MAML Problem formulation

- Each task is a MDP with horizon H .
- Each task $\mathcal{T}_i$ contains initial state distribution $q_i(x_1)$ and transition distribution $q_i(x_{t+1} \mid x_t, a_t)$
- Model being learned $f_\theta : X_t \to A_t$, where X_t is the set of states at time t and A_t is the set of actions at time t. Such that $t \in \{1, ..., H\}$
- The loss for task $\mathcal{T}_i$ and model f_ϕ is given by:

$$L_{\mathcal{T}_i}(f_\phi) = -\mathbb{E}_{x_t, a_t \sim f_\phi, q\mathcal{T}_i}\left[\sum_{t=1}^{H} R_i(x_t, a_t)\right]$$

- Uses Policy gradient methods to estimate gradient both for model and meta-optimization, because the dynamics are unknown. Also have the option of using trust region optim.

VPG Pseudo Code (Spinning Up)

https://spinningup.openai.com/en/latest/algorithms/vpg.html

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, ...$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end for

TRPO Pseudo Code (Spinning Up)

https://spinningup.openai.com/en/latest/algorithms/trpo.html

Algorithm 1 Trust Region Policy Optimization

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K

for $k = 0, 1, 2, ...$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.

Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end for

Pseudo Code

```python
def task_rollout(task):

    cumulative_disc_rew = 0

    for i in range(run_length):

        observation = get_observation()
        action = self.model(observation)

        disc_reward, log_prob = step(action) # Discounted reward, log prob from action distrib
        cumulative_disc_rew += disc_reward * log_prob ### UNSURE ABOUT THIS ##################

        if run is done: # if in terminal state, or run_length reached
            return cumulative_disc_rew


def sample_trajectories(task, K):

    rewards_for_task = torch.zeros(K) # Cumulative sum of rewards * logprob for each K task
    for i in range(K):
        cumulative_rew = task_rollout(task)
        rewards_for_task[i] = cumulative_rew

    return rewards_for_task


def train_maml():
    Tasks = [t_1, t_2, t_3, ..., t_n] # List of tasks
    K: int                          # Number of rollouts per task
    kl_div = []                     # KL divergence for each model distrib. after update
    updated_model_reward = []

    while not done:
        for task in Tasks:

            self.model = copy.deepcopy(self.meta) # copy meta model for EACH sample
            exp_rewards_for_tasks = sample_trajectories(task, K)

            ### Update self.model with VPG
            loss = - exp_rewards_for_tasks.mean() # (Eq 4 in paper, and line 6 in VPG)
            loss.backward()
            self.optim.step()

            exp_rews_updated_model = sample_trajectories(task, K)
            updated_model_reward.append(exp_rews_updated_model)

            # KL Div. of updated and old policy
            kl = torch.kl_divergence(self.model._distrib, self.meta._distrib)
            kl_div.append(kl)

        ### Update self.meta with TRPO
        meta_grad = 0
        for exp_rew in updated_model_reward:
            meta_grad += - exp_rew.mean()
        meta_grad.backward() # This is g_k in TRPO, line 10 in MAML code
        #  meta_grad will be same size as network self.model
        ### UNSURE HERE ^^^^^ ########################

        # KL Divergence can be computed for some non closed form sample, we can say model is actually some unknown distrib.
        kl_average = kl_div.mean(axis=1) # axis=1, i.e. (10x5) -> (10x1)
        kl_hessian = compute_hessian(kl_average) # size: ((obs_dim x obs_dim) x action_dim)???
        x = inverse(kl_hessian) * meta_grad # Line 8 in TRPO pseudo code, exchange w/ conj. grad?
        ### UNSURE HERE ^^^^^ ########################

        update_val = backtrack_line_search(kl_hessian, x) # Line 9 in TRPO pseudo code
        self.meta = update_model(update_val) # Update model parameters
self.meta = torch_network()
self.model = None
```

```
train()


### Questions ---
"""



Paper says: "In order to avoid computing third derivatives, we use finite differences to compute
    the Hessian-vectorproducts for TRPO.
Where is this 3rd derivative? If I’m not mistaken it should be in the backtracking line search?

"""
```