


1. 請描述如何將助教的程式碼 (包含 classification 與 regression) 從二階的 MAML 改成一階的 MAML (作答請盡可能詳盡，以免助教誤會)，並且比較其最後在 classification 上的 accuracy (5-way-1-shot)。因此你的 GitHub 上會有 p1\_classification.py 和 p1\_regression.py 兩個檔案，分別是 classification 和 regression 的一階版本。

配分: classification 修改 (1) regression 修改 (1) report 一階做法在 classification 上的 accuracy (0.5)

(1) regression 修改：

`little_l.backward(create_graph = false)`

因為一階 loss 的公式近似為

$$\nabla_{\phi} L(\phi) = \nabla_{\phi} \sum_{n=1}^N l^n(\hat{\theta}^n) = \sum_{n=1}^N \nabla_{\phi} l^n(\hat{\theta}^n)$$


$$\frac{\partial l(\hat{\theta})}{\partial \phi_i} = \sum_j \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_j} \frac{\partial \hat{\theta}_j}{\partial \phi_i} \approx \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_i}$$

因此 meta\_loss 只需要計算當下的 task 對更新完一次的  $\theta$  的微分，不用考慮對不同 task 的  $\theta$  的微分，也不用算  $\theta$  對原始參數  $\phi$  的微分。實作上，將 `create_graph` 設為 `false`，在更新完參數第二次呼叫 `backward` 時，就只會算 loss 對當下參數的微分。

(2) classification 修改：

`grads = torch.autograd.grad(loss, fast_weights.values(), create_graph = False)`

原因與(1)相同。

(3) classification accuracy:

二階：0.9466666666666668

一階：0.8033333333333333

2. 請將 classification 的程式碼改成 inner loop 更新 5 次，並且使用 `adagrad` 與二階的 MAML，寫出其 pseudo code 與回報 accuracy (5-way-1-shot omniglot 分類任務)。並且以 outer loop 的更新次數為橫軸，分類的準確率為縱軸作圖，比較其差異。因此你的 GitHub 上要有 p2.py，對應本題的程式碼。

配分: pseudo code (1) 作圖 (1) report accuracy (0.5)

(1) pseudo code

for all tasks do

eps 設為 0.08。

inner loop learning rate 設為 0.1。

初始化一個 list 叫作 `adagrads`，存放和 model 每層 `params` 相同大小、數值全為 0 的 `tensors`。

for 每個 inner loop 的 iteration(更新 5 次) # 原本演算法的第 6~8 行

計算  $\text{grads} = \nabla_{\theta} \text{Loss}_{\text{task } i}(f_{\theta})$

每個 `adagrads` 裡的 `tensor` += 相對應 `grads` 的 `tensor` \*\* 2

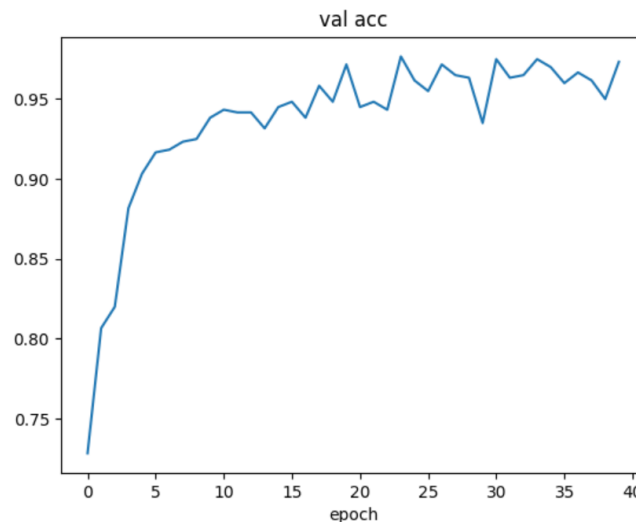
更新  $\theta$

$\theta -= \text{inner loop learning rate} * \text{grads} / \text{sqrt}(\text{adagrads} + \text{eps})$

validation 和 testing 時，更新次數和 learning rate 和訓練時相同

因為 `grads` 裡存的值偏小，若 `eps` 設定太小，參數一次會更新太大一步，validation accuracy 會完全壞掉；若 `eps` 設定太大，`adagrads` 裡累積的值對 learning rate 的影響比率會太小。折衷調整認為 `eps` 設為 0.08 能有較好的結果。

(2) validation accuracy 對 outer loop epoch 作圖



(3) testing accuracy: 0.9633333333333334

訓練時，只更改 inner loop 的訓練次數和參數(如(1)所述)，outer loop 的訓練次數和參數和 sample code 相同。testing 時，inner loop 更新 5 次，參數更新方式和 training 時 inner loop 更新方式相同。

3. 實作論文 "How to train your MAML" (<https://arxiv.org/abs/1810.09502>) 中的一個 tip，解釋你使用的 tip 並且比較其在 5-way-1-shot 的 omniglot 分類任務上的 accuracy。助教其實已經實作了一個，請找出是哪一個 tip 並且不要重複。因此你的 GitHub 上要有 p3.py，對應本題的程式碼。

配分：report 實作 tip 後的 accuracy (1) 解釋你使用的 tip (1) 找出助教實作的 tip (0.5)

(1) validation accuracy: 0.9766666666666666

testing accuracy: 0.9533333333333333

inner loop 更新五次，並使用 `adagrad`。outer loop optimizer 使用

`Adam`，learning rate 設為，並使用 `cosine annealing`，跑 80 個

epoch。Testing 時，inner loop 更新 5 次，參數更新方式和 training 時

inner loop 更新方式相同。

(2) 使用的 tip

實作 Cosine Annealing of Meta-Optimizer Learning Rate。對 meta optimizer (outer loop 的 learning rate) 使用 cosine annealing scheduling。實作方式：對 meta\_model 的 optimizer 使用 `torch.optim.lr_scheduler.CosineAnnealingLR`。

(3) 助教實作的 tip：

Per-Step Batch Normalization Running Statistics。

4. 請實作 `reptile` 在 `omniglot dataset` 上，訓練 5-way-1-shot 的分類任務，並且回報其 accuracy。這題應該在 GitHub 上會有 `reptile.sh` 的 shell script，執行方式詳見投影片。

配分：程式碼 (2) 回報 acc (0.5)

實作方式：

inner loop 在 `train_set` 上更新五次，inner learning rate = 0.1、使用 `Adagrad`。

將五次 gradient 相加，並以 learning rate = 0.001 更新 meta model 的參數。

outer loop 跑 40 個 epoch。

validation accuracy: 0.9633333333333334

testing accuracy: 0.93