UNIVERSITY OF APPLIED SCIENCES COLOGNE
FACULTY OF INFORMATION, MEDIA AND ELECTRICAL ENGINEERING

A MASTER THESIS IN PARTIAL FULLFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

A MODEL-AGNOSTIC META-LEARNING FRAMEWORK FOR SOLVING FEW-SHOT TASKS IN PARALLEL

SUBMITTED BY:
FLORIAN SOULIER

SUPERVISED BY
PROF. DR. RER. NAT. BEATE RHEIN
M. SC. JAN BOLLENBACHER

SUBMITTED IN
MAY 2020

By far, the greatest danger of Artificial Intelligence is that people conclude too early that they understand it.

— Elizer Yudowsky

ACKNOWLEDGMENTS

My special thanks go to my supervisors. I would like to thank Prof. Dr. Beate Rhein for supervising my master thesis and for allowing me the freedom to pursue my research without being restricted or pushed in one direction.

I would like to thank Jan Bollenbacher, who offered me a topic from his research focus that has brought me deep into current research. Thanks for the many food for thought as well as the intensive supervision during the execution. The results of this work were made possible by working together on our deep learning framework, which I used to perform the measurements presented in this thesis.

Lastly I would like to thank my family, especially my parents, my grandmother and my uncle, who have always supported my desire for continuous education with advice, patience and money, even though my studies took a little longer than expected.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

DMAML	Distributed Model-Agnostic Meta-Learning
DQN	Deep Q Network
FOMAML	First-Order Model-Agnostic Meta-Learning
GAI	General Artifical Intelligence
GD	Gradient Descent
MAML	Model-Agnostic Meta-Learning
MANN	Memory-Augmented Neural Networks
ML	Meta-Learning
MSE	Mean Squared Error
PPO	Proximal Policy Optimization
SGD	Stochastic Gradient Descent
TRPO	Trust Region Policy Optimization
RL	Reinforcement Learning

1

INTRODUCTION

MOTIVATION

Humans have the remarkable ability to learn new concepts and tasks using only a few samples. To achieve this, they build on previously gained experience and reuse the concepts learned there. However, this is not done by pure reapplication but by the ability of abstraction. For example, humans are able to learn to grasp a cup and can also apply this to a bottle or a hammer without having learned it explicitly beforehand. The question that arises is how do we manage to build machine learning systems that can learn in the same very versatile way?

In machine learning, the widespread standard paradigm prevails of focusing on a task and training a model or a policy for it from end to end. The emphasis is on training randomly initialized models. This is the "gold standard" because it does not require substantial domain-specific human knowledge to learn the task [4]. From the human learning standpoint, this is like teaching a child to distinguish between dogs and birds, but without understanding the fundamental differences that make the difference. If we want systems that reach the generality of human intelligence, they must be able to learn tasks, concepts or environments without requiring millions of data points.

Take the Omniglot dataset as an example: It is a set of different letters from 50 alphabets. It has more than 1600 classes, but has only 20 examples for each class. A human being can learn to distinguish a class from other classes with just a few examples. A classifier trained with supervised learning would instead need thousands of examples from each class to classify them correctly. The significant difference between humans and the classifier is that humans benefit from having already learned similar concepts and abstracted them. We benefit from already learned experience and therefore quickly adapt to a new task, whereas the classifier does not rely on experience and starts learning from scratch. Research in the field of Meta-Learning (ML) deals with this kind of problem.

OUR CONTRIBUTION

The area of meta-learning is currently enjoying great popularity and new papers are regularly published in this field. There are four popular approaches how such a meta-learner works. In this study, we are dealing with the model-agnostic approach [5] [4]. Since Model-Agnostic Meta-Learning (MAML) can be applied to all gradient descent based algorithms, it is a very versatile application and research opportunity.

In this study we implement the MAML algorithm in a way that all three classic use cases, regression, classification and reinforcement learning, are realized together in a single framework in the current TensorFlow and Python versions. Currently there are only a few implementations in TensorFlow 2 and none that provide all three use cases in a single framework. Furthermore we give the possibility to investigate many different research topics by a large number of controllable hyperparameters without the need to change the source code.

Beyond that we extend the MAML algorithm by a parallelization possibility, which is based on top of the MAML algorithm. The idea is based on the success of parallelization algorithms like A3C [9] or Ape-X [6] from the reinforcement learning area. The research focus of this study is to determine whether the MAML algorithm can be accelerated by parallelization. For this purpose, we propose the Distributed Model-Agnostic Meta-Learning (DMAML) algorithm, an encapsulation of the MAML algorithm, which allows parallelization.

Part I

THEORY

2

META-LEARNING

2.1 LIMITS OF CLASSIC MACHINE LEARNING APPROACHES

Classic machine learning fields are supervised learning, unsupervised learning and Reinforcement Learning (RL). Supervised learning applications are e. g. classification and regression tasks. Unsupervised learning applications deal with problems like clustering or anomaly detection. Reinforcement learning applications are those, where a software agent can learn in an interactive environment and is trained by a feedback signal (reward) that gives information about how good a chosen action in a specific state was.

In this work, we deal with problems from regression, classification and reinforcement learning. All these applications have in common that they can be used with neural networks which use Gradient Descent (GD) for updating their weights. These kinds of machine learning also share two common problems. First, they need a mass of data to be trained effectively on a problem. Secondly the applications are specialized to one specific task. If we change the classes of a classifier or the environment of a RL agent, these approaches fail.

This is where classic machine learning approaches fail, but humans can shine. Humans are generalists that can quickly learn to recognize objects from few examples or quickly learn and adapt new skills after just minutes of experience.

As mentioned in [5], this kind of learning, where a classifier or an agent needs to learn from few samples, and then further adapt as soon as new samples become available, is challenging. We call this kind of problems few-shot problems. The main problem is to integrate its prior experience with a small amount of new information, while avoiding overfitting to the new data. Since the prior experience and the new information are dependent on the actual task, the learner (meta-learner) should be general to the task.

2.2 FEW-SHOT LEARNING

In contrast to classic machine learning tasks, where a huge set of data is given for training, few-shot learning describes a setting where *fewer* data points are available. *Few* therefore stands for a small number of data points available for *training*, which is denoted by *k*. We call this a *k-shot problem*.

To better understand few-shot learning, classification is used as an example in this section. Imagine a setting, where images have to be distinguished in two classes. Such classes could be *dog*, *cat*, *bird* or *squirrel*. If only one example of each animal is used, the parameter k is equal to 1. We call this a 1-shot problem. The parameter n is used to describe the number of classes that have to be distinguished by the classifier in one task.

In this case, a task could be to distinguish *dog* and *cat*. Another task could be to distinguish *dog* and *squirrel*. So generally spoken a task is an entity being learned which corresponds to an objective, domain or environment [4]. Concretly in this example, the task is to distinguish two images and classify them into the two classes of the distribution of classes, which have been mentioned above. Since in these tasks it is always distinguished between two classes, the parameter n is equal to 2. A problem like this is called *n-way k-shot problem*. This example therefore is a 2-way 1-shot problem.

There are two key differences between a classical and a few-shot classification. First, a classical trained classifier, that is trained to distinguish *cats and dogs* will only be able to distinguish these two classes, but not *dogs and birds*, for instance. This is because, the classes changed and the classifier can not generalize to abstract features of classes. It would be necessary to train a new model for every combinations of classes.

Second, the classifier would have a low validation accuracy in classifying, never seen instances of the classes dogs and cats, because it only has been trained on very few images and therefore has *overfitted* very hard to these few examples.

To solve the problems caused by gradient descent, i.e. overfitting and lack of abstraction, few-shot learning algorithms must learn more general features than classical end-to-end machine learning algorithms.

2.3 SEPARATION OF LEARNING TERMS

Regarding literature on this topic, we stumble over various terms and concepts. Some of these concepts shall be very briefly discussed here. In this section, the terms transfer learning, multi-task learning and meta-learning are separated from each other and explained using examples.

2.3.1 *Transfer Learning*

Transfer learning is a research problem in which one task is learned to solve and the solution is applied to a smiliar but not identical other task. For example, imagine a classification task in which a huge set of different animals shall be classified. The learned features can be taken to solve a different classification task, in which e. g. a smaller subset of this dataset is used. This works, because the classifier has learned a set of features that are applicable to the new task. In this example the nature of the data is the same. To adapt the model to the new classes, the last layer can be retrained with some samples, while the rest of the model remains fixed [12].

2.3.2 *Multi-Task Learning and Meta-Learning*

In multi-task learning, a model is trained on a set of tasks with the goal, that a model is learned, that is able to solve a finite number of trained tasks with minimal amount of data. For example, imagine a RL setting: An agent is trained to navigate through a set of different mazes and will be able to navigate through *one of these* mazes, with only a few samples used for training.

In meta-RL, the goal is to make an agent learn a policy for a *new task* using only a small amount (few shots) of experience in the training. A new task to learn will then be in a new environment, or if in the same environment, will change its goal [5]. Following the previous example, an example of meta-RL would be that the agent now has to solve a completely unknown maze with a small number of samples. An intuitive illustration of this distinction is given in Figure 2.1.

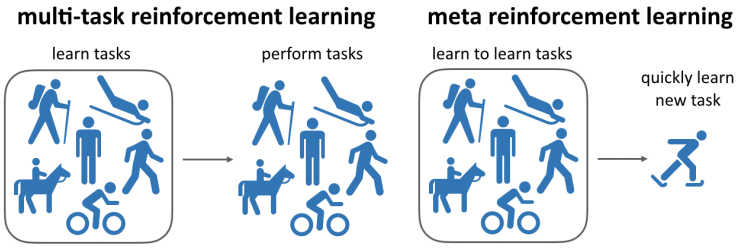


Figure 2.1: Intuitive distinction between multi-task and meta-reinforcement learning. Left: Agent can perform a task from a set of learned tasks. Right: Agent can quickly learn a new, never seen task, based on its experiences of other learned tasks.[1]

2.3.3 *Summary*

Transfer learning is based on supervised learning, where a model is learned in a classical way. The learned features are applied to a new, similar problem.

Multi-task learning aims to solve not only one but several tasks with the same model. However, the number of tasks is limited to those used in the training.

Meta-learning aims to solve new, never seen tasks with a few samples and is therefore the most universal approach to solve similar problems with the same model.

2.4 OVERVIEW OF TYPES OF META-LEARNING

As described in the previous chapter, ML aims at learning new tasks with a few samples. ML thus takes a step towards General Artifical Intelligence (GAI) and moves one step closer to human learning. So far four types of ML have been established. In the following, these are briefly described.

2.4.1 *Metric-based ML*

In metric-based ML an appropriate metric is used to train the network. An example of this approach is a classifier that shall decide if two

1 https://meta-world.github.io

images are showing the same object. For this, the network is trained to learn the features of the images. Afterwards the distance between the features of the two images is calculated to decide if the two pictures are similar or different [14]. Examples are [7], [18], [17].

2.4.2 *Model-based ML*

In model-based ML no assumption regarding the form of the probability distribution of labels is made. They depend on a model, specifically designed for fast learning. The model updates its parameters rapidly with few training steps. These updates can be achieved by its internal architecture or controlled by another meta-learner model. So-called Memory-Augmented Neural Networks (MANN) are suitable for this kind of problems. They use explicit storage buffers to remember already learned features to solve the problem [19]. Papers dealing with this kind of ML are [16], [10].

2.4.3 *Optimization-based ML*

In this approach the optimizer itself is learned. Since in Few-Shot Learning the training dataset is very small and the number of gradient steps is small, gradient descent methods fail, as described in a preceding section. Therefore, two networks are used, a base-network that tries to learn the task and a meta-network that optimizes the base network. [14] [19]. Papers dealing with this approach are [1], [13].

2.4.4 *Model-agnostic ML*

Since the approach described here is still very new in research, this classification is made differently in the literature. In [19] these approaches would count as optimization-based and in [14] it is called "learning the initilizations". In the model-agnostic approach, the algorithm does not rely on any assumptions of the model. Similar to optimization-based ML, there are two networks, a base model and a meta-model. The base model is trained to fit to the actual task. The meta-model then is trained with batches of tasks and learns from the base models. Another point is, that this approach aims at learning initial parameters of the meta-model, to provide a good starting point for learning new tasks rapidly. Papers that address this approach are [5], [11], [21], [2].

3

MODEL-AGNOSTIC META-LEARNING

In this section we describe the reasons why we chose MAML to be the subject of our research and the key ideas of this approach. Afterwards we present the theory of MAML.

3.1 KEY IDEAS OF MAML

We chose MAML to be the subject of our work, since it seems to be the most promising approach in the field of meta-learning at this time. The results of [5] are significant in a variety of few-shot learning settings, such as regression, classification and reinforcement learning.

MAML is *general to the task*, in the sense of the form of computation required to complete the task. That means the general algorithm does not change, if the problem that has to be solved changes. More specifically there are no new hyperparameters introduced, that we have to evaluate.

MAML is *model-agnostic*, in the sense that it can be directly applied to any learning problem or model that is trained using gradient descent. Most algorithms and machine learning fields use gradient descent, therefore it is an algorithm that can be widely used. Since in [5] the authors focus on deep learning, it seems to be proven, that MAML fits the needs of today's few-shot learning problems and network architectures.

As described earlier, in meta-learning, the goal of the trained model is to quickly learn a *new* task from a small amount of new data. In MAML the initial parameters θ of a meta-model are trained in such a way that the model has maximal performance on a new task after the parameters have been updated through one or more gradient steps computed with a small amount of data from that new task. Therefore, MAML does not need to expand the number of learned parameters nor place constraints on the model architecture [5].

As the authors of [5] state, the process of training a meta-model with a small amount of gradient steps can be viewed from different

standpoints. From a *feature learning standpoint*, we can describe this process as building an internal representation that is broadly suitable for many tasks. Since the representation is suitable for many tasks, fine-tuning the parameters slightly can produce good results. Therefore, it can be sufficient to only modify the last layer of a network.

From a *dynamical systems standpoint*, this learning process can also be described as maximizing the sensitivity of the loss functions of new tasks with respect to the parameters. Since the sensitivity of a trained model is high, only small changes to the parameters (by using a small number of gradient descent steps) is sufficient to lead to large improvements in performance.

3.2 META-LEARNING PROBLEM AND TERMINOLOGY

In MAML a model, denoted f is considered, that maps observations x to outputs y. During meta-learning, the model is trained to be able to adapt to a large or infinite number of tasks. Since MAML is to be applied to a variety of different meta-learning problems, a generic notation was proposed in [5]. Since the notation in [4] changed to a much more precise and clearer notation, it is used in this paper. Formally each task can be described as follows:

$$\mathcal{T} = \{\mathcal{L}_{\mathcal{T}_i}, \rho(x_1), \rho(x_{t+1} \mid x_t, y_t), \mathcal{H}\}$$

The task $\mathcal{T}$ consists of a *loss function* $\mathcal{L}$ which takes as input the model's parameters θ and a dataset $\mathcal{D}$, a *distribution over initial observations* $\rho(x_1)$, a *transition distribution* $\rho(x_{t+1} \mid x_t, y_t)$ and an *episode length* $\mathcal{H}$. In identically and independent distributed supervised learning problems $\mathcal{H} = 1$ and a dataset $\mathcal{D} = \{(x_1, y_1)^{(k)}\}$ consists of labeled input, output pairs.

In reinforcement learning setting the model f may generate samples of length $\mathcal{H}$ by choosing an output $\hat{y}_t$ at each time step t. Hence, the dataset consists of trajectories rolled-out by the model $\mathcal{D} = \{(x_1, \hat{y}_1, ..., x_H, \hat{y}_H,)^{(k)}\}$. $\mathcal{H}$ describes the maximum number of timesteps per trajectory. That means, the reward will be calculated over $\mathcal{H}$ timesteps per trajectory.

The *loss function* $\mathcal{L}_{\mathcal{T}_i}$, which is dependent on θ and $\mathcal{D}_{\mathcal{T}_i}$, provides task-specific feedback in form of missclassification loss in a classification problem or a reward signal in a RL task.

The *distribution of inital observations* $\rho(x_1)$, is the possible first observation (state) given back from a RL environment or the first set of n images of k sets in a n-way k-shot classification problem.

The *transition distribution* $\rho(x_{t+1} \mid x_t, y_t)$ describes the transition from the observation x_t at a certain timestep to the next observation x_{t+1}. The transition distribution describes what observation will be seen next, if observation x_t is observed and output y_t has been produced by the policy. This describes the policy of an RL agent, where the input x_{t+1} is the predicted next observation that the agent will receive from the environment, if observation x_t is received and action y_t is performed.

Since different terms are used in the literature regarding meta-learning, in the following the terms are uniformly defined by subject area as used in this paper.

DATASET PREPARATION. In few-shot learning, the data must be pre-splitted. Therefore we split the data into a training set $\{\mathcal{D}_{\mathcal{T}_i}\} = \{D_{\mathcal{T}_1}, D_{\mathcal{T}_2}, ..., D_{\mathcal{T}_N}\}$ and a validation set $\{\mathcal{D}_{\mathcal{T}_j}\} = \{D_{\mathcal{T}_I}, D_{\mathcal{T}_{II}}, ..., D_{\mathcal{T}_M}\}$. The training set is used to train the meta-model and the validation set is used to validate the meta-model. In the meta-learning setting there is no need for a classic three-way split of training, validation and test set, since validation and testing are the same in meta-learning and have no influence on the optimization of the model.

TRAINING. At training time a task $\mathcal{T}_i$ is drawn from a distribution of tasks $p(\mathcal{T})$ that uses data from the training set $\{\mathcal{D}_{\mathcal{T}_i}\}$. A copy of the model is trained to learn the new task from only $k_{support}$ many samples drawn from the support set ρ_i, denoted as $D^{tr}_{\mathcal{T}_i}$ and feedback $\mathcal{L}_{\mathcal{T}_i}$ generated by $\mathcal{T}_i$. Afterwards the model-copy is tested with the k_{query} many samples from the query set $D^{test}_{\mathcal{T}_i}$. The meta-model is then improved by considering how the test error on the new data from the query set $D^{test}_{\mathcal{T}_i}$ changes with respect to the parameters. The test error of $\mathcal{T}_i$ of the model-copy serves as the training error for training process of the meta-model.

VALIDATION. After meta learning, the learned algorithm (meta-model) is evaluated in its ability to learn new tasks from the validation set $\{\mathcal{D}_{\mathcal{T}_j}\}$. Therefore new tasks $\mathcal{T}_j$ are drawn from the validation tasks $\{\mathcal{T}_j\}$. For every validation task a model-copy will be created. This copy will be trained with $k_{support}$ samples from the support set and afterwards it will be tested with k_{query} samples from the query set. The test error is used as validation accuracy or loss.

To summarize the chapter, all terms are once again listed in Table 3.1.

Table 3.1: Summary of Meta-Learning Terminology Used in this Paper.

Symbol	Terminology	Explanation
$\mathcal{T}$	task	entity or objective being learned
$p(\mathcal{T})$	task distribution	distribution of tasks from which the training and validation tasks are drawn (the dataset)
$\{\mathcal{T}_i\}$	training tasks	set of tasks used for training
$\{\mathcal{D}_{\mathcal{T}_i}\}$	training set	set of datasets corresponding to the training tasks
$\{\mathcal{T}_j\}$	validation tasks	set of tasks used for evaluation
$\{\mathcal{D}_{\mathcal{T}_j}\}$	validation set	set of datasets corresponding to the validation tasks
$\mathcal{D}_{\mathcal{T}}^{tr}$	support set	training data for task $\mathcal{T}$ with $k_{support}$ data points sampled from $\mathcal{D}_{\mathcal{T}}$
$\mathcal{D}_{\mathcal{T}}^{test}$	query set	test data for task $\mathcal{T}$ with k_{query} data points sampled from $\mathcal{D}_{\mathcal{T}}$

ALTERNATIVE TERMINOLOGY. Note that often other terms are used. Training set is often called meta-training set and validation set is often called meta-test set. Analougus the support set is called training set and the query set is called test set. Since these terms can cause confusion, we remain with the in this section defined terms.

To anticipate confusion, it should be mentioned that in [5] and [4] different mathematical notations were sometimes used. In this thesis a subset of both notations will be used with the aim to improve the understanding in every situation. However, different notations for the same context are not mixed, so that the notation used here remains consistent and unambiguous.

3.3 MODEL-AGNOSTIC META-LEARNING ALGORITHM

MAML is a model- and task-agnostic algorithm for meta-learning that trains a model's parameters such that a small number of gradient updates will lead to fast learning on a new task. To achieve that, MAML is trained on batches of tasks from the distribution of tasks $p(\mathcal{T})$. This results in a pretrained model that can easily and fastly trained to a new task.

As described in [5] the intuition behind MAML is, that some internal representations (features) are more transferable than others. In other words, the model will learn features that are applicable to all tasks rather than learning features that are specific to one task while ommitting very task-specific features. This meta-model can already achieve good results on a new task, because it already has learned features, that are applicable to the new task. To achieve even better results, the meta-model can be *fine-tuned* on a specific task. Fine-tuning means, that a meta-model which was trained on a distribution of tasks $p(\mathcal{T})$ applied to a *new* task by applying one or few gradient steps.

As described in [5] these models are *sensitive* to changes in the task, such that small changes in the parameters will produce large improvements on the loss funtion of any task drawn from $p(\mathcal{T})$ when altered in the direction of that loss (Figure 3.1).

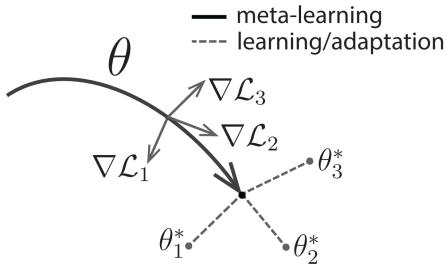


Figure 3.1: Diagram of MAML that can quickly adapt to new tasks: The meta-model with parameters θ is trained on three tasks and gets three gradients $\nabla\mathcal{L}_i$. These gradients are summarized and applied to the meta-model which results in a new position of the model in the θ-space. The new parameters are task-agnostic and adapted so that on average they can process all tasks from the distribution equally well. The model can be rapidly fine-tuned to fit to one specific task to achieve better results. The ideal parameters for a specific task are marked with an asterisk. [5]

It is assumed, that the loss function is smooth enough to use gradient descent and the model f_θ is parameterized by some parameter vector θ. When the model is adapted to a new task $\mathcal{T}_i$, the parameters become θ_i'. The updated parameters vector θ_i' is computed by using one or more gradient descent updates on the samples from support set $\mathcal{D}_{\mathcal{T}_i}^{tr}$ of task $\mathcal{T}_i$. Gradient updates used on tasks are called *fast-updates* and the resulting weights are called *fast-weights*. Equation 3.1 is used for these updates.

$$\theta_i' = \theta - \alpha\nabla_\theta\mathcal{L}_{\mathcal{T}_i}(f_\theta) \tag{3.1}$$

The learning rate α can be set as fixed hyperparameter or can be learned. The meta-model parameters are trained by optimizing for

the performance of the task-specific models $f_{\theta_i'}$ with respect to the meta-model parameters θ across tasks sampled from $p(\mathcal{T})$. That concretely means, that the sum of all task-losses shall be minimized by changing the weights of the meta-model. The meta-objective can be mathematically described with Equation 3.3.

$$\min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) = \min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)}) \tag{3.2}$$

In its simplest form, meta-optimization across a batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$ is performed via mini-batch Stochastic Gradient Descent (SGD), as described in [5], such that the meta model parameters θ are updated as follows:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) \tag{3.3}$$

The parameter β is the meta-learning rate. This parameter can also be set fixed or learned. In the published source code of [5] and in our implementation Adam Opzimizer is used. Since the paper's source code is written in TensorFlow 1 and ours in TensorFlow 2, we set the initial configuration parameters of Adam to the initial parameters of TensorFlow 1 to guarantee same behavior of their code and ours. MAML is described in Algorithm 1.

Algorithm 1 Model-Agnostic Meta-Learning by Finn et al. in [4][5]

Require: $p(\mathcal{T})$ distribution over tasks
Require: α, β learning rate hyperparameters
1: Randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample support set $\mathcal{D}^{tr}_{\mathcal{T}_i} \sim \mathcal{D}_{\mathcal{T}_i}$
6: Sample query set $\mathcal{D}^{test}_{\mathcal{T}_i} \sim \mathcal{D}_{\mathcal{T}_i}$ depending on $\mathcal{D}^{tr}_{\mathcal{T}_i}$
7: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ w. respect to $k_{support}$ examples from $\mathcal{D}^{tr}_{\mathcal{T}_i}$
8: Compute adapted parameters with GD: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ with respect to k_{query} examples from $\mathcal{D}^{test}_{\mathcal{T}_i}$
11: **end while**

The general algorithm requires a distribution over tasks, that shall be learned and the inner and meta-learning rates α and β. The meta-

model is initialized randomly. The algorithm runs for a specified boolean termination condition or a fixed number of epochs (Line 2).

In each epoch a batch of tasks is drawn and iterated through (Line 3). For every $\mathcal{T}_i$ a support set and a query set is sampled. The support set is then applied to the model-copy and the gradients of the losses are evaluated. The model-copy is updated to the fast-weights θ_i'. This is repeated for all task of the current batch. The tasks and their corresponding query sets as well as the model copy have to be stored.

After the iteration of the batch is finished, the meta-model can be improved. This is done by evaluating the query sets on the model-copies and summing up the losses. The loss sum is then applied on the meta-model, by calculating the gradient with respect to the meta-model's parameters θ (Line 10). The test losses of the model-copies serve as training errors for the meta-model.

3.3.1 *Second-Order Model-Agnostic Meta-Learning*

In the following, the mathematical aspects of the meta-optimization of Equation 3.3 are addressed. The meta-optimization across tasks via stochastic gradient descent with second order update is defined as follows:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$$

Since $\mathcal{L}_{\mathcal{T}_i}$ is differentiable, it can be written as:

$$\theta \leftarrow \theta - \beta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$$

By applying the chain rule for $\nabla_\theta(\mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})) = \nabla_\theta \theta_i' \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ the Equation changes to:

$$\theta \leftarrow \theta - \beta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \nabla_\theta \theta_i' \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$$

When the update rule in Line 8 from Algorithm 1 is applied, the equation changes to

$$\theta \leftarrow \theta - \beta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \nabla_\theta (\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})) \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) \tag{3.4}$$

By applying the association property, the ∇_θ-operator can be dragged into the brackets and it becomes visible, that in MAML a second order derivation is needed.

$$\theta \leftarrow \theta - \beta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} (\nabla_\theta \theta - \alpha \nabla_\theta^2 \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})) \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) \tag{3.5}$$

This can be further be processed to

$$\theta \leftarrow \theta - \beta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} (I - \alpha \nabla_\theta^2 \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})) \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) \tag{3.6}$$

because $\nabla_\theta \theta$ is the identity matrix I. Mathematically it is possible to summarize losses and calculate the gradient from that summed up losses or to calculate the gradient from every loss and sum up the gradients.

3.3.2 *First-Order Model-Agnostic Meta-Learning*

For computation of the update rule as shown in Equation 3.6, Hessian-vector products are needed which slow down training. This is the case because the meta-optimization goal depends on θ, so that second-order derivations are needed. The authors of [5] and [4] state that, by omitting the second-order derivative, a speed-up by 33% can be achieved in simple problems like few-shot image recognition by maintaining the same performance. In [4] the meta-optimization goal is therefore supplemented by a stop-gradient operation. With that operation, the parameter update is treated as a constant, so that no second-order derivation is used. In [11] it is shown, that in the First-Order Model-Agnostic Meta-Learning (FOMAML) version the Hessian matrix is simply replaced by the identity matrix (Equation 3.7).

$$\theta \leftarrow \theta - \beta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} I \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) \tag{3.7}$$

By using only the the first derivation a massive speed-up can be achieved by accepting only a small error, since the values of the second order derivatives of the Hessian matrix are comparatively small.

3.3.3 *Meta-Update with Sum and Arithmetic Mean*

In literature meta-training in MAML is always performed with a sum of gradients as it can be seen in Equation 3.3. They also call α and

β step size parameter (in this thesis they are called learning rates). The step size can be seen as the product of the learning rate β and the gradient sum. The gradient sum depends on the number of tasks $|\{\mathcal{T}_i\}|$ used for training.

As long as the number of tasks, for example in benchmarks, is not changed, these step sizes can be considered constant. However, as soon as the number of tasks parameter is increased or decreased, the step size is no longer constant and increases or decreases by the same factor as the number of tasks is changed.

This can have positive effects because greater gradients can result in greater step sizes in training. However, if the step size becomes larger, comparison in a benchmark becomes unfair.

For this reason, in this thesis the possibility to train the meta-model with the arithmetic mean instead of the sum of the gradients is implemented. The meta-update then changes as shown in Equation 3.8. The first-order MAML update is analogous changed. Some experiments have been performed in this way.

$$\theta \leftarrow \theta - \beta \frac{1}{|\{\mathcal{T}_i\}|} \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) \tag{3.8}$$

3.4 CLASSIFICATION TASKS

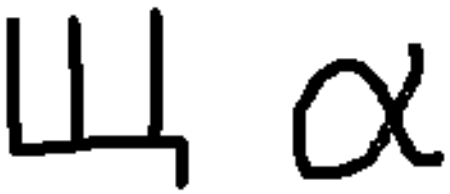
In this section a few-shot learning scenario for image classification will be described. We look at following use case: A dataset with many different classes but only few instances is given. A n-way k-shot image classifier shall be trained that is capable of classifying many different classes by being trained with very few examples of them.

3.4.1 *Image Datasets*

In this paper, few-shot classification is performed on Omniglot [8] and MiniImageNet [15] datasets as done in [5]. We use these datasets, because they are recently used as benchmark in [5], [13] and [18].

The Omniglot dataset is composed of 20 instances of 1623 characters from 50 different alphabets. Each instance was drawn by another person. Each character is treated as own class and is grayscale encoded,

they have a size of 24x24 pixels. Moreover we added versions which are rotated by multiples of 90° and treat them as own classes (see [5]). Eventually we get an Omniglot dataset with 6492 classes. We use a split from [8] with 3760 training classes and 2732 validation classes[1]. In this split rotations of images are not drawn from training set over to validation set and vice versa. Figure 3.2 shows typical examples of the Omniglot dataset.



(a) Cyrillic letter Schah (b) Greek letter Alpha

Figure 3.2: Omniglot dataset examples

The MiniImageNet dataset has been used by [13], [18] and [5]. It involves 600 instances of 64 training classes, 16 validation classes and 20 test classes. All images are RGB-colored and have a size of 84x84 pixels. Since we need only training and validation sets, the test classes also count as training classes, so that there are 84 training and 16 validation classes. Figure 3.3 shows typical examples of the MiniImageNet dataset.

(a) Bloodling on different backgrounds (b) Guitar in different settings

Figure 3.3: MiniImageNet dataset examples

1 https://github.com/brendenlake/omniglot

3.4.2 *Sampling and Preprocessing*

For image classification Omniglot and MiniImageNet datasets have been used. In the following sampling and preprocessing of Omniglot and MiniImageNet will be described. Every class in the Omniglot dataset has 20 instances, drawn from different persons. An example is illustrated in Figure 3.4.

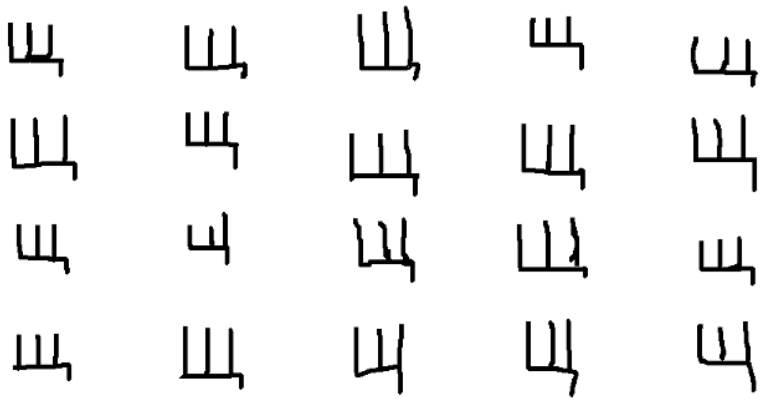


Figure 3.4: Instances of a class from Omniglot dataset

From this class $k_{support}$ many instances will be randomly drawn for the support set and k_{query} many instances for the query set. Doublings are excluded. In a 5-way 1-shot problem, there will be 5 support sets and 5 query sets, one for each class. From these sets, the task $\mathcal{T}_i$ is composed. A support set of such a task is illustrated in Figure 3.5.

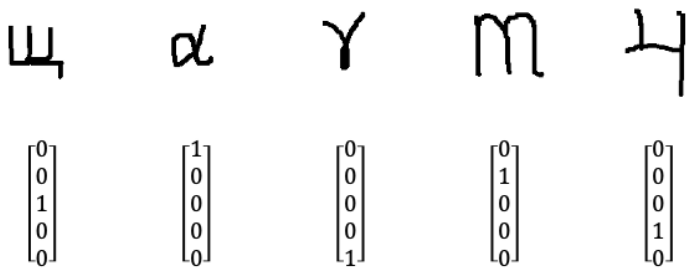


Figure 3.5: Support set of task $\mathcal{T}_i$: A support set with $n = 5$ classes and $k_{support} = 1$ instances per class with coresponding labels.

As it can be seen, the support set consists of 5 classes, with $k_{support} = 1$ instance of each class. If $k_{support}$ would be greater, more of these sets would be generated and concatenated. When the images are sampled, the classes are randomly assigned a label that remains the same for the duration of the task being trained. These are one-hot encoded and the label vector has as many elements as there are classes.

The query set is produced the same way. In MiniImageNet, for instance, $k_{query} = 5$, so that the query set will have 5 instances of this task.

A created task will be preprocessed after it has been created. The Omniglot images will be resized to 28x28 pixels to spare computation time and normalized into a range between 0 and 1 to reduce covariance shift. After that the grayscale images are inverted. This is done to support learning process. Since most pixels of the image are white (encoded as 1) the network would learn the shape that is not filled with the actual content that should be learned. By inverting all pixels, only the actual content is near or equal to 1 while everything else is 0 and the meta-learner learns the features of the actual image.

The MiniImageNet images will be resized to 84x84 pixels and normalized into a range between 0 and 1 too, but not inverted, since the pictures are fully filled with content.

The images are then permutated. The permutation keeps the mapping of image and label intact, but shuffles their positions in *support set* and *query set*. This is done to ensure that no statistical effects, for example in a normalization layer of the model, can occur.

3.4.3 *Training*

With few-shot supervised learning, the same steps are applied as in Algorithm 1. The only difference is the sampling of the datapoints. The adapted version can be seen in Algorithm 2.

Algorithm 2 MAML for Few-Shot Supervised Learning by Finn et al. in [5] [4]

Require: $p(\mathcal{T})$ distribution over tasks
Require: α, β learning rate hyperparameters
1: Randomly initialize θ
2: **while** not done **do**
3: Sample batch of task $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample $k_{support}$ datapoints $\mathcal{D}^{tr}_{\mathcal{T}_i} = \{x^{(i)}, y^{(i)}\}$ from $\mathcal{D}_{\mathcal{T}_i}$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 5.2 or Equation 5.3
7: Compute adapted parameters with gradient descent: $\theta'_i = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample k_{query} datapoints $\mathcal{D}^{test}_{\mathcal{T}_i} = \{x^{(i)}, y^{(i)}\}$ from $\mathcal{D}_{\mathcal{T}_i}$ depending on $\mathcal{D}^{tr}_{\mathcal{T}_i}$ for the meta update
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta'_i})$ using each $\mathcal{D}^{test}_{\mathcal{T}_i}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 5.2 or 5.3
11: **end while**

The algorithm requires a distribution of tasks, as well as the inner-leraning rate α and meta-learning rate β. The weights θ are initialized randomly. The algorithm runs for a predefined number of epochs or until an early stopper stops the execution (Line 2). In each epoch a batch of tasks from the distribution of tasks $p(\mathcal{T})$ is sampled. The following part is repeated for each task from the batch of tasks:

First, datapoints for the support sets are collected for the tasks (Line 5). Thereby $k_{support}$ many pairs of instances of the class and corresponding label for the support set are sampled. The process was explained in detail in the previous section. Then the gradient of the loss is calculated and the meta-model is trained (Lines 6 and 7). Then k_{query} many pairs of instances and labels for the query set are sampled.

Technically speaking, the task-specific model-copy and the query set are now stored until the batch of tasks has been processed. In Line 10, the test losses of the instances from the query set are created using the model-copy. These losses are summed up and the gradient is calculated. This gradient from the sum of test-losses of the model-copies now serves as a training error for the meta-model.

In the algorithm only one GD step is shown, but as in our implementation, it is possible to perform multiple gradient steps.

3.5 REGRESSION TASKS

In this section a few-shot learning scenario for a mathematical function regression will be described. The following use case is discussed: An application with an underlying mathematical function is investigated. The function has one or more parameters that are variable but has always the same shape. Since it is too expensive (takes too long or is too costly) to measure many samples, only a few examples of the function are available. A k-shot regressor shall be trained that is capable of regressing the underlying function fast by only having very few examples of it. Note, that this is a k-shot 1-way problem, since there is just 1 sinusoid per task to regress.

For this work sinusoids with variable amplitude and phase shift will be used for training. A classic supervised learning algorithm will fail, because there are too few examples to find an appropriate regression of a *single* sinusoid. Moreover this is a hard problem because sinusoids are periodic and with a phase shift there are a lot of outcomes y_i that overlap for the same x_i value.

Finally a single neural network trained on different sinusoid tasks will eventually return an averaged value over the distribution of tasks. The average over all possible amplitudes of sinusoids that are shifted in phase will be 0. So the neural network will not be able to learn this function at all. In the following it will be described, how MAML can be used to solve this kind of few-shot regression problems.

3.5.1 *Sampling and Preprocessing*

For the regression tasks sinusoids are used, that have a varying amplitude in the interval $[0.1, .., 5]$ and a varying phase shift in the interval $[0, .., \pi]$. Since there are infinite many different sinusoids, there is no need for a training and a validation set. For every training task, a new sinusoid will be generated and sampled. For validation it will be done the same way. The support set and query set are sampled from the same function.

Figure 3.6 shows examples for possible different sinusoids. The parameter $k_{support}$ is 5 in these cases. The examples are marked in the figure. We perform no further preprocessing on the samples.

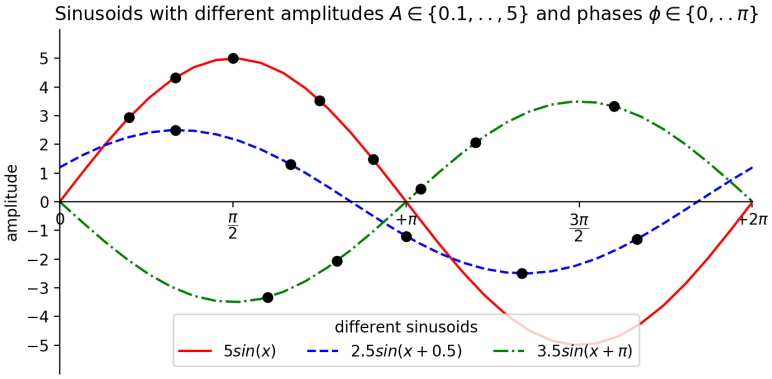


Figure 3.6: Sampling of sinusoids with varying amplitude and phase shift. Every graph has a total of $k_{support} = 5$ examples (black dots) of sample and true label pairs $\{x^{(i)}, y^{(i)}\}$.

3.5.2 *Training*

The training process is identical to the process described in Algorithm 2. The only difference is that in regression the output of the model is not a classification in the form of a label vector, but a predicted y-value.

3.6 REINFORCEMENT LEARNING TASKS

In this section a few-shot learning scenario for RL will be described. Few-shot meta-learning in the RL setting is used to enable the agent to quickly find a policy for a test task by only using a small amount of experience in the validation setting. Note that in this setting, we speak of a k-shot 1-way problem, since only one environment per task is used.

A new task might involve a *new goal* or a succeeding task in *another environment* [5]. For example an agent might learn to navigate through a maze. The goal of meta-learning is then to quickly learn to navigate through another maze or to follow another goal in the same maze, as e. g. finding another exit. In the following will be described how

MAML can be used to solve this kind of few-shot RL problems with some environments as examples.

This is different from the classic reinforcement learning scenario, since the same maze would always be used in reinforcement learning. In classical reinforcement learning an agent is only trained to solve *one* maze, but not to solve the problem class *mazes*.

3.6.1 *Sampling and Preprocessing*

Sampling in few-shot reinforcement learning works differently than in supervised setting. There is also a distribution of tasks $p(\mathcal{T})$, from which tasks are selected batchwise. The sampling of a task works as follows:

Since no data points are available in the reinforcement learning setting as in the supervised setting, the meta-model must be trained differently. In reinforcement learning an agent is trained by interacting with the environment and gaining experience. This experience is collected in the form of transition samples. A transition sample consists of an observation x_t at time t, an action y_t that the agent performs based on the observation and its policy π and an observation at the next time step x_{t+1} as well as a reward $R(x_t, y_t)$ as feedback signal of the environment.

In MAML k trajectories are collected, which contain the observations and actions as pairs $\{x_t, y_t\}$. The support set of a task $\mathcal{T}_i$ thus consists of k trajectories of $\mathcal{H}$ many $\{x_t, y_t\}$ pairs. Since $n = 1$, the support set only consists of the trajectories. The sum of the corresponding reward signals $R(x_t, y_t)$ are used in the loss function of the trajectory. An example of this process is illustrated in Figure 3.7

When enough samples have been sampled, training can begin. How the training proceeds exactly depends on the underlying reinforcement learning algorithm. A general description of the learning process is given in the next section.

The sampling for meta-learning follows after the training. As described in the previous sections, meta-learning takes place with the query set. However, since the underlying task in the meta-learning setting must not be changed, the trajectories for the query set must be obtained differently as changing the environment.

In the few-shot reinforcement meta-learning setting, a change of the task is not necessary, to produce different trajectories. The trajectories

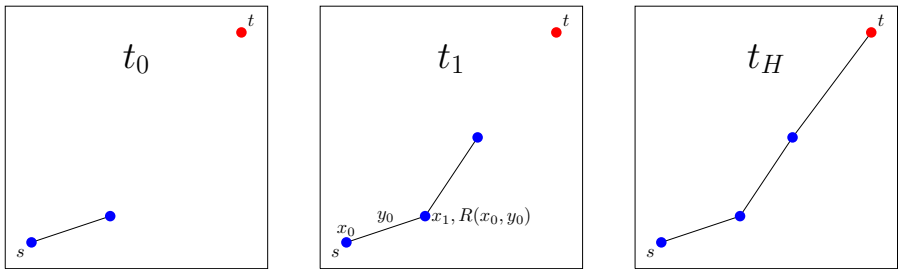


Figure 3.7: Sampling of a trajectory in a navigation environment: In this example is $k_{support} = 1$, which means that 1 trajectory is sampled for support set. A trajectory consists of $\mathcal{H} = 3$ transitions. In every transition a tuple $\{x_t, y_t\}$ with observation x_t and action y_t is recorded, as can be seen at t_1. For every transition tuple an imediate reward $R(x_t, y_t)$ is collected for the loss function. When the environment ends or $\mathcal{H}$ many samples are gathered, the trajectory is complete.

for the query set are generated in the same way as for the support set, except that the policy model is no longer based on the meta-model, but on the task-specific model-copy that has already been trained with $k_{support}$ trajectories. In this way, the task remains unchanged and keeps the same goal, but the observations and therefore the whole trajectories differ from those from the support set.

3.6.2 *Training*

The training procedure can be seen in Algorithm 3.

Algorithm 3 MAML for Reinforcement Learning by Finn et al. in [5]

Require: $p(\mathcal{T})$ distribution over tasks
Require: α, β step size hyperparameters
1: Randomly initialize θ
2: **while** not done **do**
3: Sample batch of task $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample k trajectories $\mathcal{D}^{tr}_{\mathcal{T}_i} = \{(x_1, y_1, ..., x_H)\}$ using f_θ in $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{L}_{\mathcal{T}_i}$ in Equation 5.4
7: Compute adapted parameters with gradient descent: $\theta'_i = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample trajectories $\mathcal{D}^{test}_{\mathcal{T}_i} = \{(x_1, y_1, ..., x_H)\}$ using $f_{\theta'_i}$ in $\mathcal{T}_i$
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta'_i})$ using each $\mathcal{D}^{test}_{\mathcal{T}_i}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 5.4
11: **end while**

The algorithm is identical to Algorithm 2 execpt for Lines 5 and 8, because the trajectores have to be sampled from the environment coresponding to the task $\mathcal{T}_i$, as described in the last section.

It should be mentioned that the support sets and query sets now consist of trajectories, i.e. they consist of lists of transitions. During training, the models are then trained with one trajectory each, since the reward is calculated over a complete trajectory (with horizon $\mathcal{H}$). The reward serves as loss function. This process is repeated for each task until all k trajectories have been processed.

3.6.3 *Environments*

In the following environments used with our framework are presented. Since the reinforcement learning part of our framework is still being tested, there are no experiments on this topic in this paper. However, suitable environments will be described here, because they will be used in further studies.

3.6.3.1 *OpenAI Gym*

All environments, that are used in this study follow the Gym interface[2]. OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. To build our gym environments we used the open-source library and built a reinforcement learning environment, that inerhits from the gym base class. By doing this, we implemented standardized functions like a constructor, a step function, which is performed every timestep, a reset function, that resets the environment to its inital state (e. g. when starting a new epoch) and a render function, that can be used to give text or visual feedback for a human observer. This makes it easier to compare and use different environments, since every environment can be handled the same way and adaptions of the code are avoided. Following this specifications, our framework possibly can deal with every reinforcement learning environment, that inherits from OpenAI Gym.

3.6.3.2 *Simple 2d Navigation Task*

For learning and testing purposes we developed a simple 2d navigation environment. The goal of the environment is to have a point agent

2 https://github.com/openai/gym

navigate to another point. The observation consists of the coordinates of the agent and the target. The agent also receives the distance to the target at each step. The agent can perform an action in each time step that contains of an angle and a step length. It should learn to reach the goal in as few steps as possible.

The environment is solved, if the agent can reach the goal in a minimal number of steps. In the worst scenario a perfectly learned policy needs $ceil(\sqrt{255^2 + 255^2}/10) = 37$ steps which gives a reward of 330. Thus, the environment is solved if a reward greater than 330 or a number of steps taken smaller than 38 steps respectively is regularly achieved. The environment is illustrated in Figure 3.8.

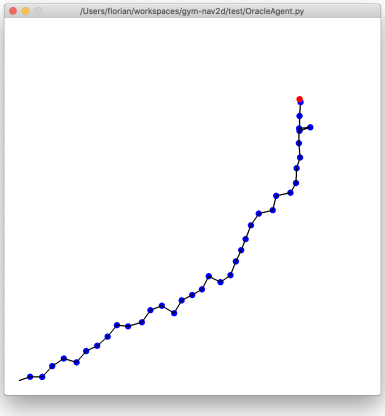


Figure 3.8: A simple 2d navigation task[3]. The position of the agent is tracked (blue dots) for every step and the way it took is drawn in black. For easy difficulty the agent's start position changes every instantiation, for hard difficulty the goal's position changes too. If seed is set, start and end point can be set constant.

As it can be seen in the figure there is a red point (the goal) and the agent (blue) illustrated. Every timestep the new position is marked by a blue dot. The dots are connected by a line, so a human observer can clearly see what actions the agent did in every timestep. The environment consists of a court which has an area of 255 x 255 units. The coordinates of the playing field are coded as float numbers, so that a very high resolution of the playing field is guaranteed.

Before the observation reaches the agent, it is normalized. The observation dimensions are shown in Table 3.2. The x and y positions of the agent and the target are normalized to a range between -1 and 1. The distance is normalized to a range between 0 and 1. There are practical reasons for this. Using different input scales can slow down the training and cause instability.

3 https://github.com/shufflebyte/gym-nav2d

Table 3.2: Observation Space of Nav2d Environment

entity	min	max
agent position	(-1, -1)	(+1, +1)
goal position	(-1, -1)	(+1, +1)
distance	0	1

Table 3.3 shows the observation and action spaces. As it can be seen, the action space is also normalized. Firstly, an angle of 0° to 360° can be set over a range of -1 to 1. Secondly a step size, from 0 to 10 units can be set in a range from -1 to 1.

Table 3.3: Action Space of Nav2d Environment

entity	min	max
(degree, step size)	Box(-1, +1)	Box(-1, +1)

With this simple environment, the sampling described in theory above will be explained again. As already stated, the task must not change when meta-learning is performed. To ensure this, a seed is defined in task-specific learning when creating the environment, which is saved together with the model. If meta-learning is then performed, the environment is initialized with the stored seed. Thus, the positions of the agent and the target are initialized in exactly the same way as in task-specific learning. However, the recorded samples differ from those recorded during task-specific training, because the model-copy has been trained since then and accordingly different actions are chosen by the policy model.

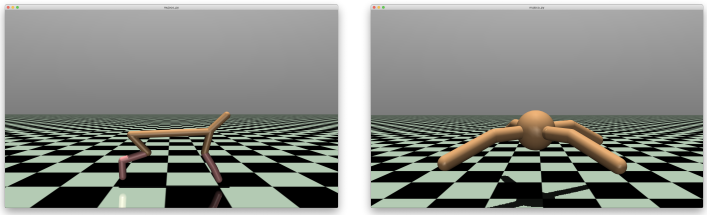
3.6.3.3 *Further Environments*

In the further course of this study we also planned experiments with the HalfCheetah and Ant Environment. These environments use MuJoCo[4] , a physics engine. With the help of MuJoCo, it is possible to model complex real world applications. We already started with the implementation of these environments . These environments and their goals are briefly described below.

HALFCHEETAH AND ANT ENVIRONMENTS. The HalfCheeta [5] (Figure 3.9a) is in comparison to the Nav2d environment a very compli-

4 http://mujoco.org/
5 https://github.com/shufflebyte/gym-halfcheetah

cated environment to solve. It consist of a halfed cheetah with only two legs and 3 joints at each leg. The action space consists of the joint torques. The HalfCheetah has a total of 6 joints (3 on the front leg and 3 on the back leg). The observation space consists of 20 different values. For the torso and for each joint the current angle and angular velocities are returned (14 values). In addition, the agent receives the position and velocity in x, y, z direction (6 values). Furthermore the reward signal is returned.



(a) HalfCheetah environment (b) Ant environment

Figure 3.9: MuJoCo environments used in [5]. Left: HalfCheetah environment; Right: Ant environment

In [5] two different experiments were defined. In the first experiment a target space of directions d was defined. In this environment the HalfCheetah can only run in two directions, therefore the goal space is $d = \{forward, backward\}$. In the second experiment a goal space of velocities v was defined. This can be defined arbitrarily, since $v \in \mathbb{R}$.

The Ant [6] (Figure 3.9b) follows the same rules as the HalfCheetah. It consists of an ant with 4 legs, each with 2 joints. The action space consists of the 8 joint torques. The observation space consists of 125 different values and is therefore much more complicated than the HalfCheetah.

METAWORLD ENVIRONMENTS. Previous studies on meta-learning have always defined only very narrow variations in goals. Changing positions in a navigation task or the target speed of a HalfCheetah are only very small changes and should rather be seen as toy examples or feasibility studies.

In [20] a framework called MetaWorld is proposed, which offers a meta-learning framework where different tasks from a broader range from the field of robotics can be investigated. In this environment a robot arm can be controlled to perform different tasks like pick and place an object or open a window. Different sized meta-learning training sets are offered with their own test tasks, which have never

6 https://github.com/shufflebyte/gym-ant

been seen or learned by the agent before. Figure 3.10 shows an example of a meta-learning task set of 10 tasks. There are 5 test tasks available for validation.

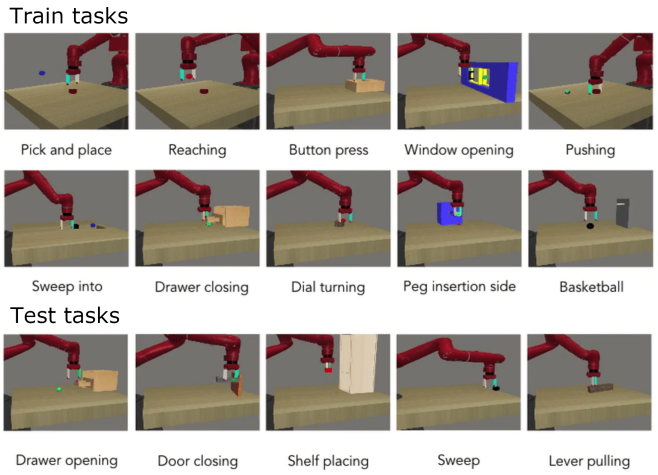


Figure 3.10: MetaWorld Meta-learning 10: A set of 10 different tasks with 5 test tasks. All test tasks have never been seen before by the agent[7]

As it can be seen from the figure, the presented tasks are clearly different from each other, so that one can actually speak of different tasks. Further studies about the MetaWorld framework are worthwhile.

7 https://meta-world.github.io

4

DISTRIBUTED MAML

In this section we describe how we came up with the idea for this study. The architecture of Distributed Model-Agnostic Meta-Learning (DMAML) is roughly outlined. The description of details about the implementation is omitted here so as not to go beyond the scope.

4.1 THE IDEA OF THIS PROJECT

In previous work, such as our student research project, we have already dealt with reinforcement learning. In [5] MAML was presented as an universal algorithm, which is also suitable for accelerating reinforcement learning. In this way we were able to combine two interesting areas, on the one hand meta-learning a new research topic for us and on the other hand reinforcement learning, which we have already investigated in previous studies.

The idea of parallelizing MAML comes from ideas from research in the field of reinforcement learning. In [6] the researchers were able to show that they could strongly resolve the learning of the action-value function in the Deep Q Network (DQN) Algorithm by massive parallelization (Ape-X).

In [9] the researchers have shown that they were able to accelerate the learning of a policy by parallelization (A3C). We wanted to adopt these ideas in our study and test whether they can be transferred to other areas such as regression or classification.

4.2 OUR CONTRIBUTION

In this thesis we have built a general framework for MAML, which can solve regression, classification and reinforcement learning tasks. Such a framework was not yet available - only seperate repositories that deal with eather few-shot supervised learning or reinforcement learning. As with [5] a sinusoid generator for regression tasks, Omniglot and

MiniImageNet for classification tasks were integrated and gym-Nav2d, a RL environment for reinforcement learning, was developed.

To add further models or data sets, only very minor changes to the source code are necessary. Everything, except the underlying reinforcement learning framework[1], has been developed by us. The framework can be run in the current Python version (Python 3.7) and was developed with TensorFlow 2.1.

The present software can be operated like the classic MAML and can also be configured with a large set of parameters without having to change the source code.

Furthermore, it is possible to parallelize the MAML algorithm, which is the subject of this thesis. We also ran a series of parallelization tests, which are described in the section experiments.

4.3 THE DISTRIBUTED MAML ALGORITHM

In the following the DMAML algorithm is presented. The algorithm is an encapsulation of the MAML algorithm. To make it easier to understand, a basic form is presented first. Then the already implemented extension with weight generators is presented.

BASIC DMAML ALGORITHM

DMAML extends the classic MAML algorithm with the parallelization component (Algorithm 4).

Algorithm 4 Distributed Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$ distribution over tasks
Require: α, β learning rate hyperparameters
Require: number of runs r, workers w
1: Randomly initialize DMAML meta-weights $\theta_{i,DMAML}$
2: **for all** r_i **do**
3: **for all** w_j **do**
4: Create MAML worker with $\theta_{i,DMAML}$
5: Run MAML and collect $\theta_{i+1,MAML,w_j}$
6: **end for**
7: $\theta_{i+1,DMAML} \leftarrow \frac{1}{w} \sum_{w_j} \theta_{i+1,MAML,w_j}$
8: **end for**

1 Copyright (c) 2019 Kei Ohta https://github.com/keiohta/tf2rl

As it can be seen, the MAML Algorithm is encapsulated in Lines 4 and 5, the Algorithm requires α, β as learning rate hyperparameters. Additionally, the number of runs and workers is required.

At the beginning the meta-weights are initialized randomly (Line 1). Then the following is repeated every run for the number of runs: The (new) meta-weights are distributed to the workers and the MAML algorithm is executed (Line 3 and 4). The new meta-weights resulting from the MAML workers are stored in a list (Line 5).

After all workers have performed their MAML Algorithm, the new common meta-weights are calculated. To obtain the new meta-weights, the arithmetic mean of the meta-weights of all workers is calculated.

WEIGHT GENERATOR EXTENSION While debugging the reinforcement learning experiments, we noticed that policies that were already well trained and produced high rewards suddenly performed poorly again.

During an investigation we noticed that a few trajectories were responsible for this, which destroyed the complete learning success of the agent. This led us to the idea of not considering such meta-weights that were delivering poor performance in the DMAML update (Line 7).

The performance is evaluated on the basis of the loss. For this purpose, the loss sums of the last 5 epochs are also stored for each MAML worker. The algorithm is extended by a weight generator. A weight generator is basically a discriminator which compares the average of the last loss sums of the individual workers and excludes individual workers from the calculation of the new meta-weights according to a certain criterion.

We have implemented some weight generators. We suspect that the weight generators may cause a performance difference in the reinforcement learning area. Since the reinforcement learning part is currently still in the test phase, no experiments are available yet. This should be investigated in further studies. The extended version of DMAML can be seen in Algorithm 5.

Algorithm 5 Distributed Model-Agnostic Meta-Learning with Weight Generators

Require: $p(\mathcal{T})$ distribution over tasks
Require: α, β learning rate hyperparameters
Require: number of runs r, workers w, and weight generator $\mathcal{G}$
1: Randomly initialize DMAML meta-weights $\theta_{i,DMAML}$
2: **for all** r_i **do**
3: **for all** w_j **do**
4: Create MAML worker with $\theta_{i,DMAML}$
5: Run MAML and collect $\theta_{i+1,MAML,w_j}$ and $\{\mathcal{L}_{\mathcal{T}_i,MAML,w_j}\}$
6: **end for**
7: $\theta_{i+1,DMAML} \leftarrow \mathcal{G}(\{\theta_{i+1,MAML,w_j}, \{\mathcal{L}_{\mathcal{T}_i,MAML,w_j}\}\})$
8: **end for**

As it can be seen, a weight generator $\mathcal{G}$ is required and a batch of loss sums is collected for each worker. At the end of the run, the arithmetic mean is replaced by a generator function, that takes the worker meta-weights and their batches of loss sums as parameters. The function generates the new meta-weights.

SCHEMATIC VIEWPOINT To further support the understanding, a schematic view of the DMAML Algorithm is given. Figure 4.1 illustrates a single run of the DMAML algorithm.

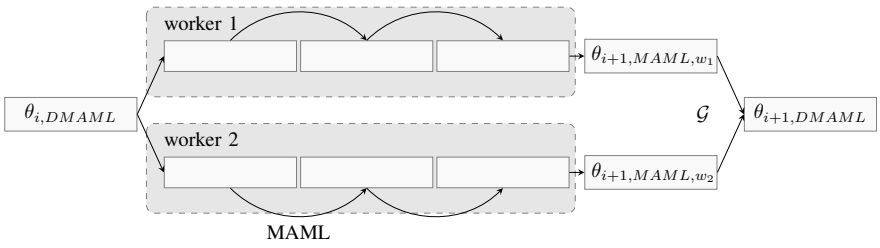


Figure 4.1: Schematic view of a DMAML run

As the figure shows, in the beginning of the i-th run the meta-model $\theta_{i,DMAML}$ is distributed to the *workers*. Then every worker runs for a number of *epochs*. In every epoch a MAML-step is done and a worker-side meta-model is improved. After a certain number of epochs the meta-model of each worker is handed over to the weight generator $\mathcal{G}$. The weight generator decides if the meta-model of a worker is used for the update of the common meta-model. The output of the weight generator is the new meta-model $\theta_{i+1,DMAML}$

WEIGHT GENERATORS. In the following, the implemented weight generators are presented. The framework can be easily extended.

The **plain weight generator** takes the weights of the meta-models and calculates the arithmetic mean. The losses and accuracies remain untouched. This is the basic merging strategy from Algorithm 4.

The **loss median weight generator** determines the median loss of the transferred losses and discards the meta-weights of the workers, that perform worse than the median performance. The remaining weights are then arithmetically averaged again. Thus, the worse workers are not considered.

The **weighted weight generator** merges the weights of the meta-models first like the plain generator. Then the newly calculated weights are multiplied by a weighting factor g (from 0 to 1) and the old weights are multiplied by (1 - g) and added up again. The idea behind this method is not to update the meta-model too much in one run and to avoid overfitting.

The **loss moving average weight generator** calculates a rolling average of the losses. Only those meta-model weights are taken over whose worker losses are below the rolling average. The remaining weights are then arithmetically averaged again. The idea behind this weight generator is to filter out those weights that do not belong to the current trend and are more likely to be considered outliers.

Part II

EXPERIMENTS

5

GENERAL SETUP OF EXPERIMENTS

In the following hardware and software are presented, which we used for all experiments. In addition, general test conditions and settings are presented.

5.1 USED HARDWARE AND SOFTWARE

For all experiments we used two identical servers of TH Köln with a Ubuntu 18.04 Linux 64-bit installed. Each server has got a memory of 256 GB RAM, Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz with 56 cores and two NVIDIA TITAN X (Pascal) GPUs with both 12 GB memory.

We used Python 3.7 and TensorFlow 2.1 for implementation. More details on Python packages can be reviewed on the enclosed CD. For experiment administration we used Neptune and for generating the plots of this work we used matplotlib and seaborn Python packages. For the reinforcement learning part we used the tf2rl framework by Kei Ohta[1].

5.2 CONFIDENCE INTERVALS

In our plots we carry out a confidence estimation with a confidence level. For this we assume a Gaussian distribution. This is an indication of the confidence interval $I = [t_1, t_2]$ determined from the sample, in which the true, unknown parameter Θ of the distribution of the population is to be expected with a high (previously determined, arbitrarily chosen) probability. This parameter corresponds to the mean value μ in our plots [3].

For each plot we used a sample size of $n = 3$ experiments, and a confidence level of $\gamma = 95\%$. This means that our calculated confidence interval is, with a probability of 95%, one of the infinitely many

1 https://github.com/keiohta/tf2rl

intervals that actually contains the desired expected value. We call it short ci95 for the rest of this study.

Put simply (but formally wrong), the searched-for expectation value lies within the calculated interval with a probability of 95%. We had the confidence intervals calculated and plotted by the plot framework seaborn.

The underlying values for the calculation are as follows: Sample size n = 3 experiments and confidence level $\gamma = 0.95$. The z value can be read from a t-distribution, e.g. from [3], for $n = 3$ with $\gamma = 0.95$ follows $z(\gamma, n) = 2.353$ The formula for the confidence interval can be seen in Equation 5.1.

$$t_1, t_2 = [\bar{x} \pm \frac{z(\gamma, n) \cdot \gamma}{\sqrt{n}}] \tag{5.1}$$

In our figures we have plotted the confidence interval, but the line plots do not correspond to the mean value. Instead, they have been approximated with a polynomyial curve fitting function of degree 10. We decided to do so because the mean values also fluctuate and, in the case of graphs with several curves, it is difficult to identify which configuration performs better. With these trend lines reality is abstracted and the plotted values deviate slightly from the true values, but the evaluation is clearer and easier.

5.3 EXPERIMENT DESIGN

Unless otherwise described, all experiments were performed three times and plotted with ci95. Different loss functions were used for the different domains. In all experiments the meta-model was initialized randomly. All experiments use Adam as meta-optimzier and first-order derivations (FOMAML).

All hyperparameters follow the specifications of [5] with the exception of the meta-learning rate, which was adapted domain-specifically by hyperparameter searches. Furthermore, in most experiments the number of epochs has been set to 1. The used hyperparameters can be seen in Tables 5.3, 5.1, 5.2.

For regression experiments, we used a sinusoid generator, that generates $k = 10$ samples of sine waves, that have varying amplitudes

between 0.1 and 5.0 and varying phases between 0 and π. The sampling has been performed between -5 and 5.

Table 5.1: Omniglot Classification Experiment Parameters.

Experiment parameters	Symbol	Value
Number of support samples	$k_{support}$	1
Number of query samples	k_{query}	1
Inner-learning rate	α	0.4
Meta-learning rate	β	0.01
Number of tasks	n	32
Number of epochs		1
Number of gradient steps		1
Number of validation steps		3
Number of validation tasks		10
Number of validation samples		1

Table 5.2: MiniImageNet Classification Experiment Parameters.

Experiment parameters	Symbol	Value
Number of support samples	$k_{support}$	1
Number of query samples	k_{query}	15
Inner-learning rate	α	0.4
Meta-learning rate	β	0.001
Number of tasks	n	4
Number of epochs		1
Number of gradient steps		5
Number of validation steps		10
Number of validation tasks		10
Number of validation samples		1

Table 5.3: Sinusoid Regression Experiment Parameters.

Experiment parameters	Symbol	Value
Number of support samples	$k_{support}$	10
Number of query samples	k_{query}	10
Inner-learning rate	α	0.01
Meta-learning rate	β	0.01
Number of tasks	n	10
Number of epochs		1
Number of gradient steps		1
Number of validation steps		1
Number of validation tasks		10
Number of validation samples		10

5.4 USED MODELS

In all experiments we used the same set of model architecture, which has been proposed by [5].

OMNIGLOT MODEL. For Omniglot classification, we used a model with 4 batches of convolutional layers with 64 filters and kernel 3x3 with padding same and strides = 2, followed by BatchNormalization Layer and ReLU nonlinearities. After the fourth ReLU layer, the neurons have been flattened and feeded into an output layer with softmax activation.

MINIIMAGENET MODEL. For MiniImageNet classification, we used a model with 4 batches of convolutional layers with 32 filters and kernel 3x3 with padding same followed by BatchNormalization Layer and ReLU nonlinearities. After that a MaxPooling Layer with a pool size of 2x2 is followed. The last layer is a flattening Layer and feeded into an output layer with softmax activation.

The size of the input layer varies, because the image size is different between Omniglot and MinimageNet. The output of the model is a n-way classification prediction vector with probabilities $\hat{y}^{(i)}$ from softmax activation.

REGRESSION MODEL. We used a neural network with an input layer of size 1, 2 hidden layers with size 40 followed by ReLU nonlinearities and an output layer of size 1. The output corresponds to the y-value of the function we are trying to approximate.

5.5 LOSS FUNCTIONS

The loss functions were chosen according to the domain according to [5] and are briefly presented in the following.

REGRESSION. For the regression experiments the Mean Squared Error (MSE) was used (Equation 5.2).

$$\mathcal{L}_{\mathcal{T}_i}(f_\theta) = \sum_{\hat{y}^{(j)}, y^{(j)} \sim \mathcal{T}_i} ||\hat{y}^{(j)} - y^{(j)}||_2^2 \tag{5.2}$$

CLASSIFICATION. Since we are doing a multi-class classification (that means one sample can only be assigned to one class and has therefore only one label) we are using categorical cross-entropy as loss function, as described in Equation 5.3.

$$\mathcal{L}_{\mathcal{T}_i}(f_\theta) = -\sum_{i=1}^{C} y^{(i)} \log(\hat{y}^{(i)}) \tag{5.3}$$

Where $y^{(i)}$ represents one element in the true label vector and $\hat{y}^{(i)}$ represents one element in the softmax output vector of the model.

REINFORCEMENT LEARNING. In the reinforcement learning case, the success of the algorithm is dependent on the reward. The loss function can be seen in Equation 5.4.

$$\mathcal{L}_{\mathcal{T}_i}(f_\theta) = -\mathbb{E}_{x_t, y_t \sim f_\theta, \rho_{\mathcal{T}_i}} \left[\sum_{t=1}^{H} R_i(x_t, y_t) \right] \tag{5.4}$$

This loss function is replaced by the loss function of the underlying RL (e.g. Proximal Policy Optimization (PPO) or Trust Region Policy Optimization (TRPO)) algorithm.

5.6 THE VALIDATION ACCURACY AND LOSS

In the following experiments the quantitiy validation accuracy (or loss) is used. This is explained and broken down in the following. Since the average losses and accuracies are equally calculated, in the following only the term loss is used, but the explanation is valid for both quantities.

The *validation loss* is calculated at the end of each DMAML run, after the new meta-model has been generated by the weight generator. The calculation is illustrated in Figure 5.1. Since we mathematically use only one worker for validation, the y-axis shows not the workers but the validation gradient steps. The x-axis shows the losses of the validation tasks. In the i-th row we can read the losses of the i-th gradient step. By averaging a row we calculate the validation loss for the i-th gradient step. This quantity is suitable for comparison along different numbers of workers.

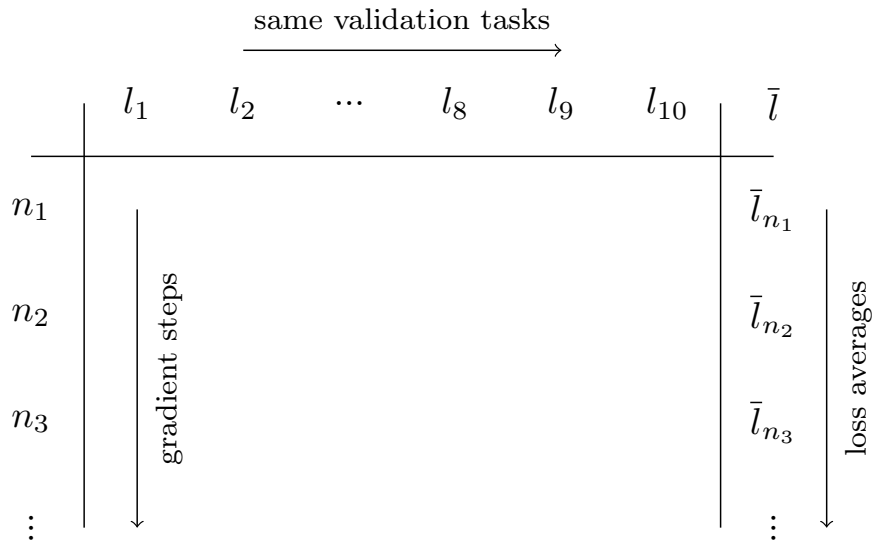


Figure 5.1: Calculation of the validation loss: The validation loss is calculated row-wise. In a row all validation task losses of a validation run are stored. We validate over a number of validation tasks (here 10). The average is calculated and results in the validation loss. We validate over a number of gradient steps, which can be parameterized beforehand.

6

PARALLELIZATION EXPERIMENTS

In this chapter a series of experiments is presented, which deal with the effects of different parameter values on the accuracy. Special attention is given to parallelization, since this study deals with the question whether parallelization of MAML can increase performance.

6.1 FIRST IMPLEMENTATION TEST EXPERIMENT

To see if our framework delivers comparable results to Finn et al., we did an informal test run after the implementation phase for classification and regression. The results are described below.

OMNIGLOT. For Omniglot classification we used the parameters shown in Table 5.1, except for meta-learning rate β which is 0.001 in this experiment. We run the code from [5] and ours for 1 time and plotted the results in Figure 6.1.

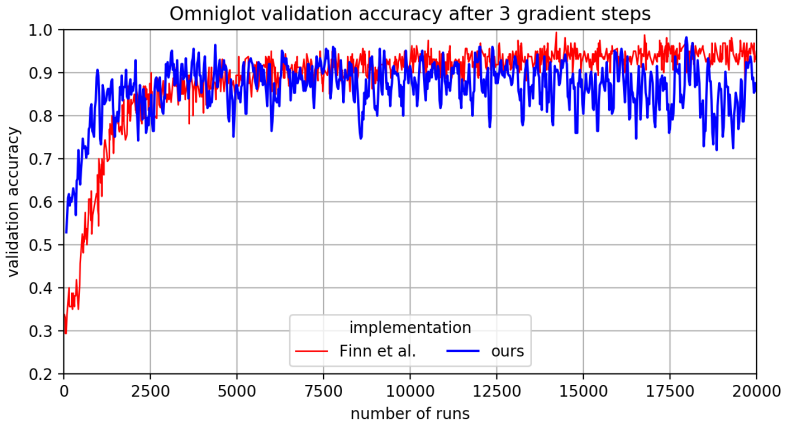


Figure 6.1: Omniglot implementation test validation accuracy after 3 gradient steps.

As it can be seen, the two graphs show similar forms. Our implementation seem to oscilate a bit more, which could be caused by different parameterization or implementation of TensorFlow components. It can

also be observed, that in our implementation the accuracy slowly decreases with increasing number of epochs after it reached its maximum accuracy.

MINIIMAGENET. We also run a test on MiniImageNet. Therefore we used the parameters from [5] (Table 5.2, except for $\beta = 0.001$).

The resulting accuracy graph, after 5 gradient steps can be seen in Figure 6.2. A similar behavior of the graphs can be observed. It becomes apparent, that our implementation seems to perform a little worse than the original implementation.

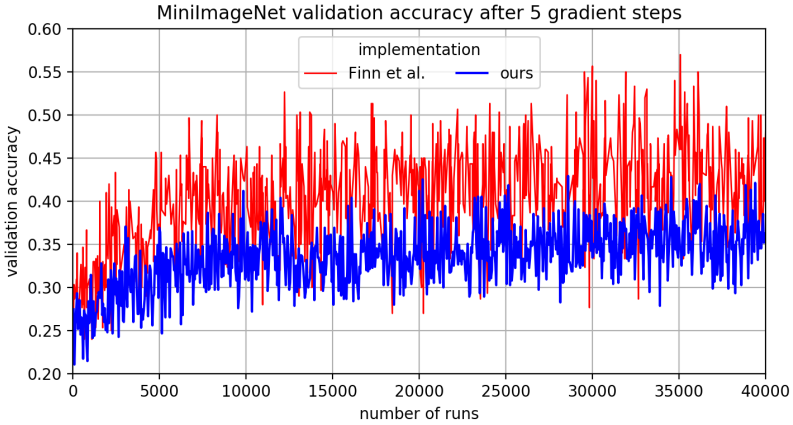


Figure 6.2: MiniImageNet implementation test validation accuracy after 5 gradient steps.

REGRESSION. The setup of the experiment can be seen in Table 5.3, except for meta-learning rate β which is 0.001 here. The result of this experiment is shown in Figure 6.3. It can be clearly seen, that our implementation fits pretty close to their implementation.

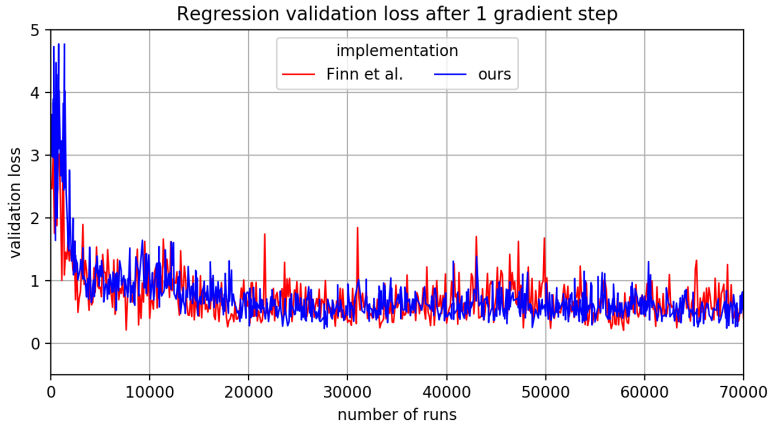


Figure 6.3: Regression implementation validation loss after one gradient step.

6.2 FIRST PARALLELIZATION EXPERIMENT

In this experiment we varyied the number of workers from 1 to 30 with the hyperparameters from [5]. The purpose of this experiment is to examine, whether just increasing the number of workers will result in a faster convergence of the average accuracy. The validation accuracy can be seen in Figure 6.4.

Figure 6.4: Omniglot validation accuracy with different number of workers. Parameters of [5] have been used.

It can be clearly observed, that simply increasing the number of workers with parameters of [5] does not lead to an significant acceleration of training. From this it follows that the parameters of [5] are not suitable for increasing the number of workers.

6.3 β-HYPERPARAMETER SEARCHES

Since the first parallelization experiment yielded such poor results, we decided to perform hyperparameter searches for the meta-learning rate β. We chose β as subject for the search, because for 1 worker the MAML algorithm already produced good results in the verification experiments and for Omniglot there was already a very high inner-learning rate of 0.4. We varyied β in the interval $\{0.001, 0.005, 0.01, 0.1\}$.

OMNIGLOT. The validation accuracy for Omniglot dataset after 3 GD steps for 1 worker can be seen in Figure 6.5.

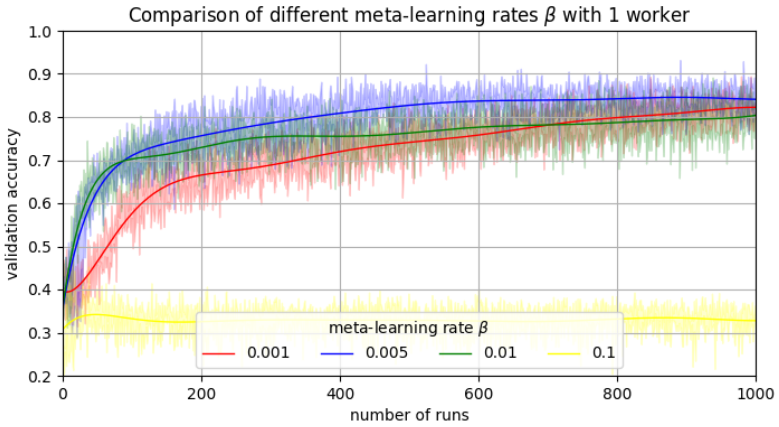


Figure 6.5: Omniglot validation accuracy with different values for meta-learning rate β after 3 validation steps and 1 worker

It can be observed, that meta-learning rates $\beta = 0.005$ and $\beta = 0.01$ perform better, especially in the first 400 runs, already for one worker. That means, that the meta-model is capable of learning faster than with the lower meta-learning rate of $\beta = 0.001$. Furthermore, we observed that we cannot increase β arbitrarily, because the experiment with $\beta = 0.1$ gets stuck at an validation accuracy of 0.35.

Since we found out, that a meta-learning rate for 1 worker of $\beta = 0.005$ or $\beta = 0.01$ is more suitable for our experiments, we also investigated, if other meta-learning rates are more suitable for a higher number of workers. Therefore we repeated the experiment for 10 workers. The results are shown in Figure 6.6.

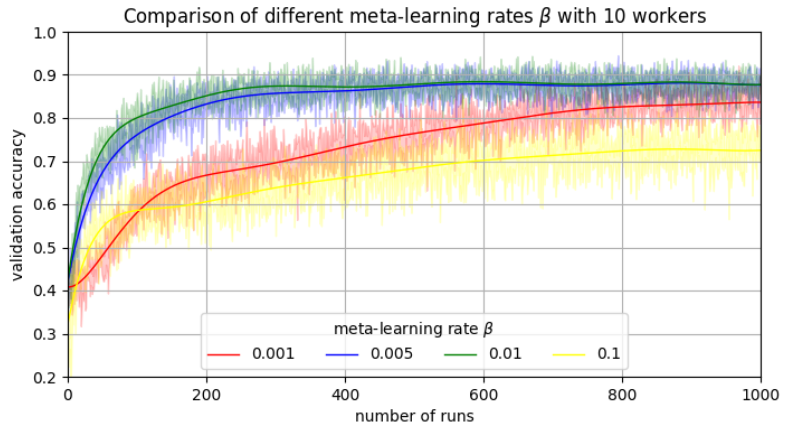


Figure 6.6: Omniglot validation accuracy with different values for meta-learning rate β after 3 validation steps and 10 workers

As it can be seen, also for 10 workers better results can be achieved by changing β. Since a meta-learning rate of $\beta = 0.01$ delivered slightly better results in the experiments with 10 workers, we chose it as the value for further experiments.

A further insight can be gained from these experiments. Comparing the validation accuracy of the experiment with one worker and 10 workers, it is noticeable that even the experiments with a not ideal β value ($\beta = 0.1$) increase faster by increasing the number of workers.

MINIIMAGENET. Inspired by the β-hyperparameter search for Omniglot, we also conducted a hyperparameter search for MiniImageNet. The validation accuracy after 10 GD steps can be seen in Figure 6.7.

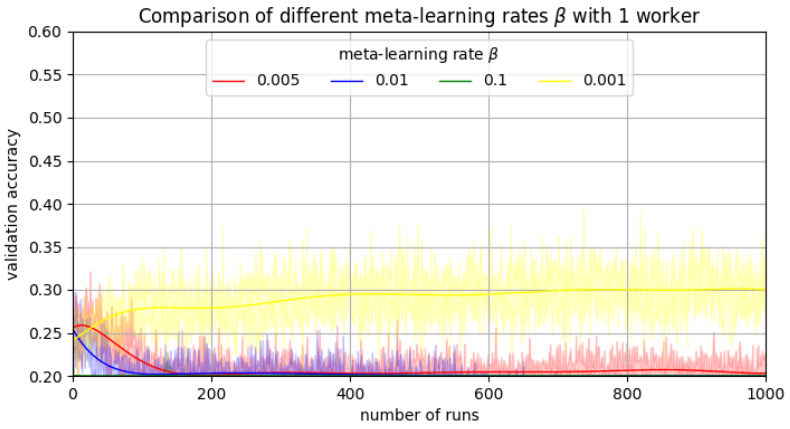


Figure 6.7: MiniImageNet validation accuracy with different values for meta-learning rate β after 10 validation steps for 1 worker

It can be clearly seen, that a meta-learning rate of $\beta = 0.001$ is the only meta-learning rate, that seems to result in a validation accuracy greater than 0.2 (which is no better than coincidence). This could be the reason why in [5] the learning rate 0.001 was chosen.

Also for MiniImageNet we wanted to investigate whether a different meta-learning rate might have better effects for a higher number of workers. For this purpose, we again recorded the validation accuracy for 10 workers, which can be seen in Figure 6.8.

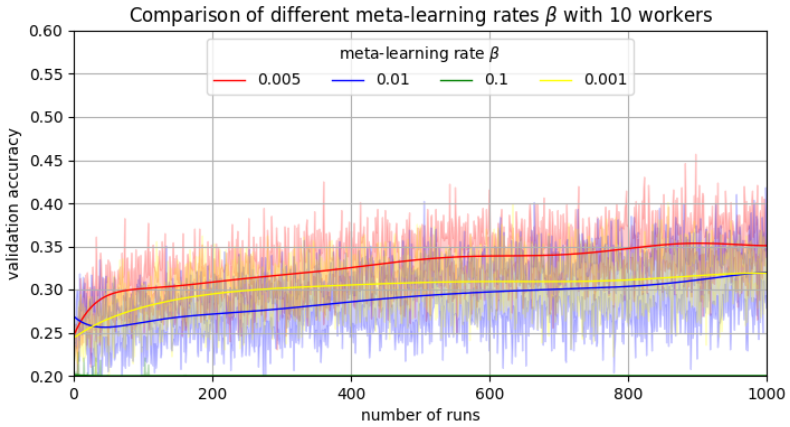


Figure 6.8: MiniImageNet validation accuracy with different values for meta-learning rate β after 10 validation steps for 10 workers

It can be seen that the meta-learning rate of $\beta = 0.005$ causes a stronger increase in validation accuracy over time.

In summary, for the purposes of this study a meta-learning rate of $\beta = 0.005$ is preferable because it gives better results with a higher number of workers. Therefore, this meta-learning rate is used in the further experiments.

REGRESSION. Since we have already discovered in the classification experiments that a hyperparameter search is worthwhile, we have also performed a corresponding search for the regression experiments. Again, we performed this search in a range of $[0.001, 0.1]$. The validation loss for 1 worker is shown in Figure 6.9.

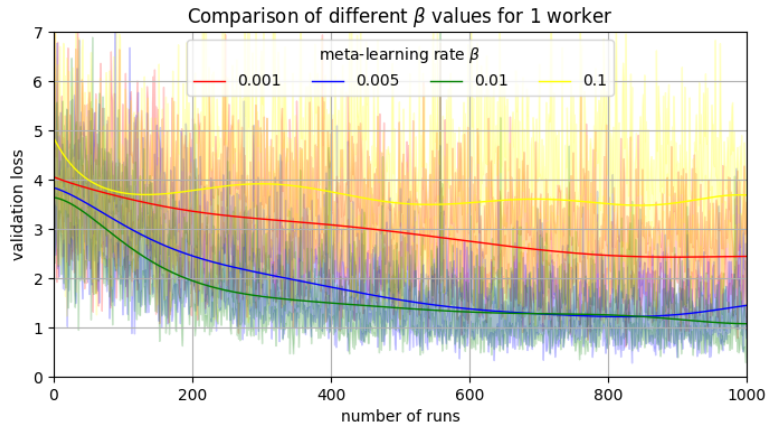


Figure 6.9: Sinusoid validation loss with different values for meta-learning rate β after 1 validation step for 1 worker

As it can be seen, the validation loss indicates that the meta-learning rate $\beta = 0.01$ and $\beta = 0.005$ causes the fastest loss reduction over time. To investigate the influence of the meta-lerning rate on a larger number of workers, we repeated the experiment for 10 workers, which can be seen in Figure 6.10.

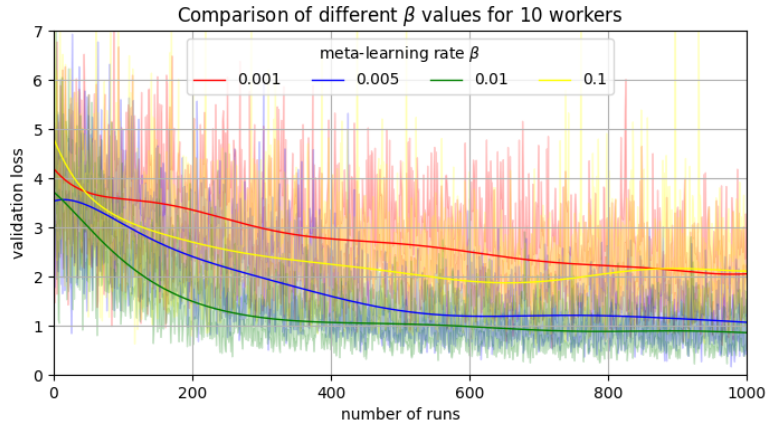


Figure 6.10: Sinusoid validation loss with different values for meta-learning rate β after 1 validation step for 10 workers

Again it turns out that a $\beta = 0.01$ is the most suitable for many workers. Especially in the first 400 runs, the loss decreases significantly faster than with the other meta-learning rates. In summary, the meta-leraning rate $\beta = 0.001$ is the most appropriate and is therefore used in the following experiments.

In summary, we found that the hyperparameters from [5] are not suitable for increasing the number of workers. Through the searches

we found new meta-learning rates for the different data sets, which are summarized in Table 6.1.

Table 6.1: Values for Meta-Learning Rate β for Different Datasets

Data set	**Value**
Omniglot	0.01
MiniImageNet	0.005
Sinusoids	0.01

6.4 PARALLELIZATION EXPERIMENTS

In the folling experiments we try to investigate, whether an increase of parallel workers leads to a faster convergence of accuracy.

OMNIGLOT. We used the parameters from Table 5.1 and varyied the number of workers in the interval $[1, 5, 10, 30, 50]$. The validation accuracies are shown in Figure 6.11.

Figure 6.11: Omniglot validation accuracy after 3 validation steps with small number of workers

As it can be observed, an increase of the number of workers clearly leads to a significant increase of the convergence of accuracy. It can be observed, that the baseline curve with only one worker (red) only increases very slowly after the first 200 runs. It catches up to the other graphs only beyond 1000 runs. Moreover it can be seen, that especially in the first 200 runs, the graph with 30 workers accelerates a bit faster than the graph with 5 workers. This suggests that an increase in the

number of workers will result in a faster increase in accuracy. Looking at the graphs, it also seems likely that the increase in accuracy does not increase in linear proportion to the number of workers, however. To substantiate this thesis, we also conducted experiments with 10, 30 and 50 workers. The results can be seen in Figure 6.12.

Figure 6.12: Omniglot validation accuracy after 3 validation steps with high number of workers

The graphs clearly show that the effect is less pronounced with increasing numbers of workers. Already with an increase from 10 to 30 workers the effect is only slightly noticeable, with an increase from 30 to 50 workers the effect is no longer graphically recognizable. Therefore, increasing the number of workers beyond 30 does not seem to have a meaningful effect. However, as stated before, the effect of increasing the number of workers from one to 5 or 10 is clearly visible.

MINIIMAGENET. Figure 6.13 shows the validation accuracy for 1, 5 and 30 workers. It is clear to see that increasing the number of workers leads to a much faster increase in accuracy. The effect is particularly strong when the number of workers is increased from 1 to 5. As in the Omniglot experiment, the faster increase is not linearly proportional to the increase in the number of workers. Hence, when the number of workers increases from 5 to 30, only a small increase in accuracy over time is observed.

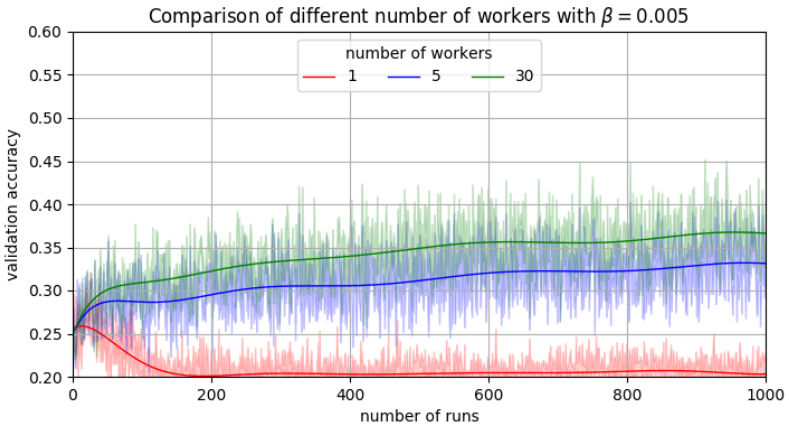


Figure 6.13: MiniImageNet validation accuracy with different number of workers

As in the previous investigation, it can be concluded, that the increase of a low number of workers has a huge effect on the increase in accuracy over time, but with higher numbers of workers this effect is less pronounced. We therefore did not measure a graph for 50 workers.

REGRESSION. We also performed parallelization experiments for the regression. Since it was already mentioned in [5] that MAML produces relatively worse results for regression than for classification, we hoped that parallelization would lead to a more significant improvement in results.

Figure 6.14 shows the validation loss for 1, 5 and 30 workers. As it can be seen, as with the classification experiments, an effect of parallelization can be perceived. At about 300 runs, a distance between the loss values of the experiments with one worker and 5 workers of about 0.5 can be seen, which is about 25% less loss at this point. In the further course of the graph, the graph with only one worker catches up to the graph with 5 workers.

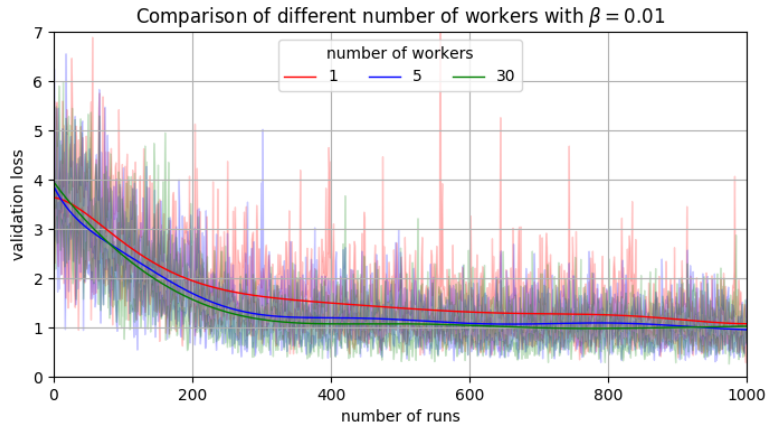


Figure 6.14: Sinusoid validation accuracy after 1 validation step with different number of workers

However, as in the previous experiments, the effect does not increase significantly when the number of workers is increased from 5 to 30. In conclusion, it can be said that the effect of the parallelization is noticeable, but less than hoped for.

6.5 INFLUENCE OF THE NUMBER OF GRADIENT DESCENT STEPS

To investigate the influence of different numbers of gradient steps, we also performed an experiment. We investigated the influence of using 1, 2 and 3 gradient steps and observed the validation accuracy. The results for 1 worker can be seen in Figure 6.15

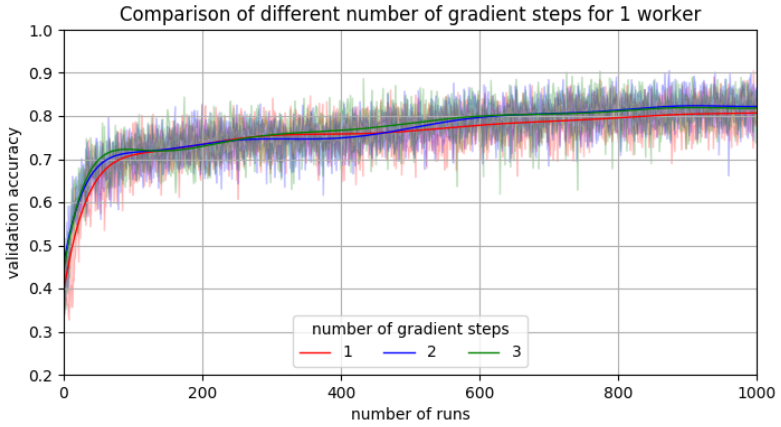


Figure 6.15: Omniglot validation accuracy with different number of gradient descent steps for 1 worker

The results confirm the statements made by [5] that an increase in the number of gradient steps does not result in large performance gains. We conclude that it is the direction of the gradients that is decisive and not the execution of several gradient steps.

To ensure that for a higher number of workers the increase in the number of gradient steps does not have significant positive effects, we recorded the validation accuracy for 10 workers, which is shown in Figure 6.16. As it can be seen, the effect is also hardly or not at all noticeable here.

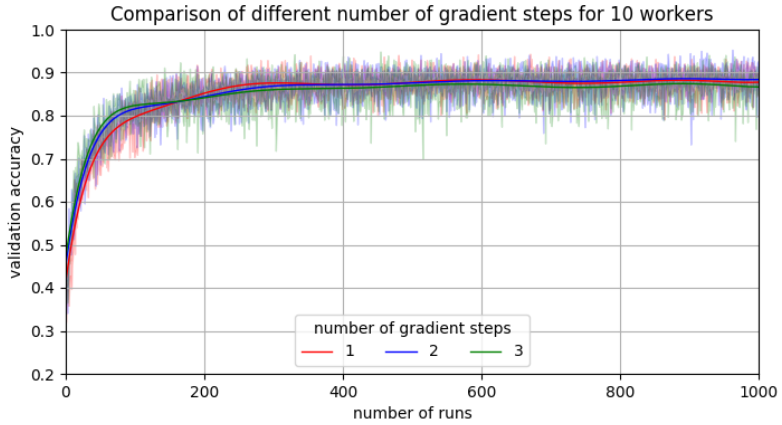


Figure 6.16: Omniglot validation accuracy with different number of gradient descent steps for 10 workers

From this we conclude that regardless of the number of workers, an increase in the number of gradient steps in the training process has no significant effect on the result. Therefore all upcoming experiments will only be executed with the number proposed by [5] and not increased beyond.

6.6 INFLUENCE OF THE PARAMETERS WORKERS, EPOCHS AND TASKS

Our implementation of the framework makes it possible to manipulate the parameters number of tasks, number of epochs and number of workers. The following series of experiments will investigate the influence of the individual parameters on the performance of the meta-model. Since the parameters influence the training of the meta-model at different points, this investigation is very interesting.

The number of *tasks* influences how many model-copies are created, whose test losses are included in the *gradient sum* used to train the

meta-model. The number of *epochs* affects the number of *repetitions* of these meta-model trainings. This parameter therefore influences the level above. The number of *workers* influences how many of these MAML trainings run in *parallel* and how many of their *weights* are then merged into a common meta-model with help of the weight generators.

The performance will be evaluated under two aspects. First, the increase in accuracy in relation to the number of runs required. The aim is to investigate whether the increase of the respective parameters changes the slope of the accuracy. The time required is not included in this analysis.

Second: The increase of the accuracy increase related to the wall time. Here it is examined how the increase of the individual parameters affects the accuracy in relation to the wall time.

The parameters are increased individually and the effect is examined. Then the results are compared and evaluated under both aspects.

The baseline for all experiments is the MAML algorithm of [5] with the parameters tasks = 32, epochs = 1, workers = 1, which has been performed by our implementation to ensure a fair comparison. In the following experiments, the parameters are increased in the same ratio (scaling ratio). For example, when increasing the number of epochs, the number of epochs 1, 2, 4, 8, 16 corresponds exactly to the scaling ratio. For the parameter number of tasks, analogously the values 32, 64, 128, 256, 512 correspond to the factors 1, 2, 4, 8, 16. This scaling was chosen in order to make the experiments comparable under the above mentioned aspects. Table 6.2 shows an overview of the concrete values of the parameters in the scaling ratio.

Table 6.2: Values of the Parameters With Different Scaling Ratios

scaling ratio	**1**	**2**	**4**	**8**	**16**
number of workers	1	2	4	8	16
nuber of epochs	1	2	4	8	16
number of tasks	32	64	128	256	512
seen tasks per run	32	64	128	256	512

The idea behind this approach is that the increase in accuracy per run depends on two factors. The first factor is the number of tasks seen. In a machine learning system, if the number of training examples is increased, the accuracy will be increased. The second factor is

that the parameter's influence of the different meta-optimizations (namely gradient based on the MAML-level optimization, weight based on the DMAML-level optimization and the number of repetitions of these) differs. We suspect that the different types of optimization work differently well. In order to investigate the second factor, we chose the uniform scaling (from 1 to 16). At each scaling ratio, each experiment saw the same number of tasks per run, no matter which parameter was changed.

INFLUENCE OF THE NUMBER OF WORKERS. This experiment is basically a repetition of a previous experiment and is performed for completeness. In this experiment, however, the number of workers was recorded equidistantly. The results are shown in Figure 6.17.

Figure 6.17: Comparison of the validation accuracy after 3 gradient steps with different number of workers.

As the figure clearly shows, the increase in accuracy is not linearly proportional and decreases with increasing number of workers. A significant increase can be seen with 1 to 2 workers and with 2 to 4 workers. After that, the benefits from increasing the number of workers are significantly lower. This makes it clear once again that it only makes sense to parallelize to a certain extent.

INFLUENCE OF THE NUMBER OF EPOCHS. In this experiment it shall be examined whether increasing the number of epochs with only one worker, in principle increases the increase of the accuracy per run. Therefore experiments with 1 to 16 epochs per run were carried out. The result can be seen in Figure 6.18.

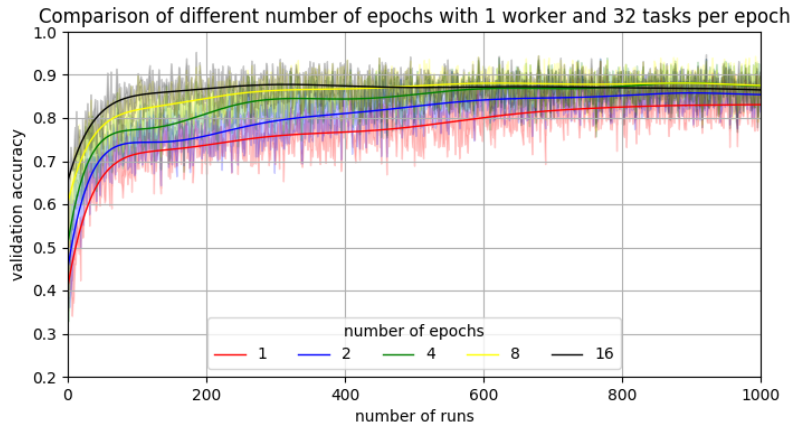


Figure 6.18: Comparison of the validation accuracy after 3 gradient steps with different number of epochs.

As it can be seen, the increase of accuracy per run increases approximately linearly proportional. This can be observed until the 400th run. After that the green graph catches up. From this it can be concluded that the effect is linear only up to a certain number of epochs and decreases beyond that.

INFLUENCE OF THE NUMBER OF TASKS. The purpose of this experiment is to determine whether increasing the number of tasks with only one worker generally increases the increase in accuracy per run. Therefore experiments with 32 to 512 tasks per epoch were carried out. The results can be seen in Figure 6.19.

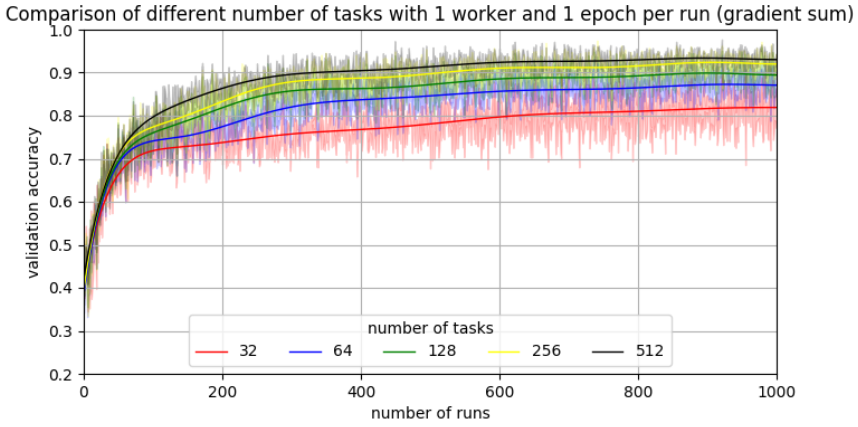


Figure 6.19: Comparison of the validation accuracy after 3 gradient steps with different number of tasks and usage of gradient sum

The results show that an increase in the number of tasks per run increases the increase of accuracy approximately linearly proportional with the number of tasks.

While we were doing these experiments, we got the idea, which we described in Section 3.3.3. If in benchmarking the learning rates are considered to be constant, then this assumption would no longer apply when changing the number of tasks. If we consider the step size as the multiplication of learning rate and number of tasks, then the step size increases when the number of tasks is increased. So if a fair comparison shall be warranted, the step size has to be divided by the number of tasks again.

We have implemented this by replacing the gradient sum with the arithmetic mean of the gradients as in the chapter mentioned above. Figure 6.20 shows the previous experiment using the arithmetic mean.

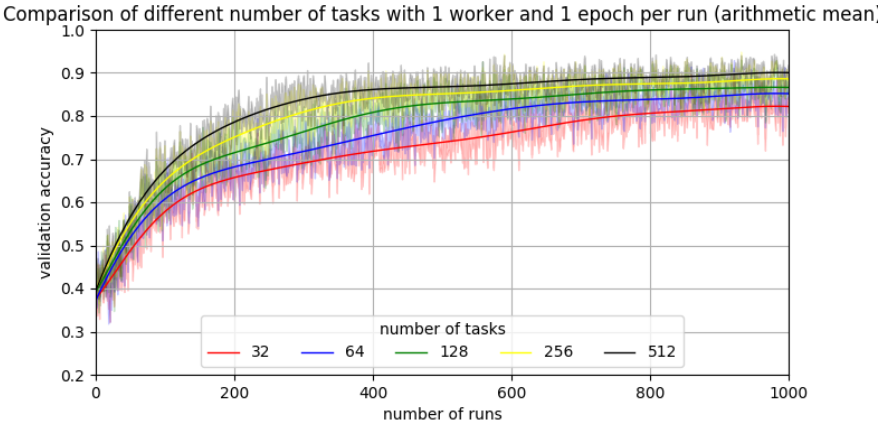


Figure 6.20: Comparison of the validation accuracy after 3 gradient steps with different number of tasks and usage of arithemtic mean

As it can be seen, the nature of the graphs is preserved. However, the graphs increase a bit slower because we used $\beta = 0.001$.

COMPARISON OF ARITHMETIC MEAN VALIDATION ACCURACIES. In the following experiments the different variants (i.e. the graphs with different parameterizations) will be compared and contrasted. Figure 6.21 shows the variants in which the respective parameters workers, epochs and tasks were increased by the ratio 16 compared to the baseline. Instead of the gradient sum, the arithmetic mean was used to generate the test loss for the meta-model and a learning rate of $\beta = 0.001$ has been used.

Figure 6.21: Comparison of validation accuracies with arithmetic mean: The graphs with one parameter each increased by ratio 16 are shown.

As it can be clearly seen, the graphs increase at different rates for a constant step size. While the graph with a worker number of 16 (blue) rises at about the same rate as the baseline, the graph rises much more strongly with the greatly increased number of tasks (512). Moreover it can be observed, that the green graph (with increased number of epochs) outperformes the other graphs significantly, until the 400th run, when the yellow graph catches up. After that the performance decreases and the variant with increased number of tasks outperformes all variants.

From the results it can be concluded that a fair comparison (constant step size) clearly shows that the accuracy per run rises much stronger with an increase in the number of epochs per run than the variants with an increased number of tasks and number of workers in the beginning. After a certain number of epochs the graph with an increased number of tasks outperforms the other variants.

Figure 6.22 shows a comparison of the influence of the parameters. The scaled graphs from Figure 6.21 are shown together with variants, where the increased parameters are mixed. It can be observed, that the compormise graphs between increased workers and epochs (green) and workers and tasks (yellow) perform significantly better than the graph where only the number of workers has been increased (red).

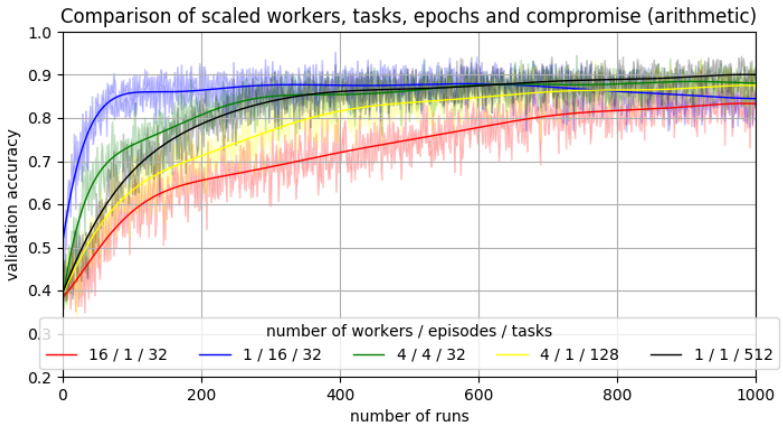


Figure 6.22: Comparison of validation accuracies with arithmetic mean: The graphs with one parameter each increased by ratio 16 are shown together with those, where two parameters have been increased with the same proportion.

Concluding these results, it can be said, that from a performance standpoint, increasing the number of epochs for the first few hundreds runs show the best performance. Even the compromise between increasing workers and epochs performs better than the rest. If the training is performed for more than these runs, a higher accuracy can be obtained with an increased number of tasks. Increasing only the number of workers performs worst.

COMPARISON OF GRADIENT SUM VALIDATION ACCURACIES. We performed the same experiments again with the gradient sum to show which results the algorithm gives when the gradient sum is used and no fixed step size is required. The comparison of the parameters, that have changed with a ratio of 16 is shown in Figure 6.23.

In principle we can see again, that the graph with increased number of epochs has the steepest slope in the beginning, but after about 220 runs the yellow and blue graphs catch up. The graph with increased number of workers performes in comparison to the arithmetic sum case significantly better, but still worse compared to the the other variants. In total the experiment with the increased number of tasks gains the best results in the long turn.

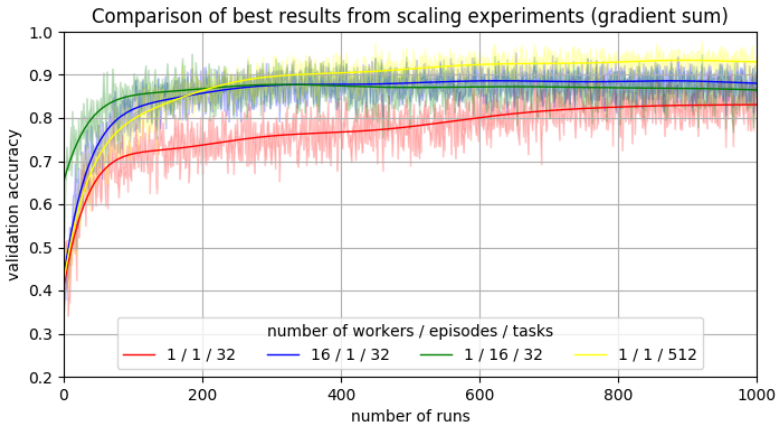


Figure 6.23: Comparison of validation accuracies with gradient sum: The graphs with one parameter each increased by ratio 16 are shown.

Also for the variant with the gradient sum, variants with mixed increases of the paramters have been performed. The results are shown in Figure 6.24.

For the gradient sum case it can be made clear, that in the first 200 runs the variants with 16 epochs (blue) and the compromise variant with 4 workers and 4 epochs (green) perform best. After about 200 runs, they perform worst and the variants with increased tasks (black) and compromise between workers and tasks (yellow) perform best and also have better overall validation accuracies.

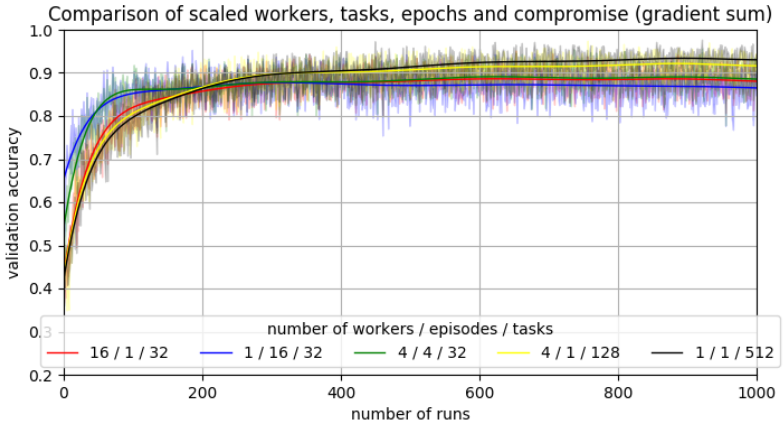


Figure 6.24: Comparison of validation accuracies with gradient sum: The graphs with one parameter each increased by ratio 16 are shown together with those, where two parameters have been increased with the same proportion.

Therefore from a accuracy increase standopoint, the variant with the increased number of tasks per epoch is to be chosen, if the gradient

sum is used. To conclude this, the foregoing experiments have only been performed under a measurement of accuracy increase per run. The effect of the parameter changes on the clock time was ignored.

6.7 TIME MEASUREMENT EXPERIMENTS

In this experiment we want to make a first evaluation how the increase of the different parameters affects the required wall time. These values are particularly interesting, since this study is mainly concerned with parallelization and the resulting time savings.

First a distinction between *computation time* and *wall time* shall be given. Since we parallelize calculations in this study, we save wall time. This corresponds to the time we have to wait until we can see the result of the calculation. The computation time, on the other hand, remains the same or increases a little bit more due to the additional parallelization processes. Although the change in computation time is also relevant, it is not examined in this study because it is very small anyway. Since machine learning engineers usually have to deal with long wall times and this work is aimed at reducing the wall time, only this will be discussed in the following.

In this experiment, concrete time measurements were made. An experiment was performed for each cell of Table 6.3. In each experiment, 10 runs were performed. The average duration of a run for each value in the Table was determined using 9 runs (2nd to 10th run). The first run was not included in the averaging because the first run lasted significantly longer than the following runs in all experiments. This might be the case, because in the first run the initialization of the TensorFlow model takes place. Furthermore, the calculated values are used for extrapolations and should therefore not contain any outliers such as the first run. The average duration of a run for each variant of the parameter values can be read in Table 6.3.

Table 6.3: Absolute Increase in Average Duration in Seconds of a Run With Different Scaling Ratio

scaling ratio	**1**	**2**	**4**	**8**	**16**
number of workers	7.67	7.49	9.59	10.05	10.97
number of epochs	7.67	14.9	28.69	56.95	114.93
number of tasks	7.67	15.02	28.92	57.19	116.67
seen tasks per run	32	64	128	256	512

This table is used for later calculation. To make these numbers easier to classify, Table 6.4 shows the relative increase in duration as a percentage. The baseline is the average wall time of an Omniglot classification experiment with 1 worker, 1 epoch per run and 32 tasks per epoch.

Table 6.4: Relative Increase in Average Duration in Percent of a Run With Different Scaling Ratio

scaling ratio	**1**	**2**	**4**	**8**	**16**
number of workers	100	97.67	124.96	131.01	143.03
number of epochs	100	194.26	374.03	742.42	1498.17
number of tasks	100	195.8	376.96	745.58	1520.85
seen tasks per run	32	64	128	256	512

The table shows line by line how a doubling of one of the parameters affects the average duration of a run.

For example, it can be seen that doubling the tasks (from ratio 1 to 2, i.e. from 32 seen tasks per run to 64 seen tasks per run) increases the relative wall time in average by 195.8%. This corresponds to almost doubling the wall time with the doubling of the tasks. However, if the number of workers is doubled, the relative wall time per run in average is even decreased by 2.33%.

In addition, the average duration of a run can be compared column by column across the variants. For example, with a scaling factor of 16 for training 512 tasks per run, the variant with 16 workers requires in average 143.03% more wall time in comparison to the baseline, whereas the variant with the increased number of tasks requires an average of 1520.85% in comparison to the baseline, which is 1377.82% more than parallelizing. Also the variant with increased epochs requires an average of 1498.17% in comparison to the baseline, which is 1355.14% more than the parallelizing variant.

These values already make it clear that if enough CPU power is available, parallelization can bring significant time savings. To make the effect visually clearer, Figure 6.26 shows the additional time consumption that occurs when the respective parameters are increased by a ratio of 16.

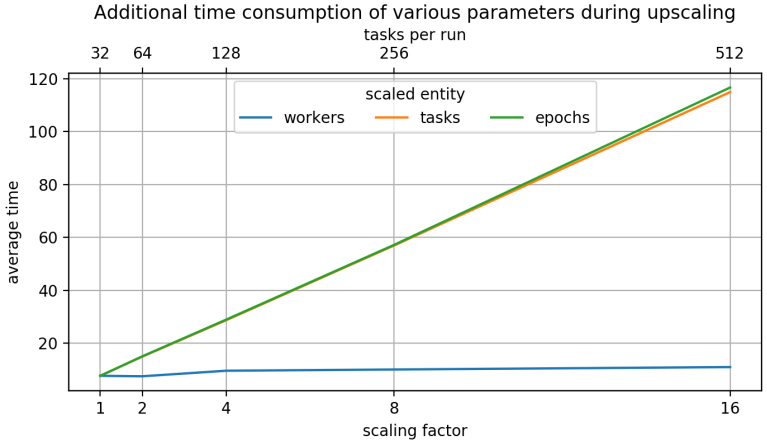


Figure 6.25: Comparison of the average wall time of a single run with different scaling factors for experiments with increased parameters workers, tasks, epochs.

It is clear to see that increasing tasks or epochs serially increases the cost of wall time liearly proportional, while the needed wall time for increasing the number of workers increses only very little. Using these findings and extrapolating the estimated time for 1000 runs, we get the graphs in Figure 6.26.

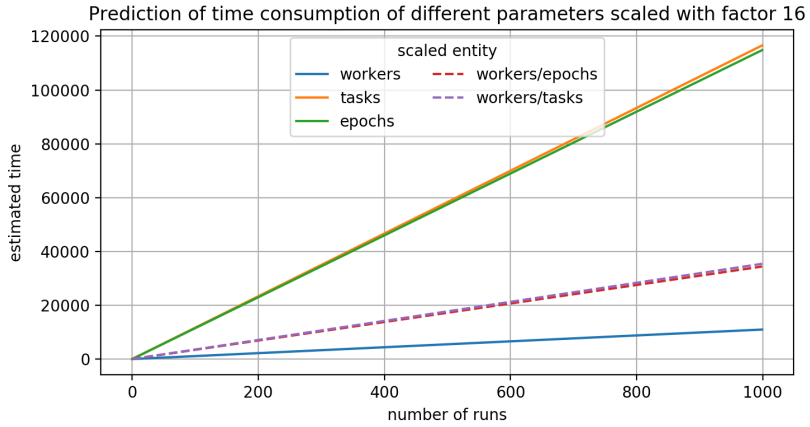


Figure 6.26: Estimated wall times for 1000 runs with different parameters scaled with a ratio of 16.

As it can be seen, the graphs with the increased number of tasks and epochs increase with a much steeper slope of the estimated time than those with increased number of workers. For example, if it is compared the time after 1000 runs of the green graph (114815 seconds) with the blue graph (10959 seconds), it comes to an eye that 94.6% of the time can be saved by parallelizing to 16 workers compared to a serial increase to 16 epochs.

The figure also shows two graphs, a mixture of 4 workers and 4 epochs (red) and a mixture of 4 workers and 128 tasks (purple), which show significantly lower runtimes. Take into account, that all experiments shown in the Figure had 512 tasks seen per run.

Taking into account the results from the previous series of experiments, where the accuracy per run increased significantly more when the number of epochs or the number of tasks was increased than when the number of workers was increased, it is concluded that the combination of both could be a good compromise between wall time and accuracy increase.

Further experiments in this regard should definitely be carried out. The increase of epochs in our experiments led to the increase in accuracy growth per run, but only to a certain number of runs. Furthermore a lower accuracy has been reached, as with an increased number of tasks. However, the wall time increases linearly proportional.

By taking into account, that is seems as just increasing the number of tasks yields the highest accuracy after a threshold of number of runs, it becomes clear, that a parallelization on the task-level would lead to better results. In a further study, the framework can be improved by implementing a parallelization on the task-level.

Part III

CONCLUSION AND OUTLOOK

7

CONCLUSIONS AND SUMMARY

This study was divided into two parts, an engineering part and a scientific part.

ENGINEERING PART. The first part was an engineering part in which a framework was created that combines the MAML algorithm of [5] for the three domains of regression, classification and reinforcement learning, since no such implementation existed at the time of the study. This framework was implemented with current versions of the programming language Python and the deep learning framework TensorFlow.

To verify the implementation of the algorithm, verification experiments with Regression, Omniglot and MiniImageNet dataset were conducted at the beginning of the study. It was found that we achieve very similar results as in [5] but perform slightly less well, at least with the MiniImageNet dataset. Therefore, in this study the baselines has been generated by our implementation to allow a fair comparison within this study for the experiments.

The reinforcement learning part is still in testing at the time of the submission of this study, which is why it was only treated theoretically here and therefore no experiments were made in this domain.

SCIENTIFIC PART. The second part of the study deals with the question whether the training of a meta-model can be accelerated by parallelization. To investigate this, we extended the algorithm of [5]. The new algorithm DMAML is an encapsulation of the MAML algorithm, which allows the parallelization of MAML workers. In the following a summary of the results of the scientific parts is given.

HYPERPARAMETER SEARCH. In the first parallelization experiments we used the parameters from [5]. Since we have found that only extremely small increases in the speed of the accuracy increases could be seen, we decided to perform hyperparameter searches for the meta-learning rate beta for all three domains and found that β was

not ideally chosen. Specifically, we found that for Omniglot $\beta = 0.01$, for MiniImageNet $\beta = 0.005$ and for Regression $\beta = 0.01$, the results were significantly better. Furthermore we can see that the parameter β can have a different ideal value for 1 worker than for many parallel workers.

CONFIRMATION OF THE IDEA OF PARALLELISATION The subsequent parallelization tests showed that in each domain the accuracy convergence (or loss convergence with regression) could be increased. In this context, it is also clearly visible that the advantage of parallelization increases very strongly if no parallelization took place beforehand, but increases much less strongly with an increasing number of workers, e.g. from 5 to 30 workers. From this we can conclude that the increase in accuracy through parallelization is not linearly proportional. The main question of this study, whether an increase of parallel workers can increase the convergence of accuracy, could already be answered with yes in these experiments.

INFLUENCE OF PARAMETERS We investigated whether the increase of the number of *gradient steps* during training with the Omniglot dataset leads to an increase in accuracy. We found that it could not be observed a significant increase in validation accuracy. We conclude that it is not the number of steps in the direction of the optimum, but rather following the direction that is decisive for learning success. This also confirms the findings of [5], where it was shown that an increase in gradient steps only adds little value from the perspective of learning success.

Furthermore the influence of the increase of the number of *workers*, *epochs* and *tasks* was examined and compared. Several observations can be made here. First, when the increase of the mentioned parameters is considered, it can be seen that in all cases an increase of the values leads to an increase of the accuracy per run.

Secondly, if each parameter is individually increased by the same factor, the increase in accuracy per run is highest when epochs are increased, at least for the first few hundrets runs. Then the graphs catch up with an increased task count and deliver higher accuracy values. The increase in the number of workers shows the smallest increase in performance, both in the increase in the first few hundred runs and in the total accuracy achieved.

Combinations were then examined in which two parameters were increased in the same ratio. Thus, the combinations of increased workers and tasks, as well as workers and epochs showed similar, but a

bit worse performance results to those in which only tasks or epochs were increased.

These investigations were performed for gradient sums, as defined in the MAML Algorithm, and also for the *arithmetic mean*. Comparing the graphs of the three parameters in this scenario, the variants perform in the same way as in the case of the gradient sums. However, it can be seen that here the graphs rise significantly steeper with an increased number of epochs. From this we conclude that the greatest effect can be achieved by increasing the epochs from the point of view of the slope of the accuracy per run in the first few hundrets runs. To maximize the total validation accuracy in long term, the number of tasks has to be increased.

Finally, we investigated the influence of increasing the individual parameters depending on the wall time. It was found that time consumption increases linearly proportional to the increase in the number of epochs and the number of tasks. However, the time consumption increases only very slightly when the number of workers is increased. We conclude from this that the wall time costs of parallelization for several workers is low. It can be seen that with an increasing factor of the increase of the individual parameters, the time saved by parallelization increases immensely. This shows that parallelization not only increases the accuracy compared to the baseline (MAML) but also has a significant advantage in the time domain compared to the increase of the parameters epochs and tasks.

As our experiments show, increasing the number of workers achieves a less sharp increase in accuracy after a few hundred runs than increasing the epochs or tasks, but with a massive wall time advantage. So if there is a need to reduce the wall time, parallelization has a tremendous effect. If there is a need to achieve the highest possible accuracies, no matter how long it takes, the number of tasks in the batch of tasks should be increased.

If a compromise between wall time and accuracy is desired, a hyperparameter search should be performed to find the right balance between the number of workers and the number of epochs or tasks.

If we go beyond this implementation it becomes clear, that a parallelization of tasks rather than MAML workers seems to be the optimal strategy for parallelizing, since the experiments have shown, that in long term the validation accuracy becomes proportionaly higher the higher the number of tasks is increased.

8

OUTLOOK AND FUTURE WORK

This paper had the main task to investigate whether increasing the number of workers can shorten the training of a meta-model in relation to wall time. In addition, the other parameters such as number of epochs or tasks were also examined. It was found that increasing the number of workers achieves a slightly worse total accuracy than the other variants and therefore, for a compromise between total accuracy and wall time, several parameters should be increased in combination.

These possible combinations should be investigated in further work and it should also be determined whether the findings are only valid for a certain dataset or the findings can be transferred to other datasets and domains. The framework should also be improved by adding the possibility to parallelize tasks. In the RL this will have a greater impact, because the tasks are much more complicated and take more time to be trained as in classification or regression.

Furthermore, experiments concerning reinforcement learning should be conducted. On the one hand, it should be investigated whether increasing the number of workers can shorten the training of the policy, on the other hand, the influence of weight generators should be investigated.

The meta-learning framework MetaWorld was presented in this paper. Since in previous papers only very narrow problems were treated, there are hardly any studies on complex and diverse environments. MetaWorld closes this gap and gives the possibility to train real different tasks.

In addition to that, in this paper only the first 1000 runs were investigated in the experiments. In further investigations, longer training times should definitely be included.

BIBLIOGRAPHY

[1] Marcin Andrychowicz, Misha Denil, Sergio Gómez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. "Learning to learn by gradient descent by gradient descent." In: *Advances in Neural Information Processing Systems* Nips (2016), pp. 3988–3996. ISSN: 10495258. arXiv: arXiv:1606.04474v2.

[2] Antreas Antoniou, Harrison Edwards, and Amos Storkey. "How to train your MAML." In: (2018), pp. 1–11. arXiv: 1810.09502. URL: http://arxiv.org/abs/1810.09502.

[3] Hans-Jochen Bartsch. *Taschenbuch Mathematischer Formeln*. 21. Aufl. Hanser Verlag, 2007. ISBN: 978-3-446-40895-1.

[4] Chelsea Finn. "Learning to Learn with Gradients." PhD thesis. University of California, Berkeley, 2018, pp. 306–340. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-105.html.

[5] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-agnostic meta-learning for fast adaptation of deep networks." In: *34th International Conference on Machine Learning, ICML 2017* 3 (2017), pp. 1856–1868. arXiv: arXiv:1703.03400v3.

[6] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. "Distributed Prioritized Experience Replay." In: (2018), pp. 1–19. arXiv: 1803.00933. URL: http://arxiv.org/abs/1803.00933.

[7] Gregory Koch, Zemel Richard, and Ruslan Salakhutdinov. "Siamese Neural Networks for One-shot Image Recognition." In: *32nd International Conference on Machine Learning* 1 (2015). ISSN: 00071447. DOI: 10.1136/bmj.2.5108.1355-c.

[8] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tnenbaum. "Human-level concept learning through probabilistic program induction." In: *Science* 350.6266 (2015), pp. 1332–1338. ISSN: 0036-8075. DOI: 10.1126/science.aab3050. arXiv: science.aab3050 [10.1126]. URL: https://www.sciencemag.org/content/350/6266/1332.full.pdf.

[9] Volodymyr Mnih, Adria Puigdomenech Badia, Lehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. "Asynchronous methods for deep reinforcement learning." In: *33rd International Conference on Machine Learning, ICML 2016* 4 (2016), pp. 2850–2869. arXiv: 1602.01783.

[10] Tsendsuren Munkhdalai and Hong Yu. "Meta networks." In: *34th International Conference on Machine Learning, ICML 2017* 5 (2017), pp. 3933–3943. arXiv: 1703.00837.

[11] Alex Nichol, Joshua Achiam, and John Schulman. "On First-Order Meta-Learning Algorithms." In: (2018), pp. 1–15. arXiv: 1803.02999. URL: http://arxiv.org/abs/1803.02999.

[12] Santanu Pattanayak. *Pro Deep Learning with TensorFlow*. 2017. ISBN: 9781484230954. DOI: 10.1007/978-1-4842-3096-1.

[13] Sachin Ravi and Hugo Larochelle. "Optimization as a model for few-shot learning." In: *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings* (2019), pp. 1–11.

[14] Sudharsan Ravichandiran. *Hands-On Meta Learning*. 2018. ISBN: 9781789534207.

[15] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge." In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252. ISSN: 15731405. DOI: 10.1007/s11263-015-0816-y. arXiv: 1409.0575.

[16] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. "Meta-Learning with Memory-Augmented Neural Networks." In: *33rd International Conference on Machine Learning, ICML 2016* 4 (2016), pp. 2740–2751. arXiv: 1605.06065.

[17] Jake Snell, Kevin Swersky, and Richard Zemel. "Prototypical networks for few-shot learning." In: *Advances in Neural Information Processing Systems* 2017-Decem (2017), pp. 4078–4088. ISSN: 10495258. arXiv: 1703.05175.

[18] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. "Matching networks for one shot learning." In: *Advances in Neural Information Processing Systems* (2016), pp. 3637–3645. ISSN: 10495258. arXiv: 1606.04080.

[19] Lilian Weng. *Meta-Learning: Learning to Learn Fast*. 2018. URL: https://lilianweng.github.io/lil-log/2018/11/30/meta-learning.html.

[20] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. "Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning." In: CoRL (2019), pp. 1–18. arXiv: 1910.10897. URL: http://arxiv.org/abs/1910.10897.

[21] Luisa M Zintgraf, Kyriacos Shiarlis, Vitaly Kurin, Katja Hofmann, and Shimon Whiteson. "Fast Context Adaptation via Meta-Learning." In: 2018 (2018). arXiv: 1810.03642. URL: http://arxiv.org/abs/1810.03642.