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

Due Monday October 24, 11:59 PM PST

SUNet ID:
Name: Seok, Jeongeum
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.
- Submit to Gradescope
    1. a `.zip` file containing your modified version of `hw2/starter/`
    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.

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

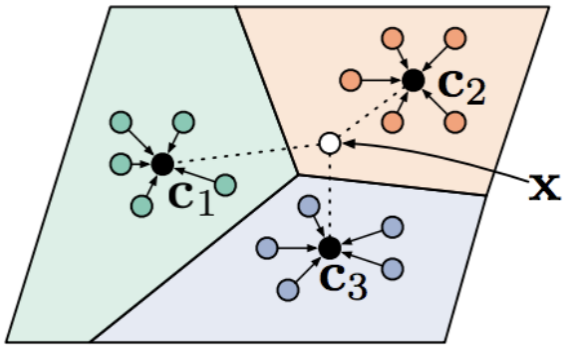


Figure 1: Prototypical networks in a nutshell. In a 3-way 5-shot classification task, the class prototypes c_1, c_2, c_3 are computed from each class's support features (colored circles). The prototypes define decision boundaries based on Euclidean distance. A query example x is determined to be class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

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

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

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

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. We have provided you with `omniglot.py`, which contains code for task construction and data loading.

   (a) (5 pt) 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.

   Shuffling examples does not change the prototype since it is represented as sum which is commutative.

2. 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/
   ```

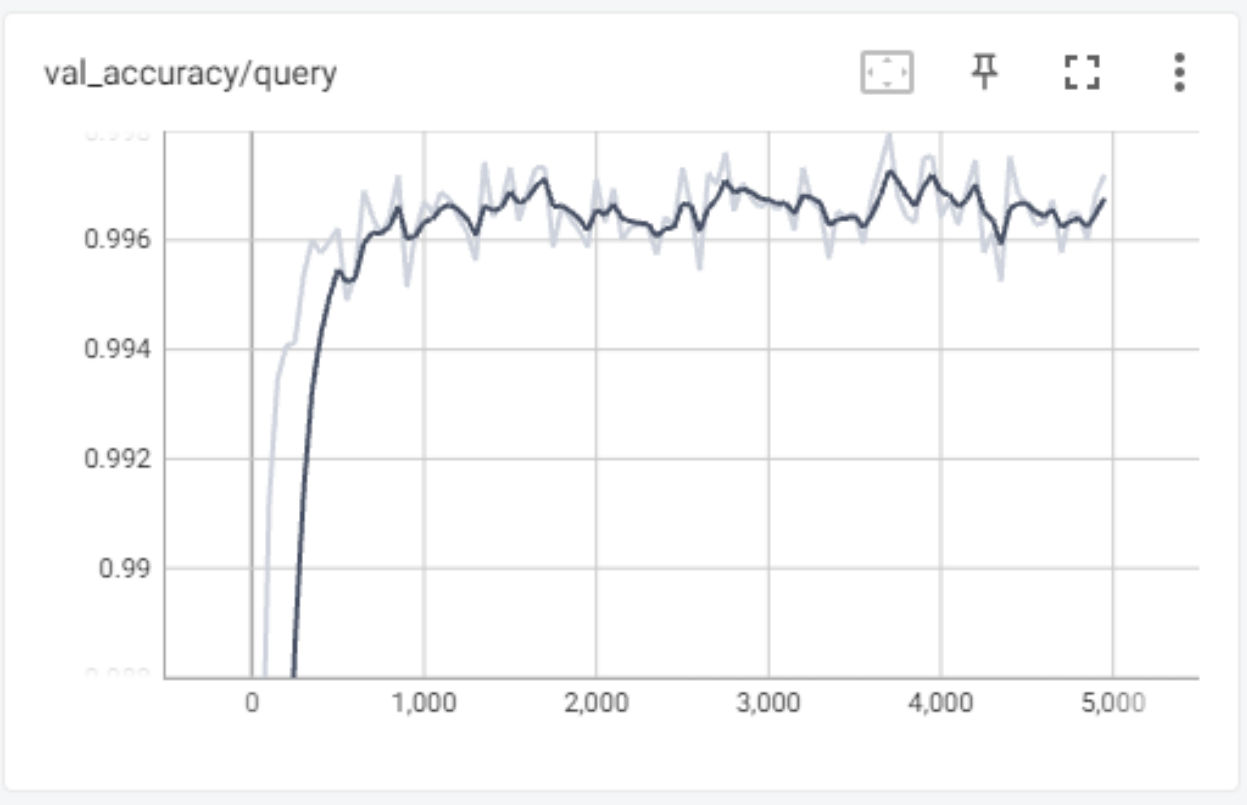
   and navigating to the displayed URL in a browser. If you are running on a remote computer with server capabilities, use the `--bind_all` option to expose the web app to the network. Alternatively, consult the Azure guide for an example of how to tunnel/port-forward via SSH.

   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.

(a) (20 pt) 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%.



3. 4 accuracy metrics are logged. For the above run, examine these in detail to reason about what the algorithm is doing.

   (a) (5 pt) 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.

   Model is placing support examples of the same class close together in feature space according to train support accuracy.

   (b) (5 pt) Is the model generalizing to new tasks? If not, is it overfitting or underfitting? Support your answer by referring to specific accuracy metrics.

   Model is generalizing to new tasks according to validation query accuracy.

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

   (a) (3 pt) 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.

train	5-way 1-shot	5-way 5-shot
mean	0.986	0.987
95% confidence interval	0.002	0.002

   (b) (2 pt) How did you choose which checkpoint to use for testing for each model?

   I chose the latest checkpoint because that will be the most fitted one.

   (c) (5 pt) Is there a significant difference in the test performance on 5-way 1-shot tasks? Explain this by referring to the protonets algorithm.

   No, computing the prototype is almost the same as choosing one datapoint.

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

Algorithm Overview

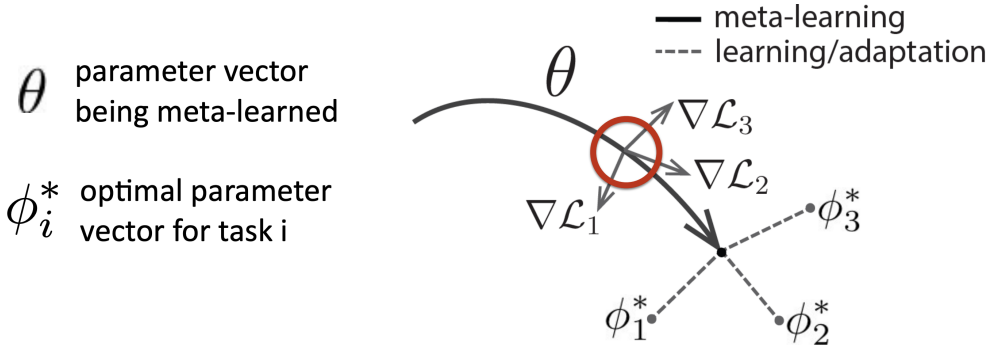


Figure 2: MAML in a nutshell. MAML tries to find an initial parameter vector θ that can be quickly adapted via task gradients to task-specific optimal parameter vectors.

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

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

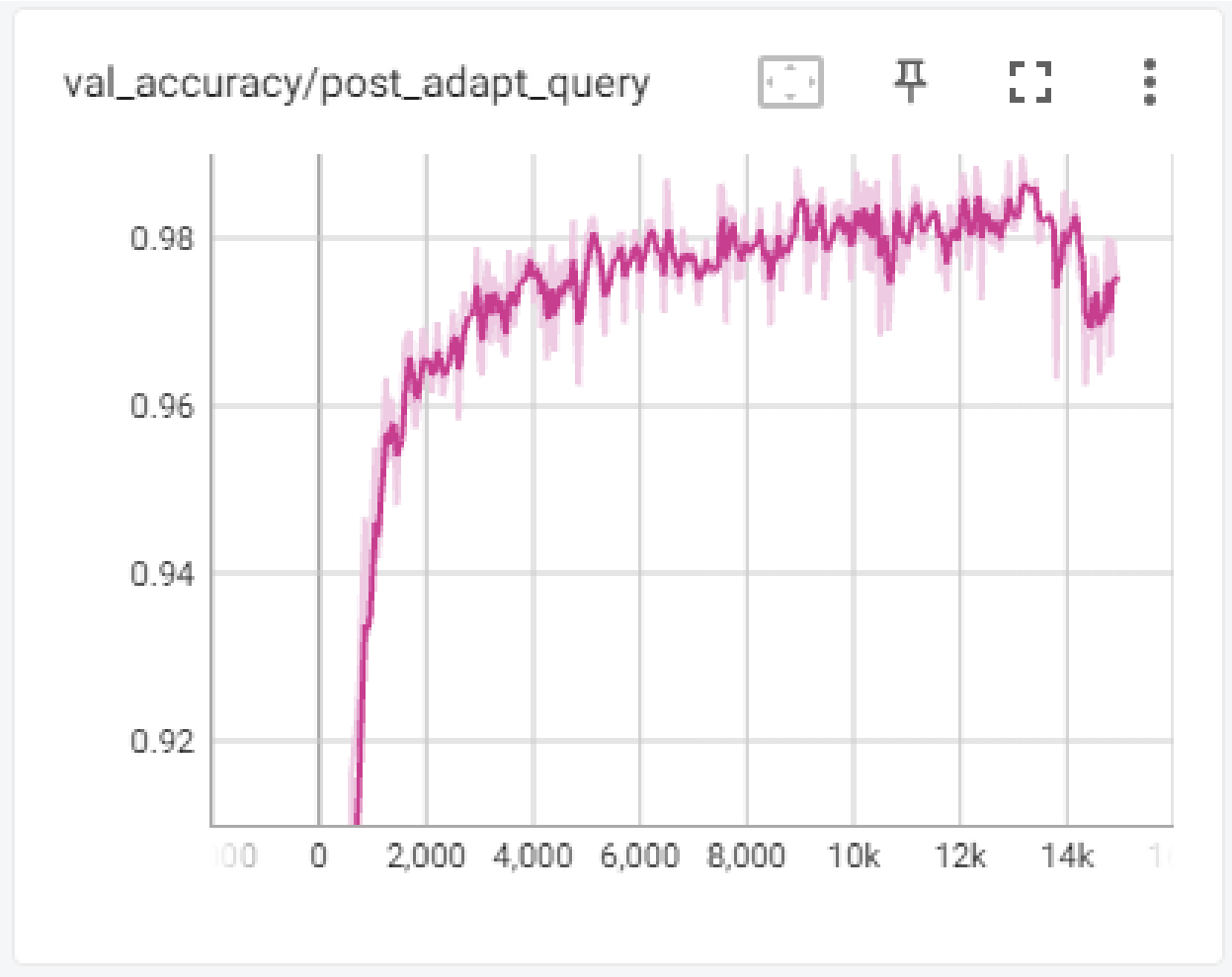
Like before, we will use minibatches to approximate (9) and use the Adam optimizer.

Problems

1. 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`.
   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.

(a) (20 pt) 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 96%.



2. 6 accuracy metrics are logged. Examine these in detail to reason about what MAML is doing.

   (a) (10 pt) 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.

   Pre-adapt accuracies are around 0.2 because tasks are randomly sampled.

   (b) (5 pt) 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.

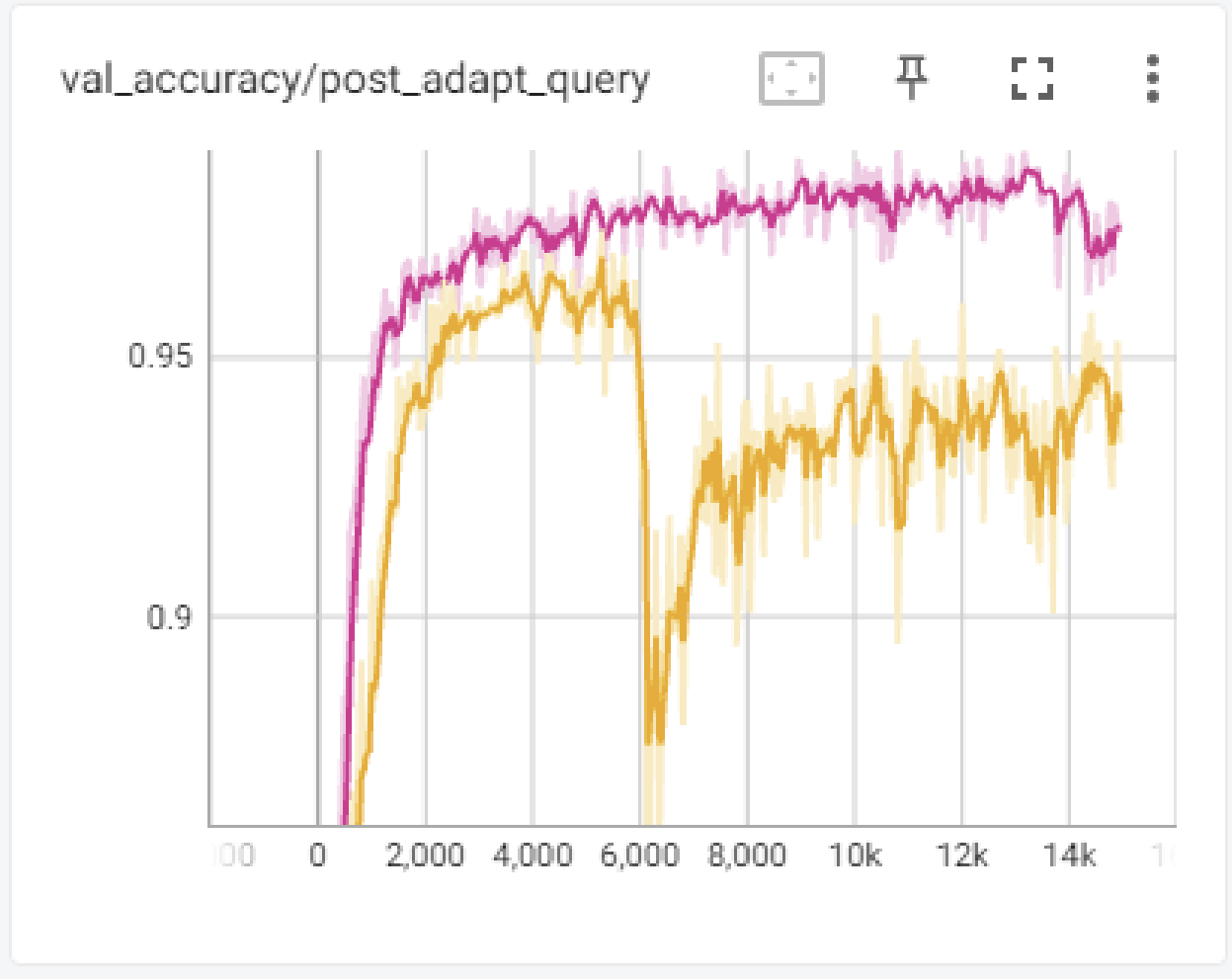
   The model has adapted to the task.

   (c) (5 pt) 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 model has adapted to the task but did not overfit to data.

3. Try MAML with the same hyperparameters as above except for a fixed inner learning rate of 0.04.

   (a) (3 pt) Submit a plot of the validation post-adaptation query accuracy over the course of training with the two inner learning rates $(0.04, 0.4)$.

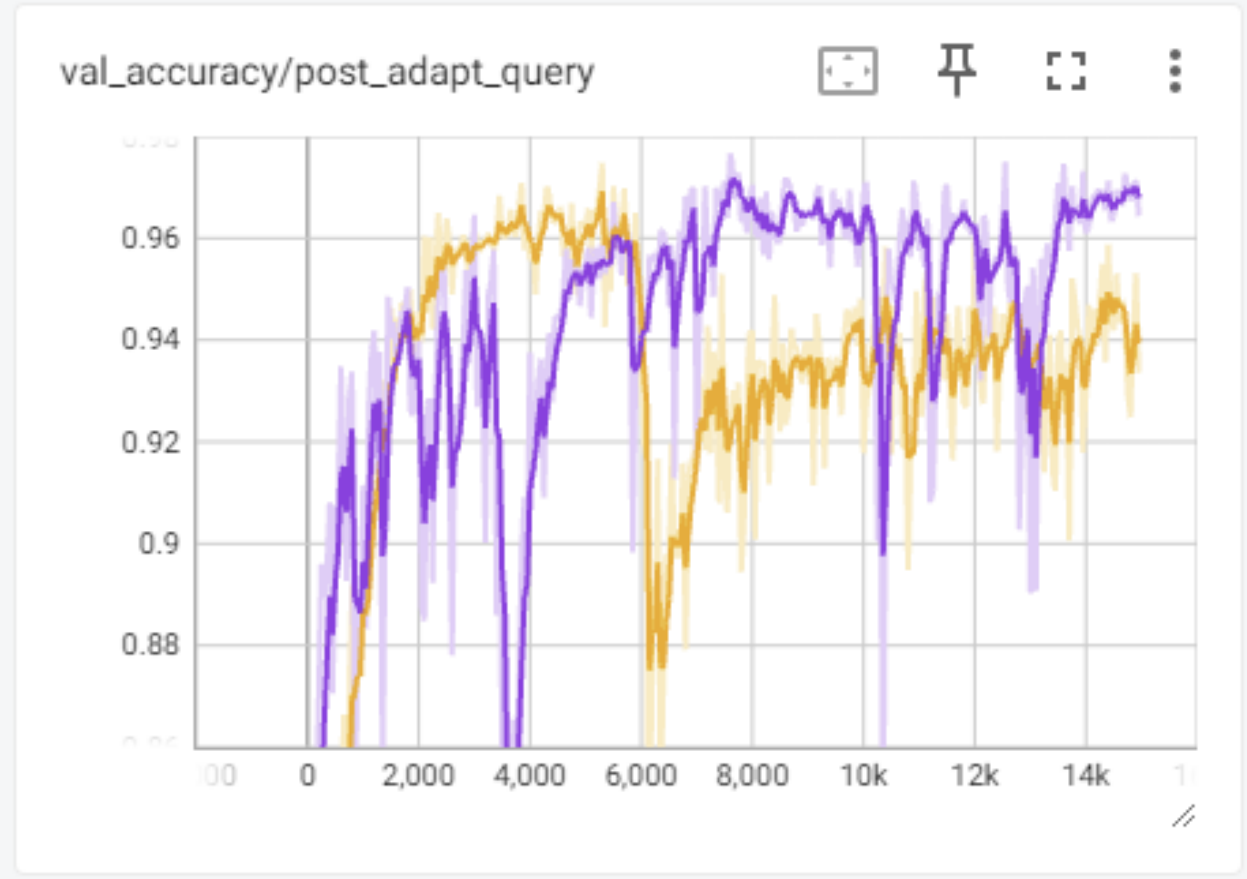


   (b) (2 pt) What is the effect of lowering the inner learning rate on (outer-loop) optimization and generalization?

   Optimization and generalization failed at one point.

4. Try MAML with a fixed inner learning rate of 0.04 for 5 inner loop steps.

   (a) (3 pt) 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.

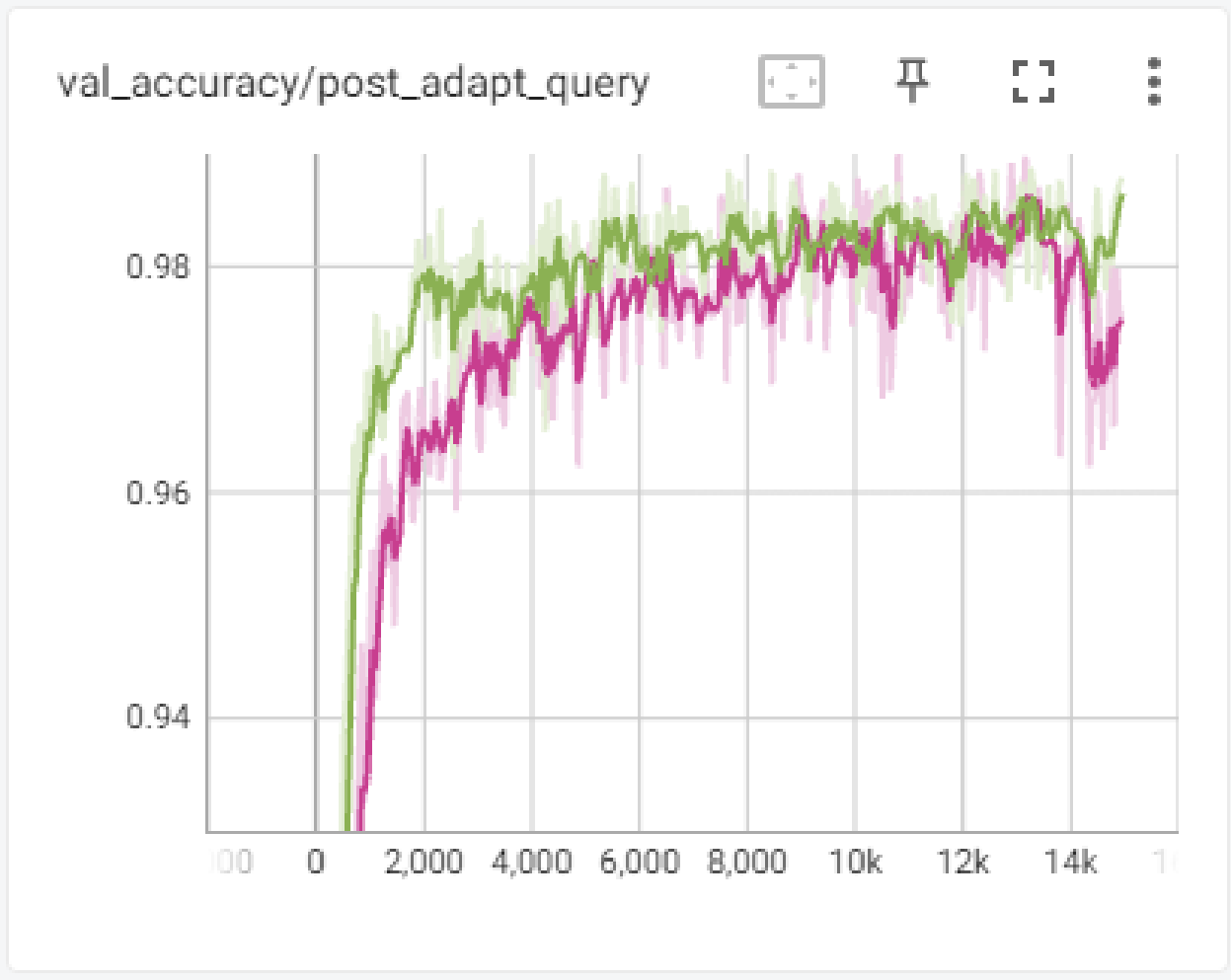


   (b) (2 pt) What is the effect of increasing the number of inner loop steps on (outer-loop) optimization and generalization?

   Optimization and generalization fail more often but recover faster.

5. Try MAML with learning the inner learning rates. Initialize the inner learning rates with 0.4.

   (a) (3 pt) 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.



   (b) (2 pt) What is the effect of learning the inner learning rates on (outer-loop) optimization and generalization?

   Optimization and generalization became more robust.

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.

1. 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.

   (a) (10 pt) 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.

   Your plot goes here.

   (b) (5 pt) How well is each model able to use additional data in a task without being explicitly trained to do so?

   Your answer goes here.

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.