Learning Conflict-Noticed Architecture for Multi-Task Learning

Zhixiong Yue[1,2], **Yu Zhang**[1,3,*], **Jie Liang**[2]

[1] Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
[2] University of Technology Sydney
[3] Peng Cheng Laboratory, Shenzhen, China
yuezx@mail.sustech.edu.cn, yu.zhang.ust@gmail.com, jie.liang@uts.edu.au

Abstract

Multi-task learning has been widely used in many applications to enable more efficient learning by sharing part of the architecture across multiple tasks. However, a major challenge is the gradient conflict when optimizing the shared parameters, where the gradients of different tasks could have opposite directions. Directly averaging those gradients will impair the performance of some tasks and cause negative transfer. Different from most existing works that manipulate gradients to mitigate the gradient conflict, in this paper, we address this problem from the perspective of architecture learning and propose a Conflict-Noticed Architecture Learning (CoNAL) method to alleviate the gradient conflict by learning architectures. By introducing purely-specific modules specific to each task in the search space, the CoNAL method can automatically learn when to switch to purely-specific modules in the tree-structured network architectures when the gradient conflict occurs. To handle multi-task problems with a large number of tasks, we propose a progressive extension of the CoNAL method. Extensive experiments on computer vision, natural language processing, and reinforcement learning benchmarks demonstrate the effectiveness of the proposed methods. The code of CoNAL is publicly available.[1].

Introduction

Multi-Task Learning (MTL) (Caruana 1997; Zhang and Yang 2021) aims to improve the generalization performance of multiple learning tasks. Compared with single-task learning, MTL can learn multiple tasks simultaneously to reduce the overall training cost and gain knowledge sharing from different tasks. Moreover, in some cases, MTL models could make multiple predictions in one forward propagation, which reduces the inference latency. Therefore, MTL has drawn much attention in recent years.

However, learning multiple tasks simultaneously can be challenging because parameter sharing may lead to negative transfer (Ruder 2017) with some empirical evidence in (Lee, Yang, and Hwang 2016; Standley et al. 2020; Tang et al. 2020). As empirically analyzed in (Yu et al. 2020a; Chen et al. 2020; Dery, Dauphin, and Grangier 2021), one main reason for this issue is the conflicting gradients of different tasks with respect to shared model parameters, which will be updated in opposite directions, leading to unsatisfactory performance. To mitigate the gradient conflict, several gradient manipulation methods (Yu et al. 2020a; Liu et al. 2021a; Chen et al. 2018; Wang et al. 2020; Liu et al. 2021b) are proposed to manipulate task gradients by adjusting gradient magnitudes or gradient directions or both of them for all the tasks.

Module	f_S^1	f_S^2	f_S^3	f_S^4	f_S^5
HPS	59.26	51.68	42.22	39.36	49.49
LTB	41.08	38.86	39.65	47.05	24.44
CoNAL	0.34	0.19	0.14	0.13	0.00

Table 1: The ratio of negative cosine similarities (%) between the gradient of different tasks with respect to shared parameters. $\{f_S^{(1)}, \ldots, f_S^{(5)}\}$ denotes the modules in the sharing architecture while learning architecture. Ratio is calculated by going through all samples in the training set.

The occurrence of gradient conflict reflects that all the tasks are not strongly related to each other, given the adopted architecture. All the aforementioned works to alleviate the gradient conflict adopt the hard parameter sharing (HPS) architecture that uses a shared encoder for all the tasks with task-specific decoders for each task. Though the HPS architecture has been proven to be effective for many MTL problems and is widely used in MTL, it implicitly requires that all the tasks should be highly related to each other. For some complicated datasets such as the NYUv2 dataset (Silberman et al. 2012), such requirement cannot be satisfied and the HPS architecture does not work well on this dataset (Brüggemann et al. 2021; Sun et al. 2021). Hence, the gradient conflict issue could be the consequence of the use of an improper architecture for a given MTL problem.

To verify that, we compare the HPS architecture with the Learning to Branch (LTB) method (Guo, Lee, and Ulbricht 2020), which is to learn an architecture for a given MTL problem, on the NYUv2 dataset. We follow (Du et al. 2018; Yu et al. 2020a) to calculate the cosine similarity between gradients of different tasks with respect to shared parameters and put detailed settings of this experiment in Appendix

*Corresponding author.

[1]https://github.com/yuezhixiong/CoNAL

A.8. According to Table 1, the gradient conflict in each module of the learned architecture in LTB is alleviated compared with the fixed HPS architecture. This suggests that architecture learning could be another way to mitigate the gradient conflict.

To the best of our knowledge, none of existing architecture learning methods for MTL considers to mitigate the gradient conflict during the architecture learning process, which is what we aim to do in this work. According to Table 1, we can see that the gradient conflict still severely affects the learned architecture in the LTB model. One possible reason is that LTB and other branch-based architecture learning methods include only partially-specific modules, which are first shared by all the tasks and then become specific for one or more tasks, in the search space. Based on its definition, during the architecture learning process, partially-specific modules will be affected by gradients of all the tasks and so weakly-related or even unrelated tasks will affect the learning of both parameters and architectures, leading to a suboptimal architecture which may suffer from gradient conflict.

To kill two birds with one stone, we propose a Conflict-Noticed Architecture Learning (CoNAL) method, which is to handle the gradient conflict during the architecture learning process. Different from existing architecture learning methods (Lu et al. 2017; Guo, Lee, and Ulbricht 2020; Bruggemann et al. 2020; Vandenhende et al. 2020; Zhao et al. 2021) for MTL, we are the first to introduce purely-specific modules into the search space of learnable architectures which also have all-shared modules shared by all the tasks. During the architecture learning process, the proposed CoNAL method could perform the conflict noticing operation to adaptively choose to use purely-specific modules when the gradient conflict is detected in the all-shared modules. As demonstrated in Table 1, the proposed CoNAL method achieves a very low ratio of the gradient conflict. Moreover, to efficiently handle MTL problems with a large number of tasks, we propose a progressive version of CoNAL called CoNAL-Pro, which can reduce the storage cost and find task subgroups during the architecture learning process. Experiments on Computer Vision (CV), Natural Language Processing (NLP), and Reinforcement Learning (RL) benchmark datasets demonstrate the effectiveness of the proposed methods. The main contributions of this paper are three-fold.

1. We propose the CoNAL method to mitigate the gradient conflict from the perspective of architecture learning.
2. In the CoNAL method, we firstly introduce both the purely-specific modules in the search space of multi-task architectures and the conflict noticing operation to guide the architecture learning process.
3. Extensive experiments on three challenging domains demonstrate the effectiveness of the proposed methods.

The CoNAL Method

In this section, we introduce the proposed CoNAL method.

Search Space

As an architecture learning method, the CoNAL method defines a search space consisting of modules, each of which could be a fully connected layer or a sophisticated ResNet block/layer depending on the MTL problem under investigation. Different from existing architecture learning methods (Lu et al. 2017; Guo, Lee, and Ulbricht 2020; Bruggemann et al. 2020; Vandenhende et al. 2020; Zhao et al. 2021), which learn a tree-structured network architecture with partially-specific modules, all the modules in CoNAL are classified into two types: all-shared module and purely-specific module, which are defined as follows.

Definition 0.1 (Partially-specific module)**.** When a module in the search space is updated by the gradient back-propagated from the loss of all tasks first and later from a fixed set of tasks during the search process, this module is said to be a *partially-specific module*.

Definition 0.2 (All-shared module)**.** When a module in the search space is always updated by the gradient back-propagated from losses of all the tasks during the search process, this module is said to be an *all-shared module*.

Definition 0.3 (Purely-specific module)**.** When a module in the search space is only updated by the gradient back-propagated from the loss of a fixed task during the search process, this module is said to be a *purely-specific module* for that task.

According to Definition 0.3, we can see that parameters in a purely-specific module are only updated by the loss of one task, which is different from the partially-specific module defined in Definition 0.1. Purely-specific modules are a key ingredient for CoNAL to mitigate gradient conflict. Specifically, when the gradient of a task is detected to be conflicting in the all-shared module, this task could use purely-specific modules to form the encoder without affecting the learning of other tasks.

Formally, for m learning tasks $\{\mathcal{T}_i\}_{i=1}^m$, the CoNAL method has a shared encoder network f_S consisting of $(P-1)$ all-shared modules for all the tasks, m task-specific architecture parameters $\{\boldsymbol{\alpha}_t\}_{t=1}^m$ to decide branch points for m tasks, and m task-specific encoder networks $\{h_t\}_{t=1}^m$ consisting of purely-specific modules for the m tasks. Hence, for task t, its model consists of all-shared modules from f_S, purely-specific modules in h_t, $\boldsymbol{\alpha}_t = (\alpha_t^{(1)}, \ldots, \alpha_t^{(P)})$, and a decoder network g_t. Here P is equal to the total number of all possible branch points before or after each module in f_S. As a binary parameter, $\alpha_t^{(p)} \in \{0, 1\}$ indicates whether task t branches at the branch point p from f_S to h_t. When $\alpha_t^{(p)}$ equals 1, there will be a branch to feed the output of the $(p-1)$-th module in f_S to the p-th module in h_t to form a network for task t, which is illustrated in Figure 1(b). Moreover, the sum of entries in $\boldsymbol{\alpha}_t$ should be 1 for any t, indicating that there is only one branch point for each task.

Figure 1 illustrates the difference between search spaces of some existing architecture learning works (i.e., LTB (Guo, Lee, and Ulbricht 2020) and BMTAS (Bruggemann et al. 2020)), and the CoNAL method. The search space of existing methods only has partially-specific modules. If task i is totally unrelated to other tasks, all the partially-specific modules will be updated by the gradient of the loss of task i, which can be detrimental to the performance of some

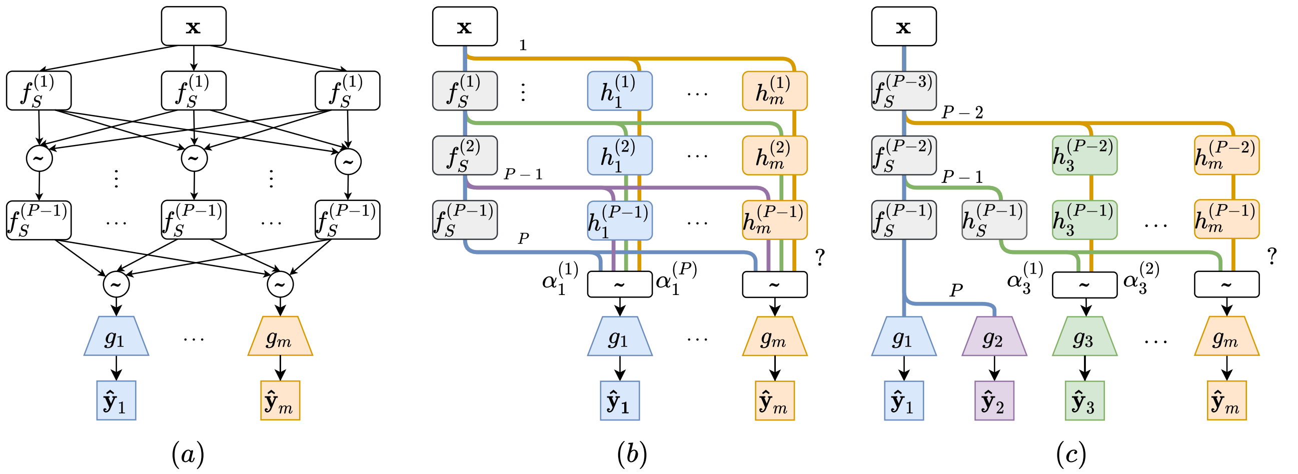

Figure 1: Illustration of the search space in various methods. Figures (a), (b), and (c) represent for the search space of the LTB, CoNAL, and CoNAL-Pro methods, respectively. Blocks represent computational modules and edges between blocks denote data flows. Blocks with the grey color stand for all-shared modules and blocks with other colors are for task-specific modules. $\{f_S^{(1)}, \ldots, f_S^{(P-1)}\}$ denotes the all-shared modules for all the tasks. In figure (a), all-shared modules can transform into partially-specific modules at the later stage of learning. In figure (b) and (c), $\{1, \ldots, P\}$ denotes possible branch points in the search space. $\{h_i^{(1)}, \ldots, h_i^{(P-1)}\}$, $\{\alpha_i^{(1)}, \ldots, \alpha_i^{(P)}\}$, and g_i $(1 \le i \le m)$ denote the task-specific encoder, task-specific architecture parameters, and task-specific decoder, respectively, for task i. $\mathbf{x}$ and $\hat{y}_i$ denote the input data and the prediction for task i. A softmax operation is represented as "$\sim$" and "?" means to search within candidate architectures.

other tasks. Differently, the CoNAL method could choose h_i for task i, making the learning of other tasks unaffected by task i. The proposed search space in the CoNAL method includes both single-task learning and HPS architectures as two extremes. When all the tasks are unrelated to each other, the network architecture in the CoNAL method could become separate networks by choosing $\{h_t\}$ for each task. For highly related or even identical tasks, the network architecture of the CoNAL method could become the HPS network by choosing f_S only. In most cases, the learned architecture by the CoNAL method is between the two extremes and more complex than them.

Architecture Learning

The CoNAL method is to find an architecture that circumvents gradient conflict for m tasks $\{\mathcal{T}_i\}_{i=1}^m$ by learning architecture parameters $\{\boldsymbol{\alpha}_t\}_{t=1}^m$.

Branch searching. If task t branches at the branch point p in f_S to h_t, $\alpha_t^{(p)}$ is set to 1 and α_t^i is set to 0 for $1 \le i \le P$, $i \ne p$. Then in this case, the entire encoder network for task t consisting of the first $(p-1)$ all-shared modules in f_S and the last $(P-p)$ purely-specific modules in h_t is denoted by $h_t(f_S(\mathbf{x}, p), p)$, where $f_S(\cdot, p)$ denotes the output of the $(p-1)$-th module in f_S and $h_t(\cdot, p)$ denotes the output of h_t starting from the p-th module. To learn $\{\boldsymbol{\alpha}_t\}$ via stochastic gradient descent methods, we relax $\{\boldsymbol{\alpha}_t\}$ to be continuous and use them to define the probability of branching at each branch point via the softmax function. Hence, the output of the entire encoder network for task t is formulated as

$$\mathbf{o}_t(\mathbf{x}, \boldsymbol{\alpha}_t) = \sum_{p=1}^{P} \frac{\exp(\alpha_t^{(p)})}{\sum_{p'}^{P} \exp(\alpha_t^{(p')})} h_t(f_S(\mathbf{x}, p), p).$$

Then $\mathbf{o}_t(\mathbf{x}, \boldsymbol{\alpha}_t)$ is fed into the decoder g_t to generate the prediction and hence the loss for task t is formulated as

$$\mathcal{L}_t(\boldsymbol{\theta}_t, \boldsymbol{\alpha}_t) = \frac{1}{|\mathcal{D}|} \sum_{\mathbf{x} \in \mathcal{D}} l_t(\mathbf{y}_t, g_t(\mathbf{o}_t(\mathbf{x}, \boldsymbol{\alpha}_t))),$$

where $\boldsymbol{\theta}_t$ includes all the parameters in f_S, h_t, and g_t, without loss of generality all the tasks are assumed to share the training set $\mathcal{D}$, $|\mathcal{D}|$ denotes the size of $\mathcal{D}$, $\mathbf{y}_t$ denotes the label of $\mathbf{x}$ in task t, and l_t denotes the loss function for task t. Then the loss over m tasks is formulated as

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha}) = \frac{1}{m} \sum_{t=1}^{m} \mathcal{L}_t(\boldsymbol{\theta}_t, \boldsymbol{\alpha}_t), \tag{1}$$

where $\boldsymbol{\theta}$ denotes all the model parameters and $\boldsymbol{\alpha} = (\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_m)$.

Architecture determining. Here architecture parameters $\boldsymbol{\alpha}$ are viewed as hyperparameters and we adopt a bi-level formulation to learn both model and architecture parameters as

$$\min_{\boldsymbol{\alpha}} \mathcal{L}_{val}(\boldsymbol{\theta}^*(\boldsymbol{\alpha}), \boldsymbol{\alpha}) \;\text{ s.t. } \boldsymbol{\theta}^*(\boldsymbol{\alpha}) = \arg\min_{\boldsymbol{\theta}} \mathcal{L}_{tr}(\boldsymbol{\theta}, \boldsymbol{\alpha}), \tag{2}$$

where $\mathcal{L}_{tr}(\cdot,\cdot)$ denotes the total loss defined in Eq. (1) on the training dataset and $\mathcal{L}_{val}(\cdot,\cdot)$ denotes the total loss on the validation dataset. We adopt the gradient-based hyperparameter optimization algorithm with the efficient first-order approximation as in (Franceschi et al. 2018; Liu, Simonyan, and Yang 2019; Ye et al. 2021) to solve problem (2). After solving it, we can learn the branching architecture for task t by determining the branch point as $\boldsymbol{\alpha}_t^* = \arg\max_p(\{\alpha_t^{(p)}\}_{p=1}^P)$, where $\boldsymbol{\alpha}_t^*$ is a P-dimensional one-hot vector with the maximum position being 1. After learning the branch points, the unused purely-specific modules are removed from the final architecture to reduce the number of parameters. For example, if a task selects a branch point $p = P$, it uses only all-shared modules, and hence all purely-specific modules for this task are removed.

Retraining. After determining the architecture, the entire model is trained from scratch to obtain the final model. Inspired by (Lin et al. 2022), the objective function in the i-th retraining iteration is formulated as

$$\min_{\boldsymbol{\theta}} \mathcal{L}^{(i)}(\boldsymbol{\theta}, \boldsymbol{\alpha}^*) = \sum_{t=1}^{m} w_t^i \mathcal{L}_t(\boldsymbol{\theta}_t, \boldsymbol{\alpha}_t^*), \tag{3}$$

where w_t^i is a sampled loss weight from the uniform distribution over $[0, 1]$ for task t in the i-th iteration after normalizing to satisfy $w_t^i \geq 0$ and $\sum_{t=1}^{m} w_t^i = 1$. Such setting of $\{w_t^i\}$ could avoid the cost of manually tuning loss weights. The retraining process can be naturally combined with gradient manipulation strategies, which is studied in Section .

Conflict Noticing

During the architecture learning process, we propose the conflict noticing operation to handle the gradient conflict. When there is a gradient conflict detected in a all-shared module between two tasks where the cosine similarity of the two task gradients on this module is negative, we will zero out the gradients on this and successive all-shared modules and force all the tasks to update the subsequent purely-specific modules. Specifically, the conflict noticing operation on all-shared module p at each training iteration is formulated as

$$\text{grad}_t = \nabla_{\theta_S} \mathcal{L}_t(\boldsymbol{\theta_t}, \boldsymbol{\alpha_t}) \tag{4}$$

$$\nabla_{\theta_S^p} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha}) = \begin{cases} \sum_{t=1}^{m} \text{grad}_t^{(p)} & \text{if } \cos(\text{grad}_i^{(p')}, \text{grad}_j^{(p')}) \geq 0, \\ & \forall i, j \; \forall p' \leq p, \\ 0 & \text{otherwise}, \end{cases} \tag{5}$$

where $\cos(\cdot, \cdot)$ computes the cosine similarity between two vectors, θ_S denotes parameters in all-shared modules, θ_S^p denotes parameters in the p-th all-shared module, grad_t denotes the gradient of the loss in task t with respect to θ_S, and $\text{grad}_t^{(p)}$ denotes the gradient of the loss in task t with respect to θ_S^p. According to Eq. (5), when there exist a pair of tasks (e.g., tasks i and j) such that $\cos(\text{grad}_i^{(p)}, \text{grad}_j^{(p)}) < 0$, then $\nabla_{\theta_S^{(p)}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha}), \ldots, \nabla_{\theta_S^{(P-1)}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha})$ will be set to zero. This operation will force tasks to switch to purely-specific modules and reduce the possibility of the occurrence of the gradient conflict. Moreover, according to (Yosinski et al. 2014), feature representations transit from general to specific in deep neural networks when layer goes deeper. The conflict noticing operation updates modules before the module with the detected gradient conflict to allow those all-shared modules to learn the general feature representations for all tasks and to prevent the early layers trained to be biased towards any one particular task. On the other hand, the conflict noticing operation enforces the corresponding task to learn purely-specific modules in later layers to capture task-specific representations, which could help alleviate the gradient conflict.

Algorithm 1: Conflict-Noticed Architecture Learning

```
Input: Dataset $\mathcal{D}_{tr}$ and $\mathcal{D}_{val}$
Output: Learned architecture parameters $\boldsymbol{\alpha}$
1: Uniformly initialized $\boldsymbol{\alpha}_0$, Pre-trained initialized $\boldsymbol{\theta}_0$;
2: while not converged do
3:     Sample a mini-batch from $\mathcal{D}_{val}$;
4:     Update $\boldsymbol{\alpha}$ by minimizing the upper-level subproblem
       of problem (2);
5:     Sample a mini-batch from $\mathcal{D}_{tr}$;
6:     Compute all task loss functions and the correspond-
       ing gradients according to Eq. (4); ;
7:     Compute conflict-noticed gradient for all-shared
       modules according to Eq. (5);
8:     Update $\boldsymbol{\theta}$ according to by minimizing the lower-level
       subproblem of problem (2);
9: end while
```

Difference with Related Works. While PCGrad and its variants successfully mitigate the gradient conflict, most of them perform gradient manipulation (e.g., gradient projection) on shared parameters in manually designed architectures (e.g., hard-parameter sharing). Those methods aggregate the gradient of different tasks to prevent the shared parameters bias to any one particular task. However, such gradient manipulation operations may lead to underfitting on some less related tasks. Differently, when there is gradient conflict detected, the proposed conflict noticing operation will guide the less related tasks to learn its purely-specific modules instead of all-shared modules so that the task-specific representation for that task can be learned in purely-specific modules, which ensures that each task is fully learned. Moreover, the conflict-noticing operation is built on the proposed search space consisting of purely-specific modules and all-shared modules, as the architecture learning process can switch from all-shared modules to purely-specific modules for each task when the gradient conflicting is detected. Furthermore, instead of manipulating gradients of all the shared parameters as a whole in PCGrad and its variants, the proposed conflict noticing operation operates on fine-grained modules.

In summary, the entire algorithm for the CoNAL method is shown in Algorithm 1.

A Progressive Extension

In the original search space of the proposed CoNAL method, $m + 1$ separate networks in the full size need to be trained, where m is the number of tasks. The entire model size during the architecture learning process grows linearly with the number of tasks, which is computationally demanding when m is large. Another limitation of the CoNAL method is that the largest subgroup among multiple tasks could likely dominate the learning of the shared encoder in the training process. If multiple subgroups exist in the MTL problem, tasks in a smaller subgroup would not be effectively learned and utilized.

To effectively handle the aforementioned issues, we propose a progressive version of the CoNAL method called CoNAL-Pro, which can progressively learn the architecture

Method	Segmentation		Depth		$\Delta_I \uparrow$	Parms. (M)$\downarrow$
	mIoU $\uparrow$	Pix Acc $\uparrow$	Abs Err $\downarrow$	Rel Err$\downarrow$		
STL	68.13	91.28	0.0133	45.039	0	79.27
HPS	-0.73	-0.36	$+0.0009$	$+0.3872$	-2.3334	55.76
GradNorm	-0.75	-0.45	$+0.0008$	-0.5110	-1.6784	55.76
MGDA	-1.09	-0.69	-0.0002	-1.0496	$+0.3140$	55.76
GradVac	$+0.44$	$+0.08$	$+0.0001$	$+1.0179$	-0.6265	55.76
Nash-MTL	$+0.75$	$+0.22$	$+0.0005$	$+1.1392$	-1.2953	55.76
Pareto-MTL	$+0.48$	$+0.04$	$+0.0004$	$+0.7552$	-0.9817	55.76
Cross-stitch	-0.12	$+0.01$	$+0.0002$	-0.6144	-0.1335	79.26
MTAN	$+0.84$	$+0.31$	$+0.0003$	-1.2882	$+0.4866$	72.04
NDDR-CNN	-0.11	-0.03	$+0.0004$	-0.1728	-0.7627	101.58
AFA	$+0.79$	$+0.24$	$+0.0025$	$+2.1253$	-5.5905	87.09
RotoGrad	$+0.04$	$+0.09$	$+0.0002$	-0.0612	-0.3034	57.86
MaxRoam	-1.41	$+0.05$	$+0.0009$	-2.7490	-0.7298	55.76
TSN	$+1.62$	$+0.57$	-0.0005	$+0.8070$	$+1.2442$	50.58
MTL-NAS	-2.78	-1.08	$+0.0011$	$+2.8903$	-5.0490	87.26
BMTAS	$+1.44$	$+0.54$	-0.0011	-0.7830	$+3.1907$	79.04
LTB	$+1.58$	$+0.54$	-0.0008	$+1.6607$	$+1.2708$	70.73
CoNAL	$+1.54$	$+0.51$	-0.0010	-2.3906	**+3.8870**	77.82

Table 2: Performance on the CityScapes validation dataset, where the performance difference between each method and STL is reported. $\uparrow$ ($\downarrow$) indicates the higher (lower) the result, the better the performance. The number of parameters (abbreviated as Parms.) is calculated in MB.

for a large number of tasks with multiple subgroups. Specifically, the CoNAL-Pro method starts the architecture learning process from the last all-share module $p-1$. In the first stage, the search space only contains two branch points: p and $p-1$, for each task. For task t where $1 \leq t \leq m$, it either chooses to use the all-shared module $f_S^{(p-1)}$ or use purely-specific module $h_t^{(p-1)}$. If task t chooses $f_S^{(p-1)}$, it will branch at branch point p and be removed from the search space in the following stages. In this case, only the gradient conflict of the last all-share module needs to be calculated, which highly reduces the memory cost and enables the flexibility to handle a large number of tasks. Tasks branching out at this stage are considered as one subgroup and the number of tasks in the search space reduces after each stage. In the second stage, only tasks that use purely-specific modules are included in the search space, which contains two branch points: $p-1$ and $p-2$. Another shared module $h_S^{(p-1)}$ is added after the branch point $p-1$ for the second subgroup and it is shared by all the tasks in current search space. Figure 1(c) illustrates the second stage of the CoNAL-Pro method. This process repeats until all the tasks have been allocated to a certain branch. In the last stage, the search space will include purely-specific modules after branch point 1 for the remaining tasks, which indicates that similar to the CoNAL method, the CoNAL-Pro method could learn separate networks for those tasks.

Experiments

In this section, we empirically evaluate the proposed CoNAL method. Due to page limit, details on the experimental setup are put in Appendix A.8.

Experiments on Multi-Task CV Benchmarks

To demonstrate the effectiveness of the proposed CoNAL method, we conduct experiments on four CV benchmark datasets: **CityScapes** (Cordts et al. 2016), **NYUv2** (Silberman et al. 2012), **PASCAL-Context** (Mottaghi et al. 2014), and **Taskonomy** (Zamir et al. 2018). The baseline methods in comparison include the Single-Task Learning (**STL**) that trains each task separately, the **HPS** architecture and HPS with **GradNorm** (Chen et al. 2018), **MGDA** (Sener and Koltun 2018), **GradVac** (Wang et al. 2020), **Nash-MTL** (Navon et al. 2022) and **Pareto-MTL** (Lin et al. 2019), manual architecture designed methods including **Cross-stitch** (Misra et al. 2016), **MTAN** (Liu, Johns, and Davison 2019), **NDDR-CNN** (Gao et al. 2019), **AFA** (Cui et al. 2021), and **RotoGrad** (Javaloy and Valera 2022), and architecture learning methods such as **MaxRoam** (Pascal et al. 2021), **TSN** (Sun et al. 2021), **MTL-NAS** (Gao et al. 2020), **BMTAS** (Bruggemann et al. 2020), and **LTB**. For fair comparison, we use the same backbone (with details in Appendix A.8) for all the models in comparison.

In the four datasets, each task has multiple metrics for a thorough evaluation, where definitions of those metrics are put in Appendix A.8. To better show the comparison among all the methods in comparison, we report the overall relative performance of each method over the STL baseline as

$$\Delta_I = 100\% \times \frac{1}{m}\sum_{t=1}^{m}\frac{1}{m_t}\sum_{j=1}^{m_t}\frac{(-1)^{p_{t,j}}(\mathrm{M}_{t,j}-\mathrm{STL}_{t,j})}{\mathrm{STL}_{t,j}},$$

where for a method M, $\mathrm{M}_{t,j}$ denotes its performance in terms of the jth evaluation metric for task t, $\mathrm{STL}_{t,j}$ is defined similarly, $p_{t,j}$ equals 1 if a lower value represents a better performance in terms of the jth metric in task t and 0

Method	Segmentation		Depth		Surface Normal					$\Delta_I \uparrow$	Parms. (M)$\downarrow$
	mIoU $\uparrow$	Pix Acc $\uparrow$	Abs Err $\downarrow$	Rel Err$\downarrow$	Angle Distance		Within $t°\uparrow$				
					Mean $\downarrow$	Median $\downarrow$	11.25	22.5	30		
STL	53.98	75.38	0.3945	0.1631	22.25	15.63	38.12	64.38	74.81	0	118.91
HPS	+0.50	+0.44	−0.0106	−0.0083	+1.25	+1.43	−2.81	−3.28	−2.67	−0.5088	71.89
GradNorm	+0.40	+0.23	−0.0083	−0.0094	+1.22	+1.45	−2.77	−3.35	−2.72	−0.5735	71.89
MGDA	+0.25	+0.25	−0.0082	−0.0040	+1.09	+1.18	−2.31	−2.76	−2.32	−0.8407	71.89
GradVac	+0.11	+0.31	−0.0106	−0.0091	+0.90	+1.05	−2.08	−2.40	−2.02	−0.0320	71.89
Nash-MTL	+0.37	+0.43	−0.0299	−0.0122	+0.25	+0.87	−1.74	−1.73	−1.00	+1.6967	71.89
Pareto-MTL	+0.19	+0.22	−0.0081	−0.0046	+0.96	+1.10	−2.20	−2.59	−2.09	−0.6773	71.89
Cross-stitch	−0.52	+0.11	−0.0141	−0.0076	+0.76	+0.70	−1.11	−1.96	−1.79	+0.1493	118.89
MTAN	+0.76	+0.40	−0.0149	−0.0082	+0.72	+0.67	−1.21	−1.75	−1.49	+0.7658	92.35
NDDR-CNN	−0.14	−0.15	−0.0074	−0.0071	+0.35	+0.44	−0.45	−0.95	−0.89	+0.4106	169.10
AFA	−2.44	−1.47	+0.0085	+0.0061	+1.98	+1.77	−3.05	−4.16	−3.85	-4.7187	136.88
RotoGrad	+0.11	−0.13	−0.0145	−0.0061	+0.80	+0.87	−2.14	−2.51	−1.89	-0.1745	75.03
MaxRoam	+0.55	+0.23	−0.0066	+0.0069	−0.35	−0.65	−1.54	−1.50	−2.58	-0.4803	71.89
TSN	+0.35	+0.61	−0.0172	−0.0068	+1.51	+1.87	−3.68	−4.33	−3.40	-0.9746	50.58
MTL-NAS	−0.06	−0.17	−0.0098	−0.0101	+0.16	+0.38	+0.03	−0.17	−2.23	+0.9666	183.40
BMTAS	−0.14	+0.04	−0.0055	−0.0016	+0.00	−0.11	+0.49	−0.03	−0.12	+0.4793	116.04
LTB	−0.56	+0.05	−0.0040	−0.0095	−0.01	+0.08	−0.37	−0.29	−0.07	+0.8511	86.85
CoNAL	+0.08	+0.27	−0.0095	−0.0069	−0.26	−0.43	+1.20	+0.69	+0.34	**+1.7586**	93.96

Table 3: Performance on the NYUv2 dataset, where the performance difference between each method and STL is reported.

Method	$\Delta_I \uparrow$	Speedup$\uparrow$	Parms. (M)$\downarrow$
CoNAL	+1.76	1.0x	93.96
-w/o conflict-notice	+1.16	1.44x	86.85
-w/o conflict-notice and pure	-0.68	1.46x	71.89
-w/o module splitting	+0.97	1.07x	95.40
-w/o random loss weighting	+1.65	1.00x	93.96

Table 4: Ablation study of the CoNAL method on the NYUv2 dataset. Speedup is computed over the CoNAL during architecture learning.

Method	Train Speedup $\uparrow$	Mean Success Rate $\uparrow$
Single-task policy	1.0x	0.78$\pm$0.042
Multi-task SAC	7.50x	0.44$\pm$0.060
Multi-head SAC	6.98x	0.58$\pm$0.069
Soft Modularization	5.36x	0.68$\pm$0.088
PCGrad	3.09x	0.65$\pm$0.092
CAGrad	3.28x	0.73$\pm$0.068
CoNAL	6.25x	**0.76**$\pm$0.049

Table 5: Comparison on mean success rates for MT10 tasks. Results are calculated for three independent runs.

otherwise, and m_t denotes the number of evaluation metrics in task t.

Tables 2 and 3 as well as Tables 9 and 10 in Appendix A.3 show the performance of all methods in comparison on the four CV benchmark datasets. Compared with the STL counterpart, the proposed CoNAL method improves the performance of all tasks in terms of all the evaluation metrics, while having smaller numbers of parameters in the learned architectures. This indicates that the CoNAL method circumvent negative transfer on those four CV datasets.

Although some manually designed architecture methods (e.g., MTAN) can also circumvent negative transfer on the CityScapes dataset, those methods perform worse than STL in some tasks of the other three larger datasets. For example, on the NYUv2 dataset, existing architecture learning methods (e.g., MTL-NAS) mitigate the negative transfer when compared with HPS but still exhibit inferior performance to the STL method on the surface norm prediction task. Moreover, the proposed CoNAL method achieves the best Δ_I on the four datasets when compared with baseline methods, which demonstrates the effectiveness of the proposed CoNAL method. The proposed CoNAL method learns an architecture with a medium model size and the best performance (in terms of Δ_I) comparing with various MTL methods.

Ablation Study

We provide the ablation study for CoNAL in Table 4. For "w/o conflict-notice", we simply add purely-specific modules into the search space without the gradient noticing operation. For "w/o conflict-notice and pure", we replace purely-specific modules with partially-specific modules in the search space and do not conduct the gradient noticing operation. Compared with the original CoNAL method, the performance of those two variants degrades, which verifies the usefulness of the purely-specific modules and the gradient noticing operation.

For "w/o module splitting", we replace all-shared and purely-specific modules with the entire all-shared and

Method	Parms. (M) ↓	Speedup ↑	Total Test Error ↓
STL	100.56	1.0x	49.59±0.12
HPS	11.21	8.84x	49.78±0.26
UW	11.21	8.34x	49.47±0.86
PCGrad	11.21	2.59x	48.66±0.48
CAGrad	11.21	2.44x	48.75±0.71
TAG-2	22.37	1.29x	49.23±0.05
LTB	77.39	2.10x	49.47±0.45
CoNAL	57.86	2.51x	49.02±0.45
CoNAL-Pro	30.58	3.18x	**48.63**±0.40

Table 6: Results on the CelebA 9-task dataset. Speedup is computed over the STL model during training. The mean and standard deviation of the total test error are calculated over three independent runs.

purely-specific encoders in the search space of CoNAL. This variant has a larger model size and worse performance than the CoNAL method, which verifies the usefulness of fine-grained modules. For "w/o random loss weighting", we replace randomly sampled task loss weights with the same fixed weights as in LTB. This variant has slightly worse performance than the CoNAL method, which verifies that the random loss weighting can improve the performance a bit, and reduce the tedious cost to tune loss weights.

Experiments on Multi-Task RL

To further examine the proposed CoNAL method for RL tasks, we evaluate it on the MT10 challenge from the Meta-World environment (Yu et al. 2020b). By following (Yang et al. 2020; Sodhani, Zhang, and Pineau 2021), we train the policy with Soft Actor-Critic (SAC) (Haarnoja et al. 2018) and compare with multi-task SAC (i.e., SAC with a shared model), multi-head SAC (i.e., SAC with a shared policy network and task-specific head), Soft Modularization (Yang et al. 2020), PCGrad (Yu et al. 2020a) and CAGrad (Liu et al. 2021a).

According to the results shown in Table 5, the proposed CoNAL method outperforms most baseline methods and the significant t-test with 95% confidence shows that the CoNAL method performs better than PCGrad, CAGrad and Soft Modularization. Though the single-task policy performs slightly better on one task (i.e., the pick-place task), the CoNAL method achieves comparable performance with the single-task policy in terms of the mean success rate and has a much faster training speed.

Experiments for the CoNAL-Pro Method

To evaluate the proposed CoNAL-Pro method, we conduct experiments on the CelebA dataset (Liu et al. 2015). We follow (Fifty et al. 2021) to filter the 40 tasks down to 9 tasks, leading to CelebA 9-task dataset, which has a larger number of tasks with multiple task subgroups.

On the CelebA 9-task dataset, we compare the proposed CoNAL and CoNAL-Pro methods with the HPS method, the **TAG-2** method (Fifty et al. 2021) that performs the two-split task grouping, LTB, PCGrad, and CAGrad. According to experimental results shown in Table 6, the CoNAL-Pro methods has the lowest total test error, which demonstrates the effectiveness of the proposed method. The model architecture learned by the CoNAL-Pro method has fewer parameters but better performance than the LTB method as it learns more subgroups. This demonstrates that the CoNAL-Pro method can obtain an architecture that can learn task grouping for better knowledge sharing among tasks.

Method	Speedup ↑	Δ_I ↑
STL	1.0x	0
HPS	1.69x	-0.5088
HPS-PCGrad	0.78x	-0.1205
HPS-CAGrad	0.61x	-0.1817
CoNAL	1.18x	+1.7586
CoNAL-PCGrad	0.42x	**+2.5271**
CoNAL-CAGrad	0.38x	+1.8932

Table 7: Combination with gradient manipulation methods on the NYUv2 dataset.

Combination and Comparison with Gradient Manipulation Methods

The retraining process of the CoNAL method can straightforwardly incorporate various gradient manipulation methods based on Eq. (3). To demonstrate that the performance of the CoNAL method can be improved even further, we combine the proposed CoNAL model with the PCGrad and CAGrad methods, which manipulate task gradients to alleviate the gradient conflict. According to experimental results shown in Table 7, combining with PCGrad and CAGrad can further improve the performance of the proposed CoNAL model. Compared with the HPS-PCGrad method that improves about 0.4 over HPS in terms of Δ_I, CoNAL-PCGrad improves about 0.8 over CoNAL, which indicates that the architecture learned by the CoNAL method is more preferred than HPS while combining with the PCGrad method. Moreover, as the PCGrad and CAGrad methods are built on the HPS architecture, we can see that the CoNAL method performs better than the PCGrad and CAGrad methods, which are just HPS-PCGrad and HPS-CAGrad in Table 7, and this result indicates that learning a suitable architecture could be more important to the performance improvement.

Conclusion

In this paper, we propose the CoNAL method to learn multi-task network architectures to alleviate the gradient conflict issue. We first introduce purely-specific modules to the design of the search space and propose a conflict-noticed algorithm to mitigate gradient conflict. We further propose an extension of the CoNAL method to enable the learning on many tasks and the identification of multiple subgroups. We validate the CoNAL method on multiple MTL benchmarks across three challenging domains. For future work, we are interested in extending the CoNAL method to other multi-task learning problems.

Acknowledgments

This work is supported by NSFC key grant 62136005, NSFC general grant 62076118, and Shenzhen fundamental research program JCYJ20210324105000003.

References

Bruggemann, D.; Kanakis, M.; Georgoulis, S.; and Gool, L. V. 2020. Automated Search for Resource-Efficient Branched Multi-Task Networks. In *The British Machine Vision Conference 2020*.

Brüggemann, D.; Kanakis, M.; Obukhov, A.; Georgoulis, S.; and Van Gool, L. 2021. Exploring relational context for multi-task dense prediction. In *CVPR 2021*.

Caruana, R. 1997. Multitask Learning. *Machine Learning*.

Chen, L.-C.; Papandreou, G.; Kokkinos, I.; Murphy, K.; and Yuille, A. L. 2017a. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4).

Chen, L.-C.; Papandreou, G.; Schroff, F.; and Adam, H. 2017b. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*.

Chen, Z.; Badrinarayanan, V.; Lee, C.-Y.; and Rabinovich, A. 2018. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *ICML 2018*.

Chen, Z.; Ngiam, J.; Huang, Y.; Luong, T.; Kretzschmar, H.; Chai, Y.; and Anguelov, D. 2020. Just pick a sign: Optimizing deep multitask models with gradient sign dropout. *Advances in Neural Information Processing Systems*.

Cordts, M.; Omran, M.; Ramos, S.; Rehfeld, T.; Enzweiler, M.; Benenson, R.; Franke, U.; Roth, S.; and Schiele, B. 2016. The cityscapes dataset for semantic urban scene understanding. In *CVPR 2016*.

Cui, C.; Shen, Z.; Huang, J.; Chen, M.; Xu, M.; Wang, M.; and Yin, Y. 2021. Adaptive Feature Aggregation in Deep Multi-task Convolutional Neural Networks. *IEEE Transactions on Circuits and Systems for Video Technology*.

Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; and Fei-Fei, L. 2009. Imagenet: A large-scale hierarchical image database. In *CVPR 2009*.

Dery, L. M.; Dauphin, Y. N.; and Grangier, D. 2021. Auxiliary Task Update Decomposition: the Good, the Bad and the neutral. In *ICLR 2021*.

Du, Y.; Czarnecki, W. M.; Jayakumar, S. M.; Farajtabar, M.; Pascanu, R.; and Lakshminarayanan, B. 2018. Adapting auxiliary losses using gradient similarity. *arXiv preprint arXiv:1812.02224*.

Fifty, C.; Amid, E.; Zhao, Z.; Yu, T.; Anil, R.; and Finn, C. 2021. Efficiently identifying task groupings for multi-task learning. *Advances in Neural Information Processing Systems*, 34.

Franceschi, L.; Frasconi, P.; Salzo, S.; Grazzi, R.; and Pontil, M. 2018. Bilevel programming for hyperparameter optimization and meta-learning. In *ICML 2018*.

Gao, Y.; Bai, H.; Jie, Z.; Ma, J.; Jia, K.; and Liu, W. 2020. MTL-NAS: Task-Agnostic Neural Architecture Search Towards General-Purpose Multi-Task Learning. In *CVPR 2020*.

Gao, Y.; Ma, J.; Zhao, M.; Liu, W.; and Yuille, A. L. 2019. Nddr-cnn: Layerwise feature fusing in multi-task cnns by neural discriminative dimensionality reduction. In *CVPR 2019*.

Guo, P.; Lee, C.; and Ulbricht, D. 2020. Learning to Branch for Multi-Task Learning. In *ICML 2020*.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML 2018*.

Hu, J.; Ruder, S.; Siddhant, A.; Neubig, G.; Firat, O.; and Johnson, M. 2020. Xtreme: A massively multilingual multi-task benchmark for evaluating cross-lingual generalisation. In *ICML 2020*.

Javaloy, A.; and Valera, I. 2022. RotoGrad: Gradient Homogenization in Multitask Learning. In *ICLR 2022*.

Kendall, A.; Gal, Y.; and Cipolla, R. 2018. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *CVPR 2018*.

Lee, G.; Yang, E.; and Hwang, S. 2016. Asymmetric multi-task learning based on task relatedness and loss. In *ICML*.

Liang, J. Z.; Meyerson, E.; and Miikkulainen, R. 2018. Evolutionary architecture search for deep multitask networks. In *Proceedings of the Genetic and Evolutionary Computation Conference, 2018*.

Lin, B.; YE, F.; Zhang, Y.; and Tsang, I. 2022. Reasonable Effectiveness of Random Weighting: A Litmus Test for Multi-Task Learning. *Transactions on Machine Learning Research*.

Lin, B.; and Zhang, Y. 2022. LibMTL: A Python Library for Multi-Task Learning. *arXiv preprint arXiv:2203.14338*.

Lin, X.; Zhen, H.-L.; Li, Z.; Zhang, Q.-F.; and Kwong, S. 2019. Pareto multi-task learning. *Advances in neural information processing systems*, 32.

Liu, B.; Liu, X.; Jin, X.; Stone, P.; and Liu, Q. 2021a. Conflict-Averse Gradient Descent for Multi-task Learning. *Advances in Neural Information Processing Systems*, 34.

Liu, H.; Simonyan, K.; and Yang, Y. 2019. DARTS: Differentiable Architecture Search. In *ICLR 2019*.

Liu, L.; Li, Y.; Kuang, Z.; Xue, J.; Chen, Y.; Yang, W.; Liao, Q.; and Zhang, W. 2021b. Towards impartial multi-task learning. In *ICLR 2021*.

Liu, S.; Johns, E.; and Davison, A. J. 2019. End-to-end multi-task learning with attention. In *CVPR 2019*.

Liu, Z.; Luo, P.; Wang, X.; and Tang, X. 2015. Deep learning face attributes in the wild. In *Proceedings of the IEEE international conference on computer vision*.

Loshchilov, I.; and Hutter, F. 2018. Decoupled Weight Decay Regularization. In *ICLR*.

Lu, Y.; Kumar, A.; Zhai, S.; Cheng, Y.; Javidi, T.; and Feris, R. S. 2017. Fully-Adaptive Feature Sharing in Multi-Task Networks with Applications in Person Attribute Classification. In *CVPR 2017*.

Maninis, K.; Radosavovic, I.; and Kokkinos, I. 2019. Attentive Single-Tasking of Multiple Tasks. In *CVPR 2019*.

Misra, I.; Shrivastava, A.; Gupta, A.; and Hebert, M. 2016. Cross-stitch networks for multi-task learning. In *CVPR 2016*.

Mottaghi, R.; Chen, X.; Liu, X.; Cho, N.; Lee, S.; Fidler, S.; Urtasun, R.; and Yuille, A. L. 2014. The Role of Context for Object Detection and Semantic Segmentation in the Wild. In *CVPR 2014*.

Navon, A.; Shamsian, A.; Achituve, I.; Maron, H.; Kawaguchi, K.; Chechik, G.; and Fetaya, E. 2022. Multi-Task Learning as a Bargaining Game. In *ICML 2022*.

Pascal, L.; Michiardi, P.; Bost, X.; Huet, B.; and Zuluaga, M. 2021. Maximum Roaming Multi-Task Learning. In *AAAI 2021*.

Pham, H.; Guan, M.; Zoph, B.; Le, Q.; and Dean, J. 2018. Efficient neural architecture search via parameters sharing. In *ICML 2018*.

Rosenbaum, C.; Klinger, T.; and Riemer, M. 2018. Routing Networks: Adaptive Selection of Non-Linear Functions for Multi-Task Learning. In *ICLR 2018*.

Ruder, S. 2017. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*.

Sener, O.; and Koltun, V. 2018. Multi-task learning as multi-objective optimization. *Advances in neural information processing systems*, 31.

Silberman, N.; Hoiem, D.; Kohli, P.; and Fergus, R. 2012. Indoor segmentation and support inference from rgbd images. In *European conference on computer vision*.

Sodhani, S.; Zhang, A.; and Pineau, J. 2021. Multi-task reinforcement learning with context-based representations. In *ICML 2021*.

Søgaard, A.; and Goldberg, Y. 2016. Deep multi-task learning with low level tasks supervised at lower layers. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*.

Standley, T.; Zamir, A.; Chen, D.; Guibas, L.; Malik, J.; and Savarese, S. 2020. Which tasks should be learned together in multi-task learning? In *ICML 2020*.

Strezoski, G.; Noord, N. v.; and Worring, M. 2019. Many task learning with task routing. In *CVPR 2019*.

Sun, G.; Probst, T.; Paudel, D. P.; Popović, N.; Kanakis, M.; Patel, J.; Dai, D.; and Van Gool, L. 2021. Task Switching Network for Multi-Task Learning. In *CVPR 2021*.

Sun, X.; Panda, R.; Feris, R.; and Saenko, K. 2020. AdaShare: Learning What To Share For Efficient Deep Multi-Task Learning. In *Proceedings of the 33rd Advances in Neural Information Processing Systems*.

Tang, H.; Liu, J.; Zhao, M.; and Gong, X. 2020. Progressive layered extraction (ple): A novel multi-task learning (mtl) model for personalized recommendations. In *Fourteenth ACM Conference on Recommender Systems*.

Vandenhende, S.; Georgoulis, S.; Gool, L. V.; and Brabandere, B. D. 2020. Branched Multi-Task Networks: Deciding what layers to share. In *The British Machine Vision Conference 2020*.

Vandenhende, S.; Georgoulis, S.; Van Gansbeke, W.; Proesmans, M.; Dai, D.; and Van Gool, L. 2021. Multi-task learning for dense prediction tasks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Wang, Z.; Tsvetkov, Y.; Firat, O.; and Cao, Y. 2020. Gradient Vaccine: Investigating and Improving Multi-task Optimization in Massively Multilingual Models. In *ICLR*.

Wolf, T.; Chaumond, J.; Debut, L.; Sanh, V.; Delangue, C.; Moi, A.; Cistac, P.; Funtowicz, M.; Davison, J.; Shleifer, S.; et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*.

Yang, R.; Xu, H.; Wu, Y.; and Wang, X. 2020. Multi-task reinforcement learning with soft modularization. *Advances in Neural Information Processing Systems*, 33.

Ye, F.; Lin, B.; Yue, Z.; Guo, P.; Xiao, Q.; and Zhang, Y. 2021. Multi-objective meta learning. *Advances in Neural Information Processing Systems*, 34: 21338–21351.

Yosinski, J.; Clune, J.; Bengio, Y.; and Lipson, H. 2014. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014*.

Yu, T.; Kumar, S.; Gupta, A.; Levine, S.; Hausman, K.; and Finn, C. 2020a. Gradient Surgery for Multi-Task Learning. *Advances in Neural Information Processing Systems*, 33.

Yu, T.; Quillen, D.; He, Z.; Julian, R.; Hausman, K.; Finn, C.; and Levine, S. 2020b. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning*.

Yue, Z.; Guo, P.; Zhang, Y.; and Liang, C. 2022a. Learning Feature Alignment Architecture for Domain Adaptation. In *2022 International Joint Conference on Neural Networks*.

Yue, Z.; Lin, B.; Zhang, Y.; and Liang, C. 2022b. Effective, Efficient and Robust Neural Architecture Search. In *2022 International Joint Conference on Neural Networks*.

Zamir, A. R.; Sax, A.; Shen, W.; Guibas, L. J.; Malik, J.; and Savarese, S. 2018. Taskonomy: Disentangling task transfer learning. In *CVPR 2018*.

Zhang, Y.; and Yang, Q. 2021. A Survey on Multi-Task Learning. *IEEE Transactions on Knowledge and Data Engineering*.

Zhao, J.; Lv, W.; Du, B.; Ye, J.; Sun, L.; and Xiong, G. 2021. Deep multi-task learning with flexible and compact architecture search. *International Journal of Data Science and Analytics*.