CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine **Discussion 12**

This discussion focuses on Meta Learning.

1 Overview of Meta-Learning

Machine learning models often require training with a large number of samples. However, in practice, we may have very little number of samples. Meta learning aims to solve the problem of designing a machine learning model that can quickly adapt to new tasks. In essence, instead of training a model to solve one particular task, we instead train it *learn to learn*. We consider a few concrete meta learning tasks,

- A game bot that can quickly master a new game
- Classifier that can tell whether a given image belongs to a class (possibly unseen during training time) after being provided a few examples of that class.

2 Meta Learning for Supervised Learning

For supervised meta-learning, models are typically trained over a variety of learning tasks. Each task is associated with a labeled dataset $\mathcal{D}$ that contains both feature vectors and true labels. We split each dataset into two parts, $\mathcal{D}^{tr}$ for adapting and a prediction set $\mathcal{D}^{ts}$ for evaluating. For example, in the few-shot classification framework, we consider the case where $\mathcal{D}^{tr}$ contains a few labelled examples for each of classes.

To make predictions on D^{ts} for each task, we first *adapt* our model using the training sets $\mathcal{D}^{\sqcup\nabla}$ before using the updated model to make predictions.

In regular supervised learning, we typically computed $\theta^* = \arg\min_\theta \mathcal{L}_\theta(\mathcal{D}^{tr}) = f_{\text{learn}}(\mathcal{D}^{tr})$ where $\mathcal{L}_\theta(\cdot)$ is our loss function and f_{learn} is our learning algorithm.

However, in the case of meta-learning, we attempt to learn,

$$\theta^* = \arg\min_\theta \sum_{i=1}^{n} \mathcal{L}(\phi_i, \mathcal{D}_i^{ts})$$

where $\phi_i = f_{\text{adapt}}(\theta, \mathcal{D}_i^{tr})$ is the adapted per-task parameters, obtained by running an adaptation procedure on the training set with some "hyperparameters" θ.

Notice that during meta-learning, the model is trained to learn tasks in the meta-training set, and there are two optimization loops – the inner-loop learner, which tries to solve each individual task using the training sets, and the meta-learner, which controls how the inner-loop learner learns in order to maximize how well it generalizes to the test sets.

2.1 Black-Box Meta-Learning

One way of doing black-box meta-learning is to train a recurrent model, like a RNN or an LSTM. The main idea is to read the training set sequentially, then process new inputs from the task. For example, in an image classification setting, this may involve passing in a set of (image, label) pairs of a dataset sequentially, followed by new examples that must be classified.

Here, the meta-learner uses gradient descent and the learner rolls out the recurrent model.

2.2 Non-parametric Meta Learning

Matching Networks Task of Matching Networks is to learn a classifier for any given small training set. This classifier defines some probability distribution over output labels y given a test sample x, and the classifier output is defined as the sum of labels of support weighted by an attention kernel.

$$p_\theta(y_j^{ts}|x_j^{ts}, \mathcal{D}_i^{tr}) = \sum_{k:y_k^{tr}=y_j^{ts}} p_{nearest}(x_k^{tr}|x_j^{ts})$$

The attention kernel depends on two embedding functions f and g. f is the embedding function for encoding the test sample and g is the embedding function for encoding the training set $\mathcal{D}^{tr}$.

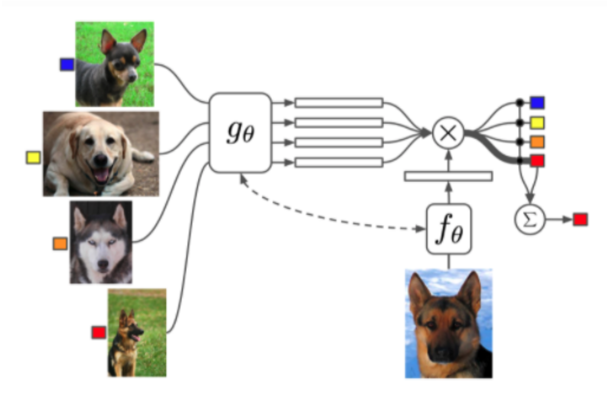

Figure 1: Architecture of Matching Networks, where f is the embedding function for encoding the test sample and g is the embedding function for encoding the training set $\mathcal{D}^{ts}$

Attention weight is defined as,

$$p_{nearest}(x_k^{tr}|x_j^{ts}) \propto \exp\Big(g(x_k^{tr}, \mathcal{D}_i^{tr})^\top f(x_j^{ts}, \mathcal{D}_i^{tr})\Big)$$

Prototypical Networks Prototypical Networks two embedding functions f and g. f is the embedding function for encoding the test sample and g is the embedding function for encoding the training set $\mathcal{D}^{tr}$. We define a *prototype* feature for every class $c \in C$, as the mean vector of the embedded training samples in the class,

$$c_y = \frac{1}{N_y} \sum_{k:y_k^{tr}=y} g(x_k^{tr})$$

Here, the distribution over classes for a given test input is given by,

$$p_\theta(y|x_j^{ts}, \mathcal{D}_i^{tr}) \propto \exp\Big(c_y^\top f(x_j^{ts})\Big)$$

2.3 Model-Agnostic Meta-Learning (MAML)

One of the biggest success stories of transfer learning has been initializing vision networks using pre-training. When approaching any new vision task, a well-known paradigm is to first collect labeled data for the task, acquire a network pre-trained on ImageNet classification, and then fine-tune the network on the collected data. This way, the neural network can learn new image-based tasks more efficiently by drawing on the features learned by the network pretrained on ImageNet.

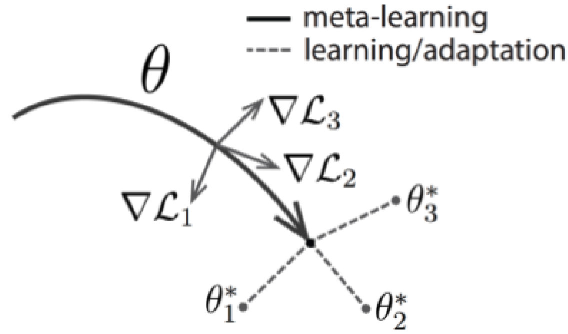

Figure 2: During the course of meta-learning (bold line), MAML optimizes for a set of parameters such that when a gradient step is taken with respect to a task $\mathcal{T}_i$ (gray lines), parameters are close to the optimal parameters θ_i^* for task $\mathcal{T}_i$

Model-Agnostic Meta-Learning (MAML) extends this idea by explicitly optimizing the "pretrained" network to be able to adapt in a few gradient steps to a variety of tasks. MAML directly optimizes over the finetuning gradient descent procedures for each task in order to compute an optimal initialization.

We let our model be f_θ with model parameters θ, and we assume we are given tasks $\mathcal{T}_i$ and associated datasets. Then, when we are given a new task, we can update the model parameters by a few gradient steps,

$$\theta_i^{'} = f_\theta(\mathcal{D}_i^{tr}) = \theta - \alpha\nabla_\theta\mathcal{L}_i(\theta, \mathcal{D}_i^{tr})$$

Notice this only optimizes for one task. To find a good generalization across a variety of tasks, we update θ across all tasks,

$$\theta \leftarrow \theta - \beta\sum_i \mathcal{L}(\theta - \alpha\nabla_\theta\mathcal{L}(\theta, \mathcal{D}_i^{tr}), \mathcal{D}_i^{ts})$$

A summary of the algorithm can be seen below:

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
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 3: This algorithm uses slightly different notations than ours, with $\nabla_\theta\mathcal{L}_{\mathcal{T}_i}(f_\theta) = \nabla_\theta\mathcal{L}_i(\theta, \mathcal{D}_i^{tr})$

Problem 1: Gradient Update of MAML

Derive the gradient update for MAML outer loop. Let $\mathcal{L}^{(0)}$ and $\mathcal{L}^{(1)}$ represent the loss at the first and second mini-batches. Assume in the inner loop, we have already performed k inner gradient steps,

$$\begin{aligned}
&\theta_0 = \theta_{meta} \\
&\theta_1 = \theta_0 - \alpha \nabla_\theta \mathcal{L}^{(0)}(\theta_0) \\
&\quad \dots \\
&\theta_k = \theta_{k-1} - \alpha \nabla_\theta \mathcal{L}^{(0)}(\theta_{k-1})
\end{aligned}$$

You are given your meta-objective in the outer loop after sampling a new data batch is,

$$\theta_{meta} \leftarrow \theta_{meta} - \beta g_{MAML}$$

Derive g_{MAML}.
Hint: Use the chain rule.

Solution 1: Gradient Update of MAML

$$\begin{aligned}
g_{MAML} &= \nabla_\theta \mathcal{L}^{(1)}(\theta_k) \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot (\nabla_{\theta_{k-1}} \theta_k) \dots (\nabla_{\theta_0} \theta_1) \cdot (\nabla_\theta \theta_0) && \text{; following the chain rule} \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \Big(\prod_{i=1}^{k} \nabla_{\theta_{i-1}} \theta_i \Big) \cdot I \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \prod_{i=1}^{k} \nabla_{\theta_{i-1}} (\theta_{i-1} - \alpha \nabla_\theta \mathcal{L}^{(0)}(\theta_{i-1})) \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \prod_{i=1}^{k} (I - \alpha \nabla_{\theta_{i-1}} (\nabla_\theta \mathcal{L}^{(0)}(\theta_{i-1})))
\end{aligned}$$

If $k = 1$, then we would have derived,

$$g_{MAML} = \nabla_{\theta_1} \mathcal{L}^{(1)}(\theta_1) \cdot (I - \alpha \nabla_\theta^2 \mathcal{L}^{(0)}(\theta_0))$$

MAML has a number of advantages. MAML does not make any restrictions or assumptions on the form of model beyond being learned through gradient descent. It is also does not need to introduce any additional parameters into the network, and only uses gradient descent in both the inner and outer loops. Lastly, the authors showed it can be easily applied to a number of domains, and the method substantially outperformed a number of existing approaches on popular few-shot image classification benchmarks.

3 Meta-Reinforcement Learning

The meta reinforcement learning setup is analogous to the meta learning set up, except we maximize expected rewards, rather than minimizing loss.

$$\theta^* = \arg\max_\theta \sum_{i=1}^{n} \mathbb{E}_{\pi_{\phi_i}(\tau)}[R(\tau)]$$

where $\phi_i = f_\theta(\mathcal{M}_i)$ and $\{\mathcal{M}_i\}_{i=1}^n$ is the set of training MDPs.

The overall configuration of meta RL algorithms similar to an ordinary RL algorithm, with the key difference between that the policy now has to, in some way, incorporate past experiences in the current MDP to adapt and perform well in the current MDP. This can be viewed as solving a partially observed MDP, where we do not directly observe what MDP we are currently solving and have to reason about the MDP using the current trajectories.

Keeping Track of History in Meta-RL One natural approach to meta-RL is to feed a history of experiences in the current MDP into the model, so the policy can adapt to the current MDP. An intuitive model that arises from this intention is using recurrent neural networks or its variants (like an LSTM).

Given a new test MDP, $\mathcal{M}$, our policy would simply take trajectories in the MDP, updating the RNN hidden states as it processes these new experiences.

During training, our algorithm would proceed as follows:

1. Sample new MDP $\mathcal{M}_i$
2. Reset the hidden state of the model
3. Collect multiple trajectories, updating hidden states as we go, and finally update the model weights using the policy gradient. Do not reset the hidden states between episodes.
4. Repeat from Step 1

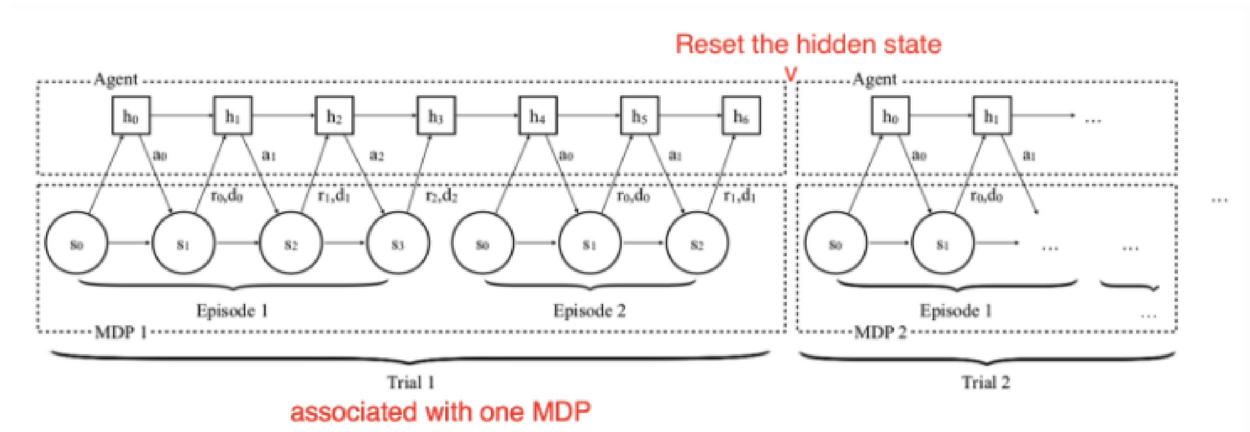

Figure 4: Model as described in the RL^2 paper. This is an illustration of the procedure of the model interacting with a series of MDPs in training time. Between episodes in each MDP, model weights are not reset, but across MDPs, they are reset.

Numerous architectures for summarizing past experiences have been developed for meta RL, including usage of standard RNN (LSTM) architecture, attention and temporal convolutions and parallel permutation-invariant context encoders.

Problem 2: Resetting the Hidden State

In the algorithm, we reset the hidden states between tasks, but not between episodes.

Solution 2: Resetting the Hidden State

We reset the hidden states between tasks to avoid information leaking across different MDPs, but we do not reset them between episodes because we would like to retain our memory across episodes in one MDP.

MAML for RL In using MAML for reinforcement learning, we specify a new loss for task $\mathcal{T}_i$ and model f_ϕ. We let $q(x_{t+1}|x_t, a_t)$ be the transition distribution, and loss $\mathcal{L}_{\mathcal{T}_i}$ correspond to the (negative) reward function r. Then the entire task $\mathcal{T}_i$ is a MDP with horizon H, and learner is allowed to query a limited number of sample trajectories for few-shot learning.

Then, our loss takes the form,

$$\mathcal{L}_{\mathcal{T}_i}(f_\phi) = -\mathbb{E}_{s_t, a_t \sim f_\phi, q_{\mathcal{T}_i}} \left[\sum_{i=1}^{H} r_i(x_t, a_t) \right]$$

Since the expected reward is not directly differentiable like in our usual supervised learning settings, we use policy gradient methods to estimate the gradient for model gradient updates and meta-optimization.

Below is the general algorithm for MAML for reinforcement learning,

Algorithm 3 MAML for Reinforcement Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
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: Sample K trajectories $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using f_θ in $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 4
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample trajectories $\mathcal{D}_i' = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using $f_{\theta_i'}$ in $\mathcal{T}_i$
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 4
11: **end while**

Figure 5: General MAML for RL algorithm. Equation 4 is the loss function as stated above