Deep Meta-Learning

AutoML - Assignment 2

1 Introduction

Deep neural networks have enabled great successes in various areas. However, they are notoriously data-hungry: they require a lot of data to achieve a good level of performance. Naturally, this raises the question of how we can let them learn more quickly (from fewer data) just like humans.

Meta-learning is one approach that could achieve this by learning how to learn. In this assignment you will study, implement, and investigate the popular meta-learning technique, namely **M**odel-**A**gnostic **M**eta-**L**earning (MAML) [2]. You will also be required to depict knowledge on other meta-learning tasks, such as **P**rototypical **Net**work (PNET) [5] and Reptile [4].

2 Few-shot learning

To investigate and study these techniques, we will use the few-shot learning setup, where techniques have to learn new tasks from a limited amount of data. More specifically, we will use the N-way k-shot image classification setting. Here, every task consists of a support set (the training data) and a query set (the test data) following these criteria:

- For a given task, the classes in the support and query set are the same (we want to measure how well we have learned on the support set, by evaluating on new inputs from the same classes)
- There are exactly N classes in the support and query set per task
- The support set contains exactly k examples per class
- The query set may contain a variable number of examples per class

An example of a task following the N-way k-shot setting is displayed in Figure 1.

2.1 Omniglot

We will be using Omniglot [3]. The dataset consists of 1 623 classes (distinct characters) and contains 20 examples (black-white images) per class. This dataset is frequently used as a few-shot learning benchmark in the literature.

Meta-learning has 3 stages:

- Meta-training (train on meta-training tasks)
- Meta-validation (validate learning algorithm on new classes)
- Meta-test (evaluate final learning algorithm on never seen before classes)

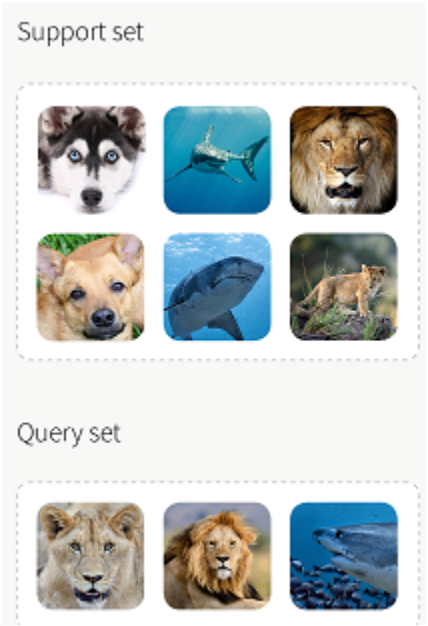

Figure 1: Example of a N-way k-shot task. Note that there are exactly N classes and k examples per class in the support set. Image source: [1].

The classes that are used for these 3 stages are distinct to ensure that we measure how well our techniques can learn new concepts/classes.

2.1.1 Setup of data

We provide the Omniglot dataset through Brightspace for you to download. Extract the data.zip such that you have a folder called `data` in the **same** directory as your Python files.

3 Methods

Below, we give a short summary of the method that we will investigate as well as some pointers to help you correctly implement the techniques.

3.1 Prototypical network

Prototypical network is a metric-based meta-learning techniques that computes predictions for new query input embedding $g_\theta(\hat{\mathbf{x}})$ by comparing it to class centroids $\mathbf{m}_n = \frac{1}{|X_n|} \sum_{\mathbf{x}_i \in X_n} g_\theta(\mathbf{x}_i)$ using a kernel $k_\theta(\hat{\mathbf{x}}, \mathbf{m}_n)$. The used kernel/similarity measure is the negative squared Euclidean distance, but other options exist. The class of the centroid closest to the query embedding $g_\theta(\hat{\mathbf{x}})$ is then predicted.

3.2 Reptile

Reptile is an optimization-based meta-learning technique that focuses on quick adaptation to new tasks by iteratively adjusting model parameters in the direction of task-specific gradients. It works by repeatedly fine-tuning a model's parameters in a single inner loop optimization based on task-specific gradients.

3.3 Model-agnostic meta-learning

Model-agnostic meta-learning, or MAML, aims to learn good initialization parameters θ such that new tasks can be learned within a few gradient update steps. This requires performing backpropagation through an optimization trajectory. In Pytorch, this can be a bit tricky as it needs to recognize that the task-specific parameters originate from the initialization. Below are some hints to help you with the implementation:

- Make a copy of the initial network weights by using `param.clone()`, where param is a tensor from the list of parameters. PyTorch then knows that the copy (called *fast weights*) originated from the initialization parameters. You can then adjust this copy using gradient update steps utilizing the `torch.autograd.grad()` and appropriate gradient descent with the inner learning rate (similarly to how it was performed with the *SGD* optimizer in the higher package).
- Make sure to pass these *fast weights* when computing predictions, i.e., `preds = network(inputs, weights=fast_weights)`
- Make sure to study this resource before implementing it. This should be helpful for the implementation of both first and second-order MAML.
- Do not call `loss.backward()` in the code other than the call that is already in there as this can interfere with the meta-gradient signals.

3.4 Hyperparameters

Use the following hyperparameters in your experiments, unless explicitly requested otherwise.

- inner learning rate: 0.4
- outer learning rate: 0.001
- meta-batch size: 1
- outer optimizer: Adam
- number of inner gradient updates: 1
- number of meta-iterations: 40 000
- validation interval: 500 episodes (meta-batches)
- backbone: convolutional neural network (Conv4 in `networks.py`)

4 Theory questions

Below are the questions that should be answered in your report. Answering these questions will also help you with the implementation of the methods.

1. Explain how from a given dataset with classes $\mathcal{Y}$, we can create meta-training, meta-validation, and meta-test tasks in an N-way k-shot manner. What for and when are the validation tasks used? What do we evaluate on the meta-test set?
2. When is the meta-gradient in MAML zero equal to zero? Interpret your findings: when does MAML update its initialization parameters θ and when does it not?
3. You have meta-trained MAML and ProtoNet on tasks from a single dataset. You now encounter a new task that is distant from the training task distribution. Which method do you expect to work better under the assumption that both techniques work equally well on tasks from the training distribution? Explain why.
4. MAML and Reptile differ in their underlying optimization procedures and the procedure for updating model parameters. In what scenarios would you expect each of the algorithms to outperform the other, and why?

5 Empirical questions

In your report, investigate the following empirically:

1. Show a plot of the meta-validation and meta-training losses over time in the 5-way, 1-shot setting (using 15 query examples/shots per class) first-order MAML, and second-order MAML. Interpret your findings.
2. Investigate the effect of the number of inner gradient update steps (T) on the performance of first-order and second-order MAML in a single plot for the 5-way 1-shot setting. Interpret your results.
3. Investigate the effect of increasing the number of classes per task on the performance of first- and second-order MAML. Interpret your results.
4. Investigate the effect of the meta-batch size on the performance of MAML. What do you find?

6 Provided materials

We provide you with the file `a2.zip` which contains the following files:

- `main.py`: The main script to run for experiments. You do **not** have to change any code here. This code performs meta-training, meta-validation, and meta-testing.
- `networks.py` Code that contains the two backbone networks: a feed-forward classifier and a convolutional one. In the experiments, we use the convolutional backbone. You do not need to change the file.

- `dataloaders.py` The file containing the data loader code (you should not change this file)
- `maml.py` The place to implement MAML. You only have to complete the apply function, which learns the task and makes predictions on the query inputs.

6.1 Setup

For this assignment, you are provided an accompanying `requirements.txt` file that can be used to install all of the necessary packages. After activating your environment you can run the command shown in Listing 1.

Listing 1: Code depicting how to install the required packages.

```
#!/bin/bash
$ pip install -r requirements.txt
```

7 Requirements/grading

The following requirements hold for this project:

- Do not plagiarize code or text. There will be a strict plagiarism check on both the code and the report that you submit.
- You will write a report **of at most 4 pages (excluding references) using the provided template**. You are not allowed to deviate from this template (it is also not allowed to change the margins or font size in the template). The report should answer all questions above. We use the following criteria for grading
  - Correctness
  - Completeness
  - Rigorous experimental design that is suited for answering the empirical research questions posed in this assignment.
  - Detailed and clear description of the experiments, rendering them reproducible.
  - Good data visualization (think carefully about how to represent results!!–what type of plot, aggregation methods, use of caption, use of legend, etc.)
  - Display of understanding: good analysis and interpretation of the experimental results. Clear and logical conclusions.
- You are **NOT** allowed to use existing libraries/repositories for the implementation of MAML with the exception of the imports we have already done in the provided files.
- The report should be concisely written and void of spelling and grammar mistakes (consider using the Grammarly Chrome extension).
- To make a submission, upload all your code and the report to BrightSpace through TurnitIn.
- The project deadline is due **11 December 2023 23:59 CET**.

8 Helpful resources

You may find the resources below useful:

- https://lilianweng.github.io/posts/2018-11-30-meta-learning/
- https://link.springer.com/content/pdf/10.1007/978-3-030-67024-5.pdf
- https://pytorch.org/tutorials/beginner/basics/intro.html
- http://luiz.hafemann.ca/libraries/2018/06/22/pytorch-doublebackprop/
- https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html
- https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

And of course, the original papers [2, 5, 4].

9 Tips

The experimentation phase is expected to take a considerable amount of time and resources. It would be highly advised to utilize the DSLab machines to run your experiments with the dedicated GPUs. Additionally, you can make use of `nohup` command to execute the runs in the background. Refer to Listing 2 for an example code utilizing the `nohup` command to run a python script. In case you need to terminate the execution you can use the example code of the Listing 3. In order to mitigate induced randomness, execute 5 repetitions for each run.

Listing 2: Example code for usage of nohup. This will write the output of the execution of the python script to the file *output.log*.

```
#!/bin/bash
$ nohup python /path/to/test.py > output.log &
```

Listing 3: Example code for terminating run in the background.

```
#!/bin/bash
$ ps ax | grep test.py
$ kill -9 <PID>
```

References

[1] Tutorial few-shot learning. https://www.borealisai.com/en/blog/tutorial-2-few-shot-learning-and-meta-learning-i/. Accessed: 04-19-2022.

[2] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, ICML'17, pages 1126–1135. PMLR, 2017.

[3] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[4] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[5] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. *Advances in neural information processing systems*, 30, 2017.