Replication study of MAML in reinforcement learning

By André Henkel

Summerterm 2020

University of Ulm

Due Date: 18.08.2020

1 Table of contents

2 Table of figures

3 Tables

4 Abbreviations

CG – Conjugate Gradient

FC – Fully Connected

GAE – General Advantage Estimation

KL – Kullback-Leibler

LS – Line search

MAML – Model-Agnostic Meta Learning

ReLU – Rectified Linear Unit

RL – Reinforcement Learning

TRPO – Trust region policy optimization

5 Abstract

This report shows a python pytorch implementation of the MAML algorithm in the domain of reinforcement learning. (Chelsea Finn, 2017) For this the continuous, static and deterministic 2D-Navigation environment is used. (Yan Duan, 2016)

It will shortly explain the context of MAML and it's strength within this environment. The used methods will be shortly explained. Furhtermore the implementation of this reacreation study as well as notable differences to the original are discussed.

The MAML paper asked three questions, which were in the later parts of it answered. These questions are evaluated partly in this report as well.

Results of experiments with different hyperparameters will be displayed and discussed. These results will directly compare to the data given by the paper. Also an comparison to a normal trained network is presented with the same changes of hyperparameters as for the MAML algorithm.

Problems and a general discussion of this replication study for the MAML paper and its findings are discussed at the end as well.

6 Introduction

Being able to learn fast and with few data is an indicator of human intelligence. (Chelsea Finn, 2017) If we learn to use a new tool with just a few minutes time and a limitation on gathering experience, we are still able to fastly adapt and make use of this tool. We do that by also using our past experience and by generalizing quickly.

To also be able to adapt quickly on a new problem setup with Neural Networks one can use meta-learning. This approach learns to learn and to give a good basis of the Neural Network to further adapt with few further gradient updates (few-shot learning) and with a small amount of data. (Chelsea Finn, 2017)

The paper this report is focussed on suggests in its abstract that the introduced model-agnostic learning method is broadly applicable to a variety of learning problems, such as classification, regression and reinforcement learning, in which a gradient descent algorithm is used. (Chelsea Finn, 2017)

“The goal of meta-learning is to train a model on a variety of learning tasks, such that it can solve new learning tasks using only a small number of training samples.” (Chelsea Finn, 2017)

Rather then training the model on a specific task, to do well for that one, in meta-learning the model’s goal is to learn a new task from a small amount of new data. The model is therefore trained by the meta-learner to learn on a large number of different tasks. The goal of meta-learning is therefore learning an initialization for the network, from which the model can then be adapted very fast and with few data to a new task. (Chelsea Finn, 2017)

This report concentrates on the reinforcement learning domain with the 2D Navigation environment as problem setup. Code for the implementation was used and adapted from (Deleu, 2018).

The difference to prior meta-learning approaches is now that MAML does not need additional parameters nor does it have constraints on the used Neural Network. (Chelsea Finn, 2017)

Speaking differently, this approach can be viewed as building an internal representation that is broadly sutable for many tasks. Also it can be seen as maximizing the sensitivity of the loss functions of new tasks with respect to the parameters. High sensitivity leads with small local changes to large improvements in the task loss and therefore the achieving of the task. (Chelsea Finn, 2017)

7 2D-Navigation Environment

The 2D-Navigation Environment is a continuous control task on a two dimensional playing field. The observation contains solely the current position of the agent. And the goal is to move the agent towards a goal position. This goal position stays the same over the period of one playthrough.

The only indicator to where the agent has to move in order to win is the received reward after each step. This reward is the negative distance between the new position of the agent and the goal position.

To illustrate this problem setup one can think of the childs-game “Hot cold”, in which the player has to find an object on the gamefield, but is completely blind while searching for it. The indication of this is simply a feedback by the game director, which in our case would be the environment, that tells the player if you are getting closer (hotter) or further away (colder) from the object the player has to find. The player is never explicitly told where the goal is he has to get to.

Similarly to this game the agent in the 2D-Navigation environment can only assume where he has to go after receiving the reward of each step. For adapting to this new knowledge one has to update the policy which in our case is represented by a Neural Network with a gradient step. In order to adapt to a new task within only a few updates (2-4) a.k.a. k-shot learning the network has to have a sensibility to adapt.

The implementation of this environment was done using the OpenAI-gym library. (openai)

The environment has an horizon of 100 steps and a random goal distribution within 0.5 units in each direction.

The difficult task for the agent is that he has to sense where the goal is, rather than being told by it’s observation where he has to go. This sense is expressed as reward of the negative distance to the goal position.

Training a policy on one task would need a complete retraining of the agent for a new task, since the policy only evaluates the current position and not the received rewards. Those rewards are implemented in the policy in form of a cost function and gradient steps in order to minimize this loss.

The environment is continuous in its state- and action-space, static as it doesn’t change without the interaction of the agent (excluding the reset of the environment for new tasks) and deterministic as the same action has always the same effect on the environment respective to its position and no stochastics are included (i.e. not adding a distribution to the used action within the environment).

8 MAML

Model-agnostic meta learning(MAML) introduced by (Chelsea Finn, 2017) is a method for training a Neural Network on meta-basis to quickly adapt to new tasks. Such as in the field of classification, regression and reinforcement learning.

For the latter it helps to accelerate the fine-tuning process of policy gradient reinforcment learning with neural network policies. This is done by training the policy on a variety of tasks and learning the meta objective of these tasks. Strictly spoken by minimizing the overall loss over a batch of these tasks for every meta update. (Chelsea Finn, 2017)

A simple graphical illustration of the meta-learning process with its adaptation afterwards is shown in Figure 1. There the thick line illustrates the gradient of the meta model, which goes in the averaged direction of the three losses depicted as L1-L3. These were calculated in the inner-update steps. After that the model is at a new point, from which it can adjust to newly presented tasks in an optimal fashion, depicted as Theta1-3 asterix. This illustration aims to explain that the model with it's parameter Theta is in an optimal position to quickly adapt to new tasks, such that it can change its parameter easily.

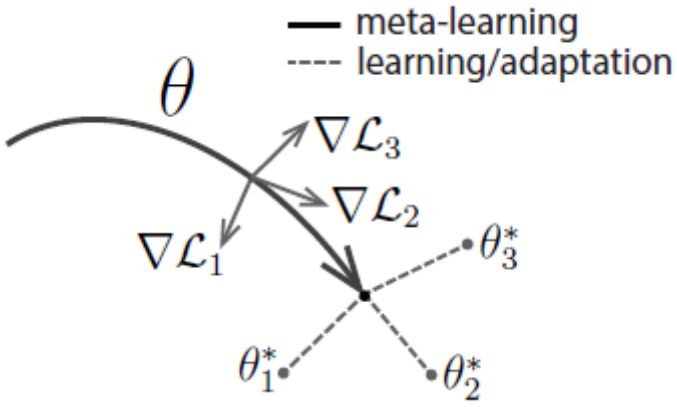

Figure 1 Diagram of MAML algorithm that can quickly adapt to new tasks. (Chelsea Finn, 2017)

The general algorithm for MAML in reinforcement learning is shown Figure 2.

Algorithm 3 MAML for Reinforcement Learning

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 tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample K trajectories $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using f_θ in $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 4
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample trajectories $\mathcal{D}_i' = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ 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 $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 4
11: **end while**

Figure 2 MAML algorithm for reinforcement learning (Chelsea Finn, 2017)

The algorithm consists of three parts. It starts with the initialization, then goes into the inner update loop in which a batch of tasks Is seletected to calculate the meta loss. This loss is then processed in the outer-loop meta-update to adapt the meta policy accordingly.

A practical implementation of this algorithm is discussed in chapter 10.

The algorithm requires sampling of different tasks, which are then batch-wise used to train the model on one meta-update. In each of these tasks K trajectories consisting of state-action pairs are gathered. These trajectories are evaluated with the loss function and with a linear feature baseline. (Yan Duan, 2016) Taking the just evaluated loss a copy of the meta-policy is created and updated according to the trajectories and the just evaluated loss. This training can be done multiple times. After that the just trained policy is used to gather more trajectories with its policy in place.

The so gathered trajectories are again evaluated to define their loss. All losses of these with the updated policies for each task gathered trajectories are then accumulated and used to update the meta-model.

The updated meta-model is then used from that point on for the next iteration of the algorithm. This continues until the model performs well in terms of its initialization for the learning of new tasks.

9 Methods

9.1 Trust Region Policy Optimization(TRPO)

TRPO was introduced by (John Schulman S. L., 2017).

The MAML paper suggests to use TRPO for the meta-learning update step. (Chelsea Finn, 2017)

For a more practical approach towards TRPO the authors of the paper suggest the following steps:

1. Collecting state action pairs with their respective Q-values
2. By averaging over the samples, constructing the estimated objective and constraint shown in Figure 2.
3. Approximately solving this constrained optimization problem to then update the parameter of the model. For that the paper suggests to use the conjugate gradient followed by a line search. This is also mentioned by (Yan Duan, 2016) where the line search follows the gradient direction computed by the conjugate gradient algorithm.

This is then repeated after each step. This approximation is only slightly more expensive then computing the gradient itself. (John Schulman S. L., 2017)

$$\begin{aligned} &\underset{\theta}{\text{maximize}}\ \mathbb{E}_{s\sim\rho_{\theta_{\text{old}}},a\sim q}\left[\frac{\pi_\theta(a|s)}{q(a|s)}Q_{\theta_{\text{old}}}(s,a)\right] \qquad (14)\\ &\text{subject to}\ \mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}}\left[D_{\text{KL}}(\pi_{\theta_{\text{old}}}(\cdot|s)\parallel\pi_\theta(\cdot|s))\right]\leq\delta. \end{aligned}$$

Figure 3 TRPO estimated objective with constraint

9.2 Linear Feature Baseline

As suggested by (Yan Duan, 2016) a baseline can be subtracted from the empirical return of the gathered data to reduce the variance of the optimization. This works for all gradient based algorithms except REPS. For this a Time-variing feature vector is introduced, using the follow vector shown in Figure 4 Time variing feature vector, where “s” is the state vector and the produt of the second variable is calculated element-wise. (Yan Duan, 2016)

$$\phi_{s,t} = \text{concat}(s, s\odot s, 0.01t, (0.01t)^2, (0.01t)^3, 1)$$

Figure 4 Time variing feature vector

This Linear Feature Baseline is used in the update steps for both the inner-update and the meta-update. This was also done in the original implementation of MAML. (Chelsea Finn, 2017)

9.3 Generalized Advantage Estimation (GAE)

Other than with the normal Bellman equation, where all rewards are summed up, with GAE a discounted sum of TD residuals are used. With that an estimator of the k-step discountd advantage is computed.

(John Schulman P. M., 2018) introduces an generilized estimator of the advantag function. This leads to a trade-off between bias and variance using the parameter lambda which is between 0 and 1. When using Lambda = 0 it can be compared with TD(0) so that only the next step is evaluated. Whereas a use of Lambda = 1 means that it evaluates all steps simply spoken. (Breloff)

9.4 KL-Divergence

Kullback-Leibler(KL)-Divergence is used to measure the distance between continuous distributions and can be used for performing direct regression over the space of continuous output distributions. (KLDIVLOSS)

This gives a measurement of the difference between two distributions. With that a difference can be calculate and a loss in the RL context can be computed.

10 Implementation

10.1 Program overview

The implementation of this paper was done by the authors using Tensorflow. For this replication study the implementation was done using Pytorch.

The program consists of the maml algorithm, a dedicated agent to sample and interact with the environment, the neural network policy, the linear feature baseline and some utility functionalities.

The environment is seperately implemented using the open-ai gym environment library. With that the environment can be installed locally and can be accessed independently from the program.

10.2 Code

The code for this replication study was partly taken and adapted from (Deleu, 2018). His implementation was tested as well.

The implementation closely follows the schematic of Figure 2. It starts in the overall algorithm function and initializes all necessary objects: Agent, Environment, Policy and the Linear Feature Baseline.

Then the algorithm starts. For that the agent gathers 20 trajectories with the current meta policy then creates a deep copy of this policy to train it on these trajectories with one gradient update and a fixed step size. This trained copied model is then used to gather another 20 trajectories. Both trajectory batches will then be returned by the agent. This is done for 20 tasks and the losses of all trajectories will be accumulated and normalized by dividing by the number of tasks to then be used as total loss for the update of the meta model.The before created deep copies will then be destroyed.

For the evaluation the whole batch of 20 trajectories with up to 100 steps is used. Then the loss is calculated with GAE together with the Linear Feature Baseline (Yan Duan, 2016). For better performance of the training the total reward was divided by the amount of used tasks per meta step, which let empiricially to better training results.

For the meta update TRPO is used as optimization, just like in the paper. With a KL-divergence constraint and line search with conjugate gradient, to improve the policy in its meta-update. Within the line search the loss after a gradient update of the meta-model is compared to previous losses after 1 gradient update as well. When this loss improves, namely decreases the returned mean loss over a batch of tasks and the kl-divergence between both distributions satisfies the KL-constraint, the just made update on the meta-model stays and is used from then on for the next TRPO meta optimization.

The implementation with Pytorch in Python for the meta update is a bit unintuitive at first. In order for the pytorch autograd engine to calculate the gradients for the update of the neural network, it needs to forward propagate the input through this network. But since we have to use a deep copy of the meta model there's no connection for the engine to see this. Therefore it needs to gather the created training and validation trajectories that are created from the agent during the inner-update

learning. All trajectory batches over all tasks being used for one meta update are forwarded through the network, in order for the Pytorch engine to recognize the input. Then the previously calculated total loss is used to generate the gradients for each unit within the network via backpropagation. This backpropagation can be achieved with Pytorchs autograd library, that then returns the gradients of this network w.r.t. the input and the loss. Calculating the gradients of the inner update and then using them to create the gradients of the overall meta-gradient update means that the second derivative is used. Using on top of this the TRPO optimization the derivative is calculated three times. The paper suggests that an approximation of the TRPO derivative can be achieved with finite differences. (Chelsea Finn, 2017)

10.3 Training and performance check

One performance check is the output of the total loss per update step to measure the performance during training. This on the other hand doesn't account for the different distances of the training tasks.

It can also be every n-th meta update an evaluation conducted in which the meta policy is trained on a new task with the evaluation step sizes in order to see how the meta policy can adapt. The test-task used for the evaluation during training was kept the same and not used within the training task batch, in order to see how the policy performs w.r.t. the meta optimization for new tasks.

Both methods just give a glimpse of the actual performance of the meta model. In order to see if it keeps learning over the whole duration, one can output die KL-divergence and the improvement with the TRPO meta-optimization, since during this optimization new tasks are gathered and evaluated in order to improve the policy.

10.4 Testing

In order to compare and test the model with others, the same tasks and the same amount of gradient updates need to be used.

For comparison the mean value of the total amount of rewards gathered within one episode after each gradient step for each model over an amount of tasks was used. Additionally to give also perspective to the variance of the different models the best and worst performance for each model in each gradient step was used to show with the matplotlib errorbar how good the model performed in this regard. Just taking the mean over an amount of tasks would not necessarilly account for the variance as well.

For a good distribution of different tasks an amount of 20-200 tasks for the tests were used. Although after most tests, the changes between a comparison of 20 and a comparison of up to 200 tasks didn't differ much in the empirical evaluation.

The scripts used for testing are provided together with this report. The script for comparing models on multiple tasks is called test.py and the script for a 2D display of the behaviour of the task before any update and after k-shot learning is called show.py. The parameters have to be adjusted within the script.

10.5 Table of hyperparameters

In the following tables is a summary of the used parameters, for the environment, training and optimization.

Starting with Table 1 it uses the same parameters as used in the paper. Table 2 gives some general parameters. Table 3 gives the parameters for the used Neural Network. And Table 4 gives the hyperparameters used for the TRPO meta-optimization. Most of these parameters are used from the official Github repo of the paper (cbfinn, 2017) and another pytorch implementation on Github for MAML (Deleu, 2018).

Batch-size Tasks	Batch-size Episodes	Episode Horizon	State size	Action size	Task range	Action space
20	20	100	2	2	-0.5 to +0.5	-0.1 to +0.1

Table 1 Parameters regarding Tasks

Inner update learning rate	Inner updates	Sigma (init, min, mx)	Trajectory reduction gamma	GAE lambda	LFB Regulation coefficient	HVP damping
0.01	1	0.1 1e-6 1.0	0.99	1.0	1e-5	1e-2

Table 2 General parameters

Network type	Input size	Output size	Hidden layer	HL activation function	Output Activation function	Output distribution	Optimizer (inner learning)
FC feed forward	2	2	(100,100)	ReLU	tanh	Torch.Independent	SGD (0.9 momentum)

Table 3 Neural Network parameters

CG iterations	Max line searches	Kl Constraint	CG damping	Linesearch Backtrack ratio	Initial step size	improvement
10	15	1.0e-2	1.0e-5	0.8	1.0	Any better result

Table 4 TRPO parameters

11 Results

11.1 Methodology

One thing to consider is that the total loss per episode is a sum of the distances from all states to the goal state. This means it could happen that with some tasks the goal position is so close to the initial position that the model from the beginning onwards performs well. This is overcome by using more tasks to give a better distribution. Since we don't know which tasks were used for the paper results and neither do we know the variance in performance per step we can only indicate a comparison by the initial total reward value before the first update step. In the paper this value was around -40.

For the shown results within one illustration all policies used the same tasks. Also the same tasks were compared for each gradient update of the models.

The results are created with implementations of different hyperparameters such as the Hidden Layer sizes, the output activation of the neural network policy and the initial sigma value for the variance of the output distribution, for learning and evaluating. It was also compared by evaluating the policy without a distribution, which means by taking the outputted mean without a variance.

For the evaluation of each gradient update step a batch size of 40 trajectories was used, just as in the paper. Smaller batch-sizes of 20 trajectories for each gradient update didn't change the performance significantly. This was empirically tested over the course of this replication study.

The paper mentions a big first gradient step size with smaller steps for all subsequent steps. Although the paper states to use 0.1 with 0.05 as the subsequent step sizes, the results in this report were created with much smaller sizes.

There is an empirical correlation between the initial sigma value and the step sizes needed in order to improve the current policies within the evaluation phase. A smaller initial variance also leads to a smaller variance after the training. The smaller the variance in the evaluation phase is, the less exploration is done while gathering the 40 trajectories for one update step. The smaller the variance, the smaller the step size needed to adjuste the policy optimally was found empirically while trying out different settings.

In comparison the pretrained model with the same sigma value needs larger step sizes to adapt optimally. The sigma after training are around 0.03 when initialized with 0.1 and they are around 0.3 when initialized with a sigma value of 1.0 at the beginning. The sigma values are also capped in order to no let them grow and therefore to not do too much exploration.

Also the subsequent step sizes which are in the paper which are just half the size of the initial don't work as good as taking only a fifth to a tenth of the first step size. This is for both, the MAML as well as for the normal pretrained model.

The paper states it uses for all reinforcement learning setups two hidden layers with ReLU as activation function and per hidden layer 100 units. With empirical testing there was no significant difference between using ReLU and tanh as activation function for the hidden layers. Since the problem setup around the 2D Navigation environment is rather simple a test with using only 16 units

per hidden layer was conducted. The result was that the smaller network performs almost as good as the larger network, using otherwise the same hyperparameters and using the same test tasks for comparison. The only difference was the variance after each step, where in the smaller sized network some trajectories returned way worse performance than the indicated mean, which is shown as the errorbars within the illustration for each model and each gradient step during evaluation. This test was conducted over 200 test tasks in order to give a good distribution.

11.2 MAML and pretrained model

As well as in the paper the resulsts show a comparison to the normal pretrained model which didn't use meta learning, but in the same evaluated way as the other meta-trained models.

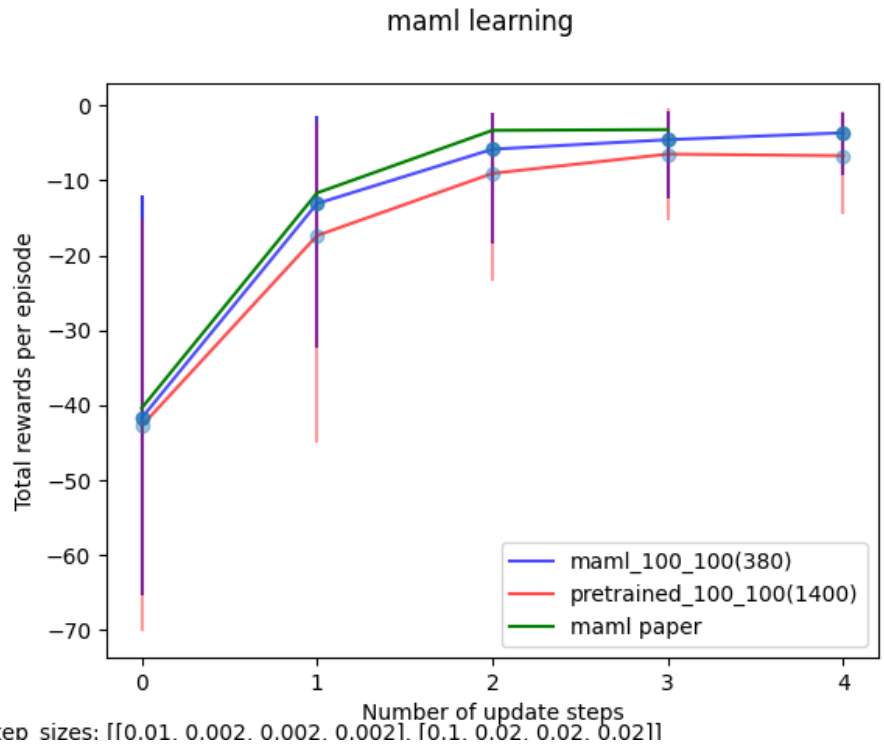

Figure 5 Comparison of a normal pretrained policy to a MAML pretrained policy with 100 Hidden Layers over 4 gradient steps

For the replication study a comparison to the original paper data is shown in Figure 5 as well as a normal pretrained model and the a with the MAML algorithm pretrained model with two hidden layers each with 100 units, as stated in the paper. At each step the mean of all evaluated tasks per model was calculated and displayed. The numbers in the brackets in the illustration show the updates to the model that has been used for this test. The vertical lines on each step are errorbars indicating the best and worst performance of each model per step. They are overlaying with an alpha value.

Comparing the recreated MAML policy to the given data, it shows that the performance of the model could be recreated. Not just for the first step, but also for the consecutive steps, which was one important question the authors stated at the beginning of their paper.

There was no indication of how big the variance was in the original MAML policy from the authors, so we can only compare the mean. Looking at the errorbars indicates that the MAML approach not just performs slightly better than on average than the normal pretrained model, but also has less variance in it's performance per step.

As mentioned before this evaluation was created with over 20 tasks for each model. The same tasks were used. Also the meta model loss before the first gradient update is in a similar value region as the paper, which makes it comparable. Also as shown later the models perform similar as the model in the paper in terms of their behaviour before any gradient update.

On the bottom of the illustration the gradient step sizes for each model are shown. The left one is for the recreated MAML algorithm and the right one for the normal pretrained one. The paper uses update steps of 0.1 with following 0.05. This was way to big for the recreated policy, as is shown later with a 2D demonstration. The normal pretrained model instead uses similar update steps as given by the paper. This will be discussed in chapter 13Discussion .

In Figure 6 Results for a task distribution further away is shown how the policy performs with a worse starting point than in comparison to the data given by the MAML paper.

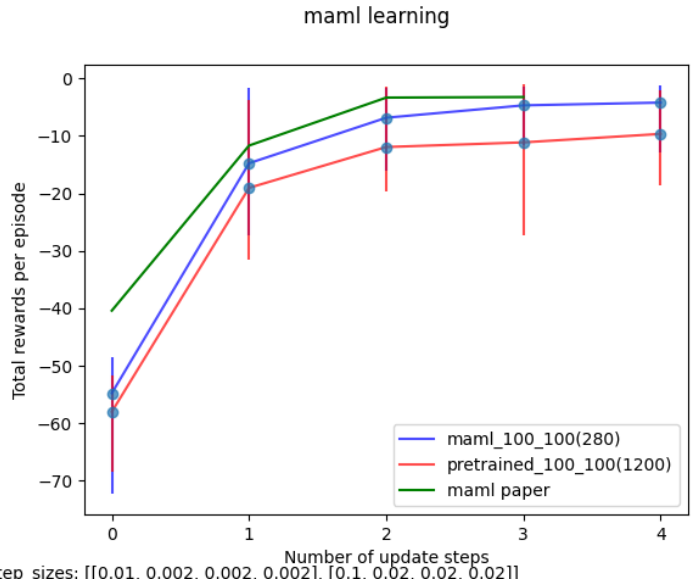

Figure 6 Results for a task distribution further away

The corresponding values are displayed in Figure 7 Results in numbers for a farther task distribution. Starting from the left the first number is the mean total reward of the policy before any gradient update with the test task. Each number towards the right is then one additional gradient update with the same task. The upper row are the numbers for the MAML algorithm and the bottom one is for the normal pretrained model.

```
[[-54.77609504 -14.75902894  -6.83326397  -4.67877125  -4.19812207]
 [-58.0111011  -19.03932046 -11.93192704 -11.12265229  -9.64867223]]
```

Figure 7 Results in numbers for a farther task distribution

This shows that the performance values can be even compared to tasks, which in the preupdate evaluation have a worse starting position. Comparing those numbers to the data presented from the paper shown in Figure 8 the same trend of improving with additional steps and the total mean reward are in the same region of performance. The recreated MAML algorithm performs slightly worse though.

num. grad steps	0	1	2	3
context vector	−42.42	−13.90	−5.17	**−3.18**
MAML (ours)	−40.41	**−11.68**	**−3.33**	−3.23

Figure 8 Data from the MAML paper on the 2D Navigation task as mean total reward (Chelsea Finn, 2017)

11.3 Small Neural Network

The next comparison was done with a smaller Neural Network for both the normal pretrained model, as well as the MAML pretrained model. For this the two hidden layers in the network were reduced to 16 units each. This is shown together with the original MAML data given by the paper in Figure 9.

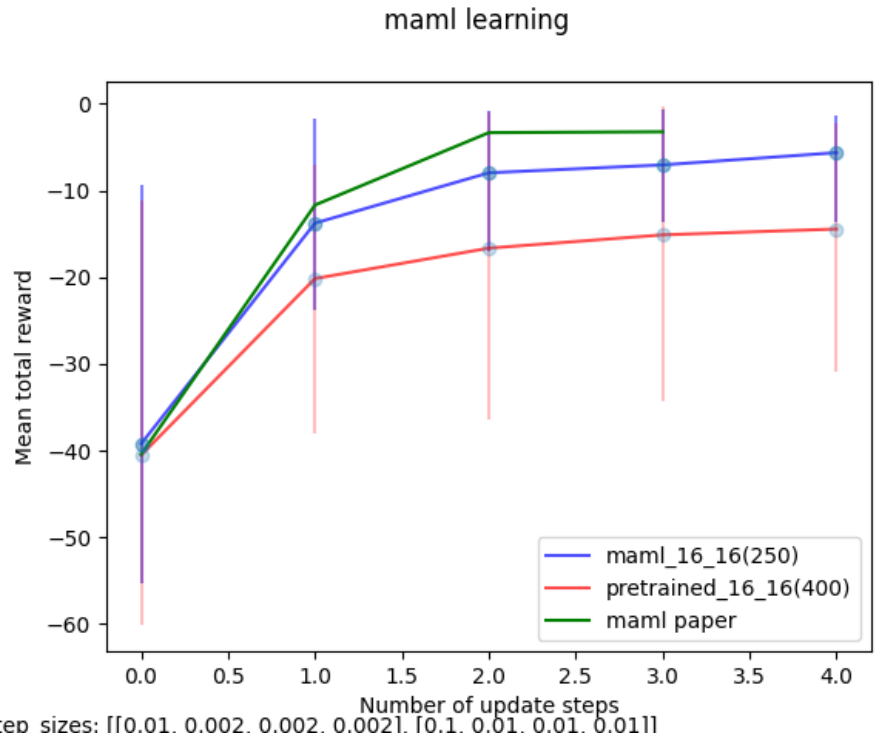

Figure 9 Comparison of pretrained MAML and normal pretrained policy on multiple gradient step updates with 16 units per hidden layer

With 16 units per layer the performance of both networks is slightly worse in comparison to the same models that used 100 units and otherwise the same evaluation procedure.

On the bottom are again the used step sizes for each model. These were figured out empirically and the best ones were used for each network.

Good to see is that the errorbar of the normal pretrained model is larger than the maml one in all steps, and also doesn't become smaller. On the other hand the MAML errorbars indicate that its performance becomes better over all tasks which leads to a fewer variance and overall better results in the display of the mean values.

Overall do both networks not compare to the results of the paper.

11.4 Effect of meta updates

Next up is a comparison on the performance for a different amount of meta updates. The paper states 500 meta updates, which didn't perform better than the same algorithm used on fewer meta updates. This was evaluated empirically throughout this replication study. The comparison for four different meta-update cycles is shown in Figure 10.

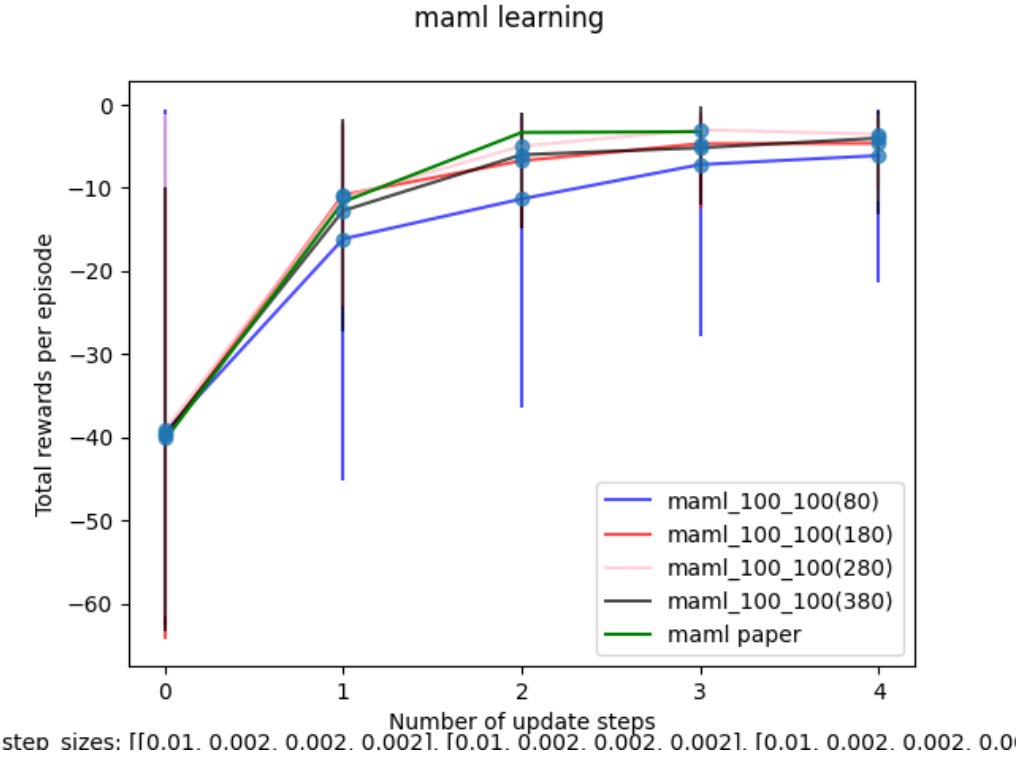

Figure 10 Comparison of the MAML algorithm on four different meta-update cycles

This evaluation was again created with the same conditions for all models and on 20 different tasks. These results are again well comparable to the original paper data since they start all at around -40 before any update.

The model performs worst with 80 updates both for the mean and the variance per step. It then becomes better and performs best with 280 meta-updates. With 380 it then becomes slightly worse. This is not necessarily an indication, that the model performs worse with more than 280 meta-updates, since the tasks might have fitted slightly better for one than the other.

11.5 Out of Taskrange performance

In this evaluation the performance of the model was tested on tasks, that weren't inside the meta trained task range, namely within -0.5 to +0.5 in both directions around the origin.

The test focusses on ranges from |1.0| to |2.0| in all directions. So that no task is within the meta-trained taskrange.

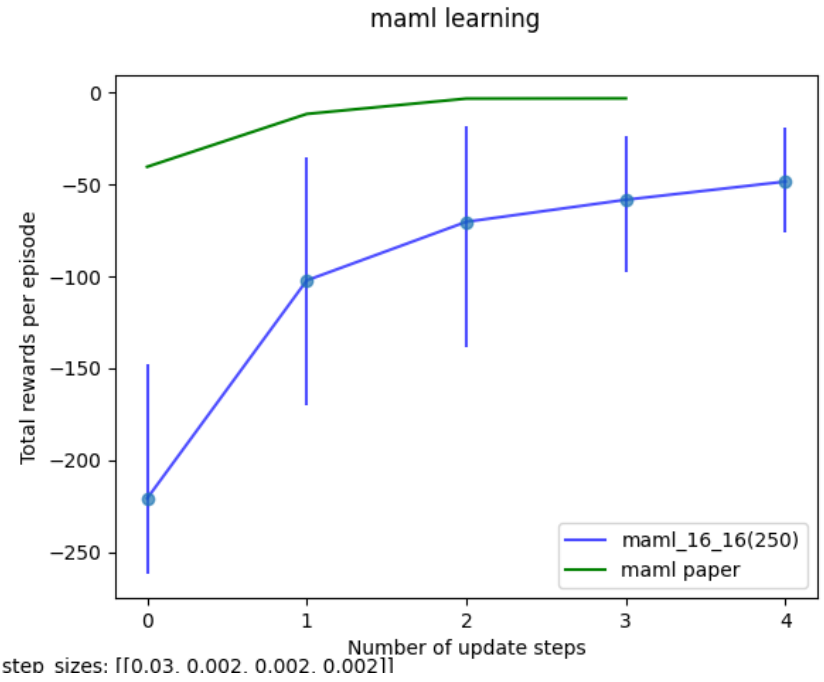

Figure 11 MAML performance on tasks out of the meta learned task range

The initial mean return is way lower than the maml paper comparison, due to the nature of the environment. The main point that can be seen here, is that the model still increases the performance over multiple steps and also reduces the variance. This test was conducted again with at least 20 tasks.

Since it was discussed earlier, that the smaller network performs similarly good as the big network, the 16 units network was used for this evaluation and still increases it's performance over multiple steps.

11.6 Effect of policy output variance

The effect of policy variance is shown in detail here.

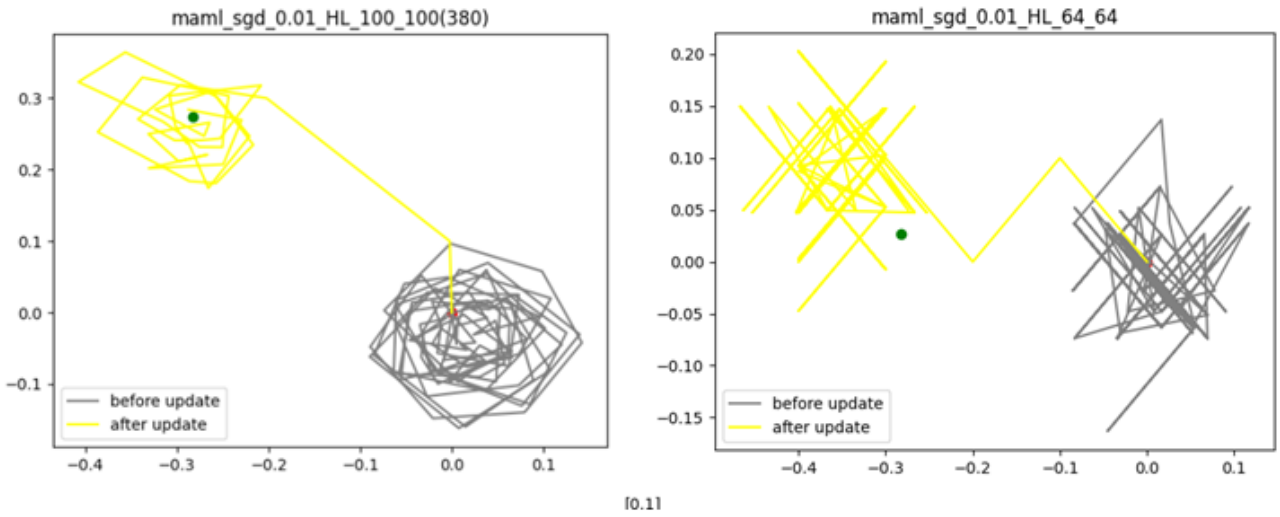

Figure 12 Comparison of different initial sigma values. The left model was initialized with a smaller sigma value than the right one. Otherwise the training procedure and duration was similar.

Note that in the comparison the goal state is slightly different. This was done to illustrate the jagged path the model on the right took to reach the goal destination. Also different hidden layer sizes were used. But as explained before and as tested empirically these changes didn't change the behaviour according to the sigma value. The left illustration was created with a model that was initialized with 0.1 as sigma and the right one with a sigma value of 1.0.

After similar many update steps, the left model sigma value had a value of around 0.03, while the right model still had a large sigma of 0.3. Also after 500 meta-updates the sigma value wasn't small enough to perform as good as the small initial sigma value. This jagged behaviour is due to the fact, that the policy can't output a value within the action space of the environment with such large variance. Furthermore when removing the distribution effect and plainly taking the outputted mean as action, then the model behaves similarly, since it didn't learn to converge into the action space within 500 meta-updates.

The left image shows a small turn before going straight to the goal position. This might be due to the small variance and hence small exploration the agent can perform within this single update step, which takes 40 gathered trajectories for the gradient update.

A test with more than 500 meta-updates, that would test at which point the network converges was not done, due to time and resource limitations. This comparison was done, because in the official Github repository of the paper an initial sigma value of 1.0 was used.

Furthermore the convergence to the action space with a small initial sigma value was increased by using an tanh as activation function for the mean output of the policy defining the action. With that the policy was already outputting values between -1 and +1. Starting from that the network was closer to the convergence towards the action space of -0.1 to +0.1 per action value. Without this step the used action values were still up to the area of |5| on similarly many meta-updates and with a initial sigma value of 1.0.

11.7 Sensibility of the model

The paper describes the meta-learning as creating a sensibility of the network for gradient updates in order to reduce the loss. This means that just one small gradient update step reduces the loss of the task quite significantly and therefore increases its performance.

This was tested empirically throughout the course of this replication study. Even small changes away from the optimal step size created big differences. These differences are shown in the following 2D displays of the behaviour of a 16 units network. It is to say, that the networks with 100 units behaved in a similar fashion. And the illustrations are just presented to give the reader a visualization of this change.

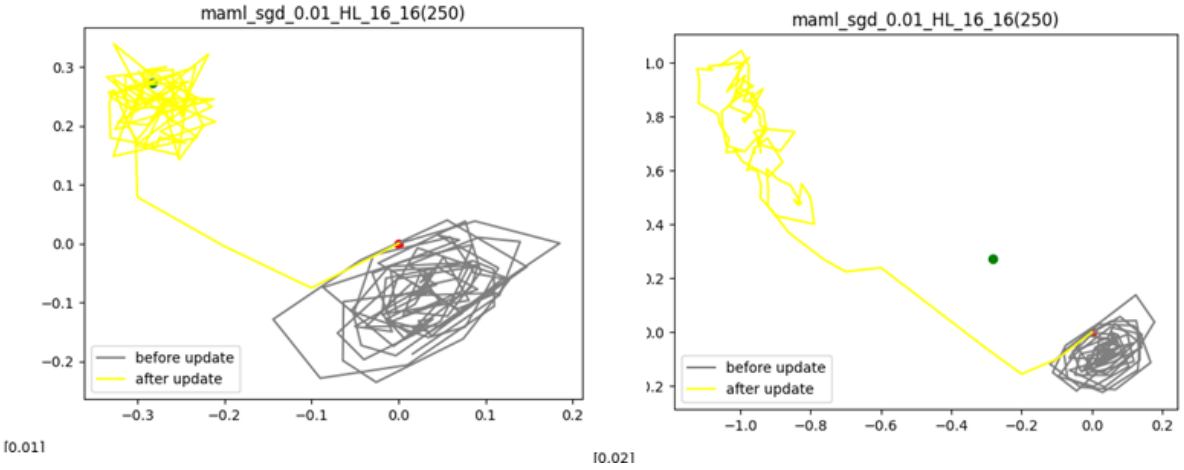

Figure 13 Exemplatory comparison of gradient step sizes for the demonstration of the sensitivity of the policy

For the comparison shown in Figure 13 the same test task, depicted as green dot, and the same network were used. The initial state depicted as red dot is as in all results shown at [0,0] in the environment. Note the different scales of both illustrations.

The left 2D illustration uses an optimal single step with a step size of 0.01 and the right comparison a bigger one with 0.02. While the optimal single step size shows good performance, the bigger one seems to overstretch the sensibility and overshoots the target. It seems to have sensed the correct direction, but has a bad understanding of the correct location of the goal position.

11.8 Multiple update steps

One of the three initially asked questions in the MAML paper was whether a model learned with MAML can continue to improve with additional gradient updates and/or examples. (Chelsea Finn, 2017) This was shown in the previous parts of this chapter that this is the case. The paper gives a table with results as shown in Figure 8 in which the mean total reward is shown for a numer of evaluated tasks. Our results give data for four update steps, as they still continue to improve, even if it is just a small improvement. Over the course of this replication study, multiple update steps of up to 20 were tested and the improvement didn't continue to increase, hence only four update steps are presented in the results of this study. This can be easily tested within the provided code for this study in the test.py script, when adding additional step sizes to the STEP_SIZES_DICT parameter.

12 Problems

The implementation with pytorch was unintuitive at first, because it didn't have a native Hessian vector product as compared to Tensorflow. Also the implementation for the update of the meta model requires to use the native autograd engine and an extraction of the parameters of the model in order to correctly backpropagate the total loss as described in chapter 10.2 Code.

Also using the output of the last fully connected layer as mean for the distribution was difficult to train. It had problems of converging into the action space of -0.1 to 0.1 per output unit. Therefore a tanh activation function was on it to then be passed as mean into the distribution.

The REINFORCE algorithm converges early, as mentioned in (Yan Duan, 2016). This is probaly one reason why the comparison with a straight forward trained REINFORCE algorithm on the 2D navigation task is finding it's optimum in a strange solution. This might have been a problem why the the initial variance is of importance. In this test a policy was trained from start on just one task and still had problems with diverging to the action-space.

The paper doesn't explicitly state the initial sigma value for the variance of the policy distribution, but in the official GitHub repository (cbfinn, 2017) an initialization of 1.0 is used. This leads to good exploration during training, but it again means it is hard to converge and therefore exploit. Using a initial sigma value for the distribution of 0.1 led to better performance. When visualised with a small variance the policy after some evaluation steps takes a curved approach towards the goal, rather than a straight line as shown in the paper. This is due to the fewer exploration being done during the evaluation phase. Nevertheless the total reward of these trajectories are comparable to the results presented in the paper.

13 Discussion and Outlook

The results show that the normal pretrained neural network policy performs similarly good to the maml algorithm with many units per hidden layer. But not as good with fewer units per hidden layer.

This is why I believe a replication study of this paper with the 2D-Navigation environment is a good way to start with meta-learning but it does not show the full potential of MAML, as it probably would on other tasks like the 3D-ant and the half-cheetah environments which have a higher action space and are of a more complicated nature. This is shown in the paper where the authors compare and outperform the context vector algorithm with their MAML algorithm in these more difficult environments.

Importance sampling wasn't implemented for the meta optimization with TRPO, which could be used to speed up learning a bit and might further increase performance. This wasn't done, since the obtained results showed sufficient performance.

Also paralellisation for the gathering of the training and validation trajectories within one meta-step can be executed in paralell, since there is no cross dependency of the deep copied models for the inner update.

The paper stated asking three questions, which the authors answered within their paper. Those questions are the following, which this report confirmed within it's scope.

Experimental evaluation questions from the paper:

- Can MAML enable fast learning on new tasks?
  ➔ Yes, this was shown and the performance was almost replicated in terms of the loss.

- Can MAML be used for meta-learning in multiple different domains, including supervised regression, classification and reinforcement learning?
  ➔ This paper focussed only on one environment within the reinforcement domain. Hence it can only confirm this area in which MAML worked.

- Can a model learned with MAML continue to improve with additional gradient updates and/or examples?
  ➔ This was tested with different network sizes and was confirmed, as the policy improved after the first gradient update further with additional gradient updates and newly gathered examples.

The used hyperparameters in this replication were different from the ones given in the paper, or used in their official repository. Furthermore adaptation to the network output was done in form of an tanh activation for the mean value. This might be some area to look closer into for future work.

As described in chapter 12 the small variance in the replication leads to a curved approach towards the goal position. This could be investigated further, such as changing the variance for the distribution during the trajectory gathering in the evaluation phase to a bigger number.

14 References

Breloff, T. (n.d.). *Deep Reinforcement Learningwith Online Generalized Advantage Estimation*. Retrieved August 16, 2020, from Deep Reinforcement Learningwith Online Generalized Advantage Estimation: www.breloff.com/DeepRL-OnlineGAE/

cbfinn. (2017, July 26). *Github*. Retrieved July 19, 2020, from Github: https://github.com/cbfinn/maml_rl

Chelsea Finn, P. A. (2017, July 18). Model-Agnostic Meta-Learning for fast Adaptation of Deep Networks.

Deleu, T. (2018). *Model-Agnostic Meta-Learning for Reinforcement Learning in PyTorch*. Retrieved August 15, 2020, from GitHub: https://github.com/tristandeleu/pytorch-maml-rl

John Schulman, P. M. (2018, October 20). HIGH-DIMENSIONAL CONTINUOUS CONTROL USING GENERALIZED ADVANTAGE ESTIMATION.

John Schulman, S. L. (2017, April 20). Trust Region Policy Optimization. Lille, France.

KLDIVLOSS. (n.d.). Retrieved August 16, 2020, from pytorch: pytorch.org/docs/stable/generated/torch.nn.KLDivLoss.html

openai. (n.d.). Retrieved July 19, 2020, from https://gym.openai.com/

Yan Duan, X. C. (2016, May 27). Benchmarking Deep Reinforcement Learning for Continuous Control. New York, NY, USA.