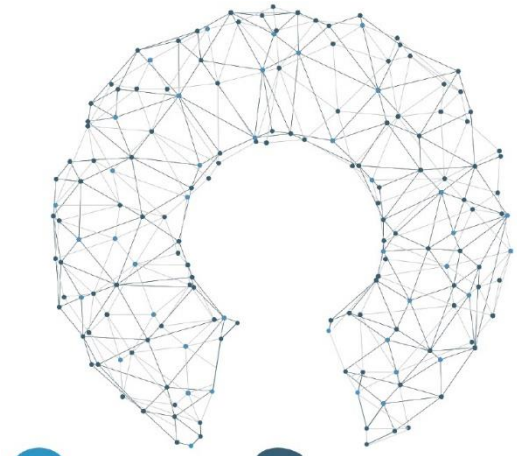


www.openswarm.eu

Call: HORIZON-CL4-2022-DATA-01

Type of action: RIA

Grant agreement: 101093046

Deliverable n°3.2 : Report on Mechanisms for Low-Power Online Learning and Knowledge Transfer

Work Package n°3: Collaborative energy-aware AI

imec

WP Lead: imec

This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101093046.

Funded by the European Union

Document information	
Author(s)	Matthias Hutsebaut-Buysse, Thomas Avé, Wei Wei, Kevin Mets
Reviewers	Brian Coffey
Submission date	31-Oct-2024
Due date	31-Oct-2024
Type	Report
Dissemination level	PU

Document history			
Date	Version	Author(s)	Comments
28-Jun-2024	01	Matthias Hutsebaut-Buysse	preview
07-Oct-2024	02	Thomas Avé	Deliverable

1. Table of contents

2. Executive Summary

In order to develop true collaborative and distributed smart nodes, the sensors on these nodes will need to become “smart sensors”. This means that they do not simply rely the raw observations, but are able to process observations on the devices themselves. When confronted with high-dimensional or complex processing patterns machine learning techniques can be utilized. The scope of this document is providing insights into how current techniques can be utilized to perform on-device ML.

This document is split into three sections, in which we each describe the state of the art, and describe the methods that were developed in order to go beyond the state of the art to solve the challenges defined in the OpenSwarm project.

In the first section we describe how machine learning can be utilized on low-power devices. This is typically done by compressing models, and thus reduce unnecessary parts. We describe such a method that is able to compress Reinforcement Learning models with continuous action spaces. We also propose a method in this section to not only compress the model, but to also compress the amount of decisions that need to be made by an RL policy.

The second part of this documents describes how we can train ML models on-device. Due to the often limited memory capacity of edge devices, it is not always feasible to store all training data locally. Additionally the main question that we answered in our original work is how can we train a base model that is optimally suited to be adapted on device as new data arrives.

The final part of this report specially tackles the problem of learning new tasks, the so called continual learning setting. Within this section we describe a novel method that is capable of efficiently storing a representable data sample in order to not forget about old tasks while learning novel tasks.

3. Introduction

In a typical swarm sensor data is transmitted in its raw form in order to be processed. However, in order to support true collaborative and distributed smart nodes in a large swarm, sensors will need to become smart. Becoming smart in this context means that they will need to be capable of processing the raw sensor observations themselves. If the relationship between raw sensor observations and desired outcomes is straightforward, then a static function can be programmed and embedded in the device. However, in many cases the sensor observations are high dimensional (audio, images, etc.) and the patterns between inputs and outputs cannot be expressed easily. The domain of Machine Learning (ML) offers an alternative workflow. Instead of relying on a human expert to process raw sensor data into a desired compact representation, we could utilize a learning approach and facilitate the nodes to learn how to process the raw sensor observations themselves. To make this approach feasible we need to develop methods that allow to utilize machine learning models on low-power swarm devices, and how we can efficiently update existing methods when new data becomes available.

The goal of this document is to describe the steps we took to enable machine learning (i.e., both training and inference) on ultra-low-power swarm devices deployed in the field, effectively evolving from Tiny Machine Learning (TinyML) to Tiny Online Learning (TinyOL). To scale down the energy consumption for in-the-field online learning by several orders of magnitude, existing and novel neural network compression algorithms, will be described. This allows lightweight learning of complex features, dynamically adapting the trade-off between classification accuracy and energy consumption.

Especially with the recent advancements of large (language) models, considering the energy consumption of AI models both during training and inference has become extremely important if we do not want to put significant additional strain on the energy supply due to the usage of AI applications.

This report consists of three parts. In the first part the concept of TinyML is introduced which resolves around utilizing pre-trained machine learning models on energy-constrained devices. A second part of this report will introduce TinyOL, where AI models that have been deployed on the devices are updated as new data arrives. In the final part of this report, we describe a continual learning framework we developed to take online learning a step further and balance the usage of both old and novel observations to prevent catastrophic forgetting.

4. Papers

Most of the results which are outlined in this report have also been published in the following papers:

- Thomas Avé, Matthias Hutsebaut-Buysse, Kevin Mets, "**Temporal Distillation: Compressing a Policy in Space and Time**" (under review Springer Machine Learning)
- Thomas Avé, Matthias Hutsebaut-Buysse, Wei Wei and Kevin Mets, "**Online Adaptation of Compressed Models by Pre-Training and Task-Relevant Pruning**" The 32nd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN), Bruges, Belgium, 9-11 October 2024
- Thomas Avé, Tom De Schepper, and Kevin Mets. 2024. "**Policy Compression for Intelligent Continuous Control on Low-Power Edge Devices**" *Sensors* 24, no. 15: 4876. https://doi.org/10.3390/s24154876

5. KPIs

To measure the outcome of the work presented in this document a KPI was set on the target average added current draw of a field device due to AI model inference with a target value of <100µA. Furthermore, the MAX78000 microcontroller from Analog Devices was highlighted as the prime candidate for the deployment of the AI model due to its capabilities for hardware acceleration of Convolutional Neural Networks (CNNs). After some initial investigations, we found that a maximal power draw of 2220µA during inference could be expected on the MAX78000 microcontroller at the highest clock frequency of 100MHz and 3.0V, significantly more than the target KPI by an order of magnitude. Additionally, this microcontroller only supports up to 442,000 bytes of weight memory (at 1,2,4 or 8-bit precision).

Note, however, that the KPI target refers to the average current draw over time and not the instantaneous current draw of the device. This is especially important as we only need to run active inference a certain number of times within a given time frame. Therefore, depending on the used power mode, this inference step can be performed in a short time frame, while the device can be in a low-power mode for the remaining time to achieve the target average current draw. A trade-off can further be made between the various power modes of the device, as a higher clock frequency will lead to a higher instantaneous power draw during inference, but also to a shorter duration in which the CNN engine needs to be active.

Nevertheless, the restrictions introduced by this low level of precision, limited weight capacity, and target power draw require the introduction of specialized model compression methods before the AI models developed in T3.3 and T3.4 can be deployed.

To validate the feasibility of reaching the KPI and to find the best balance between the available power modes and the desired inference frequency, we performed a set of quantitative measurements of the current draw when deploying models of varying sizes and complexities on the MAX78000 and performing inference at all supported clock

speeds. These findings can then serve as a target for architectures when developing and evaluating our model compression methods later in this document. An overview of the used models and corresponding measurements is given in the table below.

Parameters / Operations	Power Mode	Current Draw (µA)	Inference Time (ms)
169472 / 8402528	IPO (100 MHz)	24423	2,58
169472 / 8402528	ISO (60 MHz)	18332	4,33
169472 / 8402528	INRO (0.03 MHz)	9683	8990,66
169472 / 8402528	IBRO (7.37 MHz)	10323	34,72
381792 / 18636416	IPO (100 MHz)	20023	3,26
381792 / 18636416	ISO (60 MHz)	15611	5,46
381792 / 18636416	INRO (0.03 MHz)	9287	11197,33
381792 / 18636416	IBRO (7.37 MHz)	10076	43,8
302602 / 36481536	IPO (100 MHz)	25975	4,85
302602 / 36481536	ISO (60 MHz)	19141	8,13
302602 / 36481536	INRO (0.03 MHz)	9351	16454,64
302602 / 36481536	IBRO (7.37 MHz)	10526	65,18
425941 / 17464576	IPO (100 MHz)	29959	3,6
425941 / 17464576	ISO (60 MHz)	21505	6,05
425941 / 17464576	INRO (0.03 MHz)	9253	12377,99
425941 / 17464576	IBRO (7.37 MHz)	10734	48,49
726628 / 43182528	IPO (100 MHz)	21072	4,2
726628 / 43182528	ISO (60 MHz)	16227	7,05
726628 / 43182528	INRO (0.03 MHz)	9316	14347
726628 / 43182528	IBRO (7.37 MHz)	10158	56,51
71158 / 10883968	IPO (100 MHz)	20007	1,48
71158 / 10883968	ISO (60 MHz)	15653	2,48
71158 / 10883968	INRO (0.03 MHz)	9319	5297,62
71158 / 10883968	IBRO (7.37 MHz)	10106	19,9

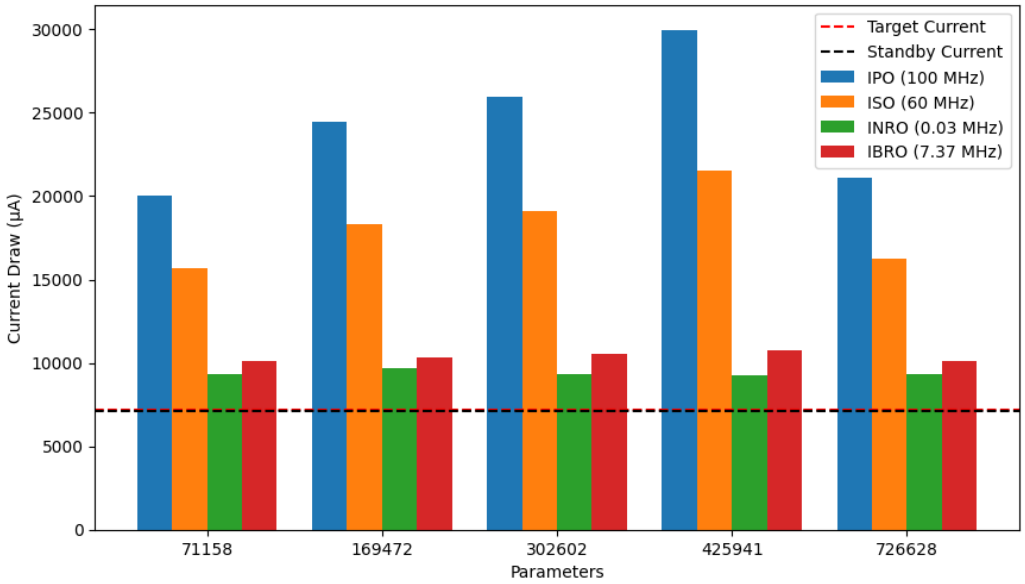


Figure 1: Current draw measured for different model sizes

We represent this visually in the Figure 1 to provide a clear overview of the results and compare them to the target current draw defined by the KPI.

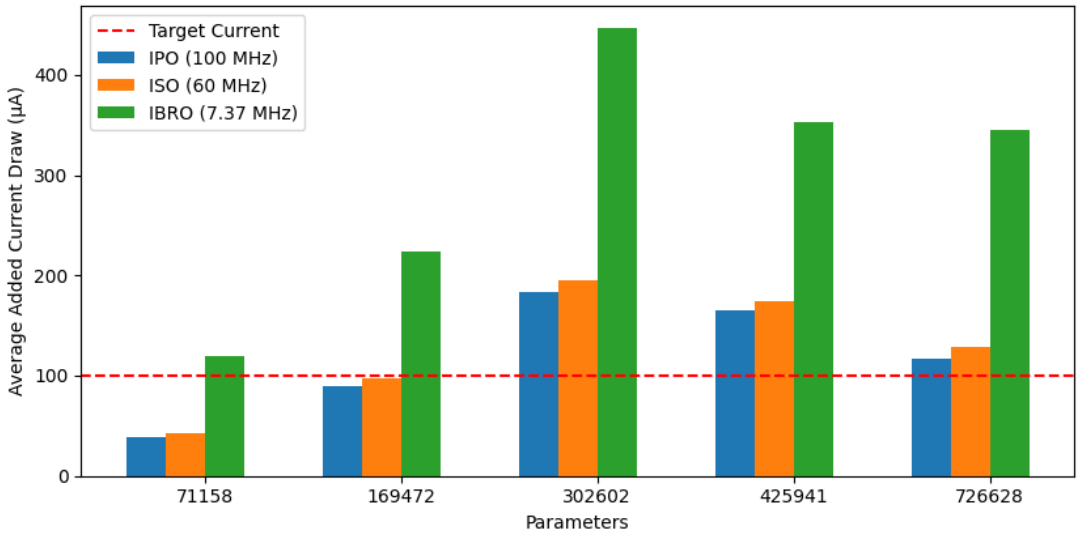


Figure 2: Average added current draw measured for different model sizes

We specifically care about the average added current draw when performing ML inference on top of the base current of the microcontroller. For this base current, we measured a value of 7100µA when the microcontroller is in the lowest possible power state (Standby). Unfortunately, when running continuous inference on the device and

keeping the CNN engine active for the entire duration, the measured current draw was even higher than the initial estimation by an additional order of magnitude. However, for the scenarios outlined in the OpenSwarm project, continuous inference should not be required, and the microcontroller can be put into a low power state between inference operations. For example, at an inference rate of 2Hz, which is already relatively high for the envisioned use cases, we instead observe in Figure 2 the following average added current draw for each model and clock speed.

We did remove the lowest clock speed of 0.03MHz from this figure as this was too slow to reach the target inference rate of 2Hz, while only providing a minor reduction in power consumption. Since we evaluated a wide range of not only model sizes, but also different types of architectures, the models with the highest number of parameters are not necessarily the most complex ones that require the most operations to compute the result, potentially resulting in a lower average current draw than models with fewer parameters. From this figure, we can see that the target KPI of <100µA can be reached at 2Hz when the models are compressed to an architecture that is small and simple enough. Surprisingly, the most efficient clock speed in this deployment configuration is IPO, which draws the highest amount of current, but can perform the inference operation in the shortest amount of time, allowing the microcontroller to return to a low power state faster.

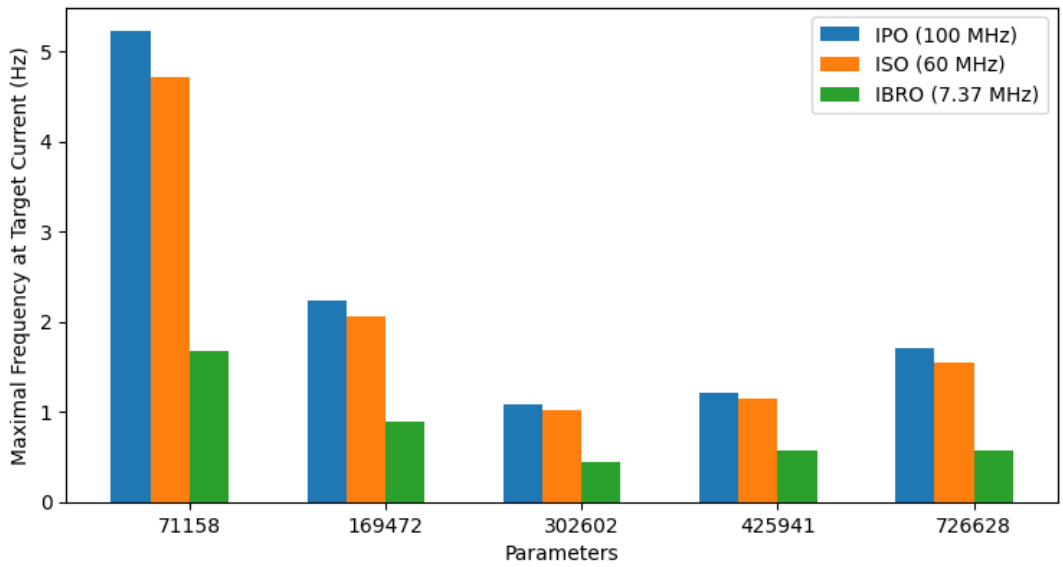


Figure 3: Maximum frequency at target current

Finally, we change our perspective by looking at the highest inference rate that can be achieved for each model size and power mode while still reaching the KPI in Figure 3.

We conclude that all models evaluated here can be deployed on the MAX78000 microcontroller while reaching the target KPI of <100µA at a reasonable inference rate, but that we should ideally aim for a model that is comparable in architecture to the smallest one in the figure above for the largest impact on the average current draw. In the following sections, we take these findings into account and propose various compression methods using both distillation and pruning to obtain models that reach this objective. Furthermore, in the section on "Temporal Distillation", we developed a novel method for intelligently predicting when the next inference operation will be required, allowing the microcontroller to be put into a low power state for longer periods of time.

6. TRL level

Currently we are at TRL3 (experimental proof of concept). Through further efforts in the proof of concepts we aim to increase the TRL-level towards 5, which is the overall target of the OpenSwarm project.

7. Background

7.1. Machine Learning (ML)

In a lot of cases, it is easier to show an example than to in detail describe why this example is a valid solution of the task at hand. However, typically, when working with machines we cannot simply show some examples yet, but we need a detailed set of rules to transform inputs into the desired outcomes.

Within the area of Machine Learning (ML), techniques have been developed that are able to learn the appropriate rules from examples. The typical example that is often used in this context is a digit recognizer which can recognize individual hand-written digits from looking at the pixel values of an input image. Such a system can be obtained by providing a ML system combinations of example inputs (images) and their labels (the digits in the images).

A typical ML pipeline consists of three phases.

1. **Training:** In the first phase a dataset is utilized to "train" the model, so that when provided with a certain input it becomes more likely that the model outputs the correct class label.
2. **Inference:** In the second phase the model is typically deployed in the real-world and is utilized to make decisions based on the provided inputs. During this phase the model is typically frozen, and it is not updated as new data arrives.
3. **Evaluation:** Once we have real-world predictions of the model, it might be possible to evaluate the performance of the model in real-world conditions on data it did not see during training. In a lot of cases such a dataset is often also utilized to evaluate models offline.

Machine Learning techniques are often divided in multiple categories depending on the availability of labeled data:

- **Supervised Learning:** in this setting both example inputs and labeled outputs are available up front. The task consists of learning a mapping function from inputs to outputs.
- **Unsupervised Learning:** in this setting there typically are no labels available. Algorithms are thus not used to predict labels, but they are typically utilized to discover patterns in the dataset itself (e.g., finding clusters of similar data).
- **Semi-supervised Learning:** in this combined setting the dataset does not contains any explicit labels, but a part of the input data is typically utilized as the predictive label. Typical examples here include next-word prediction or trying to reconstruct a blacked-out spot in an image or sentence.
- **Reinforcement Learning (RL):** while RL in a large state/action space setting is partially an application of supervised learning, it is often considered a separate

category. The main difference however between supervised learning and RL is that within the RL-framework we typically do not have an up-front dataset but need to devise a smart way to collect the data through interaction with the (simulated) environment.

Within this report the focus will be on supervised learning as it currently is the most widely used application of ML in a low-power swarm setting. We will however also consider the RL setting, as utilizing RL as the control mechanism in a large swarm (e.g., a swarm of mobile robots) can have many potential applications.

7.2. Reinforcement Learning (RL)

Reinforcement learning (RL) is a machine learning technique in which an agent learns a policy through trial and error by interacting with an environment, which is specified using a Markov Decision Process (MDP). Unlike supervised learning, where models learn from labeled examples, RL agents dynamically perform actions based on an observation of the current state of the environment and receive feedback in the form of a numerical reward signal (as illustrated in Figure 4). By taking this action, the state of the environment is updated to reflect the consequences. This mapping from states to actions is called the policy, and the goal of the agent is to learn a policy that maximizes the (discounted) cumulative reward over an episode, called the return. An episode is a sequence of states, actions, and rewards that starts at the initial state and ends when a terminal state is reached. In DRL, the policy takes the form of a neural network that takes the state as input and outputs the action to be taken. This policy network is optimized using the reward signal to encourage or discourage certain behavior.

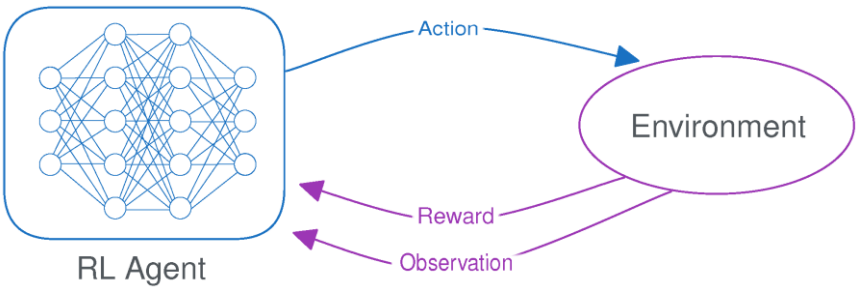


Figure 4: An illustration of the reinforcement learning loop

MDPs can work with either discrete or continuous action spaces, depending on the nature of the task. An illustration of both is given in Figure 2. For discrete tasks, the agent can choose from a limited set of options, such as a cardinal direction to move in. In that case, the neural network outputs a single value for each possible action and the policy consists of either choosing the action with the highest value or sampling from the distribution of these values. 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. Within this document we will describe a method that can effectively compress such a policy network for continuous action spaces. DRL algorithms for continuous action spaces, such as PPO (Schulman, Wolski, Dhariwal, Radford, & Klimov), A2C (al. V. M., 2016), SAC (Haarnoja, Zhou, Abbeel, & Levine), or TD3 (Scott Fujimoto), work by modeling 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 5, so this will be our focus. 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 modeling 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.

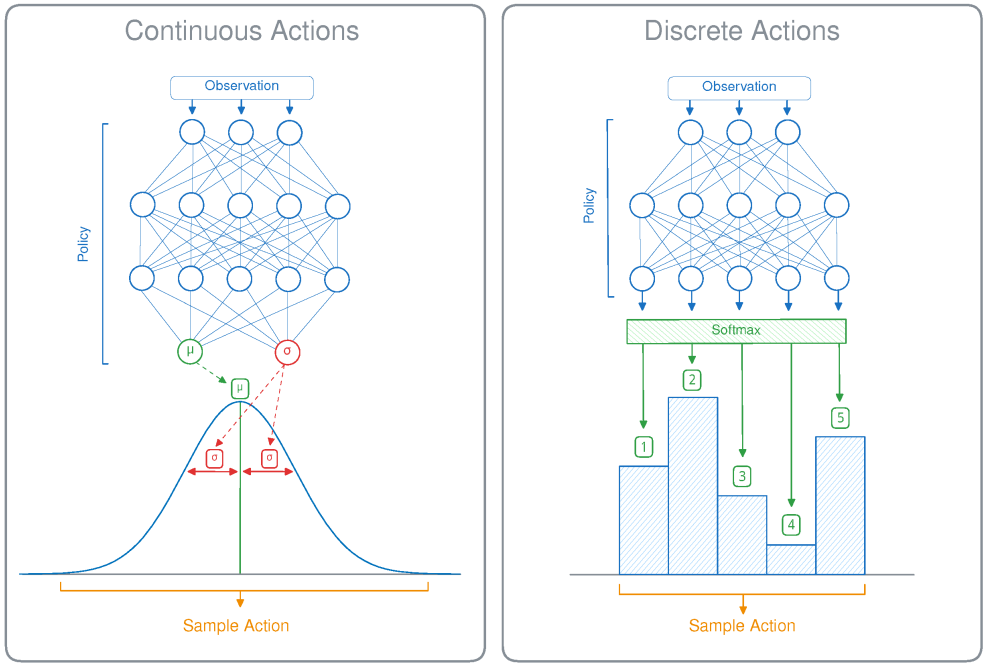


Figure 5: Discrete and continuous action spaces for stochastic DRL policies

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 σ=0) will produce the best results during evaluation (al. T. H.), 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 optimal. Since the policy is trained with this stochasticity in place, it is sometimes detrimental to remove it, as the policy has learned to rely on it. This is especially true for SAC agents, which are trained to maximize an entropy-regularized return and, therefore, to obtain the highest possible return while also remaining as stochastic as possible.

7.3. Policy Distillation (PD)

The concept of Knowledge Distillation (KD) as a model compression method was first introduced in the context of supervised learning (Cristian Buciluundefined) and later extended to deep Reinforcement Learning (DRL) policies by Rusu et al. (Rusu, et al.). It works by compressing a Deep Neural Network (DNN) that is designated as the teacher and training a smaller student network to emulate the output of the larger teacher. After training on a more powerful computing instance, the students can be deployed

efficiently on low-power edge devices, where the teacher would be too large to run effectively. Training data are collected by recording the observations and the teacher's network outputs in a replay memory (D) while interacting with the environment by choosing actions according to the teacher's policy. This replay memory is periodically refreshed to widen the distribution of states encountered by the student.

The original policy distillation (Rusu, et al.) method was teacher-driven, meaning that the policy of the teacher is followed while collecting transitions to fill D. With a student-driven control policy, the actions are chosen by the student while still storing the teacher outputs in D (Czarnecki, et al.). 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 automatically 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.

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 (Rusu, et al.). 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. Using overcomplete DRL models also helps with alleviating optimization issues, such as becoming stuck in local minima, which occur less when learning to emulate an existing network in distillation (Rusu, et al.).

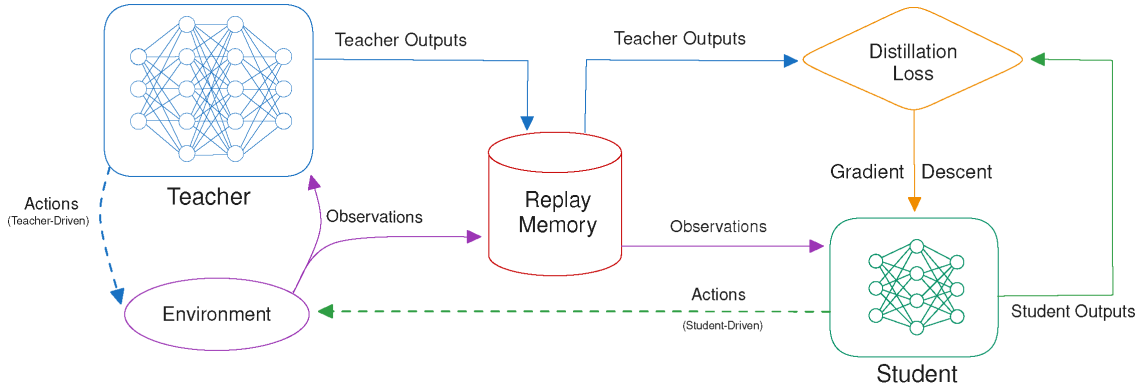


Figure 6: An illustration of the policy distillation algorithm

Policy distillation 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 all the teacher network outputs (Hinton, Vinyals, & Dean). Since policy distillation was originally developed for DQN teachers, the other network outputs correspond to the state-action (Q) values for all possible discrete actions. 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. The student is trained using the Kullback–Leibler divergence (KL) between the teacher (qT) and the student (qS) outputs, with θS the trainable student parameters and τ a temperature used to sharpen or smoothen the teacher outputs:

$$L_{KL}(D, \theta_S) = \sum_{i=1}^{|D|} \text{softmax}\left(\frac{q_i^T}{\tau}\right) \ln\left(\frac{\text{softmax}\left(\frac{q_i^T}{\tau}\right)}{\text{softmax}(q_i^S)}\right) \quad \text{(Eq. 1)}$$

In this original definition of the policy distillation loss for DQN teachers, the network outputs (q) correspond to a list of Q-values, one for each possible action. The softmax function is used to transform these Q-values into a probability vector over the actions, to be used as input for the KL-divergence. This approached is outlined in Algorithm 1.

Algorithm 1: Policy Distillation

Input: Fully trained teacher M_t, RL environment *env*
Output: Fully trained student M_s
Randomly initialize 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 $L_{KL}(D_i, \theta_s)$ and q^S (from Equation (1));
  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

Other work has extended this approach for use in combination with actor-critic teacher algorithms for discrete action spaces by applying a similar KL-divergence loss between the two policies directly (Green, Vineyard, & Koç) and optionally including an additional term to learn the critic values (Avé, Mets, De Schepper, & Latre). Later in this document,

we look at how to adapt this loss for the distillation of continuous actions and for learning how many times an action can be repeated before a new decision is required.

8. TinyML

8.1. Frameworks

In this section we will provide a detailed SOTA on which deep learning frameworks are currently available to perform machine learning (and especially the inference part) on low-power devices.

Memory is a big issue when it comes to performing deep learning on microcontrollers. There are numerous previous works which proposes different ways to reduce the total memory usage during the training process of the machine learning model.

POET (Patil) proposes the rematerialize and page to flash techniques to reduce the peak memory usage. Rematerialize technique deletes the intermediate neuron activations during the forward pass, and recreate it during backward propagation, such that the peak memory usage is limited. While the page to flash technique unloads the intermediate activations to the flash storage during the forward pass, and loads them back during the backward propagation.

Tiny Training Engine (TTE) (Lin) introduced the quantization-aware scaling and sparse update algorithm to automatically scale the gradient of a quantized neural network to improve the performance of on-device training. Next, different system-level optimizations, such as compile time auto-diff, graph pruning and operation re-ordering to reduce the memory consumption for tinyML.

MiniLearn (Profentzas) suggests to dequantize the learnable parameters of some intermediate layers during the on-device learning stage to improve the training efficacy. The intermediate output from earlier layers were used as input dataset. They used static pruning based on L1-norm of the activations to enable tinyML. To compensate the

information loss after pruning, the final fully connected layer is retrained with a stored fine-tuning dataset after the on-device training completes.

The scope of this document is on the algorithms which enable tinyML, i.e., the quantization and pruning algorithms that are used to compress the model while minimizing the impact on the model's performance. We will discuss them in detail in the next section.

8.2. Pruning

The quantization of the baseline model enables the learning on the micro-controller, but the energy-consumption may still be high. Additionally, the micro-controller only has a limited weight count capacity compared to the cloud servers the baseline model was trained on, so reducing the number of parameters in the network is essential in deploying it effectively. To reduce the energy-consumption of the model on the micro-controller, pruning methods can be used. These methods will create a sparse neural network that requires fewer computations compared to the baseline. It will therefore consume less energy and require a smaller memory and storage footprint.

8.2.1. Layer-Wise Relevance Propagation (LRP)

This technique was first proposed by Bach et al. (Bach, 2015) to explain the predictions of neural networks by attributing relevance scores to features in the input data. LRP 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 Rl j 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. Iterative Magnitude Pruning

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

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.

8.2.2. Finetuning of pruned models

This is a common step, either as an intermediary step of progressive pruning or to regain accuracy after pruning (Blalock). This is typically done using the original dataset, however, to recover existing behavior, not to adapt to new data after compression. Gordon et al. (Gordon) did evaluate applying unstructured IMP on the BERT 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. We instead focus on structured pruning, where entire neurons or filters are removed from the network, at the potential cost of less expressiveness for the same size. Srivastava et al. [5] also combine pruning with adaptation to new tasks but in the context of multi-task continual learning where the network size is kept constant, whereas we use pruning for compression.

8.3. Knowledge Distillation

In knowledge distillation, a large teacher model is compressed by transferring its knowledge to a smaller student model. Instead of learning from the real labels, the student is trained to emulate the predictions in the outputs of the teacher network. This is done by first feeding inputs from the training data into both the teacher and the student. The student is then updated using the Kullback–Leibler (KL) divergence between the probability vectors given by the network outputs of both models, while the teacher's weights stay frozen. The reasoning behind this is that the teacher outputs contain additional 'dark' knowledge for all labels and not only the `correct' one, which cannot be obtained by training the student directly on the true labels. Often, the teacher's output distribution is first softened by applying a temperature parameter to ensure that more additional knowledge is transferred, increasing the generalization capabilities of the student.

8.4. Quantization

The OpenSwarm project is built around micro-controllers such as the 'MAX78000' developed by Analog Devices. It includes hardware acceleration for Convolutional

Neural Networks (CNNs). However, the hardware only supports 1-, 2-, 4-, and 8-bit weights.

CNNs are traditionally trained in 32-bit, single-precision floating point format, or 64-bit, double-precision floating point format. Therefore, quantization methods are required to convert the high-precision weights of the pre-trained baseline model to the corresponding low-precision weights which can be deployed on the micro-controller.

We evaluate different existing quantization methods on varying levels of precision allowed by the micro-controller hardware. This is required for both the on-device inference and the on-device fine-tuning of the baseline model.

8.4.1. Post-Training Quantization (PTQ)

This technique starts with a machine learning model that has already been trained to a desired level of accuracy using full-precision floating-point numbers. PTQ then works by converting this model's weights and activations, to a lower precision format, while approximately preserving its behavior. This conversion process significantly reduces the overall size of the model, making it easier to store and transfer, but also perform inference on devices such as the MAX78000 micro-controller by ADI that have hardware acceleration for neural networks with up to 8-bit integer weights. In the context of PTQ, this conversion needs to maintain the original numerical representation of the input values. This means that only linear quantization functions can be used, where the relative distance between values in the input domain is maintained after applying the quantization. Reducing the precision also introduces inaccuracies, however, which accumulate when propagating forward through the network. This makes PTQ a flexible method to be able to deploy existing models on such low-power devices, but this usually comes with a significant drop in accuracy.

8.4.2. Quantization-Aware Training (QAT)

Quantization-Aware Training (QAT) is often preferred instead to account for these inaccuracies, where the quantization transformation is part of the architecture while training. This also allows non-linear quantization functions to be used, where certain

regions in the original representation can be represented more accurately than others, more closely matching the distribution of values that need to be quantized using the limited representational power. By having these functions as part of the computational graph during backpropagation, the optimizer can account for the noise introduced by the loss of precision and learn values for the weights and biases that better align with the transformation boundaries. This process is significantly less flexible, however, since the network architecture needs to be modified to include these additional components. In practice, this often means replacing the existing modules in traditional deep learning frameworks with quantization-aware variants, which might no longer be interoperable with other advanced model optimization techniques such as (un)structured pruning.

8.5. Policy Compression for Continuous Actions

Deep reinforcement learning (DRL) methods have been shown to be highly effective at solving discrete tasks in constrained environments, such as energy-aware task scheduling (Xun, et al., Deep Reinforcement Learning for Delay and Energy-Aware Task Scheduling in Edge Clouds.) and offloading (Alhartomi, Salh, Audah, Alzahrani, & Alzahmi; Tang & Wong) in edge networks, 5G beamforming and power control (Mismar, Evans, & Alkhateeb, 2020), and network function (NF) replica scaling (Avé, Soto, Camelo, De Schepper, & Mets) in software-defined networking (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 (Zhang & Chen), the continuous control of drone steering (Azar, et al.), the amount of resources to allocate in micro-grids (Lei, et al.), and multi-beam satellite communication (Wei, Guo, & Yang). 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 each environment. Different types of DRL algorithms have been developed that can learn this type of behavior 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 optimizes for a stable connection while roaming between several access points (APs). When modeled as a discrete task, the agent can decide at each time step to which from a set of known APs 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 optimize 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 optimization, are best modeled using continuous action spaces.

Learning continuous behaviors is often more challenging than learning discrete actions, however, as the range of possible actions to explore before converging on an optimal policy is infinite. A common approach is, therefore, to discretize the continuous actions into a fixed set of possible values (Mismar, Evans, & Alkhateeb, 2020), 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 (Tang & Agrawal). 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 Soft Actor-Critic (SAC) algorithm (Haarnoja, Zhou, Abbeel, & Levine) for example.

Since many tasks are carried out on battery-powered mobile platforms, additional constraints apply in terms of computing resources and power consumption. This can make it practically infeasible to deploy large models on low-power edge devices that need to perform such tasks. Some methods have been introduced to combat this, by reducing the size and therefore computational complexity of the deep neural networks (DNNs) through which the DRL agent chooses its actions, without decreasing its effectiveness at solving the task. In DRL, one of the most popular model compression techniques is policy distillation (Rusu, et al.). Here, a Deep Q-Network (DQN) can be

compressed by transferring the knowledge of a larger teacher network to a student with fewer parameters. Compressing DRL models enables low-power devices to perform inference using these models on the edge, increasing their applicability, reducing cost, enabling real-time execution, and providing more privacy. These benefits have recently been demonstrated in the context of communication systems and networks for the compression of DRL policies that dynamically scale NF replicas in software-based network architectures (Avé, Soto, Camelo, De Schepper, & Mets).

However, the original policy distillation (Rusu, et al.) method was only designed for policies from DQN teachers, which can only perform discrete actions. Most subsequent research has also continued in the same direction by improving distillation for teachers with discrete action spaces (Avé, Soto, Camelo, De Schepper, & Mets; Czarnecki, et al.; Avé, Mets, De Schepper, & Latre; Green, Vineyard, & Koç). 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 section, 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 (Haarnoja, Zhou, Abbeel, & Levine) and PPO (Schulman, Wolski, Dhariwal, Radford, & Klimov) teacher on the popular HalfCheetah

and Ant continuous control tasks (Duan, Chen, Houthooft, Schulman, & Abbeel). Through these benchmarks, in which the agent needs to control a robot with multi-joint dynamics, we focus on autonomous mobile robotics 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.

8.5.1. 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 policy distillation. 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. (Czarnecki, et al.) for discrete action spaces, but they note that this method saturates early on for teachers with suboptimal critics. Xu et al. (Xu, Wu, Che, Tang, & Ye) take this approach for a multi-task policy distillation, where a single agent is trained based on several teachers that are each specialized in a single task, to train a single student that can perform all tasks. They first used an MSE loss to distill 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. (Czarnecki, et al.). Lai et al. (Lai, Zha, Li, & Hu) 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 (Stanton, Izmailov, Kirichenko, Alemi, & Wilson) and allows us to more effectively distill 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 (Konda & Tsitsiklis). 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 behavior indicating which actions are viable in each 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 (Mysore, Mabsout, Mancuso, & Saenko). 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 behavior of the teacher in the form of the policy, as our goal is compression for low-power inference on edge devices.

Berseth et al. (Berseth, Xie, Cernek, & de Panne) also distill the teacher policy directly in their PLAID 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.

8.5.2. 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 7. 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 behavior 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.

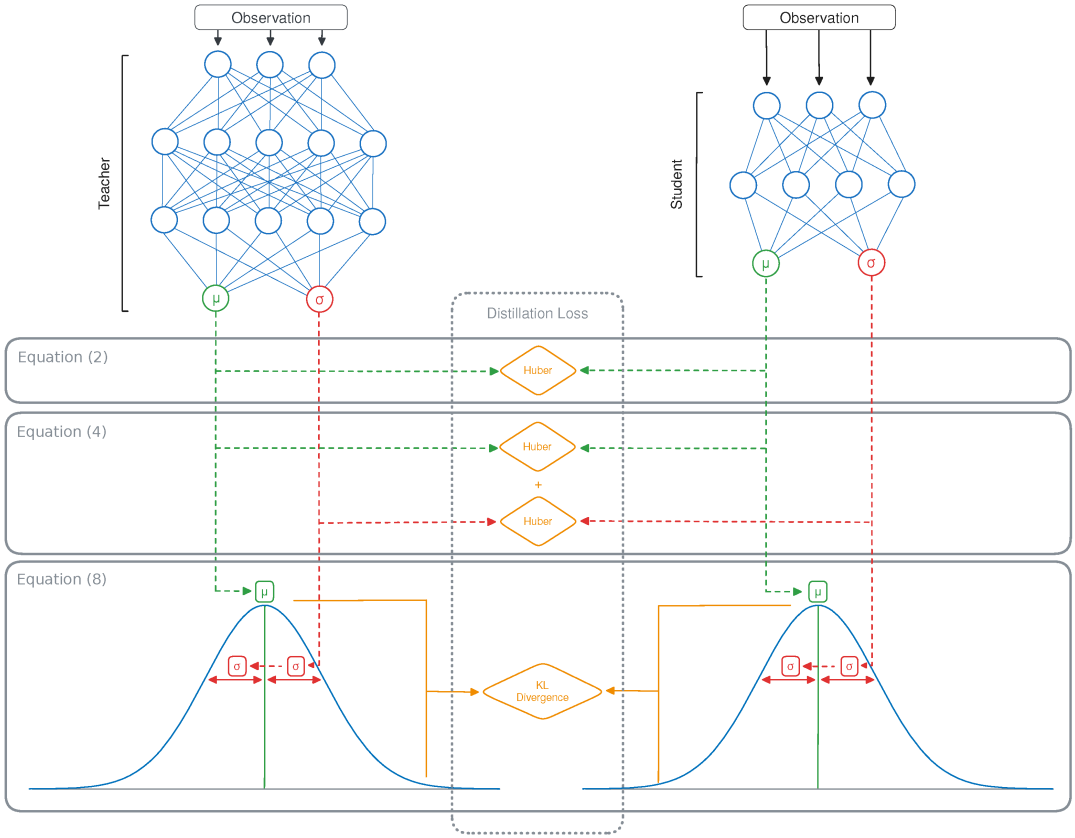


Figure 7: An illustration of the proposed distillation losses

Distilling the Mean Action

The first loss is the simplest, serving as a baseline that is mostly 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. (Berseth, Xie, Cernek, & de Panne). The original policy distillation loss is based on the KL-divergence between two probability vectors, which cannot be used for learning a single mean value for each action (Rusu, et al.). Instead, we propose to use the Huber loss between the mean of the student (μS) and the teacher (μT) for each action (a):

$$L(D, \theta_S) = \sum_{i=1}^{|D|} \sum_{a \in A} \text{Huber}\left(\mu_{i,a}^{S}, \mu_{i,a}^{T}, 1\right) \qquad \text{(eq. 2)}$$

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

We chose to make use of the Huber loss for this baseline instead of the MSE loss used by Berseth et al. (Berseth, Xie, Cernek, & de Panne) 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.

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 policy distillation for discrete action spaces (Rusu, et al.). Learning σ is even more important when using student-driven policy distillation, 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 generalization 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 behavior. 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. 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:

$$L(D, \theta_S) = \sum_{i=1}^{|D|} \sum_{a \in A} \text{Huber}\left(\mu_{i,a}^S, \mu_{i,a}^T, 1\right) + \lambda \cdot \text{Huber}\left(\sigma_{i,a}^S, \sigma_{i,a}^T, 1\right)$$

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 N(μ, σ), which was not possible before. 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.

Distilling the Action Distribution

In traditional policy distillation, the student is also trained using a probability distribution over actions, where this is defined using the probability vector given by the teacher outputs (Rusu, et al.). Instead of learning to reproduce the same precise values as the teacher, such as what we proposed in Equation (2) 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 deep learning. 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:

$$\mathrm{KL}(P,Q) = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx \qquad \text{(Eq. 5)}$$

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

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

To substitute this in Equation (5), we first focus on the log division:

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

The full equation then becomes:

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

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

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

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:

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

Finally, we apply this for policy distillation of continuous action spaces:

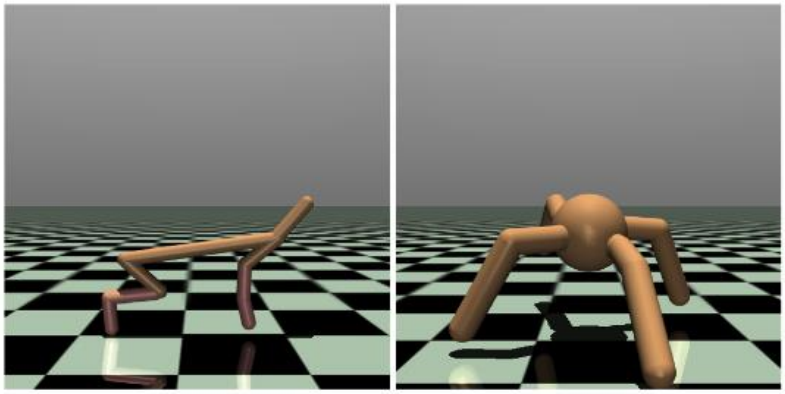
$$L_{KL}(D,\theta_S) = \sum_{i=1}^{|D|}\sum_{a\in A} 1 \log\frac{\sigma_{i,a}^T}{\sigma_{i,a}^S} + \frac{\left(\sigma_{i,j}^S\right)^2 + \left(\mu_{i,a}^S - \mu_{i,a}^T\right)^2}{2\left(\sigma_{i,a}^T\right)^2} - \frac{1}{2} \qquad \text{(Eq. 8)}$$

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

8.5.3. Experimental Setup

Evaluation Environments

We evaluate the effectiveness of the loss functions proposed in Section 4 in two continuous control environments (Ant-v3 and HalfCheetah-v3) that are part of the Gymnasium project, shown in Figure 8. These environments were chosen as they are arguably the two most prevalent benchmarks in the MuJoCo suite (Todorov, Erez, & Tassa), 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 8: A graphical render of the environments used in this section

Environments in this suite have complex, high-dimensional continuous state and action spaces and require sophisticated control strategies. 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 (Todorov, Erez, & Tassa), 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 center 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.

Model Architectures

To use as teachers for training our students and as baselines for uncompressed models, we make use of two state-of-the-art DRL algorithms: SAC (Haarnoja, Zhou, Abbeel, & Levine) and PPO (Schulman, Wolski, Dhariwal, Radford, & Klimov). For increased reproducibility, we make use of publicly available pre-trained agents from the Stable Baselines3 project for our teacher networks. These teachers have different behaviors 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 1, which shows the average return and standard deviation for 200 episodes using our teacher models on the used environments and for both evaluation methods.

Table 1: Average return and standard deviation for our PPO and SAC teachers on 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 Stable Baselines3 project uses a different network size for these teachers depending on the environment, as shown in Table 2. Note that we did not include the

Table 2: The number of parameters in our student networks, SAC, and PPO teacher.

Network	Ant-v3	HalfCheetah-v3
Student 6 (μ only)	16,008	9862
Student 6 (μ and σ)	16,528	10,252
SAC Teacher	98,576	73,484
PPO Teacher	23,312	142,348

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.

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 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.

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 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

Table 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

five independent runs, resulting in the mean and corresponding standard deviations of the return and entropy at each epoch (discussed later).

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:

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

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.

8.5.4. Results and Discussion

In this section, we investigate the effectiveness of the three loss functions proposed 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 behavior. This is measured in terms of the average return, but we also analyze the entropy of the action distribution to evaluate how well the stochasticity of the teacher is maintained as part of the distillation process. Afterward, 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 analyze 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.

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 (2) instead, the

replay memory will only contain deterministic trajectories, which are not always optimal (see Table 1).

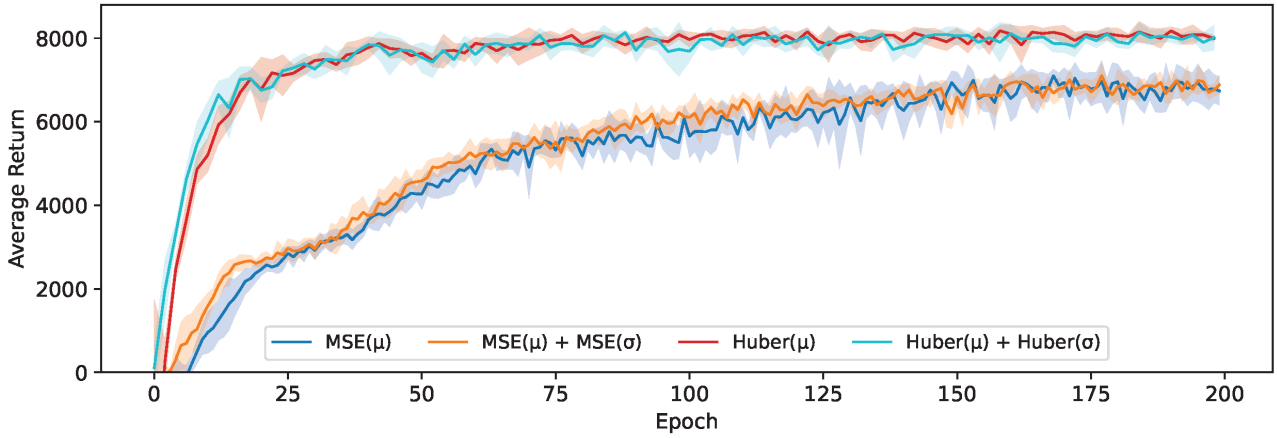


Figure 9: The average return obtained using either an MSE or Huber-based distillation loss

As a baseline, we start by comparing the MSE-based loss originally proposed by Berseth et al. (Berseth, Xie, Cernek, & de Panne) between mean actions μ to our Huber-based loss functions ((2) and (4)), as well as an analogous MSE loss with both μ and σ. The results of this are shown in Figure 9. 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.

Looking at Figure 10, we do see that our proposed loss based on the KL-divergence (Equation (8)) to transfer the action distribution performs significantly better than those based on a Huber loss. This does align with our hypothesis that the loss landscape when shaping the probability distribution of the student to match that of the teacher more closely is smoother than learning the two concrete values independently, leading to a better optimization. Learning these values separately, as was the case in Figure 9, 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 by comparing these results with different student sizes.

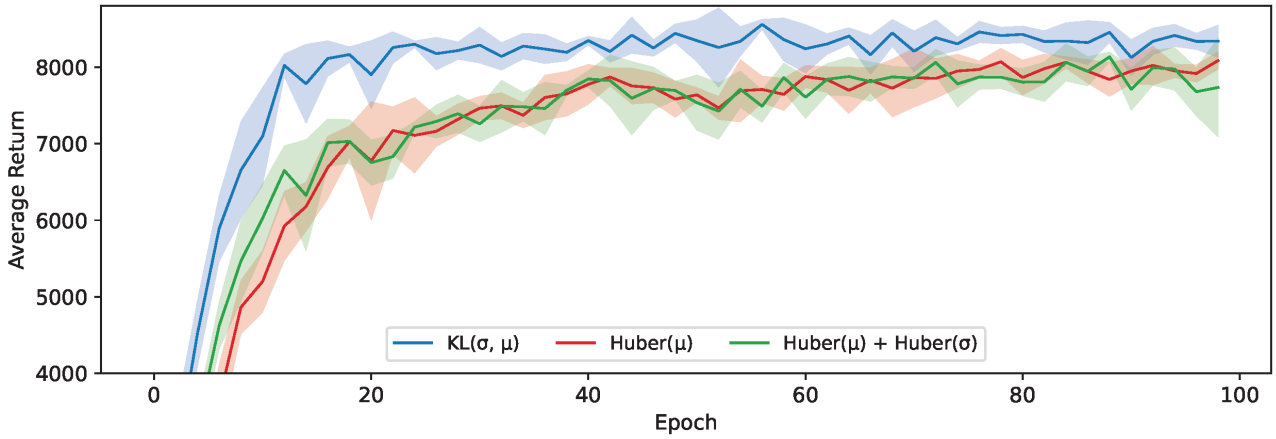


Figure 10: The average return of 5 students trained using student-driven distillation on the HalfCheetah-v3 environment with a SAC teacher.

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.

Impact on Average Return

The effects of this distribution shift can be seen in Figure 11, which shows the average return for both control policies and all loss functions on 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 (8)) being considerably more sample efficient, followed by the agent trained using the Huber loss for both μ and σ (Equation (4)) and finally the agent that only learns a deterministic policy in the form of the mean actions μ (Equation (2)). 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.

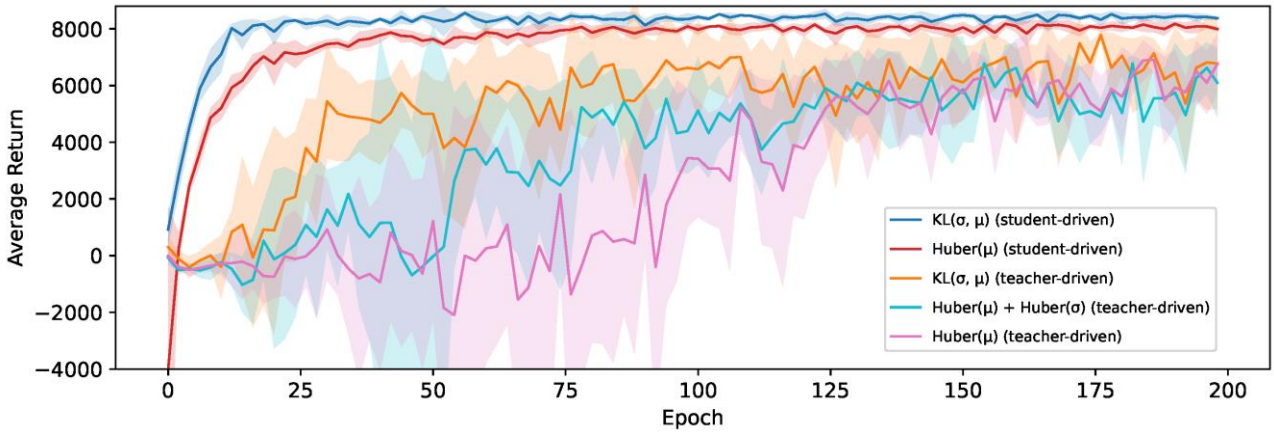


Figure 11: The average return during training for student and teacher-driven distillation on the HalfCheetah-v3 environment and an SAC teacher.

Figure 12 also shows the average return for all configurations, but this time on 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 on 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. 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.

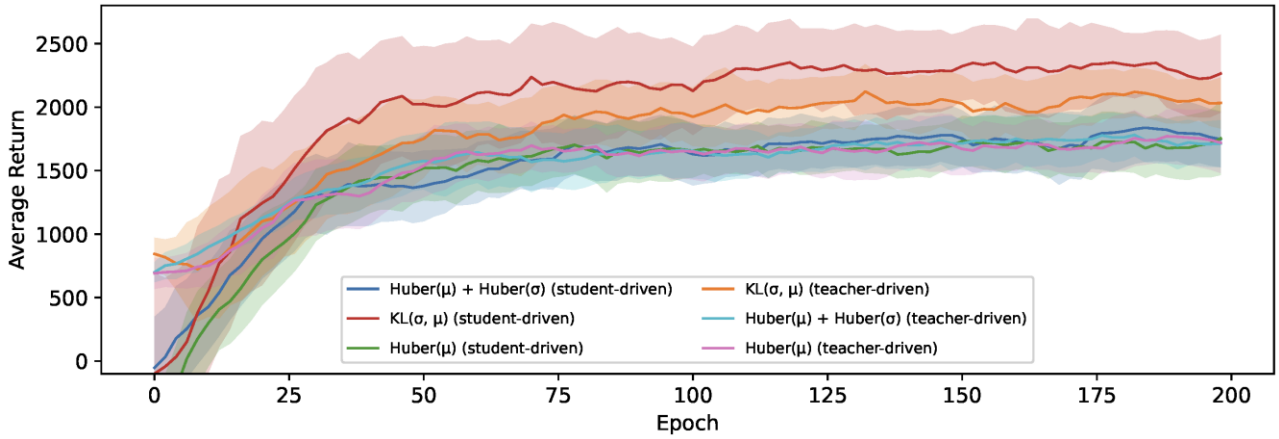


Figure 12: The exponential running mean of the average return during training for student and teacher-driven distillation on the Ant-v3 environment and an SAC teacher.

Impact on Policy Entropy

We hypothesize that this difference in performance between the distillation losses is mainly due to the maintained accuracy of the policy stochasticity. This has particular importance to reach a high degree of fidelity with teachers such as SAC, which are optimized to maximize an entropy-regularized return (Haarnoja, Zhou, Abbeel, & Levine). To verify this, we measure the entropy of the action distribution predicted by the students during testing, as can be seen in Figure 13. 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.

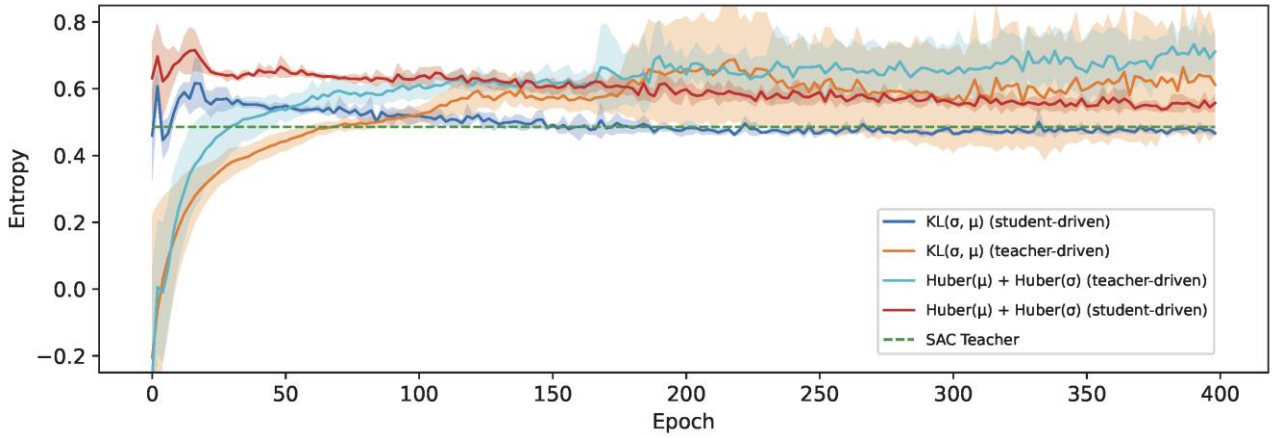


Figure 13: The average entropy measured for students trained using a loss that includes σ on the HalfCheetah-v3 environment and an SAC teacher.

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 behavior and thus results in faster learning and reducing once the control policy is stabilizing. 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.

Teacher Algorithm

In this section, we evaluate how generic our proposed methods can be applied with different teacher algorithms, focusing on the two most used for continuous control tasks: SAC and PPO. The SAC algorithm tries to optimize 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 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 11 on the HalfCheetah-v3 environment and in Figure 15 on 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.

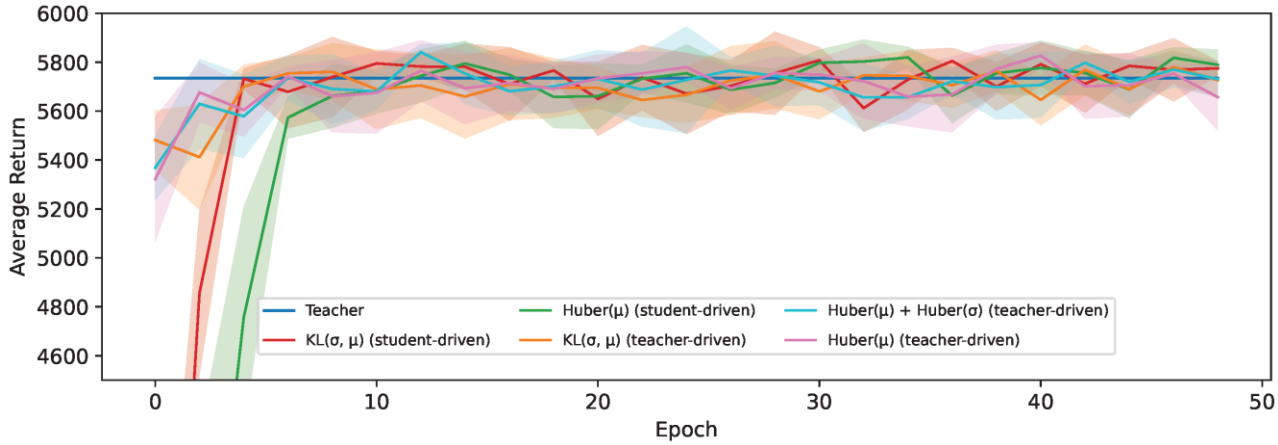


Figure 14: The average return during training for student and teacher-driven distillation on the HalfCheetah-v3 environment and a PPO teacher.

What is more notable about the PPO results however is that the students outperform their teacher on the Ant-v3 environment. In the context of policy distillation for discrete action spaces, this phenomenon has also been observed and attributed to the regularization effect of distillation (Rusu, et al.). These students (Figure 15) 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.

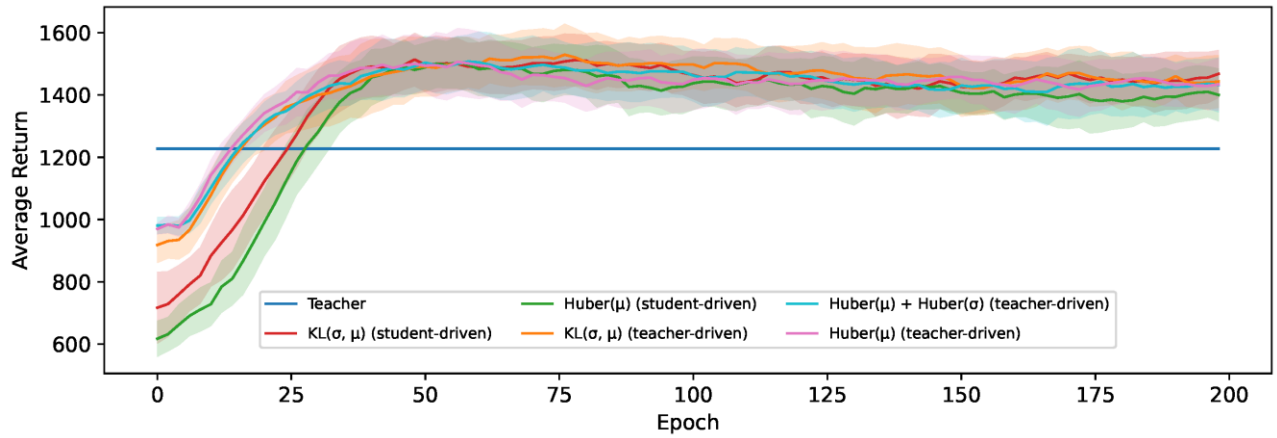


Figure 15: The exponential running mean of the average return during training for student and teacher-driven distillation on the Ant-v3 environment and a PPO teacher.

This outcome relates to the work by Stanton et al. (Stanton, Izmailov, Kirichenko, Alemi, & Wilson), who have shown in a supervised learning context that knowledge distillation does not typically work as commonly understood where the student learns to exactly match the teacher's behavior. 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 generalization of our students first improves, but as training progresses, this shifts to improving their fidelity.

The students that were trained based on an SAC teacher performed slightly worse compared to their teacher on the HalfCheetah-v3 environment, and a more significant performance hit was observed on 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.

Compression Level

We investigate the compression potential of our methods by repeating the experiments for a wide range of student network sizes, as listed in Table 3. In Figure 16, our loss based on the KL-divergence (Equation (8)) was used, while Figure 17 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. 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.

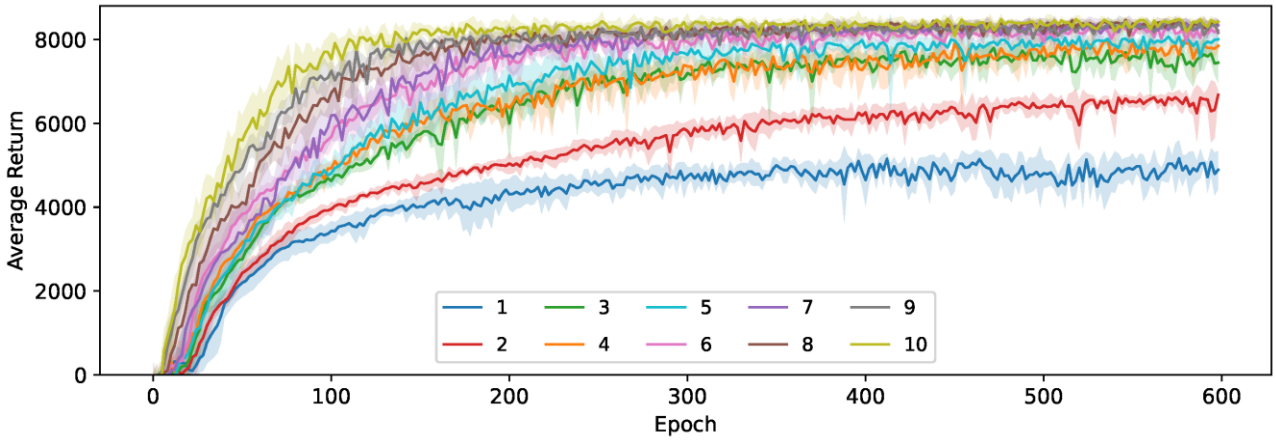


Figure 16: The average return for 10 student sizes during training using Equation (8) (KL).

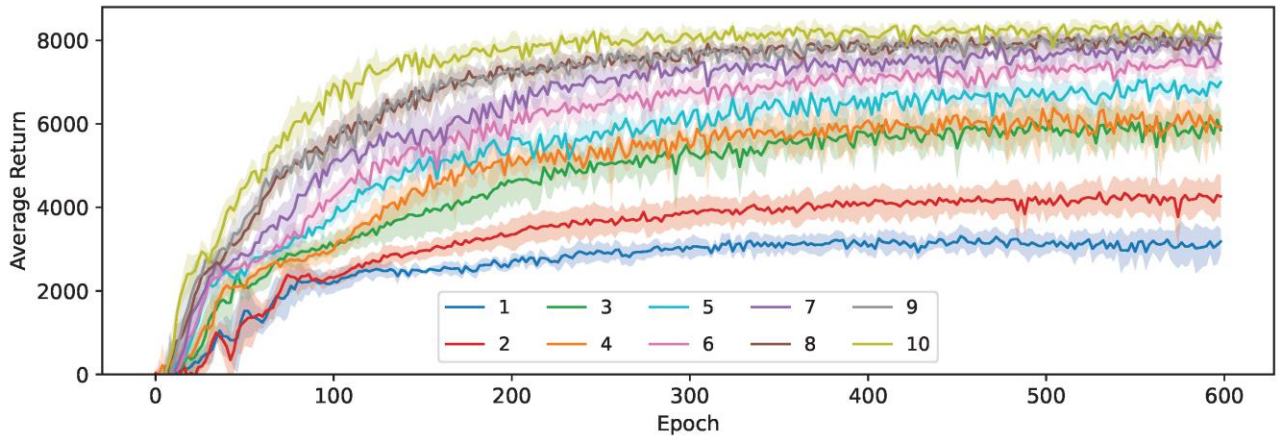


Figure 17: Average return for 10 student sizes during training using Equation (4) (Huber).

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 behavior 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.

Runtime Performance

Finally, we analyze how this compression to the various student architectures (see Table 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 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 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 parallelization 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 neural network 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 3 and multiplying it by 4. The popular Arduino Uno R3 microcontroller, for example, has only 32 kB of available ROM, which is only enough to store up to student 5, with a size of 24 kB.

Measuring the direct impact of policy distillation 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. Instead, to optimize 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 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 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.

8.5.5. Conclusions

Deploying intelligent agents for continuous control tasks, such as drones, AMRs, or IoT devices, directly on low-power edge devices is a difficult challenge, as their computational resources are limited, and the available battery power is scarce. This section addressed this challenge by proposing a novel approach for compressing such DRL agents by extending policy distillation 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. 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 distill 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% on the HalfCheetah-v3 environment and 34% on 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 regularization effect of policy distillation 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.

8.6. Temporal Distillation

The field of Deep Reinforcement Learning (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 (Matthias Hutsebaut-Buysse), agriculture (Bu) and drone delivery (Munoz). However, the deployment of these RL agents in resource-constrained settings, such as IoT devices or embedded systems, remains a significant challenge (Duisterhof, 2021). 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 neural network that represents the policy (Rusu, et al.). We refer to this as spatial compression, as it reduces the storage space 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, previous work focusing on spatial compression alone has reported that the

increase in inference speed is typically not proportional to the reduction in the number of parameters (Avé, Mets, De Schepper, & Latre; Avé T. D., 2024). Smaller networks with

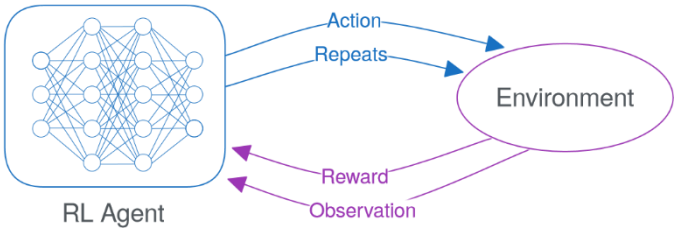


Figure 18: Our proposed method can be considered an extension to the traditional reinforcement learning loop

narrower layers tend to have a limited potential for parallelization, 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 initializing the inference engine also becomes the bottleneck at smaller network sizes, limiting the potential speed-up that can be achieved.

We therefore propose utilizing time as an additional dimension for compression, which does scale proportionally with every increase in 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. We introduce a form of temporal abstraction to the traditional RL loop, depicted in Figure 18. This allows the agent to not only predict what action to perform next, but also *when* a new action will be required. 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 (Biedenkapp, 2021). 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. Including such auxiliary tasks has even been shown to have the potential for improved policy performance and generalization in previous work.

The concept of frame-skipping, where the same action is repeated for a fixed number of frames, is already standard practice in environments with high frame rates such as the Atari benchmarks (Mnih, 2013). 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 (Kalyanakrishnan, 2021). 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 behavior. 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 aims to bridge this gap by allowing discrete agents to predict a continuous value that represents a variable-length sequence of the same action, effectively compressing the policy in the time domain. Although inspired by continuous movement, this approach is more flexible than simply using larger continuous actions to represent a longer stretch of time. It can also extend to tasks that

inherently require discrete decisions, such as the common Atari benchmarks (Bellemare), turning on or off appliances in a smart power grid (Bertolini), or replica scaling in a cloud environment ([17] Soto).

By combining spatial and temporal compression in a single compression method, we can 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 19. We base our method on the concept of policy distillation (Rusu, et al.), where a small student network is trained to emulate the learned behavior 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 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 behavior, without requiring any modifications to the original training process. Experiments on two embodied MiniGrid environments (Chevalier-Boisvert) show that our method is able to reach up to a 1347% increase in the effective step rate compared to 403% through spatial compression alone, while maintaining a similar average return as the original teacher.

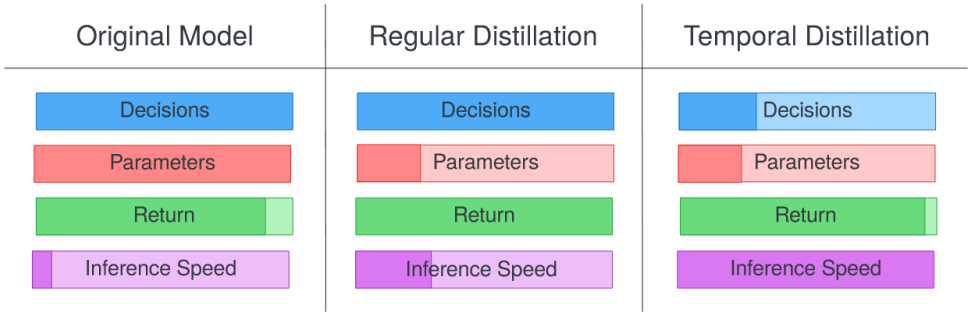


Figure 19: A relative comparison using the FourRooms environment and an ESP32 microcontroller of metrics obtained by the original teacher model, a student compressed by 400% using only regular (spatial) policy distillation, and the same student trained using our temporal distillation method

8.6.1. Related Work

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

Extending Policy Distillation

Several works have extended the concept of policy distillation 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. (Czarnecki, et al.) formalized 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 policy distillation (Rusu, et al.) 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 extract more valuable knowledge from actor-critic teachers by utilizing the critic's value function for bootstrapping or as an intrinsic reward. Avé et al. (Avé, Mets, De Schepper, & Latre) also use the critic during distillation, but as an auxiliary task to enhance the student's understanding of the environment. Although we are using an actor-critic teacher in this method, we opted not to include these extensions to maintain the broad applicability of our core method to other teacher algorithms, such as the DQN for which the policy distillation method was originally designed. Integrating especially the auxiliary critic loss as part of our method should be trivial for future work and has the potential to improve our student's performance even further. Our work is to the best of our knowledge the first to extract temporal 'dark' knowledge from a teacher network to enhance policy distillation. 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.

Learning to Repeat Actions

Dynamic action repetition (DAR) was first introduced by Lakshminarayanan et al. (Lakshminarayanan) by extending the DQN, A2C and A3C 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 optimization 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 framework (Sharma, 2017), 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 $\{1,\dots,J\}$ 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 (Biedenkapp, 2021) further note that decoupling the repeat value from the action, but still training them using a shared objective based on maximizing 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 optimizing 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 optimize 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 generalize from these suboptimal examples.

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

8.6.2. 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

Figure 20: An illustration of the reduced and extended trajectory variants of the replay memory

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. 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 20 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 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 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 21. In all our architectures, we use a shared two-layer MultiLayer Perceptron (MLP) to extract features from the flattened observation, followed by a single parallel layer for the action head and repeat head. These heads are trained using separate optimization objectives, which are then combined to form the modified distillation loss:

$$L_{KL}(D, \theta_S) = \sum_{i=1}^{|D|} 1\,(\mathrm{softmax}(\frac{q_i^T}{\tau}) ln(\frac{\mathrm{softmax}(\frac{q_i^T}{\tau})}{\mathrm{softmax}(q_i^S)}) + (\frac{r_i^S + |r_i^S|}{2} - \frac{r_i^T}{\lambda})^2)$$

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 normalizes 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 mean squared 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.

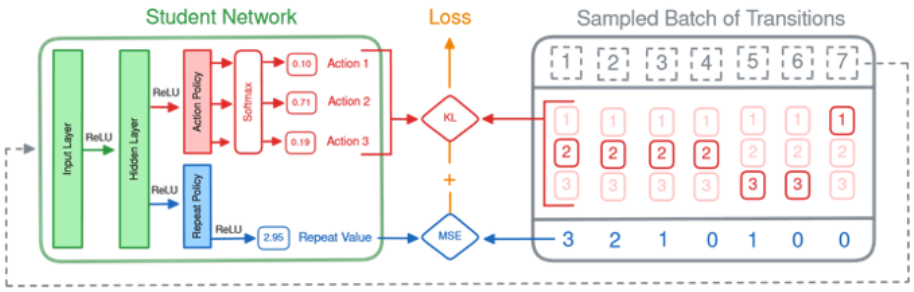


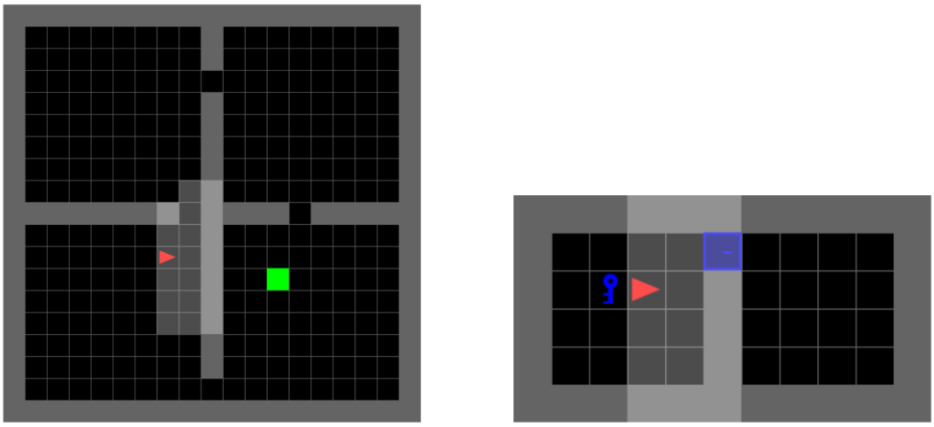
Figure 21: An illustration of the student network architecture and the distillation loss for the actions and additional repeat head

8.6.3. Experimental Setup

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 22: A visualization 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 (Chevalier-Boisvert) provides a wide range of popular benchmark tasks that meet these criteria, with the FourRooms and Unlock environments being selected for our experiments. A visualization of these environments is shown in Figure 22. Here, the positions of the agent, goal, door, key and the gaps in the walls are all randomized 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.

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 1. Adding the auxiliary repeat-prediction task could benefit the internal representation and therefore the generalization 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 policy distillation.

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 21, 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 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 Stable Baselines3 library as our teacher networks. The exact model checkpoints can be found on the HuggingFace page for FourRooms and Unlock. These teacher agents obtain an average return of 0.480 and 0.939 respectively in our testing, as shown in Table 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 favored using publicly available agents as a baseline. Our method is compatible with recurrent neural network (RNN) student architectures, however, requiring only a minor modification in Algorithm 1 to sample entire trajectories instead of random individual transitions.

Table 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

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 a learning rate 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 2 had a larger impact on the training process and required tuning for each student size and environment, as shown in table 3.

Table 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 the section "Impact of Repeat Scale".

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 format, and then used the Onnx2C tool to convert them to C code that can be compiled and deployed on the microcontroller. The other benchmarks are performed using the OnnxRuntime library 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.

8.6.4. 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 policy distillation 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 analyzing the potential difference in behavior between the teacher and student networks.

Behavioural Analysis

Figure 23 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 behavior 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. In case of the reduced trajectory variant, this results in 7 entries, 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 generalize to a policy that only required 3 decisions to reach the goal.

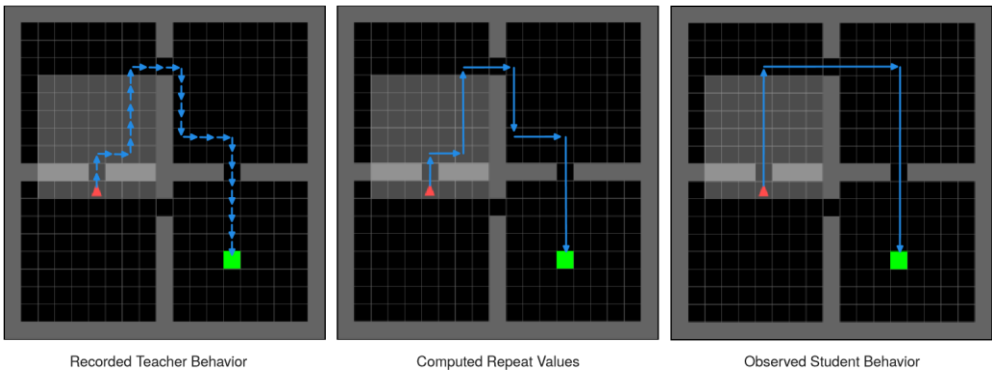


Figure 23: A visualization of the same FourRooms task solved by the teacher and student, with each action represented by an arrow

We attribute this to the regularization effect of knowledge distillation, which was shown to have the potential of improving the generalization capabilities of the student network beyond the teacher's behavior, enabling it to obtain higher average returns than the training data (Rusu, et al.). 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. Still, Figure 24 shows that our students in general learn to perform considerably more repeats than 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.

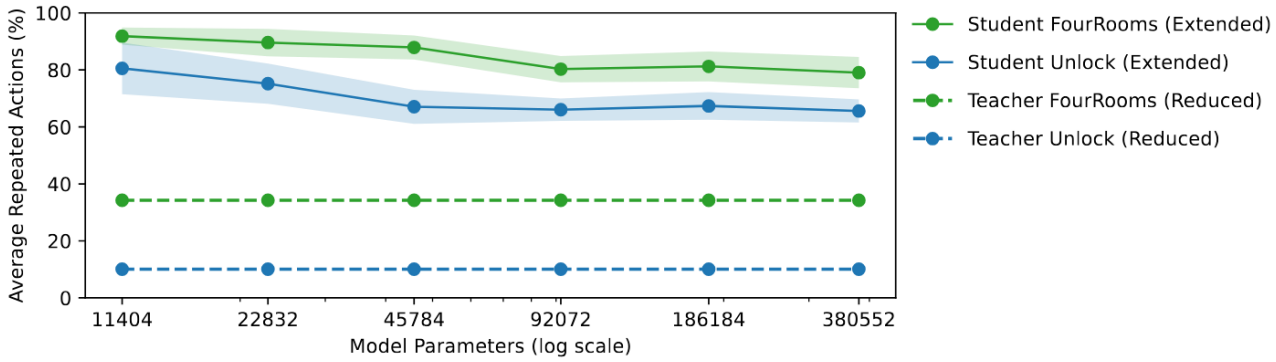


Figure 24: Percentage of action repeats performed by student networks of different sizes

Agent Performance

When looking at the average return obtained in the FourRooms environment, shown in Figure 25, 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. 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 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 behavior, 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 end of the episode, due to the lack of memory. We hypothesize that the loss for this

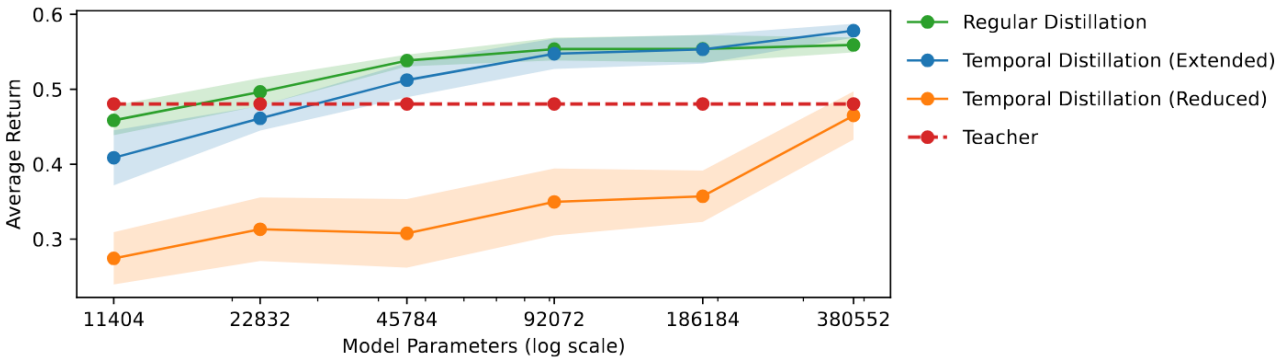


Figure 25: Average return for different student sizes on the FourRooms environment

transition would quickly be minimized, causing the student to focus on the more optimal examples in the replay memory. It also, once again, confirms the potential of policy distillation to improve the generalization of the student through the introduced regularization effect.

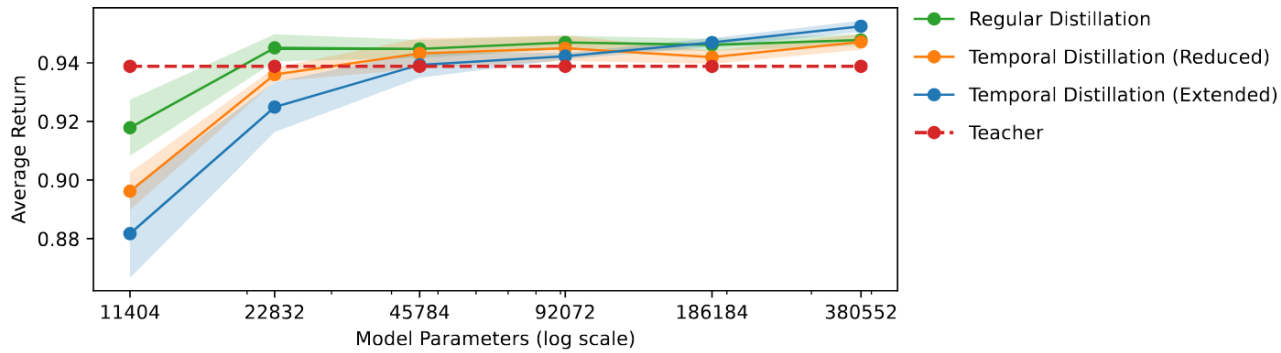


Figure 26: Average return for different student sizes on the Unlock environment

The results from the Unlock environment, shown in Figure 26, 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 27.

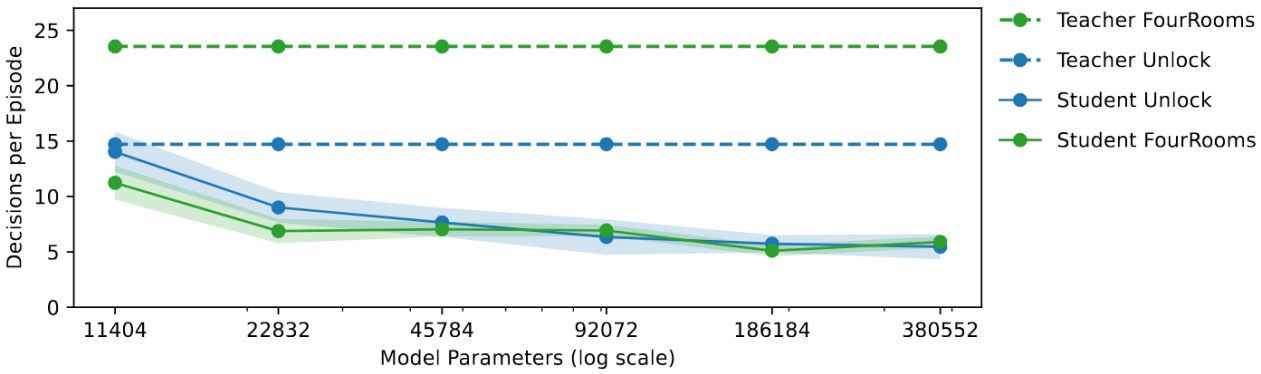


Figure 27: Average number of decisions needed to successfully solve the environments

We previously showed in Figure 24 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 27 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.

Impact of Repeat Scale

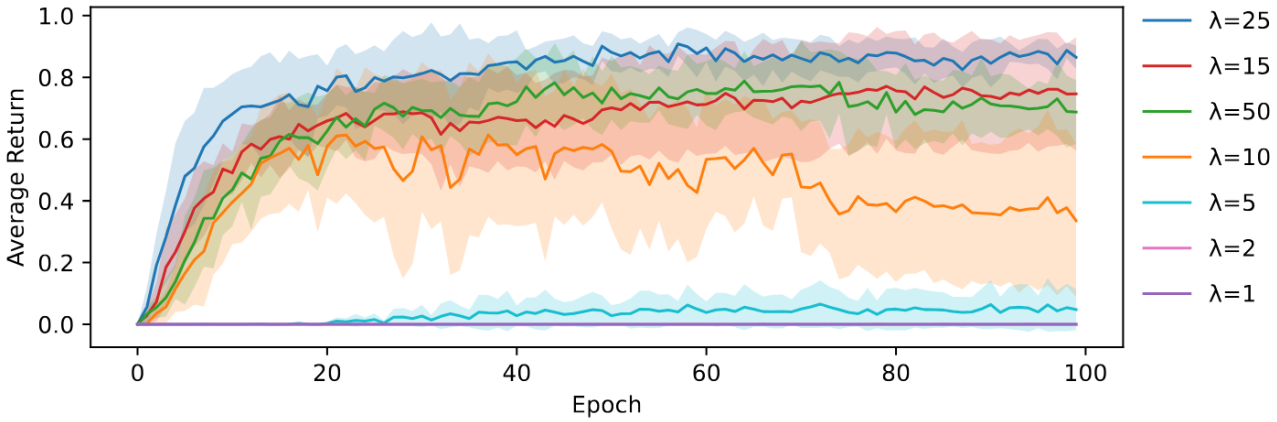


Figure 28: Average return on the Unlock environment for the smallest student and various values for the repeat scale λ in Equation 2.

As mentioned in the experimental setup section, tuning the value of λ in Equation 2 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 28 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 (λ = 15) or too high (λ = 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 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 29. It seems that in general, the student is more likely to

perform more repeats when λ is set to a higher value. 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 less often on average. It also results in a noisier repeat policy, however, causing inaccurate movements and an inability to complete the task. 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.

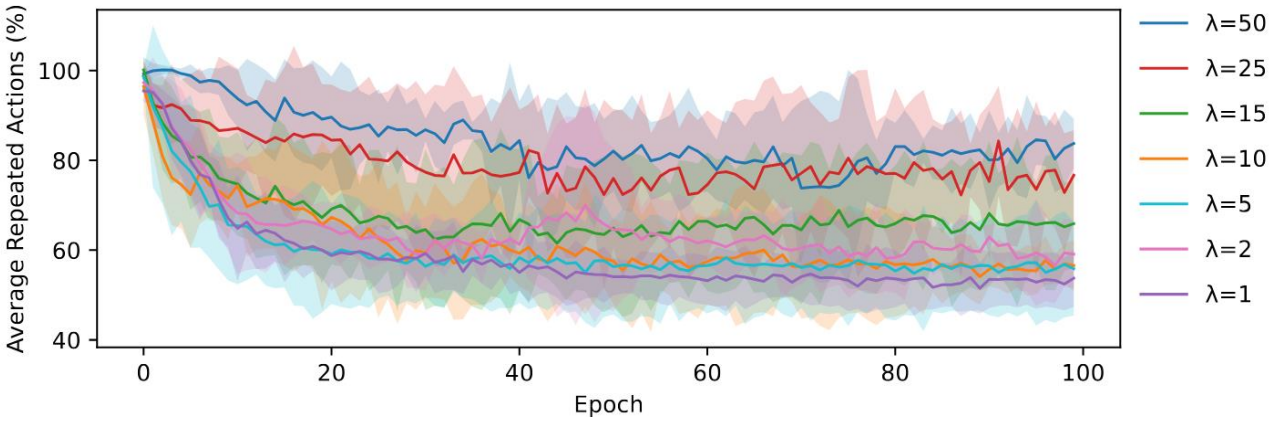


Figure 29: Average percentage of repeated actions on the Unlock environment for the smallest student and various values for the repeat scale λ in Equation 2

Students with more capacity, such as our largest student in Figure 30, are less sensitive to the choice of λ. Only the lowest values (λ = 1 & λ = 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.

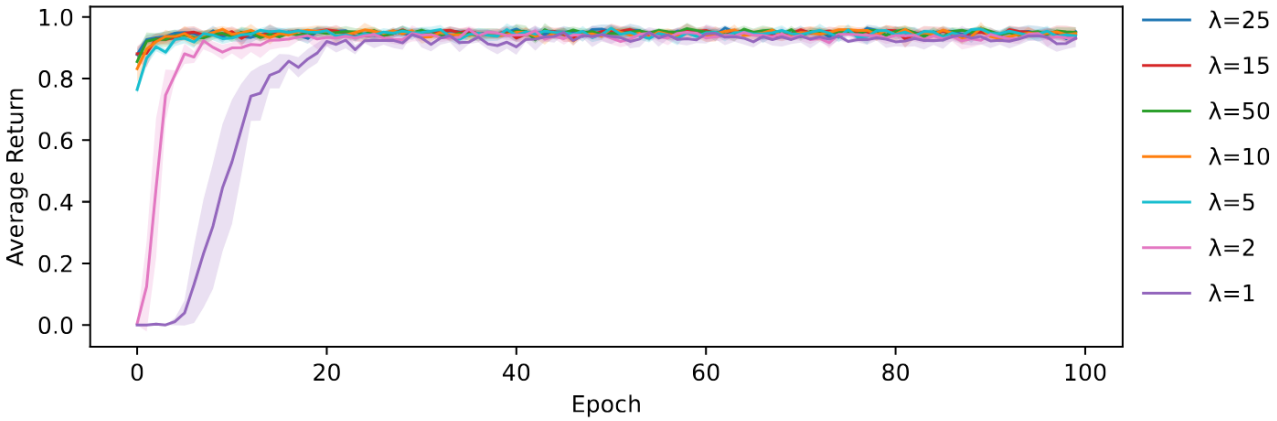


Figure 30: Average return on the Unlock environment for the largest student and various values for the repeat scale λ in Equation 2

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.

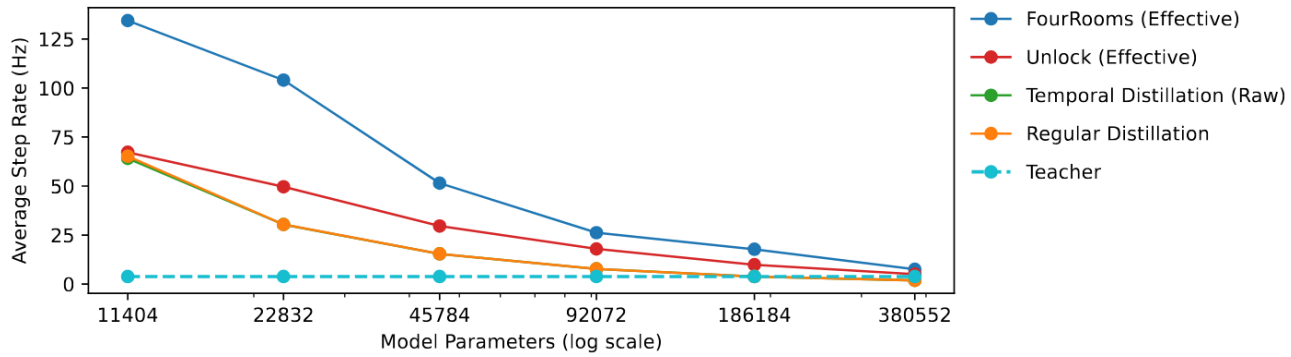


Figure 31: Raw and effective steps per second for different student sizes on an ESP32 microcontroller

Figure 31 shows 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.

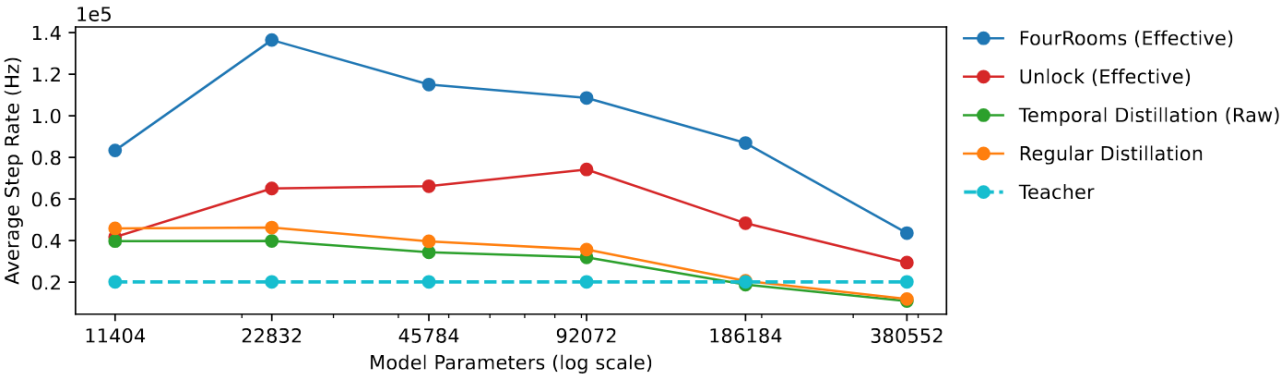


Figure 32: Raw and effective steps per second for different student sizes on a smartphone with a Snapdragon 845 processor

Going from a microcontroller (~100mW) to a smartphone processor (~4W) in Figure 32 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 utilize up to eight threads to parallelize the computations. Since the largest networks benefit the most from parallelization, spatial compression becomes less advantageous on this hardware. This shifts the balance in 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 fair outweighed by its effective benefits.

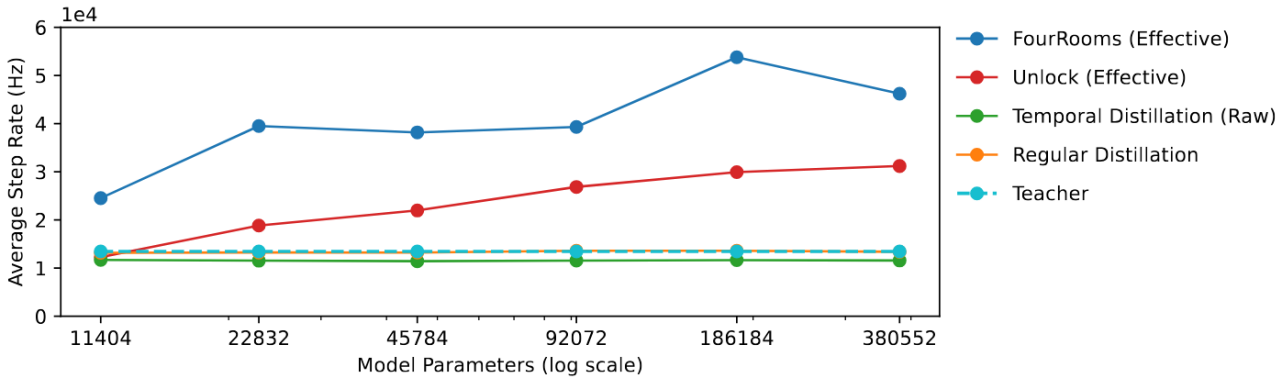


Figure 33: Raw and effective steps per second for different student sizes on an NVIDIA Tesla V100 data centre GPU

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 33. Due to its massive parallelization 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.

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 optimized for deep-learning.

8.6.5. 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. We propose a novel method that can decompose the policy into sequences of repeated actions, which can be learned by a student network through policy distillation. 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 generalize 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) policy distillation, 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.

9. TinyOL

Current TinyML framework aims at enabling inference on the microcontrollers. However, ML models tend to become stale over time as conditions change over time. Additionally, if novel (labeled) data becomes available, the accuracy of the model can be improved by utilizing the novel input-output pairs. TinyOL (Ren, 2021) proposes to train only the last layer of the neural network model to make online learning on-device possible.

If we are not in an energy constrained environment it is still not trivial to efficiently update existing ML models. Continual learning techniques have been proposed that merge both old and new observations when updating the model. However, effects such as catastrophic forgetting, in which the model forgets about previously discovered patterns, are still observed frequently.

In energy constrained environments at least two additional complexities need to be considered:

- Models are often compressed and represented differently
- Memory is often limited, which in most cases does not allow us the store the entire dataset.

9.1. Fine-Tuning of Compressed Models

9.1.1. On-device Online Learning

In order to investigate its feasibility, we recreated the TinyOL architecture and experiments using MNIST hand-written digit dataset. In our recreation experiment, an autoencoder is trained first as the baseline model that imitates the models to be

deployed on the microcontroller. Next, small distribution shifts were added to the input data by rotating the input image by 3 degrees.

Using the distributionally shifted dataset, we performed two different experiments, attempting to recreate the results of TinyOL on MNIST.

The first experiment involves using online learning to reduce the mean squared error loss of the autoencoder on the distributionally shifted data. The results could not be recreated and the loss on new data did not decrease.

The second experiment involves finetuning the autoencoder into a classifier for anomaly detection. In our experiment, we finetune it for digit recognition. In this experiment, similar results as the paper could be recreated, and the model could successfully classify digits from 0-9 with around 90% accuracy after online learning.

9.1.2. Potential Issues with Knowledge Distillation

A naive approach could be to simply train a large teacher network, compress it using knowledge distillation and later use a regular cross-entropy loss to update the student for the new labels. This might work to a limited degree, but it also has some significant flaws that would need to be addressed. A first problem is that there is a significant distribution shift between the soft-targets the student is trained on during distillation and the one-hot encodings used for real labels. This makes it difficult to effectively learn the new features, but also increases the risk of catastrophic forgetting. We hypothesize that this could potentially be somewhat alleviated by applying a transformation to the real labels based on the measured sharpness of the teacher outputs and using a similar KL-divergence loss to reduce the distribution shift. The problem remains however that, due to their limited capacity, the students would not have reached the same accuracy without the dark knowledge provided by the teacher, which is not available for this new data.

The existing work within knowledge distillation that could be relevant to our objective can be divided into three main areas:

1. Online Knowledge Distillation,
2. Continual learning through knowledge distillation

3. Fine-tuning of students in Policy Distillation

9.2. Online Knowledge Distillation

In traditional knowledge distillation, there is a prerequisite fully-trained teacher network with frozen weights that needs to be emulated. This is not the case in online knowledge distillation. Here, either the teacher is trained together with the student, or there is no teacher in the classic sense. These approaches should be more flexible for fine-tuning when new data becomes available, but still require computations outside single edge devices. Qian et al. (al. B. Q.) train a student and teacher at the same time but split this into two modes: an expert mode where the teacher weights are fixed and a learning mode where both models are optimized together. This strikes a balance between a more stable target for the student during the expert mode, while also reducing the distillation gap in the learning mode. An automatic switching strategy is devised to appropriately switch between the two modes. By training the two together, it would allow our already compressed student to be more easily adapted to new data, but it also requires off-device computations.

Deep Mutual Learning on the other hand is a distillation strategy where, instead of using a teacher, multiple smaller students are trained on both ground-truth labels and a distillation target derived from all models. Although ideally, we want to fine-tune completely locally on our ultra-lower-power swarm devices, it might be feasible to cooperatively train a group of such devices. Guo at al. (al. Q. G.) take this approach, with the loss for each student comprised of a cross-entropy loss for with the ground-truth labels and a KL-divergence loss between group-derived soft targets. They propose several methods to generate these soft-targets based on a combination of the student logits and ground-truth labels. Input images are randomly distorted before being processed by each independent student to generate some diversity among the group and increase generalization. Note that in our setting, each device individually gathers its own data for fine-tuning, while the same (distorted) input is required for each student to generate the group-derived soft targets. To apply a similar approach, we would therefore have to share the new data with neighboring swarm devices in the same training group. It also limits the ability of devices to be optimized specifically for their individual position in the swarm.

Chen et al. (al. D. C.) propose a similar method, but also introduce the concept of a group leader, which serves as the final model. This leader learns a weighted aggregate of the predictions from all group members, with weights that are learned similarly to self-attention. They don't apply any distortion to the input, so the difference between students is purely based on a different initialization. The introduction of a leader does improve overall accuracy, but also requires an extra network and needs many training devices to create a single final model. The work by Bhat et al. (Prashant Shivaram Bhat) initially seemed very promising, since it is also focused on online distillation strategies specifically for tiny networks. They define a distillation loss between the embeddings of two students to augment the contrastive loss in the latent space. However, the self-supervised pretraining setting does not fit with our objectives, since we're looking for post-training optimization, not pre-training.

Another strategy is to combine the smaller students into a larger architecture with some initial shared layers that split into multiple branches which form the peer networks. Wu et al. (Gong) Propose such an architecture and use the various heads to derive a peer ensemble teacher and peer mean teacher to increase stability and generalization during training. While this approach was shown to be very performant in terms of final model accuracy, it does once again require a single large and complex network for training, which is not possible on our ultra-low-power swarm devices. Li et al. (Hoiem) also take this approach in the context of heatmap prediction using a pixel-wise KL-divergence. Their method is mostly focused on aggregation of these heatmaps from students, which does not align with our tasks.

9.2.1. Continual Learning through Knowledge Distillation

Some existing work has also applied knowledge distillation in the context of continual learning, with the goal of avoiding catastrophic forgetting. These are generally not designed with compression in mind however, although it could still be possible to apply them on tiny models.

Li et al. (al. Z. L.) propose a method where a model is continually trained on new data, while simultaneously being updated through knowledge distillation using an older version of the model as a teacher. This approach still has the issue that our tiny models

are not able to learn directly from ground-truth labels effectively. It could however still be a valid strategy if the checkpoints used for distillation are updated very frequently, such that the amount of new data learned at every step and therefore change in the model is small enough.

Hou et al. specifically focused on incremental multi-task object categorization. They use distillation to adapt to a new task by first training a new expert teacher specifically on that task. The student then learns this task from the new teacher, while using a small subset of inputs and student predictions from old tasks to reduce forgetting (Retrospection). In this approach, the students don't learn directly from the ground-truth, only through knowledge distillation, so it should likely work in combination with compression. However, the fact that a new teacher needs to be trained specifically on this new task makes it infeasible for us, as well as the fact that our models only need fine-tuning of the same task.

9.2.2. Fine-tuning of students in Policy Distillation

The specific concept of fine-tuning students after compression through distillation has been applied before in the context of reinforcement learning by Green et al. (Sam Green) They were able to do this effectively because they omitted the temperature scaling in the distillation loss, thereby potentially sacrificing some dark knowledge transferred to the student, for a reduction in the distribution shift with the teacher. After pretraining the student through distillation, they also used the regular PPO algorithm for fine-tuning, which slightly improved the student performance. We argue that the difference in state distribution encountered by the student during fine-tuning could be seen as a form of new data. This establishes some precedent that further fine-tuning after distillation can be beneficial when no temperature is applied to the teacher outputs, although the loss used for training in PPO is very different from the cross-entropy loss typically used for supervised classification. It should also be noted that this improvement over distillation is likely due to the reduction in distribution shift between the states encountered by the teacher and the student, which can also be achieved by using student-driven policy distillation instead.

9.2.3. Pruning

Fine-tuning is commonly used in pruning, either as an intermediary step of progressive pruning or to regain accuracy after the pruning step. This is mostly done using the original dataset however, to recover behavior that was already present, not for adapting to new data after the network has been compressed. A promising paper by Gordon et al. (Mitchell A. Gordon) did look at the effects of pruning the BERT model before and after fine-tuning for downstream tasks (transfer learning). They concluded that lower levels of pruning (30-40%) had no negative impact for later downstream tasks and more notably that the model could be pruned once during pretraining, rather than separately for each task, without affecting accuracy. This indicates that it should be feasible for us to apply pruning to our models before sending them to the swarm devices and perform local fine-tuning without affecting accuracy compared to pruning after the fine-tuning step. It should be noted however that this paper used magnitude weight pruning, a form of unstructured pruning, which is only beneficial to runtime performance for devices that are optimized for sparse matrices. This is unfortunately not the case for our swarm devices. Another form of pruning exists, namely structured pruning, that would directly result in lower runtime complexity, but it remains to be seen whether the results from this paper would still be valid. Srivastava et al. (al. S. S.) propose a method that uses pruning for multi-task learning to prevent catastrophic forgetting. They freeze the old parameters when training a new task but apply (unstructured) pruning to provide capacity for the new task instead of increasing the network size. The fact that unstructured pruning is used instead of structured pruning makes less of a difference here since the parameters will be reused for the new task. We could take a similar approach by applying pruning (in addition to another compression method) to the model before sending them to our swarm devices but leaving some capacity for fine-tuning.

9.3. Self-Supervised Learning

A typical problem with continual learning and online adaptation is that swarm devices are expected to continue training while deployed at the edge, but obtaining accurate classification labels for the locally gathered data is typically not possible. We therefore

evaluated whether the user of self-supervised learning can be applied to train the swarm devices in a continual learning scenario.

Self-supervised learning is a type of machine learning approach where the model learns from the input data without relying on human-annotated labels or explicit supervision. In this setting, the model learns to identify patterns, relationships, and structures within the data itself, without being told what to look for or how to classify the data.

In contrast to traditional unsupervised learning, self-supervised methods learn by generating their own supervisory signals from the input data, usually in the form of auxiliary tasks that help the model learn useful representations. For example, in natural language processing, a common pretext task is to predict the next word in a sentence given the previous words. In computer vision, a model might learn to predict the rotation of an image, fill in missing parts of an image, or even reproduce the image from a condensed representation in case of autoencoders. Learning these representations is not the end goal, however, but rather one of the steps in a larger supervised learning pipeline, where these representations are used to improve the performance of a downstream task. Datasets specifically designed with self-supervised learning in mind, such as STL-10, DABS, or CREMA-D therefore typically contain many unlabeled and only a few labelled samples.

Self-supervised approaches have therefore been shown to be effective in learning useful representations from large amounts of unlabeled data as a pre-training step. However, this does not align with our goal of post-training fine-tuning or continual learning, after the model has been extended for the downstream supervised task. Gomez-Villa et al. (Alex Gomez-Villa, 2022) and Fini et al. (E. Fini) have applied the concept of self-supervised learning in a continual learning context, but they still rely on labelled data in their proposed methods.

Work by Dipu Kabir (Kabir) takes a different approach in using the unlabeled data in STL-10 for supervised learning, by including this as an additional background class. In our paper on "Online Adaptation of Compressed Models by Pre-Training and Task-Relevant Pruning", which will be discussed in more detail during the next chapter, we have taken a similar approach by including several additional classes that are not relevant to the

main task to enhance the internal representation of the model. The connections in the model that encode only for these irrelevant classes are then removed using knowledge-based pruning, leaving only the task-relevant connections that have been enhanced by the auxiliary training data. Although we concluded these investigations without being able to apply the typical self-supervised paradigm to online adaptation, they have served as valuable inspiration for an analogous approach through which our models become more adaptable for online fine-tuning or continual learning.

9.4. Online Adaptation of Compressed Models

Deep learning 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 optimized.

Model compression addresses this by reducing the size of neural networks 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. 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 generalization capabilities of compressed models to enable online adaptation without sacrificing compression rates. Generalization 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 generalization (al. K. e., 2022), 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. Methods such as Iterative Magnitude Pruning (IMP) use weight magnitudes to determine their importance, making it difficult to differentiate between task-relevant and additional knowledge used to improve generalization. This is counterproductive for compressing and adapting models for edge devices, as it limits the compression potential.

To address this issue, we leverage Layer-wise Relevance Propagation (LRP), a technique originally developed for interpretability (Bach, 2015). 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 generalization and adaptability when fine-tuning to new subclasses and continual learning on new classes, as illustrated in Figure 34.

In this section, we first investigate how well a model compressed through structured pruning can still be adapted to new data. Then, we evaluate if our pre-training and task-relevant pruning method can effectively improve the generalization 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 generalization in both adaptation scenarios.

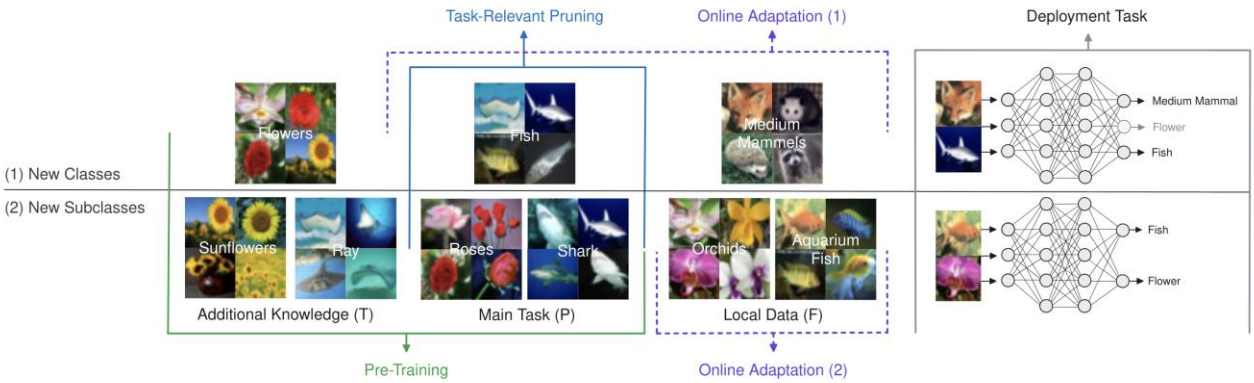


Figure 34: Illustration of the two different online adaptation scenarios on CIFAR-100.

9.4.1. Methodology

An illustration of our proposed approach is shown in Figure 35. Compared to a baseline trained only on the main task (right), the model trained with extra knowledge (left) learns

more general features that are retained after pruning. Due to the more limited training data, the model on the right is prone to memorize 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 generalization 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.

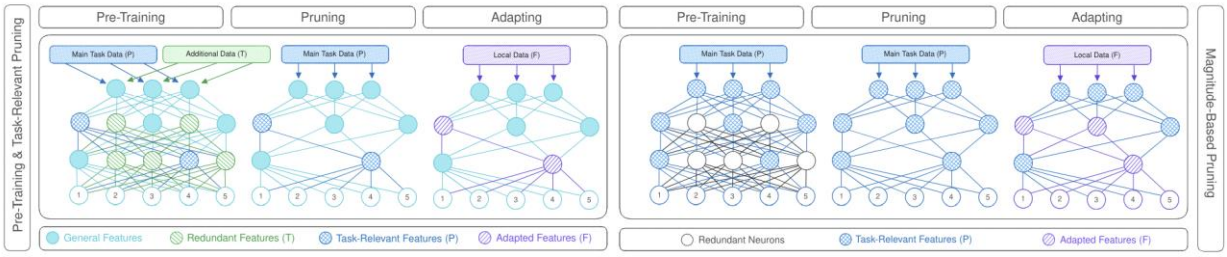


Figure 35: Illustration of the proposed pre-training and task-relevant pruning approach, which aims to enhance the generalization capabilities of compressed models for online adaptation.

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 on class or subclass level, depending on the evaluated scenario. These are then used as input for algorithm 1.

- **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 ∪P, prune it to retain only the knowledge necessary for classifying P and adapt it after pruning using either F (New Subclasses) or P ∪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 = ø, to validate that the extra training knowledge led to better generalization. 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 1: Pre-Training and Task-Relevant Pruning

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

9.4.2. 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 2 contains the resulting splits, grouped in sets of two: one with additional training knowledge and one where T = ø, as shown in Table 1.

Scenario	New Classes		New Subclasses	
Pre-train on	$T \cup P$	$P \quad (T = \varnothing)$	$T \cup P$	$P \quad (T = \varnothing)$
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

Table 1: Overview of our experiment configurations.

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 2: 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 l1 norm in algorithm 1, Our base architecture is ResNet-50 with 23.5M parameters, which is trained using a learning rate 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.

9.4.3. Results

In this section, we evaluate how well the models can be optimized 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 36 confirms that the validation accuracy after online adaptation is consistently higher when the model is pre-trained on additional knowledge to improve generalization. The first point (23.5M) represents the full model accuracy, without any pruning, on P and after online adaptation on F or P ∪ 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 generalization. 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 generalizable features to be pruned, especially for the DomainNet dataset in Figure 36. The IMP experiments show a similar initial improvement, but only for the first pruning iterations, as it cannot differentiate between task-relevant and additional features. This results in more re-training on P to recover the accuracy after pruning, overriding the general features learned in pre-training.

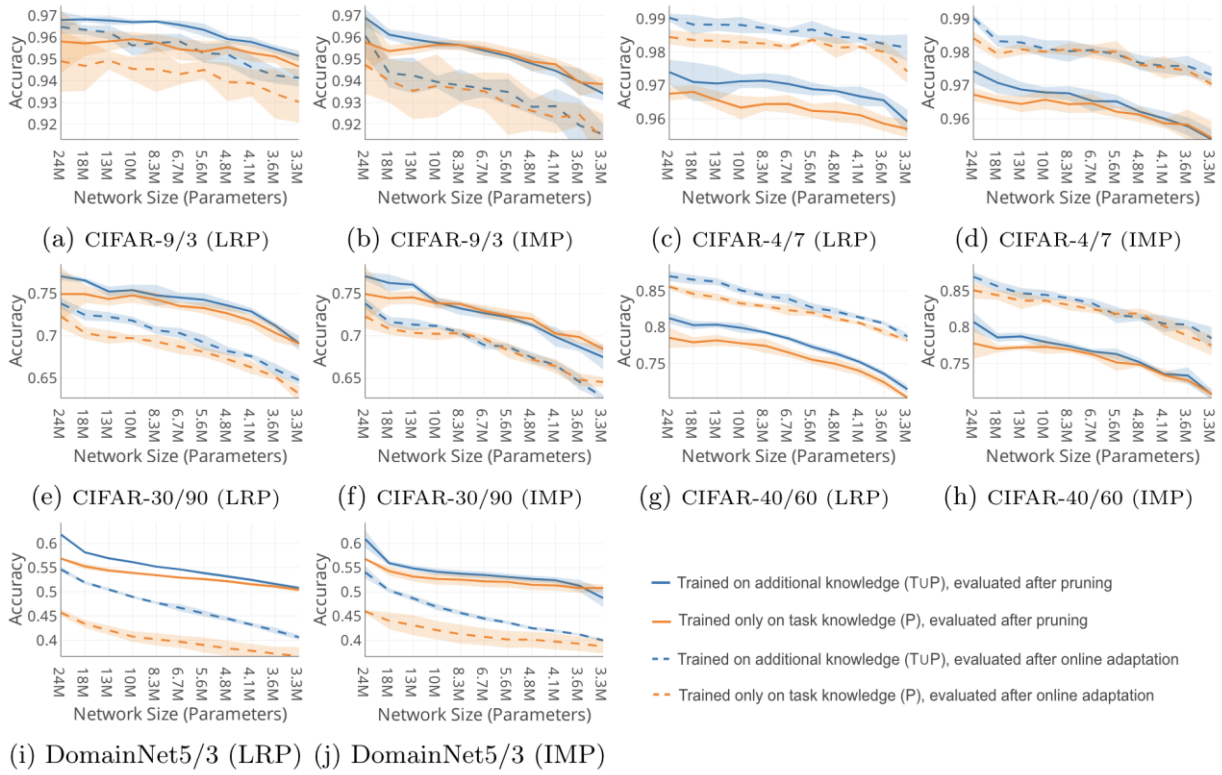


Figure 36: Pruning and validation accuracy for all datasets and adaptation scenarios.

We also conclude that the models are generally able to adapt to new data after pruning with reasonably high accuracy. When adapting to recognize additional classes, the accuracy on F ∪P is generally lower than on P after pruning, but this is because the classification task is harder at the same capacity. The accuracy on the new subclasses is higher than on the original classes after pruning for the CIFAR-10 and CIFAR-100 datasets, since F contains fewer subclasses, making the classification task easier. Experiments on DomainNet show both the largest drop in accuracy, but also the biggest benefit from pre-training on additional knowledge since the style for F (painting) is a lot to closer those in T (real, sketch) than the ones in P (infograph, clipart, quickdraw). A lot of general knowledge is therefore lost during heavier pruning or was never acquired for DomainNet3, making it harder to adapt to the new style with a network specifically optimized for abstract styles. The accuracy on P for the last two IMP iterations with DomainNet5 even drops below Domainet3 as it prunes less intelligently, although their adaptation performance is still higher. Based on these results, we can confidently conclude that our proposed approach based on pre-training and task-relevant pruning

can more effectively compress models while maintaining their generalization capabilities for online adaptation.

9.4.4. Conclusions

Adapting to new data is crucial for edge devices in dynamic environments, but their limited resources constrain the size and complexity of models that can be deployed. This makes model compression techniques such as pruning essential, but these methods typically do not account for online learning scenarios. In this section, we proposed a novel approach that extends the generalization capabilities of pruned models to enable online adaptation without sacrificing compression rates. By pre-training the model on additional knowledge and using Layer-wise Relevance Propagation to compute relevance scores, we identified the neurons that encode task-relevant knowledge. Retaining only these neurons during pruning allowed us to effectively compress the model, while still benefiting from the increased generalization introduced from pre-training.

Our experiments on the CIFAR-10, CIFAR-100 and DomainNet datasets all confirmed that this increased generalization resulted in higher accuracies on the validation set, which was more effectively maintained after pruning than a baseline Iterative Magnitude Pruning approach. This increased generalization in turn significantly improved the accuracy on new data during online adaptation, both when learning to classify new classes and when fine-tuning to new categories of subclasses. Through our proposed approach, a model can be effectively compressed while maintaining high generalization on new data, making it more suitable for online adaptation scenarios.

9.5. Pruning of Low-Precision Models

The pre-training and task-relevant pruning method described above should theoretically allow us to efficiently apply these models on our low-power swarm devices and continue to adapt them based on local deployment conditions. There remains one hurdle, however, before actual deployment can occur on the MAX 87000 target device highlighted in the OpenSwarm proposal. Through pruning, we have significantly reduced the size of the model, but the model still contains floating-point weights and activations. Floating-point operations are not supported, and thus we need

to convert the model to fixed-point arithmetic using 8-bit or lower integer parameters through quantization, a second type of model compression. We evaluated three potential strategies for combining our pruning method with quantization.

1. The simplest approach is to quantize the model after pruning through Post-Training Quantization (PTQ). This method does not require any modification to our existing pruning setup, but it comes with significant drawbacks in terms of reduced accuracy. PTQ is also not ideal in the context of online adaptation, as the types of quantization functions used in this context are not optimized for training, in contrast to non-linear function often used with Quantization-Aware Training (QAT). It remains to be seen if our compressed models, which are enhanced specifically for fine-tuning, can still effectively adapt to new data once the model has been quantized. One potential solution is to maintain a copy of the full-precision model for adapting to new data and re-apply PTQ after each update for re-deployment. This still allows for online adaptation to new data, but this adaptation can no longer happen at the edge itself.

2. The second strategy that does not require any modification to our pruning setup is to still quantize the model after pruning but use QAT to optimize the model to account for the inaccuracies introduced by quantization. This should result in higher accuracies than just PTQ and be more compatible with post-quantization training when adapting the model to new data on the edge. However, QAT requires changing the weights of the model after pruning, which defeats the purpose of the knowledge-aware pruning process that relies on the value of the weights to determine their importance. Although this will likely work better than the first strategy in terms of final accuracy, it might require less aggressive pruning to maintain optimal performance.

3. The final option is the integrate QAT in the knowledge-aware pruning process, where low-precision weights are pruned based on the quantized model. This method has the greatest compression potential, as the model is optimized for quantization from the start and the pruning process can more accurately determine the importance of the final low-precision weights. However, this method is the most complex and requires significant modifications to the existing pruning setup. The LRP method would need to be modified to compute the relevance score of integer weights. Additionally, existing implementations for QAT and pruning require specialized replacements of standard

deep learning modules, which are not interoperable. Therefore, new implementations for these modules would have to be created based on the quantization modules provided by ADI and our modules for structured knowledge-aware pruning. We raised an issue to support this use case in the neural_compressor library developed by Intel, as this is a popular library recommended by the developers of Pytorch Lighting, that supports both structured pruning and QAT, but not simultaneously. Unfortunately, after an initial confirmation of the value of this use case, no further developments have occurred.

We conclude that the final strategy is the most promising in terms of accuracy and compression potential, but it is also the most complex and requires significant further development effort. Considering that, a trade-off can be made based on the first two approaches, depending on how essential it is that online adaptation can occur on the edge itself.

10. Continual Learning

An additional interesting angle is the possibility of applying Continual Learning (CL) algorithms on micro-controllers. On-device continual learning is currently only explored by Avi et al. (Avi) in the context of gesture and visual smart sensors. On-device continual learning enables the edge-device to learn new classes / environment on-the-fly without forgetting the old classes / environment. This extension keeps the model always up to date with dynamic requirements.

For example, a device can drift from shallow-water to deep-water, when we apply online learning, the model will adapt to predict accurately in the new scenario. However, when the model drifts back to shallow-water region, it will need to adapt again in this 'new' environment. With continual learning, we the model will try to keep as much knowledge about shallow-water as possible while adapting to the deep-water environment.

10.1. Condensed Latent Replay (CLaRe)

The demand for a machine learning model that can learn continuously from new data is increasing in the big-data era. The research field of continual learning (CL) aims to solve the catastrophic forgetting problem, where the model forgets past knowledge when trained on new incoming data. However, the majority of the work in CL assumes ample time and computational resources, which is not realistic. In this work, we tackle the more difficult and realistic real-time online continual learning (OCL) problem at the edge devices, where the model needs to learn from a streaming dataset in real-time, and need to run with constraint computational resources.

We propose a novel framework with a Condensed Latent Replay buffer (CLaRe). CLaRe decouples, and parallelizes the processing of incoming streaming data, and the training of the machine learning model. The incoming streaming data is only used for statistical feature matching process to generate a synthetic replay buffer. The machine learning model only continuously learns from the replay buffer. This approach completely

decouples the requirement of the need for real data during the training phase, making it possible to handle extremely fast incoming streaming data.

10.1.1. CLaRe framework architecture

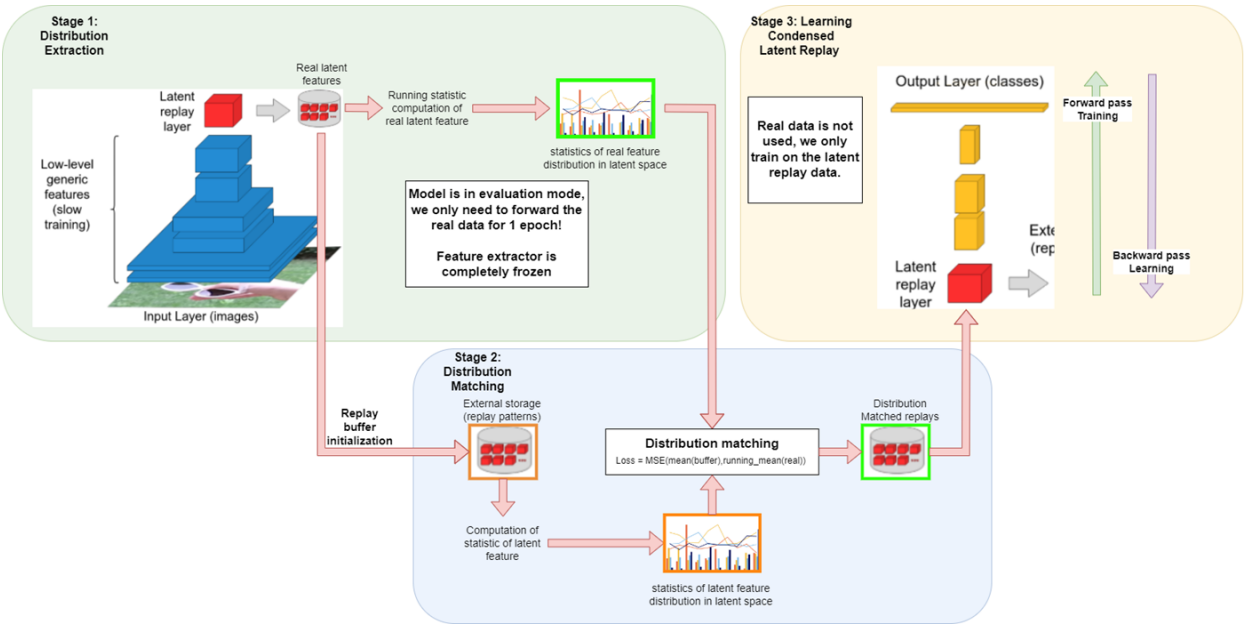


Figure 37: CLaRe framework architecture

The CLaRe framework (Figure 37) consists of three different stages. Stage 1 of CLaRe is inspired by the previous study (Pellegrini) which only uses latent representation as the replay buffer. In stage 1, the input is passed though a frozen and pretrained feature extraction network. Next, the latent feature of some samples are selected to be added or to replace other samples in the replay buffer. Afterward, the centroid for each class in the latent space is updated with a running mean computation.

In stage 2, a maximum mean discrepancy based feature/distribution matching method (Zhao, 2023) is used to optimize the latent replay buffer. I.e., we optimize the data in the replay buffer by minimizing the maximum mean discrepancy between the mean of the real data distribution and the one that exists in the synthetically generated latent replay buffer.

Finally, in stage 3, the classification layers learn from the distribution aligned replay buffer, this process can be invoked at any time and will have a constant computational cost, as the replay buffer has a constant size.

10.1.2. CLaRe initial results

Initial experiments are performed with the AR1*-free as baseline on the CORe50 dataset in the NiCv2-391 setting (Lomonaco) In this setting, the model is first pretrained using 3000 images of 10 classes in the CORe 50 dataset. Then, it needs to sequentially learn from small batches of 300 images of one class. The class in the small batch may be a novel class which the model has never seen before. The performance of our framework and the baseline is measured by the final accuracy of the model on the test set after sequentially learning all batches. The results of the experiments are shown in the table below. The experiments are repeated 3 times and the mean and standard deviations are reported.

Category	Methods	CORe50		
Replay buffer size (for replay-based methods)		500	1000	1500
Lower bound	Naive Fine tuning †	10.0%	10.0%	10.0%
Replay-based	AR1*free	57.97% ± 0.42%	69.51% ± 0.43%	73.34% ± 0.47%
Ours	CLaRe without condensation	66.93% ± 1.4%	71.05% ± 0.8%	73.80% ± 0.79%
	CLaRe (10 condensation iteration)	**71.39%** ± **1.5%**	**75.65%** ± **0.35%**	**76.8%** ± **0.23%**
Upperbound	Joint learning †	85.0%	85.0%	85.0%

Table 1. Comparison of the final accuracy of the different continual learning approaches on the NiCv2-391 scenario. The † symbol denotes a temporary placeholder value taken from the paper of the AR1*free baseline. CLaRe outperforms AR1*free significantly on lower replay buffer size. CLaRe without condensation (e.g. the model only learn from the replay buffer with real images) also yields a notable performance increment compared to the baseline.

From the table, we can see that CLaRe and CLaRe without condensation outperforms the AR1*FREE baseline notably. Moreover, CLaRe with a replay buffer size of 500 can outperform AR1*FREE with a buffer size of 1000, and the increase in performance is more significant when the buffer size is small. Thus, CLaRe utilizes the scarce replay buffer space of the edge device more efficiently compared to the baselines. The same phenomenon can be observed when comparing CLaRe with and without condensation. With a replay buffer size of {500,1000,1500}. CLaRe with condensation outperforms CLaRe without condensation for {4.46%,4.6%,3%} respectively. This denotes that the condensation stage is crucial, especially when the replay buffer size is small.

In Figure 38, we visualize the evolution of accuracy during the continual learning stage of CLaRe in green, CLaRe without condensation in orange and AR1*free baseline in blue, in the figure below, the accuracy is measured every 50 tasks/encountered batches.

We observe that CLaRe with and without condensation outperforms the baseline in the whole CL stage, exception on the first encountered batch. This is expected, as AR1*FREE learns from the full 3000 images at the first task, while CLaRe only learns from the condensed 500 images, which contains less information.

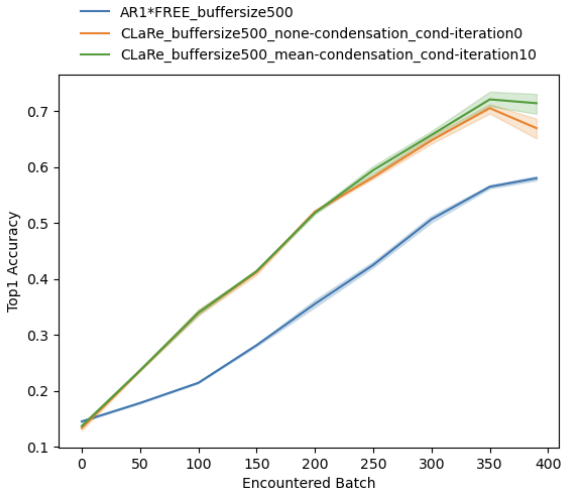


Figure 38: Evolution of accuracy during the continual learning stage of CLaRe

11. Conclusions

In this report, we discussed how we can effectively apply both existing concepts and novel contributions to enable efficient online learning on ultra-low-power swarm devices. Through TinyML and model compression, we can deploy pre-trained machine learning models on energy-constrained devices, such as the MAX78000 microcontroller on which we validated the KPIs. Our methods in the realm of TinyOL then allow us to update these models on the edge as new data arrives. Finally, we introduced a continual learning framework that balances the usage of both old and novel observations to prevent catastrophic forgetting.

For TinyML, we started by performing a literature study on the existing SOTA methods for pruning, knowledge distillation and quantization, and evaluated their applicability to our use cases. The final deployment strategy requires a combination of all three methods, to obtain a model that is both small enough to fit on the limited weight capacity of the MAX78000 and remains accurate enough for low-precision inference. To address the limitations of the existing work in this domain, we developed the following novel contributions:

1. We proposed a method based on knowledge distillation for the compression of deep reinforcement learning policies that operate on continuous action spaces, allowing their deployment on the ultra-low-power swarm devices, something that was previously not possible.
2. To further reduce the average added current draw of running our deep learning models, we developed a novel method for intelligently predicting when a new prediction will be needed, so the microcontroller only needs to wake up from a low-power state when necessary.

Combined, these methods enable the efficient deployment of powerful deep learning models on ultra-low-power swarm devices, while intelligently managing their energy requirements.

We then investigated the feasibility of fine-tuning these compressed models on the edge, in a transition to TinyOL, by first exploring and replicating existing work in this field. Unfortunately, some compression methods change how the model represents its predictions, making it no longer feasible to update the model with new data using the original training method. Additionally, since these compression methods are designed to reduce the capacity of the model by only retraining the most important features for the task, this also potentially involves removing features or capacity that would be beneficial for generalization to new data.

We therefore introduced a novel training and compression approach for improved online adaptation of compressed models on the edge. For this, we first pre-train on additional data to enhance the generalization capabilities of the model, but this can also make it more difficult to identify redundant neurons for pruning. We therefore leveraged layer-wise relevance propagation to identify the features that encode task-relevant knowledge, allowing us to prune the model while maintaining the general features that improve online adaptation. By applying this approach, we ensure that the model remains capable of adapting to new data, while still being effectively compressed to fit on the target swarm devices.

Finally, we focused on the challenge of avoiding catastrophically forgetting the knowledge learned during the first training phase, when updating the model to local deployment conditions.

12.Bibliography

[17] Soto, P. e. (n.d.). Network intelligence for nfv scaling in closed-loop architectures. *IEEE Communications Magazine 61, 66–72 (2023).*

al., B. Q. (n.d.). Switchable Online Knowledge Distillation. *Computer Vision - ECCV 2022 - 17th European Conference*.

al., D. C. (n.d.). Online Knowledge Distillation with Diverse Peers. *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*.

al., K. e. (2022). A Broad Study of Pre-Training for Domain Generalization and Adaptation'.

al., Q. G. (n.d.). Online Knowledge Distillation via Collaborative Learning. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020*.

al., S. S. (n.d.). Adaptive Compression-based Lifelong Learning. *30th British Machine Vision Conference 2019*.

al., T. H. (n.d.). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *Proceedings of the 35th International Conference on Machine Learning*.

al., V. M. (2016). Asynchronous Methods for Deep Reinforcement Learning. *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016.*

al., Z. L. (n.d.). Online Knowledge Distillation for Efficient Pose Estimation. *2021 IEEE/CVF International Conference on Computer Vision*.

Alex Gomez-Villa, B. T. (2022). Continually Learning Self-Supervised Representations with Projected Functional Regularization. *CVPR Workshops* , 3866-3876.

Alhartomi, M., Salh, A., Audah, L., Alzahrani, S., & Alzahmi, A. (n.d.). Enhancing Sustainable Edge Computing Offloading via Renewable Prediction for Energy Harvesting. *IEEE Access 2024, 12, 74011–74023.*

Avé, T. D. (2024). Policy compression for intelligent continuous control on low-power edge devices. *Sensors, 24*.

Avé, T., Mets, K., De Schepper, T., & Latre, S. (n.d.). Quantization-aware Policy Distillation (QPD). *In Proceedings of the Deep Reinforcement Learning Workshop NeurIPS, Virtual, 9 December 2022*.

Avé, T., Soto, P., Camelo, M., De Schepper, T., & Mets, K. (n.d.). Policy Compression for Low-Power Intelligent Scaling in Software-Based Network Architectures. *Proceedings of the NOMS 2024 IEEE Network Operations and Management Symposium, Seoul, Republic of Korea, 6–10 May 2024; pp. 1–7*.

Avi, A. A. (n.d.). Incremental online learning algorithms comparison for gesture and visual smart sensors. *2022 International Joint Conference on Neural Networks (IJCNN)*.

Azar, A., Koubaa, A., Ali Mohamed, N., Ibrahim, H., Ibrahim, Z., Kazim, M., . . . al., e. (n.d.). Drone Deep Reinforcement Learning: A Review. *Electronics 2021, 10, 999*.

Bach, S. B. (2015). On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one, 10(7)*.

Bellemare, M. G. (n.d.). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research 47, 253–279 (2013)*.

Berseth, G., Xie, C., Cernek, P., & de Panne, M. (n.d.). Progressive Reinforcement Learning with Distillation for Multi-Skilled Motion Control. *In Proceedings of the International Conference on Learning Representations, Vancouver, BC, Canada, 30 April–3 May 2018*.

Bertolini, A. M. (n.d.). Power output optimization of electric vehicles smart charging hubs using deep reinforcement learning. *Expert Systems with Applications 201, 116995 (2022)*.

Biedenkapp, A. R. (2021). Temporl: Learning when to act.

Blalock, D. W. (n.d.). What Is the State of Neural Network Pruning?

Bu, F. &. (n.d.). A smart agriculture iot system bas ed on deep reinforcement learning. *Future Generation Computer Systems 99, 500–507 (2019)*.

Chevalier-Boisvert, M. e. (n.d.). Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks.

Cristian Buciluundefined, R. C.-M. (n.d.). Model Compression. *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Czarnecki, W., Pascanu, R., Osindero, S., Jayakumar, S., Swirszcz, G., & Jaderberg, M. (n.d.). Distilling Policy Distillation. *In Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, Okinawa, Japan, 16–18 April 2019; Chaudhuri, K., Sugiyama, M., Eds.; PMLR: London, UK, 2019; Volume 89, pp. 1331–1340*.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., & Abbeel, P. (n.d.). Benchmarking Deep Reinforcement Learning for Continuous Control. *In Proceedings of the 33rd International Conference on Machine Learning, New York City, NY, USA, 19–24 June 2016; Balcan, M.F., Weinberger, K.Q., Eds.; PMLR: London, UK, 2016; Volume 48, pp. 1329–1338*.

Duisterhof, B. P. (2021). Tiny robot learning (tinyrl) for source seeking on a nano quadcopter.

E. Fini, V. G.-P. (n.d.). Self-Supervised Models are Continual Learners. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Gong, G. W. (n.d.). Peer Collaborative Learning for Online Knowledge Distillation. *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*.

Gordon, M. A. (n.d.). Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning.

Green, S., Vineyard, C., & Koç, Ç. (n.d.). Distillation Strategies for Proximal Policy Optimization.

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (n.d.). 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, 10–15 July 2018; Dy, J.G., Krause, A., Eds.; PMLR: London, UK, 2018; Volume 80, pp. 1856–1865*.

Hinton, G., Vinyals, O., & Dean, J. (n.d.). Distilling the Knowledge in a Neural Network.

Hoiem, Z. L. (n.d.). Learning Without Forgetting. *Computer Vision - ECCV 2016 - 14th European Conference*.

Kabir, H. M. (n.d.). Reduction of Class Activation Uncertainty with Background Information.

Kalyanakrishnan, S. e. (2021). An analysis of frame-skipping in reinforcement learning.

Konda, V., & Tsitsiklis, J. (n.d.). Actor-Critic Algorithms. *In Proceedings of the Advances in Neural Information Processing Systems 12, NIPS Conference, Denver, CO, USA, 29 November–4 December 1999; Solla, S.A., Leen, T.K., Müller, K., Eds.; The MIT Press: Cambridge, MA, USA, 1999; pp. 1008–1014*.

Labrosse, J. J. (1998). *MicroC/OS-II.* R&D Books.

Lai, K., Zha, D., Li, Y., & Hu, X. (n.d.). Dual Policy Distillation. *In Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI 2020, Yokohama, Japan, 11–17 July 2020; Bessiere, C., Ed.; IJCAI: San Diego, CA, USA, 2020; pp. 3146–3152*.

Lakshminarayanan, A. S. (n.d.). Dynamic action repetition for deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence 31 (2017)*.

Lei, L., Tan, Y., Zheng, K., Liu, S., Zhang, K., & Shen, X. (n.d.). Deep Reinforcement Learning for Autonomous Internet of Things: Model, Applications and Challenges. *IEEE Commun. Surv. Tutor. 2020, 22, 1722–1760*.

Lin, J. e. (n.d.). On-Device Training Under 256KB Memory . *Advances in Neural Information Processing Systems 35 (2022): 22941-22954.*

Lomonaco, V. M. (n.d.). Rehearsal-Free Continual Learning over Small Non-IID Batches. *In CVPR Workshops (Vol. 1, No. 2, p. 3)*.

Matthias Hutsebaut-Buysse, F. G. (n.d.). Directed Real-World Learned Exploration. *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Mismar, F., Evans, B., & Alkhateeb, A. (2020). Deep Reinforcement Learning for 5G Networks: Joint Beamforming, Power Control, and Interference Coordination. *IEEE Trans. Commun.*, 1581–1592.

Mitchell A. Gordon, K. D. (n.d.). Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning. *Proceedings of the 5th Workshop on Representation Learning for NLP, RepL4NLP@ACL 2020*.

Mnih, V. e. (2013). Playing atari with deep reinforcement learning.

Munoz, G. B. (n.d.). Deep reinforcement learning for drone delivery. *Drones 3 (2019)*.

Mysore, S., Mabsout, B., Mancuso, R., & Saenko, K. (n.d.). Honey. I Shrunk The Actor: A Case Study on Preserving Performance with Smaller Actors in Actor-Critic RL. *In Proceedings of the 2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, 17–20 August 2021; pp. 1–8*.

Patil, S. G. (n.d.). POET: Training neural networks on tiny devices with integrated rematerialization and paging. *International Conference on Machine Learning (pp. 17573-17583). PMLR.*

Pellegrini, L. G. (n.d.). Latent replay for real-time continual learning. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Prashant Shivaram Bhat, E. A. (n.d.). Distill on the Go: Online Knowledge Distillation in Self-Supervised Learning. *IEEE Conference on Computer Vision and Pattern Recognition Workshops*.

Profentzas, C. A. (n.d.). MiniLearn: On-Device Learning for Low-Power IoT Devices. *EWSN*.

Ren, H. A. (2021). Tinyol: Tinyml with online-learning on microcontrollers. *International joint conference on neural networks (IJCNN)* .

Rusu, A., Colmenarejo, S., Gülçehre, Ç., Desjardins, G., Kirkpatrick, J., Pascanu, R., . . . Hadsell, R. (n.d.). Policy Distillation. *In Proceedings of the 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, 2–4 May 2016; Bengio, Y., LeCun, Y., Eds.; ICLR: Appleton, WI, USA, 2016*.

Sam Green, C. M. (n.d.). Distillation Strategies for Proximal Policy Optimization.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (n.d.). Proximal Policy Optimization Algorithms.

Scott Fujimoto, H. v. (n.d.). Addressing Function Approximation Error in Actor-Critic Methods. *Proceedings of the 35th International Conference on Machine Learning*.

Selander, G., Mattsson, J. P., & Palombini, F. (2024). *Ephemeral Diffie-Hellman Over COSE (EDHOC).* Internet Engineering Task Force.

Sharma, S. L. (2017). Learning to repeat: Fine grained action repetition for deep reinforcement learning.

Stanton, S., Izmailov, P., Kirichenko, P., Alemi, A., & Wilson, A. (n.d.). Does Knowledge Distillation Really Work? *In Proceedings of the Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, Virtual, 6–14 December 2021; Ranzato, M., Beygelzimer, A., Dauphin, Y.N., Liang, P., Vaughan, J.W.,* .

Tang, M., & Wong, V. (n.d.). Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing Systems. *IEEE Trans. Mob. Comput. 2022, 21, 1985–1997*.

Tang, Y., & Agrawal, S. (n.d.). Discretizing Continuous Action Space for On-Policy Optimization. *In Proceedings of the AAAI-20, 34th Conference on Artificial Intelligence, 32nd Innovative Applications of Artificial Intelligence Conference, 10th Symposium on Educational Advances in Artificial Intelligence, New York, NY, USA, 7–12 February 2020; AAAI P*.

Todorov, E., Erez, T., & Tassa, Y. (n.d.). MuJoCo: A physics engine for model-based control. *In Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, Vilamoura-Algarve, Portugal, 7–12 October 2012; pp. 5026–5033.*

Vilajosana, X. a. (2020). IETF 6TiSCH: A Tutorial. *IEEE Communications Surveys & Tutorials*, 595--615.

Watteyne, T. a. (2015). *RFC7554 - Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement.* Internet Engineering Task Force.

Watteyne, T., Handziski, V., Vilajosana, X., Duquennoy, S., Hahm, O., Baccelli, E., & Wolisz, A. (2016). Industrial Wireless IP-Based Cyber Physical Systems. *Proceedings of the IEEE*, 1025-1038.

Wei, D., Guo, C., & Yang, L. (n.d.). Intelligent Hierarchical Admission Control for Low-Earth Orbit Satellites Based on Deep Reinforcement Learning. *Sensors 2023, 23, 8470*.

Xu, Z., Wu, K., Che, Z., Tang, J., & Ye, J. (n.d.). Knowledge Transfer in Multi-Task Deep Reinforcement Learning for Continuous Control. *In Proceedings of the Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, Virtual, 6–12 December 2020; Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H., Eds.; Ne*.

Xun, M., Yao, Y., Yu, J., Zhang, H., Feng, S., & Cao, J. (n.d.). Deep Reinforcement Learning for Delay and Energy-Aware Task Scheduling in Edge Clouds. *In Proceedings of the Computer Supported Cooperative Work and Social Computing, Harbin, China, 18–20 August 2023; Sun, Y., Lu, T., Wang, T., Fan, H., Liu, D., Du, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2024; pp. 436–450*.

Xun, M., Yao, Y., Yu, J., Zhang, H., Feng, S., & Cao, J. (n.d.). Deep Reinforcement Learning for Delay and Energy-Aware Task Scheduling in Edge Clouds. *In Proceedings of the Computer Supported Cooperative Work and Social Computing, Harbin, China, 18–20 August 2023; Sun, Y., Lu, T., Wang, T., Fan, H., Liu, D., Du, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2024; pp. 436–450.*

Zhang, Y., & Chen, P. (n.d.). Path Planning of a Mobile Robot for a Dynamic Indoor Environment Based on an SAC-LSTM Algorithm. *Sensors 2023, 23, 9802*.

Zhao, B. &. (2023). Dataset condensation with distribution matching. *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*.

13.