CS 282 Deep Neural Networks **Fall 2022**

Lecture 24: Meta-Learning

Lecturer: Anant Sahai *Scribe: Terrance Wang, Wyame Benslimane*

1 . Recap: MAML - Model Agnostic Meta-learning

1.1 What is the problem we are trying to solve:

In previous lecture, we discussed meta-learning. To put it simply, we are trying to learn algorithms that learn from other learning algorithms. Model-Agnostic Meta-Learning (MAML) extends this idea and tries to do multi-task learning by fine-tuning. Therefore we want to optimize for a pre-trained model that can adapt to a variety of tasks in a few gradient steps.

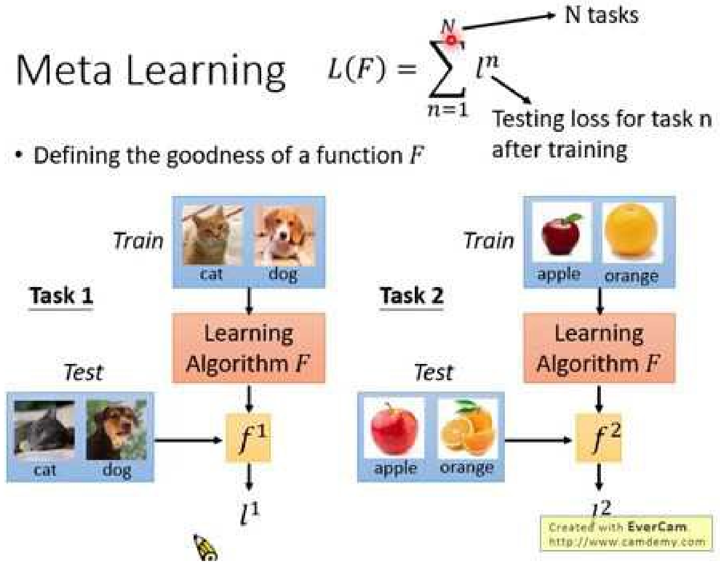

Meta Learning
$L(F) = \sum_{n=1}^{N} l^n$
N tasks
Testing loss for task n after training
• Defining the goodness of a function F
Train
cat
dog
Task 1
Learning Algorithm F
Test
cat
dog
f^1
l^1
Train
apple
orange
Task 2
Learning Algorithm F
Test
apple
orange
f^2
l^2
Created with EverCam. http://www.camdemy.com

Figure 24.1: Meta Learning principle

1.2 What does the test-time look like?

The first step in this process is to start with a pre-trained model. Our choices are:

1. A model that already has a 'ready to modify' task head (see figure 24.2): In all deep models, the last block (linear layer) is responsible for converting the features learnt by the deep model to outputs for the problem the model is trying to solve. This block is what we call the task head. For example, in a classifier, this block is responsible for outputting the probability scores of different classes.
2. Start our pre-trained model with randomly initialized task specific head.

The next step in this process is to train the model using task specific training data, this can also be done in 2 different ways:

1. Fine tune the entire model.
2. Just train the task specific head.

Then we evaluate performance on task specific held out data. This step gives us a quantitative measure of how well we did.

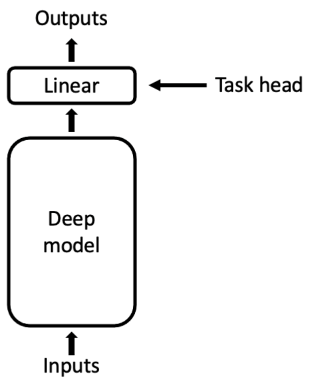
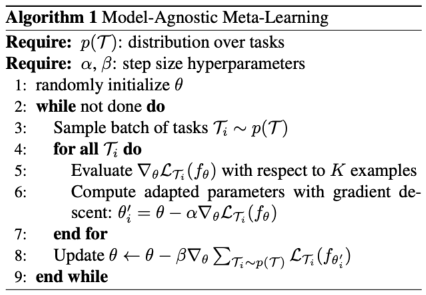


Figure 24.2: features generated by the deep model get fed into the task head to be converted to outputs.



```
Algorithm 1 Model-Agnostic Meta-Learning
Require: $p(\mathcal{T})$: distribution over tasks
Require: $\alpha, \beta$: step size hyperparameters
1: randomly initialize $\theta$
2: while not done do
3:     Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4:     for all $\mathcal{T}_i$ do
5:         Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to $K$ examples
6:         Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7:     end for
8:     Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: end while
```

Figure 24.3: MAML Pseudo-code Source

1.3 How do we optimize this problem?

In order to train any model, the first thing we need to have is **training data**. Therefore we need a lot of example tasks where each task has labeled training data. The next step is to use the standard approach for solving this problems in deep learning: SGD. This means that we are going to iterate the process below until it converges (or is stopped early):

1. Pick a mini-batch of tasks from training data.
2. Use the current parameters to evaluate the model on this batch (as though it was test time) and compute gradients using back-propagation.

3. Update the model's starting parameters with a step of size η_{outer} times the negative gradient.

This training procedure has a key difference with traditional model training. MAML requires us to evaluate the model on the held out set with a differentiable loss function. (e.g.: cross entropy loss instead of a binary right/wrong for a classification model) This is a necessary difference because instead of simply determining model performance on the held out set, we also want to update our initial weights. Furthermore, each task in the mini-batch is trained independently, and within a batch, each task starts with the same initial weights.

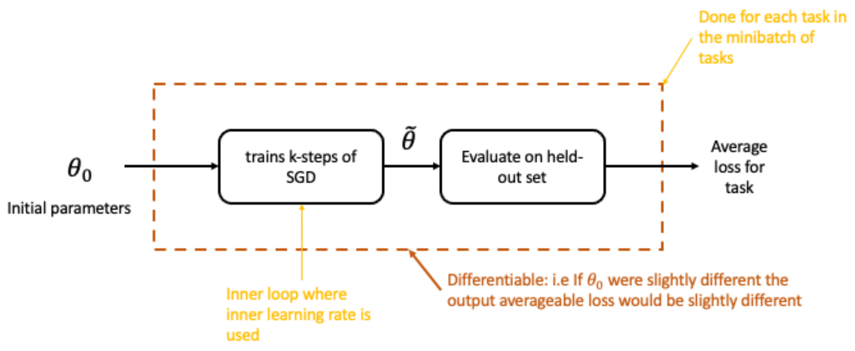

Done for each task in the minibatch of tasks
θ_0
Initial parameters
trains k-steps of SGD
$\tilde{\theta}$
Evaluate on held-out set
Average loss for task
Inner loop where inner learning rate is used
Differentiable: i.e If θ_0 were slightly different the output averageable loss would be slightly different

Figure 24.4: illustration of MAML training loop

For the previous figure 24.4, we can see that MAML trains 2 different loop. Remember that our training data points are different specific tasks. Therefore our 'inner loop' trains on task-specific labeled training data while the 'outer loop' trains on the different tasks.

Why do we need a differentiable loss function for MAML?

If we consider the block from the previous figure 24.4 where we train k-steps of SGD, the parameter $\theta_{t+1} = \theta_t + \eta_{inner}(-\Delta\mathcal{L}(\theta_t))$ is something that looks like an RNN because we are using k steps of SGD (See figure 24.7). Therefore we need the loss to be differentiable so that the gradient from the held out set can be passed back and used to update the initial weights.

It is important to differentiate between the two learning rates η_{outer} and η_{inner}, which correspond to the two training loops found in MAML. We can see that η_{outer} is used in the outer training loop, which tries to find a good initialization of the network and updates the model's starting parameters at the end of each mini-batch. η_{inner} is used in the inner loop, which updates the model for k steps on each mini-batch task before the gradient is calculated to update the initial parameters.

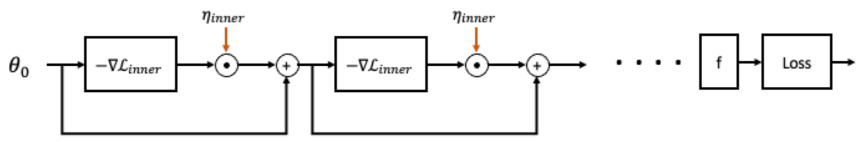


Figure 24.5: the unraveled inner loop resembles an RNN structure

1.4 What are the main challenges of MAML?

Exploding gradient during training: as k (the number of inner steps) increases, the network gets deeper, so exploding gradients become a problem.

Memory: large values of k require storing all the intermediate activations for back-propagation, and memory issues arise as a result.

In order to avoid these challenges, we can only used a limited k for our training tasks. Therefore at training time, k will small compared to the number of steps we will take to fine tune the model at test time.

Design choice:
If we are using randomly initialized task specific heads, when we resample a previously seen task, we can either use randomly initialize a new task specific head each time, or store and reuse the previously trained task specific head.

- the latter technique can be useful because the model will not be close to seeing all the data points in a given task within k inner loop steps.
- This can also be useful because the randomly initialized task head will not perform well at early iterations, and therefore will not be effective at correctly updating the model early on.

1.5 Step Back: Why does this work?

The first question to answer is: What are we learning?

- We are learning 'good features' for the deep network. Conventionally, we think of a deep neural network as an embedding that turns our input into a features. In this case the features are the outputs of the model that we are fed into the task specific head in order to get our predictions.
- We can also think of features as derivative features (tangent view). In this case, we consider the outputs of each layer of our deep network as features too. Therefore, when fine-tuning, we want these derivative features to let our model match the task quickly.
- Few shot fine tuning succeeds when the derivative features appropriately capture the task we're interested in. Because we are seriously overparameterized in few shot learning, we need the principal features of the tangent view do the work of capturing the task.

2 . Meta Learning Alternatives

Alternative options to MAML:

In practice, there are methods other than the MAML procedure described above that can be used for few shot learning. One option is to train the model on the union of all tasks. This means to simply run the standard supervised learning on a dataset that contains all data points from all the tasks, and train multiple task heads for each task at the same time. This alternative baseline method can do as well or better than MAML. One way to reason about why this works is that this is equivalent to MAML with k set to 0, or with the inner learning rate set to 0.

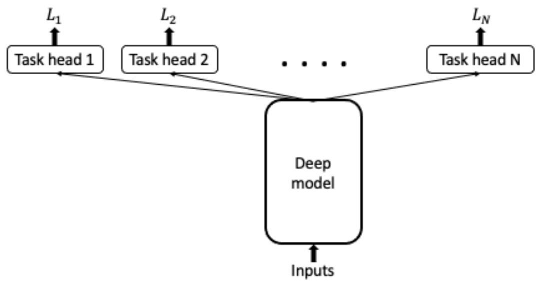


Figure 24.6: Task specific heads

Another alternative is to run MAML with a negative inner learning rate. This effectively makes the inner loop amplify the what's wrong in the model, and causes gradient descent in the outer loop to focus on parts of the model that are more wrong in the outer loop.

ANIL/Meta Opt Net/R2D2 approach

Motivation: The inner task head during training isn't very good when just initialized, so gradients aren't as helpful at improving the deep network as we might like.

Main idea: let's do a two phase approach

1. Freeze the "feature extractor network" and just optimize the linear task head.This is typically a convex problem with a closed form solution.
2. Now differentiate with respect to the parameters inside feature extractor.

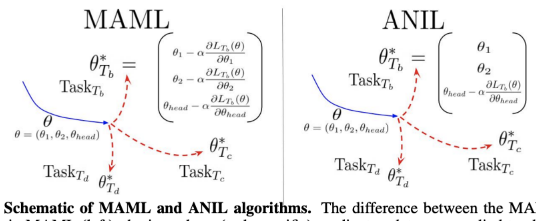

MAML
$\theta^*_{T_b} = \begin{pmatrix} \theta_1 - \alpha \frac{\partial L_{T_b}(\theta)}{\partial \theta_1} \\ \theta_2 - \alpha \frac{\partial L_{T_b}(\theta)}{\partial \theta_2} \\ \theta_{head} - \alpha \frac{\partial L_{T_b}(\theta)}{\partial \theta_{head}} \end{pmatrix}$
Task_{T_b}
θ
$\theta = (\theta_1, \theta_2, \theta_{head})$
$\theta^*_{T_c}$
Task_{T_d}
$\theta^*_{T_d}$
Task_{T_c}
ANIL
$\theta^*_{T_b} = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_{head} - \alpha \frac{\partial L_{T_b}(\theta)}{\partial \theta_{head}} \end{pmatrix}$
Task_{T_b}
θ
$\theta = (\theta_1, \theta_2, \theta_{head})$
$\theta^*_{T_c}$
Task_{T_d}
$\theta^*_{T_d}$
Task_{T_c}

Figure 4: **Schematic of MAML and ANIL algorithms.** The difference between the MAML and ANIL algorithms: in MAML (left), the inner loop (task-specific) gradient updates are applied to all parameters θ, which are initialized with the meta-initialization from the outer loop. In ANIL (right), only the parameters corresponding to the network head θ_{head} are updated by the inner loop, during training **and** testing.

Figure 24.7: MAML vs ANIL Source

REPTILE - simpler than MAML

The inner loop of REPTILE is the same as MAML: take update from our initial parameters for k steps. But in the outer loop, don't take the derivative and just update parameters in the direction of the final weights from the inner loop.