CS 330 Autumn 2022 Homework 2
Prototypical Networks and Model-Agnostic Meta-Learning

Due Monday October 24, 11:59 PM PST

SUNet ID: 06531892
Name: Leon Kloker
Collaborators: -

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Overview

In this assignment, you will experiment with two meta-learning algorithms, prototypical networks (protonets) [**?**] and model-agnostic meta-learning (MAML) [**?**], for few-shot image classification on the Omniglot dataset [**?**], which you also used for Homework 1. You will:

1. Implement both algorithms (given starter code).
2. Interpret key metrics of both algorithms.
3. Investigate the effect of task composition during protonet training on evaluation.
4. Investigate the effect of different inner loop adaptation settings in MAML.
5. Investigate the performance of both algorithms on meta-test tasks that have more support data than training tasks do.

Expectations

- We expect you to develop your solutions locally (i.e. make sure your model can run for a few training iterations), but to use GPU-accelerated training (e.g. Azure) for your results, since the maml training could take a while on CPU. **To change the training to happen on GPU, use our provided command line argument** `--device gpu` **when you run** `maml.py` **and** `protonet.py`**.**
- **Submit to Gradescope**
  1. the two python files in the submission folder, namely `protonet.py` and `maml.py`
  2. a `.pdf` report containing your responses
- You are welcome to use TensorBoard screenshots for your plots. Ensure that individual lines are labeled, e.g. using a custom legend, or by text in the figure caption.
- Figures and tables should be numbered and captioned.

Autograding

As in previous homework, we provide autograder for this assignment to facilitate your development. You can simply run:

```
python grader.py
```

to unit-test your implemented code. The maximum points you can see is 13 points, we also leave 19 points to the hidden cases, which you will see when you submit to Gradescope. This makes a total of 32 points for the coding section.

Preliminaries

Notation

- x: Omniglot image
- y: class label
- N (way): number of classes in a task
- K (shot): number of support examples per class
- Q: number of query examples per class
- c_n: prototype of class n
- f_θ: neural network parameterized by θ
- $\mathcal{T}_i$: task i
- $\mathcal{D}_i^{\text{tr}}$: support data in task i
- $\mathcal{D}_i^{\text{ts}}$: query data in task i
- B: number of tasks in a batch
- $\mathcal{J}(\theta)$: objective function parameterized by θ

Part 1: Prototypical Networks (Protonets) [?]

Algorithm Overview

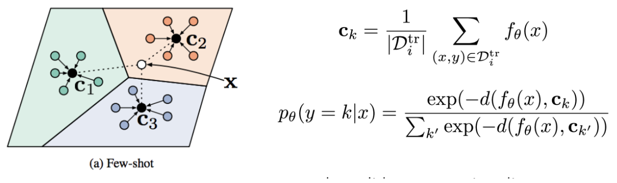

$\mathbf{c}_1$
$\mathbf{c}_2$
$\mathbf{c}_3$
$\mathbf{x}$
(a) Few-shot
$\mathbf{c}_k = \frac{1}{|\mathcal{D}_i^{\mathrm{tr}}|} \sum_{(x,y)\in\mathcal{D}_i^{\mathrm{tr}}} f_\theta(x)$
$p_\theta(y=k|x) = \frac{\exp(-d(f_\theta(x),\mathbf{c}_k))}{\sum_{k'}\exp(-d(f_\theta(x),\mathbf{c}_{k'}))}$

Figure 1: Prototypical networks in a nutshell. In a 3-way 5-shot classification task, the class prototypes c_1, c_2, c_3 are computed from each class's support features (colored circles). The prototypes define decision boundaries based on Euclidean distance. A query example x is determined to be class 2 since its features (white circle) lie within that class's decision region.

As discussed in lecture, the basic idea of protonets is to learn a mapping $f_\theta(\cdot)$ from images to features such that images of the same class are close to each other in feature space. Central to this is the notion of a *prototype*

$$c_n = \frac{1}{K} \sum_{(x,y)\in\mathcal{D}_i^{\mathrm{tr}}:y=n} f_\theta(x), \tag{1}$$

i.e. for task i, the prototype of the n-th class c_n is defined as the mean of the K feature vectors of that class's support images. To classify some image x, we compute a measure of distance d between $f_\theta(x)$ and each of the prototypes. We will use the squared Euclidean distance:

$$d(f_\theta(x), c_n) = \|f_\theta(x) - c_n\|_2^2. \tag{2}$$

We interpret the negative squared distances as logits, or unnormalized log-probabilities, of x belonging to each class. To obtain the proper probabilities, we apply the softmax operation:

$$p_\theta(y=n|x) = \frac{\exp(-d(f_\theta(x), c_n))}{\sum_{n'=1}^{N} \exp(-d(f_\theta(x), c_{n'}))}. \tag{3}$$

Because the softmax operation preserves ordering, the class whose prototype is closest to $f_\theta(x)$ is naturally interpreted as the most likely class for x. To train the model to generalize, we compute prototypes using support data, but minimize the negative log likelihood of the query data

$$\mathcal{J}(\theta) = \mathbb{E}_{\mathcal{T}_i\sim p(\mathcal{T}),(\mathcal{D}_i^{\mathrm{tr}},\mathcal{D}_i^{\mathrm{ts}})\sim\mathcal{T}_i} \left[\frac{1}{NQ} \sum_{(x^{\mathrm{ts}},y^{\mathrm{ts}})\in\mathcal{D}_i^{\mathrm{ts}}} -\log p_\theta(y = y^{\mathrm{ts}}|x^{\mathrm{ts}}) \right]. \tag{4}$$

Notice that this is equivalent to using a cross-entropy loss.

We optimize θ using Adam [**?**], an off-the-shelf gradient-based optimization algorithm. As is standard for stochastic gradient methods, we approximate the objective (4) with Monte Carlo estimation on minibatches of tasks. For one minibatch with B tasks, we have

$$\mathcal{J}(\theta) \approx \frac{1}{B} \sum_{i=1}^{B} \left[\frac{1}{NQ} \sum_{(x^{\text{ts}}, y^{\text{ts}}) \in \mathcal{D}_i^{\text{ts}}} - \log p_\theta(y = y^{\text{ts}} | x^{\text{ts}}) \right]. \tag{5}$$

Problems

1. **Analysis of No Required Shuffling**

   (a) **[3 points (Written)]** We have provided you with `omniglot.py`, which contains code for task construction and data loading. Recall that for training black-box meta-learners in the previous homework we needed to shuffle the query examples in each task. This is not necessary for training protonets. Explain why.

   Blackbox models that process the support and query set as one long sequence might pick up on the fact that the first query example always belongs to class 0, the second one to class 1 and so on. Hence, the model does not learn features to actually classify unseen samples coming in abritrary order. In the case of protonet, however, each sample from the query set is processed via an individual forward pass such that there is no real notion of ordering that protonet could learn from.

2. **Implementation**

   (a) **[8 points (Coding)]** In the `protonet.py` file, complete the implementation of the `ProtoNet._step` method, which computes (5) along with accuracy metrics. Pay attention to the inline comments and docstrings.

   Assess your implementation on 5-way 5-shot Omniglot. To do so, run

   ```
   python protonet.py
   ```

   with the appropriate command line arguments. These arguments have defaults specified in the file. To specify a non-default value for an argument, use the following syntax:

   ```
   python protonet.py --argument1 value1 --argument2 value2
   ```

   Use 15 query examples per class per task. Depending on how much memory your GPU has, you may want to adjust the batch size. Do not adjust the learning rate from its default of 0.001.

   As the model trains, model checkpoints and TensorBoard logs are periodically saved to a `log_dir`. The default `log_dir` is formatted from the arguments, but this can be overriden. You can visualize logged metrics by running

   ```
   tensorboard --logdir logs/
   ```

   and navigating to the displayed URL in a browser. If you are running on a remote computer with server capabilities, use the `--bind_all` option to expose the web app to the network. Alternatively, consult the Azure guide for an example of how to tunnel/port-forward via SSH.

   To resume training a model starting from a checkpoint at `{some_dir}/state{some_step}.pt`, run

   ```
   python protonet.py --log_dir some_dir --checkpoint_step some_step
   ```

   If a run ended because it reached `num_train_iterations`, you may need to increase this parameter.

   (b) **[3 points (Plots)]** Submit a plot of the validation query accuracy over the course of training.
   Hint: you should obtain a query accuracy on the validation split of at least 99%.

   Your plot goes here.

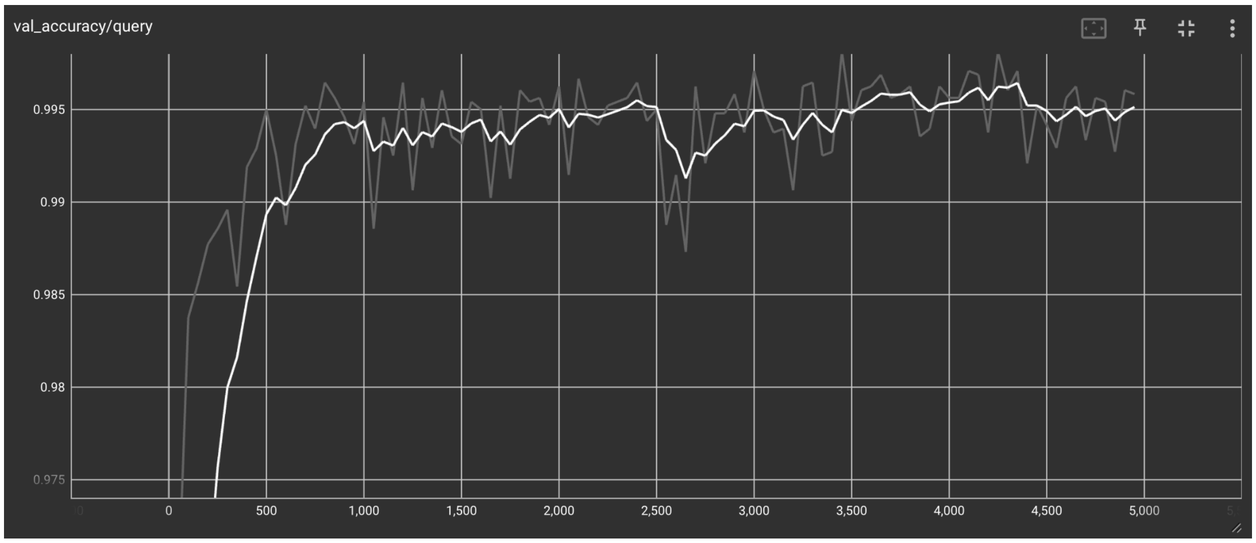


Figure 2: Validation accuracy of the query set.

(c) **Further Analysis** Four accuracy metrics are logged. For the above run, examine these in detail to reason about what the algorithm is doing.

(a) **[3 points (Written)]** Is the model placing support examples of the same class close together in feature space or not? Support your answer by referring to specific accuracy metrics.

Yes, the model learns to cluster examples (both support and query) close together in feature space in order to minimize the distance between them. This is due to the fact that maximizing the prediction probability for the actual class also corresponds to minimizing the logits of all the other classes before feeding them into the softmax function. Hence, the distance to the other classes is maximized and the distance to the own class simultaneously minimized, which can be done by mapping all samples from the same class to the same, distinct region in feature space. The high train and validation accuracy on the support set 3 corroborates this statement as consequently the support examples are almost always closer to their center than to the center of any other class (hence they are clustered).

(b) **[3 points (Written)]** Is the model generalizing to new tasks? If not, is it overfitting or underfitting? Support your answer by referring to specific accuracy metrics.

The model generalizes very well to unseen data as the accuracy on query sets (also on the support sets) of validation tasks is equally high as on the training tasks. This can be seen in plot 2 of the query validation accuracy.

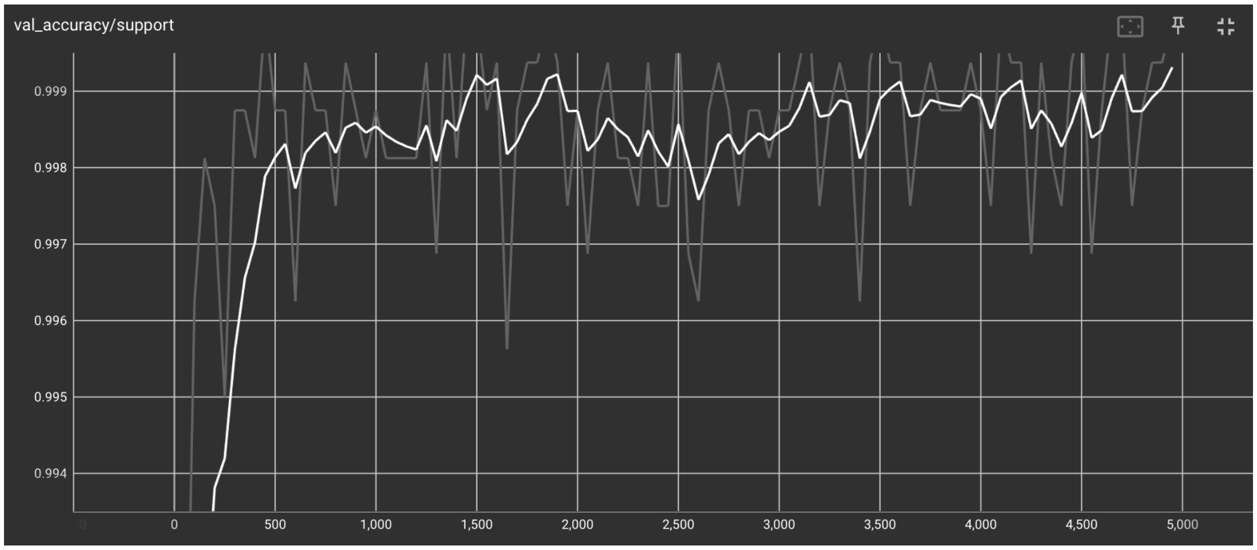


Figure 3: Validation accuracy of the support set.

(d) **Comparison** We will now compare different settings at training time. Train on 5-way 1-shot tasks with 15 query examples per task.

i. **[2 points (Written)]** Compare your two runs (5-way 1-shot training and 5-way 5-shot training) by assessing test performance on 5-way 1-shot tasks. To assess a trained model on test tasks, run

```
python protonet.py --test
```

appropriately specifying `log_dir` and `checkpoint_step`. Submit a table of your results with 95% confidence intervals.

	5-shot	1-shot
Accuracy	0.985	0.984
95% confidence	0.002	0.002

Table 1: Performance of both models on the 5-way 1-shot test tasks. Even though the 5-way 5-shot protonet wasn't trained on 1-shot tasks, it still performs really well in the 1-shot test case, even slightly outperforming the 1-shot model.

ii. **[2 points (Written)]** How did you choose which checkpoint to use for testing for each model?

The checkpoint with the highest validation query accuracy was chosen in order to compare the models which generalize the best to unseen data on the 5-way 1-shot test task.

iii. **[2 points (Written)]** Is there a significant difference in the test performance on 5-way 1-shot tasks? Explain this by referring to the protonets algorithm.

The performance of both models on the 5-way 1-shot test tasks is very similar. During training both models learn to map inputs from the same class to the same, class-specific region in feature space, such that the resulting Voronoi regions of the support centroids separate the different classes. This is done regardless of the amount of support examples the network receives. Thus, as both networks try to learn a similar mapping to feature space, their performances are very alike.

Part 2: Model-Agnostic Meta-Learning (MAML) [?]

Algorithm Overview

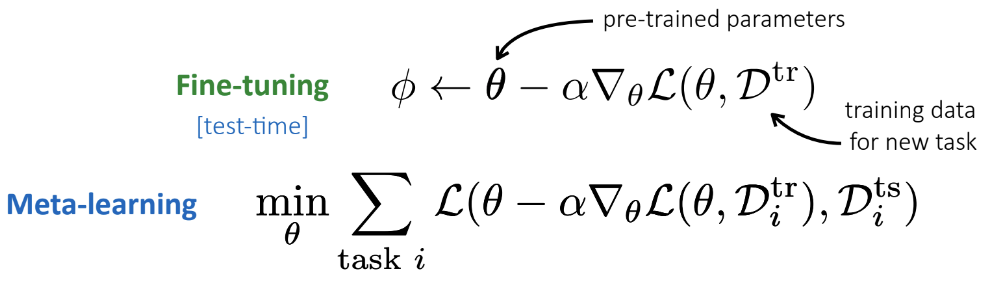

pre-trained parameters
Fine-tuning
[test-time]
$\phi \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}^{\text{tr}})$
training data for new task
Meta-learning
$\min_\theta \sum_{\text{task } i} \mathcal{L}(\theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}_i^{\text{tr}}), \mathcal{D}_i^{\text{ts}})$

Figure 4: MAML in a nutshell. MAML tries to find an initial parameter vector θ that can be quickly adapted via task gradients to task-specific optimal parameter vectors.

As discussed in lecture, the basic idea of MAML is to meta-learn parameters θ that can be quickly adapted via gradient descent to a given task. To keep notation clean, define the loss $\mathcal{L}$ of a model with parameters ϕ on the data $\mathcal{D}_i$ of a task $\mathcal{T}_i$ as

$$\mathcal{L}(\phi, \mathcal{D}_i) = \frac{1}{|\mathcal{D}_i|} \sum_{(x^j, y^j) \in \mathcal{D}_i} - \log p_\phi(y = y^j | x^j) \tag{6}$$

Adaptation is often called the *inner loop*. For a task $\mathcal{T}_i$ and L inner loop steps, adaptation looks like the following:

$$\begin{aligned} \phi^1 &= \phi^0 - \alpha \nabla_{\phi^0} \mathcal{L}(\phi^0, \mathcal{D}_i^{\text{tr}}) \\ \phi^2 &= \phi^1 - \alpha \nabla_{\phi^1} \mathcal{L}(\phi^1, \mathcal{D}_i^{\text{tr}}) \\ &\vdots \\ \phi^L &= \phi^{L-1} - \alpha \nabla_{\phi^{L-1}} \mathcal{L}(\phi^{L-1}, \mathcal{D}_i^{\text{tr}}) \end{aligned} \tag{7}$$

where we have defined $\theta = \phi^0$.

Notice that only the support data is used to adapt the parameters to ϕ^L. (In lecture, you saw ϕ^L denoted as ϕ_i.) To optimize θ in the *outer loop*, we use the same loss function (6) applied on the adapted parameters and the query data:

$$\mathcal{J}(\theta) = \mathbb{E}_{\mathcal{T}_i \sim p(\mathcal{T}), (\mathcal{D}_i^{\text{tr}}, \mathcal{D}_i^{\text{ts}}) \sim \mathcal{T}_i} \left[\mathcal{L}(\phi^L, \mathcal{D}_i^{\text{ts}}) \right] \tag{8}$$

For this homework, we will further consider a variant of MAML [?] that proposes to additionally learn the inner loop learning rates α. Instead of a single scalar inner learning rate for all parameters, there is a separate scalar inner learning rate for each parameter group (e.g. convolutional kernel, weight matrix, or bias vector). Adaptation remains the same as in vanilla MAML except with appropriately broadcasted multiplication between the inner loop learning rates and the gradients with respect to each parameter group.

The full MAML objective is

$$\mathcal{J}(\theta, \alpha) = \mathbb{E}_{\mathcal{T}_i \sim p(\mathcal{T}), (\mathcal{D}_i^{\text{tr}}, \mathcal{D}_i^{\text{ts}}) \sim \mathcal{T}_i} \left[\mathcal{L}(\phi^L, \mathcal{D}_i^{\text{ts}}) \right] \tag{9}$$

Like before, we will use minibatches to approximate (9) and use the Adam optimizer.

Problems

(a) **Implementation**

i. **[24 points (Coding)]** In the `maml.py` file, complete the implementation of the `MAML._inner_loop` and `MAML._outer_step` methods. The former computes the task-adapted network parameters (and accuracy metrics), and the latter computes the MAML objective (and more metrics). Pay attention to the inline comments and docstrings.

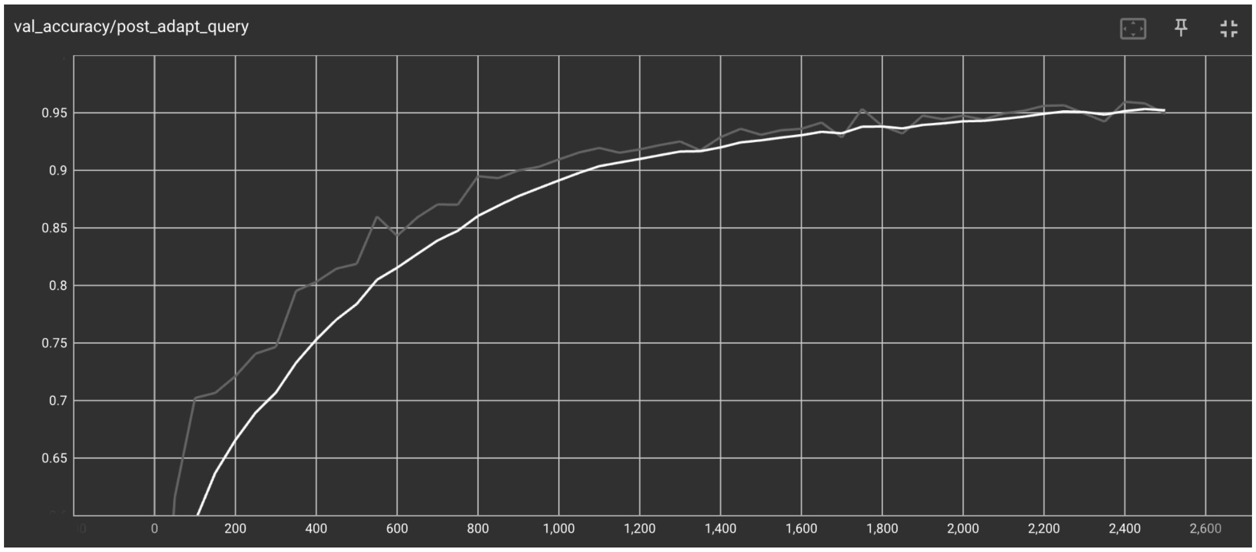
Hint: the simplest way to implement `_inner_loop` involves using `autograd.grad`. Check the documentation here on how to use and call the function. In essence, the function computes and returns the sum of gradients of outputs with respect to the inputs. Compared with the PyTorch `backward` function which we typically deal with, `autograd.grad` is a non-mutable function and will not accumulate the gradients on the model parameters.
Hint: to understand how to use the Boolean `train` argument of `MAML._outer_step`, read the documentation for the `create_graph` argument of `autograd.grad`.

Assess your implementation of vanilla MAML on 5-way 1-shot Omniglot. Comments from the previous part regarding arguments, checkpoints, TensorBoard, resuming training, and testing all apply. Use 1 inner loop step with a **fixed** inner learning rate of 0.4. Use 15 query examples per class per task. Do not adjust the outer learning rate from its default of 0.001. Note that MAML generally needs more time to train than protonets. Run the command:

```
python maml.py
```

Unfortunately, the quota increase on Azure to get a GPU instance took more than 2 days such that I wasn't able to run all experiments up to the desired number of iterations. Instead, I set up a CPU virtual machine on which each run takes significantly more time. Hence, some plots have runs with only about 3000 instead of 15000 iterations. In order to be able to answer all the questions, I looked at plots of classmates of complete runs.

ii. **[3 points (Plots)]** Submit a plot of the validation post-adaptation query accuracy over the course of training.
Hint: you should obtain a query accuracy on the validation split of at least 97%.



(b) **Analysis** Six accuracy metrics are logged. Examine these in detail to reason about what MAML is doing.

(a) **[10 points (Written)]** State and explain the behavior of the `train_pre_adapt_support` and `val_pre_adapt_support` accuracies. Your answer should explicitly refer to the **task sampling process**.
Hint: consult the `omniglot.py` file. Your explanation should explicitly refer to i) the task format, ii) the model's pre-adaptation parameters, and iii) how images in each task are labeled.

Both accuracies are around 0.2 on average. This makes sense as we consider 5-way classification tasks where random guessing leads to a 0.2 accuracy. This is due to the fact that the pre-adaption parameters are not adapted to the task at hand at all. Moreover, as in each task the class labels correspond to entirely different classes (due to our sampling of tasks in the meta-learning setting), the pre-adapted model does not even know which output label corresponds to which class. Hence, the model can't do better than 0.2 accuracy on average.

(b) **[5 points (Written)]** Compare the `train_pre_adapt_support` and `train_post_adapt_support` accuracies. What does this comparison tell you about the model? Repeat for the corresponding `val` accuracies.

Before adaption, we are in the setting I just outlined for the previous exercise where the model can't possibly perform better than random-guessing on average. The support accuracy post-adaption is either one or very close to one. This is reasonable as the inner loop adjusts the parameters of the model exactly in such a way that the cross-entropy loss when classifying the support samples is minimized. Moreover, during the outer loop the pre-adaption parameters are chosen in such a way that running the inner loop leads to a very high accuracy on the support samples.

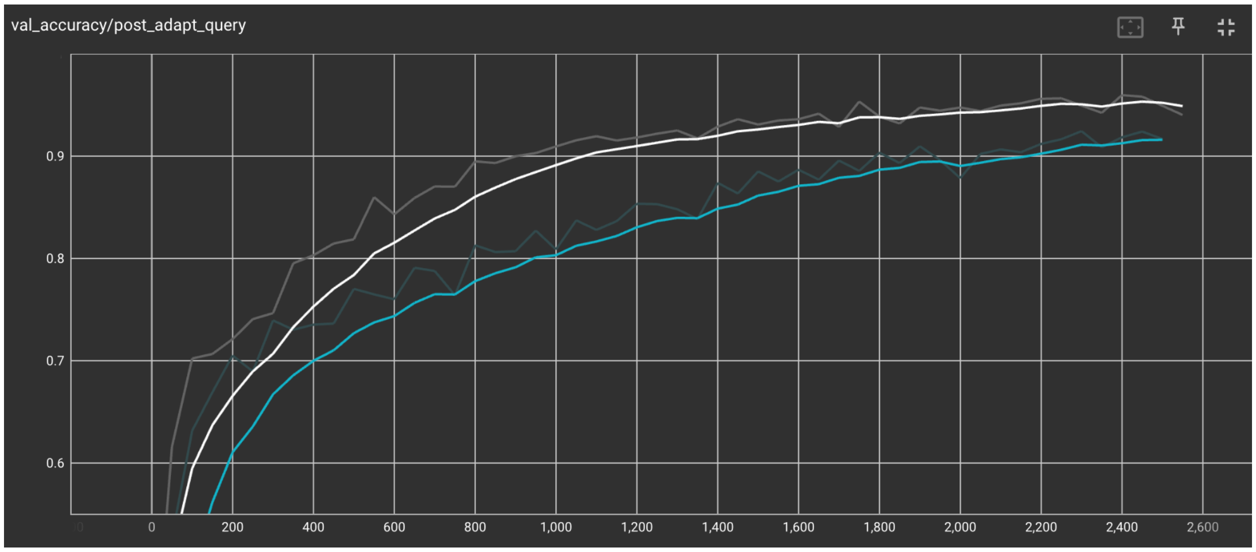
(c) **[5 points (Written)]** Compare the `train_post_adapt_support` and `train_post_adapt_query` accuracies. What does this comparison tell you about the model? Repeat for the corresponding `val` accuracies.

The post-adaption support accuracy is pretty much one, as expected. Moreover, the post-adaption accuracy on the query samples is increasing up to 95% during training. Thus, the inner loop doesn't just overfit on the support samples but instead adjusts the parameters in a way that they generalize well to unseen data.

(c) **Experiments** Try MAML with the same hyperparameters as above except for a fixed inner learning rate of 0.04.

(a) **[3 points (Plots)]** Submit a plot of the validation post-adaptation query accuracy over the course of training with the two inner learning rates (0.04, 0.4). Run the command:

```
python maml.py --inner_lr 0.04
```



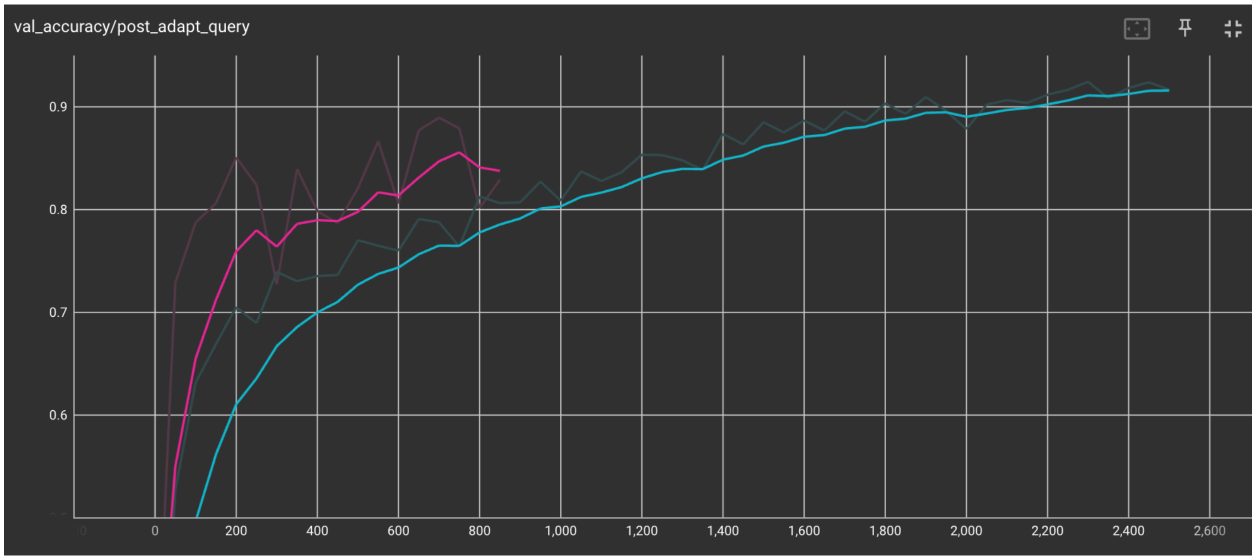
(b) **[2 points (Written)]** What is the effect of lowering the inner learning rate on (outer-loop) optimization and generalization?

As visible (blue curve is lr 0.04), lowering the inner learning rate leads to slower convergence and possibly a worse optimum even after training for longer. With a small learning rate, the pre-adaption parameters need to be close in parameter space to the optimal parameters for each of the tasks. It might be more difficult to find such a set of pre-adaption parameters than in the case of a larger inner loop learning rate where the pre-adaption parameters can be further away from each task-specific optimum. Morever, if the tasks are too different from each other, there might no be a set of pre-adaption parameters such that a gradient step with small learning rate gets close to the optimal task specific parameters, leading to an overall worse performance.

(d) **Experiments** Try MAML with a fixed inner learning rate of 0.04 for 5 inner loop steps.

(a) **[3 points (Plots)]** Submit a plot of the validation post-adaptation query accuracy over the course of training with the two number of inner loop steps $(1, 5)$ with inner learning rate 0.04. Run the command:

```
python maml.py --inner_lr 0.04 --num_inner_steps 5
```



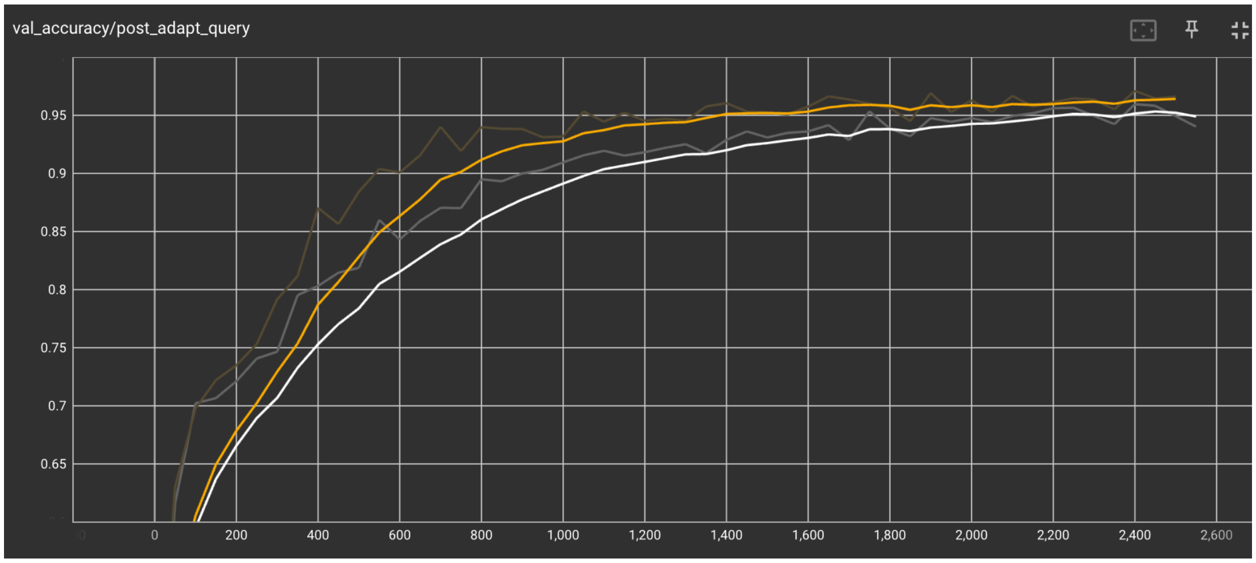
(b) **[2 points (Written)]**What is the effect of increasing the number of inner loop steps on (outer-loop) optimization and generalization?

A larger number of inner loop optimization steps leads to a higher validation query accuracy (checked on plot of a full run). More steps alleviate the problem outlined in the previous exercise allowing the parameter adjustment to span larger differences between pre-adaption and task specific optimal parameters. However, it also leads to harder optimization problem and longer training times as the gradient descent steps are expensive in terms of GPU usage and introduce mroe noise into the training process. The longer training time is visible in the plot as the training with 5 inner steps only has about a third of the iterations as with one inner step after the same time.

(e) **Experiments** Try MAML with learning the inner learning rates. Initialize the inner learning rates with 0.4.

(a) **[3 points (Plots)]** Submit a plot of the validation post-adaptation query accuracy over the course of training for learning and not learning the inner learning rates, initialized at 0.4. Run the command:

```
python maml.py --learn_inner_lrs
```



(b) **[2 points (Written)]** What is the effect of learning the inner learning rates on (outer-loop) optimization and generalization?

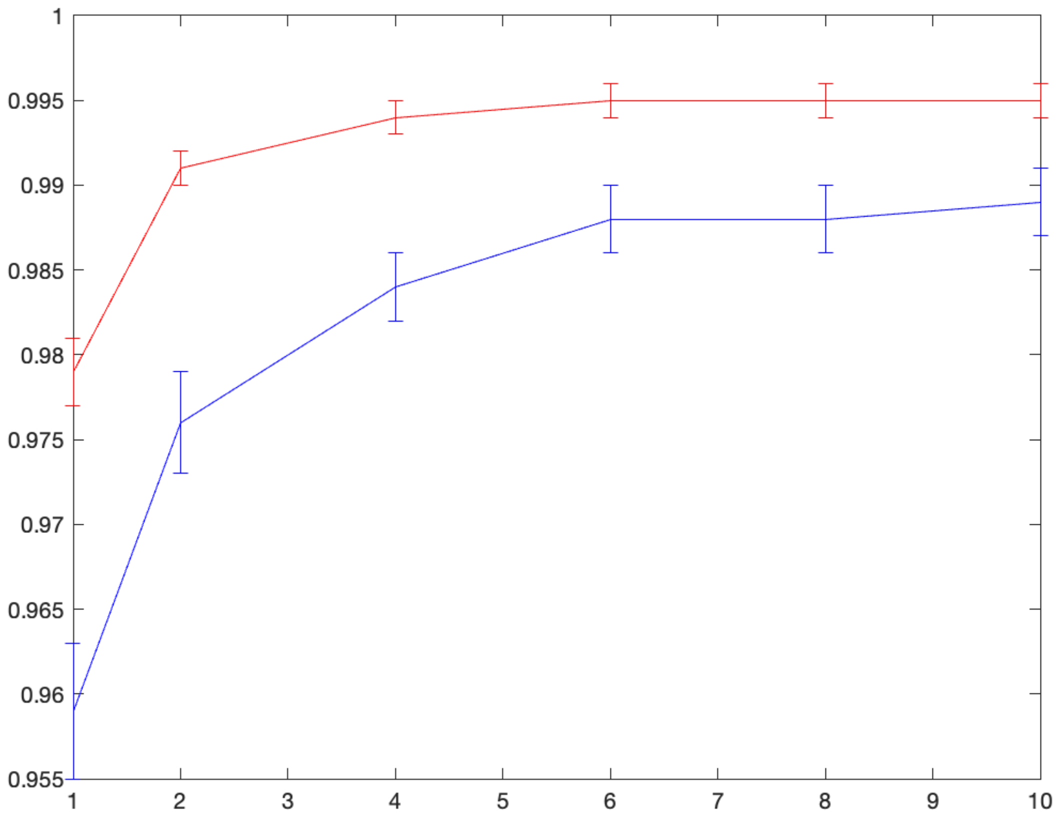
The performance during training increases faster (orange curve with learned lrs) and the best query validation accuracy is also slightly better than without learning rate adaptation (not visible in this plot but seen of plot of full run). This makes sense as maml can now learn by itself how far the task specific optimal parameters are away from each other and which step size is required to reach them from the pre-adaption parameters. As this introduces new parameters, the training become slighlty more computationally expensive, but the difference is not large.

Part 3: More Support Data at Test Time

In practice, we usually have more than 1 support example at test time. Hence, one interesting comparison is to train both algorithms with 5-way 1-shot tasks (as you've already done) but assess them using more shots.

(a) **Experiment** Use the protonet trained with 5-way 1-shot tasks, and the MAML trained with **learned** inner learning rates initialized at 0.4. Try $K = 1, 2, 4, 6, 8, 10$ at test time. Use $Q = 10$ for all values of K. **Please closely check** `protonet.py` **and** `maml.py` **and the commands we provided in above questions on how to set these hyper-parameters with command line arguments**.

i. **[10 points (Plots)]** Submit a plot of the test accuracies for the two models over these values of K with the 95% confidence intervals as error bars or shaded regions.



ii. **[5 points (Written)]** How well is each model able to use additional data in a task without being explicitly trained to do so?

Both protonet (red) and maml (blue) seem to benefit from more support data. For protonet, mroe support data leads to more informed or general prototypes for each class. For maml, more support samples means more data about the task at hand to specifically optimize the parameters and also less noisy gradients in the inner loop. Here, protonet also performs slightly better than maml.

Part 4: Meta-learning for Real Dataset

In the previous homework and this homework, you have been experimenting with the Omniglot dataset. In this section, you are going to run your implemented Prototype network on the TDC Metabolism dataset [**?**], a real bio-related dataset used to predict compound properties. In TDC Metabolism, the authors select 8 sub-datasets related to drug metabolism from the whole TDC dataset [**?**], including CYP P450 2C19/2D6/3A4/1A2/2C9 Inhibition, CYP2C9/CYP2D6/CYP3A4 Substrate. The aim of each dataset is to predict whether each drug compound has the corresponding property. Correspondingly, the input to your Protonet is going to be a vector of molecule features. Take a look at the referenced paper if you are interested.

As the first step, please download the dataset here. Next, run the following command (can be found in `run_bio.sh`):

```
python3 run_metabolism.py --num_way 2 --num_support 5 --num_query 10 --batch_size
4 --num_train_iterations 8000 --learning_rate 0.0005 --datadir [DIR] --device
gpu
```

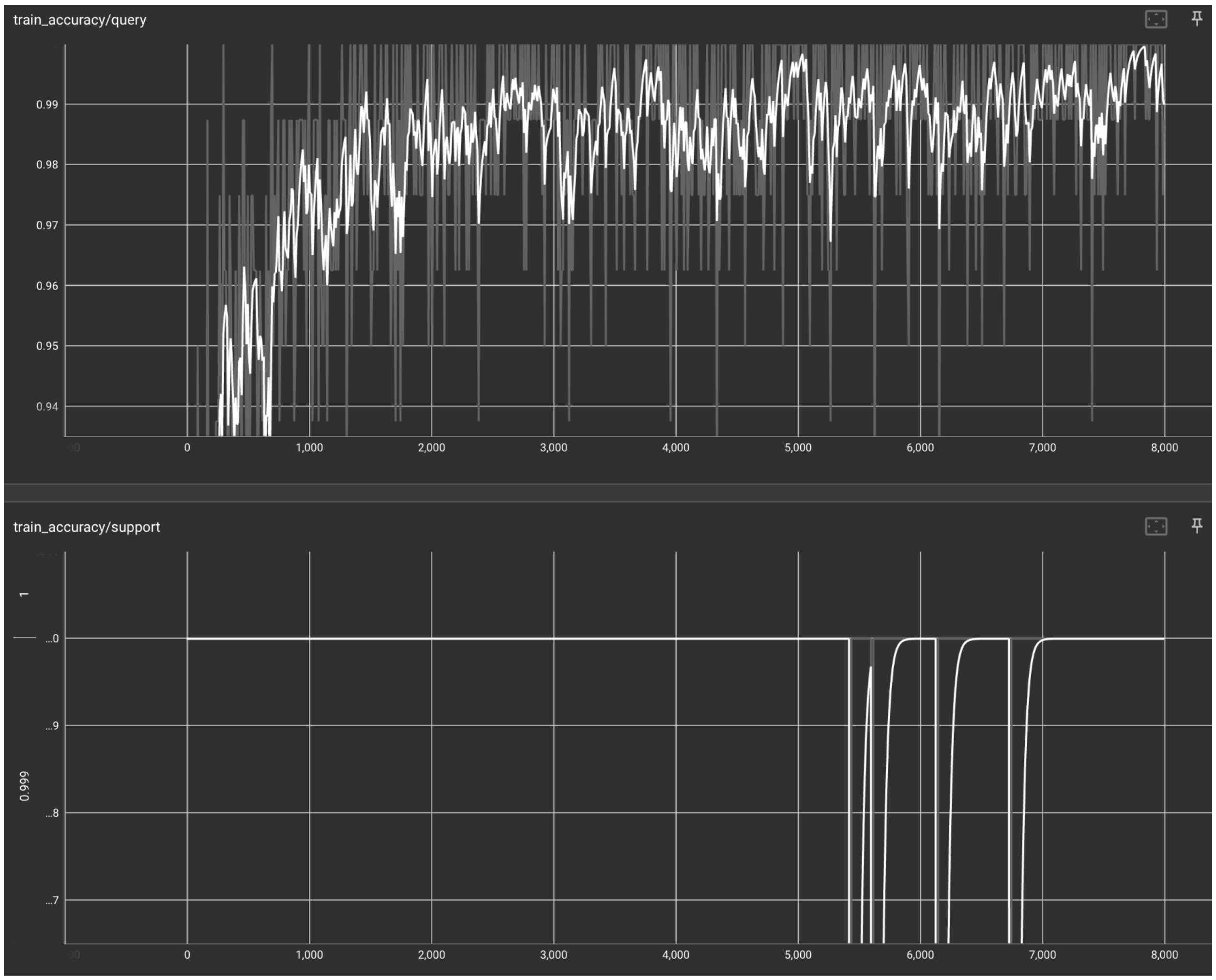
Replace [DIR] with the folder path containing your downloaded dataset.

(a) **[3 points (Written and Plots)]**Attach a screenshot of Tensorboard logs and report your final printed validation query accuracy with ci95(confidence interval 95%) here:

The final validation accuracy of the query set post-adaption is 76% with a 95% confidence interval of 0.004.

A Note

You may wonder why the performance of these implementations don't match the numbers reported in the original papers. One major reason is that the original papers used a different version of Omniglot few-shot classification, in which multiples of 90° rotations are applied to each image to obtain 4 times the total number of images and characters. Another reason is that these implementations are designed to be pedagogical and therefore straightforward to implement from equations and pseudocode as well as trainable with minimal hyperparameter tuning. Finally, with our



use of batch statistics for batch normalization during test (see code), we are technically operating in the *transductive* few-shot learning setting.

References