universität innsbruck

Department of Computer Science

Exploring Hypernetworks for Continual Learning from Demonstration

Sebastian Bergner

B.Sc. Thesis

Supervisor: Sayantan Auddy

19th February 2024

Abstract

Learning from demonstration (LfD) offers a natural and intuitive approach to teach robots a certain skill, eliminating the need for explicit programming by a human instructor with expertise in robotics. However, most works in the field of LfD only teach a single motion, which is impractical in real-world situations. Recent work, such as those by Auddy et al. (2023a,b), address this limitation by proposing methods using hypernetworks to enable learning of multiple real-world motions, but many aspects that influence the performance of hypernetworks were not investigated thoroughly. The main objective of this thesis is to do a deep dive into these aspects. Specifically focusing on optimizers, initializers, and neural network architecture and investigating their effects on the continual learning process. We found that Adam as optimizer and Kaiming as initializer in combination with a wider hypernetwork and deeper targetnetwork were performing the best. These findings highlight the importance of a careful selection of deep learning aspects in continual learning from demonstration.

Acknowledgments

I would like to use this opportunity to express my gratitude and appreciation to the following people who have played a major role in the completion of this thesis.

First, I would like to express my gratitude to my supervisor Sayantan Auddy. Thank you for giving me the opportunity to work on this exciting and interesting thesis thank you for your guidance and support, and for all the time that you have spared to discuss progress as well as problems. I am truly grateful for your supervision and mentoring.

To my beloved girlfriend Anna and to my father Günther, thank you for your continuous support and encouragement throughout the time that I've worked on this thesis.

Last but not least I would like to thank my friends Fabian Kranewitter and Stefan Wagner for their support, time for discussions, and feedback which helped me.

In addition, I would like to thank my whole family, all professors and lecturers, the IIS team,the institution and all other persons who guided me to my profession and supported me in my decisions.

Contents

List of Figures

List of Tables

Declaration

By my own signature I declare that I produced this work as the sole author, working independently, and that I did not use any sources and aids other than those referenced in the text. All passages borrowed from external sources, verbatim or by content, are explicitly identified as such.

Signed:.....Date:.....

Chapter 1

Introduction

In recent years, the practical applications of robots have evolved to work in complicated real-world scenarios, requiring a complex level of interaction within dynamic environments. Robot systems have to balance between repetitive yet coherent actions, which involves re-adjusting to adapt to changes. This would require to have a developer with the needed knowledge on-site, which is often not a viable option. This calls for simplification to program or train the Robot in an abstract fashion. The technique for training a robot by showing the desired action is called programming by demonstration or learning from demonstration Billard et al. (2016). There are several ways to demonstrate the desired trajectories to a robot. Ravichandar et al. (2020) have categorized them into 3 classes: Kinesthetic teaching, teleoperation and passive observation.

In kinesthetic teaching the robot is moved by the user to do the desired action and all required data is recorded by already built-in sensors. Teleoperation, requires the user to control the robot using a keyboard, joystick or a similar input device. Whilst in passive observation the robot itself is not part of the environment and data collection is done by sensors like cameras, this is pretty simple acquiring the data by itself but comes with problems caused for example by visual occlusion. Each method offers some advantages and disadvantages. A novel way of utilizing kinesthetic learned data has emerged in recent years where a dynamic vector field is constructed to represent a corresponding trajectory as used by Hersch et al. (2008), Urain et al. (2020), Khansari Zadeh and Billard (2011) and Kolter and Manek (2019). Using this allows for learning the required parameters for one particular trajectory in the environment, which may suffice in the rarest cases, but would be impractical as most problems consist of more than just one trajectory. Humans are able to learn and remember movements thanks to the remarkable brain's abilities. The brain has developed throughout many evolutions built-in mechanisms that enable learning important information, forgetting unnecessary data, and abstracting to enable more efficient learning. Such a technique would prove advantageous in this case as the robot system would be able to learn multiple trajectories and store them in a condensed and efficient way.

Many approaches for continual learning exist Parisi et al. (2018) that try to avoid catastrophic forgetting, which means that the neural network would forget essential parts of a certain task or even the entire task. Continual learning can be applied in various contexts, including lifelong learning. This refers to a system that is capable of learning and improving its performance over the entire lifetime, thus not only adjusting the parameters during the training phase but also after training. This would be helpful as often enough training data is very limited and learning with that can lead to false predictions in the real-world, without any proper fix besides retraining with a larger, more general, dataset. There have been many methods in recent years which borrow ideas from nature.

Most of the research in the field of continual learning is in the field of computer vision such as Shin et al. (2017), Aljundi et al. (2018) and von Oswald et al. (2019). Only a few explore continual learning in the field of robotics like Gao et al. (2021) or Huang et al. (2020).

One of the current state-of-the-art works in continual learning from demonstration for robots

is proposed by Auddy et al. (2023a) they use chunked and regular hypernetworks introduced by von Oswald et al. (2019) and Ha et al. (2016) to generate parameters for either Neural Ordinary Differential Equation (NODE) solver Chen et al. (2019) or a recent novel stable Neural Ordinary Differential Equation (sNODE) solver Auddy et al. (2023b) Kolter and Manek (2019).

In our work, we use the same approach to this problem, but explore the impact on performance of utilizing different initializers, optimizers and architectures of the hypernetwork and targetnetwork.

This is interesting and valuable as deep learning systems such as this, depend on many factors some of the most influential are the chosen initializer, optimizer and the used architecture. The previous work (Auddy et al. (2023a), Auddy et al. (2023b)) in the field of continual learning from demonstration for robots, has not explored the impact of these aspects.

Through a series of empirical experiments, we found that, the optimizers Adam Kingma and Ba (2017) and RMSProp Tieleman and Hinton (2012) were both able to outperform SGD Rosenblatt (1958) in all scenarios. In the case of initializers we found that Adam in combination with Kaiming initialization He et al. (2015) performed overall the best with Principled Weight Initialization for Hypernetworks Chang et al. (2020) performing similarly well. Xavier initialization struggles with chunked hypernetworks possibly due to a gain parameter issue. Architecture-wise we observed a better performance with wider hypernetworks and deeper targetnetworks with some nuances. All of our experiments were conducted with the $RoboTasks9$ dataset Auddy et al. (2023b), as it offers real robot data that enables a realistic challenge for continual learning.

In the upcoming chapter 2, we delve into similar and related work. Following, in the Background chapter 3 we explain the utilized algorithms and technologies. We elaborate on the setup parameters and experimental procedure in Methods 4. Thereafter, we present our results in chapter 5 and finally, discuss the experiments and their impact on the field of robotics and hypernetworks in general in the conclusion chapter 6.

Chapter 2

Related Work

Continual learning, which is also known as lifelong learning, is a paradigm in machine learning where a system tries to continuously adapt to newly seen data whilst retaining past learned knowledge. The motivation behind such a system is to be able to use systems in real-world applications, where static systems would prove unsuitable as such an environment is highly dynamic. In addition, these systems should avoid catastrophic forgetting, where it forgets prior knowledge. There exist several different approaches to continual learning, that try to avoid catastrophic forgetting, with major classes being *replay*, *regularization* and *dynamic architectures* Parisi et al. (2018). *Dynamic architectures* can extend the set of parameters for each new task, for example, Rusu et al. (2022) proposed *progressive networks*, where a neural network trained on a task is locked down, and any new task is allocated to a newly generated sub-network. Additionally, these networks reuse feature-mapping knowledge from previously learned tasks. This approach is able to prevent catastrophic forgetting, at the expense of rapid increase of parameters and added complexity. *Regularization* approaches try to avoid drastic changes to parameters θ of a network when learning a new task. Kirkpatrick et al. (2017) proposed the elastic weight consolidation (EWC) model, which uses a quadratic penalty on the difference of parameters in the Loss function $\mathcal{L}$. This avoids harsh changes to the parameters, thus allowing the retention of previously learned tasks, but at the same time makes fitting new tasks more difficult. *Replay* is another method to avoid catastrophic forgetting, it works by showing the neural network previous data either fully or partially. Some methods compress the data for example using a generative approach, which is able to extract the most important data. Shin et al. (2017) used this approach for a CNN, while Gao et al. (2021) applied it in robotics. Hypernetworks have been successfully used for continual image classification by von Oswald et al. (2019).

Learning from Demonstration (LfD) in robotics is a technique where a human demonstrator is able to teach a robot, trajectories without prior knowledge Billard et al. (2016). The training data can easily be collected either (1) by kinesthetically guiding the robot and recording the output of the sensors in each joint, or (2) by utilizing a motion recording system where the desired trajectory, which is shown by a human, can be extracted, or (3) by using a teleoperation devices that controls the robot movements, which again are recorded Billard et al. (2016) Ravichandar et al. (2020). To utilize the recorded data one of many available algorithms can be applied. Hersch et al. (2008) Khansari Zadeh and Billard (2011) utilize Gaussian mixture models/Gaussian mixture regression, while Ijspeert et al. (2002) for example, use sets of nonlinear differential equations to form a control policy. A fairly recent trend of encoding recorded trajectories has emerged, where the trajectory is converted into a vector field Hersch et al. (2008); Urain et al. (2020); Auddy et al. (2023a); Kolter and Manek (2019); Ijspeert et al. (2002); Khansari Zadeh and Billard (2011). Auddy et al. (2023a) approached LfD utilizing *Neural Ordinary Differential Equation solvers* (NODEs) which showed great success, despite NODEs not being utilized for LfD before.

Auddy et al. (2023b) were also able to improve on their state-of-the-art performance in continual learning from demonstration using a stable Neural ODE (sNODE) solver in combination with hypernetworks. Here the sNODE solver is a system that takes time steps in addition to a start state and tries to predict a stable trajectory prediction, implying that the system ensures a boundedness and convergence to a steady state. The hypernetwork or chunked hypernetwork acts as the continual learning system to generate the parameters of the solver.

Our work is similar to theirs, employing hypernetworks in combination with NODE and stable NODE for continual learning from demonstration. However, we explore different aspects trying to improve the capabilities of the system. These aspects are the optimizer, initializer, and neural network architecture. Each of these has a substantial impact on the performance of a neural network. In our case, it influences both the LfD and the CL system. This knowledge is crucial as robots that utilize such a system should follow all of the trained trajectories with the least possible error since humans or the robot itself could be harmed if the system deviates significantly. To represent a real-world situation for this system we utilized the RoboTask9 Auddy et al. (2023b) for our experiments.

Chapter 3

Background

The system utilized in this work consists of two subsystems, consisting of one for LfD and one for CL. The LfD component utilizes one of two neural ordinary differential equation solvers, NODE or stable NODE. For the CL side we use either a hypernetwork or a chunked hypernetwork. This work explores the intricacies of different deep learning aspects. We explore the impact of different initializers, such as Kaiming He et al. (2015), Xavier Glorot and Bengio (2010) and Principled Weight Initialization for Hypernetworks Ha et al. (2016) on generation of the LfD system. Another aspect we analyze is the impact of different optimizers, namely Adam Kingma and Ba (2017), RMSProp Tieleman and Hinton (2012) and SGD Rosenblatt (1958), and their impact on convergence and training efficiency.

3.1 Hypernetworks (HN)

In recent years there have been many changes and lots of development of new innovative techniques for improving upon continual learning. One of them has been developed by Ha et al. (2016).

A hypernetwork is a neural network $\mathcal{H}(E;\Phi)$ (left side in Fig. 3.1) which takes $e^m \in E$ as a task-specific embedding vector and has Φ as its weights and biases. e^m (left blue box in Fig. 3.1) is at first drawn from a normal distribution $\mathcal{N}(\mu = 0.0, \sigma = 1.0)$ and acts as a trainable parameter during training. In a forward pass, the hypernetwork $\mathcal{H}$ generates the weights and biases θ^m for a task-specific targetnetwork $\mathcal{F}(X;\theta^m)$ (also called mainnet; right side in Fig. 3.1). The targetnetwork can be seen as a regular NN that takes $x \in X$ as input and generates an output $\hat{y}^m$. The training of the hypernetwork is done by backpropagation, but is done in two steps. This way of memory-efficient backpropagating has been introduced by von Oswald et al. (2019). First a candidate change $\Delta\Phi^m$ which should minimize the loss $\mathcal{L}^m$ for the current task m is computed.

$$\mathcal{L}^m = \mathcal{L}^m(\theta^m, \hat{y}^m) \text{ with } \theta^m = \mathcal{H}(e^m, \Phi) \tag{3.1}$$

Now the candidate change can be used to compute the regularized loss $\tilde{\mathcal{L}}^m$ of the hypernetwork.

$$\tilde{\mathcal{L}}^m = \mathcal{L}^m(\Phi^m, \hat{y}^m) + \frac{\beta}{m-1} \sum_{l=0}^{m-1} || \mathcal{H}(e^l, \Phi^*) - \mathcal{H}(e^l, \Phi + \Delta\Phi^m) ||^2 \tag{3.2}$$

Φ^* denotes the parameters of the hypernetwork before attempting to learn task m.

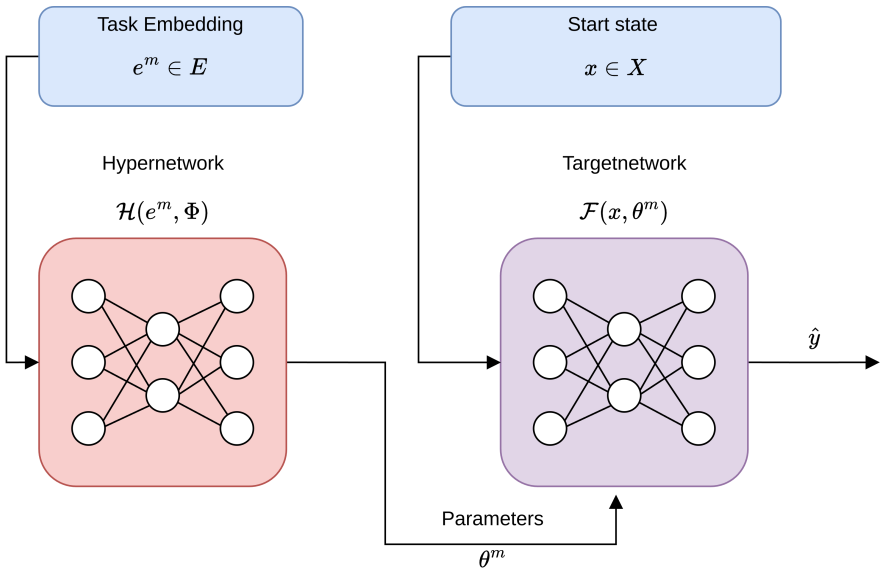


Figure 3.1: Illustration of a hypernetwork. Left: Hypernetwork $\mathcal{H}$ (colored in) takes an task embedding e^m as input and generates in a forward-pass, parameters θ^m of the targetnetwork $\mathcal{F}$ (colored in). $\mathcal{F}$ takes a start state x as input and generates a prediction $\hat{y}$.

3.1.1 Chunked Hypernetworks (CHN)

Hypernetworks offer advantages in continual learning compared to generic neural networks. But depending on the size of the targetnetwork or rather the number of parameters that have to be generated all at the same time makes the HN less efficient as the output of the HN will be very high dimensional. To solve this constraint von Oswald et al. (2019) introduced chunked hypernetworks. These do not generate the entire targetnetwork at once, but rather in multiple smaller chunks. To differentiate each of the chunks, an additional chunk embedding is used. Compared to hypernetworks, these chunked ones can be smaller in size while still being comparable in performance. Auddy et al. (2023a) were able to show that a CHN is capable of learning several LfD tasks with a comparable performance to that of a HN, and is even better suited as the total number of parameters is less, when the number of tasks is relatively small.

3.2 Initializers

In terms of deep neural networks, initialization schemes are vital as the chosen one can decide how fast or even if a neural network converges to an optimal minimum. When hidden units on the same layer are initialized to the same value, they will be updated in the same way, due to the way backpropagation works. The neurons will learn the same features, creating an undesired symmetry, which must be avoided or so-called "broken". Another problem that can occur when the neural network is initialized with zeros, especially when ReLU is used as an activation function, is dying neurons where they get stuck and can't be changed anymore during backpropagation, as the gradient is zero. There are multiple approaches with the most naïve ones being just picking the weight and if not initialized to a specific value also bias values, from a uniform distribution with range $[0.1, 0.1]$ or from a normal distribution with $\mu = 0$ and $\sigma = 0.1$. There are more sophisticated methods that have been proven more reliable for example Kaiming initialization which is elaborated in detail below.

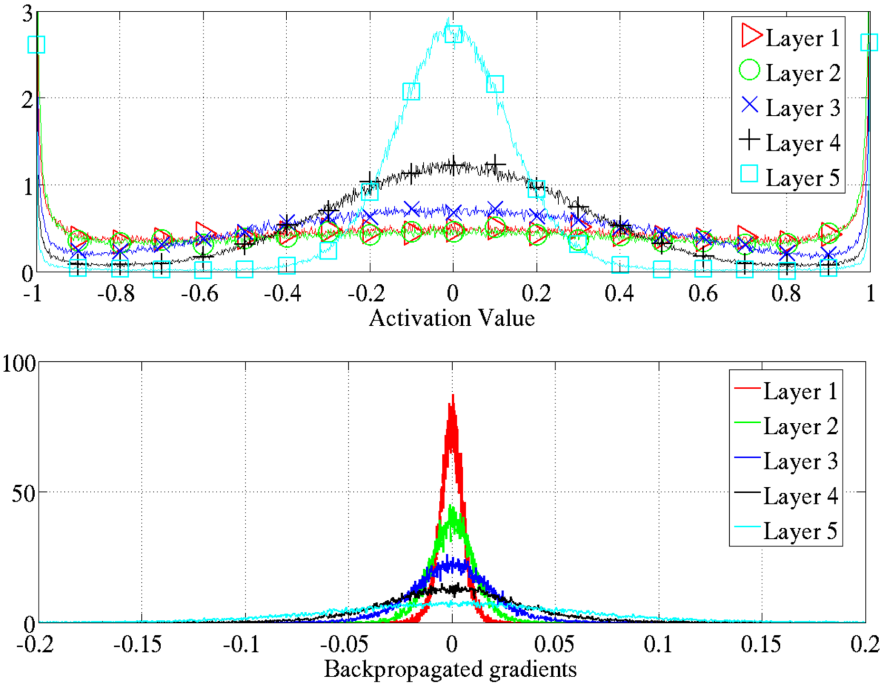


Figure 3.2: Top: activation value of a neural network, utilizing hyperbolic tangent. Bottom: backpropagated gradients with common init. Source: Glorot and Bengio (2010)

3.2.1 Xavier

The Xavier or Glorot initializer has been proposed by Glorot and Bengio (2010) as a method to initialize the weights of a deep neural network, which utilizes the hyperbolic tangent and sigmoid as its activation functions. Despite not being designed for usage with neural networks utilizing ReLU, it is performing better than the naïve random initialization. In addition Chang et al. (2020) compared the performance of their proposed initializer against Xavier initialization and Kaiming initialization.

The top plot in figure 3.2 shows the activation value of a trained neural network utilizing hyperbolic tangent as its activation function. In image 3.2 the backpropagated gradients can be observed in the bottom plot. The network has been initialized with a commonly used heuristic, as stated by Glorot and Bengio (2010):

$$W \sim \mathcal{U}[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}] \tag{3.3}$$

Here $\mathcal{U}[-a, a]$ is the uniform distribution within the interval $(-a, a)$, n is the number of neurons in the previous layer.

Due to the nature of backpropagation the layers closest to the output will have higher absolute gradients. With a reduction in the number of layers, as seen in the bottom plot figure 3.2, the gradient tends to be zero more frequently. This means that only the layers close to the output will be properly optimized with respect to the error.

They proposed a new normalized initialization, shown in equation 3.4 which adds a gain depending on the number of the layer.

$$W \sim \mathcal{U}[-\sqrt{\frac{6}{n_j + n_{j+1}}}, \sqrt{\frac{6}{n_j + n_{j+1}}}] \tag{3.4}$$

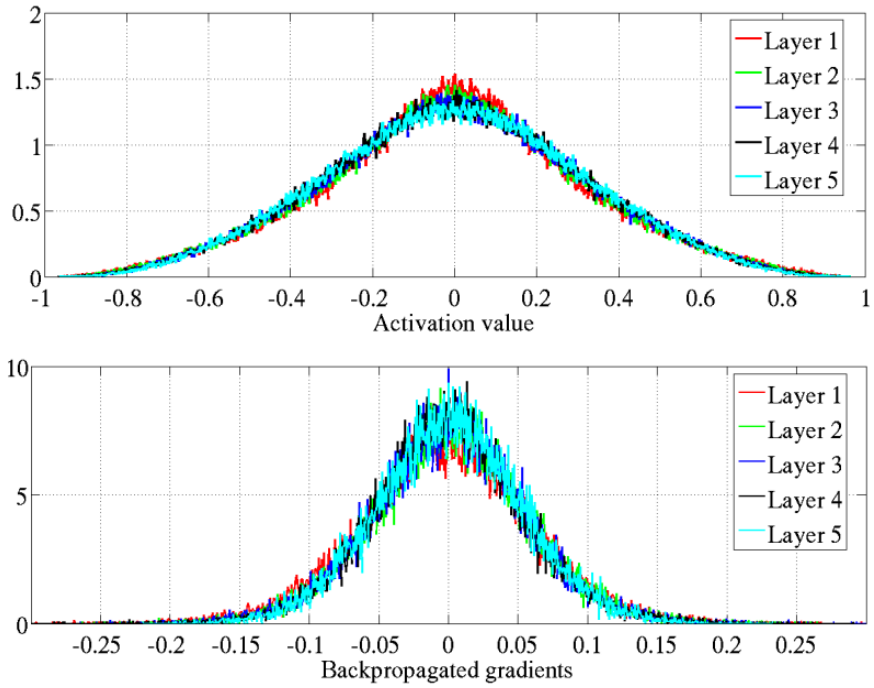


Figure 3.3: Top: activation values of a network initialized with Xavier init. Bottom: Backpropagagated gradients of Xavier initialized neural network. Source: Glorot and Bengio (2010)

Here n_j is the input dimension into the layer j (also called fan_in) and n_{j+1} is the output dimension (also called fan_out).

The impact of their proposed Xavier initialization can clearly be seen in figure 3.3 which shows an even distribution of activation values and gradients when compared between the layers. This impacts the performance of the neural network. In their tests they confirmed that networks which utilized hyperbolic tangent as its activation function, performed better when using equation to initialize weights 3.4 instead of 3.3.

3.2.2 Kaiming

The Kaiming initializer, which is also known as the He initializer, introduced by He et al. (2015) is a widely used weight initialization technique in deep learning. This initializer has been developed to work with the ReLU and PReLU activation functions, which for example are commonly utilized when using a CNN for its ease of training and good results. Xavier init would not work properly as it assumes a symmetrical activation function which neither ReLU nor PReLU are. That is also the reason why there are two distinct "modes" for initialization. One tries to keep the variance during the forward pass and one in the backward one. For initializing a network that utilizes ReLU activation functions the following uniform distribution has been introduced[1]:

$$W_{ij} \sim \mathcal{U}[-\sqrt{\frac{6}{\text{fan\_mode}}}, \sqrt{\frac{6}{\text{fan\_mode}}}] \tag{3.5}$$

Here the fan_mode represents the direction where the variance should be preserved.

[1] `https://pytorch.org/docs/stable/nn.init.html`

3.2.3 Principled Weight Initialization for Hypernetworks

The introduction of the hypernetwork by Ha et al. (2016) also introduced the question of how to initialize such a hypernetwork. Chang et al. (2020) tried to solve this problem with their work of Principled Weight Initialization for Hypernetworks (PWI). They review other methods for the initialization of hypernetworks, where 3 out of 4 used Kaiming initialization in some way. These initialization schemes are not well suited for use with hypernetworks as they can lead to a wrong scaling of the target network. They proposed 4 different ways of initializing, depending on what mode to initialize *Hyperfan_in* or *Hyperfan_out* and whether the hypernetwork generates solely the weights or weights and the biases for the targetnetwork. We used the *Hyperfan_out* initialization with only the weight generation. The variance is in this case defined as:

$$\mathrm{Var}(H^i_{jk}) = \frac{1}{d_j d_k \mathrm{Var}(emb)} \tag{3.6}$$

Where d_j denotes the input dimension of layer j, d_k is the input dimension of the last layer. $\mathrm{Var}(emb)$ is the variance of the task embedding.

The required weights are then drawn from either a uniform distribution:

$$X \sim \mathcal{U}[-\sqrt{3\mathrm{Var}(X)}, \sqrt{3\mathrm{Var}(X)}] \tag{3.7}$$

Or a normal distribution:

$$X \sim \mathcal{N}(0, \mathrm{Var}(X)) \tag{3.8}$$

They also tested their initializer against Xavier and Kaiming in 4 different experiments, which showed promising results.

3.3 Optimizers

Optimizers play a crucial role in training a neural network. They are used to set the new value of each weight and bias after the backpropagation. Optimizers try to avoid landing in a local minima or not propagating when on a plateau. Adaptive optimizers such as RMSProp Tieleman and Hinton (2012), Adagrad Duchi et al. (2011) and Adam Kingma and Ba (2017) also try to handle the problem of deep neural networks where vanishing or exploding gradients can be a problem.

3.3.1 Stochastic Gradient Descent

Stochastic Gradient Descent or SGD also vanilla SGD, is one of the most well-known optimization algorithms tracing back to Rosenblatt (1958). At its core, SGD utilizes the gradient of the loss function which is calculated during the backward pass. In contrast to batch gradient descent, which computes the gradient with respect to θ for the entire dataset, SGD updates gradients w.r.t. θ for each sample making it more suitable for larger datasets. There are different variants for example SGD with Nesterov Momentum utilizes an additional parameter that should help with flat regions. We used the implementation[2] provided by the PyTorch framework Paszke et al. (2019). An advantage of SGD is that it is simple which makes it fast, but this is also a disadvantage as it cannot handle the loss gradient as well as some other more sophisticated optimizers such as Adam. The pseudo-code for SGD is shown in algorithm 1. The parameter γ is the learning rate and θ_0 are the weights. Whereas λ is a weight decay causing the learning rate to get less per each iteration. If we would want to use Nesterov momentum we would use μ as the momentum which helps make larger steps through

[2] `https://pytorch.org/docs/stable/generated/torch.optim.SGD.html`

flat loss "landscape", whilst τ acts as a dampening factor to reduce the step size when gradients are large. The parameters *nesterov* and *maximize* are booleans that enable the momentum and change the direction of the steps taken respectively, where *maximum* would be used, as the name implies, for a maximization problem.

Algorithm 1 Stochastic Gradient Descent
input: γ(lr), θ_0 (params), $f(\theta)$(objective), λ(weight decay), μ(momentum), τ(dampening), *nesterov*, *maximize*

for t = 1 **to ...** **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_t - 1)$
  if $\lambda \neq 0$ **then**
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  end if
  if $\mu \neq 0$ **then**
    if $t > 1$ **then**
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    end if
    if *nesterov* **then**
      $g_t \leftarrow g_t + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
    end if
  end if
  if *maximize* **then**
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$
  end if
end for
return θ_t

3.3.2 RMSProp

RMSProp is a sophisticated, adaptive optimizer presented by Geoffrey Hinton in one of his lectures Tieleman and Hinton (2012). It is based on Rprop by Riedmiller and Braun (1993). To use Rprop we have to do a full-batch optimization, this allows us to get a direction of the gradient which represents the distribution of the data better. Rprop combines the direction or sign of the gradient with adapting each weight on its own. Thus we should be able to adapt better w.r.t. the parameters θ. Now the downside of Rprop is that it only works using a full-batch optimization, instead of mini-batches. This makes it unsuitable for large datasets as they are often too large to fit into the available memory. If tried to use with mini-batches it won't be stable as it violates the idea of SGD which is using a small learning rate γ, we can approximately average the gradient. RMSProp tackles this, by utilizing a moving average of squared weights for each weight $\tilde{v_t}$. Which is squared and then used for the update step $\theta_t \leftarrow \theta_{t-1} - \gamma g_t/(\sqrt{\tilde{v_t}})$. As we are utilizing a moving average over mini-batches, RMSProp is able to adjust the gradient properly in the direction in which we want to converge. The pseudo-code below is from PyTorch [3].

[3] `https://pytorch.org/docs/stable/generated/torch.optim.RMSprop.html`

Algorithm 2 RMSProp
input: α(alpha), γ(lr), θ_0 (params), $f(\theta)$(objective), λ(weight decay), μ(momentum), *centered*
initialize: $v_0 \leftarrow 0$(square average), $\mathbf{b}_0 \leftarrow 0$(buffer), $g_0^{ave} \leftarrow 0$

for t = 1 **to ...** **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_t - 1)$
  if $\lambda \neq 0$ **then**
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  end if
  $v_t \leftarrow \alpha v_{t-1} + (1-\alpha)g_t^2$
  $\tilde{v_t} \leftarrow v_t$
  if *centered* **then**
    $g_t^{ave} \leftarrow g_{t-1}^{ave}\alpha + (1-\alpha)g_t$
    $\tilde{v_t} \leftarrow \tilde{v_t} - (g_t^{ave})^2$
  end if
  if $\mu > 0$ **then**
    $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + g_t/(\sqrt{\tilde{v_t}} + \epsilon)$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\mathbf{b}_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t/(\sqrt{\tilde{v_t}} + \epsilon)$
  end if
end for
return θ_t

3.3.3 Adam

Adam is a fairly recent optimizer developed by Kingma and Ba (2017). The authors of Adam describe it as a combination of the benefits of AdaGrad Duchi et al. (2011) and RMSProp Tieleman and Hinton (2012). Where both utilize a per-weight learning rate and AdaGrad works well on problems with sparse gradients and RMSProp is able to work with noisy gradients (due to its averaging window mechanism). Adam utilizes the mean (1st moment), and its uncentered variance (2nd moment). m_t denotes the exponential moving average of the gradient and v_t the average squared gradient. The hyperparameters β_1 and β_2 represent the exponential decaying rates for m_t and v_t respectively. The authors point out that due to both averages being initialized with zeros, there would be a bias towards zero, especially in the starting phase of execution. To counteract they proposed using bias correction, resulting in $\widehat{m_t}$ and $\widehat{v_t}$. Adam is very popular as it is a very effective optimization algorithm. Ruder (2017) writes in his review about optimizers that even though RMSProp, Adadelta and Adam behave very similar, Adam would probably be the best overall choice, as it is able to edge both out, especially when gradients become sparse. The below pseudo-code is from PyTorch[4]. The most important hyperparameters here are the learning rate (γ), both betas β_1 and β_2. β_1 is an exponential decay factor for the 1st moment with a recommended value of 0.9 and β_2 is the exponential decay factor for the 2nd moment recommended to be 0.999.

[4] `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`

Algorithm 3 Adam
input: γ(lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$(objective), λ(weight decay), *amsgrad*, *maximize*
initialize: $m_0 \leftarrow 0$(first moment), v_0(second moment), $\widehat{v_0}^{max} \leftarrow 0$

for t = 1 **to ...** **do**
  if *maximize* **then**
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  end if
  if $\lambda \neq 0$ **then**
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  end if
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
  $\widehat{m_t} \leftarrow m_t/(1-\beta_1^t)$
  $\widehat{v_t} \leftarrow v_t/(1-\beta_2^t)$
  if *amsgrad* **then**
    $\widehat{v_t}^{max} \leftarrow max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t}/(\sqrt{\widehat{v_t}^{max}} + \epsilon)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t}/(\sqrt{\widehat{v_t}} + \epsilon)$
  end if
end for
return θ_t

3.4 Neural Ordinary Differential Equation Solver

3.4.1 NODE

An Ordinary Differential Equation solver is a system that can describe a dynamic non-linear system. Chen et al. (2019) presented a way to train a neural network to approximate a vector field f_{true}. The available data consists of N observed trajectories $\mathcal{D} = y^{(0)}_{0:T-1}, ..., y^{(N-1)}_{0:T-1}$, with each trajectory $y^{(i)}_{0:T-1}$ consisting of a sequence of T states and each sample $y^{(i)}_t \in \mathbb{R}^d$. Also each observation $y^{(i)}_t$ has some unknown noise ϵ, thus $y^{(i)}_t = x^{(i)}_t + \epsilon$ where $x^{(i)}_t$ is the unknown true state. From Heinonen et al. (2018):

$$x_t = x_0 + \int_0^t f_{\text{true}}(x_\tau)d\tau \tag{3.9}$$

x_0 is the true starting state. The NODE solver is able to learn the parameters θ of a neural network f, to approximate f_{true}. The loss $\mathcal{L}$ has to be computed using y_t as we do not have the true states x_t.

$$\mathcal{L} = \frac{1}{2}\sum_t || \, y_t - \hat{y}_t \, ||_2^2 \text{ where } \hat{y}_t = \hat{y}_0 + \int_0^t f_\theta(\hat{y}_\tau)d\tau \tag{3.10}$$

$\hat{y}_t$ is the predicted state of the NODE solver.

3.4.2 Stable NODE (sNODE)

NODE has as presented above in section 3.4.1, no guarantee of being stable meaning it is not bounded, can oscillate or can even diverge from the goal. Due to this unresolved stability issue, Kolter and Manek

(2019) proposed to learn a dynamics model $\hat{f}_\theta : \mathbb{R}^n \to \mathbb{R}^n$ in combination with a *Lyapunov* function that should guarantee stability. The Lyapunov function $V_\gamma(x) : \mathbb{R}^n \to \mathbb{R}$ has the property of being positive definite, meaning $V_\gamma(x) \geq 0$ for $x \neq 0$ and $V_\gamma(0) = 0$. The projection of $\hat{f}_\theta(x)$ that satisfies the condition:

$$\nabla V_\gamma(x)^T \hat{f}_\theta(x) \leq -\alpha V_\gamma(x) \tag{3.11}$$

ensures a stable dynamic system. Such a system has the property that any perturbation eventually diminishes over time, meaning the system will converge to a given state.

Integrated into one function that generates stable trajectories:

$$f_{\theta,\gamma} = \text{Proj}(\hat{f}_\theta(x), f : \nabla V_\gamma(x)^T \hat{f}_\theta(x) \leq -\alpha V_\gamma(x)) \tag{3.12}$$

$$= \hat{f}_\theta(x) - \nabla V_\gamma(x) \frac{\text{ReLU}(\nabla V_\gamma(x)^T \hat{f}_\theta(x)) + \alpha V_\gamma(x)}{|| \nabla V_\gamma(x) ||_2^2} \tag{3.13}$$

The Lyapunov function V_γ is learned using an input-convex neural network Amos et al. (2017). During training, both V_γ and $\hat{f}_\theta$ are learned together. Auddy et al. (2023b) has proposed to input, in addition to the start state x_0, time steps $t_0, ..., t_{n-1}$ to achieve more accurate predictions.

Chapter 4

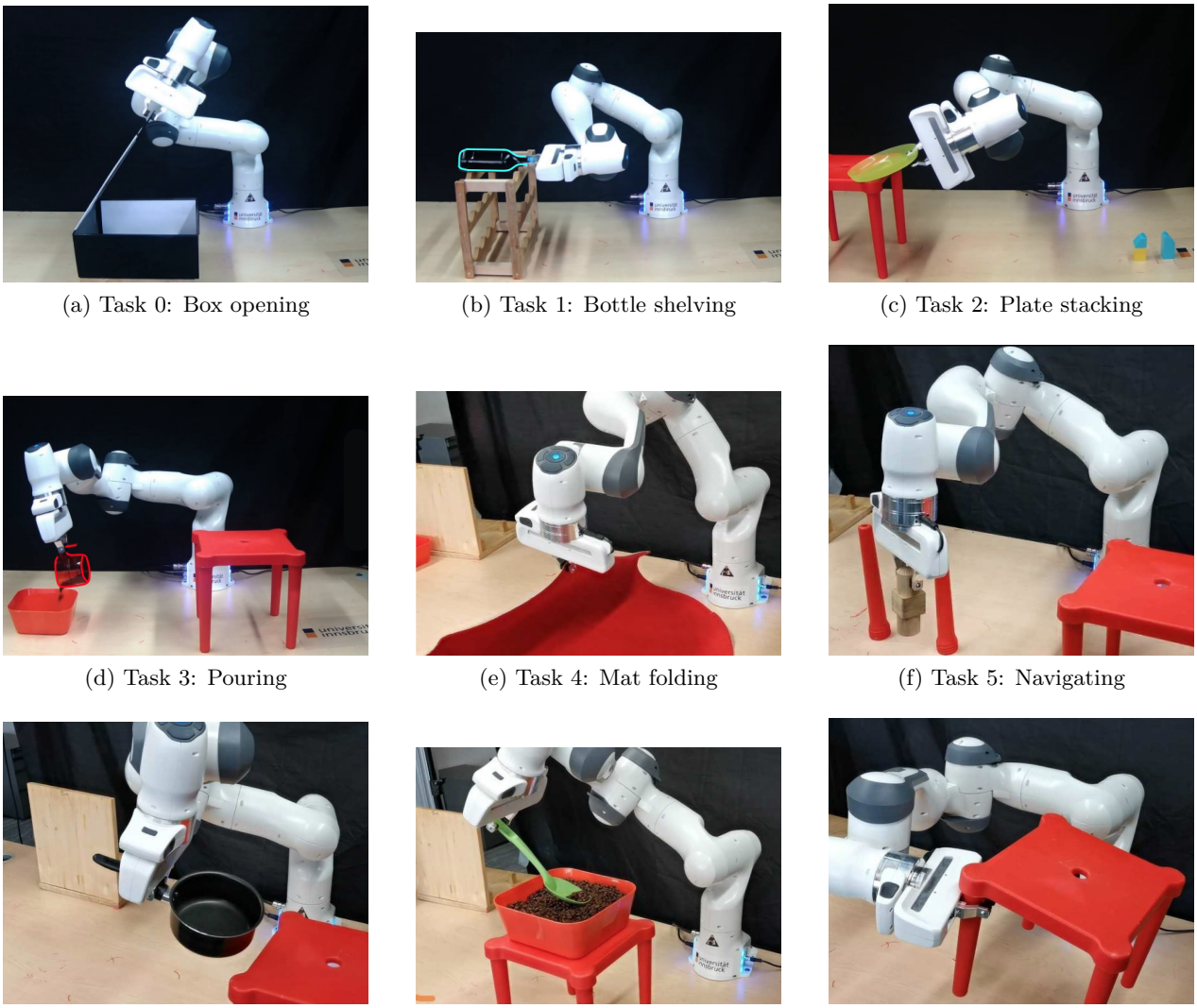
Methods

The experiments conducted should find how different deep learning aspects would affect the continual learning and general performance of the hypernetwork. These are important as robots utilized in the real world have to follow any trained path as closely as possible. If the robot diverges drastically from the trained paths, nearby humans could get harmed. We chose to conduct the experiments with the *RoboTasks9* dataset as it was recorded kinesthetically on a real robot, and the trajectories represent some real-world applications. For metrics, we utilize two metrics, one for the positional error, the DTW error and one for orientational error quaternion error. These allow us to track the error over time. We also report the CL metrics as these allow comparing how well the different aspects compare against each other.

4.1 Dataset

We used the *RoboTasks9* dataset (Auddy et al. (2023b)) to train the hypernetwork. This dataset consists of 9 real-world tasks. These are (i) *box opening* (Fig. 4.1a) the lid of a box is opened, (ii) *bottle shelving* (Fig. 4.1b) a bottle is placed on a shelf, (iii) *plate stacking* (Fig. 4.1c) a plate is placed on a table, (iv) *pouring* (Fig. 4.1d) a cup of coffee beans is poured into a container, (v) *mat folding* (Fig. 4.1e) a mat is folded in half, (vi) *navigating* (Fig. 4.1f) an object is navigated through obstacles, (vii) *pan on stove* (Fig. 4.1g) pan is moved from a hanging position to a table, (viii) *scooping* (Fig. 4.1h) coffee beans are scooped with a spatula, (ix) *table flipping* (Fig. 4.1i) table laying on its side is flipped to stand on its feet.

The robot demonstrations shown in figure 4.1 each consist of 9 kinesthetically recorded demonstrations each with different start position and orientation. A trajectory consists of 1000 steps and each consisting of a recorded position $p \in \mathbb{R}^3$ and a unit quaternion for orientation $q \in \mathbb{S}^3$. This dataset allows us to evaluate our system's continual learning capability in real-world scenarios. Compared to the popular LASA dataset Khansari Zadeh and Billard (2011), this dataset contains more variability in the demonstrations making it more challenging.



(a) Task 0: Box opening (b) Task 1: Bottle shelving (c) Task 2: Plate stacking

(d) Task 3: Pouring (e) Task 4: Mat folding (f) Task 5: Navigating

(g) Task 6: Pan on stove (h) Task 7: Scooping (i) Task 8: Table flipping

Figure 4.1: Illustration of tasks in the RoboTasks9 dataset. Source: Auddy et al. (2023b)

4.2 Metrics

The metrics are mostly in line with Auddy et al. (2023a). We mainly utilize the widely known *Dynamic Time Warping error* for positional errors Urain et al. (2020) Jekel et al. (2019). For orientational error, we use the common *quaternion error* like Ude et al. (2014) Saveriano et al. (2019). We also report the continual learning metrics Accuracy (ACC), Remembering (REM), Model Size Efficiency (MS) and Sample Storage Size Efficiency (SSS) as presented by Díaz-Rodríguez et al. (2018). In addition, we provide the metrics Time Efficiency (TE) and Final Model Size (FS) presented by Auddy et al. (2023a).

4.3 Experiments

To evaluate what impacts the system performance the most, we conduct empirical experiments where we change optimizers, initializers and hypernetwork/targetnetwork depth. The experiments are not executed in a grid-search-like pattern where each parameter combination is tested, as this would result

in too many combinations which of course are time-consuming to execute. Instead, we separate the optimizers, initializers and architecture experiments and evaluate them one after another. This allows us to reduce the number of experiments drastically from 576 to 40. With 5 independent executions per experiment configuration, this would equate to 200 executions in total.

4.3.1 First Experiment: Optimizer

In the first experiment, we evaluate the empirical performance of Adam against RMSProp and SGD. These 3 optimizers are applied on 4 distinct systems utilizing HN and CHN for continual learning and NODE Auddy et al. (2023a) or sNODE Auddy et al. (2023b). As initializer we decided to use Kaiming init as most prior papers utilized it too, such as Auddy et al. (2023b) (not explicitly mentioned in their paper[1]) or von Oswald et al. (2019). The architecture of both hypernetwork (HN/CHN) and targetnetwork (NODE/sNODE) is fixed to 3 layers for each (details in appendix A). It is important to know how the optimizers interact while training as it has an impact on several aspects such as convergence speed, generalization, stability and avoiding local minima. Each of the selected optimizers has some advantage. Adam is generally in most cases the go-to optimizer Ruder (2017), while RMSProp may have a more stable behavior and SGD usually offers the best generalization despite being prone to land in local minima and having low convergence speed.

4.3.2 Second Experiment: Initializer

In the second experiment, we use the best-performing optimizer of the first experiment and use the initializer as our variable. We use Kaiming init He et al. (2015), Xavier init Glorot and Bengio (2010) and Principled Weight Initialization for Hypernetworks Chang et al. (2020). The architectures of both networks are again fixed to 3 layers each. Again this knowledge gained has a substantial impact as the chosen initalizer can also impact the speed of convergence and degree of generalization. The choice of initializer can also have an impact on vanishing or exploding gradients, which is even more important in deep neural networks.

4.3.3 Third Experiment: Architecture

For the last experiment, we again take the best overall performing combination of the previous experiment. The information inferred from this experiment is non-trivial. Deeper networks are able to represent more complex functions while wider networks can memorize better. Now to test the architectures we evaluate each combination. We used 2,3,4 and 8 layers on both the hypernetwork that generates the targetnetwork and the targetnetwork that is generating the prediction of the trajectory, to distinguish the performance when using a wide versus a deep network. It is important to mention that despite utilizing varying layer numbers, we try to keep the overall parameter size of the hypernetworks and targetnetworks as similar as possible, this is done by reducing the neurons per layer in deeper networks and increasing the neurons per layer in shallower networks, further details can be found in the appendix A. Choosing the right depth of a network greatly influences the achievable performance. Deeper networks generally can extract more hierarchical information from the data and are able to infer more complex relations. With wider networks being able to store more global relational information. This is strongly dependent on the data and use case.

[1] `https://github.com/sayantanauddy/clfd/blob/main/imitation_cl/model/hypernetwork.py`

Chapter 5

Results

In the following chapter, we present the results of our experiments. We analyze the results for the first experiment and use the best overall performing optimizer for the following experiments. To visualize the data we use line plots for DTW error and Quaternion error, this should visualize the performance at the start and over the course of learning new tasks. This plot is accompanied by a box plot showing the overall positional performance of each optimizer after training for all tasks. And as CL has many facets in regard to metrics such as accuracy, remembering, time efficiency and more we use polar plots to visually compare the performance of each.

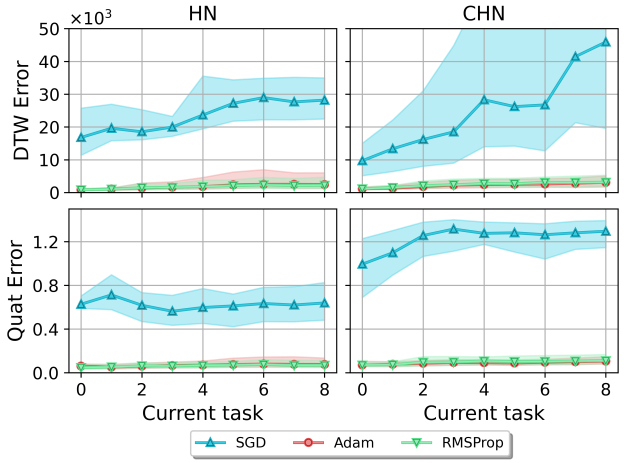
A similar approach is used in the second experiment where first an analysis of the visualized data is done and then the best overall performing initializer is picked for the last experiment. The visualization is similar to the first experiment, where line plots show the positional and orientational performance over the course of all learned tasks. A box plot shows the overall DTW error. A polar plot visualizes the metrics of CL.

In the last experiment, the impact of architecture is evaluated. In this experiment we have more combinations, 16 per CL-LfD configuration (HN/CHN - NODE/sNODE), summing up to 64 in total. To properly display the nuances we utilize heatmaps, where the median DTW error is evaluated for each combination of hypernetwork layers and targetnetwork layers.

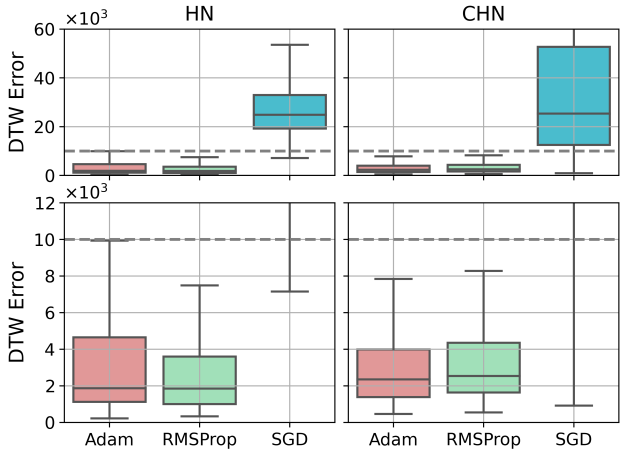
5.1 Experiment 1: Optimizer

For the first experiment, we vary the optimizer used while keeping both initializer and architecture fixed. The used initializer is Kaiming as this initializer was applied in prior papers and seemed to work reasonably well. Similarly, we chose the architecture. The HN and CHN have 3 layers each, as well as the LfD (NODE/sNODE) system. The optimizers all use the same learning rate, all other hyperparameters are set to the recommended default values[1].

[1] `https://pytorch.org/docs/stable/optim.html`



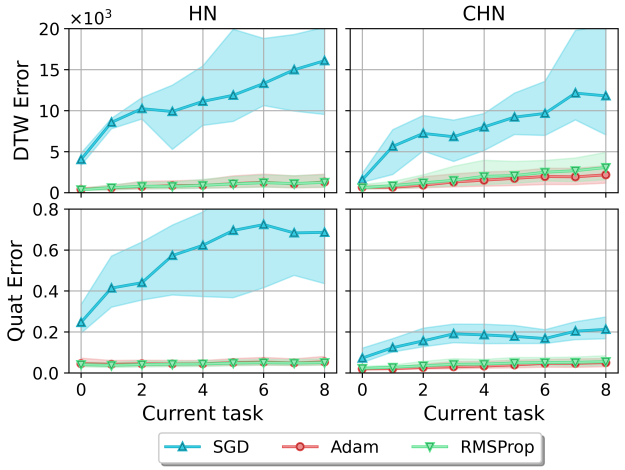
(a) Line plots for NODE with DTW error (top) and quaternion error (bottom)



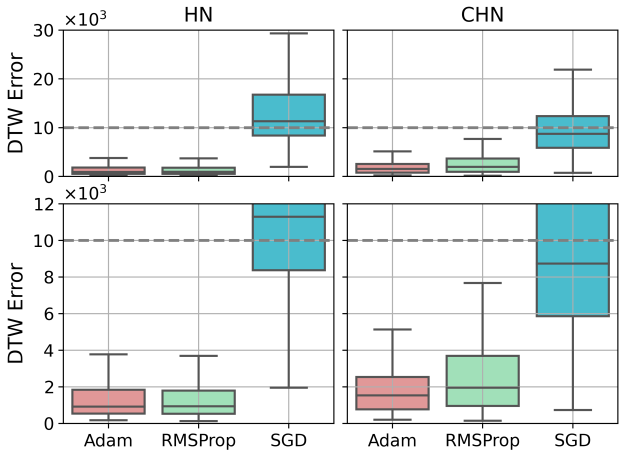
(b) Box plots for NODE showing DTW error (bottom: zoomed-in)

Figure 5.1: Left: Line plots shows optimizer error in position and orientation (for both lower is better). The x-axis shows the current task. The line is the median error and the shaded area above and below represent the lower and upper quartile. SGD performs worst in both HN and CHN. Adam and RMSProp performing very similar over the entire range of tasks. Right: Box plots show the error of all predictions (lower is better). SGD has a similar median for HN and CHN, but varies much more in the CHN case. Adam and RMSProp performing close, with Adam having a better median DTW error with the CHN, but a greater IQR with the HN.

Figure 5.1 shows in plot Fig. 5.1a how the different optimizers affect the learning of each task. The line plots on top, show the DTW errors (y-axis) of the predicted, against the real trajectories. The bottom line plots visualize the orientation error (Quaternion Error). The lines show the median and the shaded areas are bounded by the upper and lower quartiles of 5 independent executions. The box plots in Fig. 5.1b display the DTW error after training all of the 9 trajectories. For all the plots in Fig. 5.1, a higher DTW and Quaternion error is worse. The inter quartile range (IQR), depicted as the shaded areas in the line plots and the colored boxes in the box plots, is also better when its smaller meaning that there is less variance in the trajectory of the 5 independent executions. From plot Fig. 5.1a we can infer that SGD struggles to learn any task with the HN, while in combination with the CHN it is able to learn the first few tasks, but then gets continually worse. While Adam and RMSProp are both performing well over the entire dataset. This can also be observed in the box plots where SGD is much worse and if we zoom in (bottom plots) we see that Adam and RMSProp are performing very similarly. While having the same median with the HN system Adam has a larger IQR. And in combination with the CHN system, both optimizers perform well. With Adam, it has a slightly superior performance.



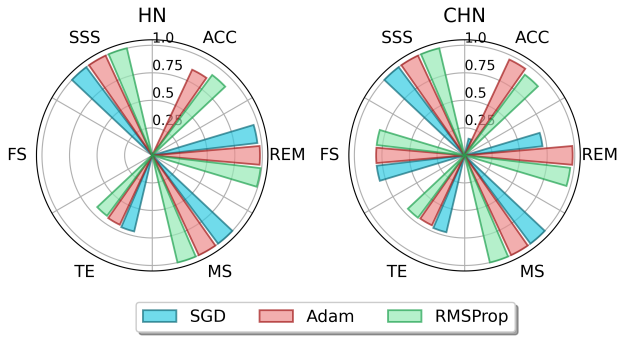
(a) Line plots for sNODE with DTW error (top) and quaternion error (bottom)



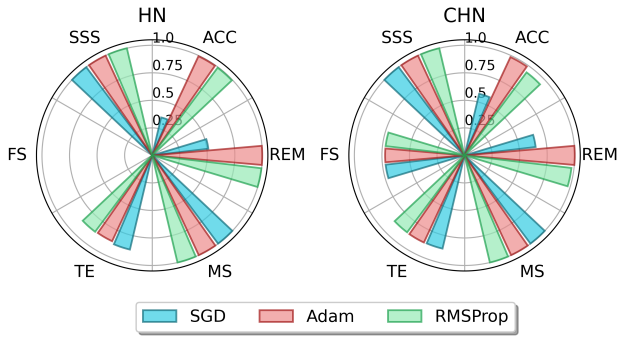
(b) Box plots for sNODE showing DTW error (bottom: zoomed-in)

Figure 5.2: sNODE Performance on different optimizers. 5.2a depicts the positional (top) and orientational (bottom) error over each task (for both lower is better). SGD is able to learn the first task but then gradually becomes worse. Adam and RMSProp perform similar, with Adam performing better in more learned tasks with CHN. 5.2b shows the DTW error over all tasks (lower is better). Adam and RMSProp performing about the same with HN and Adam a bit better with CHN.

The results in Fig. 5.2 show analog to figure 5.1 the performance of stable NODE in combination with HN and CHN while varying the different optimizers. In Fig. 5.2a we can see, again SGD being worse than RMSProp and Adam. In combination with sNODE, SGD seems to learn, especially the first task, a bit better in both cases. With CHN and SGD being able to perfom comparably to both other optimizers in the first task, but then gradually getting worse with each new learned task. In the box plot Fig. 5.2b we can observe SGD performs with sNODE much better being more stable and less than half the error. Adam and RMSProp, perform very closely, with Adam being again a bit better with CHN.



(a) CL metrics for NODE (0: worst, 1: best)



(b) CL metrics for sNODE (0: worst, 1: best)

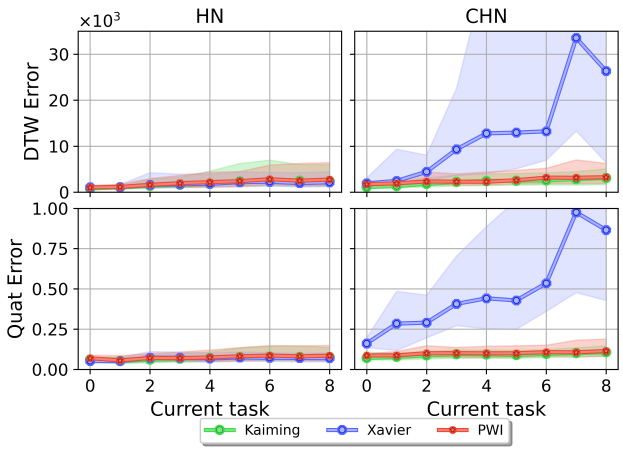
Figure 5.3: Continual learning metrics (0: worst, 1: best) for different optimizers with NODE and sNODE. 5.3a shows Adam and RMSProp performing very similar in ACC and REM. The HN plot (left) shows 0 for FS as it has more parameters than the CHN. 5.3b shows SGD performing a bit better with sNODE and especially with the CHN. Adam and RMSProp again perform very good.

The CL metrics for both NODE and sNODE are depicted in Fig. 5.3. The most interesting metrics

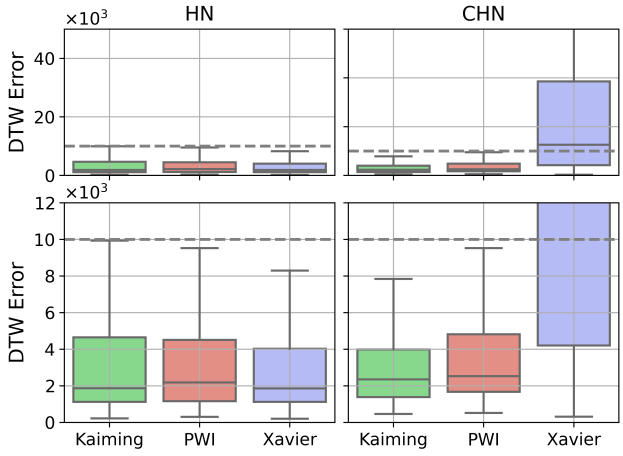
here are ACC and REM, where we can again see both RMSProp and Adam performing very well in Fig. 5.3a and Fig. 5.3b. SGD scores badly when used with NODE. But is able to achieve about 50% of accuracy in combination with sNODE and CHN.

5.2 Experiment 2: Initializer

From the previous experiments, we can see Adam as being the overall best-performing optimizer. In this second experiment, we use Adam and change the initializer to Kaiming, Xavier and PWI. The architecture of the overall system is again fixed to the same 3 layers for the CL and LfD systems.



(a) Line plots for NODE with DTW error (top) and quaternion error (bottom)



(b) Box plots for NODE showing DTW error (bottom: zoomed-in)

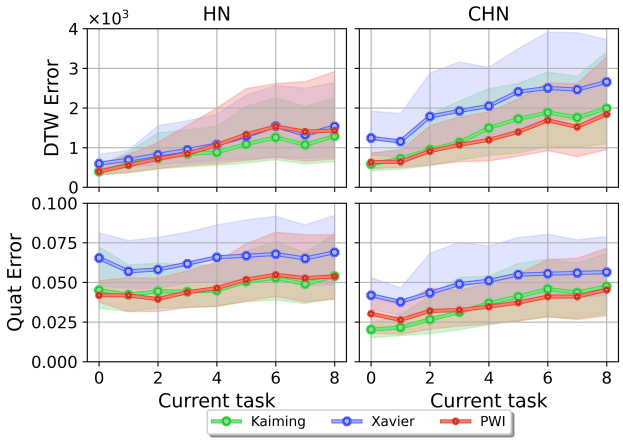
Figure 5.4: Performance of different initializers with NODE. 5.4a shows DTW and Quaternion error (lower is better) against the current task. The line is the median of 5 independent executions and the shaded areas are the upper and lower quartiles. With HN all three initializers seem to perform well in position and orientation. Using CHN Xavier learns the first task, but then becomes gradually worse. 5.4b shows the positional error over all tasks (lower is better). The line in the colored boxes is the median error and the edges of the box represent the upper and lower quartile (less distance between is better). With the HN Xavier performed the best albeit all being close to another. With the CHN Xavier performs poorly while both other initializers perform well.

Figure 5.4 shows the performance utilizing different initializers. On the left Fig. 5.4a displays the positional and the orientational error against the learned task. And on the right Fig. 5.4 box plots of the DTW error after training. With HN all three initializers are performing well which can be seen in both line and box plots. If Xavier and NODE with CHN are utilized the resulting performance is poor. From the line plots, we can see that this combination is able to learn the trajectory, but is already much worse than both other optimizers. Compared the results of Kaiming and PWI are very close. With a HN they both practically perform the same. With the CHN, Kaiming is a little bit better, also having a smaller IQR meaning it has more stable results.

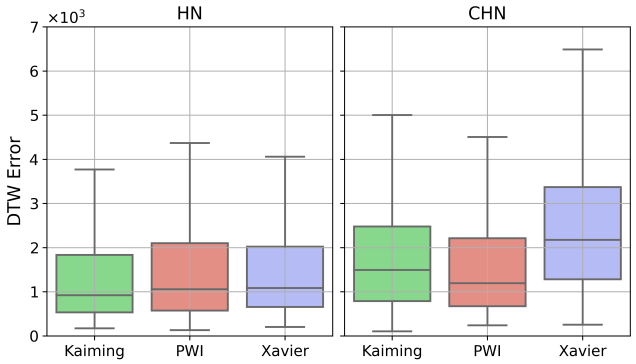
The results of this second experiment, where we utilized different initializers, are diverging. We should disclose that when we utilized Xavier init (implementation from PyTorch[2]) we first used the hyperparameters presented by PyTorch (being $\sqrt{2}$ for ReLU), but could not use them as the errors

[2] `https://pytorch.org/docs/stable/nn.init.html`

reported for the metrics (DTW and quaternion error) were extremely large. Being off by a few orders of magnitude, compared to what was expected. As soon as the gain parameter was ≤ 1.0 the reported errors landed in a reasonable range. We did all further experiments with a gain for Xavier of 1.0. This could be the culprit of Xavier not performing that well with CHN. Xavier was developed to work with hyperbolic tangent and sigmoid not with ReLU, which was used with the hypernetworks.



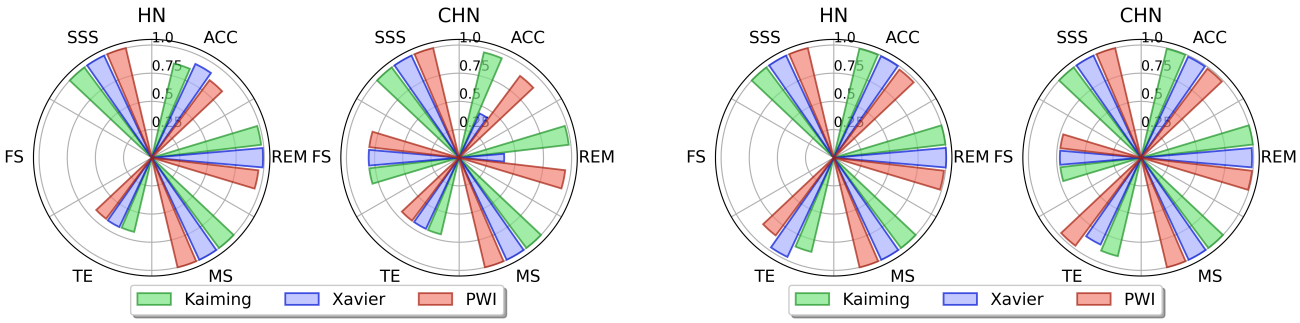
(a) Line plots for sNODE with DTW error (top) and quaternion error (bottom)



(b) Box plot for sNODE showing DTW error

Figure 5.5: Performance of different initializers with sNODE. 5.5a shows DTW and Quaternion error (for both lower is better) against the current task. The line is the median of 5 independent executions and the shaded areas are the upper and lower quartiles. All three initializers perform well with the HN, Xavier has an offset in orientational error for all tasks. The same can be observed in CHN positional and orientational errors. 5.5b shows the positional error over all tasks (lower is better). The line in the colored boxes is the median error and the edges of the box represent the upper and lower quartile (less distance between is better). All seem to perform well with CHN Xavier has a slightly larger IQR, than PWI and Kaiming.

In Fig. 5.5 we can observe the performance of sNODE with the different initializers. In both plots Fig. 5.5a and Fig. 5.5b all initializers are well suited producing low errors that remain relatively stable over the entire training of the dataset. Xavier performs a bit worse overall but not drastically. Also Xavier in combination with the CHN it shows the largest IQR, meaning it is less stable over the course of the 5 executions. PWI performs best with the CHN, while Kaiming is best with HN.



(a) CL metrics for NODE (0: worst, 1: best)

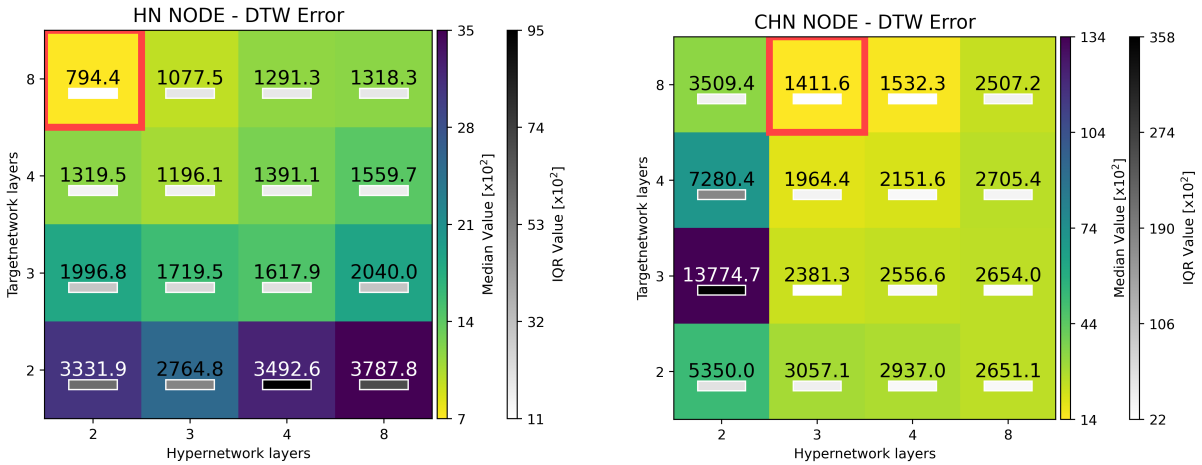
(b) CL metrics for sNODE (0: worst, 1: best)

Figure 5.6: Continual learning metrics (0: worst, 1: best) for different initializers with NODE and sNODE. 5.6a shows Xavier being the most accurate (ACC) with HN, but the least with CHN. Similar for REM. PWI and Kaiming perform very close. The HN plot (left) shows 0 for FS as it has more parameters than the CHN. 5.6b shows all three performing very good in ACC and REM. Xavier has best TE with HN. Looking at CHN PWI has the best TE.

The polar plot shows CL for NODE in Fig. 5.6a and sNODE Fig. 5.6b show the CL metrics of this experiment. NODE with HN Xavier performing best having good ACC and very good REM, whilst not showing promising results with the CHN. The sNODE plots don't show as much change besides TE, where for HN Xavier is performing best and PWI in the case of CHN, implying a more consistent training time for consecutive trajectories.

5.3 Experiment 3: Architecture

In the third experiment, we want to find out what the impact of the architecture is. The initializer we use in this experiment is the top-performing one of the previous experiment. Kaiming and PWI were both performing very well, but Kaiming edged PWI out by a small margin. We vary the hypernetwork as well as NODE/sNODE (targetnetwork) architecture from shallow and wide to deep and narrow. The total number of parameters is about the same with only small deviances. We used 303 neurons per layer with the 2-layer HN and 309 for the CHN, the deepest hypernetworks with 8 layers have 292 and 265 for the HN and CHN respectively. The same is done for the targetnetworks NODE and sNODE (details in the appendix A). All following heatmaps show the hypernetwork layers on the x-axis and targetnetwork layers on the y-axis. With the box colored to the corresponding median DTW error. The smallest error is highlighted by a surrounding red box. Inside each box is a smaller rectangle displaying the corresponding IQR of that combination.



(a) Heatmap for HN NODE with median DTW error in cells, rectangle inside showing IQR.

(b) Heatmap for CHN NODE with median DTW error in cells, rectangle inside showing IQR.

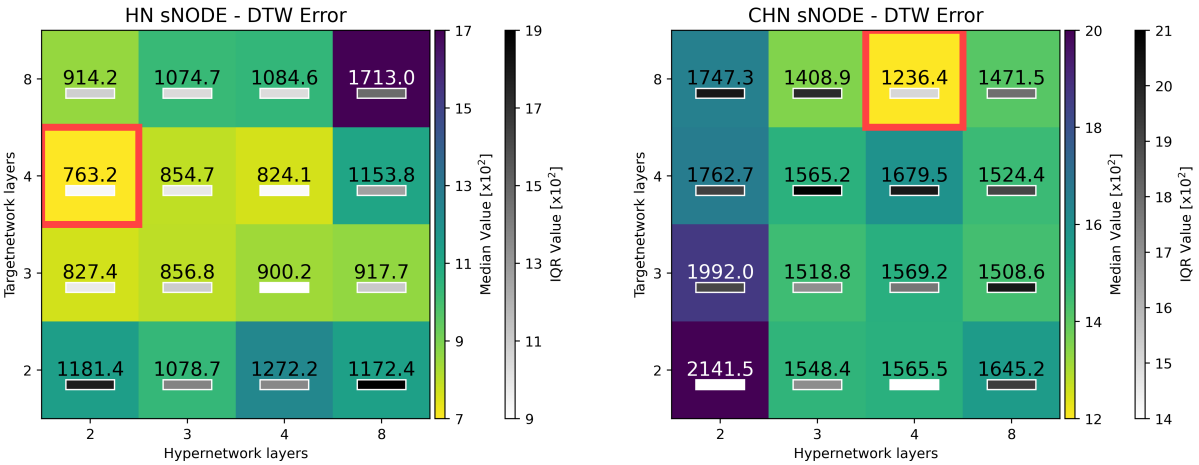
Figure 5.7: Median DTW errors (lower is better) with different architectures of all predictions. The smaller box inside shows the IQR (lower is better). In Fig. 5.7a the combination of a 2-layer hypernetwork and 8-layer targetnetwork (NODE) performed best. The widest targetnetwork performed worst in all combinations. The IQR follows this trend. With CHN in Fig. 5.7b the 3-layered CHN with 8-layered NODE performed best. The 2-layer CHN performed the worst overall with NODE.

The figure 5.7 depicts the impact of different depths of neural networks on HN and CHN systems with NODE. In Fig. 5.7a we can see the HN performance is increasing with a deeper targetnetwork (NODE). There is also a distinct pattern towards a wider hypernetwork (less error). The IQR is following the same trend, being more stable when having a wider HN and a deeper targetnetwork. The CHN in Fig. 5.7b shows a similar preference, if we ignore the column with the 2-layer hypernetwork, which compared was performing only very poorly.

For sNODE Fig. 5.8 it is a bit more difficult to see a general trend as the outputs are stabilized, which can also be seen on the color bars on the side spanning over a much smaller range. For the HN sNODE system Fig. 5.8a we can observe on each row (except the shallowest targetnetwork) a trend towards the less deep hypernetwork. On the other axis, we can see the 4 and 3-layer targetnetwork combinations performing similarly well. Looking at the CHN Fig. 5.8b there is a subtle similarity to CHN NODE Fig. 5.7b where the 2-layer hypernetworks are performing worst overall. And increasing the depth of the targetnetwork improves performance generally.

From the figures 5.7 and Fig. 5.8 we can tell for NODE that both HN and CHN tend to work better with wider hypernetworks and deeper targetnetworks. This is partly to be expected as the hypernetworks has to generate or rather "memorize" the targetnetworks. And that the targetnetworks which is deeper is able to better approximate the function f_{true} implying it is rather complex. In the case of the NODE CHN combination, the shallowest hypernetwork performed the worst, which could be due to generating the targetnetwork in chunks, implying a more complex function than for the HN.

The sNODE HN shows similarities to the NODE system. Despite not showing such a strong gradient in a direction. With shallower hypernetwork performing favorably. The sNODE CHN plot in Fig. 5.8b also shows similarities as the shallowest hypernetwork produces the worst results compared to deeper ones. The IQR range, seen on the color bar is much more condensed as with the NODE plots making the IQR markings in the plot more susceptible to change.



(a) Heatmap for HN sNODE with median DTW error in cells, rectangle inside showing IQR.

(b) Heatmap for CHN sNODE with median DTW error in cells, rectangle inside showing IQR.

Figure 5.8: Median DTW errors (lower is better) with different architectures of predictions for all tasks. The smaller box inside shows the IQR (lower is better). In Fig. 5.8a the combination of a 2-layer hypernetwork and 4-layer targetnetwork (sNODE) performed best. The widest and the deepest targetnetwork performed worse than the 3- or 4-layered targetnetworks. The IQR follows this trend, but is much closer than in the NODE experiment (Fig. 5.7). With CHN in Fig. 5.8b the 4-layered CHN with 8-layered sNODE performed best. The 2-layer CHN records the worst overall with sNODE.

Chapter 6

Conclusion

In this thesis, we explored the impact of several deep learning aspects on a state-of-the-art continual learning from demonstration system. This system should allow learning multiple tasks in a simple way. We utilized different optimizers, initializers, and neural network architectures. The experiments conducted were tested with a recent 9-task dataset that contains real-world robot data. Our work shows how this system works and how it can be improved.

Our findings show that the sophisticated initializer Adam with the initializer Kaiming was able to perform the best, overall with some nuances where RMSProp as optimizer or Xavier as initializer were performing better. Structure-wise we found that a shallower hypernetwork, with a deeper NODE or sNODE was able to outperform the default architecture.

Additionally, to our current contributions, this work also tries to identify work for future research. As only a subset of overall combinations was analyzed, it would make sense to explore the impact of non-explored combinations as there could be better-performing ones. Also, the complex relation of hyperparameters in the optimizers and initializers should be explored more densely. Especially the gain parameter of Xavier initialization should be analyzed more deeply, as it seemed to perform best in certain combinations, whilst being worst in other cases.

In conclusion, we found that the impact of the explored deep learning aspects is crucial for the performance and stability of both continual learning and learning from demonstration systems in robotics. This thesis contributes to the knowledge of the field of continual learning in robotics.

Bibliography

Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. Memory aware synapses: Learning what (not) to forget, 2018.

Brandon Amos, Lei Xu, and J. Zico Kolter. Input convex neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 146–155. PMLR, 06–11 Aug 2017. URL `https://proceedings.mlr.press/v70/amos17b.html`.

Sayantan Auddy, Jakob Hollenstein, Matteo Saveriano, Antonio Rodríguez-Sánchez, and Justus Piater. Continual learning from demonstration of robotics skills. *Robotics and Autonomous Systems*, 165: 104427, 2023a.

Sayantan Auddy, Jakob Hollenstein, Matteo Saveriano, Antonio Rodríguez-Sánchez, and Justus Piater. Scalable and efficient continual learning from demonstration via hypernetwork-generated stable dynamics model, 2023b.

Aude G. Billard, Sylvain Calinon, and Rüdiger Dillmann. *Learning from Humans*, pages 1995–2014. Springer International Publishing, Cham, 2016. ISBN 978-3-319-32552-1. doi: 10.1007/978-3-319-32552-1_74. URL `https://doi.org/10.1007/978-3-319-32552-1_74`.

Oscar Chang, Lampros Flokas, and Hod Lipson. Principled weight initialization for hypernetworks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL `https://openreview.net/forum?id=H1lma24tPB`.

Vinod Kumar Chauhan, Jiandong Zhou, Ping Lu, Soheila Molaei, and David A. Clifton. A brief review of hypernetworks in deep learning, 2023.

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011. URL `http://jmlr.org/papers/v12/duchi11a.html`.

Natalia Díaz-Rodríguez, Vincenzo Lomonaco, David Filliat, and Davide Maltoni. Don't forget, there is more than forgetting: new metrics for continual learning, 2018.

Chongkai Gao, Haichuan Gao, Shangqi Guo, Tianren Zhang, and Feng Chen. Cril: Continual robot imitation learning via generative and prediction model. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6747–5754, Sep. 2021. doi: 10.1109/IROS51168.2021.9636069.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL https://proceedings.mlr.press/v9/glorot10a.html.

David Ha, Andrew Dai, and Quoc V. Le. Hypernetworks, 2016. URL https://arxiv.org/abs/1609.09106.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Markus Heinonen, Cagatay Yildiz, Henrik Mannerström, Jukka Intosalmi, and Harri Lähdesmäki. Learning unknown ODE models with Gaussian processes. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1959–1968. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/heinonen18a.html.

Micha Hersch, Florent Guenter, Sylvain Calinon, and Aude Billard. Dynamical system modulation for robot learning via kinesthetic demonstrations. *IEEE Transactions on Robotics*, 24:1463–1467, 01 2008.

Yizhou Huang, Kevin Xie, Homanga Bharadhwaj, and Florian Shkurti. Continual model-based reinforcement learning with hypernetworks. *CoRR*, abs/2009.11997, 2020. URL https://arxiv.org/abs/2009.11997.

A.J. Ijspeert, J. Nakanishi, and S. Schaal. Movement imitation with nonlinear dynamical systems in humanoid robots. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, volume 2, pages 1398–1403 vol.2, 2002. doi: 10.1109/ROBOT.2002.1014739.

Charles Jekel, Gerhard Venter, Martin Venter, Nielen Stander, and Raphael Haftka. Similarity measures for identifying material parameters from hysteresis loops using inverse analysis. *International Journal of Material Forming*, 12, 05 2019. doi: 10.1007/s12289-018-1421-8.

Seyed Mohammad Khansari Zadeh and Aude Billard. Learning stable non-linear dynamical systems with gaussian mixture models. *IEEE Trans. Robot*, 27, 01 2011.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, March 2017. ISSN 1091-6490. doi: 10.1073/pnas.1611835114. URL http://dx.doi.org/10.1073/pnas.1611835114.

J. Zico Kolter and Gaurav Manek. Learning stable deep dynamics models. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/0a4bbceda17a6253386bc9eb45240e25-Paper.pdf.

German Ignacio Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *CoRR*, abs/1802.07569, 2018. URL http://arxiv.org/abs/1802.07569.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Harish Ravichandar, Athanasios S. Polydoros, Sonia Chernova, and Aude Billard. Recent advances in robot learning from demonstration. *Annual Review of Control, Robotics, and Autonomous Systems*, 3(1):297–330, 2020. doi: 10.1146/annurev-control-100819-063206. URL https://doi.org/10.1146/annurev-control-100819-063206.

M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: the rprop algorithm. In *IEEE International Conference on Neural Networks*, pages 586–591 vol.1, 1993. doi: 10.1109/ICNN.1993.298623.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks, 2022.

Matteo Saveriano, Felix Franzel, and Dongheui Lee. Merging position and orientation motion primitives. In *2019 International Conference on Robotics and Automation (ICRA)*, page 7041–7047. IEEE Press, 2019. doi: 10.1109/ICRA.2019.8793786. URL https://doi.org/10.1109/ICRA.2019.8793786.

Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. *CoRR*, abs/1705.08690, 2017. URL http://arxiv.org/abs/1705.08690.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, 2012. URL http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Aleš Ude, Bojan Nemec, Tadej Petrić, and Jun Morimoto. Orientation in cartesian space dynamic movement primitives. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2997–3004, May 2014. doi: 10.1109/ICRA.2014.6907291.

Julen Urain, Michele Ginesi, Davide Tateo, and Jan Peters. Imitationflow: Learning deep stable stochastic dynamic systems by normalizing flows. *CoRR*, abs/2010.13129, 2020. URL https://arxiv.org/abs/2010.13129.

Johannes von Oswald, Christian Henning, João Sacramento, and Benjamin F. Grewe. Continual learning with hypernetworks. *CoRR*, abs/1906.00695, 2019. URL http://arxiv.org/abs/1906.00695.

Appendix A

Experiment Hyperparameters

Hyperparameter	HN	CHN
Train iterations	40000	40000
Learning rate	5.0^{-5}	5.0^{-5}
NODE output dim.	6	6
NODE hidden layers	$[150] \times 3$	$[1000] \times 3$
NODE activation	elu	elu
sNODE hidden layers $\hat{f}$	$[100] \times 3$	$[1000] \times 3$
sNODE hidden layers V	$[100] \times 3$	$[100] \times 3$
Task Emb. dim.	256	256
HN hidden layers	$[300] \times 3$	$[300] \times 3$
β	5.0^{-3}	5.0^{-3}
Chunk Emb. dim.	-	256
Chunk dim.	-	8192
Kaiming uniform a	$\sqrt{5}$	$\sqrt{5}$
Xavier uniform gain	1.0	1.0

Table A.1: Hyperparameters Experiment 1 & 2

HN	NODE - HN	CHN	NODE - CHN
$[303] \times 2$	$[211] \times 2$	$[309] \times 2$	$[1400] \times 2$
$[300] \times 3$	$[150] \times 3$	$[300] \times 3$	$[1000] \times 3$
$[298] \times 4$	$[123] \times 4$	$[292] \times 4$	$[800] \times 4$
$[292] \times 8$	$[81] \times 8$	$[265] \times 8$	$[600] \times 8$

Table A.2: Experiment 3 Layers for HN and CHN in combination with NODE.

HN	$\hat{f}$ - HN	V - HN	CHN	$\hat{f}$ - CHN	V - CHN
$[302] \times 2$	$[182] \times 2$	$[70] \times 3$	$[310] \times 2$	$[1412] \times 2$	$[100] \times 3$
$[300] \times 3$	$[130] \times 3$	$[70] \times 3$	$[300] \times 3$	$[1000] \times 3$	$[100] \times 3$
$[298] \times 4$	$[108] \times 4$	$[70] \times 3$	$[291] \times 4$	$[817] \times 4$	$[100] \times 3$
$[290] \times 8$	$[70] \times 8$	$[70] \times 3$	$[264] \times 8$	$[535] \times 8$	$[100] \times 3$

Table A.3: Experiment 3 Layers for HN and CHN in combination with sNODE.

Appendix B

Experiment Results

B.1 Experiment 1: Optimizers

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.858765	0.978395	0.999930	0.695162	0.000000	1.000000	0.755375	0.611595
CHN	0.960494	0.980864	0.999643	0.700547	0.802754	1.000000	0.907384	0.874271

Table B.1: Experiment 1 NODE Adam

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.910617	0.990741	0.999930	0.684980	0.000000	1.000000	0.764378	0.606501
CHN	0.914568	0.965432	0.999643	0.706706	0.802754	1.000000	0.898184	0.880650

Table B.2: Experiment 1 NODE RMSProp

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.018765	0.961111	0.999930	0.709651	0.000000	1.000000	0.614909	0.518571
CHN	0.155556	0.711728	0.999643	0.717315	0.802754	1.000000	0.731166	0.689718

Table B.3: Experiment 1 NODE SGD

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.994074	0.998765	0.999900	0.859963	0.000000	1.000000	0.808784	0.599934
CHN	0.985185	1.000000	0.999643	0.873088	0.721402	1.000000	0.929886	0.886468

Table B.4: Experiment 1 sNODE Adam

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.993580	0.998148	0.999900	0.865822	0.000000	1.000000	0.809575	0.599880
CHN	0.940741	0.976543	0.999643	0.871792	0.721402	1.000000	0.918353	0.892186

Table B.5: Experiment 1 sNODE RMSProp

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.354074	0.512963	0.999900	0.879068	0.000000	1.000000	0.624334	0.594463
CHN	0.574815	0.649383	0.999643	0.870197	0.721402	1.000000	0.802573	0.818717

Table B.6: Experiment 1 sNODE SGD

B.2 Experiment 2: Initializers

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.858765	0.978395	0.999930	0.676205	0.000000	1.000000	0.752216	0.610931
CHN	0.960494	0.980864	0.999643	0.698256	0.802754	1.000000	0.907002	0.873517

Table B.7: Experiment 2 NODE Kaiming

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.918025	0.988889	0.999930	0.679123	0.000000	1.000000	0.764328	0.605904
CHN	0.433086	0.401235	0.999643	0.696525	0.802754	1.000000	0.722207	0.736219

Table B.8: Experiment 2 NODE Xavier

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.859259	0.954321	0.999930	0.676564	0.000000	1.000000	0.748346	0.613600
CHN	0.905679	0.949383	0.999643	0.697720	0.802754	1.000000	0.892530	0.879635

Table B.9: Experiment 2 NODE PWI

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.994074	0.998765	0.999900	0.858395	0.000000	1.000000	0.808522	0.599973
CHN	0.993580	1.000000	0.999643	0.897729	0.721402	1.000000	0.935392	0.887685

Table B.10: Experiment 2 sNODE Kaiming

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.993580	0.999383	0.999900	0.971764	0.000000	1.000000	0.827438	0.594495
CHN	0.988642	0.985802	0.999643	0.850709	0.721402	1.000000	0.924366	0.885185

Table B.11: Experiment 2 sNODE Xavier

MET	ACC	REM	MS	TE	FS	SSS	CL_{sco}	CL_{stab}
HN	0.975802	0.985802	0.999900	0.860894	0.000000	1.000000	0.803733	0.602756
CHN	0.985679	0.991975	0.999643	0.974913	0.721402	1.000000	0.945602	0.889763

Table B.12: Experiment 2 sNODE PWI