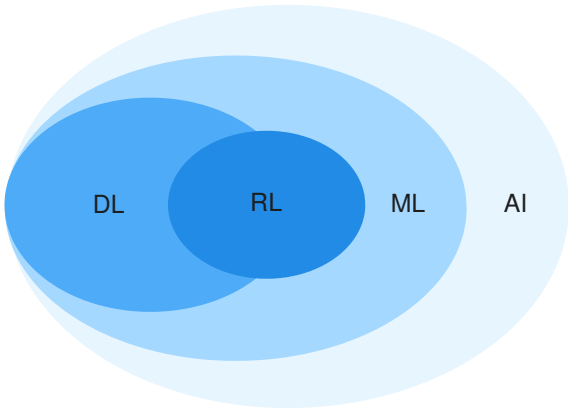


Figure 2.1: An overview of the AI hierarchy, including ML, RL and DL.

The most common type of DNN is an MLP (Multi-Layer Perceptron): a feedforward network where the data flows in one direction, from the input layer, through one or more hidden layers, to the output layer:

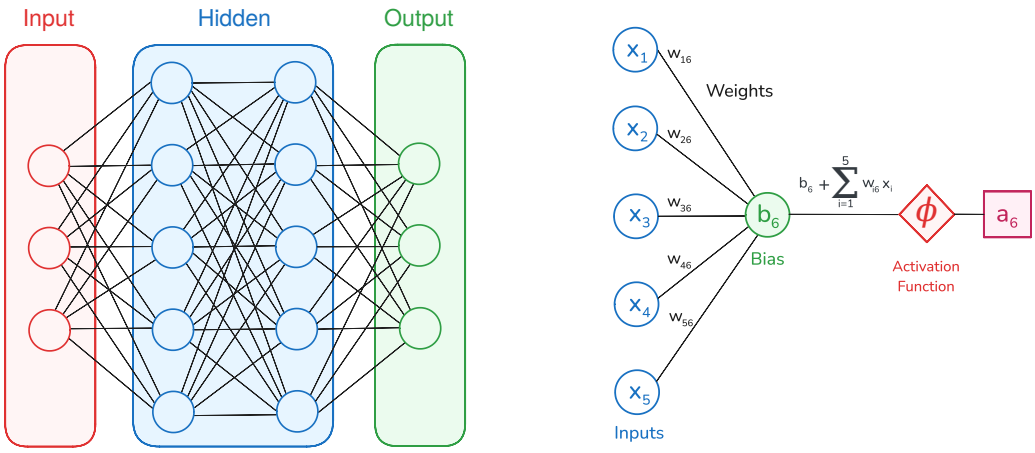
- **Input Layer:** The first layer of the network, which receives the input data, with each neuron corresponding to a feature of the input.
- **Hidden Layers:** Intermediate layers that process the input data through a series of transformations. The presence of these layers (in addition to their activation functions) allows for the capture of non-linear relationships in the data. A NN is considered "deep" when it has more than one hidden layer.
- **Output Layer:** The final layer of the network, which produces the output of the model. In classification tasks, these layer outputs are usually a probability vector over the possible classes, but this can also be a continuous value in regression tasks. DRL policies typically have at least one output neuron for each possible action, but the type as well as the number of outputs vary significantly between different algorithms.

An illustration of these different types of layers is given in Figure 2.2a.

Each neuron (also known as a perceptron, depicted as a circle in Figure 2.2a) in this interconnected network consists of a simple computation in the form of a weighted sum of its inputs with an added bias term, followed by a nonlinear activation function [111]. Formally, the output (or activation) of a given neuron j is defined as:

$$a_j = \phi(\sum_i w_{ij} x_i + b_j) \tag{2.1}$$

Here are w_{ij} the weights connecting the inputs x_i to neuron j, b_j the bias term, and ϕ the activation function. Common activation functions include the sigmoid, tanh, and ReLU (Rectified Linear Unit) functions [40], which introduce non-linearity and enables



(a) DNN Layers **(b)** Perceptron

Figure 2.2: An illustration of DNN components.

the network to learn more complex patterns. The activations of the neurons in one layer are then used as inputs for the neurons in the next layer, and this computational process (referred to as the forward pass) is repeated until the output layer is reached [40]. This computation and dependence on the previous layer is illustrated in Figure 2.2b. In practice, the activations for all neurons in a layer are computed in parallel using matrix operations, which can be efficiently implemented on modern hardware, such as GPUs or NPUs. More complex architectures than the MLP, such as CNNs (Convolutional Neural Networks) [80] or RNNs [113], can be constructed by adding additional layers or by modifying the connections between neurons. These advanced structures are particularly well-suited for tasks such as image recognition or processing of sequential data, respectively.

2.1.2 Training

In essence, DNNs can be seen as complex, high-dimensional functions that map input data to the desired output, with the weights and biases of the neurons acting as the parameters of this function [40]. These parameters are initialised randomly and are trained by adjusting them to minimise the difference between the network output and the expected output, using a loss function that quantifies this difference [99]. Common loss functions include the Huber loss [55] or the MSE (Mean Squared Error) for regression tasks and the cross-entropy loss for classification tasks.

To optimise the parameters of the network, the gradient of the loss function is computed using the backpropagation algorithm [113]. This is done in a backward pass through the network, by applying the chain rule of calculus to compute the gradient of the loss with respect to each parameter. Backpropagation is an efficient way to compute these gradients, as it reuses intermediate computations based on dynamic programming principles.

The parameters are then updated using an optimisation algorithm such gradient descent, which iteratively adjusts the parameters in the direction that minimises the loss:

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial L}{\partial w_{ij}} \tag{2.2}$$

Here is η the LR (Learning Rate), which controls the step size during each update, and $\frac{\partial L}{\partial w_{ij}}$ is the gradient of the loss with respect to weight w_{ij}. The bias b_j is updated analogously. Choosing an appropriate LR is crucial for training a DNN, as a value that is too high can lead to oscillations or divergence, while a too low value can result in slow convergence. Up until now, we made the assumption that the loss is computed over the entire training data at once, but this is usually not feasible in practice, so the dataset is divided into smaller batches and the parameters are updated after each batch. Additionally, the optimisation algorithm can be further improved by using techniques such as momentum, which helps to accelerate convergence by adding a fraction of the previous update to the current one, or by using adaptive LRs, which adjust the LR for each parameter based on its past gradients. This is done by algorithms such as SGD (Stochastic Gradient Descent) [113], Adam (Adaptive Moment Estimation) [70], or RMSprop (Root Mean Square Propagation) [51].

2.2 Reinforcement Learning

RL is an ML technique in which an agent learns a behavioural strategy through trial-and-error by interacting with an environment [131]. Unlike SL, where models learn from labelled examples, RL agents sequentially choose actions based on an observation of the current state of the environment. They learn by receiving feedback in the form of numerical rewards, which are used to assign credit to previous actions and refine future decision-making [131]. Rewards are often sparse or delayed, meaning that the agent needs to learn to associate actions with their long-term consequences, rather than just the immediate feedback. By taking this action, the state of the environment is updated to reflect the consequences. This iterative process is shown in Figure 2.3. Here, the agent is represented with a DNN to facilitate decision-making, as we focus on DRL models in this thesis, but that is not exclusively the case for RL in general.

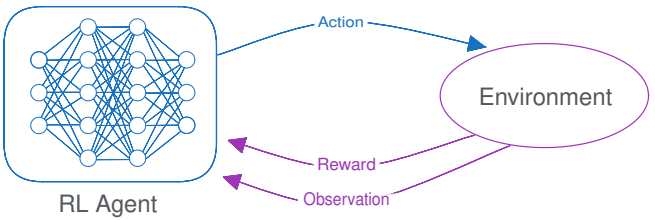


Figure 2.3: An illustration of the RL loop.

The learned mapping from states to actions is called the policy (π), and the goal of the agent is to learn a policy that maximises the (discounted) cumulative reward over an episode, called the return [131]. An episode is a single attempt by the agent to solve the task, starting at the initial state and ending at a terminal state. A sequence of states (s), actions (a), and rewards (r) encountered in an episode is called a trajectory (t). This policy is optimised using the reward signal to encourage or discourage certain behaviour. In the remainder of this section, we will provide a more formal definition of RL and these concepts, and discuss the most common approaches used to train DRL policies.

2.2.1 Markov Decision Processes

2.2.1.1 Definition

The environment an agent interacts with is typically modelled as an MDP, which is defined as a 4-tuple (S, A, P, R_a) [131, 105, 76], where:

- S is the set of possible states of the environment, called the state space. The state space can be either discrete or continuous, depending on the task. A state encompasses all the information necessary to perfectly describe the environment at a particular point in time.
- A is the set of possible actions that can be taken, called the action space, which can also be either discrete or continuous. Taking an action influences the transitions between states of the environment and the reward received by the agent.
- $P(s'|s, a) \in [0, 1]$ is the transition probability function that taking action a in state s will lead to state s'.
- $R_a(s, s') \in \mathbb{R}$ is the immediate reward received after transitioning from state s to state s' after taking action a.

State transitions follow the Markov property, meaning that future states only depend on the current state and action, and not on the history of previous states and actions:

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, a_0, s_0) \tag{2.3}$$

POMDPs extend this definition to partially observable environments, where the agent does not have full access to the state of the environment. Although the states still follow the Markov property, the agent receives observations that are only a partial or noisy representation of the true state and might require the agent to maintain an internal state to model an optimal policy [63]. This can also lead to state aliasing, where two identical observations represent different underlying states, requiring tailored solutions for the agent to handle effectively, such as including a form of memory or working with stochastic policies. We therefore refer to an observation given to the RL agent instead of the state, in Figure 2.3 and throughout this thesis, as this is agnostic to whether the environment is fully observable or not.

2.2.1.2 Objective

A policy $\pi(a|s) \rightarrow [0,1]$ is defined as the probability of taking action a in state s, and models the behaviour of the agent when interacting with the MDP. When the policy is deterministic, this is often denoted as $\pi(s) = a$. The objective in an MDP is to find an optimal policy π^* that maximises the expected discounted return G_t [131], which is the sum of the rewards received over an episode, weighted by a discount factor $\gamma \in [0,1]$ to encourage an agent to favour taking rewarding decisions early rather than postponing it:

$$G_t = \sum_{t=0}^{T} \gamma^t R_{a_t}(s_t, s_{t+1}) \tag{2.4}$$

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}[G_t] \tag{2.5}$$

In finite MDPs, the episode ends after a finite number of steps, called the horizon T. This can be either a fixed number of steps, or based on a terminal state that the agent reaches. Non-episodic tasks have an infinite horizon ($T = \infty$), and the discount factor ($\gamma < 1$) can be used to ensure that the return remains finite. There, the agent aims to maximise the expected return at each time step, rather than over the entire episode [86].

To evaluate the quality of a policy, the value function $V^\pi(s)$ can be used, which is defined as the expected return when starting in state s and following policy π:

$$V^\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{a_{t+k+1}}(s_{t+k}, s_{t+k+1}) \,\middle|\, s_t = s\right] \tag{2.6}$$

Intuitively, if $V^\pi(s_0) > V^{\pi'}(s_0)$, then π is a better solution than π', as it results in a higher expected return when starting in state s_0. It also provides an indication of how valuable it is to be in a particular state, which can be used to guide the agent towards states that are more likely to lead to a high return. Multiple policies can be optimal through different behaviour, but they share the same optimal value function $V^*(s)$.

Similarly, the state-action value function $Q^\pi(s,a)$ is defined as the expected return when starting in state s, taking action a, and following policy π afterwards:

$$Q^\pi(s,a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] \tag{2.7}$$

This expresses the value of taking a particular action in a given state, and can be used to evaluate the quality of different actions.

2.2.2 Dynamic Programming

When the state and action spaces are discrete, and the transition and reward functions are known, the optimal policy can be found using dynamic programming methods, such as value iteration or policy iteration. These methods rely on the Bellman equations [14],

which express the relationship between the value of a state and the values of its successor states. The Bellman equations can be written for the action-value and value functions, respectively, as:

$$Q^{\pi}(s,a) = \sum_{s' \in S} P(s'|s,a) \left[R_a(s,s') + \gamma V^{\pi}(s') \right] \tag{2.8}$$

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s)\, Q^{\pi}(s,a) \tag{2.9}$$

These equations state that the value of taking action a in state s is equal to the immediate reward received, combined with the discounted value of the expected next state. This allows us to break down the problem of finding the optimal policy into smaller sub-problems through a recursive decomposition of the optimisation problem, which can be solved iteratively.

Policy iteration is such a dynamic programming algorithm that alternates between policy evaluation and policy improvement steps to find the optimal policy [131]:

1. *Policy Evaluation:* Given a policy π, we compute its value function $V^{\pi}(s)$ for all states $s \in S$. This is done by iteratively applying the Bellman expectation equation until the change between the old and new value function becomes smaller than a chosen integer for every state:

$$V^{\pi}(s) \leftarrow \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P(s'|s,a)[R_a(s,s') + \gamma V^{\pi}(s')] \tag{2.10}$$

2. *Policy Improvement:* We improve the policy by choosing actions that maximise the expected return based on the current value function:

$$\pi'(s) = \arg\max_{a} \sum_{s' \in S} P(s'|s,a)[R_a(s,s') + \gamma V^{\pi}(s')] \tag{2.11}$$

These steps are repeated until the policy no longer changes, at which point we have found the optimal policy π^*.

Value iteration is another iterative algorithm that directly updates the value function estimate without explicitly maintaining a policy [131]. It is based on the Bellman optimality equation, which states that the optimal value of a state is the maximum expected return achievable by taking any action in that state [14]. Value iteration uses this equation to iteratively update the value function estimate for every state $s \in S$:

$$V_{k+1}(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a) \left[R_a(s,s') + \gamma V_k(s') \right] \tag{2.12}$$

Here is $V_k(s)$ the estimate of the optimal value function at iteration k.

Once the optimal value function has been found, the optimal policy can be obtained by acting greedily with respect to it:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s'|s,a) \left[R_a(s,s') + \gamma V^*(s') \right] \tag{2.13}$$

Both policy iteration and value iteration are guaranteed to converge to the optimal policy in a finite number of iterations for finite MDPs [131].

2.2.3 Learning through Interaction

While dynamic programming methods offer a way to find the optimal policy for an MDP when the transition and reward functions are known, this is often not the case in real-world problems. RL addresses this by allowing an agent to learn through direct interaction with the environment, without requiring a model of the environment [131]. Instead of computing the exact value for every state, the agent learns by trial-and-error, and updates its policy based on the rewards it receives. This makes RL applicable to a wider range of problems, including where the transition and reward functions are unknown or difficult to model explicitly, or where the state space is too large to be enumerated at each iteration. RL algorithms can theoretically even be used for state spaces that are infinite.

Not needing to visit every state-action pair to learn a policy is a significant advantage of RL over dynamic programming methods, but it also introduces new challenges. The first challenge is the exploration-exploitation trade-off, where the agent needs to balance between trying out new actions to discover potentially better strategies (exploration) and refining its current knowledge to maximise its rewards (exploitation). Balancing these two competing objectives is crucial for the agent to learn an optimal policy.

One common strategy to manage this trade-off is the ϵ-greedy policy [142], where the agent chooses a random action with probability ϵ (exploration) and the action that maximises the estimated return with probability $1 - \epsilon$ (exploitation). As the agent learns more about the environment, ϵ is typically decayed, favouring exploitation of the learned policy as its estimates become more accurate. Alternatively, some algorithms model the policy as a probability distribution over actions, and sample actions directly from this distribution, which naturally balances exploration and exploitation as the policy improves [145].

Another challenge in RL is the credit assignment problem, which refers to the difficulty of determining which actions were responsible for observed rewards. Rewards may be delayed or sparse, making the consequences of an action not immediately apparent. Sometimes the agent needs to take actions that result in fewer short-term rewards to achieve a higher long-term return, with those actions being crucial for the final outcome. Several mechanisms have been developed to address this problem, such as TD (Temporal Difference) learning and eligibility traces, which bootstrap the value function estimates based on the differences between consecutive predictions.

2.2.4 Types of Algorithms

The different approaches that can be taken to learn a policy in RL can broadly be classified in the following categories:

1. One distinction is between **on-policy** and **off-policy** methods. On-policy methods learn the value of the policy that is currently being used to select actions and gather training data. An example of this is SARSA (State-Action-Reward-State-Action) [114]. Off-policy methods, on the other hand, can learn from data generated from a different policy. This means that they can learn from older experiences or from a different agent, and reuse this data to learn about an improved policy. This has the advantage that they can be more flexible and sample efficient, as they can learn from a larger amount of data. A well-known example of an off-policy algorithm is Q-learning [142], which we discuss in the next section (2.2.5). SARSA can take its exploration strategy into account when learning and can be more stable by being more cautious and avoiding risky situations. Q-learning is focused on obtaining the highest possible return, even if this strategy has more potential for catastrophic failure.

2. Both Q-learning [142] and SARSA [114] are examples of **model-free** methods, which directly learn a policy or value function without explicitly modelling the environment dynamics. These algorithms rely on trial-and-error learning and update their estimates based on the observed rewards and state transitions. **Model-based** methods, on the other hand, learn a model of the environment, such as the transition probability function and the reward function [131]. This model can then be used to plan and predict future outcomes, and to learn an optimal policy using methods such as dynamic programming. While model-based methods can be more sample efficient than model-free methods, they require a good model of the environment, which can be difficult to obtain in practice. Model-free methods can therefore be more flexible and be less computationally expensive, which is why we focus on these types of methods throughout this thesis.

3. Finally, similar to the dynamic programming methods we discussed earlier, RL methods can either learn a policy explicitly and are therefore classified as **policy-based**, or they can derive a policy implicitly from a value function in a **value-based** approach, such as Q-learning [142]. Policy-based methods, such as REINFORCE [145], typically represent the policy as a parameterized function and update the parameters of this function to maximise the expected return. This is usually done using a DNN, which we will discuss in more detail in Section 2.2.6. A subcategory of policy-based methods are **actor-critic** algorithms, which explicitly learn a policy (the actor) but use a value-function (the critic) to provide a direction in which to update the actor to increase the likelihood of taking actions that lead to higher returns. Examples of this are PPO [122] and A2C (Advantage Actor-Critic) [92].

2.2.5 Q-Learning

Q-learning is a model-free, off-policy, and value-based RL algorithm that learns an optimal policy by estimating the action-value function, $Q(s, a)$ [142]. It is the most well-known traditional RL algorithm, meaning that it does not rely on DNN and function approximation to model the policy. Although we focus exclusively on DRL models in this thesis, it is still useful to first get an intuition on Q-learning before transitioning to the DRL equivalent Deep Q-Learning or DQN [94].

The core idea behind Q-learning is to iteratively update estimates of the Q-values for each state-action pair using the Bellman optimality equation, and store these in the Q-table. This method is therefore often referred to as *tabular* Q-learning, in contrast to the analogous *deep* Q-learning. The Q-table is typically initialised with arbitrary values, usually all zeros. As the agent interacts with the environment, it updates the Q-values based on its experiences, using the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_a(s_t, s_{t+1}) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.14}$$

Here, s_t and a_t are the current state and action respectively, s_{t+1} the next state, and $\alpha \in (0, 1]$ the LR, which determines the step size of the update. The term in square brackets is often referred to as the TD-error, as it represents the difference between the current estimate and the target value based on the observed reward and the estimated value of the next state, weighted by the discount factor γ. It essentially updates the Q-value of the current state-action pair in the direction of the computed target value. Q-learning is guaranteed to converge to the optimal action-value function $Q^*(s, a)$ under certain conditions, provided the agent explores all state-action pairs sufficiently (using the ϵ-greedy strategy), the LR α decays appropriately, and the environment is a finite MDP. The policy is then derived by selecting the action that maximises $Q^*(s, a)$ for each state s:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s, a) \tag{2.15}$$

This works well for small state and action spaces, but quickly becomes infeasible for larger or continuous spaces, as the Q-table needs to store and update a value for every possible state-action pair. When deployed, it requires the agent to have already visited the exact state-action pair during training, which is often not practical in real-world problems. To address this, function approximation techniques such DNNs can be used instead, which we will cover in the next section (2.2.6).

2.2.6 Deep Reinforcement Learning

DRL combines the power of DL with the sequential decision-making capabilities of RL. By leveraging DNNs as function approximators, DRL can tackle complex tasks with high-dimensional state and action spaces that were previously intractable for traditional RL algorithms, but are often encountered in real-world problems [94]. By using DNNs to approximate value functions or policies, DRL agents can handle complex tasks, such

as playing video games [94], controlling robots and drones [96], or optimising resource allocation [133]. This is due to the ability for DRL models to learn how to extract and process relevant features from raw data, such as images or sensor readings, without requiring significant domain expertise or feature engineering. It also allows these models to generalise to states that were not encountered during training, but contain similar features that can be recognised to still make relevant decisions.

2.2.6.1 Network Outputs

In the introduction (Section 1.2), we highlighted several properties unique to DRL that could be exploited in more advanced compression methods, many of which are related to the network architecture and the type of output values. We will discuss these properties in more detail in this section, starting with the potential action spaces. DRL policies can be considered end-to-end, as they directly feed observations received from the environment as input to their DNN and output a representation of the action to be taken. In contrast to tabular methods such as Q-learning [142], these actions can be both continuous and high-dimensional, in addition to discrete actions. An illustration of the difference in network architecture between both types of (stochastic) action spaces is given in Figure 2.4.

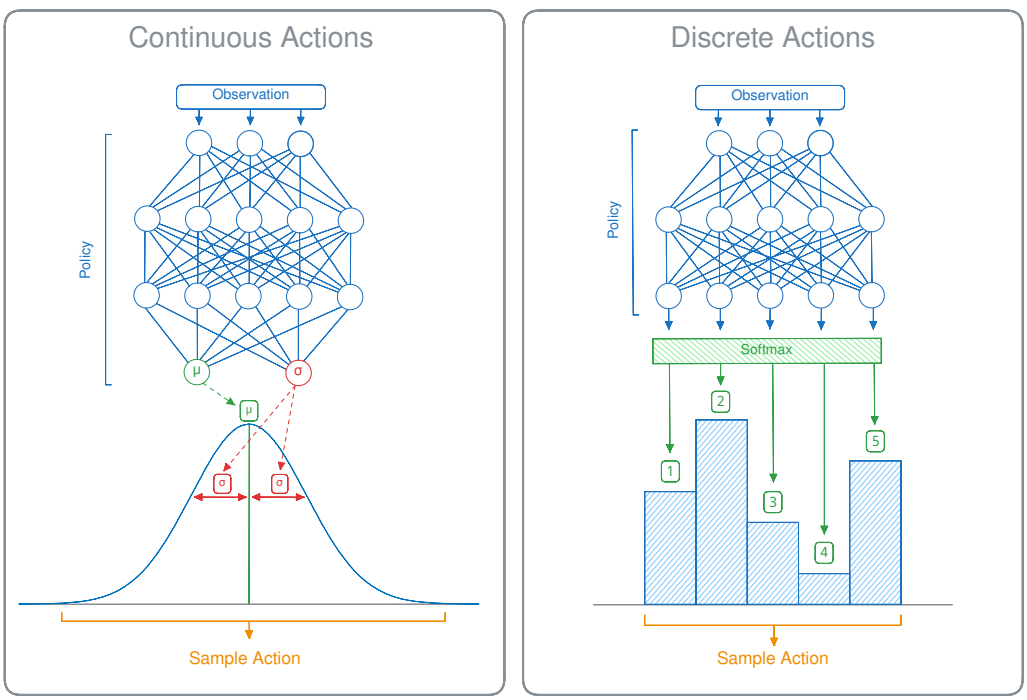


Figure 2.4: Discrete and continuous action spaces for stochastic DRL policies.

Discrete Action Spaces For discrete tasks, the agent can choose from a limited set of options, such as a cardinal direction to move in. Here, the DNN typically outputs a single value for each possible action. In Figure 2.4, we show the output of the network as a probability distribution over the possible actions, which is common for policy-based methods, such as REINFORCE [145], which model the policy $\pi(a|s)$ directly as a mapping from states to action probabilities. Actions are selected according to this policy by sampling from the predicted distribution, which is optimised to increase the probability of actions that lead to higher returns.

In contrast, value-based methods such as DQN [94] are deterministic (omitting the softmax layer in Figure 2.4), with the policy derived by selecting the action with the highest output value. These network outputs correspond to the estimated state-action value $Q^{\pi}(s, a)$, as explained in Section 2.2.1.2, that represent the expected discounted return when taking this action and following the same policy afterwards.

Continuous Action Spaces For continuous tasks, the agent can perform actions using a combination of real-valued numbers, such as the distance to move or how much torque to apply. DRL algorithms that support continuous action spaces, such as PPO [122], A2C [92], SAC [47], or TD3 (Twin Delayed Deep Deterministic Policy Gradient) [38], work by modelling the policy as a continuous probability distribution from which actions are sampled.

In practice, this almost always takes the form of a normal distribution, as shown in Figure 2.4, so this will be our focus in this thesis. The model then predicts a mean value (μ) for each action, and the actual policy consists of sampling actions based on this mean and a standard deviation (σ). This standard deviation can in effect be used to control the trade-off between exploitation and exploration, using a lower or higher value of σ, respectively.

There are several methods for modelling σ, either algorithmically or learned by the model. Depending on the implementation, either a representation is learned that is dependent on the current state of the environment or one that simply consists of a state-independent vector. In the state-dependent setting, the model can learn to increase or decrease exploration for certain parts of the environment, depending on its degree of uncertainty.

For most algorithms and environments, the learned σ should generally gradually decrease during training as the certainty about the environment increases. Often, the deterministic policy that consists of always choosing the predicted mean action (with $\sigma = 0$) will produce the best results during evaluation [47], but this is not always the case.

For environments that are either only partially observable (POMDPs), are non-deterministic, or contain state aliasing, a stochastic policy can be necessary. Since the policy is trained with this stochasticity in place, it can be detrimental to remove it, as the policy has learned to rely on it. This is especially true for SAC [47] agents, which are trained to maximise an entropy-regularised return and, therefore, to obtain the highest possible return while also remaining as stochastic as possible.

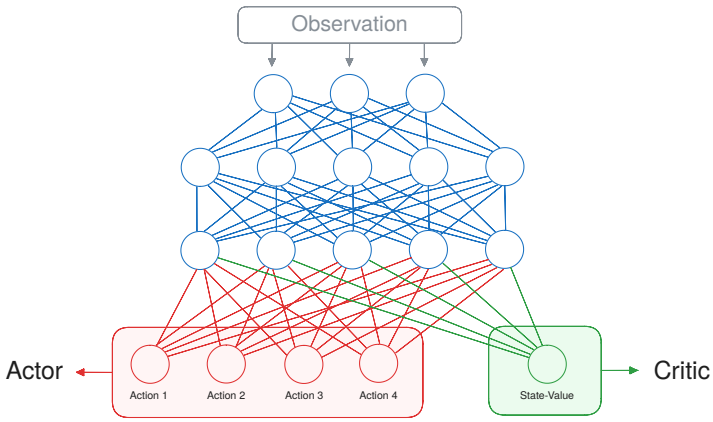


Figure 2.5: An illustration of an actor-critic architecture for discrete action spaces with shared layers.

Actor-Critic Architectures Most policy-based methods mentioned previously, such as PPO [122], A2C [92] and SAC [47], can be further classified as actor-critic algorithms. In addition to learning a policy (the actor) in the form of a probability vector with an entry for each action, these models also output a value for estimating the state-value function (the critic). Sometimes this critic is learned as an entirely separate network, but often it shares many initial layers with the actor, as this has the potential to improve sample efficiency and the shared internal representation.

An illustration of this actor-critic architecture with common layers is given in Figure 2.5. This critic head is used during training to provide a direction in which to update the actor, by estimating the value of the state the agent is in. During evaluation, however, the critic is not used and can be discarded from the computation graph.

Other Network Outputs This summarises the network outputs of the most common types of DRL algorithms, which we will use throughout this thesis. Other algorithms also exist with more complex network architectures, such as QR-DQN (Quantile Regression Deep Q-Network) [27], which models the Q-values as a distribution over possible returns using one output for each (quantile, action) pair, to capture the inherent uncertainty in value estimation. Similarly, in D4PG (Distributed Distributional Deterministic Policy Gradients) [12], the critic outputs a distribution over the Q-values for each action, by learning a probability vector over a discretised range of Q-values. Policies can also use RNN-based architectures, in which case the hidden state can be viewed as another type of network output.

2.2.6.2 Deep Q-Networks

We now discuss several DRL algorithms in more detail, starting with DQNs (Deep Q-Networks). Although our contributions do not improve upon these methods directly, it is essential to have a deeper understanding of their unique properties to inform the design of compression methods tailored their characteristics. These algorithms are also used to

train the agents that we will compress, so understanding the differences in their inner workings is crucial when comparing the compression potential of different model types. We therefore focus on their core principles, leaving out some implementation details for the sake of brevity.

Deep Q-learning [94] addresses the limitations of tabular Q-learning by leveraging DNNs to approximate the action-value function, $Q(s, a)$. This replaces the Q-table, as such a tabular approach becomes impractical when dealing with large or continuous state spaces, requiring an intractable number of resources to maintain the Q-values for every possible state-action pair. Instead, a DQN is used to approximate the Q-values, with the observation as input and the Q-values for each action as output of the DNN.

Loss Function The DQN is trained using a loss function that is based on the same principle of updating the values using the Bellman equation, to minimise the difference between the predicted Q-value for a given state-action pair and the target Q-value, which is computed using the observed reward and the maximum estimated Q-value for the next state:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[(r + \gamma \max_{a'} Q^{\theta^-}(s', a') - Q^{\theta}(s, a))^2\right] \tag{2.16}$$

Here, θ represents the parameters of the Q-network, and θ^- those of a target network: a second DNN that is used to stabilise training by holding its weights fixed for several iterations before being updated. The target network is updated periodically to match the Q-network, which helps to mitigate the instability and divergence issues that can arise when training a Q-network directly. Otherwise, this target value would be constantly changing at each update step, leading to a moving target that can make training difficult.

Experience Replay In order to approximate the expectation, we can optimise for the mean loss using a mini-batch of samples (s, a, r, s') from an *experience replay* buffer D. Deep Q-learning introduces this buffer to improve learning efficiency and stability. Instead of learning from each interaction with the environment in real-time, the agent stores its experiences, consisting of the transition tuple (state s_t, action a_t, reward r_t, next state s_{t+1}, done flag d_t) in a replay buffer. This done flag is set if state s_t is terminal and the episode has ended. In that case, the TD error in Equation 2.16 simply becomes the difference between the predicted Q-value and the final reward. Since this buffer can quickly become very large, only the most recent transitions are stored and older ones are overwritten.

The DQN is then trained by uniformly sampling random mini-batches of experiences from this buffer to compute an estimate of the mean loss. This technique helps to break the correlation between consecutive experiences, reducing the variance between updates, and allows the agent to learn more efficiently from past interactions by potentially sampling the same transition many times. The network parameters are updated by minimising this loss function using a form of SGD, such as Adam or RMSprop, which iteratively adjusts the weights of the network to minimise the difference between the predicted and target Q-values.

Extensions Several extensions to the original DQN have been proposed to improve its performance and stability. One such extension is Double DQN [136], which addresses the issue where the action-values would be overestimated under certain conditions. In Equation 2.16, the same values are used to select the best action and to obtain the action-value, which makes it more likely that when this action-value is overestimated, it is also selected as the best action. Double DQN decouples the action selection from the target Q-value estimation by using the Q-network (with weights θ) to select the action, but using the target network (with weights θ^-) to evaluate it.

Another important extension is Prioritized Experience Replay [119], which improves the efficiency of experience sampling by prioritising transitions based on their TD (Temporal Difference) error. Instead of sampling uniformly from the replay buffer, transitions with higher TD errors are sampled more frequently, as the agent still has much to learn from these experiences.

Finally, Dueling DQN [141] introduces a new network architecture that separates the estimation of the state-value and the advantage of taking a particular action in that state as two distinct streams, which are finally combined to produce the Q-value for each action. By decoupling the estimation of state-value and advantage, the network can learn the value of each state more efficiently from every sample, and the advantage stream can focus on learning the relative importance of each action in a given state.

2.2.6.3 Proximal Policy Optimization

PPO [122] is an example of a policy gradient method, a subset of policy-based methods that estimate the gradient of the expected reward with respect to the policy parameters and adjust the policy in the direction of increasing expected reward. Unlike value-based methods, such as DQN [94], PPO directly optimises the policy $\pi(a|s)$ that maps states to actions. It further belongs to the family of actor-critic methods, which maintain both a policy (actor) and a value function (critic) to guide the learning process.

PPO improves upon earlier policy gradient methods, such as REINFORCE [145], by introducing a clipped objective function that prevents the policy from moving too far from the current policy during each update step. This approach helps to prevent excessively large policy changes that can lead to divergence. The core idea is to maximise the following objective:

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a\sim\pi_\theta}\left[\min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}A^{\pi_\theta}(s,a),\ \text{clip}\big(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, 1-\epsilon, 1+\epsilon\big)A^{\pi_\theta}(s,a)\right)\right] \quad (2.17)$$

The range of allowable policy updates is controlled by the hyperparameter ϵ, which ensures that these updates are clipped within the range $(1-\epsilon, 1+\epsilon)$. A^{π_θ} is an estimator of the advantage function, which measures how much better an action is compared to the average action in that state. It can be computed using the difference between the observed return at time t and the value function estimate:

$$A_t^{\pi_\theta} = r_t + \gamma V(s_{t+1}) - V(s_t) \quad (2.18)$$

The value function $V(s)$ is estimated using the critic network (or head), which is updated using MSE between the observed return $\hat{V}_t$ and the value function estimate:

$$\mathcal{L}_{\text{critic}}(\theta) = \mathbb{E}_{s\sim\pi_\theta}\left[(V_\theta(s) - \hat{V}_t)^2\right] \tag{2.19}$$

Here, $\hat{V}_t$ is the target value for state s_t, typically computed using either Monte Carlo estimates or bootstrapping (as in TD-learning). GAE (Generalized Advantage Estimation) [121] can further be used to reduce the variance and improve the accuracy of the value estimates by calculating a smoothed form of the advantage function.

Similarly to DQN, the expectation in the objective functions is approximated using mini-batches of transitions from interacting with the environment, and the parameters θ are adjusted using gradient ascent with an optimizer, such as Adam [70]. Unlike DQN, PPO is primarily an on-policy algorithm (although an off-policy variant exists), meaning that it learns the policy that is currently being used to interact with the environment, so it does not maintain a replay buffer with transitions from past policies.

We also use a similar algorithm called A2C [92] in our experiments, which can be seen as a special case of PPO, where the clipped objective is replaced by a simpler form of the policy gradient [53].

2.2.6.4 Soft Actor-Critic

SAC [47] is an off-policy actor-critic algorithm designed for continuous action spaces, which combines elements of policy optimisation and entropy maximalisation to encourage exploration. Unlike traditional RL methods that focus on maximising the expected return, SAC aims to maximise both the reward and the entropy of the policy. This dual objective is designed to yield more robust policies by ensuring sufficient exploration, preventing premature convergence to suboptimal policies.

The SAC algorithm maintains three networks: an actor, which represents the policy $\pi(a|s)$, and two critics, which approximate the action-value functions $Q(s, a)$. The use of two critics is inspired by the Double Q-learning framework, aiming to mitigate the overestimation bias commonly seen in value-based methods. Additionally, it maintains a target network for each critic, which is slowly updated towards the current critic parameters using a soft update rule to stabilise training. The policy network outputs the parameters of a Gaussian distribution $N(\mu, \sigma)$, allowing for continuous action spaces.

The overall objective of SAC can be framed as follows:

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}\left[\sum_t \gamma^t \left(r_t + \alpha\mathcal{H}(\pi(\cdot|s_t))\right)\right] \tag{2.20}$$

Here, $\mathcal{H}(\pi(\cdot|s_t))$ represents the entropy of the policy at state s_t, and α is a temperature parameter that controls the trade-off between reward and entropy maximalisation.

The Q-networks are trained to minimise the soft Bellman residual:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}} \left[\left(Q_{\theta_i}(s_t, a_t) - y(r_t, s_{t+1}) \right)^2 \right] \tag{2.21}$$

$$y(r_t, s_{t+1}) = \mathbb{E}_{a_{t+1} \sim \pi(\cdot|s_{t+1})} \left[r_t + \gamma \left(\min_{j=1,2} Q_{\theta_j'}(s_{t+1}, a_{t+1}) - \alpha \log \pi(a_{t+1}|s_{t+1}) \right) \right] \tag{2.22}$$

Here, $\log \pi(a_{t+1}|s_{t+1})$ denotes the entropy term that is subtracted from the Q-value to encourage exploration. The expectation of the next action is sampled with the current policy, and the target Q-value is computed using the minimum of the two target critics to reduce overestimation bias. These target critic networks (with parameters θ') are updated using a soft update rule (through Polyak averaging) to slowly track the main critic networks (θ_i with $i \in \{1, 2\}$), which are in turn updated individually using the MSE between the predicted Q-value and the computed target Q-value. As SAC is an off-policy algorithm, similar to the DQN, transitions are stored and samples from a replay buffer $\mathcal{D}$ to train the networks.

The policy network is updated to maximise the following objective:

$$\mathcal{L}(\phi) = \mathbb{E}_{s \sim \mathcal{D}} \left[\mathbb{E}_{a \sim \pi_\phi} \left[\min_{i=1,2} Q_{\theta_i}(s, a) - \alpha \log \pi_\phi(a|s) \right] \right] \tag{2.23}$$

In this case, ϕ represents the policy's parameters, and the objective aims to find actions that maximise both the Q-value (exploitation) and maintain a high entropy (exploration). The temperature parameter α can either be a fixed constant or adapted during training to balance exploration and exploitation dynamically. When adaptive, α is adjusted using a loss function designed to keep the entropy near a target value $\mathcal{H}_{target}$:

$$\mathcal{L}(\alpha) = \mathbb{E}_{a \sim \pi_\phi} \left[-\alpha \left(\log \pi_\phi(a|s) + \mathcal{H}_{target} \right) \right] \tag{2.24}$$

2.3 Model Compression

The demand for computationally efficient models has led to the development of various model compression techniques, which aim to reduce the memory, storage, and computation requirements of DNNs without significantly sacrificing performance [23]. These methods make it feasible to run DL models on low-power devices, accelerating inference and reducing latency. Among the most prominent techniques for model compression are KD (Knowledge Distillation), pruning, and quantization (see Figure 1.3 for an overview).

We mostly focus on KD, since it shows the highest compression potential for use on general-purpose low-power devices. The other two methods also have their unique advantages, however, which we will explore throughout this thesis. For example, iterative pruning generally has less impact on the internal representation and network output distribution, allowing for more straightforward retraining after compression. Quantization, on the other hand, is often a requirement for deployment on embedded devices, which commonly

lack the computational resources to perform floating-point operations. To get the best of both worlds, a combination of these methods can be used, as they are often complementary and can be applied either sequentially or simultaneously.

2.3.1 Knowledge Distillation

KD is a model compression method that was first introduced by Hinton et al. [52] for SL. It compresses a DNN by transferring knowledge from a larger teacher network to a smaller and, therefore, more efficient student network. The core idea behind KD is to leverage the softened outputs of the teacher model, typically consisting of the class probabilities rather than hard class labels, to train the student model. This enables the student to capture the teacher's nuanced behaviour, especially for examples that are difficult to classify, where the teacher might produce output probabilities that reflect its uncertainty between specific options.

Formally, the teacher model's output for a given input is denoted by a probability distribution p_T, and the student model's output is p_S. The student model is trained by minimising a loss function that combines two components: the traditional loss (e.g., cross-entropy) on the hard labels and a distillation loss on the soft predictions of the teacher. This distillation loss is typically the KL-divergence between p_T and p_S, weighted by a temperature scaling factor τ that smoothens the output distributions:

$$\mathcal{L} = \alpha \, \mathcal{L}_{\text{hard}}(y, p^S) + (1 - \alpha) \, \mathcal{L}_{\text{soft}}(p^T, p^S, \tau) \tag{2.25}$$

Here, α is a hyperparameter that controls the balance between the two losses. By learning from both the hard labels and the more informative soft targets, the student model can generalise better than it would by only learning from the hard labels, thereby achieving a compressed model with competitive performance.

Once both networks are trained on a more powerful computing platform, the student can be efficiently deployed on low-power edge devices, where the number of parameters in the teacher network would require too much memory and computational power.

2.3.1.1 Policy Distillation

This concept was later extended to DRL policies (specifically DQNs) by Rusu et al. [115] in the form of PD (Policy Distillation), where a student policy is trained to predict the same actions as the teacher when given the same observations as input.

Replay Memory In supervised KD, the student is typically trained using the same dataset as the teacher, but with labels that are derived from the teacher's output in addition to the ground truth. But in DRL, the original training data is generated based on the interactions between the agent and the environment, and depends on the actions selected by a policy that evolves over time. In contrast to supervised KD, the student therefore only learns from the teacher's policy, not from any ground truth labels.

In PD, the training data is collected by observing the teacher's interactions in the environment, and storing the observations and the teacher's network outputs in a replay memory (D). This replay memory is periodically updated to broaden the distribution of states the student can learn from.

Control Policy The original PD method [115] was teacher-driven, meaning that the policy of the teacher is followed while collecting transitions to fill the replay memory. With a student-driven control policy, the actions are chosen by the student while still storing the teacher outputs for the same observations in D [26]. This reduces the distribution shift between the data the student is trained on and what it encounters during testing, compared to the teacher-driven approach. Small inaccuracies of the student policy can be an insignificant contribution to the distillation loss but have a large impact on the task performance when this causes a transition to a part of the state space that is not encountered when only following the optimal trajectories sampled from the teacher policy.

By following the student as the control policy instead, these mistakes will also be encountered during training. This increases the distillation loss for those suboptimal transitions and allows the student to learn how the teacher would recover from them. In theory, those errors should not occur when the student is trained to accurately emulate the teacher policy. But especially in the context of model compression, where the students are only a fraction of the size of their teacher, this is a difficult objective to achieve without overfitting.

For some environments and teachers, this distribution shift is more pronounced than for others, making the student-driven approach not necessarily the best choice. Sometimes, it can also lead to slower convergence because the first collected trajectories are suboptimal, and it is more expensive during training since the network outputs of both the teacher and the student are needed for each transition during data collection.

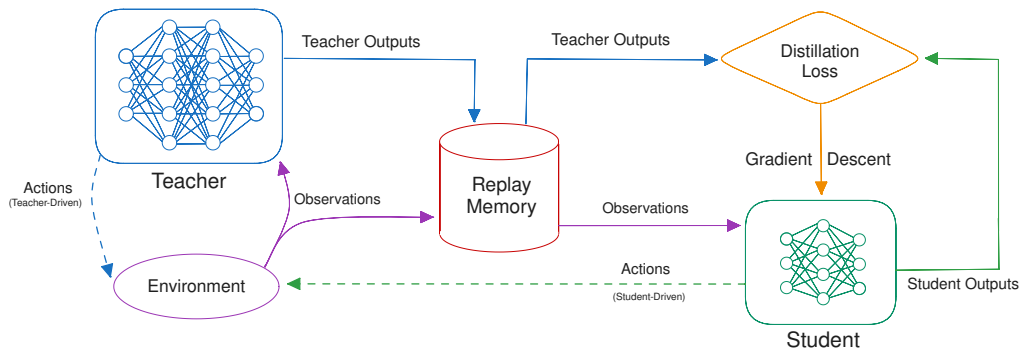


Figure 2.6: An illustration of the PD algorithm.

Motivation Instead of directly training a smaller network, the student only needs to learn how to follow the final teacher policy, while the teacher still contains redundant exploration knowledge about suboptimal trajectories [115]. This knowledge is necessary to find the optimal policy but not to follow it, so it can be omitted from the student. In

student-driven distillation, the student also learns more exploration knowledge, but in practice, it will still follow the final teacher policy relatively closely.

DRL models are also known to be highly overparameterized, with many more parameters than strictly necessary to represent the policy, as this helps with alleviating optimisation issues, such as getting stuck in local minima [98]. These issues are less likely to occur when learning to emulate an existing network in distillation [115] through a process that resembles SL.

PD distinguishes itself from imitation learning by not simply learning the best action given a state of the environment, but also valuable secondary 'dark' knowledge that is expressed in the network outputs for all actions [52]. This can, for example, indicate a level of certainty or provide alternative policies that are still viable. Even if the evaluation is deterministic, this has the potential to increase generalisation.

Algorithm 1: Policy Distillation.

Input: Fully trained teacher M_t, RL environment env
Output: Fully trained student M_s
Randomly initialise θ_s of M_s ;
Create empty replay memory D with size S_D;
$D \leftarrow$ update_memory(M_t, M_s, S_D) ;
Function `update_memory`(M_t, M_c, $steps$)**:**
    for $i \leftarrow 0$ **to** *steps* **do**
        if *env.done()* **then**
            $o \leftarrow$ env.reset() `// o = state observation`
        end
        $q^C = M_c(o)$ `// Control policy for action selection: can be either the teacher (`M_t`) or student (`M_s`) network.`
        $q^T = M_t(o)$ `// Teacher outputs are recorded in the replay memory`
        $D \leftarrow$ replace oldest $D_0 \in D$ with (o, q^T) ;
        $a \leftarrow$ argmax(q^c) ;
        $o \leftarrow$ env.step(a) ;
    end
while M_s *has not converged* **do**
    for $i \leftarrow 0$ **to** *batch_size* **do**
        Sample $D_i \subset D$, with size ($S_D/batch\_size$) ;
        $q^S = M_s(D_i)$;
        $\theta_s \leftarrow$ gradient descent using $\mathcal{L}_{KL}(D_i, \theta_s)$ and q^S (from Equation (2.26));
    end
    if *teacher-driven* **then**
        `// Generate trajectories according to the teacher policy.`
        $D \leftarrow$ update_memory(M_t, M_t, $refresh\_steps$) ;
    else if *student-driven* **then**
        `// Follow the student policy to generate transitions.`
        $D \leftarrow$ update_memory(M_t, M_s, $refresh\_steps$)) ;
    end
end

Distillation Loss Since PD was originally developed for DQN teachers, the other network outputs correspond to the state-action (Q) values for all possible discrete actions. Remember that these Q-values represent the expected (discounted) return when taking that particular action in the current state, with the highest value indicating the best action. To train the student, the KL-divergence between the teacher (q^T) and the student (q^S) outputs is minimised, with θ_S the trainable student parameters and τ a temperature used to sharpen or smoothen the teacher outputs:

$$\mathcal{L}_{KL}(D, \theta_S) = \sum_{i=1}^{|D|} \text{softmax}(\frac{q_i^T}{\tau}) \ln(\frac{\text{softmax}(\frac{q_i^T}{\tau})}{\text{softmax}(q_i^S)}) \tag{2.26}$$

The softmax function is used to transform these Q-values into a probability vector over the actions, from which the KL-divergence can be computed. This results in Algorithm 1 and Figure 2.6. Green et al. [43] have extended this approach for use in combination with PPO-based agents for discrete action spaces. There, a similar KL-divergence loss between the two policies is applied directly. This corresponds to substituting the Q-values with the probability logits in Equation (2.26), and using $\tau = 1$.

2.3.2 Quantization

In quantization, models are compressed by reducing the precision of network parameters and activations. At its core, quantization involves mapping a continuous or high-precision set of values to a discrete or lower-precision set [39]. In addition to reducing the number of bits used to represent each value, quantization usually also restricts the range of values that can be represented, by going from a floating-point to a fix-point or integer representation. For example, weights and biases can be approximated using 8-bit integers instead of 32-bit floating-point numbers, leading to significant reductions in memory and hardware complexity [88].

When designing embedded devices optimised for energy efficiency, only supporting 8-bit integers operations can yield between 18.5x and 30x energy savings compared to 32-bit floating-point operations [28], and between 27x and 116x reduction in die area [157]. Integer computations can also be slightly faster than floating-point operations, as they require fewer clock cycles to complete [88], although these gains in inference speed are generally less pronounced than reducing the number of computations in distillation and pruning. There are two main ways of quantizing models:

- **Post-Training Quantization (PTQ):** A full-precision model is transformed into low-precision after training, while approximately preserving its behaviour [39]. This involves mapping the original floating-point values to a smaller range of quantized numbers. It can be applied quickly and easily to existing models without the need for retraining, but simply reducing the precision introduces inaccuracies that can accumulate when propagating forward through the network, likely leading to a measurable decrease in performance.

- **Quantization-Aware Training (QAT):** Here, the network is trained with quantization in mind, with the quantization transformation as part of the architecture. By simulating low-precision arithmetic during training, the model learns to adjust and mitigate the impact of the quantization noise, enabling higher accuracy post-quantization [54]. Optimising low precision parameters directly can be unstable, so a transformation from high to low-precision representations is often used instead [90].

2.3.2.1 Quantization Functions

The core of quantization lies in mapping high-precision values to a lower-precision representation. This mapping can be done using either **linear** or **nonlinear** quantization functions.

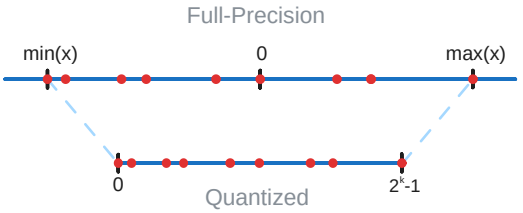


Figure 2.7: An illustration of linear quantization.

In **linear quantization**, the range of values (e.g., floating-point numbers) is divided evenly into bins that correspond to lower-precision values, with equal precision across the range. In linear quantization, the relative distance between values in the original representation is generally maintained after quantization, as illustrated in Figure 2.7. This property is required for (de-)quantizing the network inputs and outputs, and in combination with PTQ for network parameters. This type of quantization is straightforward and works well when the data distribution is approximately uniform. Formally, given a range $[x_{\min}, x_{\max}]$, linear quantization maps a value x to a quantized value $q(x) \in [0, 2^k - 1]$ using k bits as:

$$q(x) = \text{round}\left(\frac{x - x_{\min}}{x_{\max} - x_{\min}} \cdot (2^k - 1)\right) \tag{2.27}$$

This is an example of **asymmetric** quantization, where a zero-point is determined by the minimum and maximum values in the range, in addition to the scale factor, which can better capture the full range of values in skewed distributions. In **symmetric** quantization, only a maximum absolute value is used, which defines both the scale and zero-point, with the range being $[-x_{\max}, x_{\max}]$, centred around zero [153]. Although less flexible, symmetric quantization can be more efficient and easier to implement in hardware.

Equation 2.27 assumes that this original range of values is known and fixed, such as for network parameters of pre-trained models, or the inputs from a dataset where the possible range of values is well-defined, such as images. In this case, **static** quantization can be used, where the quantization parameters (scale and zero-point) are determined once and fixed for the lifetime of the model. On the other hand, in **dynamic** quantization,

the range of values is determined at runtime, which can be useful when the range of values varies significantly between different inputs or during training, but it's also more computationally expensive [29]. In both cases, it can be important to remove outliers when determining the range, as they can have a significant impact on the quantization error and reduce the effective precision of better behaved values.

In contrast to this linear transformation, **nonlinear quantization** allocates more bits to represent values within certain important ranges (e.g., near zero) while using fewer bits for less significant regions [29]. Nonlinear quantization is therefore often preferred when dealing with data that exhibits high dynamic range, as it can help preserve precision where it is most needed, and more closely match the distribution of values that need to be quantized [39]. Techniques like logarithmic quantization are used for nonlinear mappings, which can be more effective in cases where the distribution of values is heavily skewed or has long tails. It uses intervals that grow exponentially, with the difference between consecutive values proportional to the magnitude of the value itself. Similarly, the DoReFa quantization scheme by Zhou et al. [155] uses a tanh function (for $k > 1$) to map arbitrary floating-point values to the range $[-1, 1]$, which is then rescaled based on the maximum value in the network (≤ 1) and remapped to $[0, 1]$:

$$f(x) = \frac{\tanh(x)}{2 \max_{x_i \in \theta_f}(|\tanh_{x_i}|)} + \frac{1}{2} \tag{2.28}$$

With θ_f the set of all values in the network, and x_i the individual values. Finally, the quantized value is derived by applying a linear quantization function to the rescaled value:

$$q(x) = 2\Big(\frac{round((2^k - 1)f(w_f))}{2^k - 1}\Big) - 1 \tag{2.29}$$

Such nonlinear transformations can be less flexible, as they need to be combined with QAT to be effective, because the original value is not preserved in the quantized representation. However, since they more accurately capture the common distribution of parameter values in DNNs, which are rarely uniform, they can be more effective in practice. It also allows for a smoother optimisation with higher accuracy between small values and naturally handles outliers better by allocating fewer bits to represent them [54].

2.3.2.2 Optimisation

To train quantized models in QAT, the entire network needs to be differentiable, which is an essential property for gradient-based optimisation algorithms, which rely on back-propagation to update model parameters. Unfortunately, this is not the case for Equations 2.27 and 2.29, as they contain the non-differentiable rounding operation. To address this, the STE (Straight-Through Estimator) can be used, which approximates the gradient of the quantization function as the identity function during backpropagation [29]. In effect, the gradient of the quantized value is passed through to the previous layer, as if the rounding operation was not present. Although there is no theoretical foundation for this approach, it has been shown to work well in practice, with strong empirical results [74].

2.3.3 Pruning

In pruning, models are compressed by removing redundant or unnecessary connections, neurons, or even entire filters or layers [19]. At its core, pruning involves identifying and eliminating the least important components of a DNN, while preserving its overall functionality and accuracy. By reducing the number of parameters and computations required, pruning can lead to significant reductions in memory and computational complexity, resulting in improved energy efficiency. This approach relies on the observation that many DNNs are overparameterized, containing redundant or unnecessary connections that contribute minimally to the overall performance [37]. For example, weights with near-zero values or neurons with consistently low activation can often be removed without significantly impacting the model's accuracy.

The lottery ticket hypothesis [37] suggests that, within every large dense network, there exists a smaller subnetwork (the "winning ticket") that can achieve the same performance when trained in isolation for at most the same number of epochs. Networks are therefore overparameterized to facilitate optimisation and increase the likelihood of containing a good winning ticket, but many of these parameters are not necessary for the final performance. A winning ticket can be extracted by training the larger network and then pruning the least important weights, leaving behind a smaller subnetwork that performs equally well.

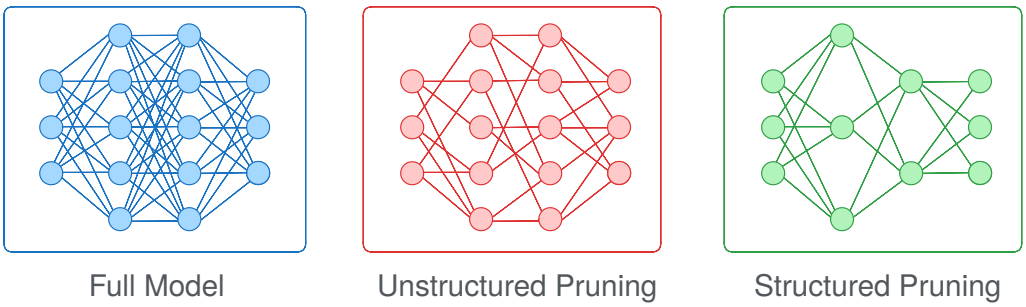


Figure 2.8: An illustration of the difference between structured and unstructured pruning.

There are two main types of DNN pruning, as illustrated in Figure 2.8:

Unstructured: In unstructured pruning, individual weights or connections between neurons are removed from the network, without any constraints on their position or organisation, leading to a sparse structure. This approach is more flexible, provides a fine-grained level of control, and can lead to higher compression rates, but it can be challenging to implement efficiently on hardware due to the lack of structure of the sparse network. The outputs of network layers are typically computed using large matrix operations. Masking individual entries in these matrices does not necessarily lead to a reduction in computational complexity on hardware that is not specifically optimised to handle sparse matrices.

Structured: In structured pruning, entire neurons, filters, or even layers are removed from the network, leading to a more regular reduced structure. This approach is generally more efficient, as it directly reduces the number or size of the network's weight matrices, leading to a proportional reduction in computational complexity on any hardware. While structured pruning tends to achieve less granular compression compared to unstructured pruning, it can provide greater practical benefits in common deployment settings.

2.3.3.1 Pruning Criteria

Identifying which components of a network to prune is a non-trivial task, as it requires determining the importance of each parameter to the overall performance. This can be done using various criteria, such as:

Magnitude-Based: One of the simplest and most widely used methods for identifying weights to prune is based on their magnitude [79]. The underlying assumption is that weights with smaller magnitudes contribute less to the network's output and can be removed with minimal impact on the network behaviour. In magnitude-based pruning, weights below a certain threshold (or a percentage of the smallest weights) are pruned. This approach is effective for unstructured pruning and can also be adapted for structured pruning by applying the magnitude criterion to groups of weights, such as entire filters or channels. Several methods exist to encourage sparsity in the network, such as L_1 regularisation, which adds a penalty term to the loss function based on the absolute value of the weights, thereby increasing the likelihood of low magnitude weights that can be easily pruned.

Gradient-Based: More sophisticated pruning techniques evaluate the gradients of the network's loss with respect to its parameters to identify parts of the model that have minimal impact on the optimisation process [48, 79]. Connections with small gradients are considered as less important, as they contribute less to the loss reduction during training. In some variants, higher order derivatives of the loss function with respect to the weights can also be used to assess the curvature of the loss landscape, further guiding pruning decisions [48].

Importance Scores: In some cases, auxiliary metrics or heuristics are used to rank weights based on their importance. Filters or neurons that consistently produce low activations across the dataset are considered less important and can be pruned. One version, first proposed by Balemans et al. [11], uses the LRP method to identify neurons that are relevant for making specific predictions based on example inputs. This can be particularly effective for pruning a network to optimise it for a subset of the original tasks it was trained for, especially when working with pre-trained models.

LRP was first proposed by Bach et al. [10] in the context of interpretability to explain the predictions of DNNs by attributing relevance scores to features in the input data. It works by propagating a network's output back through the layers and assigning partial prediction contributions to each neuron, depending on its activation strength.

The relevance score R_j^l of neuron j in layer l is computed (in case of LRP-0) by summing the relevance scores of all neurons in $l+1$ connected to j, weighted by their connection strength for the given input.

$$R_j = \sum_{k \in l+1} \frac{a_j w_{jk}}{\sum_{i \in l} a_i w_{ik}} R_k \tag{2.30}$$

The relevance of output neurons is first set to their activation, followed by propagating this computation back through all network layers to assign the contribution of each input feature. When applied to pruning, neurons with consistently low relevance scores can be identified and removed from the network, as they are less important for making predictions on the given dataset.

Search-Based: Some methods use optimisation algorithms to find the optimal subset of weights to prune, based on an objective function. These methods can be computationally expensive, as they require evaluating the performance of different pruning configurations, but they can lead to more effective compression rates. Evolutionary algorithms [104], RL [45], and other optimisation techniques can be used to search for the best pruning strategy, often in combination with other criteria such as magnitude or gradient-based pruning.

2.3.3.2 Pruning Strategies

Besides the decisions on how granular the pruning should be applied and which criteria to use for identifying which components to prune, there are also different strategies for how much of the network to prune at once.

One-shot pruning involves removing a large portion of the network's weights in a single step after training is complete, optionally followed by a fine-tuning phase to recover any lost accuracy [37]. This method is computationally efficient, as it only requires one round of pruning and fine-tuning, making it appealing when resources are limited or quick model compression is needed. However, pruning too many weights at once can lead to significant accuracy degradation, as the network may struggle to adapt to the sudden reduction in parameters. While one-shot pruning is effective in highly overparameterized networks, it can be less precise, potentially removing weights that collectively contribute to the accuracy.

Iterative pruning, on the other hand, removes a small fraction of weights for multiple steps, with each pruning phase followed by retraining (or fine-tuning) to restore accuracy before further pruning [37]. This gradual approach allows the model to adapt and redistribute importance among the remaining parameters, leading to more effective compression and better retention of the network's effectiveness. Although iterative pruning typically results in higher accuracy, it is more computationally expensive due to the repeated cycles of pruning and fine-tuning.

Chapter 3

Actor-Critic Distillation

The contributions presented in this chapter are based on the following publication:

> **Thomas Avé**, Kevin Mets, Tom De Schepper, and Steven Latré. *"Quantization-aware Policy Distillation (QPD)."* In Deep Reinforcement Learning Workshop, NeurIPS 2022, 9 December, 2022, pp. 1-15. 2022.

3.1 Introduction

As discussed in the previous chapters, DRL has achieved remarkable success across various domains, from playing complex games [94] to controlling robotic systems [59], but the deployment of DRL models in real-world applications often faces challenges related to computational efficiency and resource constraints. We also highlighted PD as the most promising approach to address these challenges, as it improves model efficiency in a way that is effective on all general purpose hardware, while providing strong compression results and even a form of regularisation that can benefit generalisation. Traditionally, PD was developed for the value-based DQN method, but recently actor-critic methods such as PPO and A2C have risen in popularity as the preferred choice for many DRL tasks due to their high sample efficiency, stability and flexibility [122, 92].

Existing work has had some success in distilling PPO models by focusing only on the policy head and applying the same loss as for DQNs, but without the temperature in Equation 2.26 [43]. Our work extends on this by investigating the effects of the teacher algorithm choice more closely, including for an additional A2C teacher.

We also believe that the state-value predictions from the critic head provide additional knowledge that can be used to derive a more informative learning signal for the student model, potentially improving its internal representation and generalisation capabilities. A closer look at how the distillation loss is impacted by the different types of output values could also provide key insights for improving this process, as opposed to simply removing the scaling factor altogether.

We therefore begin the contributions of this thesis by addressing how we can improve the distillation process for arguably the most popular class of DRL algorithms, by exploiting properties unique to actor-critic models. This results in a proposed novel distillation loss

that incorporates an auxiliary component for distilling state-value predictions from the critic, alongside the policy distillation from the actor.

Not only did we observe an improvement to the internal representations learned by the student model when applying this approach, but we also measured a 7% increase in the average return it obtained, all without impacting the computational overhead of the final model. Additionally, we show how softening the teacher's outputs can facilitate the transfer of secondary knowledge inherent in the stochastic policies of policy gradient teachers. We evaluate these enhancements in the Atari Breakout environment using PPO and A2C models, for seven student sizes, and show how they outperformed vanilla PD with a DQN teacher both in terms of absolute and relative improvements, suggesting that actor-critic teachers can be more effectively distilled.

3.2 Methodology

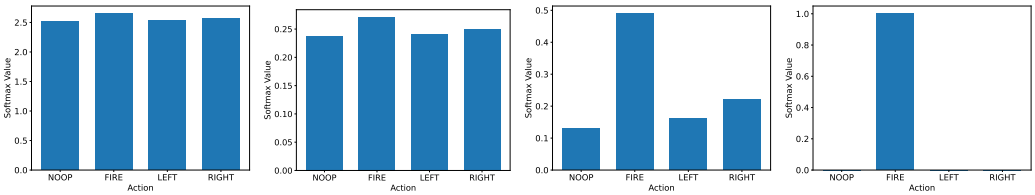
In contrast to the original DQN teachers, which model a state-action value (Q) function, actor-critic algorithms such as PPO [122] and A2C [92] operate by modelling the policy (π) in the actor directly, in addition to a critic head for state-value predictions, as illustrated in Figure 2.5. In this chapter, we focus on architectures for discrete action spaces, where the policy is modelled using a probability distribution over the possible actions. Another difference is that the stochasticity of the actor-critic policy needs to be maintained when transferred to the student, while for a value-based teacher, the student only needs to learn to choose the same deterministic actions.

Due to this difference between learning a value function or an explicit policy, we show that some teachers are more compatible with the distillation process, and later in Chapter 4 that some representations are more easily transferable to a lower-precision student, but that this also requires changes to how the teacher outputs are distilled into the student.

3.2.0.1 Actor Smoothing

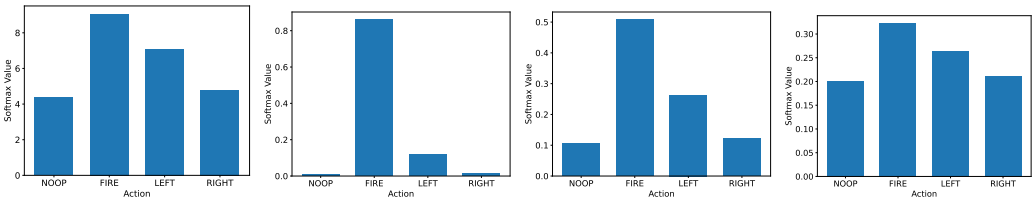
We first explore the characteristics of the Q-value predictions in a DQN and compare this to the action-probability logits of an A2C model. Each predicted Q-value is often very similar to the other action-values for the same state, since the value of being in a certain state in the environment does not usually change significantly when choosing a suboptimal action once, partly due to the ability for the agent to correct a mistake and still reach the goal. Small inaccuracies in the predicted action-values can therefore have a large impact on the effectiveness of the trained student network, although the loss for these actions is relatively small.

Such DQN outputs can be seen in Figure 3.1a, while in Figure 3.1b the softmax in Equation 2.26 is applied with $\tau = 1$. Rusu et al. [115], therefore, choose a temperature value $\tau < 1$ in Equation 2.26, the effects of which can be seen in Figure 3.1c. They sharpen the DQN outputs to make the differences between the possible actions larger, thereby increasing the likelihood of the student selecting the correct one.



(a) Before softmax **(b)** Softmax, τ=1 **(c)** Softmax, τ=0.1 **(d)** Softmax, τ=0.01

Figure 3.1: DQN teacher outputs, before and after softmax and with different τ values.



(a) Before softmax **(b)** Softmax, τ=1 **(c)** Softmax, τ=3 **(d)** Softmax, τ=10

Figure 3.2: A2C teacher outputs, before and after softmax and with different τ values.

In contrast, the outputs of policy gradient teachers are already more peaked, as can be seen in Figure 3.2b. But these teacher types suffer from the opposite problem during distillation. The loss of the action with the highest probability dominates over the other actions, resulting in little secondary knowledge being transferred. Learning this secondary knowledge is critical for generalisation, as it provides insight for recognising similar situations and viable alternative trajectories. If the action-probabilities are similar, the policy determined it is beneficial to choose these actions similarly often, so transferring knowledge on all viable actions to the student is essential.

Rusu et al. [115] empirically chose $\tau = 0.01$ for DQN teachers, resulting in basically all secondary information being lost, as seen in Figure 3.1d. For policy gradient based teachers, we suggest using $\tau \in [1, 5]$ instead, so more secondary knowledge is transferred, but alternative actions are not overvalued significantly (we used $\tau = 3$, as in Figure 3.2c).

3.2.0.2 Auxiliary Critic Loss

In addition to distilling the actual policy, we explore learning an auxiliary loss for the critic that contains additional dark knowledge. This critic is often learned through a dual-head architecture, because there is overlapping logic required to compute both heads. Distilling the state values in the student will probably have a similar benefit to the internal representation of the policy. Learning from more data should also increase sample efficiency and increase teacher similarity.

We therefore propose a new distillation loss for dual-headed students:

$$\mathcal{L}_{\text{Distillation}}(D, \theta_S) = \left(\lambda \frac{\mathcal{L}_{\text{Actor}}}{v(\mathcal{L}_{\text{Actor}})} + (1-\lambda)\frac{\mathcal{L}_{\text{Critic}}}{v(\mathcal{L}_{\text{Critic}})}\right) * (v(\mathcal{L}_{\text{Actor}}) + v(\mathcal{L}_{\text{Critic}})) \tag{3.1}$$

$$\mathcal{L}_{\text{Actor}} = \sum_{i=1}^{|D|} \text{softmax}(\frac{a_i^T}{\tau}) \ln(\frac{\text{softmax}(\frac{a_i^T}{\tau})}{\text{softmax}(a_i^S)}) \quad \text{and} \quad \mathcal{L}_{\text{Critic}} = \sum_{i=1}^{|D|} \mathcal{L}_{\text{Huber}}(c_i^T - c_i^S) \tag{3.2}$$

Here is a_i the outputs of the actor head, c_i those of the critic, λ a parameter for tuning the relative importance of each head and v a function that returns the value of a scalar, while decoupling it from the backward propagation as if it were a constant. The regular distillation loss based on the KL divergence is used for the actor outputs, while the critic values are distilled using a Huber loss. The critic head returns a single value for each state as opposed to a distribution over actions for the actor, so the KL divergence that is typically used in distillation is not well-suited for transferring this type of value.

A composition of different loss functions is therefore needed, as is done in Equation 3.1. By normalising both sub-losses before combining them, we ensure the loss from one of the heads does not heavily dominate over the other in the weight updates at any point during training. Due to the differences in the magnitude and type of loss functions, they decrease at different rates, meaning that a simple linear combination would not yield the same desired effect for each update step. But through normalisation, the loss from each head is always equally represented in the combined loss (when $\lambda = 0.5$). The multiplier in Equation 3.1 is introduced to ensure that our loss theoretically decreases monotonically for each update step, a property that is required for some optimizers.

Using λ, we can adjust how much each head influences the internal representation of our student model, with 1 corresponding to only learning the action-probabilities (i.e. regular distillation) and 0.5 corresponding to learning from both heads equally. Prioritising the critic loss too heavily could also be detrimental, as this only serves to improve the internal representation, while only the actor head is essential to follow the actual policy. This also means that, after training, the critic head can be completely pruned from the architecture, resulting in no overhead in terms of model size compared to traditional PD.

3.3 Experimental Setup

A trade-off was made by focussing on the Atari Breakout benchmark environment from the Gymnasium project [46] to provide a more in-depth investigation into the impact of different teachers and student sizes, instead of repeating the same experiment for more environments. In order to make our results more reproducible, we chose to use publicly available pre-trained checkpoints from the popular SB3 (Stable Baselines3) project [108] for our DQN, A2C, and PPO teacher models. This project aims to provide reliable implementations for a variety of DRL algorithms that are representative of their respective capabilities. These teachers obtain an average return across 200 episodes of 373, 248 and 339, respectively. This difference is already substantial, so the student results need to be compared both in absolute performance and relative to their respective teacher.

In particular, the A2C checkpoint significantly underperforms, as it should be comparable to PPO [122], but this published checkpoint has already started to regress. Training the A2C teacher ourselves yielded a mean return of 342, but our distillation results did not change significantly when used instead (see the Section 3.4.3 for this experiment). We therefore decided to use the SB3 checkpoint for better reproducibility and to highlight this counter-intuitive result that clearly demonstrates the regularisation effect of PD.

3.3.1 Network Architectures

We consider 7 sizes for the student networks, as shown in Table 3.1, ranging from exceeding the teacher size (XXL), to being almost fifty times smaller (XXS). The architectures for our XS, S, M and XL students are the same as what was used in the original *Policy Distillation* paper [115]. On top of this, we added the architectures for our XXS, L and XXL students following a similar pattern to get a more complete picture of the influence these network sizes have on the distillation results.

All teachers have roughly the same size as the XL student, as shown in Table 3.2, so the level of compression is independent of the used teacher model. For each (student size, teacher algorithm) combination, we collect the average return of 50 episodes for each of 600 training epochs. A training epoch is completed when the student has seen all 540,000 transitions stored in the replay memory (D). Transitions are sampled in a random order from D in sets of mini-batches. After each epoch, the oldest 10% of transitions in D are replaced through new teacher-environment interactions.

These hyperparameters were chosen based on the original findings by Rusu et al. [115]. For the LR and τ value in the distillation loss (Equation 3.2), we instead performed a combination of grid and random search to find the best values for each student size, as these are more sensitive to the chosen architecture and teacher. Ultimately, we found that a LR of 10^{-4} worked well for all student sizes, while τ was set to 3 for the A2C and PPO students, and 0.01 for the DQN students.

Each experiment configuration is repeated using 5 independent runs, and the collected average return for the best student of each run is again averaged.

Table 3.1: Sizes for the students used in our experiments.

Network	Conv. 1	Conv. 2	Conv. 3	Linear 1	Compression Ratio	Parameters
Student XXS	16	16	16	32	47.1x	35 796
Student XS	16	16	16	64	27.6x	61 044
Student S	16	16	16	128	15.1x	111 540
Student M	16	32	32	256	4.0x	424 276
Student L	32	64	64	256	1.9x	882 084
Student XL	32	64	64	512	1x	1 686 180
Student XXL	64	64	64	1024	0.5x	3 335 364

Table 3.2: The number of parameters in the teacher models used in our experiments.

Teacher Model	A2C	PPO	DQN
Parameters	1 686 693	1 686 693	1 686 180

The teacher architectures are the default policies provided by the SB3 project [108] for these algorithms. More specifically, the A2C and PPO network architectures are exactly the same, while the DQN architecture differs only by not having a critic head.

3.4 Results and Discussion

3.4.1 Actor Critic Architectures

Figure 3.3 shows a clear benefit of distilling the critic head in addition to the actor. Students that learned a state-value function as an auxiliary task consistently outperformed those without our proposed loss. The exact choice of λ does not significantly influence the results, so no extensive hyperparameter search for this is necessary, as long as the model is able to learn from it to some degree.

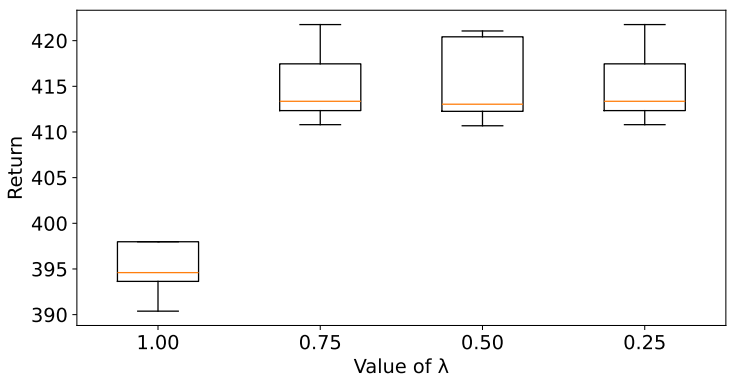
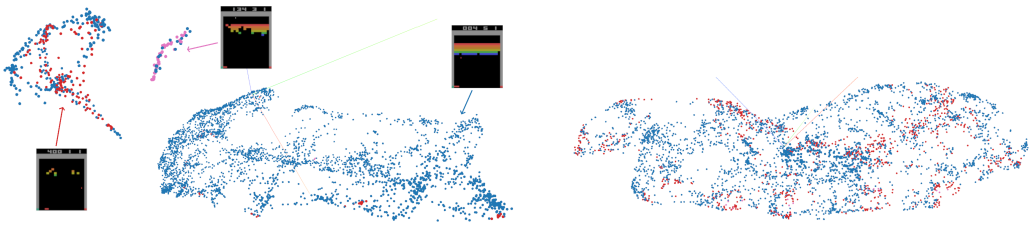


Figure 3.3: Distribution of returns for 5 runs with student size S, an A2C teacher and 4 values for λ in Equation 3.1.

We also investigated the impact of this auxiliary loss on the internal representation of our students by creating UMAP [87] plots (Figure 3.4) of the activations of the last shared network layer between the actor and critic head. Three distinct clusters appear in the activation values when state values ($\lambda = 0.5$) are learned, but not when using the regular distillation loss ($\lambda = 1$). These clusters clearly match with a particular range of critic values, as indicated by the colours. The red cluster mostly contains states with very few blocks remaining, so little expected reward is still to be received. States in the pink cluster correspond to the ball reaching above the blocks for a continued period of time (the tunnel strategy), resulting in a large short-term return.

We conclude that distilling the critic as an auxiliary loss noticeably impacts the internal representation, leading to a clear improvement on the chosen task, without increasing the final model size.



(a) $\lambda = 0.5$ **(b)** $\lambda = 1$

Figure 3.4: UMAP (Uniform Manifold Approximation and Projection) plots of the activations of the last shared layer between the actor and critic head for a student model with size S, an A2C teacher, and two values of λ. Activations with low, medium and high critic values were assigned red, blue and pink respectively.

3.4.2 Teacher Algorithms & Student Sizes

Next, we investigate how the teacher type and student parameter count affects distillation performance. Our actor-critic loss from Section 3.2 using PPO and A2C teachers is compared to a DQN baseline with vanilla PD to determine whether either an explicit policy or an action-value function respectively is more compatible with distillation.

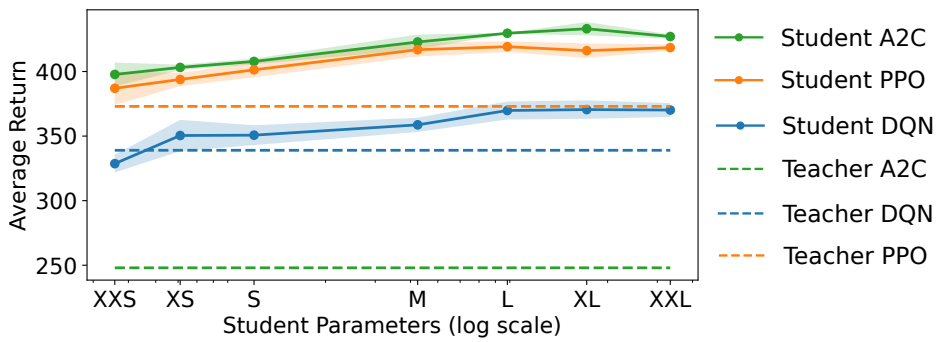


Figure 3.5: Average return for student networks of different sizes and teacher models.

The PPO & A2C students clearly see the best absolute results, as seen in Figure 3.5, suggesting actor-critic methods were more suited for distillation than the value-based DQN teacher. We hypothesise that this is partly due to a smoother loss for learning a probability distribution than for the transformed action-values. Our new actor-critic loss also provides students additional state-values to learn from. Action-values from a DQN contain similar information, but this is lost when applying the softmax function and temperature scaling in Equation 2.26. The DQN teacher was also already weaker to begin with, so less useful knowledge can be transferred, as seen in Figure 3.6, where the DQN students perform more similarly to the PPO ones when viewed relative to their teachers.

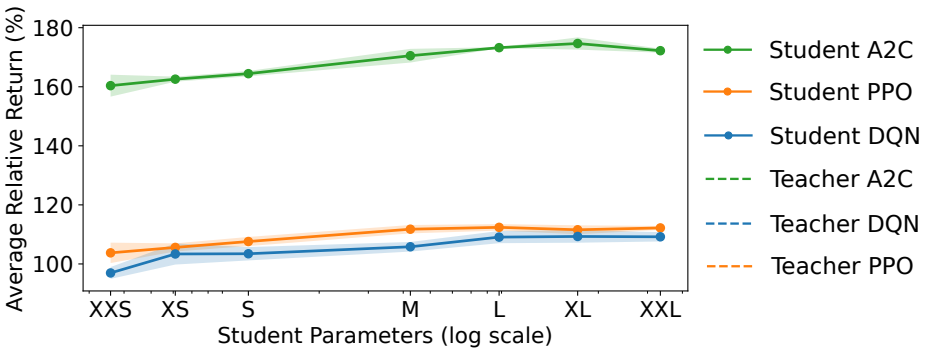


Figure 3.6: Average return for student networks of different sizes, relative to their teacher.

The relative improvement of the PPO students is still measurably higher, however. DQN distillation suffers more from the limited capacity as well, with the XXS size performing worse than its teacher. All students eventually outperform their teacher by between 9% and 90% in this environment, while being up to 47 times smaller. This is a clear example of the regularisation effect of distillation, as was also observed by Rusu et al. [115], especially for the A2C students. The A2C teacher could no longer follow its optimal policy due to regression, but the students were able to recover through this effect. This is in contrast to the DQN teacher, which never learned an action-value function for states with such higher scores, so it cannot be transferred to a student.

All our PPO students significantly outperform even the largest student (similar to our M in size) in the actor-distillation results by Green et al. [43], both compared to their absolute mean return of 248 and relative to their teacher with 277. We also conclude that our proposed distillation loss can outperform traditional PD, both in terms of absolute and relative improvements.

3.4.3 Comparison of Policy Distillation with Different A2C Teachers

The PD process requires the use of teacher networks, which are already fully trained, that transfer their knowledge into (smaller) student networks. As mentioned in Section 3.3, we used the pre-trained checkpoints provided by the SB3 project for our teacher models. However, the published checkpoint for the A2C algorithm and the Atari Breakout environment at the time of writing does not reflect the performance that is reported in other state-of-the-art work.

In our testing, this model obtains an average score of 248, while Shulman et al. [122] reportedly obtained 303, which was higher than their own PPO algorithm could achieve. When looking at the training logs associated with this checkpoint, a higher average score is in fact observed during training, but this is no longer the case for the checkpoint that was eventually published. We therefore retrained the A2C teacher model to make sure that our distillation procedure was not significantly impacted by this discrepancy, yielding a new average score of 342.

We then compared our distillation results using this new teacher model to the results obtained using the checkpoint provided by SB3, resulting in Figure 3.7. Note that this

experiment was done by using $\lambda = 1$ in Equation 3.1 to isolate the influence of the actual policy performance, leading to a deviation from the results shown in Section 3.4.2. These results indicate that, although the new A2C model does perform significantly better, there was no benefit to using this network as a teacher compared to the checkpoint provided by SB3. In fact, student performance was worse when using this teacher for all but the smallest network size. This is a clear testament of the regularisation effect of PD, by which the students using the teacher from SB3 are able to overcome some limitations of the teacher in its current state.

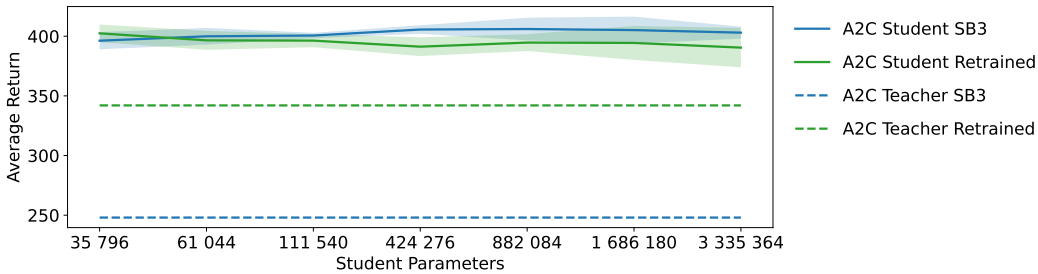


Figure 3.7: A comparison of the average score obtained in the Atari Breakout environment by students trained using different A2C teacher models.

The fact that the teacher which performs better itself is worse at transferring its knowledge to a student might seem contradictory, but as Stanton et al. [129] have shown, KD does not typically work as commonly understood where the student simply learns to exactly match the teacher's behaviour. There is a large discrepancy in the predictive distributions of teachers and their final students, even if the student has the same capacity as the teacher and therefore should be able to match it precisely. In this case, the output distributions provided by the SB3 teacher were a bit more informative for the students, even if the average return obtained for the corresponding actions and states was lower.

For this reason and for better reproducibility, we decided to use the pre-trained checkpoint provided by SB3 as our A2C teacher for all experiments throughout Chapters 3 and 4, even though this teacher performs worse than what is reported in the state-of-the-art.

3.4.4 Real-world improvements on inference speed

In our experiments, we distilled three different teacher models into 7 student models of varying sizes to study the impact of the student capacity on the distillation performance. Our largest student model is 93x the size of the smallest one, but it remains to be seen whether this actually translates to a 9300% speed-up in real-world inference time.

The relative improvement is heavily dependent on the class of device the model is running on, as seen in Table 3.3. This table lists the average number of forward passes per second (FPS) obtained for each device and network type. These results are obtained by running a single observation through the network each time, as this more accurately reflects how the model will be used in deployment compared to running several observations at once, which can be parallelised more effectively.

The PPO and A2C teacher models are combined in this table, as they use the same underlying network architecture. In these benchmarks, the critic head of the PPO and A2C models is also computed to give a more complete picture, although this is usually not necessary for model deployment. If this head were omitted, they would instead share the same network architecture as the DQN and therefore perform equally in this benchmark.

Having more available CPU (Central Processing Unit) cores or access to a GPU (Graphics Processing Unit) generally limits the relative speed-up that can be gained by applying policy compression, since there are fewer computations that can be performed in parallel for smaller networks. This can more clearly be seen in Table 3.4, which shows the maximum relative speed-up for each device type. The only exception that sees a larger speed-up than some CPUs in our testing is the GPU that is part of the Jetson TX2, which was designed specifically for low-power operations.

Table 3.3: Average network forward passes per second for all used architectures and various popular low and high-power devices.

Device	XXS	XS	S	M	L	XL	XXL	PPO / A2C	DQN
Raspberry Pi 3B	135	128	116	100	72	46	25	39	41
Jetson TX2 CPU	79	78	73	67	52	45	32	42	45
Jetson TX2 GPU	815	808	815	816	762	760	736	280	568
Intel NUC i5-4250U CPU	1322	1287	1118	1071	706	535	381	388	487
Nvidia GTX 1080 Ti GPU	2560	2598	2610	2603	2599	2584	2541	1630	1811
Tesla V100 GPU	3620	3628	3638	3484	3466	3458	3625	2212	2477
Ryzen 9 3900X CPU	4535	4456	4327	3743	2649	2272	1442	1685	1934

Table 3.4: The maximal speed-up achieved on various devices by switching from the teacher to the smallest student model.

Device	Maximal speed-up
Raspberry Pi 3B	3.46x
Intel NUC i5-4250U CPU	3.41x
Jetson TX2 GPU	2.91x
Ryzen 9 3900X CPU	2.69x
Jetson TX2 CPU	1.88x
Tesla V100 GPU	1.64x
Nvidia GTX 1080 Ti GPU	1.60x

Due to the lower potential to perform computations in parallel and fixed overheads from context switching, we observe far from the theoretical improvement on inference speed in practice. An improvement of 3.46x from 39 to 135 FPS can still be significant, however, as this is the difference between fluid motion and perceivable lag, especially when considering that in a real deployment there are additional processing steps required for each frame. The least powerful devices see the largest relative improvement, which are coincidentally exactly the target devices for which a lower inference time is critical to enable edge deployment of these models.

The possible degree of parallelisation is not only impacted by the number of parameters in the network, but also by the exact network architecture. When running on a GPU device, our largest student network obtains a higher FPS than the teachers, although its number of parameters is higher. The structure of the network architecture, therefore, also needs to be carefully designed for operation on the target device before applying PD.

3.5 Conclusions

In this chapter, we explored novel approaches to improve PD for actor-critic DRL models, using the PPO and A2C algorithms as representative teachers. Our proposed distillation loss leverages the additional information contained in the state-value predictions of the critic as an auxiliary term to enhance the internal representations learned by the student model and improve its generalisation. An analysis of these representations using UMAP plots revealed that distilling the critic head leads to more structured and interpretable latent spaces, facilitating better decision-making by the student model.

In practice, this yielded a 7% increase in the average return of the student models in the Atari Breakout environment, compared to traditional PD methods, without adding any computational overhead. We also noted that the actor-critic teachers consistently outperformed the value-based DQN teacher in terms of distillation effectiveness, both in absolute performance and relative improvement over the teacher. This suggests that these types of models are more compatible with the distillation process, either due to them natively predicting a probability distribution over actions as expected by the KL-divergence, or through the more informative learning signal that can be achieved by including the auxiliary state-value prediction task in the distillation loss.

We also highlighted how adjusting the temperature parameter in the distillation loss helps to transfer secondary knowledge, which is crucial for actor-critic models that rely on stochastic policies. Evaluating these proposed enhancements on additional, more stochastic environments than Atari Breakout, which is deterministic except for its initial state, would be a valuable next step to further validate their importance. By focussing on a single environment, we were able to conduct a more in-depth analysis of the impact of different teacher types and student sizes on distillation performance, but this also limits the generalisability of our results.

Additionally, we observed a clear regularisation effect of PD, with students often outperforming their teachers by 9% to 90%, even when using significantly smaller network architectures. This effect was particularly pronounced for the A2C teacher, where students were able to recover from teacher regression, showcasing the potential of distillation to overcome limitations in teacher behaviour through generalisation. Our analysis of real-world inference speed improvements revealed that, while theoretical speed-ups based on model size reduction do not directly translate to practical performance gains, significant improvements can still be achieved, especially on resource-constrained devices.

Finally, we conclude that the success of incorporating critic knowledge in the distillation process suggests that there may be other valuable sources of information within complex DRL models that could be leveraged to further improve student performance.

Chapter 4

Quantization-aware Policy Distillation

The contributions presented in this chapter are based on the following publication:

> **Thomas Avé**, Kevin Mets, Tom De Schepper, and Steven Latré. *"Quantization-aware Policy Distillation (QPD)."* In Deep Reinforcement Learning Workshop, NeurIPS 2022, 9 December, 2022, pp. 1-15. 2022.

4.1 Introduction

DRL recently achieved superhuman performance on Atari games [93], Go [120] and Starcraft [137]. But at the same time, their policy networks have become increasingly larger and more complex. Running inference for such a network, such as ResNet-200 [49], can take half a second on a GPU and a large amount of memory. So while these models excel in high-performance computing environments, their application on low-power embedded devices presents significant challenges.

Many embedded systems, such as those found in IoT (Internet of Things) devices, mobile robots, or drones, lack the computational resources and available power to effectively run large full-precision DRL models. In these constrained environments, significant compromises must be made in the balance between energy consumption and performance. One such trade-off is to only support reduced precision arithmetic units on the device, going from 32-bit floating-point to 8-bit integer representations, which has been shown to require up to 30x less energy [28] and 4x less memory.

For a simple sensor node that communicates its gathered data to a central server for processing, 8-bit precision is typically plenty [6]. But for more complex tasks, such as real-time sequential decision-making that requires a DRL model to solve effectively, this precision becomes problematic. Training of DRL models is often already relatively unstable compared to SL when optimising full-precision parameters, and working in low-precision only exacerbates this issue [18]. PTQ can be used to convert an existing model to a lower precision, but this often results in a significant loss of performance [39].

To address this issue, we build on our success in distilling DRL policies in the previous chapter (Chapter 3), and combine it with a novel QAT method. An overview of the general concept and comparison of results can be seen in Figure 4.1.

Through quantization, the precision of the DNN parameters is reduced, requiring less memory and enabling inference on simpler embedded hardware [129]. Distillation can reduce the number of DNN parameters, and therefore computations, by transferring knowledge of a larger teacher network to a student network with fewer parameters. By combining these in QPD, we create a model with (4x) smaller and (up to 47x) fewer parameters, while still maintaining and even exceeding the performance of the teacher model.

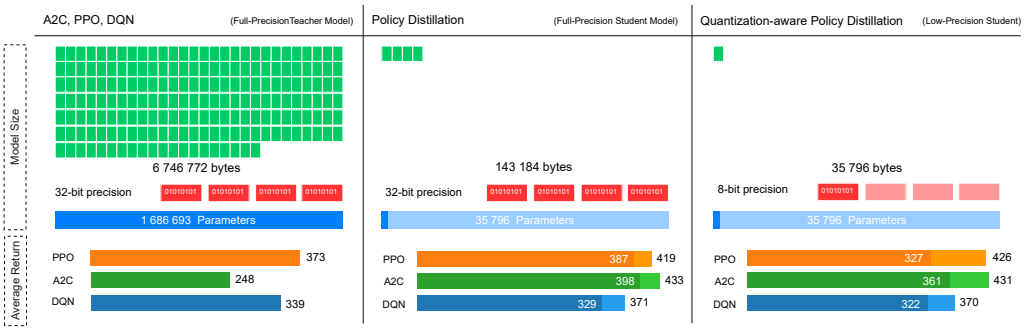


Figure 4.1: The size and performance differences between the original teachers, our PD methods of Chapter 3 and our QPD algorithm. The average returns on the Atari Breakout environment are shown for the best and worst performing student size.

Our main contributions in this chapter are twofold.

1. We outline a novel compression method (QPD), for quantizing DRL networks by providing a smoother transition from high to low-precision weights, which is able to overcome the unstable optimisation that is encountered when training directly in low-precision.
2. We demonstrate how well different DRL teacher algorithms are suited for compression under varying constrained conditions, including limited parameter count, precision, and both combined.

These results indicate that the choice of teacher has a larger impact than simply how well they perform themselves, and that the best suited teacher type depends on what constraints are in place.

4.2 Related Work

This work relates to two main research areas: quantization through distillation and quantization of DRL policies. The following sections highlight not only how our approach differs from previous work in these areas individually, but we also successfully combine them, including with our earlier methods outlined in Chapter 3.

4.2.1 Quantization through Knowledge Distillation

Although the concept of distilling a full-precision model into one with lower-precision parameters is not new, it has thus far only been applied in SL. Our work therefore already differs by integrating this for a DRL setting instead. Mishra et al. [90] proposed to initialise low-precision student weights with the full-precision weights of a teacher with the same size and architecture, and fine-tuning them using KD. We initialise our quantized weights similarly, except that we apply PTQ to a full-precision distilled student instead of to the teacher, as our students are smaller than their teachers.

Kim et al. [69] made the transfer of knowledge smoother by first training the teacher to behave more similarly to the student using a combined loss with the true labels and each other's distributions. They argue that these changes are necessary due to the regularisation effect of distillation, further diminishing the already poor representational power of a quantized model. Their methods do not translate well to a DRL setting, however, where there is no set of true labels. QPD solves the same issue by using a similar concept of smoothing the knowledge transfer, but without depending on labelled data.

4.2.2 Policy Quantization

Preliminary work on the quantization of DRL policies was done by Krishnan et al. [74] in their QuaRL paper. They successfully applied PTQ and QAT using standard uniform affine quantization on policies trained using various DRL algorithms (A2C, DDPG, DQN, D4PG, PPO), resulting in a relative error between 2% and 5% for 8-bit PTQ and no performance loss down to 6-bit QAT.

Björck et al. [18] were instead able to train a DRL policy directly in a native 16-bit floating-point format using SAC, instead of using a quantization function. They proposed 6 improvements to the numerical stability of SAC and the Adam optimizer to make training as stable as in 32-bit floating-point.

Our work differs by using PD for training 32-bit parameters to initialise the 8-bit ones and to continue training after quantization using the same state distribution, in addition to enabling even further compression by also reducing the parameter count. Furthermore, we employ a nonlinear function in both PTQ and QAT for quantizing the trainable parameters, which should more accurately approximate the true distribution of the full-precision parameters.

4.3 Methodology

Combining quantization and PD has the potential for a network to be compressed significantly more than simply applying both techniques separately. Both techniques show diminishing returns for very small networks, either because of excessive quantization noise [74] or limited representational power, respectively. But employing both methods simultaneously before reaching their respective limit could yield considerably more compression, while still having viable performance.

Quantization and PD also have distinct non-overlapping benefits, which can now both be taken advantage of (see Section 2.3). Optimisation in DRL is generally less stable than in SL, which is only amplified when working in low-precision [18]. QPD alleviates this, partly by being more similar to SL than RL in terms of loss optimisation. Increased precision to accurately represent the slight advantage of certain trajectories can also be beneficial to converging on a more optimal policy, but this is less necessary when emulating an existing policy.

Simply training a low-precision student based on a full-precision teacher network through PD fails to converge to any sensible policy however, as shown later in Section 4.4.1, due to the reduced representational power of the quantized model failing to accommodate for the regularisation effect of PD, which was also observed by Kim et al. [69] in a supervised DL setting. To address these issues, we propose the QPD algorithm, which applies a smoother transition from the full-precision parameters of the teacher to the low-precision student parameters.

4.3.1 Algorithm Overview

This smoother transition is done by first training a set of full-precision student parameters (θ_f) using our novel PD methods discussed in Section 3.2 (phase 1), which serves as a good initialisation for the low-precision parameters (θ_q) by applying PTQ (phase 2). The loss when training with randomly initialised low-precision parameters is very steep, as large adjustments are necessary, which is made infeasible by the optimisation instability due to quantization. Quantizing the full-precision parameters and using them directly is also not possible in combination with the nonlinear quantization method, but they can be used as part of the quantization function when introduced in the computation graph of the network (phase 3). These quantized outputs are already much closer to the final low-precision parameter values compared to using randomly initialised values.

The state distribution encountered before and after the second phase remains identical, since the training data in teacher-driven PD is independent of the student behaviour, making converging back to the already discovered solution easier. After this transformation, the low-precision parameters mainly need to be adjusted to account for the nonlinear distribution shift and to reduce the effects of noise caused by the loss of precision, which is significantly more stable than training completely from scratch.

Furthermore, we continue to use and adapt the full-precision parameters θ_f to optimise the low-precision parameters θ_q while training. Activations in the forward pass are computed using θ_q, while the gradient of the distillation loss ∇L (see Equation 2.26) is back-propagated in full-precision. Since the activations are based on the quantized parameters θ_q, the gradient can take the error introduced by the reduced precision into account, which we can then use to update the full-precision parameters θ_f. By computing the gradient in full-precision during backpropagation, we can achieve a more stable optimisation process, without losing the benefits provided by QAT.

This is done through the use of an STE to circumvent the non-differential property of the quantization function, which consists of simply using the gradient for θ_q to optimise θ_f as-is. After θ_f is updated, we recompute the values for θ_q, such that the gradient updates are also propagated to the low-precision parameters for the next update step.

This procedure is then repeated until the student using θ_q stops improving on the DRL environment, as summarised in Algorithm 2.

Algorithm 2: Quantization-aware Policy Distillation

Input: Trained full-precision teacher M_t, RL environment env
Output: Trained low-precision student M_s

Randomly initialise θ_f of M_s and fill replay buffer D using interactions between M_t and *env*;

```
/* Phase 1: Train a student with full-precision parameters          */
```
while M_s *has not converged* **do**
  for $i \leftarrow 0$ **to** *update steps* **do**
    Sample $D_i \subset D$;
    $\theta_f \leftarrow$ Update θ_f using Equation 3.1 and D_i;
  end
  $D \leftarrow$ Update oldest $D_o \subset D$ using new interactions between M_t and *env*;
end

```
/* Phase 2: Quantize the distilled student network (PTQ)            */
```
$\theta_q \leftarrow$ Quantize θ_f using Equation 2.29;

```
/* Phase 3: Continue training with quantized weights (QAT)          */
```
while M_s *has not converged* **do**
  for $i \leftarrow 0$ **to** *update steps* **do**
    Sample $D_i \subset D$ and quantize it using Equation 2.27;
    Perform forward pass using Equation 3.1, memory D_i and parameters θ_q;
    Compute full-precision gradient ∇L based on low-precision activations;
    $\theta_f \leftarrow$ Update($\theta_f, \nabla L$) ;
    $\theta_q \leftarrow$ Quantize θ_f using Equation 2.29 (QAT);
  end
end

4.3.2 Quantization Functions

Both linear and more complex nonlinear quantization functions are used in QPD, each for a different value type. The inputs of the first network layer and the activations of the final layer need to maintain their approximate value after the transformation, for which we use standard uniform affine quantization (Equation 2.27).

During the second and third phase, we apply the nonlinear method proposed by Zhou et al. [155] (DoReFa, Equation 2.29) to create a set of quantized parameters $\{w_q \in \theta_q\}$ based on the full-precision parameters $\{w_f \in \theta_f\}$ computed during the first phase.

4.4 Results and Discussion

4.4.1 Training Low-Precision Students Directly

To appropriately verify the effectiveness of our QPD method, we start by demonstrating why the first two phases of our QPD algorithm are essential in overcoming the instability of training in low-precision. We do this by performing an experiment where only the third phase is performed, using randomly initialised low-precision parameters instead of deriving them from a full-precision model through phases one and two. Aside from this, the same experimental setup is used as in the previous chapter (see Section 3.3), but focused on an A2C teacher and M sized student.

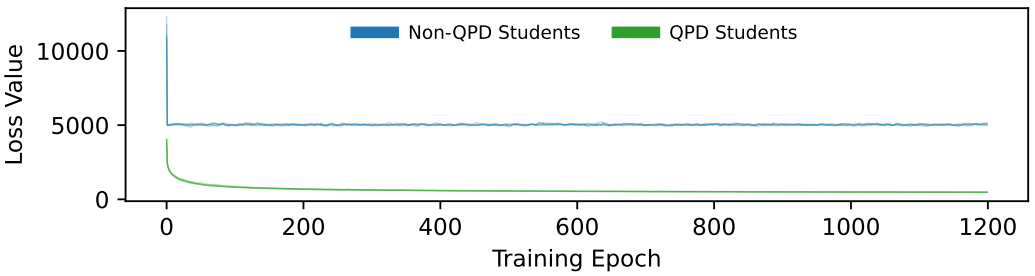


Figure 4.2: The QAT loss for 5 students trained using our full QPD algorithm compared to training a low-precision student directly using PD.

Figure 4.2 shows the training loss of the third phase (QAT) for 5 independent students that were trained using only the last phase (non-QPD) and 5 students with the full QPD algorithm. The loss for non-QPD students drops steeply after the first training epoch, but the models fail to learn anything during subsequent epochs.

The instability of training can also clearly be observed, as the loss is very noisy compared with the smoother descent in our QPD experiments. It also shows how our QPD loss starts at a point that is already lower than the non-QPD loss ever reaches, due to the parameters after applying PTQ already being much closer to their final values, even though the quantization introduced significant initial noise and a nonlinear transformation was used.

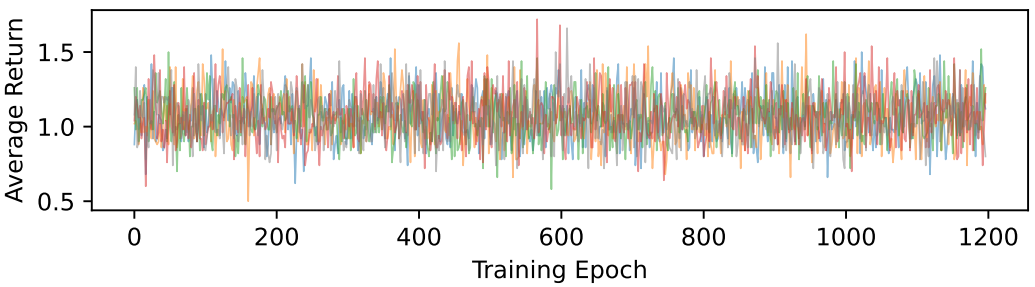


Figure 4.3: The mean return for 5 low-precision students trained directly using PD.

Figure 4.3 depicts the mean return after each training epoch across 50 episodes obtained by

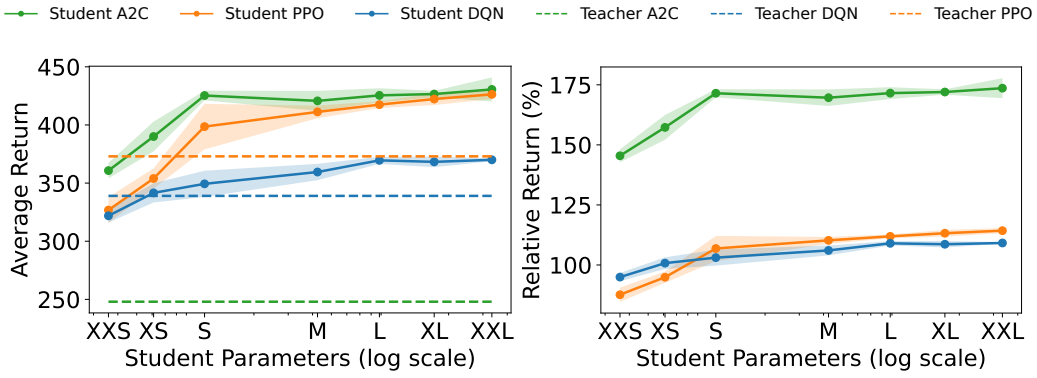
these 5 non-QPD students. This once again confirms that the students are not able to learn any useful behaviour when trained starting with randomly initialised low-precision parameters. The maximum mean return obtained by any of these students is 1.66, compared to 1.4 for a random agent and 412 for our QPD students.

We can therefore conclude that a mechanism for transitioning more smoothly from the full-precision teacher parameters to low-precision student parameters is necessary to alleviate these optimisation issues encountered when training in low-precision, which QPD aims to provide.

4.4.2 Quantization-aware Policy Distillation

This brings us to the investigation into the effectiveness of the QPD algorithm itself. For this, we use the same experimental setup as in the previous chapter (see Section 3.3), to be able to compare the results between the two levels of precision directly, with otherwise exactly the same architecture and training data gathered from identical teachers.

The choice of teacher algorithm could be even more impactful here, due to the limited precision to represent either an action-probability or transformed action-value distribution, as well as the difference in KL-divergence and Huber loss as part of our actor-critic distillation loss (Equation 3.1). Varying the number of student parameters might also have a larger influence, as the representational power is now limited by both the reduced parameter count and precision.



(a) Absolute results

(b) Relative to teacher

Figure 4.4: Average return for students of varying sizes and teachers after applying QPD.

Looking at Figure 4.4, especially the models using policy gradient teachers indeed suffer more significantly from the reduced precision for the smallest network sizes. For both A2C and PPO teachers, the average return drops by 9-13% going from S to XS and again 8% from XS to XXS size, compared to roughly 2% for the DQN teacher and in our full-precision results (in Figure 3.5).

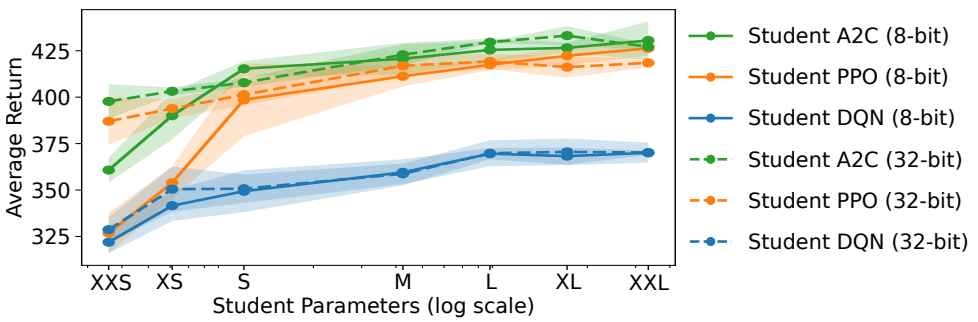


Figure 4.5: Average return for students of varying sizes, teachers, and both precisions.

However, starting from the S size (15x compression), performance mostly recovers to a level comparable to our 32-bit precision distillation, as seen in Figure 4.5. At this size, the policy gradient methods once again outperform the DQN teacher, which is consistent with our previous findings in Chapter 3.

Krishnan et al. [74] (QuaRL) report a score below 400 using 8-bit QAT for both PPO and A2C, which our students outperform starting at size S for A2C and M for PPO, without even considering that we reduce the parameter count in addition to the precision. This confirms that our QPD method is indeed more effective at training low-precision DRL models than simply applying QAT or PTQ with traditional algorithms.

Based on these results, we conclude that QPD is effective at training low-precision DRL models, without any clear performance detriment for networks with a sufficient number of parameters to compensate for the reduced representational power of working in low-precision. The total network size in terms of bytes, which is an important metric in memory-constrained devices, is still 28% lower for the quantized model with size S than the full-precision model with size XXS, while performing significantly better in this environment. The quantized model can additionally be deployed much more efficiently on embedded hardware optimised for low-precision arithmetic. A trade-off is therefore to be made between parameter precision and count, depending on the available hardware optimisations.

4.5 Conclusions

A DRL policy can be compressed by either reducing the parameter count or precision. We developed the novel QPD method that employs both compression types through PD and quantization. QPD uses the improvements introduced in Chapter 3 to first train a set of full-precision parameters, which are then transformed to low-precision (PTQ) and continuously optimised for this transformation by including a quantization function as part of the computation graph (QAT).

Our experiments with a wide range of student sizes (in terms of parameter count) confirmed the high effectiveness of this approach, without any loss in the obtained average return compared to full-precision equivalents when using networks with a sufficient number of parameters. Notably, the choice of teacher algorithm does have a significant

impact on the quantization performance for the smallest networks, with policy gradient teachers suffering more from the reduced precision than the DQN teacher.

This contrasts with our findings in Chapter 3, where students with policy gradient teachers performed better than ones with the DQN teacher in full-precision distillation. We once again observed the same trend in the QPD experiments for students with a sufficient number of parameters, however, so the optimal teacher choice depends on the constraints in place and where you make the trade-off between efficiency and effectiveness.

We conclude that using QPD, it is possible to build a model with up to 47x fewer and at least 4x smaller parameters, while still outperforming their original teacher model. This can significantly reduce deployment costs of these models and allows them to be effectively applied on low-power edge devices.

Chapter 5

Distilling Continuous Actions

The contributions presented in this chapter are based on the following publication:

> **Thomas Avé**, Tom De Schepper, and Kevin Mets. *"Policy Compression for Intelligent Continuous Control on Low-Power Edge Devices."* Sensors 24, no. 15 (2024): 4876.

5.1 Introduction

DRL methods have been shown to be highly effective at solving discrete tasks in constrained environments, such as energy-aware task scheduling [149] and offloading [2, 133] in edge networks, 5G beamforming and power control [91], and NF replica scaling [7] in SDN. These tasks can be solved by performing a sequence of actions that are chosen from a discrete set, such as whether to offload a task or process it locally. However, many task solutions cannot be effectively decomposed in such a way, such as fluid movement in robotics pathfinding that allows precise control [152], the continuous control of drone steering [9], the amount of resources to allocate in micro-grids [83], and multi-beam satellite communication [143]. These types of problems are referred to as continuous control tasks.

In RL, the action space refers to the set of possible actions an agent can take in a given environment. Different types of DRL algorithms have been developed that can learn this type of behaviour by working with continuous action spaces. In contrast to discrete action spaces that take the form of a limited (usually fixed) set of actions to choose from, these algorithms can perform actions using real-valued numbers, such as the distance to move or how much torque to apply. This distinction has significant implications for the types of tasks and the models used to solve them.

Discrete actions are best suited for tasks that involve some form of decision-making in an environment with less complex dynamics. Take, for example, a wireless edge device that optimises for a stable connection while roaming between APs (Access Points). When modelled as a discrete task, the agent can decide at each time step to which AP from a known set it should connect. This might not always be the one with the strongest signal, as the predicted path of the agent could align better with another AP. There are other ways, however, to optimise a connection with an AP before having to initiate handover that require more granular control, such as adjusting the transmission power and data

rates. Such problems, which require continuous control and parameter optimisation, are best modelled using continuous action spaces.

Continuous behaviour is often more challenging to learn than discrete actions, as there is an infinite range of actions to explore before finding an optimal policy. A common approach is, therefore, to discretise the continuous actions into a fixed set of possible values [91], but this can lead to a loss of accuracy when using a large step size or drastically increase the action space and therefore learning complexity [134]. It also removes any inherent connection between values that are close to each other, making it more difficult to converge to an optimal policy. Another solution is to learn a policy that samples actions from a highly stochastic continuous distribution, which can increase robustness and promote intelligent exploration of the environment. This method is employed by the SAC algorithm [47], for example.

Since many of these types of tasks are carried out on battery-powered mobile platforms, additional constraints apply in terms of computing resources and power consumption. In the previous chapters, we proposed new model compression methods based on PD to reduce the size of the underlying DNNs that model the DRL policies while maintaining their effectiveness, making it feasible to deploy complex models on low-power edge devices that need to perform such tasks. However, the original PD [115] method was only designed for policies from DQN teachers, which can only perform discrete actions. Most subsequent research, including our earlier contributions, has also continued in the same direction by improving distillation for teachers with discrete action spaces [26, 8, 43].

DQNs are also fully deterministic, meaning that, for a given observation of the environment, they will always choose the same action. But policies for continuous action spaces are generally stochastic, by predicting a distribution from which actions are sampled. In this chapter, we therefore:

1. Propose three loss functions that allow for the distillation of continuous actions, with a focus on preserving the stochastic nature of the original policies.
2. Highlight the difference in effectiveness between the methods depending on the policy stochasticity by comparing the average return and action distribution entropy during the evaluation of the student models.
3. Provide an analysis of the impact of using a stochastic student-driven control policy instead of a traditional teacher-driven approach while gathering training data to fill the replay memory.
4. Measure the compression potential of these methods using ten different student sizes, ranging from 0.6% to 100% of the teacher size.
5. Benchmark these architectures on a wide range of low-power and high-power devices to measure the real-world benefit in inference throughput of our methods.

We evaluate our methods using an SAC [47] and PPO [122] teacher in the popular HalfCheetah and Ant continuous control tasks [33]. Through these benchmarks, in which the agent needs to control a robot with multi-joint dynamics, we focus on an AMRs (Autonomous Mobile Robots) use case as a representative example of a power-constrained stochastic continuous control task. However, our methods can be applied to any DRL task defined

with continuous action spaces, including the previously mentioned resource allocation tasks.

These experiments demonstrate that we can effectively transfer the distribution from which the continuous actions are sampled, thereby accurately maintaining the stochasticity of the teacher. We also show that using such a stochastic student as a control policy while collecting training data from the teacher is even more beneficial, as this allows the student to explore more of the state space according to its policy, further reducing the distribution shift between training and real-world usage. Combined, this led to faster convergence during training and better performance of the final compressed models.

5.2 Related Work

Several existing papers have already employed some form of model distillation in combination with continuous action spaces, but most of these methods do not learn the teacher policy directly, so they would not strictly be classified as PD. Instead, the state-value function that is also learned by actor-critic teachers is used for bootstrapping, replacing the student's critic during policy updates. This has also been described by Czarnecki et al. [26] for discrete action spaces, but they note that this method saturates early on for teachers with suboptimal critics.

Xu et al. [148] take this approach for a multi-task PD, where a single agent is trained based on several teachers that are each specialised in a single task, to train a single student that can perform all tasks. They first used an MSE loss to distil the critic values of a TD3 teacher into a student with two critic heads. These distilled values are later used to train the student's policy instead of using the teacher's critic directly, as proposed by Czarnecki et al. [26]. Lai et al. [77] propose a similar method, but in a setting that would not typically be classified as distillation, with two students and no teacher. These two students learn independently based on a traditional actor-critic RL objective but use the peer's state-value function to update their actor instead of using their own critic if the peer's prediction is more advantageous for a given state.

Our work differs from these methods by learning from the actual policy of the teacher instead of indirectly from the value function. This more closely maintains student fidelity to their teacher [129] and allows us to more effectively distil and maintain a stochastic policy. The state-value function predicts the expected (discounted) return when starting in a certain state and following the associated policy [72]. This provides an estimate of how good it is to be in a certain state of the environment, which is used as a signal to update the policy (or actor) towards states that are more valuable. It is not intrinsically aware of the concept of actions, however, so it cannot model any behaviour indicating which actions are viable in a given state. The student therefore still needs to learn their own policy under the guidance of the teacher's critic using a traditional DRL algorithm, preferably the same that was used to train the teacher.

Often, the critic requires more network capacity than the actor, so using the larger critic from the teacher instead of the student's own critic could be beneficial for learning [97]. However, the critic is no longer necessary during inference when the student is deployed and can therefore be removed from the architecture to save resources, eliminating any

potential improvement in network size. The general concept of distillation for model compression, where the knowledge of a larger model is distilled into a smaller one, does not apply here. Instead, these existing works focus on different use cases, such as multi-task or peer learning, where this approach is more logical. We therefore focus on distilling the actual learned behaviour of the teacher in the form of the policy, as our goal is compression for low-power inference on edge devices.

Berseth et al. [15] also distil the teacher policy directly in their PLAID (Progressive Learning and Integration via Distillation) method, but by using an MSE loss to only transfer the mean action, any policy is reduced to being deterministic. Likewise, their method is designed for a multi-task setting, without including any compression. We included this method as a baseline in our experiments and proposed a similar function based on the Huber loss for teachers that perform best when evaluated deterministically, but our focus is on the distillation of stochastic policies. Learning this stochastic student policy also has an impact on the distribution of transitions collected in the replay memory when a student-driven control policy is used, so we compare this effect to the traditional teacher-driven method.

5.3 Methodology

We propose several loss functions for the distillation of continuous actions based on a combination of the teacher's mean (μ) and standard deviation (σ), with an overview given in Figure 5.1. Such a loss function should accurately define the similarity between the two policies, based on the action distributions predicted by both networks, with a lower value corresponding to the student matching their teacher's behaviour more closely. The expected effectiveness of the proposed losses depends on whether the teacher performs best in a deterministic or stochastic evaluation and whether a teacher-driven or student-driven setting is used.

5.3.1 Distilling the Mean Action

The first loss is the simplest, serving as a baseline that is most useful in combination with deterministic teachers in the teacher-driven scenario or in case σ was not learned by the model. It consists of only distilling the mean action that the student needs to follow, resulting in a fully deterministic policy, similar to what is proposed by Berseth et al. [15]. The original PD loss (Equation (2.26)) is based on the KL-divergence between two probability vectors, which cannot be used for learning a single mean value for each action [115]. Instead, we propose to use the Huber loss between the mean of the student (μ^S) and the teacher (μ^T) for each action (a):

$$\mathcal{L}(D, \theta_S) = \sum_{i=1}^{|D|} \sum_{a \in A} \text{Huber}(\mu^S_{i,a}, \mu^T_{i,a}, 1) \tag{5.1}$$

$$Huber(a, b, \delta) = \begin{cases} \frac{1}{2}(a-b)^2 & \text{for } |a-b| \leq \delta, \\ \delta \cdot \left(|a-b| - \frac{1}{2}\delta\right), & \text{otherwise.} \end{cases} \tag{5.2}$$

We chose to make use of the Huber loss for this baseline instead of the MSE loss used by Berseth et al. [15] since it is less sensitive to outliers and has a smoother slope for larger values, resulting in it outperforming the MSE loss in our initial experiments.

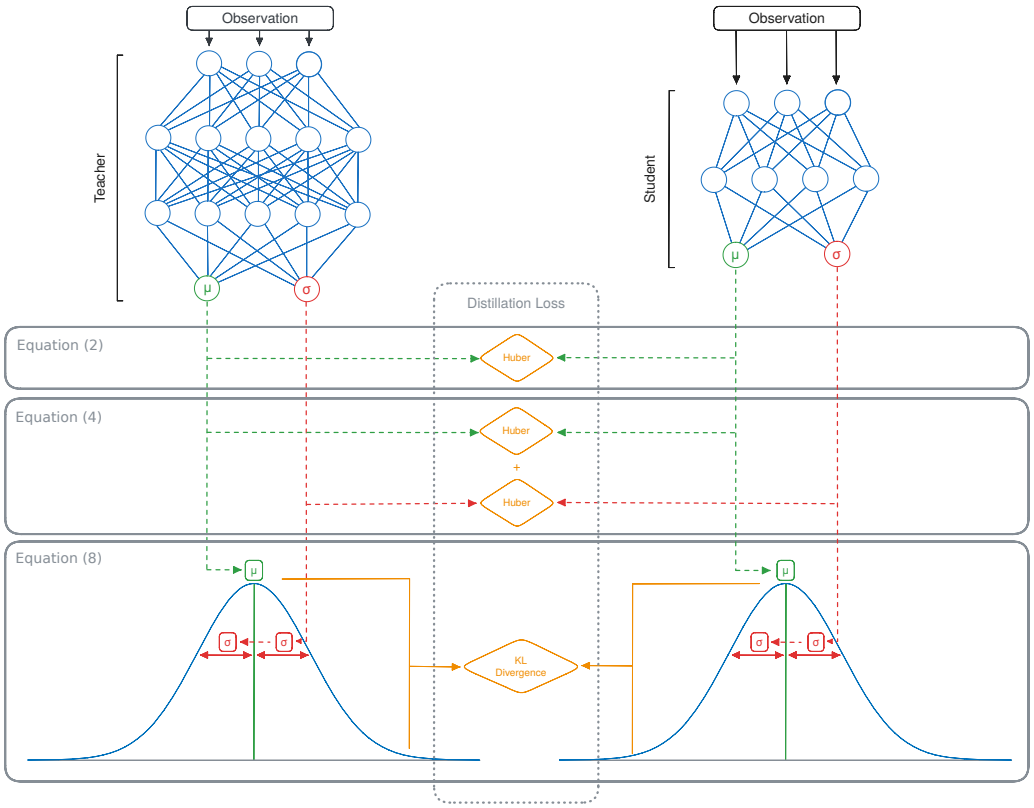


Figure 5.1: An illustration of the proposed distillation losses.

5.3.2 Distilling the Mean Action and its Standard Deviation

Some teachers perform better when actions are sampled stochastically, in which case the student should also learn the value of σ to perform optimally. By learning when precise action is required and when actions can be taken more stochastically, the agent can also build a deeper understanding of its environment. This could be seen as a different form of 'dark' knowledge, similar to the distribution of alternative actions in PD for discrete action spaces [115].

Learning σ is even more important when using student-driven PD, as this allows the student to explore more of the state space to learn multiple viable ways to obtain a high return, further enhancing its representation of the task dynamics. In effect, it enables the exploration-exploitation trade-off to apply in a distillation context, where the mean action would focus purely on exploitation. This has the potential to increase generalisation and robustness against changes in the environment.

In turn, this enables the student to recover more gracefully from mistakes caused by the remaining distribution shift, leading to increased task performance, as reflected by a higher return. It can also help prevent the student from becoming stuck in local minima, where the teacher is less knowledgeable and provides inaccurate behaviour. By encouraging the student to deviate more from this suboptimal strategy, it can move towards a region in the state space where the teacher's guidance is more effective.

If σ is state-independent, this vector can simply be copied from the teacher to the student while continuing to train using Equation (5.1). Note that, in this case, it would likely be beneficial to train the teacher using a state-dependent standard deviation instead. Otherwise, we include an additional term for distilling σ, with λ a scaling factor to ensure that the loss for σ does not dominate over the one for μ or the other way around:

$$\mathcal{L}(D, \theta_S) = \sum_{i=1}^{|D|} \sum_{a \in A} \text{Huber}(\mu^S_{i,a}, \mu^T_{i,a}, 1) + \lambda \cdot \text{Huber}(\sigma^S_{i,a}, \sigma^T_{i,a}, 1) \tag{5.3}$$

Since we have access to both μ and σ, these can be used to once again define a probability distribution. This also allows the student to sample actions based on $\text{N}(\mu, \sigma)$, which was not possible when trained using Equation (5.1). The Huber loss is still not optimally suited for defining a distance metric between two probability distributions, however; it simply defines a distance between the two values separately, without any context of how they are used together.

5.3.3 Distilling the Action Distribution

In traditional PD, the student is also trained using a probability distribution over actions, where this is defined using the probability vector given by the teacher outputs [115]. Instead of learning to reproduce the same precise values as the teacher, such as what we proposed in Equation (5.1) for learning the mean action, the student outputs are shaped to produce a similar probability curve.

This is achieved through a derivation of the KL-divergence for discrete probability distributions that is most often used in the context of DL. In the context of continuous actions, the network outputs are not in the form of probability vectors, so this loss cannot be applied to this setting. Instead, we derive the KL-divergence between two absolutely continuous univariate normal distributions, starting with the general definition of the KL-divergence for distributions P and Q of continuous random variables:

$$\text{KL}(P, Q) = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx \tag{5.4}$$

In our setting, P and Q are normal distributions defined by μ and σ, for which the probability density function is defined as follows:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{5.5}$$

To substitute this in Equation (5.4), we first focus on the log division:

$$\begin{aligned}
\log\Big(\frac{p(x)}{q(x)}\Big) &= \log p(x) - \log q(x) \\
&= \Big[\log\big(\frac{1}{\sigma_p\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu_p}{\sigma_p}\right)^2}\big) - \log\big(\frac{1}{\sigma_q\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu_q}{\sigma_q}\right)^2}\big)\Big] \\
&= \Big[-\frac{1}{2}\log(2\pi) - \log(\sigma_p) - \frac{1}{2}\Big(\frac{x-\mu_p}{\sigma_p}\Big)^2 + \frac{1}{2}\log(2\pi) + \log(\sigma_q) + \frac{1}{2}\Big(\frac{x-\mu_q}{\sigma_q}\Big)^2\Big] \\
&= \Big[\log\Big(\frac{\sigma_q}{\sigma_p}\Big) + \frac{1}{2}\Big[\Big(\frac{x-\mu_q}{\sigma_q}\Big)^2 - \Big(\frac{x-\mu_p}{\sigma_p}\Big)^2\Big]\Big]
\end{aligned}$$

The full equation then becomes:

$$\mathrm{KL}(P,Q) = \int \frac{1}{\sigma_p\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu_p}{\sigma_p}\right)^2}\Big[\log\Big(\frac{\sigma_q}{\sigma_p}\Big) + \frac{1}{2}\Big[\Big(\frac{x-\mu_q}{\sigma_q}\Big)^2 - \Big(\frac{x-\mu_p}{\sigma_p}\Big)^2\Big]\Big]dx \tag{5.6}$$

We then rewrite this using the expectation with respect to distribution P:

$$\begin{aligned}
\mathrm{KL}(P,Q) &= E_p\Big[\log\Big(\frac{\sigma_q}{\sigma_p}\Big) + \frac{1}{2}\Big[\Big(\frac{x-\mu_q}{\sigma_q}\Big)^2 - \Big(\frac{x-\mu_p}{\sigma_p}\Big)^2\Big]\Big] \\
&= \log\Big(\frac{\sigma_q}{\sigma_p}\Big) + \frac{1}{2\sigma_q^2}E_p\big[(X-\mu_q)^2\big] - \frac{1}{2\sigma_p^2}E_p\big[(X-\mu_p)^2\big] \\
&= \log\Big(\frac{\sigma_q}{\sigma_p}\Big) + \frac{1}{2\sigma_q^2}E_p\big[(X-\mu_q)^2\big] - \frac{1}{2}
\end{aligned}$$

Note that we could rewrite $(X-\mu_q)^2$ to:

$$\begin{aligned}
(X-\mu_q)^2 &= (X-\mu_p+\mu_p-\mu_q)^2 \\
&= ((X-\mu_p)+(\mu_p-\mu_q))^2 \\
&= (X-\mu_p)^2 + 2(\mu_p-\mu_q)(X-\mu_p) + (\mu_p-\mu_q)^2
\end{aligned}$$

Substituting this results in:

$$\begin{aligned}
\mathrm{KL}(P,Q) &= \log\Big(\frac{\sigma_q}{\sigma_p}\Big) + \frac{1}{2\sigma_q^2}E_p\big[(X-\mu_p)^2 + 2(\mu_p-\mu_q)(X-\mu_p) + (\mu_p-\mu_q)^2\big] - \frac{1}{2} \\
&= \log\Big(\frac{\sigma_q}{\sigma_p}\Big) + \frac{1}{2\sigma_q^2}\Big(E_p\big[(X-\mu_p)^2\big] + 2(\mu_p-\mu_q)E_p\big[(X-\mu_p)\big] + (\mu_p-\mu_q)^2\Big) - \frac{1}{2} \\
&= \log\Big(\frac{\sigma_q}{\sigma_p}\Big) + \frac{\sigma_p^2 + (\mu_p-\mu_q)^2}{2\sigma_q^2} - \frac{1}{2}
\end{aligned}$$

Finally, we apply this for PD of continuous action spaces:

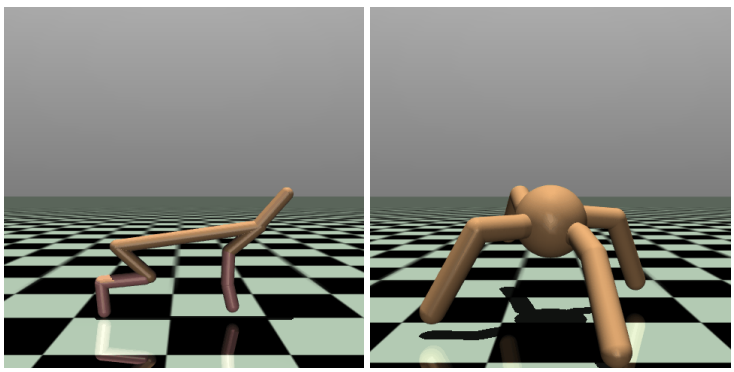
$$\mathcal{L}_{KL}(D, \theta_S) = \sum_{i=1}^{|D|} \sum_{a \in A} \log \frac{\sigma_{i,a}^T}{\sigma_{i,a}^S} + \frac{(\sigma_{i,j}^S)^2 + (\mu_{i,a}^S - \mu_{i,a}^T)^2}{2(\sigma_{i,a}^T)^2} - \frac{1}{2} \tag{5.7}$$

This should allow for a smoother optimisation objective than learning both values using the Huber loss, similar to the distillation of discrete actions.

5.4 Experimental Setup

5.4.1 Evaluation Environments

We evaluate the effectiveness of the loss functions proposed in Section 5.3 in two continuous control environments (Ant-v3 and HalfCheetah-v3) that are part of the Gymnasium project [46], shown in Figure 5.2. These environments were chosen as they are arguably the two most prevalent benchmarks in the MuJoCo suite, the de facto standard for continuous control tasks in DRL research. This allowed us to apply our methods to compress publicly available state-of-the-art DRL models, making it straightforward to compare to existing work and strongly increasing reproducibility.



(a) HalfCheetah-v3 **(b)** Ant-v3

Figure 5.2: A graphical render of the environments used in this chapter.

Environments in this suite have complex, high-dimensional continuous state and action spaces and require sophisticated control strategies [46]. This takes the form of a physics simulation of a robot with a specific morphology and complex dynamic interaction between multiple joints that need to be efficiently coordinated. Distilling a policy that consists of multiple coordinated continuous actions allows us to verify that our proposed loss functions are robust to even the most complex policy architectures. The relationship between actions (torques) and the resulting state of the creature (positions, velocities, angles) is non-linear, making it challenging to learn a compact policy that captures these complex dynamics effectively.

The difficulty of these tasks should result in a pronounced difference in the average return obtained by students of different sizes, allowing us to compare the impact of the compression level for each of the proposed methods. By relying on a physics engine that accurately models advanced dynamics [135], these environments are representative of many real-world continuous control tasks, such as robotics and drone control that require low-power operation at the edge.

The goal in the two chosen environments is to achieve stable locomotion of the robot by applying torques to its joints to move through the environment as quickly as possible. In HalfCheetah-v3, this robot takes the form of a 2D bipedal creature with six controllable joints and therefore six separate continuous values in the action space that need to be distilled. The observation space consists of 17 continuous variables, including the positions and velocities of its limbs and the angles of its joints.

Being inspired by a cheetah, the goal of the agent is to run as fast as possible, even if this means sacrificing stability. The movement of the 3D quadrupedal creature with eight controllable joints in Ant-v3 needs to be more sophisticated to achieve a high return. In addition to being rewarded for efficient forward movement, it needs to balance itself by keeping its torso within a certain height range. If it falls over, the episode is terminated early. It also has a higher-dimensional observation space of 111 continuous variables that now also includes the contact forces applied to the centre of mass of each of the body parts.

Combined, these tasks require an effective balance between exploration and exploitation to learn how to optimally coordinate the different joints to move quickly while maintaining stability to not fall over. A stochastic policy can be beneficial for both these aspects by exploring the state space more effectively and increasing the robustness to recover from unstable configurations.

5.4.2 Model Architectures

To be used as teachers for training our students and as baselines for uncompressed models, we make use of two state-of-the-art DRL algorithms: SAC [47] and PPO. For increased reproducibility, we make use of publicly available pre-trained agents from the SB3 project [108] for our teacher networks. These teachers have different behaviours when evaluated either deterministically or stochastically. Although both are trained to learn a stochastic policy, a PPO agent often performs better during final evaluation when the mean action is chosen, whereas an SAC agent performs better when actions are sampled stochastically. This can be seen in Table 5.1, which shows the average return and standard deviation for 200 episodes using our teacher models in the used environments and for both evaluation methods.

Table 5.1: Average return and standard deviation for our PPO and SAC teachers in the chosen environments using either stochastic or deterministic action selection.

Network	Ant-v3	HalfCheetah-v3
SAC Stochastic	4682 ± 1218	9010 ± 113
SAC Deterministic	1797 ± 993	8494 ± 186
PPO Deterministic	1227 ± 483	5735 ± 723
PPO Stochastic	1111 ± 453	5553 ± 791

The SB3 project uses a different network size for these teachers depending on the environment, as shown in Table 5.2. Note that we did not include the critic head for the teacher sizes in this table, as this is not required for inference. It also shows the parameter count for one of our student architectures (with ID 6) in all its configurations. This student was used to compare our proposed loss functions and the chosen control policy, as this was the smallest architecture where the number of parameters was not yet a limiting factor.

Table 5.2: The number of parameters in our student networks, SAC, and PPO teacher.

Network	Ant-v3	HalfCheetah-v3
Student 6 (μ only)	16,008	9,862
Student 6 (μ and σ)	16,528	10,252
SAC Teacher	98,576	73,484
PPO Teacher	23,312	142,348

The impact of the compression level is further evaluated using ten different student architectures, for which the number of layers, neurons per layer, and total number of parameters are shown in Table 5.3. This ranges from 0.67% to 189% of the size of the SAC teacher. In case a loss function is used that includes the standard deviation, the last layer will have two heads: one for the mean actions and one for the standard deviations.

Table 5.3: Architectures used for students with varying levels of compression.

Network ID	Layers	Neurons per Layer	Parameters
1	2	16	492
2	2	32	972
3	3	32	2028
4	4	32	3084
5	3	64	6092
6	4	64	10,252
7	6	64	18,572
8	4	128	36,876
9	3	256	73,484
10	4	256	139,276

5.4.3 Training Procedure

Each experiment runs for 200 training epochs, with 1 epoch being completed when the student has been updated based on all 100,000 transitions in the replay memory D. Transitions are sampled in random order from D in sets of mini-batches with size 64. After each epoch, the student is evaluated for 50 episodes while collecting the average return, with the action selection being stochastic or deterministic, based on which performs best for this teacher model (see Table 5.1). The oldest 10% of the transitions in D are then also replaced by new environment interactions after each epoch.

These environment interactions are either student-driven or teacher-driven, based on the current configuration of the experiment. Each experiment configuration is repeated for five independent runs, resulting in the mean and corresponding standard deviations of the return and entropy at each epoch discussed in Section 5.5.

We compute the entropy of the action distribution predicted by our students during testing to more tangibly evaluate how well the stochasticity of the teacher is maintained as part of the distillation process. Since these are continuous univariate normal distributions, we compute this using:

$$\mathcal{H}(x) = \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2} \tag{5.8}$$

For a random variable $x \sim N(\mu, \sigma)$, i.e., the action prediction. This entropy is then averaged over all steps in a trajectory.

5.5 Results and Discussion

In this section, we investigate the effectiveness of the three loss functions proposed in Section 5.3 under various circumstances. We start by performing an ablation study to isolate the effects of the chosen loss function, the control policy, and finally the teacher algorithm. This will provide us with a better understanding of how each of these components in our methodology impacts the training process and how they interact with each other to culminate in the final policy behaviour.

This is measured in terms of the average return, but we also analyse the entropy of the action distribution to evaluate how well the stochasticity of the teacher is maintained as part of the distillation process. Afterwards, we perform a sensitivity study of our methodology for different compression levels to evaluate the impact of the student size on the final policy performance. Finally, we analyse the runtime performance in terms of inference speed of each of the student architectures to gain a better understanding of the trade-offs between the different student sizes.

5.5.1 Distillation Loss

To isolate the impact of the chosen distillation loss, we compare the average return of students trained using each of the three proposed loss functions, with the same SAC teacher and a student-driven control policy. Distilling a stochastic policy (learning σ) and using this to collect training data will increase exploration and therefore widen the state distribution in the replay memory. If the student is trained using Equation (5.1) instead, the replay memory will only contain deterministic trajectories, which are not always optimal (see Table 5.1).

As a baseline, we start by comparing the MSE-based loss originally proposed by Berseth et al. [15] between mean actions μ to our Huber-based loss functions ((5.1) and (5.3)), as well as an analogous MSE loss with both μ and σ. The results of this are shown in Figure 5.3. The students trained using our baseline Huber-based loss converge much more quickly and obtain an average return that is 18% higher on average. This confirms the benefit in the context of distillation of the Huber loss being less sensitive to outliers and having a smoother slope for larger values. However, it is also notable that learning the state-dependent value of σ through an auxiliary MSE or Huber loss does not yield any noticeable benefit; instead, this results in a comparable average return to when only the mean action is distilled in this experiment.

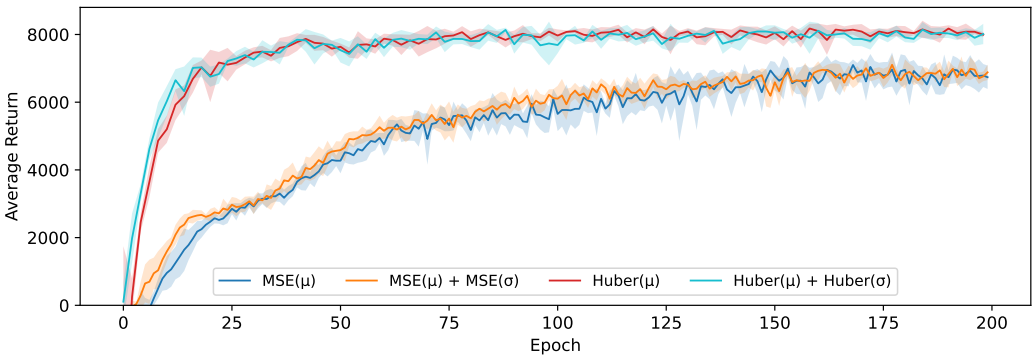


Figure 5.3: The average return obtained using either an MSE or Huber-based distillation loss, in the HalfCheetah-v3 environment and an SAC teacher.

Looking at Figure 5.4, we do see that our proposed loss, based on the KL-divergence (Equation (5.7)) to transfer the action distribution, performs significantly better than those based on a Huber loss. This aligns with our hypothesis that the loss landscape is smoother when shaping the student's probability distribution to match the teacher's, compared to learning mean and standard deviation independently. Learning these values separately, as was the case in Figure 5.3, precisely enough to accurately model the distribution might also require more capacity, resulting in this approach suffering more heavily from the limited capacity of the student. We test this conclusion more extensively in Section 5.5.4 by comparing these results with different student sizes.

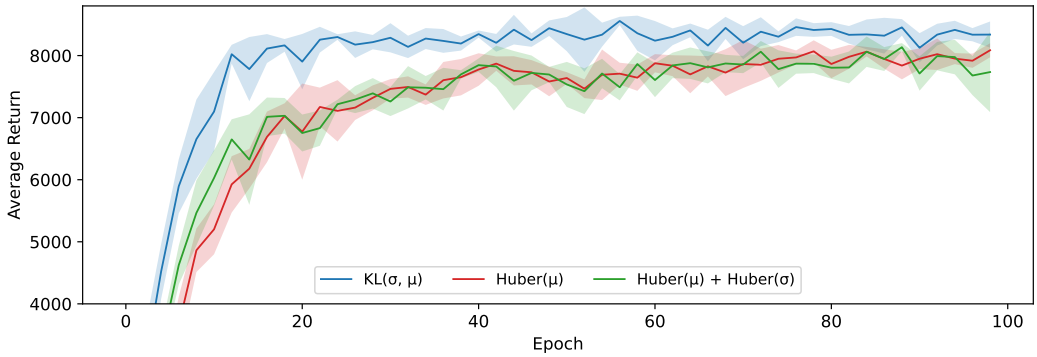


Figure 5.4: The average return of 5 students trained using student-driven distillation in the HalfCheetah-v3 environment with an SAC teacher.

5.5.2 Control Policy

Using a student-driven control policy will result in a different distribution of transitions in the replay memory compared to when using a teacher-driven control policy, where the distribution shift between the training and testing data is more pronounced. In the student-driven setting, the initial distribution will be less accurate and more exploratory, but will gradually converge to the teacher distribution as the student learns. To test this hypothesis for continuous actions, we ran the same experiment as in the previous section, but this time with a teacher-driven control policy.

5.5.2.1 Impact on Average Return

The effects of this distribution shift can be seen in Figure 5.5, which shows the average return for both control policies and all loss functions in the HalfCheetah-v3 environment. The experiments with a teacher-driven action selection perform significantly worse than their student-driven counterparts. This also results in far more variance in performance between epochs, and it takes much longer to converge. As the distillation loss becomes smaller, the students will behave more similarly to their teacher and the distribution shift will eventually reduce, but never disappear completely.

Eventually, the students in the teacher-driven configuration converge on a similar obtained average return, regardless of the used loss function. However, there is a clear order in how quickly the students reach this convergence point, with the agents trained using our loss based on the KL-divergence (Equation (5.7)) being considerably more sample efficient, followed by the agent trained using the Huber loss for both μ and σ (Equation (5.3)) and finally the agent that only learns a deterministic policy in the form of the mean actions μ (Equation (5.1)). This suggests that even though learning a stochastic policy is beneficial in this setting, the remaining distribution shift eventually becomes the limiting factor that causes all students to hit the same performance ceiling. We find that for this environment, the difference between the used control policy is more pronounced than the difference between the used loss functions, but that the KL-divergence loss is still the most effective choice.

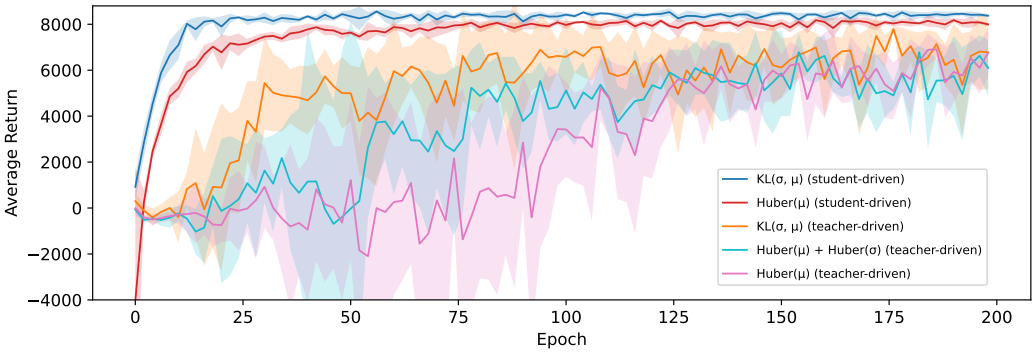


Figure 5.5: The average return during training for student and teacher-driven distillation in the HalfCheetah-v3 environment and an SAC teacher.

Figure 5.6 also shows the average return for all configurations, but this time in the Ant-v3 environment instead. The distribution shift is less pronounced in this environment, resulting in the gap between student and teacher-driven action selection disappearing for all but the students trained using our KL-divergence distillation loss, where using a student-driven control policy still has a noticeable benefit. These are also the two configurations that stand out from the others, with a significantly higher return on average, confirming the same conclusion as in the HalfCheetah-v3 environment that this loss function is the best out of the three considered options for the distillation of continuous actions with a stochastic teacher.

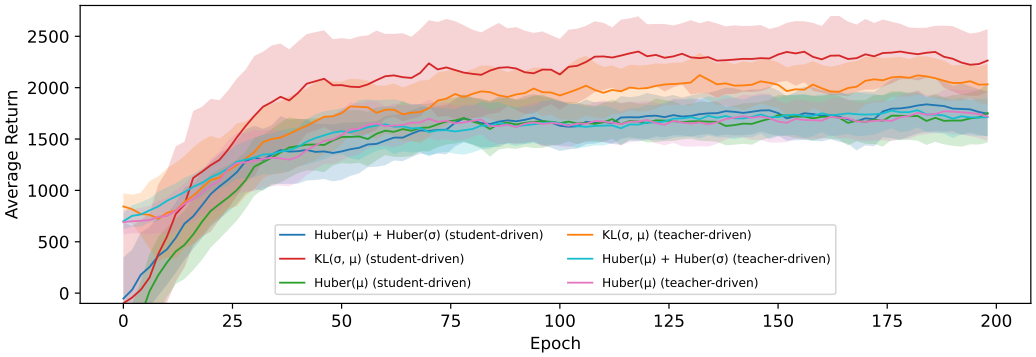


Figure 5.6: The exponential running mean of the average return during training for student and teacher-driven distillation in the Ant-v3 environment and an SAC teacher.

Note that the variance of the average return between epochs is much higher for this environment, so we show the exponential running mean with a window size of 10 in these plots to gain a clearer impression of the overall performance when using each of the loss functions.

5.5.2.2 Impact on Policy Entropy

We hypothesise that this difference in performance between the distillation losses is mainly due to the maintained accuracy of the policy stochasticity. This has particular importance in reaching a high degree of fidelity with teachers such as SAC, which are optimised to maximise an entropy-regularised return [47]. To verify this, we measure the entropy of the action distribution predicted by the students during testing, as can be seen in Figure 5.7.

This clearly shows that the relative order of the experiments is the same as for the average return, but in reverse. The student trained using our KL-divergence-based distillation loss indeed matches the entropy of the teacher the closest, and the more similar the entropy is to the teacher, the higher the average return that is obtained. However, it seems that the entropy is overestimated when using the other losses, resulting in more actions being taken that deviate too much from the teacher policy.

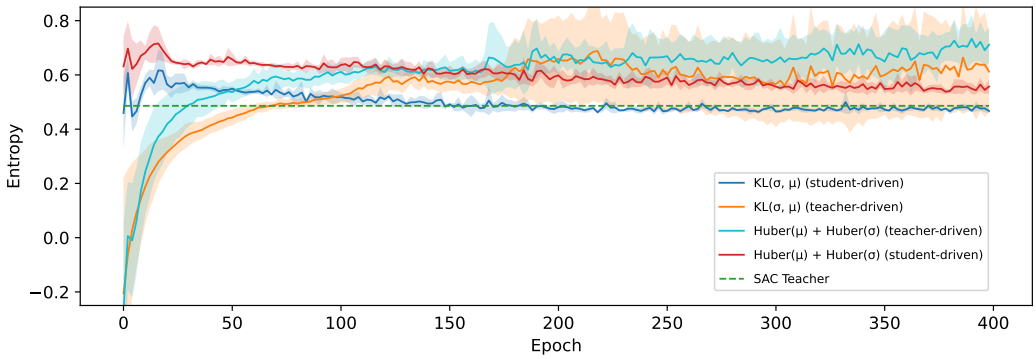


Figure 5.7: The average entropy measured for students trained using a loss that includes σ in the HalfCheetah-v3 environment and an SAC teacher.

So, the KL-divergence loss strikes a good balance between learning a stochastic policy, which our results confirm is optimal for this teacher, while staying closer to the teacher policy by not overestimating the entropy either. In the student-driven experiments, the entropy is initially higher, before gradually converging to the teacher entropy. This is beneficial for training, as the data collected during the first epochs will contain more exploratory behaviour and thus results in faster learning and reduces once the control policy is stabilising. These values are also a lot more stable and have much less inter-run variance compared to the teacher-driven experiments, which only seem to become worse over time.

5.5.3 Teacher Algorithm

In this section, we evaluate how generic our proposed methods can be applied to different teacher algorithms, focusing on the two most commonly used for continuous control tasks: SAC and PPO. The SAC algorithm tries to optimise a policy that obtains the highest return while staying as stochastic as possible. With PPO, on the other hand, the entropy generally decreases over time, as it converges on a more stable policy. This translates into the PPO

teacher achieving a higher average return when evaluated deterministically, while the SAC teacher performs better when actions are sampled stochastically, as shown earlier in Table 5.1. We have demonstrated in the previous sections that the KL-divergence loss is the most effective for distilling a stochastic policy, but it remains to be seen if this benefit remains for more deterministic teachers.

Therefore, we present the distillation results with a PPO teacher in Figure 5.8 in the HalfCheetah-v3 environment and in Figure 5.9 in the Ant-v3 environment. This shows virtually no difference in the used loss functions or the used control policy for action selection. Since the PPO teacher performs best when evaluated deterministically, there appears to be no benefit in learning the state-dependent value of σ if it is no longer used at evaluation time. By following a deterministic policy, the student is also less likely to end up in an unseen part of the environment, thereby reducing the difference between a student-driven and teacher-driven setting.

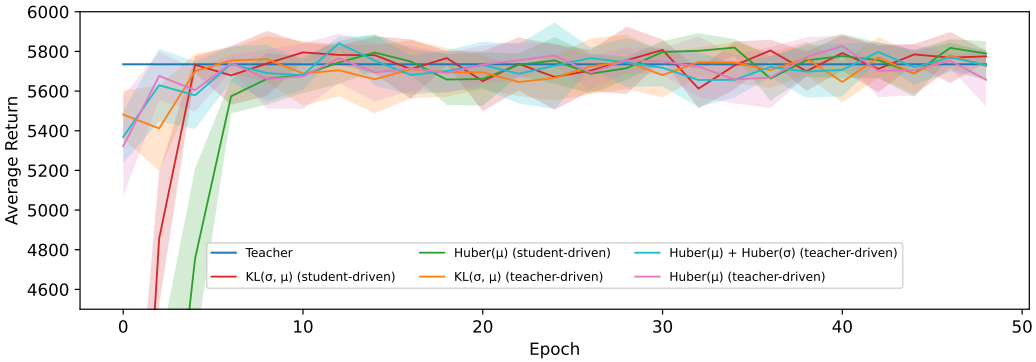


Figure 5.8: The average return during training for student and teacher-driven distillation in the HalfCheetah-v3 environment and a PPO teacher.

What is more notable about the PPO results, however, is that the students outperform their teacher in the Ant-v3 environment. In the context of PD for discrete action spaces, this phenomenon has also been observed and attributed to the regularisation effect of distillation [115]. These students (Figure 5.9) reach a peak average return after being trained for around 37 epochs, but this slowly starts to decline afterward, while their loss continues to improve. A lower loss generally indicates that the students behave more similarly to their teacher, which in this case is detrimental, resulting in regression.

This outcome relates to the work by Stanton et al. [129], who have shown in an SL context that KD does not typically work as commonly understood where the student learns to exactly match the teacher's behaviour. There is a large discrepancy in the predictive distributions of teachers and their final students, even if the student has the same capacity as the teacher and therefore should be able to match it precisely. During these experiments, the generalisation of our students first improves, but as training progresses, this shifts to improving their fidelity.

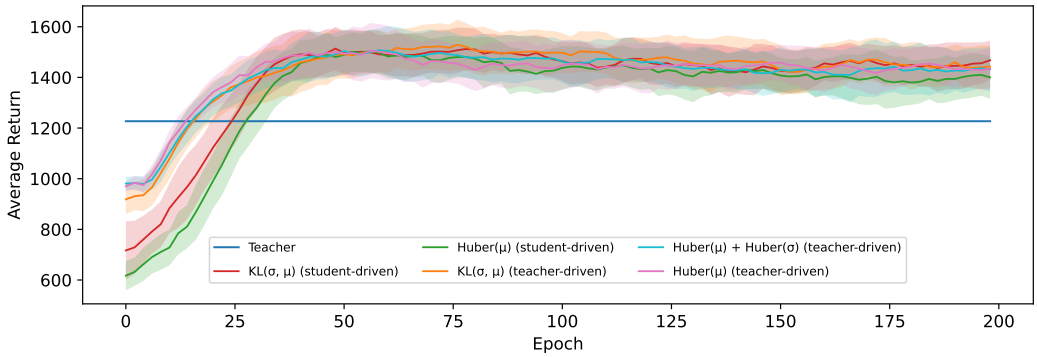


Figure 5.9: The exponential running mean of the average return during training for student and teacher-driven distillation in the Ant-v3 environment and a PPO teacher.

The students that were trained based on an SAC teacher performed slightly worse compared to their teacher in the HalfCheetah-v3 environment, and a more significant performance hit was observed in the Ant-v3 environment. This is likely due to the level of compression being significantly higher compared to the PPO distillation for this environment, as the student architecture is kept constant in this section to isolate the impact of the loss function choice.

5.5.4 Compression Level

We investigate the compression potential of our methods by repeating the experiments in Section 5.5.1 for a wide range of student network sizes, as listed in Table 5.3, focussing on the HalfCheetah-v3 environment. In Figure 5.10, our loss based on the KL-divergence (Equation (5.7)) was used, while Figure 5.11 shows the results when using the Huber-based loss for both μ and σ. Using our KL-based loss, we can reach a compression of 7.2× (student 6) before any noticeable performance hit occurs.

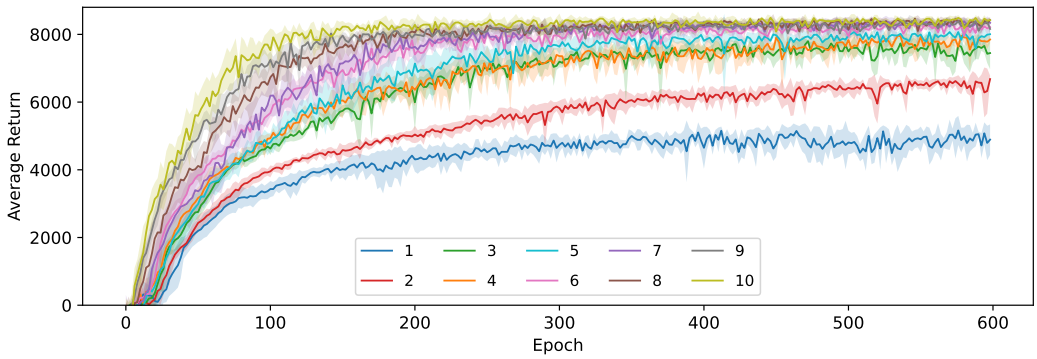


Figure 5.10: The average return for 10 student sizes, ranging from 492 to 139,276 parameters, during training using Equation (5.7) (KL) in the HalfCheetah-v3 environment and with a SAC teacher.

The average return stays relatively high at up to 36.2× compression (student 3), before dropping more significantly at even higher levels of compression. When going from student 3 to student 2, we also reduce the number of layers in the architecture from 3 to 2, which becomes insufficient to accurately model the policy for this task. The convergence rate noticeably decreases at each size step, with student 2 still improving even after 600 epochs.

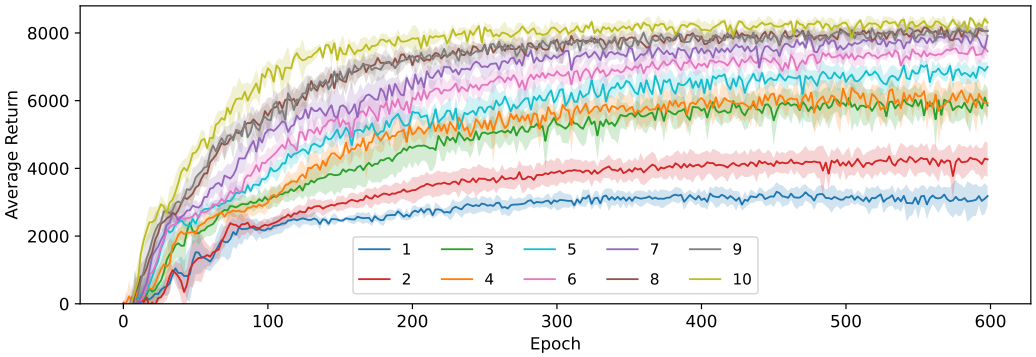


Figure 5.11: Average return during training using Equation (5.3) (Huber) in the HalfCheetah-v3 environment, for 10 students between 0.67% and 189% of their SAC teacher size.

The impact of the student size is much higher when using the Huber-based distillation loss. There is still a noticeable difference between the average return obtained by the largest (10) and second largest (9) student, even though this largest student is actually 2× larger than their teacher for this environment. This makes this loss particularly unsuited for distillation, as it requires more capacity than the original SAC teacher algorithm to reach the highest potential average return.

The largest student (10) here still performs slightly worse than the fourth-largest student (7) when trained based on the KL-divergence loss, but the performance gap does almost disappear for networks that approach the teacher in size. This means that our Huber-based distillation loss can still effectively transfer the teacher's knowledge to the student, but it requires considerably more capacity to learn two values (μ and σ) independently, making it infeasible for compression purposes. The convergence rate of these students is also slower, making it more computationally expensive at training time.

Therefore, we conclude that both proposed loss functions can be effective at distilling the stochastic continuous behaviour of the teacher, but the efficiency in terms of required network size and number of samples is significantly higher for our loss based on the KL-divergence, to the extent that the Huber-based loss becomes impractical for compression.

5.5.5 Runtime Performance

Finally, we analyse how this compression to the various student architectures (see Table 5.3) translates to benefits in terms of real-world performance. Note that we focus on

the inference performance of the final student models, as the training procedure is not intended to run on these low-power devices. This is measured on a range of low and high-power devices by sequentially passing a single observation 10,000 times through the network, which is then repeated 10 times using a random order of network sizes to ensure that any slowdown due to the prolonged experiment does not bias the results of a particular size. We then report the average number of steps per second, as shown in Table 5.4. Note that student 9 uses the same architecture as the SAC teacher, and student 10 is similar in size to the PPO teacher, so these are used as a baseline.

Table 5.4: Average steps per second for the student sizes and various low and high-power devices.

Device	1	2	3	4	5	6	7	8	9	10
AMD Ryzen 3900X (CPU)	11,284	11,246	10,051	9112	9993	9038	7631	8794	9369	8154
Nvidia RTX 2080 Ti (GPU)	3753	3726	3322	3022	3340	3050	2559	3019	3332	3049
Nvidia A100 (GPU)	3544	3552	3247	2516	3246	2998	2609	2992	3237	2994
Nvidia GTX 1080 Ti (GPU)	1804	1813	1612	1452	1612	1453	1213	1450	1605	1448
Nvidia Jetson TX2 (CPU)	946	947	840	755	825	736	603	680	679	486
Nvidia Jetson TX2 (GPU)	285	298	271	247	270	247	210	247	270	246
Raspberry Pi 3B (CPU)	702	700	620	562	603	540	452	478	428	335

An important observation is that, although the model performance in terms of average return scales with the number of parameters in the model, the story is more complicated when looking at the runtime performance. Notably, student 7 is the slowest network for most devices, even though it is only 13% as big as the largest network. It does, however, have the most network layers, being six compared to only four for student 10. This was chosen to keep a consistent increase of about 2× parameters when going from one size to the next while keeping the number of neurons per layer as a power of 2. A similar result can be seen for student 4, which also has one more layer than the surrounding ones. Having a deeper network limits the potential for parallelisation on devices with many computational units, such as GPUs or multi-core CPUs, while we did not notice a clear benefit of using more than three layers on the average return. On the lowest-power device we tested (Raspberry Pi), this difference due to the number of layers is less pronounced and the total network size becomes more important.

For high-power devices or ones designed for many parallel operations, the effective speed gain obtained by compressing these models is relatively minor, improving by only 9% worst case for a reduction to a mere 0.6% of the original size. In these cases, the overhead involved in simply running a model at all becomes the bottleneck, independent of the model itself up to a certain size. The highest improvement can therefore also be seen on the lowest-power device, the Raspberry Pi 3B, in which we can see a maximal runtime improvement of 64% compared to the SAC teacher or 109% compared to the PPO one. At this size, however, the model is no longer able to solve the task nearly as well as the teacher, so a comparison to student 3 with a runtime improvement of 44% and 85%, respectively, is more reasonable.

It is also worth noting that there is more to runtime performance, for which you might want to apply model compression than purely the achieved number of steps per second. Often, when running on embedded devices, there are additional constraints in terms

of memory or power consumption, or on devices with hardware acceleration for DNN inference, there can be a limit to the number of supported layers or parameters. In this setting, model compression can enable the use of more advanced models on devices that would otherwise not be capable of running them due to memory constraints. There, the model size in bytes becomes an important metric that impacts portability rather than performance. This can simply be derived for our models by taking the parameter count reported in Table 5.3 and multiplying it by 4. The popular Arduino Uno R3 microcontroller, for example, has only 32 kB of available ROM (Read-Only Memory) [6], which is only enough to store up to student 5, with a size of 24 kB.

Measuring the direct impact of PD on power consumption improvements is less straightforward, as this is more a property of the hardware than the individual model. You can force the device to periodically switch to a lower power state by artificially limiting the frame rate, but this difference is usually negligible compared to a switch in hardware class [110]. Instead, to optimise for this, we suggest searching for the hardware with the lowest power consumption that can still run the compressed model at an acceptable speed. For example, with a target of 600 steps per second, the Raspberry Pi 3B consumes around 4.2 W [110] and student 3 is a valid option. It will consume the same power when running the original model, but at half the inference speed. If the target is 800 steps per second, however, a jump to an Nvidia Jetson TX2 running at 15 W [100] becomes necessary.

We conclude this section by arguing the importance of carefully designing the architecture of the model with your target device in mind, performing benchmarks to evaluate the best option that meets your runtime requirements, and applying our proposed distillation method based on the KL-divergence to achieve the best model for your use case. Optionally, a trade-off can be made between the average return and steps per second to achieve the best result.

5.6 Conclusions

Deploying intelligent agents for continuous control tasks, such as drones, AMRs, or IoT devices, directly on low-power edge devices is a hard challenge, as their computational resources are limited, and the available battery power is scarce. This chapter addressed this challenge by proposing a novel approach for compressing such DRL agents by extending PD to support the distillation of stochastic teachers that operate on continuous action spaces, whereas existing work was limited to deterministic policies or discrete actions. Not only does this compression increase their applicability while reducing associated deployment costs, but processing the data locally eliminates the latency, reliability, and privacy issues that come with wireless communication to cloud-based solutions.

To this end, we proposed three new loss functions that define a distance between the distributions from which actions are sampled in teacher and student networks. In particular, we focused on maintaining the stochasticity of the teacher policy by transferring both the predicted mean action and state-dependent standard deviation. This was compared to a baseline method, where we only distil the mean action, resulting in a completely deterministic policy. We also investigated how this affects the collection of transitions on which our student is trained by evaluating our methods using both a student-driven and teacher-driven control policy. Finally, the compression potential of each method was

evaluated by comparing the average return obtained by students of ten different sizes, ranging from 0.6% to 189% of their teacher's size. We then showed how each of these compression levels translates into improvements in real-world run-time performance.

Our results demonstrate that especially our loss based on the KL-divergence between the univariate normal distributions defined by μ and σ is highly effective at transferring the action distribution from the teacher to the student. When distilling an SAC teacher, it outperformed our baseline, where only the mean action is distilled on average by 8% in the HalfCheetah-v3 environment and 34% in Ant-v3. This effect is especially noticeable in the student-driven setting, but we were also able to observe a significant increase in sample efficiency in the teacher-driven setup. When a less stochastic PPO teacher was used, all our proposed methods performed equally well, managing to maintain or even outperform their teacher while being significantly smaller. This also confirms that the regularisation effect of PD that was observed in the setting for discrete action spaces still holds for the continuous case.

In general, we recommend a student-driven distillation approach with our loss based on the KL-divergence between continuous actions as the most effective and stable compression method for future applied work. Through this method, DRL agents designed to solve continuous control tasks were able to be heavily compressed by up to 750% without a significant penalty to their effectiveness.

5.7 Addendum

In the main content of this chapter, we have presented several methods for the distillation of continuous actions, focusing on the transfer of the learned action distributions from a teacher to a student. This is the most direct way to transfer the teacher's behaviour, and it proved to be the most effective in our experiments with compression in mind as our main goal. In this addendum, we will briefly explore some alternative methods that could be used to transfer the teacher's knowledge to the student, based on other signals that are derived from the teacher model. These proved to be less effective for compression in our experiments, so were ultimately not included in any published work, but they could still be of interest for future research.

5.7.1 Distillation through Intrinsic Motivation

5.7.1.1 Value-Based Motivation

Inspired by our success in Chapter 3 and earlier work by Czarnecki et al [26], where the teacher's critic was used to provide a more informative loss signal in the context of discrete actions, we explored the possibility of using these state-value predictions to guide the student through an intrinsic reward signal for learning a policy that performs continuous actions. Instead of training the student through distillation with an auxiliary supervised task, we optimise the policy using regular SAC and environment interactions, but we compute a new reward signal based on the value-function of the teacher rather than obtaining it directly from the environment.

This idea stems from the notion that the teacher can provide more instantaneous feedback on the quality of the student's actions than the environment, which can often be noisy and sparse. While the environment will only provide a reward after a long sequence of correct actions in sparse reward settings, the critic of the teacher can already predict the expected return for a given state in advance at the start of that sequence.

So, if the student performs an action that is suboptimal according to the teacher at any point during that sequence, its value function will drop immediately, providing a more informative signal to the student. This helps with the credit assignment problem discussed in Section 2.2.3, as the student can more easily identify which actions were responsible for the drop in return and adjust its policy accordingly.

Concretely, we define the intrinsic reward r'_t as the difference between the predicted value of the current state $V(s_t)$ and the previous state $V(s_{t-1})$:

$$r'_t = V(s_t) - V(s_{t-1}) + r_t \tag{5.9}$$

Intuitively, this reward punishes the student for taking actions that will lead to a lower return, according to the teacher. We need to add the original reward r_t as an extra term to ensure that the student is not penalised for taking the final actions that actually yield the expected reward. Assuming the reward is positive, the value predicted by the teacher for

the current state $V(s_t)$ will be lower than for the previous state $V(s_{t-1})$, as the reward that was just obtained is no longer expected in the future.

5.7.1.2 Policy-Based Motivation

Similarly, we can use the teacher's policy to guide the student towards a more optimal behaviour. In this case, not by directly training the student to mimic the teacher's actions, but by rewarding the student for taking actions that are closer to those of the teacher. Whereas the previous method measures this similarity based on the effects of the actions on the environment, this method measures it based on the actions themselves. We define this intrinsic reward as the reciprocal of the squared difference between the teacher's action and the student's action:

$$r'_t = (a_{t-1} - \pi_{\text{teacher}}(s_{t-1}))^{-2} \tag{5.10}$$

This yields a reward close to one when the actions are similar and close to zero when they are not.

A potential problem with this approach is that this does not account for the length of episodes, so the reward will be higher for longer episodes, which might not always be desirable depending on the task. We evaluate this method in the HalfCheetah-v3 environment, to compare it to the distillation methods presented in the main content of this chapter, where longer episodes correspond to better performance, so this is not an issue. But for tasks that should be completed as quickly as possible, this method might not be as effective.

5.7.1.3 Results

We evaluate these methods in the HalfCheetah-v3 environment, using the same teacher algorithm as in the main content of this chapter. But unlike earlier experiments, we no longer use the publicly available pre-trained checkpoint for the teacher, so we can better compare the average return obtained during various parts of training when using the intrinsic rewards and the regular reward for the teacher. By training the teacher ourselves, we managed to obtain a higher average return than the pre-trained teacher, making these results not directly comparable to the ones presented earlier.

Figure 5.12 shows the average return obtained by models of three different sizes, with size L trained with the regular reward corresponding to the teacher. This architecture uses 64 neurons per layer, for both the actor and critic, and has three layers in total. Sizes M and S correspond to 32 and 16 neurons per layer, respectively, with the same number of layers.

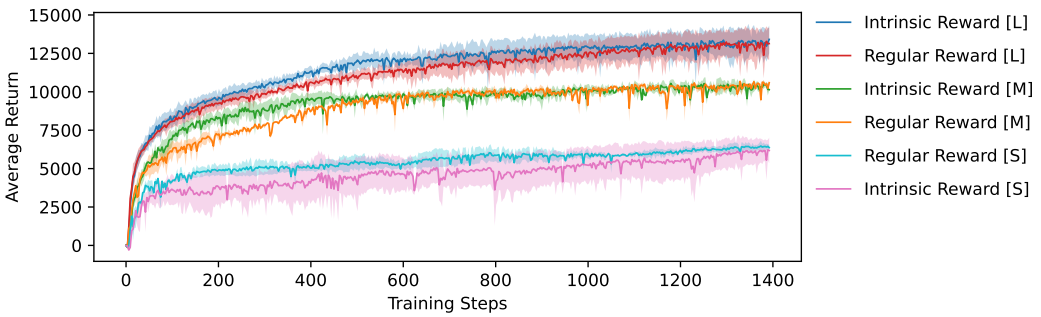


Figure 5.12: The average return in the HalfCheetah-v3 environment for students of 3 different sizes trained using intrinsic motivation based on the teacher's state-value predictions.

This figure confirms that this intrinsic reward signal based on the teacher's value function (Equation 5.9) is more informative than the regular reward signal, as the students with sizes L and M trained using this method are more sample efficient, reaching a higher average return for most of the training duration. However, as training progresses and the critics of the students themselves become more accurate, the difference between the intrinsic and regular reward disappears. At this point, the update direction provided by their own critic in Equation 2.23 becomes equally informative as the signal provided by the teacher's critic in the intrinsic reward.

Even more notable is that for the smallest student with size S, the intrinsic reward signal is actually detrimental to the training process, although it eventually also converges to the same average return as the regular reward signal. This approach might see more success in environments where the reward is more sparse, for which this method was mainly designed, but which is not truly the case for the HalfCheetah-v3 environment.

The most important observation is that this method is not as effective at transferring the teacher's behaviour into a smaller model as the distillation methods presented in the main content of this chapter. A significant performance hit is observed when going down to a smaller model size, without any real benefits compared to simply training the agents using the regular reward signal. It does, however, serve as another testament to the effectiveness of the distillation methods presented in the main content of this chapter.

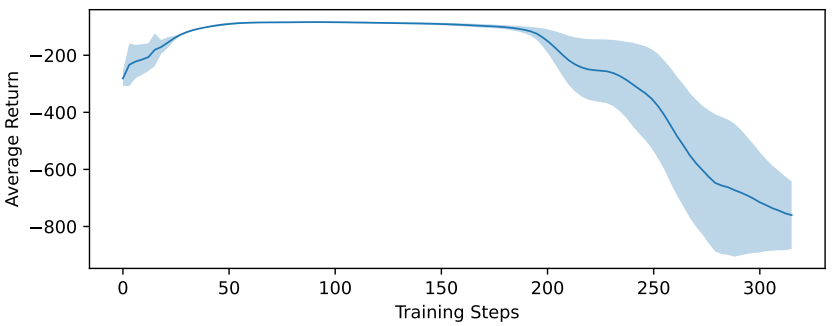


Figure 5.13: The average return in the HalfCheetah-v3 environment trained using intrinsic motivation based on the teacher's action predictions.

Figure 5.13 shows the average return obtained by the largest network size (L), trained with the intrinsic reward based on the difference between the teacher's actions and those of the student (Equation 5.10). The students do improve at the very start of training, but this improvement is short-lived, plateauing at a very low average return after a few steps and eventually regressing significantly. Although we believe the idea behind this method has merit, the way it was implemented here was clearly not effective.

Since the action space for this environment consists of 6 continuous actions, and the reward is only composed of a singular value, this signal is not informative enough to attribute the student's performance to the difference in the individual actions taken. In the distillation losses presented in the main content of this chapter, a similar difference in the action distributions is modelled as a differentiable loss function, which is more informative and can be used to guide individual actions of the student towards a more optimal behaviour.

We conclude that an intrinsic reward derived from existing teacher knowledge can be used to guide the student towards more optimal behaviour, but they are not as effective at compression as our PD methods to increase their efficiency, which is our main focus in this thesis.

Chapter 6

Temporal Distillation

The contributions presented in this chapter are based on the following publication:

> **Thomas Avé**, Matthias Hutsebaut-Buysse, and Kevin Mets. *"Temporal Distillation: Compressing a Policy in Space and Time."* Submitted to Springer Machine Learning.

6.1 Introduction

The field of DRL has seen incredible progress in recent years, with agents demonstrating superhuman performance on a wide range of challenging real-world tasks, such as the autonomous control of robots in warehouses [59], agriculture [21], and drone delivery [96]. However, the deployment of these RL agents in resource-constrained settings, such as IoT devices or embedded systems, remains a significant challenge [34]. Every inference made by the agent, selecting what action to take based on the current state of the environment, consumes energy. The energy-intensive nature of this decision-making process poses a barrier to the widespread adoption of RL in these environments. Often, these decisions need to be made in real-time to quickly react in dynamic environments, placing additional requirements on the hardware resources.

Researchers have proposed policy compression methods to address this issue by reducing the size of the DNN that represents the policy [115]. We refer to this as spatial compression, as it reduces the storage space and memory required to store the policy. This directly increases the application potential of the policy on devices with limited read-only and random access memory. Furthermore, reducing the number of parameters in the policy network leads to a reduction in the number of computations required to perform inference, speeding up the decision-making process significantly. However, our runtime performance benchmarks in the previous chapters, which focused on spatial compression alone, showed that the increase in inference speed is typically not proportional to the reduction in the number of parameters.

Smaller networks with narrower layers tend to have a limited potential for parallelisation, leading to a less efficient use of the available hardware resources. This is because the number of operations that can be performed in parallel for computing the activations of the current layer is directly proportional to the number of neurons in the previous layer. The fixed overhead associated with loading input data and initialising the inference engine

also becomes the bottleneck at smaller network sizes, limiting the potential speed-up that can be achieved.

We therefore propose utilising time as an additional dimension for compression, which does scale proportionally with every increase in (temporal) compression level. Every decision made by the agent requires a certain amount of energy, so reducing the number of decisions that the agent needs to make to complete its task directly reduces the energy consumption of the agent.

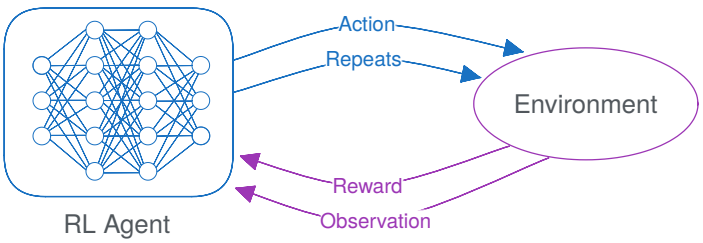


Figure 6.1: Our proposed method can be considered an extension to the traditional reinforcement learning loop.

We introduce a form of temporal abstraction to the traditional RL loop, depicted in Figure 6.1. This allows the agent to not only predict what action to perform next but also *when* it should decide about the next action to take. In the meantime, the agent can remain idle while the previous action is repeated, conserving energy. The time between decisions can also serve as a buffer for the agent to gather additional energy required for computing the next action, further increasing portability to ultra-low-power devices.

Traditional decision-making in RL is inherently reactive, making a decision at every time step after the current observation has been updated [17]. This approach can be inefficient, as it may require the agent to make frequent decisions even in regions of the environment where the optimal action remains constant for an extended period. By learning *when* it is necessary to execute new decisions, agents can adopt a more proactive strategy, leading to improved energy efficiency. Predicting how many steps to commit can be seen as an auxiliary planning task, requiring a deeper understanding of the environment to operate effectively. We demonstrated earlier in Chapter 3 how including such auxiliary tasks can even benefit policy effectiveness and generalisation.

The concept of frame-skipping, where the same action is repeated for a fixed number frames, is already standard practice in environments with high frame rates, such as the Atari benchmarks [94]. In such settings, the rate of change between consecutive observations is low, so no significant information or opportunities are lost by skipping frames. This has been shown to greatly improve the learning efficiency of agents, as it allows the agent to observe the consequences of its actions more quickly and learn from a wider distribution of states for the same number of observations [66]. However, using a static skip-size reduces the granularity in which the agents can operate, potentially leading to suboptimal performance in environments where the optimal action can change more frequently. By learning a temporal policy, the agent is more flexible in deciding when granularity is required, and when it can afford to use fewer observations and remain idle.

In continuous control tasks, agents can naturally exhibit a form of temporal abstraction. For instance, when an agent decides to move forward a certain distance, this single continuous action inherently represents a prolonged behaviour [131]. In contrast, agents that operate using discrete actions must repeatedly decide to move forward one step at a time. Decreasing the time and therefore rate of change between steps allows for more precise control, but this also increases the number of decisions and consequently computations that are necessary. Our work allows discrete agents to predict a variable-length sequence of the same action, effectively compressing the policy in the time domain. This approach is more flexible than simply using larger continuous actions to represent a longer stretch of time, as it can also extend to tasks that inherently require discrete decisions, such as the common Atari benchmarks [13], turning on or off appliances in a smart power grid [16], or replica scaling in a cloud environment [126].

By combining spatial and temporal compression in a single compression method, we are able to create agents that are not only more portable and faster in their decision-making process but also require fewer (energy expensive) decisions to complete the same tasks, as shown in Figure 6.2. We base our method on the concept of PD [115], where a small student network is trained to emulate the learned behaviour of a larger teacher network. Traditionally, this is done by recording trajectories from the teacher and training the student to predict the same actions when given the same observations as input. We continue this approach for learning which action to perform at each step, but include an additional loss term for predicting how many consecutive steps the teacher performed this action.

By learning from examples of when a change in action is required to follow an optimal policy in practice, we obtain a safe empirical lower bound on the number of steps that can be skipped without impacting the performance of the agent. It also allows for both the temporal and spatial compression of any existing policy behaviour, without requiring any modifications to the original training process. Experiments on two embodied Minigrid [24] environments show that our method is able to reach up to a 1347% decrease in the average time needed to predict an action compared to 403% through spatial compression alone, while maintaining a similar average return as the original teacher.

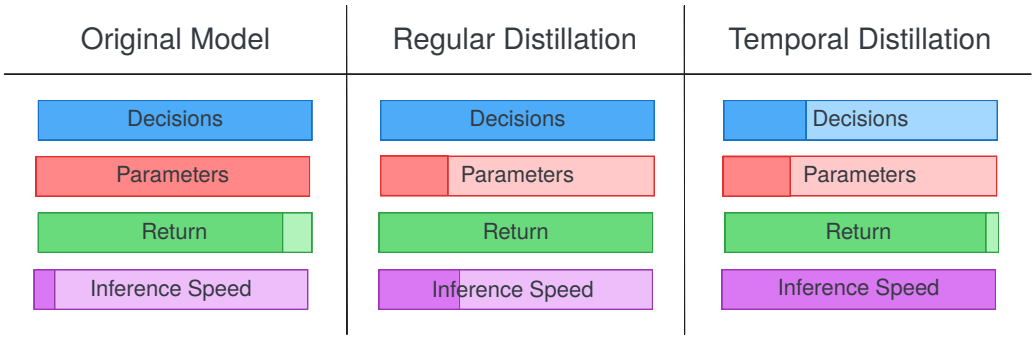


Figure 6.2: A relative comparison using the FourRooms environment and an ESP32 microcontroller of metrics obtained by the original teacher model, a student compressed by 400% using only regular (spatial) PD, and the same student trained using our temporal distillation method.

6.2 Related Work

Our temporal distillation method relates to existing work in two main ways: as an extension to PD, and by learning when to repeat actions in an RL environment.

6.2.1 Extending Policy Distillation

Several works have extended the concept of PD by extracting knowledge from the teacher network in addition to the action predictions, although mostly in the context of multitask distillation instead of compression. Czarnecki et al. [26] formalised these methods in a general framework and provided mathematical and empirical analysis for each. One of the most common extensions is student driven distillation, where the student network is used as a control policy to choose which actions are taken while interacting with the environment to gather the observations and the teacher's network outputs for the replay memory. This was shown empirically to be more sample efficient than the original teacher driven PD [115] method, but the purest definition of student driven distillation does not have the same convergence guarantees. Our approach is inherently teacher driven, since the additional action repeat knowledge needs to be extracted from trajectories sampled from the teacher policy.

The second part of their contributions is focused on how to use the critic's value function in actor-critic networks for bootstrapping. In Chapter 3, we also used the critic during distillation, but as an auxiliary task to enhance the student's understanding of the environment. Although we are also using an actor-critic teacher in this chapter, we opted not to include our earlier extensions to maintain the broad applicability of our core method to other teacher algorithms, such as the DQN for which the PD method was originally designed. Especially the auxiliary critic loss should be trivial to integrate as part of our method for future work and has the potential to improve our student's performance even further. The work in this chapter is, to the best of our knowledge, the first to extract temporal 'dark' knowledge from a teacher network to enhance PD. It also differs by not just emulating the existing teacher's behaviour more efficiently, as our students gain an entirely new ability that the teacher does not possess.

6.2.2 Learning to Repeat Actions

DAR (Dynamic Action Repetition) was first introduced by Lakshminarayanan et al. [78] by extending the DQN, A2C and A3C (Asynchronous Advantage Actor-Critic) algorithms with multiple network output heads for each action, corresponding to different action repetition rates. In essence, these heads extend the action space of the agent by the number of original actions times the number of additional repetition rates. These (action, repeats) combinations can then be learned through a regular RL objective based on the (discounted) return. By enabling this form of temporal abstraction, their agents were able to outperform their vanilla counterparts on all tested environments (Seaquest, Space Invaders, Alien and Enduro).

We note that this approach has the potential to drastically increase the action space when the number of potential repeats is high, leading to a considerable increase in the difficulty

of learning an optimal policy. In our architecture, we only have one additional continuous value that can represent any number of repeats (when rounding to the nearest integer), avoiding this potential optimisation issue that occurs exactly when learning action repeats would yield the highest benefit to run-time performance.

The same authors later address this issue themselves in their FiGAR (Fine Grained Action Repetition) framework [123], in which the policy network contains a repeat actor head that is independent of the action actor head, similar to our architecture. How many times an action should be performed consecutively is then determined by sampling from the J-dimensional probability vector modelled by the repeat actor head, resulting in $\{1, \ldots, J\}$ steps. Training is done by propagating the computed gradients with a shared critic through both action heads simultaneously. This keeps the size of the action space more reasonable, even for larger repeat values, but it still places an upper limit on the number of steps, which cannot be learned dynamically.

The authors of TempoRL [17] further note that decoupling the repeat value from the action, but still training them using a shared objective based on maximising the return, causes the FiGAR agent to learn a repetition length that works well on average for all actions. They solve this by conditioning the repeat policy on the action predicted by the behaviour policy, resulting in a learned mapping from a (state, action) pair to a probability vector over the possible repetition values. With this approach, they managed to outperform both FiGAR and DAR, which they found to overly rely on coarse control, leading to fewer decisions but also worse performance in the benchmark environments.

Although our repeat head is also decoupled from the action head, we solve this issue differently by training the repeat head through a separate distillation loss that is likewise decoupled from the action loss. Since their repeat policy is dependent on the chosen action, the two policies need to be computed sequentially during inference, slowing down the decision-making process. In contrast, our method allows the agent to predict both the action and the number of repeats in parallel from a shared representation, which can be computed in a single forward pass.

Our work primarily differs from all these existing approaches by learning to reduce the number of decisions that the agent needs to make for compression, rather than merely enabling the agent to commit to an action for multiple steps to increase sample efficiency. In that setting, the agent is not encouraged to repeat as many actions as possible, it only has the option to do so. Theoretically, fine control through minimal action repeats is inherently more advantageous when optimising purely for a high return, as the policy can still determine to continue with the same action if it remains optimal, but it can also more quickly react to unexpected changes in the environment. We also optimise for a low decision rate, however, by learning the actual number of repeats the agent should perform when following the optimal reference policy provided by the teacher. Our agents therefore learn by example, rather than through trial-and-error.

A second key difference is that we model the repeats as a continuous variable, instead of a probability distribution over a fixed number of repeats. This avoids drastically increasing the action space for environments that require granular control, with a possibility for high repeat values with a range that dynamically adapts based on the environment, rather than being fixed by the architecture. Additionally, the loss to train this value can be more informative, where being close to the actual repeat value is better than being off by a larger margin. When using discrete repeat options, each possible value is treated as independent

of the others. This is especially important for our distillation-based setup, since the teacher will not always perform the optimal number of repeats, so the student should learn to generalise from these suboptimal examples.

Finally, our proposed method is deeply integrated with PD, allowing us to learn repeat values for temporal efficiency and reduce the number of parameters at the same time. This also enables us to optimise any existing policy, while the other methods discussed above require training from scratch with a modified architecture.

6.3 Methodology

As our students learn to repeat actions from examples of identical actions that are used consecutively in the teacher's trajectories, we first need to define how these values are computed and stored. Since transitions are sampled in a random order from the replay memory during training, we need to store the repeat values explicitly for every entry.

The most intuitive way to represent this information is by storing one entry in the replay memory for each action repetition sequence, consisting of the observation, the starting action (logits) of this sequence, and the number of times it was repeated. A value of 0 repetitions indicates that the action was only performed once, as each action prediction should always result in at least one transition.

This approach corresponds to how the student policy is expected to operate after training, with a single decision that results in a sequence of transitions (steps in the environment). In effect, this reduces the length of the trajectory to the number of action repetition sequences, by merging all sequential transitions with identical actions into a single entry. We therefore refer to this approach as the reduced trajectory variant of our method.

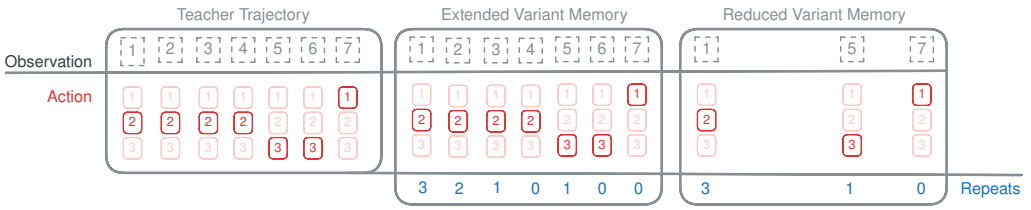


Figure 6.3: An illustration of the reduced and extended trajectory variants of the replay memory.

In the second (extended) trajectory variant, we store each atomic transition in the replay memory, with the remaining length of the sequence as the repeat value. Figure 6.3 illustrates the difference between these two variants. We compute these repeat values once per episode, by iterating over the trajectory in reverse order and storing the number of times the current action has remained the same. This provides the student with more examples of when a new action will be required, but it also takes up space in the replay memory and capacity of the student with redundant transitions that should not be encountered in practice.

If the student policy does not match the teacher perfectly, these intermediate logits and repetition lengths could prove to provide additional insight and improve fidelity to the teacher's policy. Our experiments (see Section 6.5.2) show that this additional inter-sequence knowledge has a greater benefit than simply having more best-case examples to learn from.

To learn these repetition lengths, we introduce a new head to the student network that predicts the number of times the action should be repeated as a continuous value, as illustrated in Figure 6.4. In all our architectures, we use a shared two-layer MLP to extract features from the flattened observation, followed by a single parallel layer for the action head and repeat head.

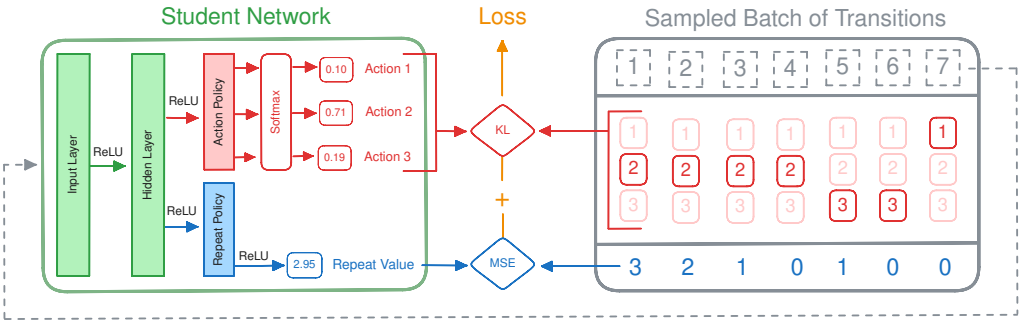


Figure 6.4: An illustration of the student network architecture and the distillation loss for the actions and additional repeat head.

These heads are trained using separate optimisation objectives, which are then combined to form the modified distillation loss in Equation 6.1.

$$\mathcal{L}_{KL}(D, \theta_S) = \sum_{i=1}^{|D|} \left(\text{softmax}(\frac{q_i^T}{\tau}) \ln(\frac{\text{softmax}(\frac{q_i^T}{\tau})}{\text{softmax}(q_i^S)}) + \left(\frac{r_i^S + |r_i^S|}{2} - \frac{r_i^T}{\lambda} \right)^2 \right) \tag{6.1}$$

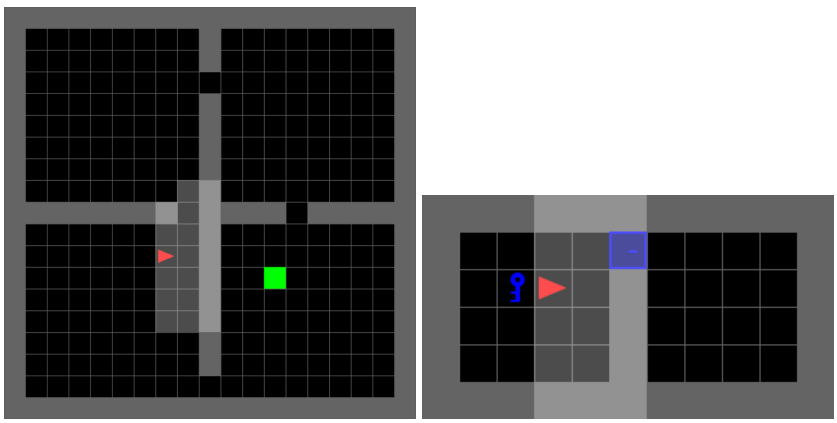
Here, r_i represents the repeat value of transition $i \in D$, with r_i^T the repeat value computed as described above, and r_i^S the student's prediction. The value of λ serves two purposes: it normalises the repeat values to make them easier to learn, and it scales the repeat loss to balance it with the action loss. The ReLU function on r_i^S is applied during training and inference to ensure that the student only learns positive repeat values, as an action cannot be repeated a negative number of times. We then use the MSE error loss to train the repeat head and combine this with the KL-divergence loss for the action head to form the total loss. During inference, the predicted r^S values need to be multiplied by λ and rounded to the nearest integer to obtain the actual number of repeats.

6.4 Experimental Setup

6.4.1 Evaluation Environments

We selected the benchmark environments for the validation of our methods based on several important criteria. First, the observations should be informative enough to allow the agent to learn some form of planning. Control should be granular, with a potential for learning a sequence of repeated actions, but also the requirement for precise control when a new action is needed. This invalidates many of the Atari benchmarks, as the agent can often afford to skip frames without losing valuable information and without significant opportunity costs for missing the correct exact timing required for an action. Of course, our methods would likely work well under these conditions, but it would not adequately demonstrate the potential for temporal compression under more challenging circumstances.

For that same reason, the agent should be actively engaged and making decisive progress towards a goal during a repetition sequence, rather than remaining idle until certain conditions are met. Ideally, the agent should therefore not only learn when to act, but when to start performing a different action. Finally, the task should be complex enough to not trivially be solvable by any tiny policy network without the need for compression.



(a) FourRooms **(b)** Unlock

Figure 6.5: A visualisation of the two Minigrid environments used in our experiments.

We found that embodied object-navigation tasks are a good fit for these requirements, as they require the agent to actively explore the environment to find one or more goal objects, while also needing precise control to accurately navigate through doors. The Minigrid environment suite [24] provides a wide range of popular benchmark tasks that meet these criteria, with the FourRooms and Unlock environments being selected for our experiments.

A visualisation of these environments is shown in Figure 6.5. Here, the positions of the agent, goal, door, key and the gaps in the walls are all randomised at the start of each episode to ensure that the agent needs to learn a dynamic policy instead of a fixed sequence of actions. The agent can move around in a grid by taking one step forward, turning left or

right, or interacting with an object. It does so based on observations that are represented by a one-dimensional list of the types, colours and states of objects in the agent's field of view (using the FlatObs wrapper), with the agent's position being implicit. The agent's vision is blocked by walls and the partially observable state is depicted in light grey, with the current position and orientation shown as a red triangle.

This results in an observation list of size 2835, composed of 49 positions in the grid, each with an object type, colour and state (open, closed, or locked). The remaining values (the large majority with size 2688) are used to one-hot encode the mission space, which is always "reach the goal" (FourRooms) or "unlock the door" (Unlock) in our experiments. The reward is sparse, with only a single reward given at the end of successful episodes, and zero otherwise. This reward takes the form of $(1 - 0.9 * (steps \,/\, max\_steps))$, with *steps* the number of steps required to complete the task and *max_steps* the maximum number of steps allowed in an episode before it is terminated (288 for Unlock and 100 for FourRooms).

In FourRooms, the goal is to find and reach the green square using the shortest path possible while navigating the maze of rooms through gaps in the walls that are placed in random locations. This environment was chosen to showcase how our method can learn to efficiently navigate through an environment with a high potential for action repetition. Overestimating the repeat value can be costly, however, as the agent will need to backtrack to the correct path if it overshoots the entry to the next room. Backtracking requires 3 additional decisions, which is a significant amount given that this task can often be completed in around 5 decisions. Since our agents have no memory and the state is only partially observable, this could even cause the agent to no longer see the room entry and therefore lose track of its objective to move through it entirely.

The task in the Unlock environment is more complex, as the agent first needs to find and interact with a key to unlock a door before it can achieve the goal of opening it. It is also smaller, with only a single room where the agent, key, and door are placed in random locations. This environment therefore has fewer opportunities for action repetition, and it requires more precise control to complete the task. By including these two environments, we aim to compare the benefits of our method across settings with different potential for action repetition and requirements for precision.

6.4.2 Model Architectures

To evaluate the spatial compression potential of our method, we applied it to students of a wide range of network sizes, from 204% down to 6% of the number of parameters in the teacher, as shown in Table 6.1. Adding the auxiliary repeat-prediction task could benefit the internal representation and therefore the generalisation capabilities of the student network, but it also increases the task complexity. We therefore need to validate that by introducing temporal compression, we do not lose the ability to compress the policy in the spatial domain. This is done by comparing the average return obtained in the environment of each temporal student to a baseline student trained using regular PD.

All our architectures consist of three fully connected layers, with a varying number of neurons per layer, and connected by ReLU activation functions. Students trained using our temporal distillation method have an additional repeat head in parallel with the action head, as shown in Figure 6.4, while our baseline students only have the action

head. The storage size increase of the additional repeat head is negligible, with only a single additional weight per neuron in the previous layer and a single bias term. We later determine whether this expanded architecture has any runtime implications, as the additional head could still potentially slightly slow down the decision-making process. Since both the observation and action spaces are identical for both environments, the same architectures are used for both tasks.

Table 6.1: Architectures used for students with varying levels of compression.

Parameters Temporal	Parameters Regular	Neurons per Layer
11,404	11,399	4
22,832	22,823	8
45,784	45,767	16
92,072	92,072	32
186,184	186,119	64
380,552	380,423	128

To increase the reproducibility of our findings, we use the publicly available PPO agents from the SB3 library [108] as our teacher networks. The exact model checkpoints can be found on the HuggingFace page for FourRooms [56] and Unlock [57]. These teacher agents obtain an average return of 0.480 and 0.939 respectively in our testing, as shown in Table 6.2. This serves as a second baseline and target reference for the performance that our students should aim to emulate. The teacher network has the same architecture as our second-largest student without the repeat head, with 186,119 parameters in total. Note that this does not include the critic head, which uses another 185,729 parameters, but this is not required for inference.

We therefore perform one experiment where only temporal compression is performed, to isolate its potential benefits without the restrictions imposed by spatial compression. Additionally, the largest student even has double the number of neurons per layer compared to the teacher, to evaluate the impact of learning the auxiliary repeat policy while ensuring this does not take up valuable network capacity that would otherwise be used for the action policy.

Since the environments are only partially observable, it would likely be beneficial to include a recurrent layer in the network to capture temporal dependencies, but we favoured using publicly available agents as a baseline. Our method is compatible with RNN student architectures, however, requiring only a minor modification in Algorithm 1 to sample entire trajectories instead of random individual transitions. We later show how this modification can be made in Chapter 8.

Table 6.2: Average and standard deviation of the return obtained by the teacher networks.

Model	FourRooms	Unlock
PPO Teacher	0.480 ± 0.380	0.939 ± 0.189

6.4.3 Training Procedure

Each experiment consists of 100 training epochs, where one epoch is completed after the student has been updated using all 100,000 transitions stored in the replay memory D. Following each epoch, we evaluate the student over 200 episodes to calculate the average return, and refresh the replay memory by replacing the oldest 10,000 transitions in D with new teacher-environment interactions. For agents trained through temporal distillation, we measure the average return based on all steps in the environment, including the repeated actions.

During training, transitions are randomly sampled from D in mini-batches of size 64, after which we perform a single step of the RMSprop optimizer with an LR of 5×10^{-4}. After performing hyperparameter tuning, we found that these values worked well for all student sizes and environments. The value of the repeat scale (λ) in Equation 6.1 had a larger impact on the training process and required tuning for each student size and environment, as shown in table 6.3.

Table 6.3: Best value of λ found for each student size and environment.

Neurons per Layer	**4**	**8**	**16**	**32**	**64**	**128**
Unlock	25	25	25	50	50	50
FourRooms	15	15	15	25	50	50

These were found to work best out of the set {1, 2, 5, 10, 15, 25, 50}, with some preliminary experiments on a more extensive set of values. We provide further analysis of the impact of this hyperparameter in Section 6.5.3.

6.4.4 Runtime Performance Evaluation

With the main goal of our method being to improve the efficiency of the student policy, it is essential to evaluate the runtime performance of the trained students. We do this by feeding a single observation through the network a fixed number of times, and repeating this process 10 times with a random order of network sizes to prevent any potential slowdown over time from skewing the results for a specific size.

For our ESP32 benchmarks, we first exported our models to the ONNX (Open Neural Network Exchange) [102] format, and then used the Onnx2C tool [65] to convert them to C code that can be compiled and deployed on the microcontroller. The other benchmarks are performed using the OnnxRuntime library [89] in C++, which allows for multithreading and GPU acceleration to speed up the inference process and make more efficient use of available hardware resources. Our Onnx2C code used 1000 iterations to measure the average network inference time, while the more powerful devices that used the OnnxRuntime library ran 100,000 iterations to ensure a stable average.

The effective step rate after applying temporal distillation was then calculated by dividing the reduction in the average number of decisions required to complete an episode by the average inference time for this network. For example, if the teacher needs an average

of 40 steps to complete an episode, while the student only needs 5 decisions, with each decision taking 2 seconds to compute, the effective step rate would be 4 steps per second. In practice, a higher effective step rate allows a device to enter a low-power state more often to maintain the same performance, or enables a more efficient device to achieve a level of performance that would otherwise require something more powerful.

6.5 Results and Discussion

To verify the effectiveness of our method, we conducted experiments on two embodied Minigrid environments, namely FourRooms and Unlock, with a wide range of student network sizes. These students are compared to both the teacher policy and to the traditional PD method, in terms of the average return obtained in the environment and the number of decisions needed to complete the task. We begin, however, by analysing the potential difference in behaviour between the teacher and student networks.

6.5.1 Behavioural Analysis

Figure 6.6 illustrates a trajectory taken in the FourRooms environment, with each arrow corresponding to an action. The teacher has learned to stick closely to the central walls for increased visibility of where the doors between rooms are located. Since the number of steps in a trajectory is based on the Manhattan distance, this behaviour is optimal in terms of steps, but not in terms of decisions. These actions are converted to (action, repeats) pairs and stored in the replay memory, which is then used to train the student network.

This results in 7 entries when using the reduced trajectory variant, while the extended variant yields 26 (observation, action, repeats) transition pairs. When the same environment configuration was presented to our largest student model, trained using the extended trajectory variant, it was able to generalise to a policy that only required 3 decisions to reach the goal.

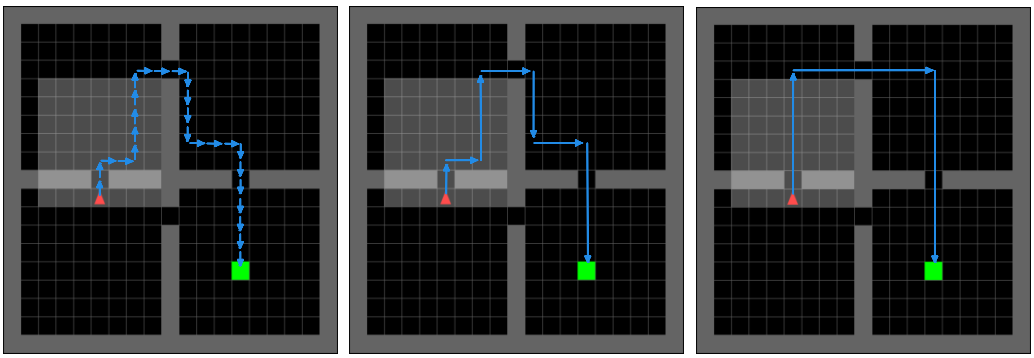


Figure 6.6: A visualisation of the same FourRooms task solved by the teacher and student, with each action represented by an arrow.

We attribute this to the regularisation effect of KD, which was shown to have the potential to improve the generalisation capabilities of the student network beyond the teacher's behaviour, enabling it to obtain higher average returns than the training data [115]. In our case, the student learned from many examples when a new action was required and applied this more broadly. This real example was chosen to highlight the potential of our method, but it remains a best-case scenario, as the student was lucky that the goal happened to be located along the pre-committed path.

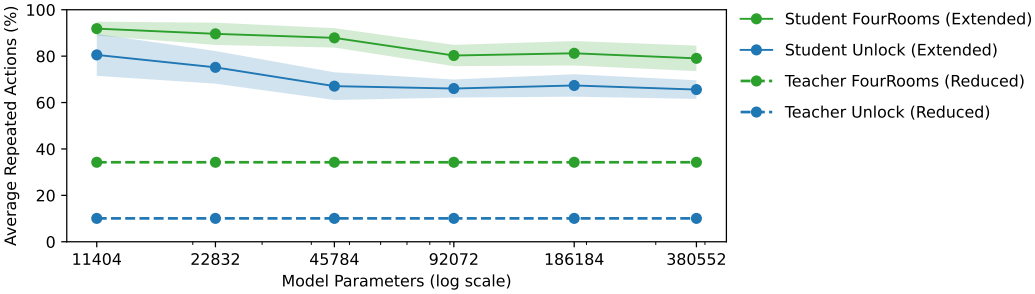


Figure 6.7: Percentage of action repeats performed by student networks of different sizes.

Still, Figure 6.7 shows that our students in general learn to perform considerably more repeats than are present in the teacher trajectories, for both the FourRooms and Unlock environments. This figure indicates that the smaller student architectures are more likely to perform repeats than the larger ones, but those additional repeated actions are not necessarily accurate.

When considering the total energy required to complete a task, there is an additional trade-off between the computational energy used to make decisions and the mechanical energy required to perform actions in the environment. For instance, taking a direct path to the goal, as the crow flies, might be optimal in terms of the total number of steps, but this could require more frequent and precise decision-making to navigate around obstacles. A more conservative approach that uses more reliable decision checkpoints might require more steps overall, but could significantly reduce the number of decisions needed. On the other hand, making inaccurate repeat decisions that result in more mechanical actions could lead to a larger overall energy cost, even if the number of complex decisions is reduced.

6.5.2 Agent Performance

When looking at the average return obtained in the FourRooms environment, shown in Figure 6.8, we indeed observe that task performance starts to drop more noticeably for the smallest three students. These are the same students that showed an increase in the number of repeats, suggesting they no longer have the capacity to model the repeat policy as accurately. Since the agents receive a penalty for each step taken in the environment and not for each decision made, this results in a lower average return if those steps are less accurate.

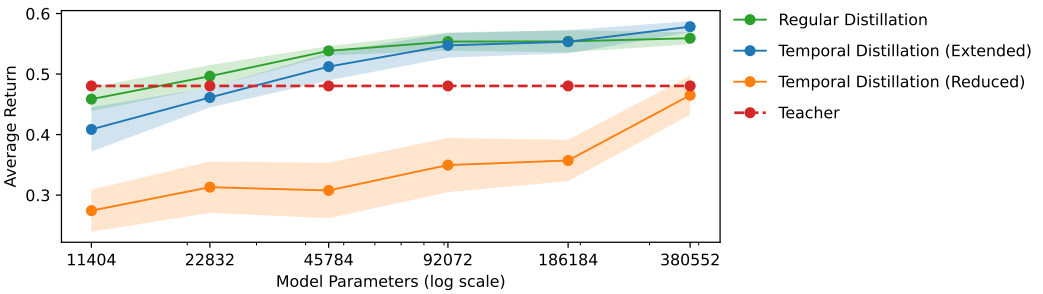


Figure 6.8: Average return for different student sizes on the FourRooms environment.

Learning this additional auxiliary repeat policy does increase the learning complexity compared to regular distillation, resulting in an average return that is lower on average for most student sizes in these experiments. It also provides a more informative signal for the student to learn from, however, increasing its understanding of the environment and actually outperforming regular distillation for the largest student and the extended variant. The reduced variant performs significantly worse in terms of average return, as it fails to properly learn the repeat policy due to the lack of examples within the repeat sequences. It is only able to reach close to the teacher's performance when the student is large enough to overfit to the training data, but this is not a scalable solution for deployment on resource-constrained devices.

Even more noticeable is that most other students are able to obtain a considerably higher average return than the teacher, despite the reduced number of decisions made. During our investigations into this behaviour, we found that the students were less likely to become stuck while walking into a wall, where the observations and actions would no longer change until the end of the episode, due to the lack of memory. We hypothesise that the loss for this transition would quickly be minimised, causing the student to focus on the more optimal examples in the replay memory. It also, once again, confirms the potential of PD to improve the generalisation of the student through the introduced regularisation effect.

The results from the Unlock environment, shown in Figure 6.9, tell a very similar story, with a few notable exceptions. Here, the reduced variant performs slightly better than the extended variant for all except the two largest students. That is, in terms of average return, but not in terms of the number of decisions needed to complete the task. In fact, none of the final students in these experiments performed any repeats at all. Since this environment is significantly smaller, with only a 11x6 grid instead of the 19x19 grid in the FourRooms environment, there is less potential for the agent to perform long sequences of the same action. Therefore, these students had significantly more examples where a new action was required at every step, leading to better overall performance, but also not enough examples of repetition sequences to learn any sensible value greater than zero.

The teacher for this environment is notably stronger than the one for the FourRooms environment, which is reflected in the higher average return obtained by the teacher and the students. This also means that the students have less opportunity to surpass the teacher, as the teacher's policy is already close to optimal. There still remains ample opportunity to learn a more efficient policy in terms of decisions, however, as shown in Figure 6.10.

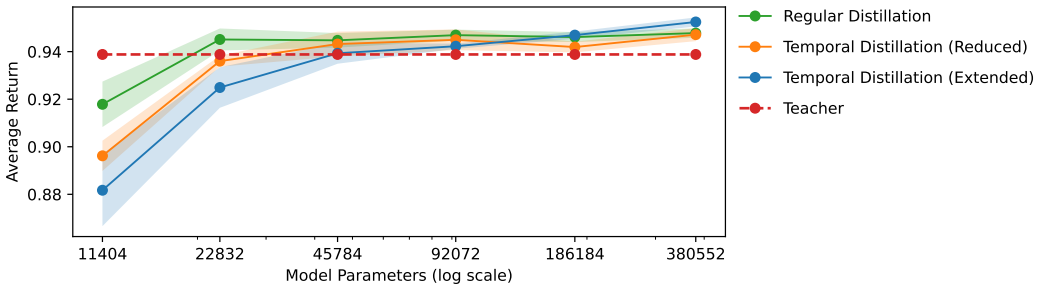


Figure 6.9: Average return for different student sizes on the Unlock environment.

We previously showed in Figure 6.7 that the students are able to perform more repeats than present in the teacher's trajectories, but this does not necessarily mean that those repeats actually lead to fewer overall decisions if those repeated actions were not effective. Making more repeats while still obtaining a high average return can already be beneficial in terms of energy management by allowing the agent to remain idle for longer periods to recharge. But the real benefit of our method is that we can reduce the number of decisions needed to complete the task, directly resulting in less energy consumption.

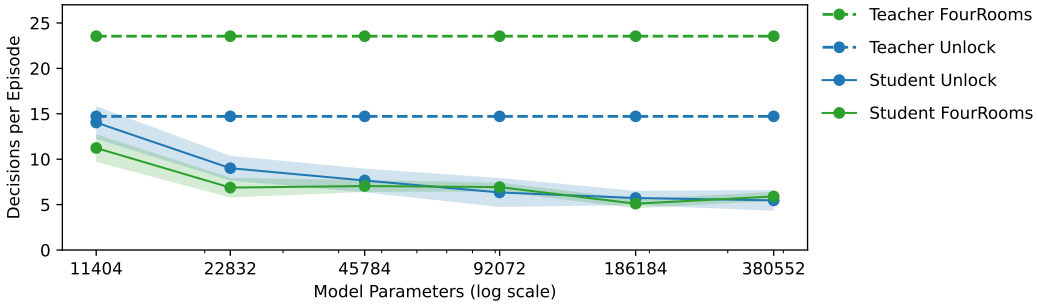


Figure 6.10: Average number of decisions needed to successfully solve the environments.

Figure 6.10 shows how our method can outperform the teacher in terms of the obtained average return, while needing up to 269% fewer decisions on Unlock and 462% on FourRooms to achieve this. It also confirms our intuition that, although the smaller models predict more repeats, they need more decisions to complete the task. This is especially apparent for the Unlock environment, where the smallest student effectively performs eight times as many steps as the teacher on average, which is also reflected in the lower average return. Note that this figure only includes statistics for successful episodes. Unsuccessful episodes are terminated after a fixed number of steps, depending on the size of the grid in the environment. This means that unsuccessful episodes would inherently require fewer decisions, skewing the results in favour of agents that simply learn a high repeat value.

6.5.3 Impact of Repeat Scale

As mentioned in the experimental setup in Section 6.4.3, tuning the value of λ in Equation 6.1 had a significant impact on the training process. This is especially apparent for the smaller student sizes, which have more difficulty accurately modelling the repeat policy

in general, as shown in Figure 6.11 for the Unlock environment. Choosing a correct value for λ is therefore crucial to ensure proper performance at solving the task efficiently, with a relatively large impact on the average return for suboptimal choices. Both a value that is too low ($\lambda = 15$) or too high ($\lambda = 50$) can lead to a similar drop in performance; it needs to be a correct match with the repeats performed by the teacher policy.

Results for the FourRooms environment are similar, but with a different optimal value for λ, as was shown earlier in Table 6.3. Since this difference in average return is due to how the repeat policy is modelled, we also measured the percentage of repeated actions performed by the student for each value of λ, depicted in Figure 6.12. It seems that, in general, the student is more likely to perform more repeats when λ is set to a higher value.

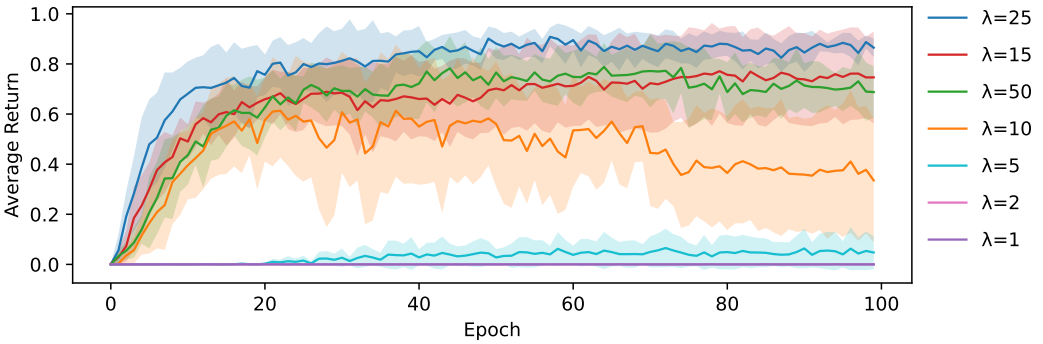


Figure 6.11: Average return on the Unlock environment for the smallest student and various values for the repeat scale λ in Equation 6.1.

This means that, counterintuitively, the students that perform the fewest repeats on average also have the lowest average return. Without proper scaling, these students are not able to predict the high repeat values that are present in the teacher's trajectories, leading to repeating actions less often on average. It also results in a more noisy repeat policy, however, causing inaccurate movements and an inability to complete the task.

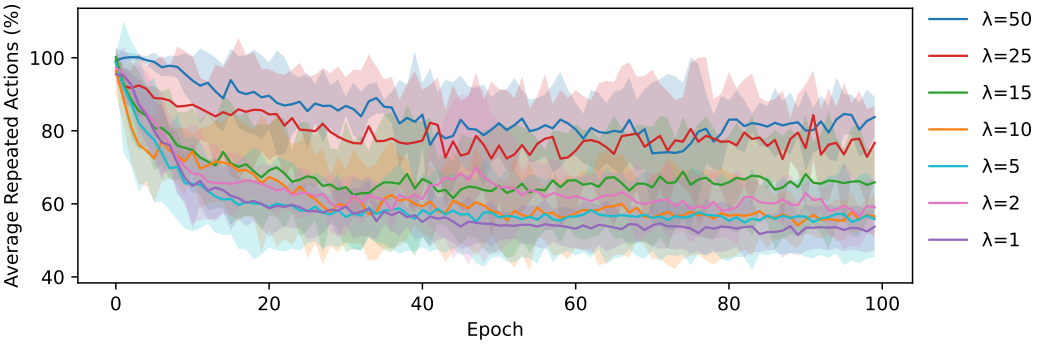


Figure 6.12: Average percentage of repeated actions on the Unlock environment for the smallest student and various values for the repeat scale λ in Equation 6.1.

The value of λ through which the best average return was obtained resulted in the second-highest percentage of repeated actions. An accurate repeat policy therefore also performs

a high average number of repeats, but does this in a more controlled manner, leading to a better average return. Students with more capacity, such as our largest student in Figure 6.13, are less sensitive to the choice of λ. Only the lowest values ($\lambda = 1$ & $\lambda = 2$) result in a considerable difference during training, but even those runs eventually converge on a similar, but still slightly lower, average return. It likewise has a more stable repeat policy for all values of λ.

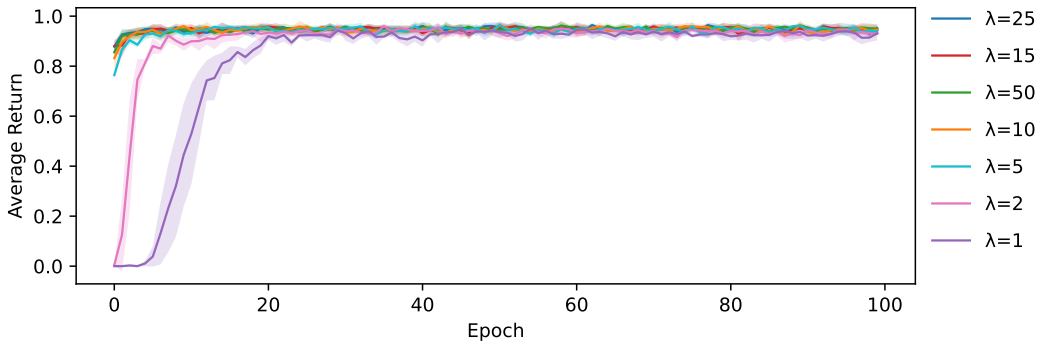


Figure 6.13: Average return on the Unlock environment for the largest student and various values for the repeat scale λ in Equation 6.1.

6.5.4 Runtime Performance

Finally, we evaluate how the combined temporal and spatial compression of our method affects the effective steps per second that can be performed on different classes of hardware. For spatial compression, the best runtime performance is achieved by the smallest student, as it has the fewest parameters and therefore the fewest computations to perform. Through temporal compression, the largest student is able to complete the task in the fewest number of decisions, causing a higher effective step rate. When combining these two factors, the best runtime performance becomes a trade-off depending on the hardware. It also depends on the degree of task effectiveness one is willing to sacrifice for a higher effective step rate.

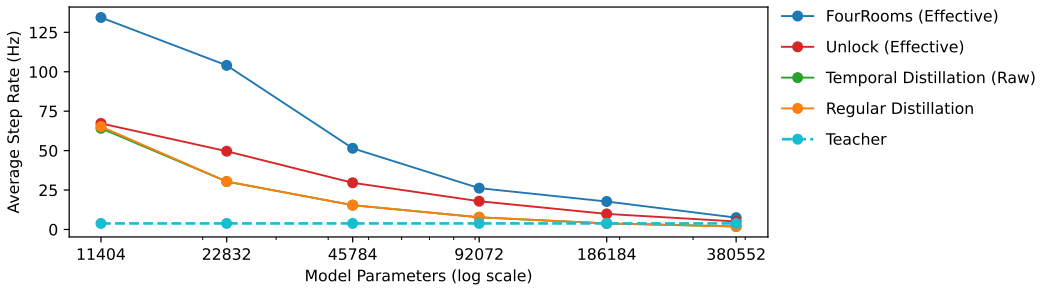


Figure 6.14: Raw and effective steps per second for different student sizes on an ESP32 microcontroller.

Figure 6.14 shows the effective step rate that can be achieved by all models on the popular ESP32 microcontroller. This metric is based on the relative improvement in the number of decisions needed to complete the task, compared to the teacher, and the raw inference

rate of the model. The network architectures for both environments are identical, so non-repeating step rates are the same for both. This figure also includes the raw inference rate of our temporal distillation method, so we can observe the runtime performance impact of the additional computations required to predict the repeat value.

On the ESP32, this difference is negligible, so applying our method is always beneficial in terms of energy efficiency. As the runtime performance on this microcontroller scales well with the number of parameters, the spatial compression becomes the most important consideration. The smallest student achieves the highest effective step rate, even in the Unlock environment, where this model needs the same number of decisions as the teacher to successfully complete the task.

When the goal is to achieve at least the same average return as the teacher, however, the third-smallest student with temporal distillation is the most efficient choice for this environment, beating the second-smallest student trained through regular distillation and being 7.75× faster than its teacher. In the FourRooms environment, our temporal distillation method scales even better than the highest spatial compression. With the same goal in mind, the same student size is again the most efficient choice, being 13.47× faster than its teacher.

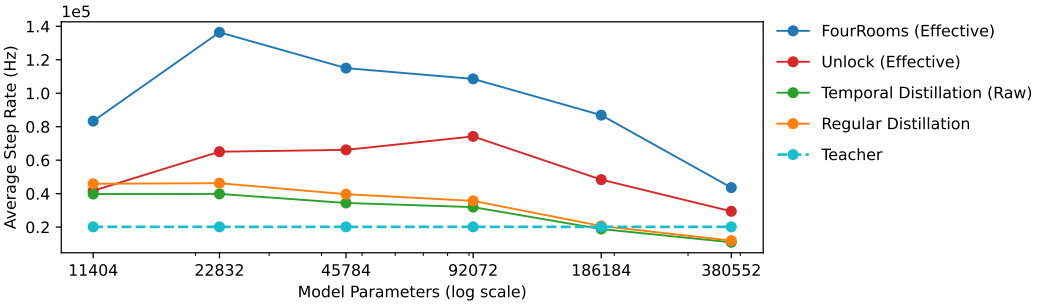


Figure 6.15: Raw and effective steps per second for different student sizes on a smartphone with a Qualcomm Snapdragon 845 processor.

Going from a microcontroller (~100mW) to a smartphone processor (~4W) in Figure 6.15 shows a significant increase in raw inference rate, by more than 5000× for the teacher. This is partially due to it simply being a more powerful processor, but also because it is able to utilise up to eight threads to parallelise the computations. Since the largest networks benefit the most from parallelisation, spatial compression becomes less advantageous for this hardware. This shifts the balance in the most efficient architecture towards the middle-sized students, with the second-smallest temporal student becoming the most efficient choice for FourRooms and the third-largest for Unlock, with a relative speed-up of 6.77× and 3.68× respectively.

These larger students are able to complete the task in fewer decisions, outweighing the increased computations required for each decision, but a balance between spatial and temporal compression is the most efficient choice here. Since the largest student architecture is twice the size of its teacher, it has the worst overall runtime performance, even though it is able to complete the task in the fewest number of decisions for the Unlock environment. We do note that the additional computations required for temporal distillation have a more noticeable impact on the runtime performance on this hardware, but this is far

outweighed by its effective benefits.

Finally, we evaluate the performance benefits of our method on a powerful data centre GPU, in this case an NVIDIA Tesla V100 (~300W), in Figure 6.16. Due to its massive parallelisation capabilities, but slow per-thread performance, the benefits of spatial compression become virtually non-existent on this hardware. Through temporal compression, however, we are still able to achieve a significant increase in effective step rate, with the second-largest student being 4× faster on FourRooms and the largest being 2.3× faster on Unlock. Counterintuitively, the most efficient model for the Unlock environment is, therefore, larger than the original network.

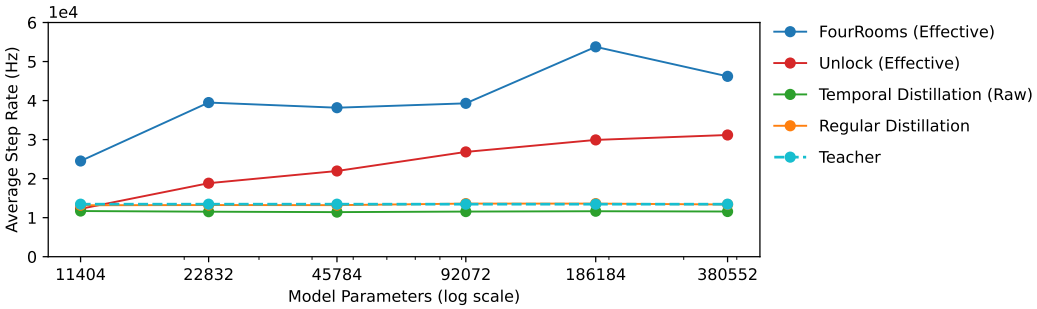


Figure 6.16: Raw and effective steps per second for different student sizes on an NVIDIA Tesla V100 data centre GPU.

We conclude that temporal distillation is effective at maintaining and even exceeding the task performance of the original policy, while drastically reducing the cost of performing each decision and the number of decisions needed to complete this task. Determining the most efficient student size is a trade-off between spatial and temporal compression, with the most efficient choice depending on the hardware that the agent will be deployed on. Temporal distillation consistently exceeded the effective step rate of regular distillation at a given average return target, regardless of whether agents are deployed on tiny microcontrollers or on high-end data centre GPUs optimised for deep-learning.

6.5.5 Safety Considerations

One limitation to the chosen environments is that they do not take safety into account, as the agent is not penalised for performing actions that could be considered dangerous or harmful, such as walking into a wall. This is true for both the teacher and the students, with only a slight incentive to avoid doing so by yielding a lower return if the agent takes more steps to complete the task. Safety is an important consideration in many real-world applications, however, and it is crucial that the compressed agent does not compromise on this aspect of the policy in exchange for higher computational efficiency.

Even if the teacher would be heavily penalised for unsafe actions through negative rewards, the student may not adequately learn to avoid these actions when learning only by example. The distillation process does not use this reward signal, and the replay memory only contains positive examples of good behaviour in this teacher-driven setting. On the other hand, the student does still learn from the teacher's entire action distribution, including implicit 'dark' knowledge, so it should avoid dangerous actions that the teacher assigns

lower probabilities to. It remains to be seen whether a student trained through temporal distillation can extrapolate this implicit safety information from the actor to the repeat head, however, or if an additional mechanism would be required to make this signal more explicit.

As shown in Figures 6.10 and 6.6, the students generally exhibit more repetitive behaviour than their teacher, which is good for energy efficiency, but it could also lead to more dangerous actions. This is especially true for the smaller students, for which the repeat policy is less precise and are therefore more likely to make costly mistakes. By introducing this form of temporal abstraction, the agent inherently has less information to base its decisions on, and can react to unexpected changes in the environment less quickly. Our approach is still safer than simply using less granular steps or a fixed frame-skip, however, as the agent can learn which parts of the environment are safe to repeat actions in, and which areas likely require more precise control.

A potential solution would be to introduce some basic safety checks that trigger a forced decision update, such as when a change in the observation becomes statistically significant, or when a learned threshold value is exceeded. Examples of this include when a sudden change in the wind speed causes an agent for drone navigation to drift off course, or when a bird suddenly flies in front of the camera. Such mechanisms should arguably be in place for safety-critical applications regardless of the used policy, as the black-box nature of DNNs and thus DRL models makes it impossible to make strict guarantees about their behaviour.

One could also argue that our evaluation methods in this thesis are not sufficient to determine the safety of the learned policy. By focussing on the average return obtained in the environment, we only measure how well the agent is generally able to complete the task, but not how it behaves in edge cases. If catastrophic failures are rare, they might not be reflected significantly in the average return, but are still of high concern in practice. For some environments, the worst-case scenario might be more important than the average case, and since these occur less frequently, their recovery might not be prevalent enough in the training data. While we believe that safety is an important consideration, we did not focus on these types of applications in this thesis, so we leave the further exploration of these topics for future work.

6.6 Conclusions

Existing compression methods for DRL agents focus on reducing the number or precision of parameters in the policy network, thereby reducing the computational cost of each inference. However, they do not consider the number of inferences required to complete a task, other than simply by training a more effective policy that requires fewer steps to reach the goal.

In this chapter, we proposed a novel method that is able to decompose the policy into sequences of repeated actions, which can be learned by a student network through PD. Each such sequence requires only a single inference, thereby reducing the number of decisions needed to complete the task. We referred to this as temporal compression, as it reduces the number points in time when an active decision is required, rather than the

spatial size reduction of the policy network.

To learn how many times an action should be repeated, we introduced an additional head to the student network that predicts this number as a continuous value. This allows the repeat range to dynamically adapt based on the environment, rather than being fixed by the architecture if a discrete set of repeat options were used. For heavy compression, we did find that tuning the scale of this learned repeat value was crucial to ensure proper performance, while larger models were less sensitive to this hyperparameter.

Learning a continuous value enables a more informative distillation loss, where being closer to the actual repeat value is preferable to being farther off. In our distillation setup, the teacher does not always perform the optimal number of repeats, but the students were able to learn from these suboptimal examples and generalise to a more efficient policy that requires even fewer decisions.

An additional contributing factor to this success was the introduction of our extended trajectory variant of the replay memory. Instead of only learning from the best-case examples based on the first transition in a repetition sequence, we also stored the logits and computed the repeat values for all subsequences with the same end transition. We showed how this additional inter-sequence knowledge yielded a significantly more accurate repeat policy, where the original (reduced) variant was either unable to learn any repeat value in the Unlock environment, or only through overfitting to the teacher policy in FourRooms.

We evaluated our method on two embodied object-navigation tasks in the Minigrid environment suite, FourRooms and Unlock, with a wide range of student network sizes. These experiments showed that our method is able to outperform the baseline teacher policy in terms of the average return obtained in the environment for all but the two smallest students. Compared to regular (spatial-only) PD, our method was able to achieve a higher average return for the largest student in both environments, but performed comparably or slightly worse when actual compression was applied. This is likely due to the additional complexity introduced by the repeat task, which has the potential to improve the internal representation of the policy, but also requires more capacity to learn effectively.

For any target average return that the student should aim to achieve, however, we showed that our temporal distillation method consistently outperformed regular distillation in terms of the effective step rate that can be achieved on different classes of hardware. Due to the trade-off between spatial and temporal compression, the most efficient student size depends on the hardware that the agent will be deployed on.

Overall, we showed how temporal compression can be as or even more effective than spatial compression in terms of energy efficiency. Our method is able to simultaneously combine both: to reduce the computational cost of each inference and the number of inferences required to complete a task. This enables the deployment of more complex DRL models on resource-constrained devices, while still maintaining the same level of task performance.

Chapter 7

Online Adaptation through Task-Relevant Pruning

The contributions presented in this chapter are based on the following publication:

> **Thomas Avé**, Matthias Hutsebaut-Buysse, Wei Wei and Kevin Mets. *"Online Adaptation of Compressed Models by Pre-Training and Task-Relevant Pruning"* In The 32nd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN), Bruges, Belgium, 9-11 October 2024.

7.1 Introduction

DL has surged in popularity over the past decade, with swarm intelligence as one of the emerging application domains. There, swarms of low-power sensor devices must coordinate to complete complex tasks. Online tuning by individual nodes is increasingly important here, as more intelligent systems are deployed in dynamic, real-world environments with varying local deployment conditions. However, limited on-device resources constrain the size and complexity of models that can be deployed and further optimised.

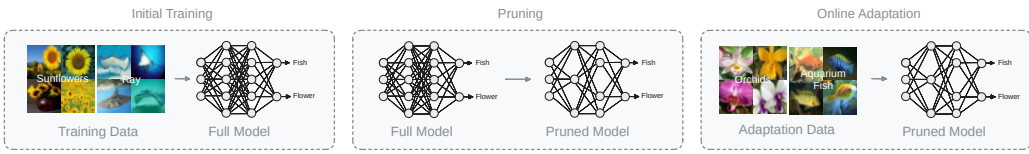


Figure 7.1: An illustration of the studied pruning and online adaptation setting. The model is trained on the main task (left) and then pruned to remove redundant features (middle). The pruned model is then adapted to new data (right).

Model compression addresses this by reducing the size of DNNs while preserving their predictive power. Pruning stands out by removing redundant parameters from a fully trained network, resulting in a more compact architecture. However, these methods typically do not account for online learning scenarios where edge devices need to continuously adapt to new data, illustrated in Figure 7.1. They instead focus on creating the smallest

static model, removing any redundancy not necessary for the current task, including features beneficial for new tasks.

We propose a novel method that extends the generalisation capabilities of compressed models to enable online adaptation without sacrificing compression rates. Generalisation refers to learning general patterns not specific to the training data, but which apply to new data as well, making it crucial for online adaptation to a change in data distribution. Prior research suggests that learning additional knowledge improves a model's generalisation [68], by pre-training on a larger dataset with different classes or more diverse samples, before fine-tuning on the main task. But this typically requires larger models that are harder to prune, as more connections are actively used to encode the additional knowledge [147]. Methods such as IMP (Iterative Magnitude Pruning) use weight magnitudes to determine their importance, making it difficult to differentiate between task-relevant and additional knowledge used to improve generalisation. This is counterproductive for compressing and adapting models for edge devices, as it limits the compression potential.

To address this issue, we leverage LRP, a technique originally developed for interpretability[10]. LRP enables us to score and identify how relevant neurons are for the main task. By pruning and retaining only features with high task relevance, we preserve the model's generalisation and adaptability when fine-tuning to new subclasses and continual learning on new classes, as illustrated in Figure 7.3.

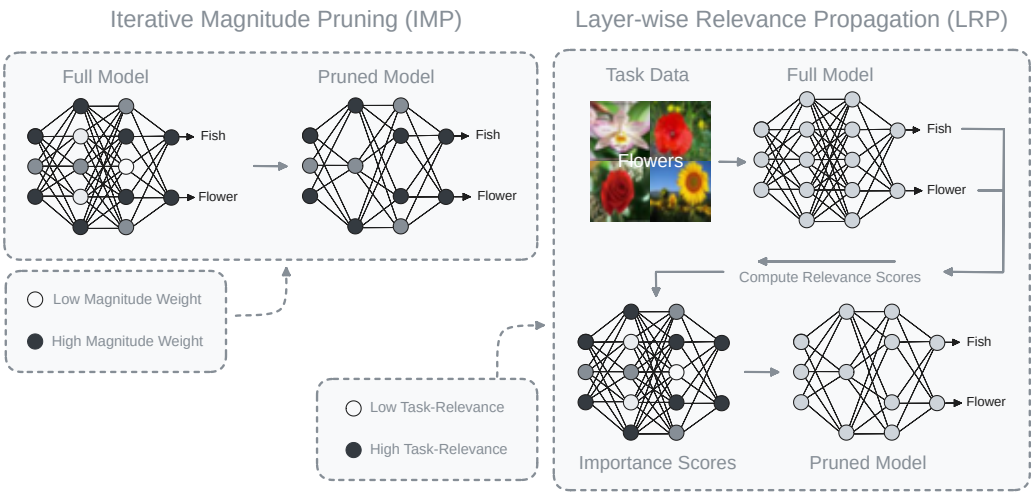


Figure 7.2: An illustration of the difference between IMP and LRP pruning. IMP only considers the weight magnitudes, while LRP computes the relevance scores of each neuron for the given input data.

In this chapter, we investigate how well a model compressed through structured pruning can still be adapted to new data. In contrast to the other contributions presented in this thesis, we focus on the problem of online adaptation in an SL setting, instead of developing methods that exploit properties that are unique to DRL. This enables us to first evaluate the effectiveness of our proposed approach in a more general context, broadening its applicability and simplifying the experimental setup to focus on the core idea.

That does not mean the proposed approach was not developed with DRL in mind, however.

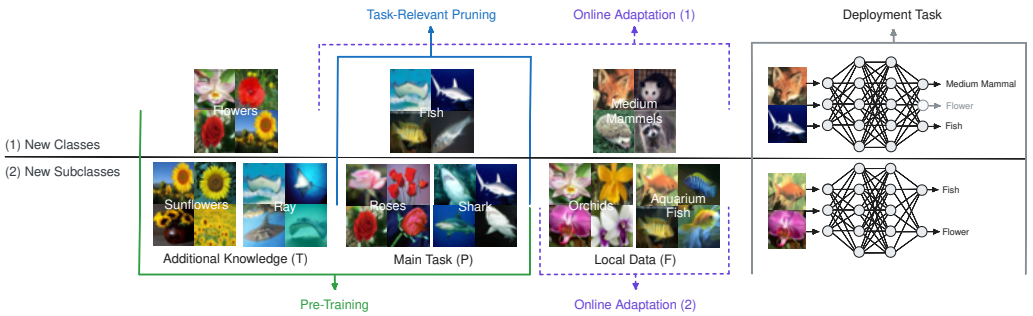


Figure 7.3: Illustration of the two different online adaptation scenarios on CIFAR-100.

Pruning, contrary to PD, does not inherently modify the output distribution of the model, which is essential for online fine-tuning after compression using regular DRL methods, such as DQN and PPO. Furthermore, the targets while training DRL models are non-stationary as they usually depend on the current version of a constantly changing policy, making adaptability even more crucial. In section 9.2, we discuss some potential future work for extending this approach to DRL.

Here, we evaluate if our pre-training and task-relevant pruning method can effectively improve the generalisation capabilities during online adaptation on the CIFAR-10, CIFAR-100 and DomainNet datasets, comparing the results with both IMP and LRP-based pruning methods. Our experiments confirm this approach can be used to train a more accurate compressed model while achieving better generalisation in both adaptation scenarios.

7.2 Related Work

Fine-tuning of pruned models is very common, either as an intermediary step of progressive pruning or to regain accuracy after pruning [19]. This is typically done using the original dataset, however, to recover existing behaviour, not to adapt to new data after compression. Gordon et al. [41] did evaluate applying unstructured IMP on the BERT (Bidirectional Encoder Representations from Transformers) model before and after fine-tuning for downstream tasks in the context of transfer learning. They concluded that low pruning levels (30-40%) had no detriment for downstream tasks and pruning once after pre-training was as effective as separately after fine-tuning to each task. Instead, we focus on structured pruning, where entire neurons or filters are removed from the network, making it actually more efficient for deployment on general-purpose hardware, at the potential cost of less expressiveness for the same size. Srivastava et al. [128] also combine pruning with adaptation to new tasks but in the context of multitask continual learning where the network size is kept constant, whereas we use pruning for compression.

7.3 Methodology

An illustration of our proposed approach is shown in Figure 7.4. Compared to a baseline trained only on the main task (bottom), the model trained with extra knowledge (top) learns more general features that are retained after pruning. Due to the more limited training data, the model on the bottom is prone to memorise more task-specific features, which are less relevant during online adaptation.

By employing knowledge-based pruning using only the relevant task data, we can still effectively identify and remove the redundant features, while preserving the generalisation capabilities of the model. This is not possible with magnitude-based pruning that only identifies neurons which don't encode any features, rather than those which are not relevant to this task. Through task-relevant knowledge-based pruning, we can still obtain high compression rates, even though more neurons were actively required during training.

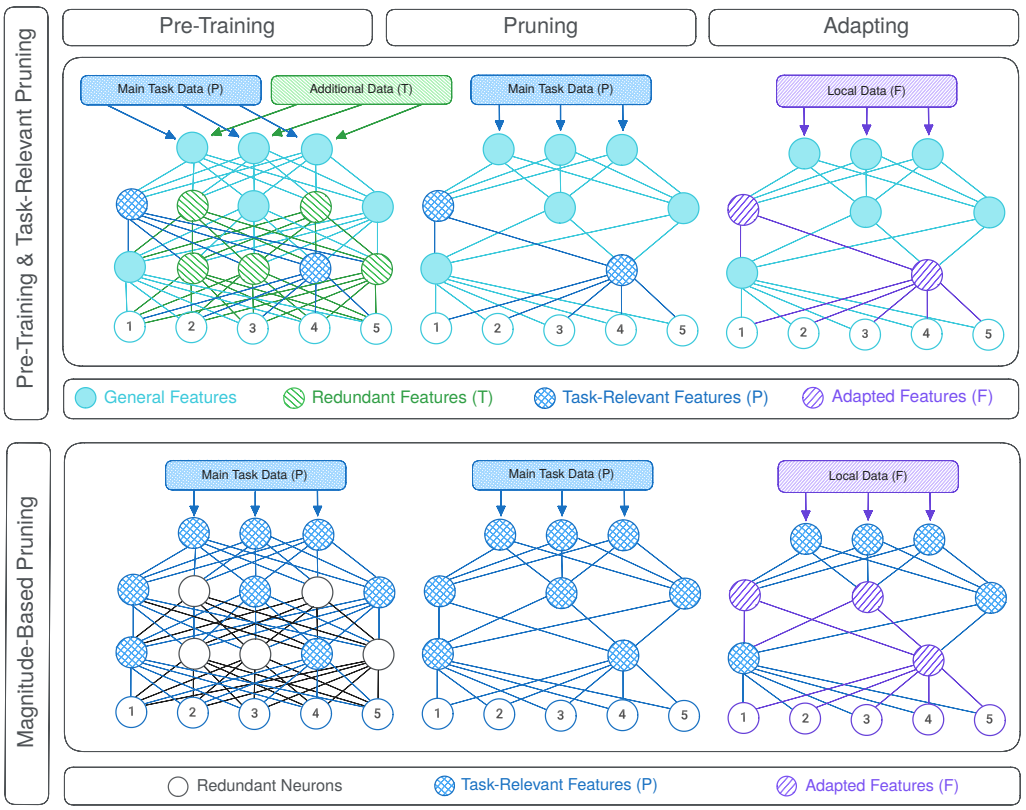


Figure 7.4: Illustration of the proposed pre-training and task-relevant pruning approach, which aims to enhance the generalisation capabilities of compressed models for online adaptation.

We consider two different online model adaptation scenarios in our experiments, with different strategies for splitting the data between training and fine-tuning:

Fine-tuning on new subclasses: In this scenario, the model is trained to classify the same classes as those present during deployment, but it needs to adapt to a shift in data distribution after pruning. We divide the classes in our datasets further into distinct subsets and reserve some subsets of each class for fine-tuning and the additional pre-training data.

Continual learning on new classes: Here, the model is trained using only a subset of the classes it will need to predict during deployment, with the remaining classes only introduced after pruning. Some additional classes are included during training, which are removed during pruning and are not present during deployment.

Formally, the data is split into the following sets, either at the class or subclass level, depending on the evaluated scenario. These are then used as input for Algorithm 3.

T: The extra data that is only used during the pre-training phase.

P: Data used during training and pruning to compute the LRP relevance scores.

F: An additional disjoint subset of data that is withheld for the fine-tuning phase.

We pre-train our model using $T \cup P$, prune it to retain only the knowledge necessary for classifying P and adapt it after pruning using either F (*New Subclasses*) or $P \cup F$ (*New Classes*), depending on the splitting strategy. When adapting to new classes, we employ replay-based continual learning with the complete P dataset for rehearsal, to ensure the model does not forget the original classes during deployment. This is compared to a baseline with $T = \emptyset$, to validate that the extra training knowledge led to better generalisation. We also run all experiments using IMP to verify that LRP-based task-relevant pruning can indeed more effectively compress the model while retaining all knowledge required for classifying P.

Algorithm 3: Pre-Training and Task-Relevant Pruning

$M_0 \leftarrow$ train$(T \cup P)$; `// Train model on additional knowledge and main task`
for $i \leftarrow 0$ **to** *prune_iterations* **do**
  $S_i \leftarrow$ initialise the list of relevance scores with zeros;
  for $d \in P$ **do**
    $a_d \leftarrow$ forward(M_i, d) ; `// Compute neuron activations for input` d
    $S_d \leftarrow LRP(M_i, a)$ using equation 2.30;
    $S \leftarrow S + S_d$;
  end
  $n_i \leftarrow$ compute l_1 norm of neurons/channels in S;
  $M_{i+1} \leftarrow$ remove neurons/channels from M_i with lowest % l_1 norms;
  $M_{i+1} \leftarrow$ train(M_{i+1}, P) ; `// Recover accuracy after pruning`
end

7.4 Experimental Setup

We evaluate our method using the CIFAR-10, CIFAR-100, and DomainNet datasets. These classes in the CIFAR datasets are easily grouped into different subsets, with CIFAR-10 having 10 classes that each belong to the superset of {vehicles, animals} and CIFAR-100 having 20 superclasses with 5 classes each. When fine-tuning on new subclasses, we therefore reduce the number of classes to 2 and 20 respectively and distribute the original classes among T, P, and F. In the continual learning scenario, a different set of classes are reserved for T, P, and F, while still maintaining the full 10 or 100 network outputs. DomainNet was designed specifically for domain adaptation, containing samples of the same 345 classes in 6 different domains: clipart, infograph, painting, quickdraw, real and sketch, so we focus on fine-tuning to new styles for this dataset. Table 7.1 contains the resulting splits, grouped in sets of two: one with additional training knowledge and one where $T = \emptyset$, as shown in Table 7.2.

Name	T	P	F
CIFAR-9/3	birds, cats, deer, dogs, frogs, horses	airplanes, automobiles, trucks	ships
CIFAR-7/4	cars, dogs, horses	airplanes, birds, frogs, ships	cats, deer, trucks
CIFAR-90/30	flowers, large man-made outdoor things, vehicles (1&2), household furniture, household electrical devices	trees, aquatic mammals, fish, food containers, reptiles, people, fruit and vegetables, insects, large carnivores, large natural outdoor scenes, large omnivores and herbivores, non-insect invertebrates	small mammals, medium-sized mammals
CIFAR-60/40	first 2×20 classes	next 2×20 classes	last 1×20 classes
DomainNet5/3	real, sketch	infograph, clipart, quickdraw	painting

Table 7.1: Overview of dataset splits

Each experiment is repeated 5 times for each pruning method (LRP & IMP), to ensure results are consistent. For IMP, we use the magnitude of the weights instead of the relevance scores when computing the l_1 norm in Algorithm 3, Our base architecture is ResNet-50 with 23.5M parameters, which is trained using an RL of 5×10^{-4}, a batch size of 32, the Adam optimizer, and an early stopping criterion based on the validation accuracy, with a patience of 4 epochs. We prune for 10 iterations, each time removing 20% of the neurons in each layer, resulting in 17.5M, 13.3M, 10.3M, 8.3M, 6.7M, 5.6M, 4.8M, 4.1M, 3.6M, and 3.3M parameters. We then continue to the next iteration using the checkpoint with the highest validation accuracy. The adaptation phase is performed

for this checkpoint after each pruning iteration, to evaluate the impact of model size on online adaptability.

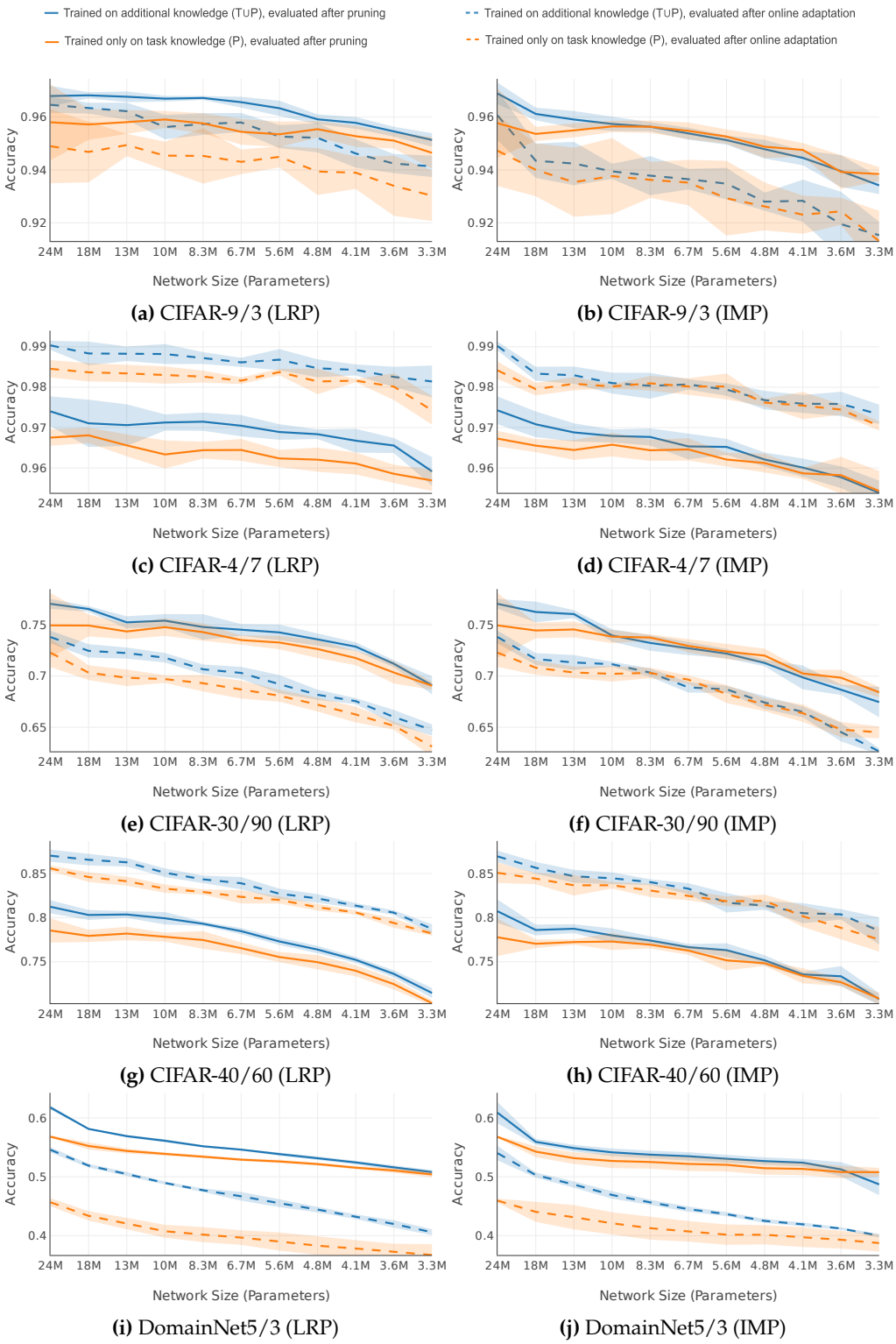
Table 7.2: Overview of our experiment configurations.

Scenario	**New Classes**		**New Subclasses**	
Pre-train on	**T ∪ P**	**P (T=∅)**	**T ∪ P**	**P (T=∅)**
CIFAR-10	CIFAR-9	CIFAR-3	CIFAR-7	CIFAR-4
CIFAR-100	CIFAR-90	CIFAR-30	CIFAR-60	CIFAR-40
DomainNet	N.A.	N.A.	DomainNet5	DomainNet3

7.5 Results

In this section, we evaluate how well the models can be optimised for new data after either IMP or LRP-based task-relevant pruning, both when pre-training on additional knowledge or using only the main task data. Figure 7.5, confirms that the validation accuracy after online adaptation is consistently higher when the model is pre-trained on additional knowledge to improve generalisation. The first point (23.5M) represents the full model accuracy, without any pruning, on P and after online adaptation on F or $P \cup F$ depending on the scenario. Even the full model sees a significant improvement in validation accuracy from pre-training on the additional knowledge due to the increased generalisation. More importantly, this improvement is maintained effectively after our LRP-based pruning, with a consistent gap between the accuracies of the two pre-training approaches on both P and F. This difference does gradually shrink as the higher compression rates cause more of the generalisable features to be pruned, especially for the DomainNet dataset in Figure 7.5i.

The IMP experiments only show a similar improvement in the first pruning iterations, as it cannot differentiate between task-relevant and additional features. This results in more retraining on P to recover the accuracy after pruning, overriding the general features learned in pre-training. We also conclude the models maintain reasonably high accuracy after pruning and adapt well to new data. In continual learning, $F \cup P$ accuracy is lower than on P after pruning due to the classification task being harder at the same capacity. For the CIFAR-10 and CIFAR-100 datasets, accuracy after fine-tuning is higher than post-pruning (Figures 7.5c, 7.5d, 7.5g, 7.5h), as F contains fewer subclasses, simplifying the task. The DomainNet experiments reveal both the largest accuracy drop and the biggest benefit from pre-training on additional knowledge because the style in F (painting) is closer to T (real, sketch) than P (infograph, clipart, quickdraw). Heavy pruning and lack of general knowledge make adaptation harder for DomainNet3 which is optimised for abstract styles. In the last two IMP iterations with DomainNet5, accuracy on P drops below Domainet3 due to less intelligent pruning, although adaptation performance remains higher. These findings suggest that our pre-training and task-relevant pruning approach more effectively compresses models while preserving their generalisation capabilities for online adaptation.



(a) CIFAR-9/3 (LRP) **(b)** CIFAR-9/3 (IMP)

(c) CIFAR-4/7 (LRP) **(d)** CIFAR-4/7 (IMP)

(e) CIFAR-30/90 (LRP) **(f)** CIFAR-30/90 (IMP)

(g) CIFAR-40/60 (LRP) **(h)** CIFAR-40/60 (IMP)

(i) DomainNet5/3 (LRP) **(j)** DomainNet5/3 (IMP)

Figure 7.5: Pruning and validation accuracy results for all datasets and scenarios.

7.6 Conclusion

Adapting to new data is crucial for edge devices in dynamic environments, but their limited resources restrict model size and complexity. This necessitates model compression such as pruning, but these typically do not account for online learning scenarios. In this chapter, we presented a novel approach that extends the generalisation of pruned models for online adaptation without compromising compression rates. By pre-training on additional knowledge and using LRP to compute relevance scores, we identified and retained only neurons encoding task-relevant knowledge during pruning, while still benefiting from the increased generalisation. Experiments on CIFAR-10, CIFAR-100, and DomainNet confirmed that this increased generalisation resulted in higher validation accuracies, which were better maintained post-pruning compared to an IMP baseline. The improved generalisation significantly enhanced accuracy during online adaptation, both for learning new classes and when fine-tuning to new subclasses. Our approach allows for effective model compression while maintaining high generalisation on new data, making it suitable for online adaptation scenarios.

Chapter 8

Domain-Specific Adaptations

In this chapter, we present two contributions that were made with a specific application in mind, rather than a general approach to compression. This doesn't mean that these methods don't generalise to other potential areas, but the focus is on applying model compression for a specific problem in a specific domain. It also serves as a set of guidelines for how to optimise the distillation process to different use cases, and adapt the environments and training strategies to make the most of the distillation process.

In the first section, we present a method for compressing a DRL model that is trained to scale the number of NF replicas based on the current workload. Then, in Section 8.2, we introduce a modification to the PD algorithm that enables the training of compressed student networks with RNN-based architectures to solve embodied point-goal navigation tasks. This last contribution was not published and consequently contains a less extensive analysis of the results, but it is included as a reference for how the distillation process can be effectively modified to handle these types of models.

8.1 Efficient Scaling of Network-Function Replicas

The contributions presented in this section are based on the following publication:

> **Thomas Avé**, Paola Soto, Miguel Camelo, Tom De Schepper, and Kevin Mets. *"Policy Compression for Low-Power Intelligent Scaling in Software-Based Network Architectures."* In NOMS 2024-2024 IEEE Network Operations and Management Symposium, pp. 1-7. IEEE, 2024.

8.1.1 Introduction

Existing networks comprise a complex set of heterogeneous devices that must be integrated to provide seamless end-to-end services. Until very recently, the planning, implementation, and management of this mix of services has been a largely manual activity with some automated assistance. However, it is recognised that these services can no longer be managed using such approaches. The integration of new technologies, such as virtualisation, 5G, and AI that together provide scalable mechanisms for managing such increased

complexity, as well as a new level of automation and intelligence in the management and provisioning of services and networks, are expected to drive the next generation of networks. This vision can only be achieved by ANs (Autonomous Networks) [36]. The objective of ANs is to provide a wide variety of autonomous "Networks/ICT" services, infrastructure, and capabilities with "Zero-X" (zero-wait, zero-touch, zero-trouble) experience based on fully automated lifecycle operations of "Self-X" (self-serving, self-fulfilling, self-assuring) to accommodate and adapt to customer needs and available resources dynamically. These services range from more efficient versions of current services to mission-critical services to new disruptive services that support new business models and innovative user experiences. Customers can benefit from increased network reliability, optimised usage, control, connectivity, and customisable services using ANs.

With the surge of data traffic and varied service needs, integrating data-driven techniques, such as ML for intelligent resource scaling within AN is essential. These advanced algorithms enable real-time analysis and decision-making, optimising resource orchestration to maximise efficiency, reduce latency, and maintain high-quality service, even amid fluctuating network loads and changing service requirements. The shift from traditional, static resource allocation methods to a more proactive, predictive AN resource management approach is critical to meet the stringent operational requirements of modern telecommunication networks. Scaling is a decision-making problem that can be modelled using an MDP, and solved using RL techniques. This way of learning, which can be translated into an agent controlling an environment in a closed-loop fashion, provides the capabilities to achieve autoscaling of resources in ANs, which are part of the self-managing and self-optimising properties.

If we focus on auto-scaling computing resources [126, 117], DRL-based controllers will perform decision-making actions that involve repeatedly adjusting the number of software-based replicas of NFs/NSs (Network Services) while trading-off network/service performance (e.g., latency or throughput) and resource utilisation (e.g., computing or memory usage), as illustrated in Figure 8.1. This self-management and optimisation capability is obtained after training a DRL-based scaler by interacting with the network environment it wants to control. Traditionally, the DRL-based scaler will learn from patterns in the workload of a particular deployment configuration to decide when it needs to scale the computing resources proactively instead of simply reacting to the increasing load.

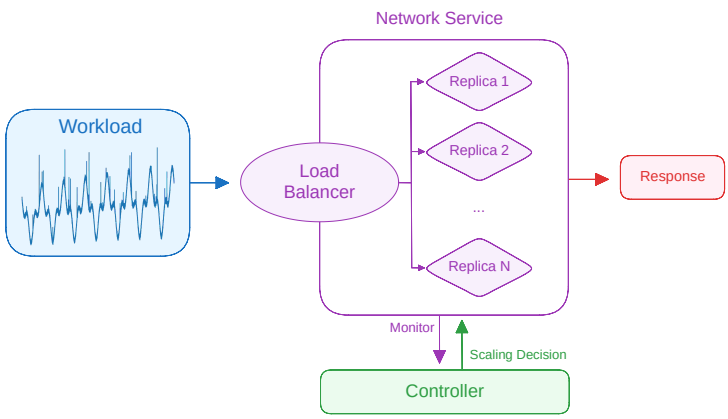


Figure 8.1: An illustration of the studied NFs scaling setting in an SDN environment.

DRL-based scalers have shown high performance in achieving a balanced trade-off between multiple KPIs (Key Performance Indicators) [126, 117] and have outperformed both rule- and control theory-based approaches [127]. However, the computational cost of running this kind of controller prevents them from running efficiently on resource-constrained devices such as the computing resources available at the edge (e.g., edge data centre [126]) and beyond it (e.g., a wireless AP itself [62]). These can even be shared with other NFs/NSs, worsening the problem. This trade-off between the performance of DRL-based scalers and their computational cost is illustrated in Figure 8.2. Not running efficiently is translated into making decisions at a speed lower than required, causing the scaler not to be able to run in real-time, which must be avoided in mission-critical and latency-sensitive applications.

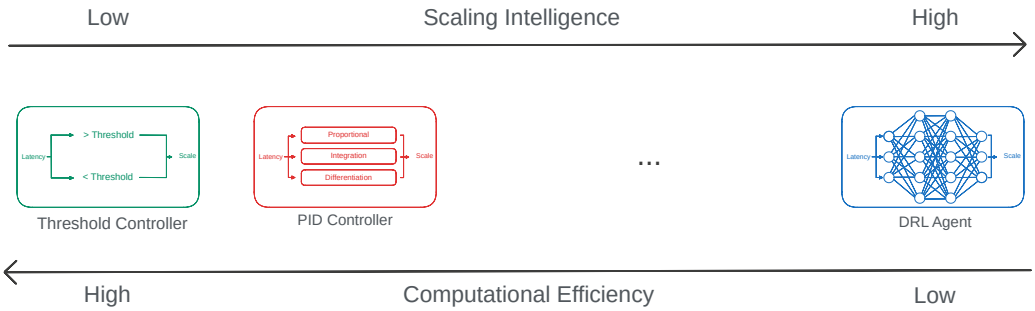


Figure 8.2: The trade-off between the computational complexity and scaling intelligence of DRL-based and traditional NF scalers.

This work aims to bridge the gap between the high performance of DRL-based controllers at solving the scaling problem and its deployment options when running in resource-constrained environments. For this, we first use a data-driven approach to learn the optimal strategy that can be tuned for a specific environment configuration. Then, the trained model is compressed, without altering its effectiveness, into a model that can be deployed in extremely low-power devices. By applying model compression on a larger model, we can avoid the optimisation problems that would be encountered when training a tiny network from scratch, thereby obtaining a smaller and more effective model than would otherwise be possible. Through this process, we can obtain a model comparable to a threshold-based scaler in computational complexity while retaining all the benefits of the data-driven DRL scaler. This tiny size also allows us to interpret and validate the behaviour of the scaler policy, ensuring it performs according to expectations under various conditions. To the author's best knowledge, this is the first work applying model compression techniques to DRL-based scalers.

The remainder of this contribution is organised as follows. In Sections 8.1.2 and 8.1.3, we first provide an overview of the related work and some background on the environment used as part of our methodology in Section 8.1.4. There, we present a 4-phase training strategy and propose modifications to the environment that improve compatibility with the distillation process. The results of our experiment are discussed in Section 8.1.5, including a runtime and scaling performance analysis, while using the increased interpretability of our compressed model to validate the final policy behaviour before concluding this work in Section 8.1.6.

8.1.2 Related Work

Recent research highlights the growing interest in employing RL for intelligent scaling decisions in network environments. Rossi et al. [112] explored both model-free (with Q-learning) and model-based (with Dyna-Q) RL approaches for auto-scaling in Docker Swarm. He et al. [50] demonstrated the combination of RL with GNN (Graph Neural Networks) in NS scaling, achieving better system cost and packet processing rates. Khaleq et al. [67] proposed an RL algorithm for Kubernetes cluster control, reducing the response time of microservices by up to 20%. Finally, Santos et al. [117] developed the "gym-hpa" framework, integrating OpenAI's platform with Kubernetes, demonstrating significant resource usage reduction and latency improvement in microservice environments compared to default Kubernetes scaling mechanisms. Collectively, these studies underscore the effectiveness of RL in optimising resource scaling, considering various network conditions and objectives.

Complementary to these results, previous work within IDLab applied DRL for dynamic scaling of replicas compared threshold-based, control-theoretic, and DRL-based solutions for scaling, observing fewer SLA (Service Level Agreement) violations with DRL despite creating more replicas [127] and highlighted the impact of the reward function variations on DRL scaling behaviour [126].

State-of-the-art research highlights the capabilities of DRL-based scalers. However, the existing literature lacks a specific exploration of the computing cost when deploying these models on resource-constrained platforms. Although policy compression techniques show promise in reducing DRL model requirements, their application to resource scalers remains unexplored.

8.1.3 Background

In this work, we base ourselves on the same RL environment proposed by Soto et al. [126], to show how our method can successfully be applied to an established simulation with a workload pattern that reflects real-world data centres. We also use the final model from this earlier work, obtained after being trained for 172800 steps, as a baseline for runtime performance and as a teacher for policy compression. This way, we demonstrate how our models achieve state-of-the-art scaling efficacy performance at only a tiny fraction of the runtime cost. Specifically, Soto et al. [126] proposed a generic framework for modelling the scaling problem through an MDP, which is defined as follows:

Reward (r) = $-1 * (w_{res} \times a + w_{perf} \times I_{\text{violation}})$, where:

$$I_{violation} = \begin{cases} 1, & \text{if peak latency > SLA threshold} \\ 0, & \text{otherwise} \end{cases}$$

State (s) = (number of replicas, CPU utilisation, peak latency).

Action (a) $\in$ {+1, -1, +0}, modifying the number of replicas.

Latency is defined here as the end-to-end RTT (Round-Trip Time) of the NF in seconds, including the processing time of a request by the NF and queuing delays if the CPU utilisation is too high to process the request immediately (at a maximum of 300 jobs/step). In the initial state, two replicas are allocated, and the transition function is modelled using the Sim-Diasca simulation engine[125] with a fixed network workload trace. Therefore, the objective of the agent DRL is to stay just below the SLA threshold (w_{perf}) with the lowest acceptable number of replicas (w_{res}). The episode ends when either 172800 steps have been made (2 days simulation time at 1s / step), or the number of replicas (> 20) or peak latency (> 2s) are too high. Several combinations for these weights were evaluated to show how the scalers can adapt their behaviour based on what they are focused on optimising. Soto et al. [126] found that the values $(w_{res}, w_{perf}) = (0.01, 0.99)$ struck the best balance between reducing SLA violations (latency > 24ms) to an absolute minimum (0.0023%), while still keeping the number of replicas as low as possible (average of 5.5). We will, therefore, also use these values in our experiments.

8.1.4 Methodology & Experimental Setup

Our goal is to produce a DRL model that can proactively scale the number of replicas based on predicted local workload patterns while being negligible in computational footprint compared to the actual NF themselves. To this end, we propose a four-phase training strategy, which is outlined in Figure 8.3. The first and last phases are performed on the edge, while the second and third phases are performed on a more powerful cloud server.

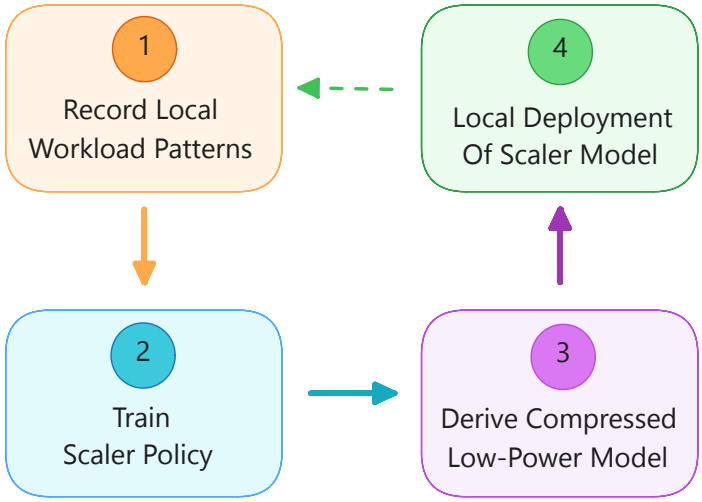


Figure 8.3: Low-Power Scaling Policy Training Strategy.

Phase 1: By deploying a set of low-power controllers, we can optimise each policy specifically for the local workload patterns. A key benefit of applying a data-driven approach is that the scaler can potentially learn patterns in the workload and thereby anticipate demand. However, these patterns can vary wildly between different deployments, so it is important to train the scaler on a workload trace that is indicative of its actual deployment

environment. In a deployment with a higher variability in the workload, it would, for example, need to keep CPU utilisation lower on average to compensate for sudden spikes. But in other settings, where the demand is a lot steadier, it can learn to scale only when it becomes necessary and use the smaller number of replicas more efficiently.

This phase, therefore, consists of gathering this local workload trace ahead of training to ensure that our simulator provides an accurate reflection of the local deployment conditions. In our experiments, we will be using the workload trace from our earlier work [126], to make comparing our final model to the existing work more straight-forward.

Phase 2: This workload pattern is used in our simulator to provide a training environment that accurately reflects the local deployment conditions. We used the same policy architecture as Soto et al. [126], which consists of the default PPO agent generated by the popular SB3 project [108], for a state and action space of size three each. This agent was trained using a single episode of length 172800, corresponding to the first two days of the workload trace, at an LR of 3×10^{-4}.

This results in a DNN with three hidden layers that are joined by the *hyperbolic tangent* activation function, resulting in 9092 trainable parameters, of which 65 belong to the critic head and are only used during training. We then confirmed that a similar 5.54 replicas were used on average, with an average peak latency of 8.5ms and CPU utilisation of 67.6%, when evaluated on the last two days of the workload trace. These metrics serve as a baseline that should be reproduced as closely as possible after compression, but they also serve an important role during compression, to find the lower bound on the compression potential.

Phase 3: During the compression phase, we use PD (See Algorithm 1) to transfer the knowledge of the PPO agent trained in the previous phase to a new student model that is a tiny fraction of its size. We first introduce an iterative process to find the smallest student size that is still able to satisfy the SLA requirements and match the teacher metrics reported earlier. An overview of this workflow process is given in Figure 8.4. This essentially consists of performing PD iteratively on smaller and smaller student architectures until their performance at replica scaling is no longer satisfactory. Each time this occurs, however, we first perform a new hyperparameter search with different architecture configurations and LRs, and continue if this yields a recovery. Otherwise, we publish the checkpoint of the student with the last architecture that did meet our requirements. Our final student model was trained using a mini-batch size of 64 and an LR of 1×10^{-5}.

We also introduced several modifications to the environment that make it better suited for the distillation processes, but left it unaltered during the previous phase to remain strictly compatible with teachers from previous work. First, we removed the condition for ending an episode early if the number of allocated replicas or the peak latency is too high. Since the reward is in the form of a cost function, this could encourage the agent to end the episode as quickly as possible, yielding a higher overall return. By removing these ending conditions, we were able to more accurately compare different student architectures and hyperparameter configurations by looking at the obtained returns.

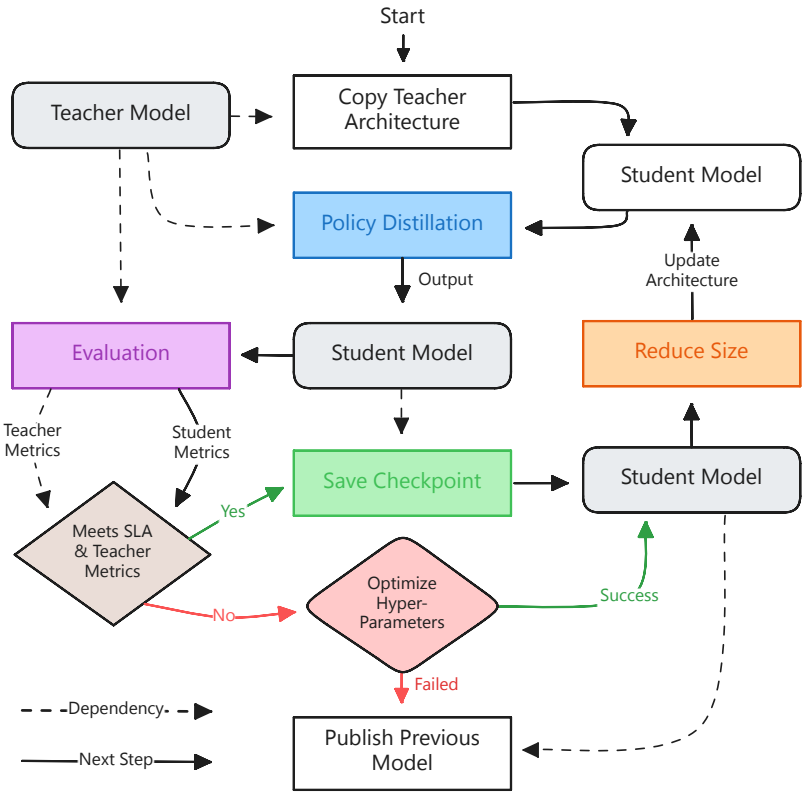


Figure 8.4: The iterative workflow of the compression phase.

Another aspect that was distinctive about this training setup is that only a single episode was used to train the PPO agent, which is designed to benefit from collecting multiple rollouts in parallel. This impacts the distillation process by requiring the entire dataset to be recorded at once in the replay memory. Normally, this memory contains transitions from multiple smaller episodes, gathered using a stochastic policy, to widen the state distribution from which the student can learn.

As described in algorithm 1, the oldest episodes in this memory are then periodically refreshed while training the student, to ensure that it does not overfit on these particular trajectories and to increase generalisation. However, since the workload trace is static and independent of the previous actions of the RL agent, we can split the data into multiple episodes, without substantially altering the dynamics of the system. We therefore split the two-day training trace into ten different sections and used parallel workers to collect transitions to fill the replay memory, with a capacity of five such episodes. While training the student, we continue collecting more trajectories and refresh the memory by replacing the oldest episode every five epochs.

The used workload trace consists of seven days in total, but only the first two and last two were used for training and validating the teacher agent, respectively. We therefore also used the first two days for training our student and the last two days to test the final model performance, but the additional three days in between are now used during this

phase for validation purposes. This was done when optimising student hyperparameters or when checking if the student is still able to satisfy the SLA requirements and whether the performance metrics still match those of the teacher. If not, we found the smallest student model that still meets our requirements, and we continue to the next phase.

Phase 4: After compression, the final policy can be deployed efficiently and effectively for inference on the local device. While deployed, additional traces can be gathered for adapting to changing workload patterns by returning to phase 1. Alternatively, it can be further fine-tuned locally if available resources allow for it, and our actor-critic distillation method from Chapter 3 is used to distil the additional critic head. Since at this point the student has already been heavily compressed, this should be feasible on most devices that support backpropagation.

8.1.5 Results

We start this section by investigating how our student behaves during training and how the final evaluation metrics compare to those of the teacher. Then, we take a close look at the exact architecture of this final student, before concluding with a runtime performance analysis.

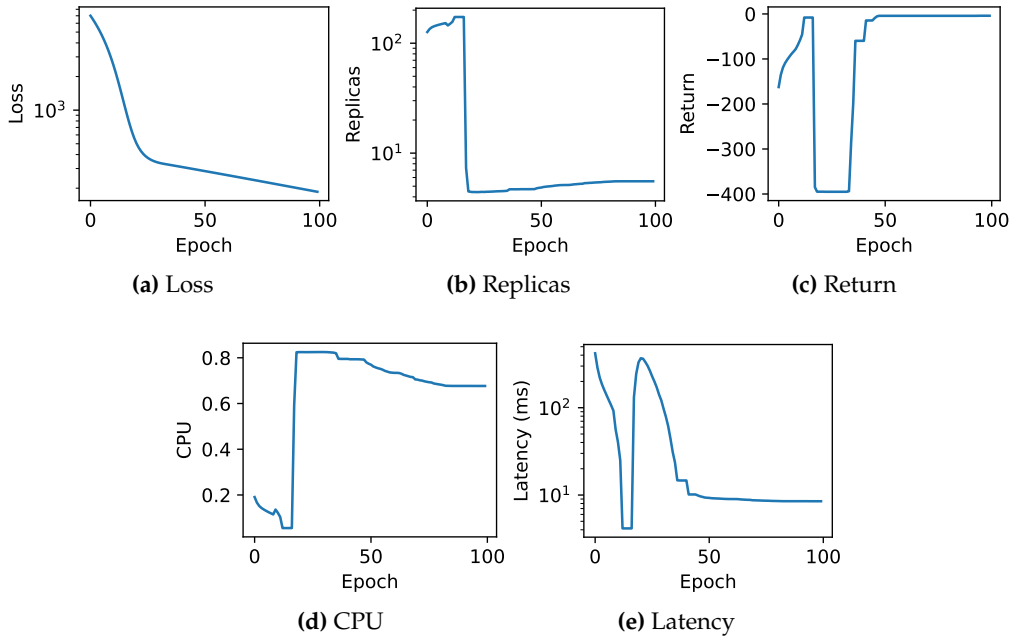
8.1.5.1 Scaling Performance

Figure 8.5 shows the metrics during a single training run of our final student architecture. Although we repeated our experiments 50 times to ensure we obtained a consistent result, the point at which each run reaches a particular stage varies significantly, so we selected a single example for discussion. The loss in Figure 8.5a indicates how similar the outputs of our students are to those of their teacher. As training continues, this loss converges steadily, in contrast to the other metrics. We see that initially, the model learns to increase the number of replicas by a large amount, which has a positive impact on the return since the SLA violations are more heavily penalised than wasted resources (i.e., the corresponding low CPU utilisation).

After 17 epochs, the student learns that the teacher generally keeps the number of replicas much lower, but it has yet to figure out when to scale it appropriately based on the workload demand, so the latency increases past the SLA threshold and the return plummets. It then immediately starts learning how to reduce the latency again, however, while keeping the number of replicas low on average. Once the latency is back under the threshold, the return quickly recovers to an all-time high and stays perfectly constant afterwards. During the final phase, the student still improves the average CPU utilisation, before matching the metrics of the teacher almost perfectly after 80 epochs of training. A comparison of the final results can be seen in Table 8.1.

Table 8.1: An overview of our final scaler metrics.

Model	Parameters	Return	Replicas	CPU	Latency
Teacher	9027	-4.0	5.5437	67.685%	8.500ms
Student	9	-4.0	5.5622	67.435%	8.486ms



(a) Loss **(b)** Replicas **(c)** Return

(d) CPU **(e)** Latency

Figure 8.5: Metrics recorded while training our final student model.

8.1.5.2 Student Architecture

This table also lists the size of our final student in terms of trainable parameters. As can be seen, we were able to reproduce the teacher's policy almost exactly using only nine parameters, a compression of 1003 times. This is the minimum number of parameters given the number of network inputs and outputs, using a single linear fully connected layer, without biases. Note that such a high compression ratio is only possible because the original teacher model was optimised purely for performance, without any regard for the number of parameters. The relative improvement would likely be more modest if the teacher architecture was also selected for efficiency.

A representation of the full network architecture is depicted in Figure 8.6. Here, the input layer has three nodes that represent the network resource metrics that are encoded in the state (observations). The output layer has three nodes that correspond to the actions that the student can take, being +1 to increase, -1 to decrease, or +0 to maintain the current number of replicas. To compute the value of each output node, the input values are multiplied by the corresponding weights in the same colour as the output and are

summed together. The student policy then consists of taking the action with the highest predicted output value. Given the tiny size of this network, we can actually interpret how the model reasons about when to perform which action. By looking at the red arrows, we see that the model increases the number of replicas when the latency or CPU utilisation is too high. The green arrows indicate that the model reduces the number of replicas if these same input values are low enough and there are already a sufficient number of replicas present. If neither of these inputs is too high, the model will keep the configuration as is.

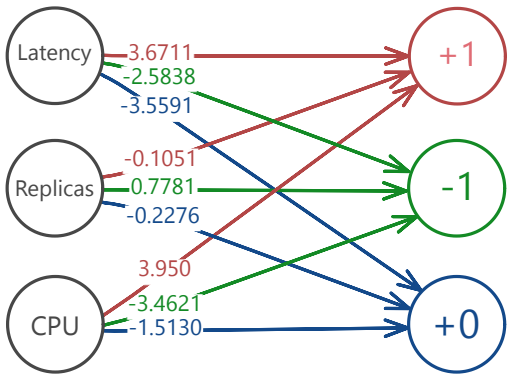


Figure 8.6: The architecture and weights of our final student model.

As a baseline, we tried training this student architecture as the actor with the same PPO setup that was used for the teacher, using a similarly sized (as the teacher) critic to give it the best opportunity to be able to learn useful behaviour. Unfortunately, in this configuration, the model was unable to learn any behaviour that was noticeably better than a randomly initialised network, mostly choosing the same action regardless of the input. This does, however, confirm the need for our distillation approach, as it was necessary to use a larger network to find this intelligent data-driven policy. But, through our proposed setup, we were able to compress it to such an extent that it becomes comparable to a threshold-based controller in computational efficiency.

8.1.5.3 Runtime Performance

To quantify this further, we measured the impact of both our teacher and our student in terms of run-time performance, using two metrics: the average number of steps that can be executed per second and the total amount of system memory required to perform a single step. We trained both our teacher and our student using PyTorch in Python, but to take a more representative measurement of how these models would typically be deployed after training on low-power devices, we exported the computational graphs of both networks as ONNX programs. These representations were then loaded by a simple C++ program using the OpenCV library [20]. To measure system memory consumption, our program performed a single inference step and printed the network predictions to the system's standard output. We then measured the maximum RSS (Resident Set Size) occupied by each process, as can be seen in Table 8.2, Although there is no similar 1003x reduction in actual memory usage due to the overhead of loading the model and computing intermediate values, we can still observe an impressive 28.5x reduction in effective memory requirements.

Table 8.2: An overview of the model runtime performances.

Model	Memory Consumption (RSS)	Steps per second
Teacher	102.5 MB	1.5335×10^5
Student	3.6 MB	2.2810×10^8

The number of steps the models can take per second (also see Table 8.2) was then measured similarly, but for a much larger number of steps to obtain an accurate average measurement. The teacher was able to perform 10^6 steps in 6.521 seconds and the student 10^9 steps in 4.384 seconds, resulting in an improvement in runtime speed of 1487x. These measurements were performed on an AMD Ryzen 9 3900X CPU, but the relative performance difference between the two models should be similar on other system configurations. This is even higher than purely the reduction in model parameters, likely due to the computational graph being not only smaller, but less complex.

8.1.6 Conclusions

This work presents a novel approach for training a tiny data-driven scaler based on DRL that can be deployed efficiently on low-power devices. The compressed model, significantly more efficient in computation and memory usage, retains the benefits of data-driven DRL scalers, ensuring high-performance scaling decisions at an accelerated rate and with minimal resource consumption. This was accomplished by introducing a 4-phase training strategy that involves recording local workload traces, using this to train a PPO agent in simulation, compressing it using our PD workflow, and finally deploying the compressed tiny scaler model on a low-power device. To increase the effectiveness of our policy compression method, we adapted the scaler environment to make it more compatible with the distillation process.

Results show that a student trained using our methods can reproduce the teacher's policy almost exactly using only nine parameters, a compression of 1003 times compared to existing work. It also increased the interpretability of the model, meaning we could validate the behaviour of the learned policy. A runtime analysis further quantified that this model could perform 1487x more scaling decisions per second while using only 3.6 MB of memory. Not only does this mean that the controller can be deployed significantly more efficiently, but it also increases the granularity at which the scaling operations can be made, leading to more efficient use of the available resources and ensuring real-time scaling on ultra-low-power devices.

8.2 Recurrent Distillation for Embodied Navigation

This section consists of novel work that has not been submitted for publication.

8.2.1 Introduction

As demonstrated in the previous sections, DRL has shown great potential in solving complex sequential decision-making tasks across a wide range of domains, including scaling in SDNs (Section 8.1) and playing Atari games (Chapters 3 and 4). These tasks typically involve learning how to make a sequence of decisions in a structured and well-defined environment that can be fully and accurately observed at each time step. This is an ideal scenario, where the agent can focus on learning the optimal policy for solving the actual task, without having to worry about partial observability or noisy observations and inconsistencies in applying actions. When deploying DRL agents in real-world applications on embedded devices at the edge, however, these ideal conditions are rarely met.

Embodied RL tasks, such as the robotics and navigation tasks used in Chapters 5 and 6 respectively, are often especially effected by this. Here, the agent is placed in a physical environment and must learn to navigate around obstacles, explore its changing surroundings, and perform a series of subtasks to achieve a goal [59]. Tasks with physical exploration or planning require the agent to keep track of where it has been, what it has seen, and what it has already done [32]. To solve these tasks, the agent must maintain an internal memory, which is typically done using an RNN to maintain a hidden state that is updated at each time step. This approach generally works well, but it also makes the models more complex and computationally expensive, which can be problematic when deploying them on resource-constrained devices.

In the previous sections, we focused on compression for feedforward networks, which only consider the current observation when making a decision. With the introduction of temporal distillation in Chapter 6, we did already introduce a temporal aspect to the distillation process, but in practice, this only involves learning an additional repeat value that is constant for a given observation. During training, transitions in the form of a tuple (observation, repeat value, teacher outputs) are still sampled completely randomly from the replay memory, which means that the order in which these transitions are encountered during training is not taken into account.

In this section, we propose a modification to the PD algorithm that enables training student networks with RNNs to solve embodied point-goal navigation tasks, while still benefiting from the compression potential of our earlier distillation methods. In our experiments, we investigate whether it is even possible to learn this behaviour that relies on maintaining a useful hidden state purely based on example trajectories from a teacher network in PD, without the need for direct on-policy interactions with the environment. To do so, we compare our approach to a baseline feedforward student network without any form of memory, trained using regular PD and the same teacher network. We show how the RNN-based student can effectively generalise to unseen paths, while the baseline student only learns to mimic the teacher's actions on the training trajectories.

8.2.2 Methodology

DRL agents that are trained to solve embodied navigation tasks typically use RNNs to maintain a form of memory that is updated at each time step [32]. These networks utilise a hidden state of a fixed size to maintain context across steps. The output of the network now also includes this hidden state, which is passed back as part of the input of the network for the next step in the environment, where it is updated again to reflect changes in the new observation and corresponding decisions. This state serves as a memory for the network, which is often necessary for tasks where future actions depend on previous choices or for environments that are only partially observable at a single point in time (POMDP).

Unlike the other network outputs discussed in the previous chapters, it does not make sense for the student network to emulate the hidden RNN states of the teacher for randomly sampled observations, since these RNNs are typically trained using backpropagation through time. It would also severely limit the degree of compression that can be achieved in the student network if the RNN size needs to match.

For these RNNs to work correctly, they rely on being trained using the same order of observations that was encountered during interactions between the agent and the environment. This is not the case when training a student network using regular PD, where the order between transitions is not stored in the replay memory, and batches are sampled randomly. The first part of the solution is to add three additional attributes to the entries in the replay memory: whether it involves an initial state, a terminal state, and a reference to the next entry. After sampling entries, but before feeding this to the student to compute the distillation loss, we augment each transition by including the student RNN hidden state from the previous sample in the trajectory, in addition to the new observation and teacher outputs.

Algorithm 4: Recurrent Transition Sampling

Input: Replay memory D, the previous batch b_{i-1},
updated student RNN hidden states h_i.
Result: The next batch b_i.

for *non-terminal state s in b_{i-1}* **do**
    Set the observation and teacher outputs in b_i to those of the next step by following the reference attribute in s;
    Set the corresponding RNN hidden states in b_i to those in h_i;
end
for *terminal state s in b_{i-1}* **do**
    Set the observation and teacher outputs in b_i to a randomly sampled initial state from D;
    Set the RNN hidden states in b_i to initial values;
end

Creating the first batch of samples then simply consists of selecting transitions from the replay memory that only include initial states, and constructing a set of initial student RNN states. These initial hidden states can either be set to a fixed value (usually zero), or be learned as parameters of the student network. All subsequent batches can then be computed as shown in algorithm 4.

By replacing batch entries that have reached final states with new initial states, we ensure that the batch size stays consistent for better GPU utilisation. It also enables the student to see a wider range of trajectory stages in a single batch, since trajectories with different lengths are replaced at different times, resulting in more stable training. After each batch, the student is optimised using the distillation loss from the last n batches, with n the number of time steps used in the backpropagation through time.

This is done by computing the distillation loss for each time step and aggregating these losses using a sum or mean operation. The aggregated loss is then back-propagated through the student network, taking into account that each output depends on the previous hidden state, so it is influenced by the same parameter multiple times [113]. Note that at every training step, only the distillation loss for those trajectories is used for which the current length is a multiple of n, or those that have reached their final states.

8.2.3 Experimental Setup

As mentioned in the introduction of this chapter, we developed this method with the goal of applying it to embodied point-goal navigation tasks. Specifically, we want to train a student that can efficiently navigate a 3D virtual scan of our university office environment, as shown in Figure 8.7.

(a) RGB **(b)** Depth

Figure 8.7: Two examples of observations in the Beacon Habitat environment, one in RGB and one with depth information.

This environment was developed using the Habitat platform [118], which provides a flexible and modular library for modelling 3D RL environments. Our research group created this scan to facilitate the training of agents in a simulated replica of our real office environment, with the eventual goal of applying a *Sim2Real* technique and deploying

them on a physical Locobot [85] that can navigate and perform tasks in the actual space. Getting there requires many challenges to be overcome, but in this work we focus on the issue of running these agents on the Locobot itself, which has a relatively low-power computing device with limited energy resources.

As part of this, many tasks were defined that the agent must be able to perform, such as finding an object of a specific category (e.g., a chair) in object-goal navigation, or navigating to a specified location (e.g., the kitchen) in point-goal navigation. In our experiments, we focus on the latter, where the agent is placed at a random location in the environment and must navigate to another random target location on the same floor, for which a map is shown in Figure 8.8. We train our teacher and student on one set of start and goal location combinations, and evaluate them on a disjunct set of task configurations to test their abilities in terms of generalisation.

Figure 8.8: The floor map of the 7th floor of our university building.

The agent receives observations in the form of depth images, as shown in Figure 8.7b, a compass reading, and its current location coordinates. This compass reading contains the Euclidean distance and angle to the goal location. We give the agent a reward $r = 10$ when it reaches the goal, a penalty of $r = -0.01$ when it doesn't move, and otherwise $r \in [-1, 1]$ based on the change in Euclidean distance to the goal, which is positive for moving closer and negative for moving further away. To move around, the agent can choose between four actions: move forward, turn left, turn right, and stop. This last action ends the episode when the agent decides it has reached the goal, after which it will be rewarded if it is within a small radius of the target location.

The teacher network used for these experiments is a DD-PPO (Decentralized Distributed Proximal Policy Optimization) [144] agent, which is an algorithm based on PPO, but optimised for distributed RL with resource-intensive simulated environments. It has 8,476,421 parameters, with a ResNet-18 backbone architecture and 2 LSTM (Long Short-Term Memory) [42] layers with 512 units each. Our student architecture is much simpler, with only 32 units in a single GRU (Gated Recurrent Unit) [25] layer, 3 convolutional layers with 32 channels each, resulting in a total of only 210,724 parameters.

Network	Parameters	Relative Size
Teacher	8,476,421	100%
Student (RNN)	210,724	2.49%
Student (No RNN)	201,284	2.37%

Table 8.3: The relative size of the student networks compared to their teacher.

We compare this to a student without an RNN layer, but otherwise an identical architecture. This non-RNN student was also trained using regular PD, instead of the recurrent transition sampling described in Algorithm 4, to evaluate the effectiveness of our proposed changes. An overview of the relative network sizes can be seen in Table 8.3.

A teacher-driven control policy was used to generate the training data, to study whether recurrent distillation is effective at learning to maintain an informative hidden state purely based on example trajectories. To determine the average return and success rate, we evaluate the models for 200 episodes. A subset of start and goal locations is reserved for testing, so that the student is not trained on these specific locations, resulting in different metrics for training and testing.

8.2.4 Results

We evaluate the effectiveness of our proposed changes by training two student networks with very similar architectures and number of trainable parameters, but one containing an RNN and the other without, with the same teacher network to guide them. These results can be seen in Table 8.4.

Network	Train Return	Test Return	Train Success	Test Success
Teacher	24.13	20.60	0.835	0.825
Student (RNN)	23.96	19.92	0.855	0.800
Student (No RNN)	20.42	7.49	0.805	0.410

Table 8.4: The average return and success rate of the student network with and without an RNN, compared to their teacher.

Both students are able to achieve a relatively high average return and success rate on the training configurations, but the key difference can be seen in the test results, where the student with an RNN is able to generalise much better to unseen locations. The simple feedforward student seems to rely heavily on memorisation (not to be confused with having a recurrent hidden state) of the exact teacher behaviour for training trajectories, but completely fails to solve the exploration problem when tested on new locations. We hypothesise that it overfits on the depth observation and does not use the compass readings effectively to navigate to the goal.

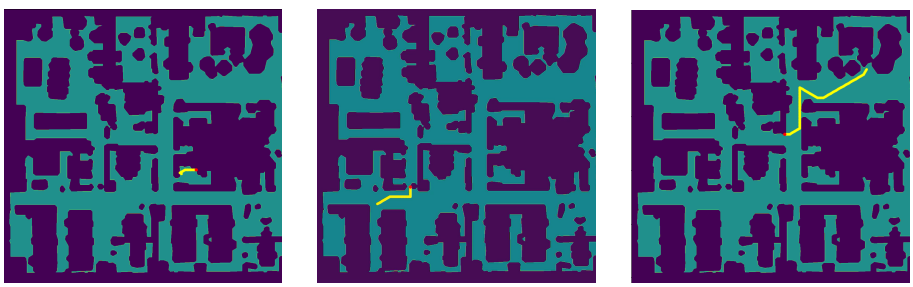


Figure 8.9: Three examples of the non-RNN student getting stuck against a wall, on the left side of the floor map.

When inspecting the trajectories of the non-RNN student, we see that it often gets stuck in corners or against walls, as shown in Figure 8.9. This is caused by the agent not noticing that it is not making progress towards the goal, due to the lack of memory, and therefore keep repeating the same actions over and over again, as the observation stays identical.

In these figures, the start and goal locations are indicated by a red dot and the yellow line shows the path taken by the agent. Note that the floor maps are purely for visualisation purposes and are not accessible to the agent, which only receives depth images and compass readings as observations.

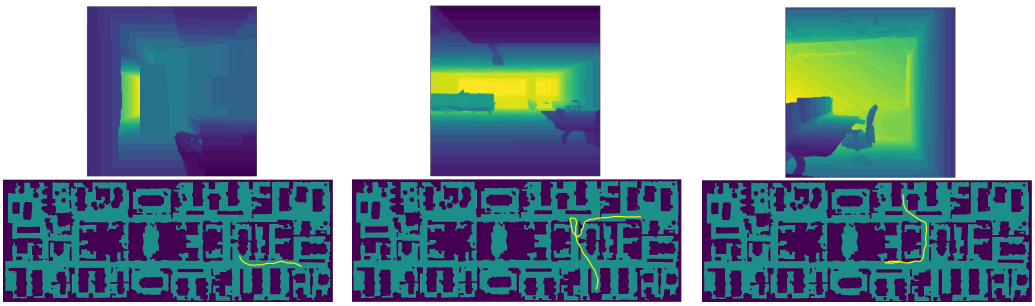


Figure 8.10: Three example trajectories of the RNN student in the middle of successful episodes, showing the floor map and its current observation.

The RNN student does not suffer from this issue, as it is able to keep track of its progress towards the goal and adjust its actions accordingly. Figure 8.10 shows three examples of successful trajectories, where the agent is able to navigate to the goal location without getting stuck. These trajectories are clearly not optimal either, but they do show that the agent is able to explore the environment and even backtrack to find a different route towards the goal location, even though this was not always necessary. It does not quite reach the same average return and success rate as the teacher, but we believe that this is an acceptable trade-off for the 40x reduction in network size.

8.2.5 Conclusions

In this section, we presented an extension to the PD algorithm that enables the compression of RNN-based policy architectures for embodied navigation tasks. Traditional PD does not account for the temporal dependencies in the teacher's trajectories when training the student, which is crucial for learning navigation strategies in partially observable environments.

Our experiments demonstrated the significant advantages of recurrent distillation over a baseline feedforward architecture in the context of embodied point-goal navigation. While both student models achieved comparable performance in terms of average return during training, the RNN-based student showed significantly better generalisation to unseen environments. The non-recurrent student showed signs of overfitting to the training trajectories, and was prone to failure during testing, often getting stuck in corners and repeating actions in a loop due to its inability to track progress.

These findings confirm that, through recurrent PD, it is possible to train a student network with a much smaller number of parameters that can effectively learn to maintain an informative hidden state for embodied navigation tasks, based only on examples gathered through teacher-driven interactions. This opens up new possibilities for deploying more advanced DRL agents on resource-constrained devices in real-world embodied applications.

Chapter 9

Conclusions

With this thesis, considerable progress was made towards enabling the deployment of complex DRL models on highly resource-constrained edge devices not only a reality but a practical and efficient solution. This has traditionally been a challenge due to the large size and consequently slow inference speed, and high energy consumption of these models. Applications that need to be performed at the edge, such as autonomous robotics, often require real-time decision-making and are particularly affected by these limitations.

Since cloud offloading, relying on hardware improvements, and directly training small enough models are not viable solutions, we need to look for alternative methods that use the available resources more intelligently and efficiently. Model compression is a promising approach to this problem, but existing work on this topic lacked the specialisation in DRL models that we believe is necessary to achieve the most efficient solutions.

We hypothesised that, by leveraging the specific characteristics of DRL models, novel methods could be developed that achieve higher compression rates and enable the compression of a wider range of models. A set of research questions was formulated to validate this hypothesis, which guided the development of the methods presented in the previous chapters. Through these contributions, we can now answer the stated research questions in Section 9.1, confirm our hypothesis, and provide a more detailed insight into the potential of policy compression. Afterwards, we will discuss the potential for future work and the remaining challenges in Section 9.2, and conclude with some final remarks in Section 9.3.

9.1 Review of Research Questions

Below, we revisit each of the research questions that were formulated in Section 1.3 and provide a summary of the answers based on the results presented in the previous chapters:

1. **Can we extract auxiliary knowledge from DRL models that enables a student policy to learn a more informative representation of its environment?** We have shown several methods that were successful in extracting additional knowledge, outside a pure focus on distilling the actions predicted by a teacher, and transferring it to a student model that demonstratively improved its effectiveness at solving the task after compression.

In Chapter 3, we introduced a novel distillation loss for actor-critic models that leverages the state-value predictions made by the teacher's critic to provide additional information to the student in the form of an auxiliary task. By plotting the activations of the second to last layer of the student network, we saw that three distinct clusters appeared when the state values were learned, but not when using the regular distillation loss, with each cluster corresponding to a particular range indicating how valuable the state was still expected to be. Through this improvement in the internal representation, we were able to obtain a 7% higher average return on the Atari Breakout environment compared to the regular distillation loss.

When considering how to compress policies for continuous action spaces in Chapter 5, we found that the entropy of the teacher's policy was an important factor to its success. By extracting the entire action distribution and distilling the standard deviation σ of the teacher's policy, in addition to the mean action predictions μ, we were able to increase behavioural fidelity of the student and obtain a reward that was up to 36% higher on average. We also experimented with a method that used the teacher's critic to provide a more informative intrinsic reward to the student. Although this had a positive impact on the convergence rate during training, it did not significantly improve the final model or the compression potential. A similar intrinsic reward based on the similarity of the action predictions likewise did not improve upon our other methods.

Our temporal distillation method, presented in Chapter 6, was able to extract temporal knowledge from the teacher's policy and teach it to the student. We do this by letting the student learn when a different action will be necessary and when it can simply continue to act as it did before, by predicting a number of times to act before it needs to re-evaluate its policy. This temporal knowledge is not explicitly modelled by the teacher, but it is implicitly present in the teacher's behaviour and manifests itself through interactions with the environment. Not only did this method reduce the number of decisions required to complete the task, and therefore significantly improve its efficiency, but explicitly learning this auxiliary task also improved the internal representation to the point that our largest student outperformed a baseline student trained through regular PD in its effectiveness at solving the task.

Similarly, in Chapter 8, we showed that by modifying the replay memory to store the time-dependence between transitions, we could train a student with an RNN-based architecture to learn how to maintain an informative hidden state as a form of memory. This way of improving its current internal representation of the environment is essential for embodied navigation tasks, where the environment is often only partially observable and where the observations and controls are often noisy. Although the teacher also explicitly models a hidden state, we implicitly extract the auxiliary information from the sequential nature of its interactions with the environment. Compared to emulating the teacher's hidden state directly, this allows for more flexibility in the type and size of RNN that can be used for a higher compression potential.

2. **Are certain ways in which a policy can be represented inherently more compressible than others?** We found in Chapter 3 that the PPO and A2C actor-critic policies were more compressible through our PD method than the value-based DQN. After distillation, they achieved both the highest average return in absolute terms and the highest relative improvement over their respective teachers. The DQN agent also suffered slightly more from the reduced capacity at the highest compression

levels. This observation is in line with the results of Raileanu et al. [109], who found that learning a value function requires more capacity than modelling an explicit policy in actor-critic settings. We now confirm this extends to the distillation of DQN agents as well, although the difference is relatively minor and the performance of the teacher is a more significant factor.

Adding quantization to the mix in Chapter 4 tells a different story, where the QPD method was able to compress the DQN agent to a higher degree than the actor-critic agents. Although the low-precision actor-critic models still outperform the value-based DQN for students with more parameters, their performance dropped significantly for the smallest two student architectures. More research is needed to determine whether this was primarily due to a difficulty in modelling a stochastic policy in reduced precision, or if learning the transformed action-value function in low-precision is more stable than representing the policy directly. We can confirm that these issues are fortunately not inherent to quantization alone, as all but the smallest two low-precision students perform just as well as their high-precision counterparts.

Comparing PD for discrete actions to our experiments with continuous action spaces in Chapter 5 is more challenging, as they cannot directly be applied to the same environments. Continuous actions are inherently more complex to represent, however, and teachers provide less 'dark' knowledge from alternative actions that can be transferred to the student. Although our results were certainly positive, we observed the ability of the students to outperform their teacher to a lesser extent than in our experiments with discrete action spaces. A compression of up to 36x without a significant loss in effectiveness was achieved, whereas our actor-critic experiments with discrete actions in Chapter 3 managed to go beyond a 47x reduction in size for the same criteria.

Distilling policies with RNN-based architectures in Chapter 8 shows great promise with a 40x reduction in size for only a 3% drop in average return, but more experiments with a wider range of student sizes are needed to properly assess the full compression potential.

3. **How detrimental is the reduced representational power of low-precision parameters to the effectiveness of a policy?** During our QPD experiments, we first observed that training a student with randomly initialised parameters with 8-bit integer precision directly was not feasible, with a very noisy loss that did not improve after the first epoch, and an average return that was comparable to a random agent. In this sense, the detrimental effect of low-precision parameters on the effectiveness of a policy is significant, with more advanced methods needed to even start the training process.

   Fortunately, we were able to mitigate these instabilities through our QPD method, which provides a smoother transition from high-to-low precision parameters for more stable optimisation, in addition to some further stability improvements. After overcoming these initial hurdles, we found that the performance of the low-precision students in terms of the obtained average return was comparable to their high-precision counterparts, with the exception of the smallest two student architectures. This suggests that the reduced representational power of low-precision parameters is not inherently detrimental to the effectiveness of a policy, but that the optimisation process is more challenging and requires additional consideration. For

most students, the impact of halving the number of parameters was more significant than the four times reduction in precision, further supporting this conclusion.

We observed a clear tipping point at a 27x reduction in parameter count, where the combination of having reduced representational power through both quantization and distillation resulted in the start of a steep decline in the average return for our actor-critic teachers. In general, we conclude that low-precision agents can still be equally effective, so long as the reduced representational power is compensated for by a sufficient number of parameters and the optimisation process is carefully managed.

4. **Is the stochasticity of a policy accurately preserved by compression?** After some initial experiments with actor-critic models in Chapter 3, we found that the stochastic behaviour of the teacher policy was not accurately preserved by traditional PD, which scales the output distribution using a temperature $\tau < 1$ in Equation 2.26 such that all secondary knowledge about actions that did not have the highest probability is completely lost.

   By inspecting how the choice of temperature τ affected the action distribution transferred to the student, we found that by smoothing the policy with a temperature $\tau \in [1, 5]$, we could better maintain the stochastic behaviour and transfer more informative additional knowledge on viable alternative trajectories. In the context of continuous action spaces, we further showed how maintaining the entropy of the original policy was integral to the success of the distillation process, and methods that were less accurate in doing so suffered from significant performance hits.

   The inability to accurately preserve the stochasticity in a low-precision representation might be one of the contributing factors why actor-critic models performed worse than using the deterministic DQN teacher for smaller students in our QPD experiments in 4. But given that this wasn't an issue for students with a sufficient number of parameters, we can conclude that this is not an inherent limitation of quantization.

   Even more important than accurately preserving the stochasticity of the teacher policy, however, is ensuring that the student could learn from a wide distribution of state transitions to improve generalisation. A stochastic control policy can provide native on-policy exploration that expands the state space coverage, using transitions that are still directly relevant for the task. Alternatives such as ϵ-greedy exploration can push the agent in a direction where the teacher itself is not knowledgeable and would therefore provide an inaccurate learning signal.

   Our experiments further indicate that a student-driven control policy is essential for continuous control tasks, to reduce the distribution shift between the states encountered during training and testing. In this setting, having the student learn stochastic behaviour is important for exploration to increase the robustness and generalisation of the learned policy.

   In summary, the policy stochasticity may not always be accurately preserved, but through deliberate effort it can be maintained, and doing so often leads to more robust and effective compressed policies.

5. **How does pruning affect a model's ability to be fine-tuned and adapt to new tasks?** We evaluated this scenario in Chapter 7 and discovered that the adaptability of a model was negatively impacted after applying only a single iteration of the magnitude-based IMP pruning method. Removing just 20% of the parameters resulted in an immediate drop in accuracy when then trained to classify new subclasses or entirely new classes, compared to performing the same adaptation with the original full-sized model. After this initial drop, the accuracy on the new task continued to decline at a more steady pace, in line with the performance degradation on the original task after pruning for additional iterations.

   When we instead use the LRP-based pruning method, we did not observe this immediate drop in accuracy, but a similar steady decline in accuracy for all iterations. This results in an overall better adaptability for any given size, with the accuracy curve on the new task maintained at a higher level in general.

   We further explored improving the adaptability of the pruned models by training them on additional knowledge that was not directly relevant to the original task, but could encourage the model to learn more general features that would not be lost through pruning. This makes it more difficult to identify redundant neurons for pruning, however, since more of the network is actively used to encode the additional knowledge. To address this, we used LRP with examples of our main task to identify the network features that are actually relevant, so we can prune the model while maintaining the general features that allow it to better adapt to new data on the edge.

   Pre-training on this additional knowledge proved to be beneficial for both pruning methods, but the improvement was more significant with the LRP-based method. Using IMP, we observed a similar increase in accuracy on the new task, but only for the first few iterations, after which the accuracy curve started to decline to a similar level as our results without pre-training. The LRP-based method, on the other hand, maintained a higher accuracy throughout the entire pruning process, with a clear improvement still noticeable for the smallest model.

   In conclusion, the adaptability of a model is negatively impacted by pruning, but this can be mitigated by using a more informed pre-training and task-relevant pruning method that maintains the general features of the model.

6. **Can we improve efficiency by reducing the number of active decisions required to complete a task?** We explored this question in Chapter 6 and found that our temporal distillation method was able to significantly reduce the number of decisions required to complete the task by predicting a sequence of identical actions with a single decision. Since the frequency of action changes can vary greatly within the same task, this effectively decomposes the solution into predicting the points in time when a different action is necessary, a form of temporal abstraction that requires planning.

   Our experiments demonstrated that a student could learn this auxiliary planning task based on examples of repeated actions in the trajectories of an existing teacher model that does not have this ability. Simply predicting several actions at once is not sufficient, however, if those actions are not productive. Results on two embodied grid-world environments confirmed that our students were able to complete the task in up to 462% fewer decisions than the teacher, while still achieving a similar or better average return.

As this method was based on PD, it can be combined with our other methods to achieve the best of both worlds: a highly efficient model that requires fewer decisions to complete the task, while still maintaining a high level of effectiveness. It turns out that the effects of temporal distillation in terms of runtime performance were even more significant than the reduction in parameter count on most devices. The impact of the reduced model size is device-dependent, resulting in a trade-off between fewer computations required for smaller models and more accurate repeat values predicted by larger ones.

9.2 Future Work

As with any research, there are always more questions to be answered and potential routes towards improvements. By no means is the work presented in this thesis exhaustive, and there are many directions in which it could be extended. In this section, we will discuss some of the most promising avenues for future work that deserve further investigation.

9.2.1 Policy Quantization

Our QPD algorithm proved to be a powerful method for quantizing DRL models to an 8-bit integer representation, which is a reasonable target for most low-precision edge devices. Since the quantization itself did not have a significant impact on the effectiveness of the policy, we believe that this method could be further extended to even lower precision representations, such as the 4-bit, 2-bit, or even binary representations supported by some devices. It remains to be seen how well this would work out of the box, and which challenges and corresponding solutions would arise from it.

Our focus in this work was on the quantization of policies for discrete action spaces, where the models only need to model a relative preference of one action over another. Many real-world tasks, particularly those requiring efficient DRL agents on the edge, require continuous control, as explored in Chapter 5. The QPD method is not inherently incompatible with continuous actions, but relying on low-precision parameters to represent a more precise continuous value might be more problematic. Some preliminary testing on simple environments showed promising results, but a deeper investigation into the impact of precision on these types of models is needed.

9.2.2 Policy Pruning

In Chapter 7, we developed a method for pruning DRL models that maintains the general features of the model. This was our only contribution outside a DRL context, but it was always meant as a precursor to the introduction of knowledge-based policy pruning. Adaptability is an important property of DRL models, which inherently chase a non-stationary target during training. The target should stabilise at the end of training when converging on a particular policy, but it shifts again due to the change in behaviour caused by pruning.

Inspired by the same concept of learning additional knowledge, followed by pruning based on examples of task-relevant data, we envision the development of the following novel methods:

Pruning Redundant Exploration: Observations actually encountered during evaluation could serve as examples to identify redundant exploration knowledge for pruning. The motivation behind this is similar to why PD can be so effective. During training, the model learns an explicit or implicit value of the actions and states it encounters while exploring the environment, but much of this knowledge is redundant after converging on the final policy. In PD, we transfer only the most relevant knowledge to the student, but in this case, we would prune the redundant knowledge from the teacher.

Pre-trained Feature Extractors: DRL models could use pre-trained feature extractors as part of their architecture to increase sample efficiency during training. This would be particularly useful in embodied object-goal navigation, where it could first learn a more general concept of what certain objects look like, so a more diverse set of instances can be recognised during deployment than what is available in the training environment, which focuses more on the navigational aspect. To maintain the general features of the model, but increase the efficiency for edge deployment, we would prune the model based on the relevance of these features to the actual task.

Pruning Multi-Task Students: An important feature of PD that we have not explored in this thesis is the ability for multiple teachers to be distilled into a single student for multi-task DRL [115]. This is generally not very useful in a compression context, as the student would need to be larger to accommodate the additional knowledge, but it could be used to learn more general strategies that can be applied to a wider range of tasks. Then, the model could be pruned based on the relevance of these strategies to the current task, to maintain better generalisation without sacrificing efficiency.

9.2.3 Fine-tuning and Adaptation at the Edge

Beyond our experiments on the adaptation of classifiers in Chapter 7, and the opportunities for knowledge-based policy pruning in the previous section, continuing to train DRL policies after compression remains mostly unexplored. This could be done simply to regain some of the performance lost during compression, but also to adapt the model to new tasks or changes in the environment at the edge.

In the original PD method, with a DQN teacher, the action-value function was not accurately preserved during the distillation process. This is not an issue when simply deterministically selecting the action with the highest value, but it becomes a problem when trying to further refine these values during fine-tuning, as the original meaning of these values has been completely lost.

Our actor-critic distillation method, on the other hand, does not suffer from this issue. The actor remains a probability vector over the actions, and we introduced an auxiliary component for training a critic head based on a Huber-loss, which should accurately transfer the state-value function of the teacher. With these two components preserved,

the student should be compatible with the original A2C or PPO algorithms for further training.

The same is true for our distillation method for continuous action spaces based on the KL-divergence, although a critic would also have to be distilled, analogous to our work in Chapter 3. We already established some insights into the adaptability of pruned models in Chapter 7, but a similar investigation should be done after distillation. Since pruning generally maintains the overall architecture and output distributions of the original model, it is also a prime target for further research in a DRL adaptation context.

9.2.4 Temporal Abstraction

We explore the concept of temporal abstraction for increased efficiency in Chapter 6, by predicting a sequence of identical actions with a single decision by observing examples of when it is safe to do so in existing teacher behaviour. This encourages the student to perform as few decisions as possible, while first prioritising learning a policy that obtains the highest possible return in the teacher.

While this approach already works well in practice, with up to 462% fewer decisions required to complete the task, we believe an even lower decision rate could be achieved by training the teacher with a form of energy-regularised return. This would be similar to the entropy-regularised return of SAC, but with the dual objective of obtaining the highest possible return while minimising action changes.

Sometimes two trajectories can be equally optimal, requiring the same number of actions to complete the task and yielding the same reward, but one is more efficient by requiring fewer decisions. Introducing this additional objective would favour the more efficient trajectory and provide a more informative signal for the student to learn from.

Our initial investigations into using temporal abstraction stemmed from an HRL (Hierarchical Reinforcement Learning) perspective, where a student would be able to learn higher-level actions that are composed of patterns of lower-level actions that occur frequently in the teacher's behaviour. When actually looking at the most frequent patterns, however, we found that they were all composed of the same action repeated multiple times, leading to a shift towards our current approach. We do still believe that other such forms of temporal abstraction could be viable, but with efficiency as the primary goal, one would need to carefully consider the trade-offs between the additional complexity and the potential benefits.

9.2.5 Advanced Architectures

Our current efforts have been focused on the most common types of model-free DRL models, such as the value-based DQN, actor-critic (A2C and PPO), SAC for continuous action spaces, and RNN-based architectures for embodied navigation tasks. While these are powerful techniques, a wide range of other DRL architectures could benefit from our methods as well. We highlight some of the more advanced architectures to explore compression for, but this list is by no means exhaustive.

An example is model-based DRL, where a model of the environment is learned and used to plan ahead [64]. This process can be computationally expensive, as the learned simulation of the environment needs to be consulted many times to make a single decision. Compressing this model using similar techniques based on distillation, pruning, and quantization could lead to a significant increase in efficiency.

Another example is HRL, where a high-level controller focuses on long-term planning and lower levels handle immediate actions [58]. One high-level controller might have multiple lower-level controllers to choose from, each with their own specialisations in sub-tasks. Each of these controllers could be compressed separately, or a more complex method could take the entire hierarchy into account and even restructure it entirely.

In multi-agent DRL, multiple agents learn to cooperate or compete to solve a task together [132]. These agents can either be trained independently in a decentralised way, or even share a common policy and value function through a centralised algorithm. Compression could therefore also take place in a decentralised or centralised manner, depending on the architecture.

9.2.6 Combining our Methods

With the exception of the first two chapters, which are closely interconnected, our other contributions were primarily developed independently to better study their individual impact and potential. We nonetheless kept in mind the possibility of combining multiple methods at the same time during their development. For instance, an RNN-based actor-critic student architecture could be trained with a temporal distillation loss that has an auxiliary critic component, while being quantized using our QPD method.

Future research could study how each method interacts with the others and compare the final compression level and effectiveness to the individual methods. This analysis could highlight potential synergies between the methods, as well as uncover challenges that arise from combining them, such as our earlier observations of the issues surrounding the quantization of actor-critic models when going below a parameter count threshold.

9.2.7 Reducing Environment Interactions

The experiments in this thesis were all conducted in virtual simulated environments, which allowed us to easily train and evaluate a wide range of models to explore and validate new methods and ideas. Here, each interaction with the environment is relatively cheap, and nothing can go catastrophically wrong if the agent makes a mistake. In real-world applications, however, each interaction with the environment can be costly in terms of time and resources. In particular, student-driven policy distillation requires a whole new set of interactions to be collected using the student policy in addition to the initial teacher training. A promising direction for future work would therefore be to explore how to reduce the number of interactions required to train a compressed model, while still maintaining the same level of effectiveness.

This relates to the Sim2Real problem, where a DRL model trained in a simulated environment does not generalise well to the real world [116]. We hypothesise that policy

distillation could be used to bridge this gap, by training a student model in the real world using the teacher's virtual behaviour as a guide. Jain et al. [61] previously showed that policy distillation can be used for domain adaptation, where a teacher was trained using observations from a top-down map perspective, with a student learning to imitate the same behaviour from a first-person embodied perspective. This demonstrates that students can learn an effective policy from different observations of the same environment, even when the teacher is trained in a different domain, as long as same the action distribution is valid for both.

A similar approach could therefore be used to transfer knowledge from a simulated teacher to a student that receives real-world observations, but this relies on the existence of a digital twin with an accurate translation of the transition function. Although this would still require physical interactions when training the student, sample efficiency should be higher when learning directly from a teacher that already has a good understanding of the (simulated) environment than training from scratch. Moreover, using a teacher-driven or hybrid setup, the student could be trained in a more controlled manner without risking catastrophic failures by performing untrained actions in the real world.

9.2.8 Analysis of Compression Effects

Throughout this thesis, we studied various properties of our compressed models, including their runtime performance, stochasticity, generalization, repetitiveness, adaptability, latent structure, and average effectiveness. Though comprehensive, this analysis does not yet paint the full picture of the effects of compression on DRL models. Other properties might shine a light on capabilities that are lost or improved through compression.

For instance, in some use cases, the minimal return is more important than the average, as this could indicate model robustness and how well it handles unexpected situations. A related property is consistency, where you might want to ensure the model behaves similarly across different runs, with limited variance in the results.

Safety is another important aspect in real-world applications, where the model should not only be effective, but also safe to use. Especially in the context of Temporal Distillation introduced in Chapter 6, where the model is trained to predict when a new action will be needed ahead of time, a sudden unexpected event could lead to a dangerous situation if this occurs in the middle of a long pre-determined action sequence. Meanwhile, for teacher-driven distillation, it might be necessary to fill the replay memory with examples of potential mistakes and not just the most optimal trajectories where everything goes perfectly according to plan, so the student can learn how to recover from similar scenarios.

Investigating these additional properties is a next step towards better understanding the effects of compression on DRL models, and how new methods can be developed to improve them.

9.3 Final Remarks

Throughout this thesis, we have demonstrated that deployment of complex DRL models on resource-constrained edge devices is not only feasible, but can be achieved without sacrificing policy effectiveness. By developing specialised compression methods that leverage the unique characteristics of DRL models, we have shown that compression ratios of up to a thousand times are achievable in extreme cases, while maintaining or even improving performance in many.

We established that auxiliary knowledge extracted from teacher models, such as state-value predictions and temporal sequences, can significantly enhance the compression process. This knowledge enables student models to develop more informative internal representations, leading to better performance with fewer parameters. We also demonstrated how the challenges of low-precision quantization can be overcome through careful optimisation, with 8-bit integer models performing comparably to their full-precision counterparts in most cases. Finally, we showed that through temporal abstraction, our temporal distillation method can substantially decrease the computational cost associated with executing DRL policies, achieving the same effectiveness with up to 464% fewer decisions.

The methods presented here are not just theoretical contributions, but practical solutions to real-world challenges. We showcased this on the compression of a data-driven NF replica scaler and a photo-realistic embodied navigation task, alongside numerous runtime performance benchmarks. These benchmarks demonstrated a real-world potential decrease in the average time it takes to perform an action by up to 1487 times. This was accomplished by decreasing both the number of parameters in a model and its architectural complexity, while other experiments have established further improvements by lowering the precision of the parameters and predicting entire sequences of actions with a single decision.

In conclusion, this thesis has made significant progress toward bridging the gap between the computational demands of intelligent DRL models and the constraints of low-power edge deployment. Through our contributions, we have shown that compression strategies tailored to the intricacies of DRL can unlock new possibilities for the deployment of applications in embodied autonomous robotics, network management systems, and other resource-constrained scenarios where real-time decision-making is crucial, without compromising on effectiveness.

Bibliography

[1] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. "A learning algorithm for Boltzmann machines". In: *Cognitive science* 9.1 (1985), pp. 147–169.

[2] Mohammed Alhartomi et al. "Enhancing Sustainable Edge Computing Offloading via Renewable Prediction for Energy Harvesting". In: *IEEE Access* 12 (2024), pp. 74011–74023. DOI: 10.1109/ACCESS.2024.3404222.

[3] Amazon Web Services. *Amazon EC2 G4 Instances*. https://aws.amazon.com/ec2/instance-types/g4/. [Online; accessed 28-September-2024]. 2024.

[4] AMD. *AMD Ryzen™ AI 300 Series Processors*. https://www.amd.com/en/partner/articles/ryzen-ai-300-series-processors.html. (Visited on 09/17/2024).

[5] Analog Devices. *MAX78000*. https://www.analog.com/en/products/max78000.html. (Visited on 09/17/2024).

[6] Arduino. *UNO R3: Tech Specs*. https://docs.arduino.cc/hardware/uno-rev3/#tech-specs. [Online; accessed 14-July-2024]. 2012.

[7] Thomas Avé et al. "Policy Compression for Low-Power Intelligent Scaling in Software-Based Network Architectures". In: *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. 2024, pp. 1–7. DOI: 10.1109/NOMS59830.2024.10575377.

[8] Thomas Avé et al. "Quantization-aware Policy Distillation (QPD)". In: *Deep Reinforcement Learning Workshop NeurIPS 2022*. 2022.

[9] Ahmad Taher Azar et al. "Drone Deep Reinforcement Learning: A Review". In: *Electronics* 10.9 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10090999. URL: https://www.mdpi.com/2079-9292/10/9/999.

[10] Sebastian Bach et al. "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation". In: *PLOS ONE* 10.7 (July 2015), pp. 1–46. DOI: 10.1371/journal.pone.0130140. URL: https://doi.org/10.1371/journal.pone.0130140.

[11] Dieter Balemans et al. "Resource efficient sensor fusion by knowledge-based network pruning". In: *Internet of Things* 11 (2020), p. 100231. ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2020.100231. URL: https://www.sciencedirect.com/science/article/pii/S2542660520300640.

[12] Gabriel Barth-Maron et al. "Distributed Distributional Deterministic Policy Gradients". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=SyZipzbCb.

[13] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[14] Richard Bellman. "On the theory of dynamic programming". In: *Proceedings of the national Academy of Sciences* 38.8 (1952), pp. 716–719.

[15] Glen Berseth et al. "Progressive Reinforcement Learning with Distillation for Multi-Skilled Motion Control". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=B13njo1R-.

[16] Andrea Bertolini et al. "Power output optimization of electric vehicles smart charging hubs using deep reinforcement learning". In: *Expert Systems with Applications* 201 (2022), p. 116995.

[17] André Biedenkapp et al. *TempoRL: Learning when to act*. PMLR, 2021.

[18] Johan Björck et al. "Low-Precision Reinforcement Learning: Running Soft Actor-Critic in Half Precision". In: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 980–991. URL: http://proceedings.mlr.press/v139/bjorck21a.html.

[19] Davis W. Blalock et al. "What is the State of Neural Network Pruning?" In: *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. Ed. by Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze. mlsys.org, 2020. URL: https://proceedings.mlsys.org/book/296.pdf.

[20] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[21] Fanyu Bu and Xin Wang. "A smart agriculture IoT system based on deep reinforcement learning". In: *Future Generation Computer Systems* 99 (2019), pp. 500–507.

[22] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. "Model compression". In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2006, pp. 535–541.

[23] Yu Cheng et al. "Model compression and acceleration for deep neural networks: The principles, progress, and challenges". In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 126–136.

[24] Maxime Chevalier-Boisvert et al. "Minigrid & Miniworld: Modular & Customizable Reinforcement Learning Environments for Goal-Oriented Tasks". In: *CoRR* abs/2306.13831 (2023).

[25] Kyunghyun Cho. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[26] Wojciech M. Czarnecki et al. "Distilling Policy Distillation". In: *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*. Ed. by Kamalika Chaudhuri and Masashi Sugiyama. Vol. 89. Proceedings of Machine Learning Research. PMLR, 2019, pp. 1331–1340. URL: http://proceedings.mlr.press/v89/czarnecki19a.html.

[27] Will Dabney et al. "Distributional reinforcement learning with quantile regression". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.

[28] William Dally. *High-Performance Hardware for Machine Learning*. https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf. 2015.

[29] Lei Deng et al. "Model compression and hardware acceleration for neural networks: A comprehensive survey". In: *Proceedings of the IEEE* 108.4 (2020), pp. 485–532.

[30] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/V1/N19-1423. URL: https://doi.org/10.18653/v1/n19-1423.

[31] Digital Trends. *Nvidia says falling GPU prices are 'a story of the past'*. https://www.digitaltrends.com/computing/nvidia-says-falling-gpu-prices-are-over/. [Online; accessed 28-September-2024]. 2022.

[32] Jiafei Duan et al. "A Survey of Embodied AI: From Simulators to Research Tasks". In: *IEEE Trans. Emerg. Top. Comput. Intell.* 6.2 (2022), pp. 230–244. DOI: 10.1109/TETCI.2022.3141105. URL: https://doi.org/10.1109/TETCI.2022.3141105.

[33] Yan Duan et al. "Benchmarking Deep Reinforcement Learning for Continuous Control". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, June 2016, pp. 1329–1338. URL: https://proceedings.mlr.press/v48/duan16.html.

[34] Bardienus P. Duisterhof et al. *Tiny Robot Learning (tinyRL) for Source Seeking on a Nano Quadcopter*. 2021. DOI: 10.1109/ICRA48506.2021.9561590.

[35] EDGE AI FOUNDATION. *EDGE AI FOUNDATION*. https://www.edgeaifoundation.org/. (Visited on 11/27/2024).

[36] ETSI. *Autonomous Networks, supporting tomorrow's ICT business*. White Paper 40. 1st Edition. Oct. 2020. URL: https://www.etsi.org/images/files/ETSIWhitePapers/etsi-wp-40-Autonomous-networks.pdf.

[37] Jonathan Frankle and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=rJl-b3RcF7.

[38] Scott Fujimoto, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1582–1591. URL: http://proceedings.mlr.press/v80/fujimoto18a.html.

[39] Amir Gholami et al. "A Survey of Quantization Methods for Efficient Neural Network Inference". In: *CoRR* abs/2103.13630 (2021). arXiv: 2103.13630. URL: https://arxiv.org/abs/2103.13630.

[40] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[41] Mitchell A. Gordon, Kevin Duh, and Nicholas Andrews. "Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning". In: *Proceedings of the 5th Workshop on Representation Learning for NLP, RepL4NLP@ACL 2020, Online, July 9, 2020*. Ed. by Spandana Gella et al. Association for Computational Linguistics, 2020, pp. 143–155. DOI: 10.18653/v1/2020.repl4nlp-1.18. URL: https://doi.org/10.18653/v1/2020.repl4nlp-1.18.

[42] Alex Graves and Alex Graves. "Long short-term memory". In: *Supervised sequence labelling with recurrent neural networks* (2012), pp. 37–45.

[43] Sam Green, Craig M. Vineyard, and Çetin Kaya Koç. "Distillation Strategies for Proximal Policy Optimization". In: *CoRR* abs/1901.08128 (2019). arXiv: 1901.08128. URL: http://arxiv.org/abs/1901.08128.

[44] Ferran Gebellí Guinjoan et al. "A multi-modal ai approach for agvs: A case study on warehouse automated inventory". In: *Proc. of the 9th Int. Conf. on Autonomic and Autonomous Systems (ICAS)*. Vol. 15. 2023, p. 34.

[45] Manas Gupta et al. "Learning to prune deep neural networks via reinforcement learning". In: *arXiv preprint arXiv:2007.04756* (2020).

[46] *Gymnasium*. https://github.com/Farama-Foundation/Gymnasium. 2022.

[47] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1856–1865. URL: http://proceedings.mlr.press/v80/haarnoja18b.html.

[48] Babak Hassibi and David G. Stork. "Second Order Derivatives for Network Pruning: Optimal Brain Surgeon". In: *Advances in Neural Information Processing Systems 5, [NIPS Conference, Denver, Colorado, USA, November 30 - December 3, 1992]*. Ed. by Stephen Jose Hanson, Jack D. Cowan, and C. Lee Giles. Morgan Kaufmann, 1992, pp. 164–171. URL: http://papers.nips.cc/paper/647-second-order-derivatives-for-network-pruning-optimal-brain-surgeon.

[49] Kaiming He et al. "Identity Mappings in Deep Residual Networks". In: *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*. Ed. by Bastian Leibe et al. Vol. 9908. Lecture Notes in Computer Science. Springer, 2016, pp. 630–645. DOI: 10.1007/978-3-319-46493-0_38. URL: https://doi.org/10.1007/978-3-319-46493-0%5C_38.

[50] Lin He, Lishan Li, and Ying Liu. "Towards chain-aware scaling detection in nfv with reinforcement learning". In: *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE. 2021, pp. 1–10.

[51] Geoffrey Hinton. *Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude*. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Coursera: Neural Networks for Machine Learning. 2012.

[52] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. "Distilling the Knowledge in a Neural Network". In: *CoRR* abs/1503.02531 (2015). arXiv: 1503.02531. URL: http://arxiv.org/abs/1503.02531.

[53] Shengyi Huang et al. "A2C is a special case of PPO". In: *CoRR* abs/2205.09123 (2022). DOI: 10.48550/ARXIV.2205.09123. arXiv: 2205.09123. URL: https://doi.org/10.48550/arXiv.2205.09123.

[54] Itay Hubara et al. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *J. Mach. Learn. Res.* 18 (2017), 187:1–187:30. URL: https://jmlr.org/papers/v18/16-456.html.

[55] Peter J Huber. "Robust estimation of a location parameter". In: *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 492–518.

[56] Hugging Face Inc. *sb3/ppo-MiniGrid-FourRooms-v0*. https://huggingface.co/sb3/ppo-MiniGrid-FourRooms-v0. [Online; accessed 01-August-2024].

[57] Hugging Face Inc. *sb3/ppo-MiniGrid-Unlock-v0*. https://huggingface.co/sb3/ppo-MiniGrid-Unlock-v0. [Online; accessed 01-August-2024].

[58] Matthias Hutsebaut-Buysse, Kevin Mets, and Steven Latré. "Hierarchical Reinforcement Learning: A Survey and Open Research Challenges". In: *Mach. Learn. Knowl. Extr.* 4.1 (2022), pp. 172–221. DOI: 10.3390/MAKE4010009. URL: https://doi.org/10.3390/make4010009.

[59] Matthias Hutsebaut-Buysse et al. "Directed Real-World Learned Exploration". In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2023, pp. 5227–5234.

[60] Intel Corporation. *Intel® Core™ Ultra 9 Processor 288V*. https://ark.intel.com/content/www/us/en/ark/products/240961/intel-core-ultra-9-processor-288v-12m-cache-up-to-5-10-ghz.html. (Visited on 09/17/2024).

[61] Unnat Jain et al. "GridToPix: Training Embodied Agents with Minimal Supervision". In: *CoRR* abs/2105.00931 (2021). arXiv: 2105.00931. URL: https://arxiv.org/abs/2105.00931.

[62] Siavash Barqi Janiar and Vahid Pourahmadi. "Deep-Reinforcement Learning for Fair Distributed Dynamic Spectrum Access in Wireless Networks". In: *2021 IEEE 18th Annual Consumer Communications and Networking Conference (CCNC)*. 2021, pp. 1–4. DOI: 10.1109/CCNC49032.2021.9369536.

[63] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial intelligence* 101.1-2 (1998), pp. 99–134.

[64] Lukasz Kaiser et al. "Model Based Reinforcement Learning for Atari". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=S1xCPJHtDB.

[65] Kalle Raiskila. *Onnx2c*. https://github.com/kraiskil/onnx2c. [Online; accessed 01-August-2024].

[66] Shivaram Kalyanakrishnan et al. "An analysis of frame-skipping in reinforcement learning". In: *arXiv preprint arXiv:2102.03718* (2021).

[67] Abeer Abdel Khaleq and Ilkyeun Ra. "Intelligent autoscaling of microservices in the cloud for real-time applications". In: *IEEE Access* 9 (2021), pp. 35464–35476.

[68] Donghyun Kim et al. "A Broad Study of Pre-training for Domain Generalization and Adaptation". In: *Computer Vision - ECCV 2022 - 17th European Conference, Tel Aviv, Israel, October 23-27, 2022, Proceedings, Part XXXIII*. Ed. by Shai Avidan et al. Vol. 13693. Lecture Notes in Computer Science. Springer, 2022, pp. 621–638. DOI: 10.1007/978-3-031-19827-4_36. URL: https://doi.org/10.1007/978-3-031-19827-4%5C_36.

[69] Jangho Kim et al. "QKD: Quantization-aware Knowledge Distillation". In: *CoRR* abs/1911.12491 (2019). arXiv: 1911.12491. URL: http://arxiv.org/abs/1911.12491.

[70] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[71] Jens Kober, J Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274.

[72] Vijay R. Konda and John N. Tsitsiklis. "Actor-Critic Algorithms". In: *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*. Ed. by Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller. The MIT Press, 1999, pp. 1008–1014. URL: http://papers.nips.cc/paper/1786-actor-critic-algorithms.

[73] Raghuraman Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". In: *arXiv preprint arXiv:1806.08342* (2018).

[74] Srivatsan Krishnan et al. "Quantized Reinforcement Learning (QUARL)". In: *CoRR* abs/1910.01055 (2019). arXiv: 1910.01055. URL: http://arxiv.org/abs/1910.01055.

[75] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. Ed. by Peter L. Bartlett et al. 2012, pp. 1106–1114.

[76] HP Künzi. "Dynamic Programming and Markov Processes-Howard, Ronald A." In: *Metrika* 5 (1962), pp. 220–221.

[77] Kwei-Herng Lai et al. "Dual Policy Distillation". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. Ed. by Christian Bessiere. ijcai.org, 2020, pp. 3146–3152. DOI: 10.24963/ijcai.2020/435. URL: https://doi.org/10.24963/ijcai.2020/435.

[78] Aravind Lakshminarayanan, Sahil Sharma, and Balaraman Ravindran. "Dynamic Action Repetition for Deep Reinforcement Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31.1 (Feb. 2017). DOI: 10.1609/aaai.v31i1.10918. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10918.

[79] Yann LeCun, John Denker, and Sara Solla. "Optimal brain damage". In: *Advances in neural information processing systems* 2 (1989).

[80] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proc. IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791. URL: https://doi.org/10.1109/5.726791.

[81] Kyuho J Lee. "Architecture of neural processing unit for deep neural networks". In: *Advances in Computers*. Vol. 122. Elsevier, 2021, pp. 217–245.

[82] Su Young Lee, Choi Sungik, and Sae-Young Chung. "Sample-efficient deep reinforcement learning via episodic backward update". In: *Advances in neural information processing systems* 32 (2019).

[83] Lei Lei et al. "Deep reinforcement learning for autonomous internet of things: Model, applications and challenges". In: *IEEE Communications Surveys & Tutorials* 22.3 (2020), pp. 1722–1760.

[84] Dor Livne and Kobi Cohen. "PoPS: Policy Pruning and Shrinking for Deep Reinforcement Learning". In: *IEEE J. Sel. Top. Signal Process.* 14.4 (2020), pp. 789–801. DOI: 10.1109/JSTSP.2020.2967566. URL: https://doi.org/10.1109/JSTSP.2020.2967566.

[85] *LoCoBot: An Open Source Low Cost Robot*. http://www.locobot.org/. 2019.

[86] Sridhar Mahadevan. "Average reward reinforcement learning: Foundations, algorithms, and empirical results". In: *Machine learning* 22.1 (1996), pp. 159–195.

[87] Leland McInnes, John Healy, and James Melville. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. 2018. DOI: 10.48550/ARXIV.1802.03426. URL: https://arxiv.org/abs/1802.03426.

[88] Gaurav Menghani. "Efficient deep learning: A survey on making deep learning models smaller, faster, and better". In: *ACM Computing Surveys* 55.12 (2023), pp. 1–37.

[89] Microsoft. *ONNX Runtime*. https://github.com/microsoft/onnxruntime. [Online; accessed 01-August-2024].

[90] Asit K. Mishra and Debbie Marr. "Apprentice: Using Knowledge Distillation Techniques To Improve Low-Precision Network Accuracy". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=B1ae1lZRb.

[91] Faris B. Mismar, Brian L. Evans, and Ahmed Alkhateeb. "Deep Reinforcement Learning for 5G Networks: Joint Beamforming, Power Control, and Interference Coordination". In: *IEEE Transactions on Communications* 68.3 (2020), pp. 1581–1592. DOI: 10.1109/TCOMM.2019.2961332.

[92] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 1928–1937. URL: http://proceedings.mlr.press/v48/mniha16.html.

[93] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". en. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14236. URL: http://www.nature.com/articles/nature14236 (visited on 09/10/2021).

[94] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[95] Gordon E Moore. "Cramming more components onto integrated circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[96] Guillem Muñoz et al. "Deep Reinforcement Learning for Drone Delivery". In: *Drones* 3.3 (2019). ISSN: 2504-446X. DOI: 10.3390/drones3030072. URL: https://www.mdpi.com/2504-446X/3/3/72.

[97] Siddharth Mysore et al. "Honey. I Shrunk The Actor: A Case Study on Preserving Performance with Smaller Actors in Actor-Critic RL". In: *2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, August 17-20, 2021*. IEEE, 2021, pp. 1–8. DOI: 10.1109/COG52621.2021.9619008. URL: https://doi.org/10.1109/CoG52621.2021.9619008.

[98] Behnam Neyshabur et al. "Observational overfitting in reinforcement learning". In: *International Conference on Learning Representations, NeurIPS*. 2020.

[99] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[100] NVIDIA Corporation. *NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge*. https://developer.nvidia.com/blog/jetson-tx2-delivers-twice-intelligence-edge/. [Online; accessed 14-July-2024]. 2017.

[101] Johan Obando-Ceron, Aaron Courville, and Pablo Samuel Castro. "In deep reinforcement learning, a pruned network is a good network". In: *arXiv preprint arXiv:2402.12479* (2024).

[102] ONNX. *Open Neural Network Exchange*. https://github.com/onnx/onnx. [Online; accessed 01-August-2024].

[103] M. Orlowski. "CMOS challenges of keeping up with Moore's Law". In: *2005 13th International Conference on Advanced Thermal Processing of Semiconductors*. 2005. DOI: 10.1109/RTP.2005.1613679.

[104] Javier Poyatos et al. "EvoPruneDeepTL: An evolutionary pruning model for transfer learning based deep neural networks". In: *Neural Networks* 158 (2023), pp. 59–82.

[105] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[106] Guanjin Qu et al. "DMRO: A deep meta reinforcement learning-based task offloading framework for edge-cloud computing". In: *IEEE Transactions on Network and Service Management* 18.3 (2021), pp. 3448–3459.

[107] Qualcomm. *Snapdragon X Elite*. https://www.qualcomm.com/products/mobile/snapdragon/laptops-and-tablets/snapdragon-x-elite. (Visited on 09/17/2024).

[108] Antonin Raffin et al. *Stable Baselines3*. https://github.com/DLR-RM/stable-baselines3. 2019.

[109] Roberta Raileanu and Rob Fergus. "Decoupling value and policy for generalization in reinforcement learning". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 8787–8798.

[110] Raspberry Pi Foundation. *Typical power requirements*. https://github.com/raspberrypi/documentation/blob/develop/documentation/asciidoc/computers/raspberry-pi/power-supplies.adoc#typical-power-requirements. [Online; accessed 14-July-2024]. 2024.

[111] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[112] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. "Horizontal and vertical scaling of container-based applications using reinforcement learning". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE. 2019, pp. 329–338.

[113] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[114] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[115] Andrei A. Rusu et al. "Policy Distillation". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1511.06295.

[116] Andrei A. Rusu et al. "Sim-to-Real Robot Learning from Pixels with Progressive Nets". In: *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings*. Vol. 78. Proceedings of Machine Learning Research. PMLR, 2017, pp. 262–270. URL: http://proceedings.mlr.press/v78/rusu17a.html.

[117] José Santos et al. "gym-hpa: Efficient Auto-Scaling via Reinforcement Learning for Complex Microservice-based Applications in Kubernetes". In: *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2023, pp. 1–9.

[118] Manolis Savva et al. "Habitat: A Platform for Embodied AI Research". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019.

[119] Tom Schaul et al. "Prioritized Experience Replay". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1511.05952.

[120] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". en. In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-03051-4. URL: http://www.nature.com/articles/s41586-020-03051-4 (visited on 09/14/2021).

[121] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1506.02438.

[122] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[123] Sahil Sharma, Aravind S. Lakshminarayanan, and Balaraman Ravindran. *Learning to Repeat: Fine Grained Action Repetition for Deep Reinforcement Learning*. 2017. URL: https://openreview.net/forum?id=B1GOWV5eg.

[124] Md Abu Bakar Siddik, Arman Shehabi, and Landon Marston. "The environmental footprint of data centers in the United States". In: *Environmental Research Letters* 16.6 (2021), p. 064017.

[125] Terence Song et al. "Performance evaluation of integrated smart energy solutions through large-scale simulations". In: *IEEE Second International Conference on Smart Grid Communications, SmartGridComm 2011, Brussels, Belgium, October 17-20, 2011*. IEEE, 2011, pp. 37–42. DOI: 10.1109/SMARTGRIDCOMM.2011.6102351. URL: https://doi.org/10.1109/SmartGridComm.2011.6102351.

[126] Paola Soto et al. "Network Intelligence for NFV Scaling in Closed-Loop Architectures". In: *IEEE Communications Magazine* 61.6 (2023), pp. 66–72. DOI: 10.1109/MCOM.001.2200529.

[127] Paola Soto et al. "Towards autonomous VNF auto-scaling using deep reinforcement learning". In: *2021 Eighth International Conference on Software Defined Systems (SDS)*. IEEE. 2021, pp. 01–08.

[128] Shivangi Srivastava et al. "Adaptive Compression-based Lifelong Learning". In: *30th British Machine Vision Conference 2019, BMVC 2019, Cardiff, UK, September 9-12, 2019*. BMVA Press, 2019, p. 153. URL: https://bmvc2019.org/wp-content/uploads/papers/0649-paper.pdf.

[129] Samuel Stanton et al. "Does Knowledge Distillation Really Work?" In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. Ed. by Marc'Aurelio Ranzato et al. 2021, pp. 6906–6919. URL: https://proceedings.neurips.cc/paper/2021/hash/376c6b9ff3bedbbea56751a84fffc10c-Abstract.html.

[130] Alexander L Strehl et al. "PAC model-free reinforcement learning". In: *Proceedings of the 23rd international conference on Machine learning*. 2006, pp. 881–888.

[131] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.

[132] Ming Tan. "Multi-agent reinforcement learning: Independent vs. cooperative agents". In: *Proceedings of the tenth international conference on machine learning*. 1993, pp. 330–337.

[133] Ming Tang and Vincent W.S. Wong. "Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing Systems". In: *IEEE Transactions on Mobile Computing* 21.6 (2022), pp. 1985–1997. DOI: 10.1109/TMC.2020.3036871.

[134] Yunhao Tang and Shipra Agrawal. "Discretizing Continuous Action Space for On-Policy Optimization". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 5981–5988. DOI: 10.1609/AAAI.V34I04.6059. URL: https://doi.org/10.1609/aaai.v34i04.6059.

[135] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[136] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[137] Oriol Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". en. In: *Nature* 575.7782 (Nov. 2019), pp. 350–354. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-019-1724-z. URL: http://www.nature.com/articles/s41586-019-1724-z (visited on 09/14/2021).

[138] DS Vohra, PK Garg, and SK Ghosh. "Power management of drones". In: *International Conference on Unmanned Aerial System in Geomatics*. Springer. 2021, pp. 555–569.

[139] M Mitchell Waldrop. "The chips are down for Moore's law". In: *Nature News* 530.7589 (2016), p. 144.

[140] Jianyu Wang et al. "Edge cloud offloading algorithms: Issues, methods, and perspectives". In: *ACM Computing Surveys (CSUR)* 52.1 (2019), pp. 1–23.

[141] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 1995–2003. URL: http://proceedings.mlr.press/v48/wangf16.html.

[142] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.

[143] Debin Wei, Chuanqi Guo, and Li Yang. "Intelligent Hierarchical Admission Control for Low-Earth Orbit Satellites Based on Deep Reinforcement Learning". In: *Sensors* 23.20 (2023). ISSN: 1424-8220. DOI: 10.3390/s23208470. URL: https://www.mdpi.com/1424-8220/23/20/8470.

[144] Erik Wijmans et al. "DD-PPO: Learning Near-Perfect PointGoal Navigators from 2.5 Billion Frames". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=H1gX8C4YPr.

[145] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8 (1992), pp. 229–256.

[146] Zhiyuan Wu et al. "Survey of knowledge distillation in federated edge learning". In: *arXiv preprint arXiv:2301.05849* (2023).

[147] Dongkuan Xu et al. "Rethinking Network Pruning - under the Pre-train and Fine-tune Paradigm". In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*. Ed. by Kristina Toutanova et al. Association for Computational Linguistics, 2021, pp. 2376–2382. DOI: 10.18653/V1/2021.NAACL-MAIN.188. URL: https://doi.org/10.18653/v1/2021.naacl-main.188.

[148] Zhiyuan Xu et al. "Knowledge Transfer in Multi-Task Deep Reinforcement Learning for Continuous Control". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. 2020.

[149] Meng Xun et al. "Deep Reinforcement Learning for Delay and Energy-Aware Task Scheduling in Edge Clouds". In: *Computer Supported Cooperative Work and Social Computing*. Ed. by Yuqing Sun et al. Singapore: Springer Nature Singapore, 2024, pp. 436–450. ISBN: 978-981-99-9637-7.

[150] Yang Yu. "Towards Sample Efficient Reinforcement Learning." In: *IJCAI*. 2018, pp. 5739–5743.

[151] Chiyuan Zhang et al. "A study on overfitting in deep reinforcement learning". In: *arXiv preprint arXiv:1804.06893* (2018).

[152] Yongchao Zhang and Pengzhan Chen. "Path Planning of a Mobile Robot for a Dynamic Indoor Environment Based on an SAC-LSTM Algorithm". In: *Sensors* 23.24 (2023). ISSN: 1424-8220. DOI: 10.3390/s23249802. URL: https://www.mdpi.com/1424-8220/23/24/9802.

[153] Xiandong Zhao et al. “Linear Symmetric Quantization of Neural Networks for Low-precision Integer Hardware”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=H1lBj2VFPS.

[154] Xiaobo Zhao et al. “Improving the Accuracy-Latency Trade-off of Edge-Cloud Computation Offloading for Deep Learning Services”. In: *2020 IEEE Globecom Workshops (GC Wkshps*. 2020, pp. 1–6. DOI: 10.1109/GCWkshps50303.2020.9367470.

[155] Shuchang Zhou et al. “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients”. In: *CoRR* abs/1606.06160 (2016). arXiv: 1606.06160. URL: http://arxiv.org/abs/1606.06160.

[156] Micah S Ziegler and Jessika E Trancik. “Re-examining rates of lithium-ion battery technology improvement and cost decline”. In: *Energy & Environmental Science* 14.4 (2021), pp. 1635–1651.

[157] Neta Zmora et al. “Neural Network Distiller: A Python Package For DNN Compression Research”. In: (Oct. 2019). URL: https://arxiv.org/abs/1910.12232.