CS 282 Deep Neural Networks **Spring 2023**

Lecture 22: MAML & Generative Models

Lecturer: Anant Sahai *Scribe: Zixun Huang, Tianyue Cheng*

1. Model Agnostic Meta-Learning

1.1 Recap: Multi-task Learning Problem

Given: A training dataset of datasets tied to many different tasks, i.e. $(X_{i,j}, Y_{i,j})$ for example j in task i.

Target: The goal is to create a model that can be effectively fine-tuned for new tasks, enabling generalization across tasks.

1. Rather than starting the training process from scratch each time a new task is encountered, we aim to develop a more effective approach.
2. Specifically, we seek to construct a potentially larger model that can be fine-tuned for the task at hand.
3. This aligns with the principles of Model Agnostic Meta-Learning (MAML), which aims to enable effective fine-tuning using only a small amount of task-specific data.

1.2 MAML Perspective

To address the multi-task learning problem, we can approach it in a similar manner as we would a standard supervised learning problem, which is to "Do SGD" (stochastic gradient descent).

1.2.1 What Does "Do SGD" Mean?

To "Do SGD", we follow these steps:

1. Select a particular example, which, in the case of MAML, refers to a single task.
2. Evaluate the performance of the model on the task and compute gradients for what the model would have been if it had been trained solely on that task.
3. Take a step in the direction of the gradient computed above, adjusting the model parameters to optimize performance on that task.

1.2.2 How to Implement "Do SGD"

The key insight of the first approach is to treat a task as the elementary "thing" in SGD. To implement this approach, we can follow these steps:

1. Initialization: randomly initialize our model parameters θ_0, or start with a pre-trained network for some other self-supervised tasks.

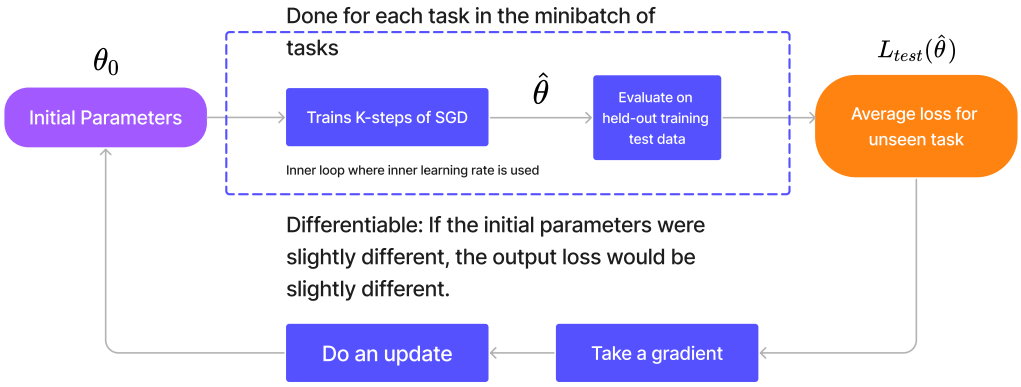

Figure 22.1: MAML gradient update illustration.

2. Randomly pick a task. Note: We need to cache the model parameters θ_0 here.
3. Fine-tune our model using the training data $(X_{i,j}, Y_{i,j})$, where i indicates the dataset index of tasks for training, and j indicates the sample index of training data in those datasets, i.e.

$$(X_{train,train}, Y_{train,train}) \tag{22.1}$$

   To achieve this, we also need to consider the following problems: A) choose a training loss function relevant to this specific random-picked task; B) choose a learning rate; C) do a train/test split for the dataset of this task; D) choose a test loss relevant to the corresponding task. With K-steps of SGD, we will obtain new estimated model parameters $\hat{\theta}$. We can then evaluate on the held-out training test data $(X_{i,j}, Y_{i,j})$, where j and i respectively indicate unseen "training" samples in unseen "test" tasks, i.e.

$$(X_{test,train}, Y_{test,train}) \tag{22.2}$$

4. Ask PyTorch to compute the gradient of the training test performance with respect to the initial model parameters θ_0 before step 3, i.e. the fine-tuning performance with respect to the thing that defines the fine-tunable model or the thing that we fine-tuned.
5. Do an update:

$$\theta = \theta_0 - \eta \frac{\partial L_{test}}{\partial \theta_0} \tag{22.3}$$

Here, θ is the updated model parameters, η is the learning rate, and $\frac{\partial L_{test}}{\partial \theta_0}$ is the gradient of the test loss with respect to the initial model parameters θ_0.

Take Away: A) To implement this approach, you need to know how to use PyTorch along with the steps mentioned above. B) It is important to understand how this approach is a manifestation of doing SGD (Figure 22.1).

1.3 Alternative Approaches & Discussions:

1.3.1 The First Alternative Way: Nomal SGD Approach

Q: What is the difference between the MAML perspective and regular SGD?

A: The MAML perspective treats each task as an elementary thing in SGD, meaning that we randomly pick a task, fine tune the model on that task's training data, and then update the model parameters based on the test performance of that task. In contrast, regular SGD (Figure 22.2) can use a shared model with different task heads and loss layers, and then do SGD on training examples across tasks. In this approach, the training dataset of datasets is treated as one big dataset with different losses attached to different examples.

We perform the tasks one at a time, with each task consisting of a single example. The shared model receives gradients from all the different tasks, while the individual task heads receive updates only from their respective tasks. After the gradient updates, we can use the shared components to fine-tune the model for an unseen test task.

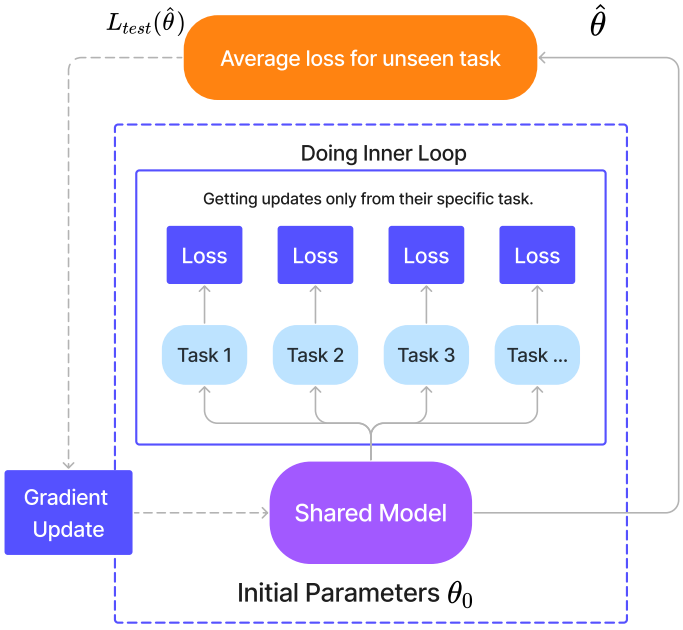

Figure 22.2: Normal SGD gradient update illustration.

The effectiveness of the normal SGD approach versus the MAML approach is currently a topic of great debate, with some modifications being added to both methods. In the normal SGD approach, we evaluate the fine-tuning performance and take gradients with respect to the updated model parameters, rather than the initial model parameters. Additionally, we only have one set of shared model parameters in the normal SGD approach for updating, whereas in the MAML approach, the fine tuning steps corresponding to step 3 are tentative and exploratory updates. The $\hat{\theta}$ in MAML represents a tentatively updated state of the model, allowing us to evaluate the fine tuning performance loss and compute the gradient of the loss with respect to the initial model parameters. For more information, please refer to the paper "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks"

1.3.2 Batch Size

Q: What would be the batch size in MAML?

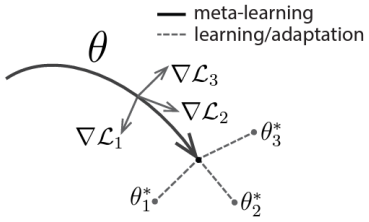

Figure 22.3: MAML gradient trajectory illustration.(Finn et al., 2017)

A: While the concept of batches can be applied in MAML, it ultimately depends on the implementation. In theory, we could fine tune multiple tasks in parallel by taking the initial condition θ_0 in the shared model and applying different steps for each task (Figure 22.3). Each task would then update the initial condition in a different direction, leading to a set of different losses that contribute to updating the initial condition.

From the implementation point of view, it will force us to make batch size decisions. Though, in this lecture, we were only talking about the first level idea in spirit, that is, literally doing everything that you would have done for a normal SGD except regarding tasks as samples, where fine-tuning is just parts of the operations.

To be more explicit, the whole step 3 is just a function evaluating the performance of the model and its loss, which has its own inner steps.

1.3.3 Comparisons in RL

Q: Is there a relationship between these ideas and the principle of on-policy or off-policy learning in Reinforcement Learning (RL) settings?

A: These ideas are independent (orthogonal), meaning that there is no inherent association between them. In the SGD approach, tasks are considered a part of a larger task to obtain an optimal solution. Conversely, on-policy or off-policy comparison in RL differs from this.

In RL, the MAML approach is more similar to Bellman iterations where the emphasis is on finding a suitable location to acquire the right solution with more data. Dynamic programming can be seen as an approach to stochastic MDPs, underscoring the significance of being in the correct location. On the other hand, off-policy techniques such as Q-learning prioritize Bellman operations over policy gradient.

For additional details, kindly refer to material from CS285.

1.3.4 A Second Alternative Method: Semi-frozen

The alternative method discussed involves utilizing a pre-trained model to perform well in a multitask learning scenario. This approach requires treating the shared model as entirely frozen for each task during training, as explained in section 1.4.3, also referred to as the Closed-form Strategy.

In contrast to the zeroth alternative method (Figure 22.1), which does not rely on the example tasks themselves but rather on other techniques to tune the model for fine-tuning, this method trains a model that is expected to be entirely frozen during test time. The closed-form strategy is employed to prevent the occurrence of exploding gradients.

The approach is hybrid since the model is not entirely frozen. During training, the model is unfrozen, and numerous updates are made on it. However, during test time, the model is semi-frozen by restoring it to a checkpoint for the parameters, training it on the task, and utilizing it for that particular task. When a new task is introduced, the model is restored to the checkpoint and trained for the new task. This approach

is a middle ground between the fully frozen approach that utilizes a pre-trained model with self-supervised losses and the constantly adapting approach, given different tasks. The model is trained on numerous tasks to achieve an optimal initial condition but is then fine-tuned in a task-specific manner.

1.3.6 Exploding Gradient & Unrolled Loop

Q: Is the gradient descent in multi-task learning really hard to solve?

A: In ideal conditions where infinite precision is available for real-valued computation, the correct gradient answer can be obtained. However, in practice, attempting to compute the gradient directly could result in NaNs.

In terms of implementation, computing gradients can be simple, and Jax provides an easy-to-use solution (Frostig et al., 2018). Conceptually, all the steps involved are iterative, and PyTorch is capable of taking derivatives. The only obstacle to taking a derivative arises when there is an $argmax$ or an if statement involved. Therefore, fine-tuning can be performed by specifying the number of steps and using an unrolled loop (refer to Figure 22.5).

Recall the inner loop (the iterations of K steps of SGD) we have in MAML (Figure 22.4), we can take derivatives and update gradients with this following equation:

$$\theta_{t+1} = \theta_t - \eta_{inner} \bigtriangledown L_{train}(\theta_t) \tag{22.4}$$

We have the option of unrolling the inner loop and creating a computation graph for MAML, similar to Figure 22.5. This is akin to a deep RNN where weights are shared across repetitive blocks with residual connections. While we don't need to be concerned about dying gradients when using this residual style connection, we must be wary of exploding gradients w.r.t θ_0, since we cannot easily use normalization. In the next section, we will discuss potential solutions to address the problem of exploding gradients.

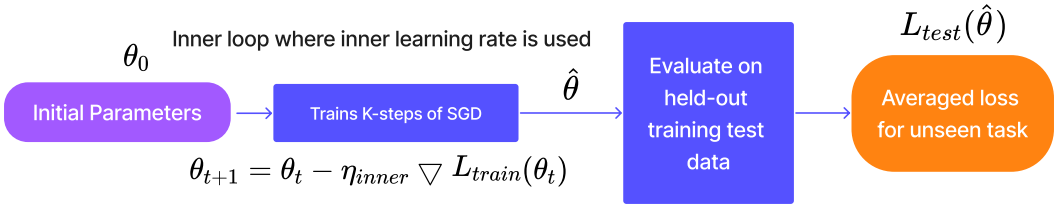

Figure 22.4: MAML

1.3.7 Take Away: Don't fear dying gradients, But Do Fear EXPLODING Gradients!

Typically, when dealing with exploding gradients, we use normalization techniques. But here we can't do this, because if we did so it would no longer reflect what would happen at test time when we're finetuning anything. So we have to use the more basic approaches, i.e. to dial the inner learning rate down or to make the net shallow.

Practically, we need to limit K and η_{inner} to prevent gradients from exploding when we fine tune a model using training data. Traditionally treating iterations, we shuffle our data, put it into batches based on memory, and then excute a set of training epochs until we get good convergence. But in this MAML problem, iterations are precious and we can not have a too deep structure, i.e. let K to be too big.

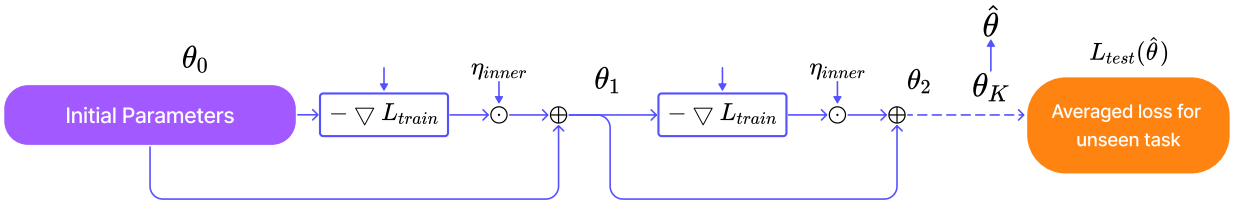

Figure 22.5: MAML with unrolled inner loop

Since we cannot have too many iterations and then we want to get more data done in one iteration, i.e. stuff bigger things into our batches, it's going to hit a memory constraint at some level. So if our data set is pretty large, we might not be able to practically go through all of our training data in K iterations.

So what can we do about it?

1.4 Practicalities

Key idea: Do what you can.

1.4.1 Subset Strategy

When we sample our task, we can make a random K step version of that task.

For example, here is one specific task having a thousand things of training data and we cannot go through them all, so we can randomly pick a subset to make a new batch size, e.g. 100 samples. And then we do inner loop with SGD in this subset.

It's not satisfying enough, and the remaining problem is that we're not able to fully fine tune this task and we cannot see what the initial condition effect would have been. Recall subsection 1.3.1, now we can see why practically speaking, with the necessary implementation twists, MAML style approaches stop being as different from normal SGD approaches since the exploratory updates are not go far and we can not see how the entire thing would fine-tune necessarily.

1.4.2 Reptile Strategy

In place of computing $-\frac{\partial L_{test}}{\partial \theta_0}$, we can just use $\theta_K - \theta_0$ as the direction we want gradients to go.

Recall the gradient trajectory picture (Figure 22.3) and the inner gradient updating equation (22.4), the fine tuning steps $\bigtriangledown L_{train}(\theta_t)$ are all pointing in a similar direction. The potentially exploding gradient is the derivative of test set $-\frac{\partial L_{test}}{\partial \theta_0}$ w.r.t the initial parameters. And if these training points are like our held out test points, then the next step will intuitively be somewhere pointing out from the $\hat{\theta}$.

(Question from the scribers: why not somewhere pointing out from θ_0, is it because the MAML method can be an approximation of the regular SGD method based on iteration size limitation?)

Executing the unrolled loop is no trouble at all, like evaluating a ResNet going forward is not necessarily exploding in any way, but the gradients coming back are what could explode. So one cheap way of doing a gradient without computing the gradient is that use "where did we move to" to replace "where we want to move", i.e. in place of computing $-\frac{\partial L_{test}}{\partial \theta_0}$ and doing chain rules, we can just use $\theta_K - \theta_0$ as the direction

that we want gradients to go. Then we can make a lot of steps and all we need to do is to make sure that our network is stable in training.

For the network by itself, we can put in normalization layers, the problem statement is that we are not allowed to put any normalization layers when we do unrolling, which may leads to exploding gradients. So this reptile strategy uses $\theta_K - \theta_0$ as a proxy, i.e. we update parameters using $\eta_{inner}(\theta_K - \theta_0)$, and we can take much more steps (Figure 22.6), like T steps ($T >> K$), so we have:

$$\theta_{t+1} = \theta_t - \eta_{inner}(\theta_T - \theta_0) \tag{22.5}$$

We start from the initial model parameters θ_0, and then take a small step in the direction that pointing out to θ_T.

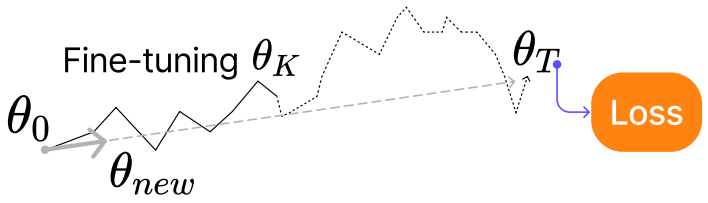

Figure 22.6: We can use much more steps (the dotted line) with reptile modification using MAML with reptile strategy.

Recall figure 22.5, a bunch of first derivatives $\bigtriangledown L_{train}$ are computed by PyTorch. Then we ask for the derivative of the final loss w.r.t the initial condition. So we need to go back, and run derivative operations back through the loop, which means we end up with taking a derivative of a derivative, i.e., a second derivative. The key thing is that traditional second order methods would look at the full Hessian which is intractable and its size is enormous.

Reptile strategy avoids ever computing the entire Hessian because we only ask for the derivative of the training gradient in a particular direction. What we need is fundamentally a second derivative type thing, and actually we didn't compute the second derivative at all in the reptile strategy. We instead use the first derivative as the kind of approximation for it. It works practically when those entire sets of training steps had the dominant term and a drift moving in a particular direction, since the dominant part should be very first derivative like.

1.4.3 Closed-form Strategy

The third strategy to deal with the exploding problem says that we can also take advantage of our task-specific heads. In the frozen model case (Figure 22.7), the task heads might be very easy to train. For example, we're turning a task head for a squared error loss and treating the frozen shared model as a black box feature generator.

If we do least square on the generated features, we can use the closed form expression for least squares, so we don't have to solve it by doing gradient descent iterations.

Recall that K steps of SGD was the approximation for fine tuning, so what if we fine tune it in a style of having the shared model be frozen? We now go for a frozen model strategy. The problem becomes trying

to get a good frozen model rather than a fine-tunable model. Then what we can do is to use a closed form expression for the solution of training w.r.t a frozen model, i.e.:

$$(\Phi^T\Phi + \lambda I)^{-1}\Phi^T\vec{y} \tag{22.6}$$

where Φ has the outputs of "frozen model". And we can see the closed-form equation 22.6 is differentiable, which means PyTorch can take a step through this equation and compute gradients w.r.t θ_0.

We still have a held-out test set like standard MAML, except that instead of doing case steps of SGD, we use a closed form expression to solve it. This avoids having a deep structure, but we still have the problem of having to make sure the expression is stable for getting gradients. So we have the regularization term to prevent the inverse from blowing up.

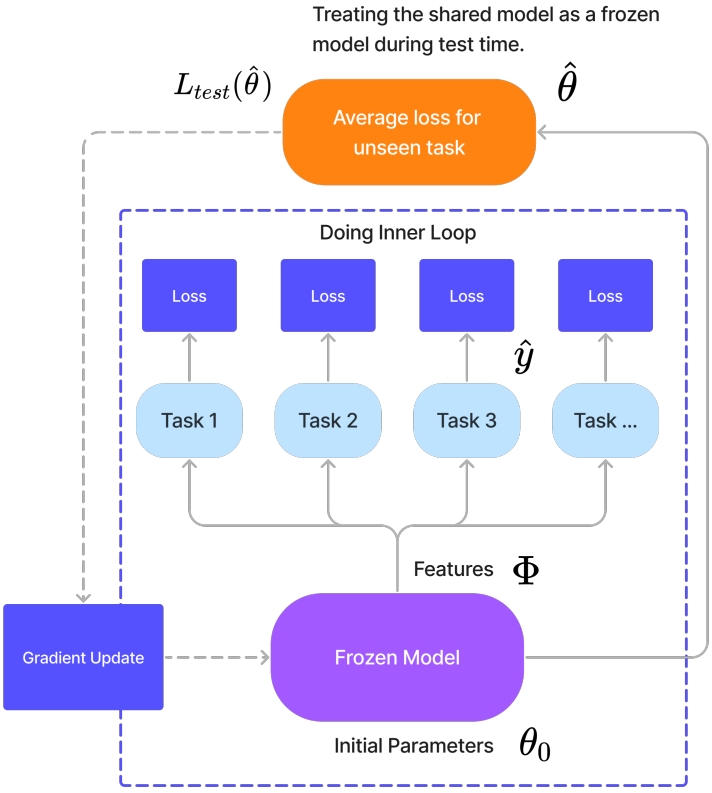

Figure 22.7: MAML with semi-frozen strategy

Then all we need to do is just to make sure that the equation fits into the memory. If we have lots of training data, the term $\Phi^T\Phi$ won't fit. But obviously, now we may be able to have a much bigger batch-size for the inner fine tuning process, since here we only consider backbone features Φ and parameters in the task-specific heads. You can check the paper "ANIL" (Raghu et al., 2019) and "R2-D2" (Bertinetto et al., 2018) for more details. Here we picked the loss for specific task heads as a squared error, because that's the

easiest one. But actually the same closed-form strategy conceptually works if we can treat the final problem as a convex problem, e.g. logistic loss or cross-entropy loss.

The main spirit here is to take advantage of better convex strategies to find something that we can differentiate, so that we don't have to involve a number of steps of SGD. Let's think about what would make a few steps of SGD succeed, it would succeed if they're pointing very strongly in only a few directions in which we can then do well in our task. So what this convex strategy is doing is that it adds pressure to train the model to have strong gradients in a few directions which are task relevant. It can purify the "frozen" model and pull out the right things for the tasks.

1.4.4 Take Away

In practice, all of the above-mentioned sophisticated approaches compete against the regular SGD approach. However, there is another idea that can be quite effective: finding relevant data to come up with a self-supervised surrogate task. For generic problems, the standard MAML is less effective compared to simply doing normal SGD or even the frozen strategy. But in a specific context, where there is no access to generic data, the standard MAML pays off quite a bit.

When comparing the frozen method with the regular SGD method, we find that the tangent features associated with tasks are actually the output of the shared model. They are always going to be relevant, so even if we want to train the shared model, getting the task heads right is good.

We can use different combinations based on the mentioned strategies. Recall the fine tuning methods, we can first train our heads on a frozen model, and then do updates. This can prevent forgetting from happening on important directions. We can do zero initialization on task heads, which is better than random initialization, but still not as good as initializing them to work with the frozen model.

2. Generative Models

Goal: In the generative or simulation problems, we want to be able to synthesize examples from some (conditional) distributions.

2.1 Basics

How do we sample from a known distribution? The problem we actually have is that we want to draw synthesized examples from some conditional distribution. We don't have access to this distribution at all, all we have access to are examples.

Before we can tackle the actual problem above, let's suppose we have a known distribution, how to draw a sample from that distribution?

2.1.1: A Discrete Distribution

Given a discrete distribution, $P_1, P_2, ..., P_N$, where $\sum P_i = 1$. We can use inverse CDF-style approach.

1. Start with a continuous uniform r.v. U.
2. Segment the unit interval as follows:

$$[0 \sim P_1 \sim P_1 + P_2 \sim ... \sim P_1 + P_2 + ... + P_{N-1} \sim 1] \tag{22.7}$$

3. See in which interval U lands and emit that label.

2.1.2: A Continuous Distribution

Given a continuous distribution f_x. To solve it, we can use inverse CDF-style approach again. It's like a counterpart of the discrete problem except infintely many intervals.

1. Take the distribution, i.e. the density of f_x.
2. Compute the CDF.
3. Use the sample view to tell where to read the answer along the CDF.

2.2 Takeaway

If we want to do generation or synthesis, we have to have some kind of structure which acts like the distribution. And we have to use randomness since we need something to generate out a set of possible things. In the spirit of the inverse CDF style, we take random things, run them through a function that should be like the inverse CDF and generate something out.

There is another way to draw samples from a known distribution. We can set up some kind of random walk whose stationary distribution is the one we're interested in. Then we can sample from that distribution by taking a simulation run of the random walk and then seeing where we end up.

2.3 Teaser

GANs and variational Auto-encoders are roughly in the inverse-CDF style family, while diffusion models will be in the other family. We'll also talk about using GPTs to generate things which is an interesting in-between approach.

3. What we wish this lecture also had to make things clearer?

1. We wish the lecture can explain more about the limitation of using normalization in MAML.
2. Demos on MAML compared to alternative approaches could be helpful.
3. The last part which briefly introduces generative models could be combined with the subsequent lecture on generative models.

References

Bertinetto, L., Henriques, J. F., Torr, P. H., and Vedaldi, A. (2018). Meta-learning with differentiable closed-form solvers. *arXiv preprint arXiv:1805.08136*.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR.

Frostig, R., Johnson, M. J., and Leary, C. (2018). Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9).

Raghu, A., Raghu, M., Bengio, S., and Vinyals, O. (2019). Rapid learning or feature reuse? towards understanding the effectiveness of maml. *arXiv preprint arXiv:1909.09157*.