CS 330 Autumn 2023 Homework 2
Prototypical Networks and Model-Agnostic Meta-Learning

Due Wednesday October 25, 11:59 PM PST

SUNet ID:
Name:
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) [1] and model-agnostic meta-learning (MAML) [2], for few-shot image classification on the Omniglot dataset [3], 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.

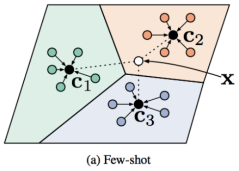
Preliminaries

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) [1]

Algorithm Overview



$$\mathbf{c}_k = \frac{1}{|\mathcal{D}_i^{\text{tr}}|} \sum_{(x,y)\in\mathcal{D}_i^{\text{tr}}} f_\theta(x)$$

$$p_\theta(y=k|x) = \frac{\exp(-d(f_\theta(x), \mathbf{c}_k))}{\sum_{k'} \exp(-d(f_\theta(x), \mathbf{c}_{k'}))}$$

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*

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

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 [4], 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.

   Solution: In black-box meta-learning used in previous homework, we input the support set sequentially. We shuffle the query set to avoid model learning patterns based on sequence order. For MANN, This strikes a balance between providing structured input (ordered support set) for building memory representations and introducing randomness (shuffled query set) for robust evaluation and learning. In protonets, the order of support set is no longer important or useful as we don't deal with them sequentially. So there's no pattern about order where we do not want the model to learn.

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%.

Solution: The results can be seen in the figure 2.

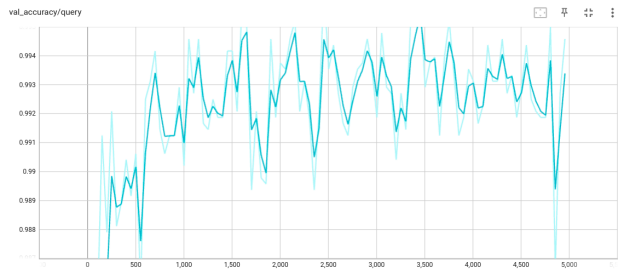


Figure 2: The accuracy of query set on the validation split. The model is trained using 5-way 5-shot.

3. **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.

   Solution: Figure 3 and 4 shows the accuracy on the support set in the train and validation splits. The model places most of the support examples in the same class together, but not all, there are still some examples from the same class that are not placed in the same class

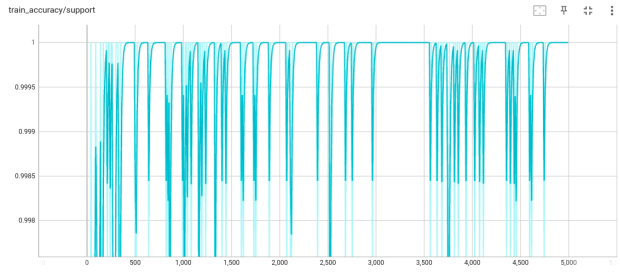


Figure 3: The accuracy of support set on the train split. The model is trained using 5-way 5-shot.

   (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.

   Solution: Figure 5 shows loss of the train and validation splits. The model indeed performs well before about step 3500, overfitting appears at the late stage where validation loss and query accuracy on the validation splits (figure 2) start to underperform.

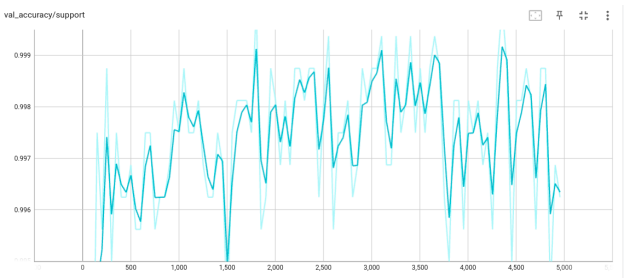


Figure 4: The accuracy of support set on the validation split. The model is trained using 5-way 5-shot.

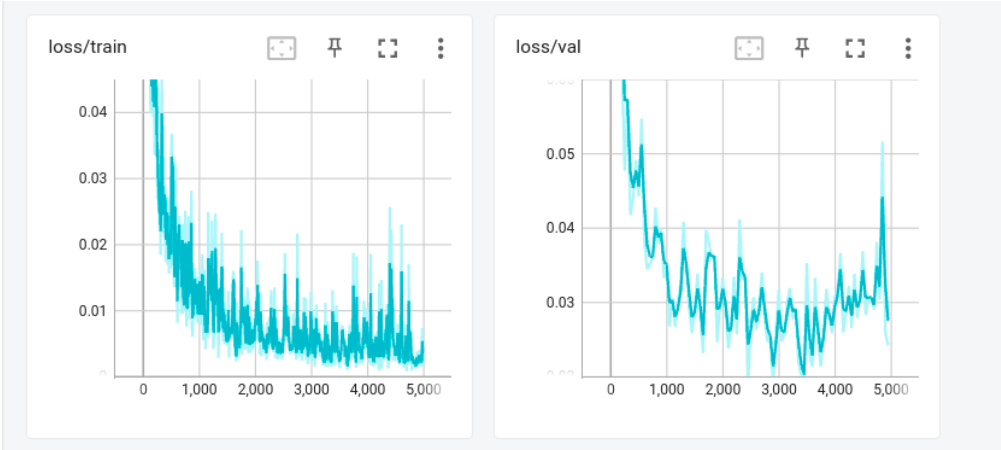


Figure 5: The loss of train and validation splits during training. It can be seen that overfit happens at about step 3500.

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

   (a) **[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.

   Solution: THe results can be seen in table 1.

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

   Solution: The last checkpoint (4900) is used for testing both models. As mentioned before, the checkpoints in the middle, like step 3500, should perform

	Accuracy
5-way 1-shot	98.5 %
5-way 5-shot	98.3 %

Table 1: The accuracy of the test splits with different experiment settings.

better cause these models do not overfit.

(c) **[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.

Solution: It looks like the difference is not significant between 5-shot and 1-shot. (I think there should be something wrong, as theoretically, more shots you have, the performance should be better.)

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

Algorithm Overview

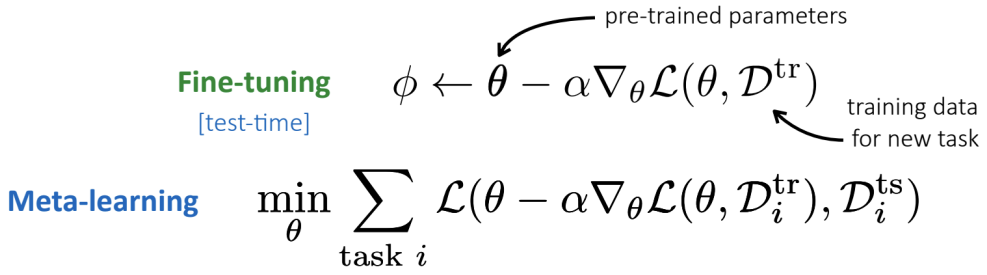


Figure 6: MAML in a nutshell. MAML tries to find an initial parameter vector θ that can be quickly adapted via task gradients to task-specific optimal parameter vectors.

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 [5] 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. **Implementation**

   (a) **[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
   ```

(b) [**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%.

Solution: The result can be seen in figure 7.

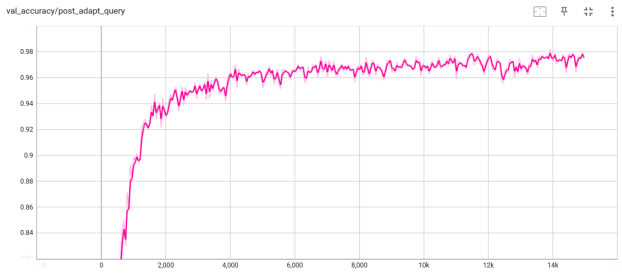


Figure 7: The validation post-adaptation query accuracy over the course of training.

2. **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.

   Solution: Figure 8 and 9 shows the logging results. Both accuracies show random performance which is normal.

   i) Each task in support set is split into separate train and validation sets, and each class is sampled from different tasks.

   ii) The parameters of the model are initialized randomly, which basically makes the model guess the label at the stage of pre-adaptation.

   iii) Image samples are labeled with orders during sampling, which indicates that the labels of the images do not contain useful information to help the model make decisions.

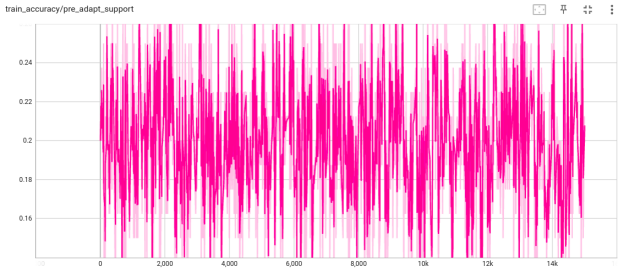


Figure 8: The accuracy on support set during training before adaptation with 5-way and 1-shot.

Figure 9: The accuracy on support set during validating before adaptation with 5-way and 1-shot.

(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.

Solution: Figure 8 and 10 compares accuracy on support set before and after inner loop adaptation. It can be seen that post-adaptation works far better than pre-adaptation. It means that model learns the tasks during inner loop.

The same comparison results also stand for corresponding validation datasets. The results can be seen in figure 9 and 11.

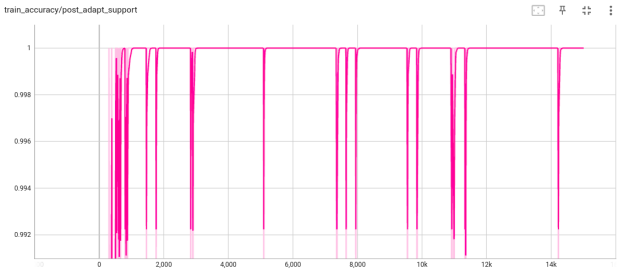


Figure 10: The accuracy on support set during training after inner loop adaptation with 5-way and 1-shot.

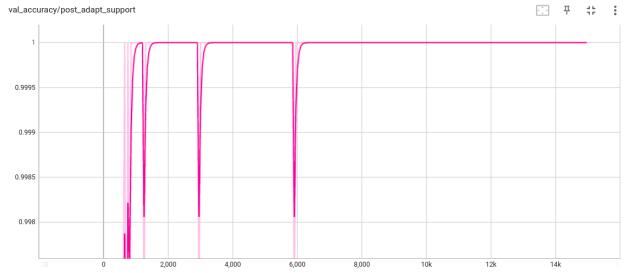


Figure 11: The accuracy on support set during validating after inner loop adaptation with 5-way and 1-shot.

(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.

Solution: As shown in figure 10 and 12. The learning process is slower on query set, as the model has been trained several times in the inner loop. Compared with support set, the data on query set has not been seen during training, which makes model perform poorly. As training continues, model learns how to quickly adapt to new tasks, which makes their performance better at the latter stage. THe similar results can be seen in figure 7 and 11 for validation.

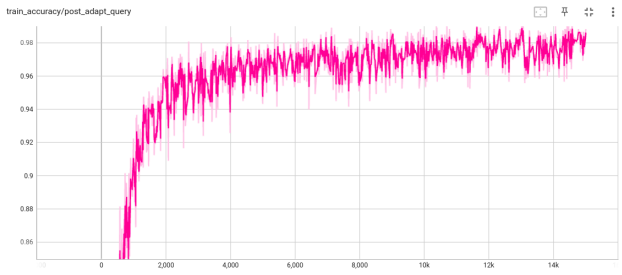


Figure 12: The accuracy on query set throughout training after inner loop adaptation with 5-way and 1-shot.

3. **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
   ```

   Solution: The results can be seen in figure 13.

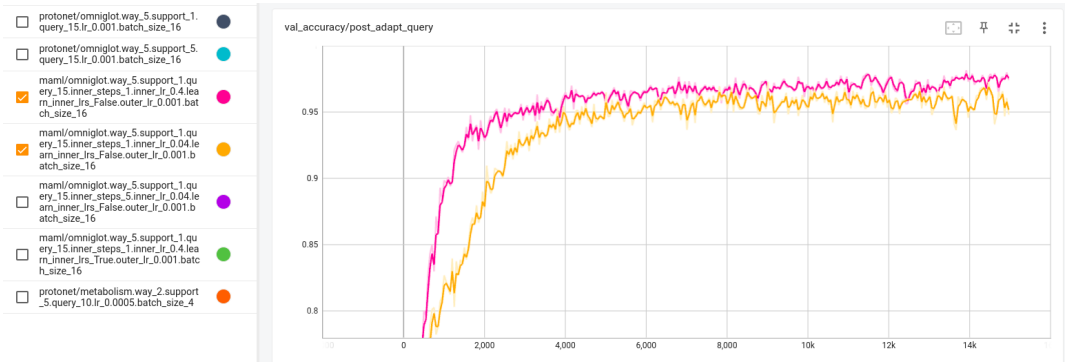


Figure 13: The accuracy comparison on query set during validation, the pink uses inner learning rate of 0.4, the yellow uses inner learning rate of 0.04.

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

   Solution: The learning rate in this case makes the learning process slower. The accuracy is worse with lower learning rate. It may indicate that the model is stuck in the local optima.

4. **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
   ```

   Solution: The results can be seen in figure 14

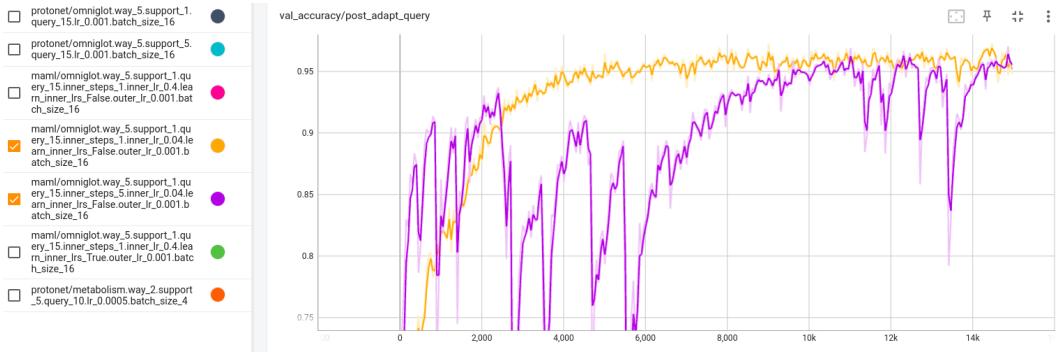


Figure 14: The accuracy comparison on query set during validation using inner learning rate of 0.04, the yellow uses inner step of 1, the purple uses inner step of 5.

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

   Solution: Increasing the number of inner loop steps harms the generalization. It can also be seen in figure 15. It could be explained by that model is overfitting on the downstream tasks such that harms the ability to learn to adapt.

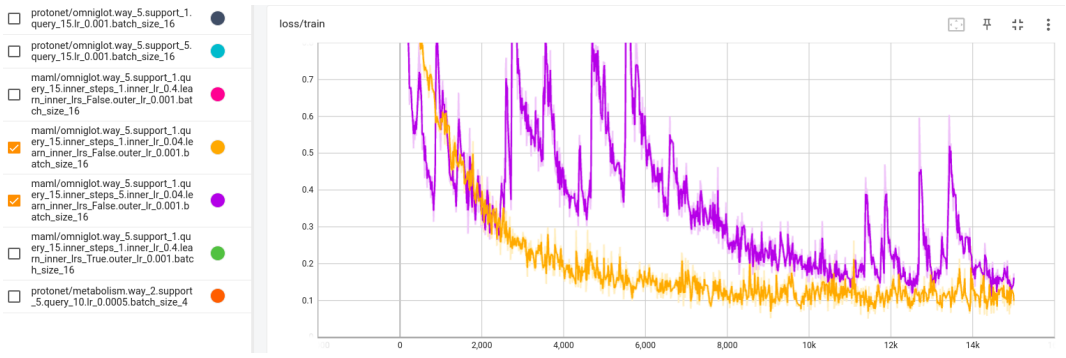


Figure 15: The outer loop loss comparison on set during training using inner learning rate of 0.04, the yellow uses inner step of 1, the purple uses inner step of 5.

5. **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`

   Solution:

Figure 16: The accuracy of learning inner learning rate vs. no learning on the query set during validation.

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

   Solution: Learning learning rate makes model adapt faster. From training steps so far, it also makes model perform better.

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. **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 hyperparameters with command line arguments**.

   (a) **[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.

   Solution: For Protonet, we use 5-way 1-shot with learning rate of 0.001; For maml, we use 5-way 1-shot with learned learning rate initialized at 0.4. The results can be seen in table 2 and figure 17.

K	ProtoNet Accuracy (%)	ProtoNet 95% CI	MAML Accuracy (%)	MAML 95% CI
1	0.985	0.003	0.979	0.003
2	0.994	0.001	0.990	0.002
4	0.995	0.001	0.993	0.001
6	0.996	0.001	0.994	0.001
8	0.996	0.001	0.996	0.002
10	0.997	0.001	0.996	0.001

Table 2: Comparison of test accuracies between ProtoNet and MAML models trained on 5-way 1-shot tasks and evaluated with varying numbers of support examples (K). Both models use Q=10 query examples, and MAML's inner learning rate is initialized at 0.4.

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

   Solution: Both models demonstrate effective use of additional support examples despite being trained only on 1-shot tasks, indicating good generalization capability. ProtoNet maintains a slight edge across all K values, but MAML shows stronger relative improvement from its starting point. This suggests both architectures have learned robust feature representations that can effectively leverage additional examples at test time, even without explicit training for this scenario.

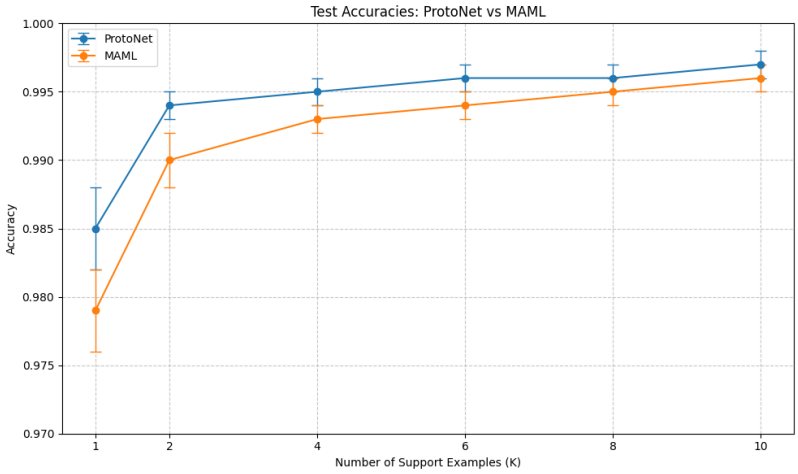


Figure 17: Comparison of test accuracies between ProtoNet and MAML models trained on 5-way 1-shot tasks and evaluated with varying numbers of support examples (K). Both models use Q=10 query examples, and MAML's inner learning rate is initialized at 0.4.

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 [6], 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 [7], 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.

1. **[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:

   Solution: The results can be seen in figure 18 an table 3.

Figure 18: The validation performance.

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.

Metrics	Values
Training Loss	0.001
Training Support Accuracy	1.000
Training Query Accuracy	1.000
Validation Loss	0.916
Validation Support Accuracy	0.903
Validation Query Accuracy	0.720
Validation CI95	0.004

Table 3: validation query accuracy with ci95(confidence interval 95%)

References

[1] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems*, pages 4077–4087, 2017.

[2] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 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] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[5] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your maml. *arXiv preprint arXiv:1810.09502*, 2018.

[6] Huaxiu Yao, Linjun Zhang, and Chelsea Finn. Meta-learning with fewer tasks through task interpolation. In *International Conference on Learning Representations*, 2021.

[7] Kexin Huang, Tianfan Fu, Wenhao Gao, Yue Zhao, Yusuf H Roohani, Jure Leskovec, Connor W Coley, Cao Xiao, Jimeng Sun, and Marinka Zitnik. Therapeutics data commons: Machine learning datasets and tasks for drug discovery and development. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.