A Gradient Manipulation Methods

In this section, we provide a brief overview of representative gradient manipulation methods in multitask/multiobjective optimization. Specifically, we will also discuss the connections among these methods.

Multiple Gradient Descent Algorithm (MGDA) [9, 35] The MGDA algorithm is one of the earliest gradient manipulation methods for multitask learning. In MGDA, the per step update d_t is found by solving

$$\max_{d\in\mathbb{R}^m} \min_{i\in[k]} \nabla \ell_{i,t}^\top d - \frac{1}{2}\|d\|^2.$$

As a result, the solution d^* of MGDA optimizes the "worst improvement" across all tasks or equivalently seeks an *equal* descent across all task losses as much as possible. But in practice, MGDA suffers from slow convergence since the update d^* can be very small. For instance, if one task has a very small loss scale, the progress of all other tasks will be bounded by the progress on this task. Note that the original objective in (6) is similar to the MGDA objective in the sense that we can view optimizing (6) as optimizing the log of the task losses. Hence, when we compare FAMO against MGDA, one can regard FAMO as balancing the *rate* of loss improvement while MGDA balances the absolute improvement across task losses.

Projecting Gradient Descent (PCGRAD) [43] PCGRAD initializes $v_{\text{PC}}^i = \nabla \ell_{i,t}$, then for each task i, PCGRAD loops over all task $j \neq i$ (in a random order, which is crucial as mentioned in [43]) and removes the "conflict"

$$v_{\text{PC}}^i \leftarrow v_{\text{PC}}^i - \frac{{v_{\text{PC}}^i}^\top \nabla \ell_{j,t}}{\|\ell_{j,t}\|^2} \nabla \ell_{j,t} \quad \text{if} \quad {v_{\text{PC}}^i}^\top \nabla \ell_{j,t} < 0.$$

In the end, PCGRAD produces $d_t = \frac{1}{k}\sum_{i=1}^k v_{\text{PC}}^i$. Due to the construction, PCGRAD will also help improve the "worst improvement" across all tasks since the "conflicts" have been removed. However, due to the stochastic iterative procedural of this algorithm, it is hard to understand PCGRAD from a first principle approach.

Conflict-averse Gradient Descent (CAGRAD) [24] d_t is found by solving

$$\max_{d\in\mathbb{R}^m} \min_{i\in[k]} \nabla \ell_{i,t}^\top d \quad \text{s.t.} \quad \|d - \nabla \ell_{0,t}\| \leq c\|\nabla \ell_{0,t}\|.$$

Here, $\ell_{0,t} = \frac{1}{k}\sum_{i=1}^k \ell_{i,t}$. CAGRAD seeks an update d_t that optimizes the "worst improvement" as much as possible, conditioned on that the update still decreases the average loss. By controlling the hyperparameter c, CAGRAD can recover MGDA ($c \to \infty$) and the vanilla averaged gradient descent ($c \to 0$). Due to the extra constraint, CAGRAD provably converges to the stationary points of ℓ_0 when $0 \leq c < 1$.

Impartial Multi-Task Learning (IMTL-G) [25] IMTL-G finds d_t such that it shares the same cosine similarity with any task gradients:

$$\forall i \neq j, \quad d_t^\top \frac{\nabla \ell_{i,t}}{\|\nabla \ell_{i,t}\|} = d_t^\top \frac{\nabla \ell_{j,t}}{\|\nabla \ell_{j,t}\|}, \quad \text{and} \quad d_t = \sum_{i=1}^k w_{i,t} \nabla \ell_{i,t}, \quad \text{for some} \quad w_t \in \mathbb{S}_k.$$

The constraint that $d_t = \sum_{i=1}^k w_{i,t}\nabla \ell_{i,t}$ is for preventing the problem from being under-determined. From the above equation, we can see that IMTL-G ignores the "size" of each task gradient and only cares about the "direction". As a result, **one can think of IMTL-G as a variant of MGDA that applies to the normalized gradients**. By doing so, IMTL-G does not suffer from the straggler effect due to slow objectives. Furthermore, one can **view IMTL-G as the equal angle descent**, which is also proposed in Katrutsa et al. [17], where the objective is to find d such that

$$\forall i \neq j, \qquad \cos(d, \nabla \ell_{i,t}) = \cos(d, \nabla \ell_{j,t}).$$

NASHMTL[32] NASHMTL finds d_t by solving a bargaining game treating the local improvement of each task loss as the utility for each task:

$$\max_{d\in\mathbb{R}^m,\|d\|\le 1}\sum_{i=1}^{k}\log\left(\nabla\ell_{i,t}^\top d\right).$$

Note that the objective of NASHMTL implicitly assumes that there exists d such that $\forall\ i,\ \ \nabla\ell_{i,t}^\top d>0$ (otherwise we reach the Pareto front). It is easy to see that

$$\max_{\|d\|\le 1}\sum_{i=1}^{k}\log\left(\nabla\ell_{i,t}^\top d\right)=\max_{\|d\|\le 1}\sum_{i=1}^{k}\log\langle\frac{\nabla\ell_{i,t}}{\|\nabla\ell_{i,t}\|},d\rangle=\max_{\|d\|\le 1}\sum_{i=1}^{k}\log\cos\left(\nabla\ell_{i,t},d\right).$$

Therefore, due to the $\log$, NASHMTL also ignores the "size" of task gradients and only cares about their "directions". Moreover, denote $u_i=\frac{\nabla\ell_{i,t}}{\|\nabla\ell_{i,t}\|}$. Then, according to the KKT condition, we know:

$$\sum_i\frac{u_i}{u_i^\top d}-\alpha d=0,\quad \alpha\ge 0\qquad\Longrightarrow\qquad d=\frac{1}{\alpha}\sum_i\frac{1}{u_i^\top d}u_i.$$

Consider when $k=2$, if we take the *equal angle descent* direction: $d_\angle=(u_1+u_2)/2$ (note that as u_1 and u_2 are normalized, their bisector is just their average). Then it is easy to check that

$$d_\angle=\frac{1}{\alpha}\left(\frac{2}{u_1^\top(u_1+u_2)}u_1+\frac{2}{u_2^\top(u_1+u_2)}u_2\right),\ \text{ where }\ \alpha=\frac{u_1^\top(u_1+u_2)}{4}=\frac{u_2^\top(u_1+u_2)}{4}.$$

As a result, we can see that **when $k=2$, NASHMTL is equivalent to IMTL-G (or the equal angle descent)**. However, when $k>2$, this is not in general true.

Remark Note that all of these gradient manipulation methods require computing and storing K task gradients before applying f to compute d_t, which often involves solving an additional optimization problem. Hence, these methods can be slow for large K and large model sizes.

B Amortizing other Gradient Manipulation Methods

Although FAMO uses iterative update on w, it is not immediately clear whether we can apply the same amortization easily on other existing gradient manipulation methods. In this section, we discuss such possibilities and point out the challenges.

Amortizing MGDA This is almost the same as in FAMO, except that MGDA acts on the original task losses while FAMO acts on the log of task losses.

Amortizing PCGRAD For PCGRAD, finding the final update vector requires iteratively projecting one task gradient to the other, so there is no straightforward way of bypassing the computation of task gradients.

Amortizing IMTL-G The task weighting in IMTL-G is computed by a series of matrix-matrix and matrix-vector products using task gradients [25]. Hence, it is also hard to amortize its computation over time.

Therefore, we focus on deriving the amortization for CAGRAD and NASHMTL.

Amortizing CAGRAD For CAGRAD, the dual objective is

$$\min_{w\in\mathbb{S}_k}F(w)=g_w^\top g_0+c\|g_w\|\|g_0\|,\tag{15}$$

where $g_0=\nabla\ell_{0,t}$ and $g_w=\sum_{i=1}^k w_i\nabla\ell_i$. Denote

$$G=\begin{bmatrix}\nabla\ell_{1,t}^\top\\ \vdots\\ \nabla\ell_{k,t}^\top\end{bmatrix}.$$

Now, if we take the gradient with respect to w in (15), we have:

$$\frac{\partial F}{\partial w} = G^\top g_0 + c\frac{\|g_0\|}{\|g_w\|}G^\top g_w. \tag{16}$$

As a result, in order to approximate this gradient, one can separately estimate:

$$\begin{aligned} G^\top g_0 &\approx \frac{\ell(\theta) - \ell(\theta - \alpha g_0)}{\alpha} \\ G^\top g_w &\approx \frac{\ell(\theta) - \ell(\theta - \alpha g_w)}{\alpha} \\ \|g_0\| &\approx \sqrt{1^\top G^\top g_0} \\ \|g_w\| &\approx \sqrt{w^\top G^\top g_w} \end{aligned}. \tag{17}$$

Once all these are estimated, one can combine them together to perform a single update on w. But note that this will require 3 forward and backward passes through the model, making it harder to implement in practice.

Amortizing NashMTL Per derivation from NashMTL [32], the objective is to solve for w:

$$G^\top G w = 1 \oslash w. \tag{18}$$

One can therefore form an objective:

$$\min_w F(w) = \left\|G^\top G w - 1 \oslash w\right\|_2^2. \tag{19}$$

Taking the derivative of F with respect to w, we have

$$\frac{\partial F}{\partial w} = 2G^\top G\left(G^\top g_w - 1 \oslash w\right) + 2\left(G^\top g_w - 1 \oslash w\right) \oslash (w \odot w). \tag{20}$$

Therefore, to approximate the gradient of w, one needs to first estimate

$$G^\top g_w \approx \frac{L(\theta) - L(\theta - \alpha g_w)}{\alpha} = \eta. \tag{21}$$

Then we estimate

$$G^\top G(\eta - 1 \oslash w) \approx \frac{L(\theta) - L(\theta - \alpha G(\eta - 1 \oslash w))}{\alpha}. \tag{22}$$

Again, this results in 3 forward and backward passes through the model, let alone the overhead of resetting the model back to θ (requires a copy of the original weights).

In short, though it is possible to derive fast approximation algorithm to approximate the gradient update on w for some of the existing gradient manipulation methods, it often involves much more complicated computation compared to that of FAMO.

C FAMO Pseudocode in PyTorch

We provide the pseudocode for FAMO in Algorithm 2. To use FAMO, one just first compute the task losses, call `get_weighted_loss` to get the weighted loss, and do the normal backpropagation through the weighted loss. After that, one call `update` to update the task weighting.

D Toy Example

We provide the task objectives for the toy example in the following. The model parameter $\theta = (\theta_1, \theta_2) \in \mathbb{R}^2$ and the task objectives are L^1 and L^2:

$$\begin{aligned} &L^1(\theta) = 0.1 \cdot (c_1(\theta) f_1(\theta) + c_2(\theta) g_1(\theta)) \text{ and } L^2(\theta) = c_1(\theta) f_2(\theta) + c_2(\theta) g_2(\theta), \text{ where} \\ &f_1(\theta) = \log\left(\max(|0.5(-\theta_1 - 7) - \tanh(-\theta_2)|,\ 0.000005)\right) + 6, \\ &f_2(\theta) = \log\left(\max(|0.5(-\theta_1 + 3) - \tanh(-\theta_2) + 2|,\ 0.000005)\right) + 6, \\ &g_1(\theta) = \left((-\theta_1 + 7)^2 + 0.1 * (-\theta_2 - 8)^2\right)/10 - 20, \\ &g_2(\theta) = \left((-\theta_1 - 7)^2 + 0.1 * (-\theta_2 - 8)^2\right)/10 - 20, \\ &c_1(\theta) = \max(\tanh(0.5 * \theta_2),\ 0) \text{ and } c_2(\theta) = \max(\tanh(-0.5 * \theta_2),\ 0). \end{aligned}$$

Algorithm 2 Implementation of FAMO in PyTorch-like Pseudocode

```
class FAMO:
def __init__(self, num_tasks, min_losses, α=0.025, γ=0.001):
    # min_losses  (num_tasks,) the loss lower bound for each task.
    self.min_losses = min_losses
    self.xi = torch.tensor([0.0] * num_tasks, requires_grad=True)
    self.xi_opt = torch.optim.Adam([self.xi], lr=α, weight_decay=γ)

def get_weighted_loss(self, losses):
    # losses  (num_tasks,)
    z = F.softmax(self.xi, -1)
    D = losses - self.min_losses + 1e-8
    c = 1 / (z / D).sum().detach()
    loss = (c * D.log() * z).sum()
    return loss

def update(self, prev_losses, curr_losses):
    # prev_losses  (num_tasks,)
    # curr_losses  (num_tasks,)
    delta = (prev_losses - self.min_losses + 1e-8).log() -
            (curr_losses - self.min_losses + 1e-8).log()
    with torch.enable_grad():
        d = torch.autograd.grad(F.softmax(self.xi, -1),
                                self.xi,
                                grad_outputs=delta.detach())[0]
    self.xi_opt.zero_grad()
    self.xi.grad = d
    self.xi_opt.step
```

E Experimental Results with Error Bars

We followed the exact experimental setup from NASHMTL [32]. Therefore, the numbers for baseline methods are taken from their original paper. In the following, we provide FAMO's result with error bars.

Method	Segmentation		Depth		Surface Normal					$\Delta m\% \downarrow$
	mIoU ↑	Pix Acc ↑	Abs Err ↓	Rel Err ↓	Angle Dist ↓		Within t° ↑			
					Mean	Median	11.25	22.5	30	
FAMO (mean)	38.88	64.90	0.5474	0.2194	25.06	19.57	29.21	56.61	68.98	-4.10
FAMO (stderr)	±0.54	±0.21	±0.0016	±0.0026	±0.06	±0.09	±0.17	±0.19	±0.14	±0.39

Table 5: Results on NYU-v2 dataset (3 tasks). Each experiment is repeated over 3 random seeds and the mean is reported. The best average result is marked in bold. **MR** and $\boldsymbol{\Delta m}\%$ are the main metrics for MTL performance.

Method	μ	α	ϵ_{HOMO}	ϵ_{LUMO}	$\langle R^2 \rangle$	ZPVE	U_0	U	H	G	c_v	$\Delta m\% \downarrow$
	MAE ↓											
FAMO (mean)	0.15	0.30	94.0	95.2	1.63	4.95	70.82	71.2	71.2	70.3	0.10	58.5
FAMO (stderr)	±0.0046	±0.0070	±3.074	±2.413	±0.0211	±0.0871	±2.17	±2.19	±2.19	±2.21	±0.0026	±3.26

Table 6: Results on QM-9 dataset (11 tasks). Each experiment is repeated over 3 random seeds and the mean is reported. The best average result is marked in bold. **MR** and $\boldsymbol{\Delta m}\%$ are the main metrics for MTL performance.

Method	CityScapes					CelebA
	Segmentation		Depth		$\Delta m\% \downarrow$	$\Delta m\% \downarrow$
	mIoU ↑	Pix Acc ↑	Abs Err ↓	Rel Err ↓		
FAMO (mean)	74.54	93.29	0.0145	32.59	8.13	1.21
FAMO (stderr)	±0.11	±0.04	±0.0009	±1.06	±1.98	±0.24

Table 7: Results on CityScapes (2 tasks) and CelebA (40 tasks) datasets. Each experiment is repeated over 3 random seeds and the mean is reported. The best average result is marked in bold. **MR** and $\boldsymbol{\Delta m}\%$ are the main metrics for MTL performance.