A Tree-Structured Multi-Task Model Recommender

Anonymous[1]

[1]Anonymous Institution

Abstract Tree-structured multi-task architectures have been employed to jointly tackle multiple vision tasks in the context of multi-task learning (MTL). The major challenge is to determine where to branch out for each task given a backbone model to optimize for both task accuracy and computation efficiency. To address the challenge, this paper proposes a recommender that, given a set of tasks and a convolutional neural network-based backbone model, automatically suggests tree-structured multi-task architectures that could achieve a high task performance while meeting a user-specified computation budget without performing model training. Extensive evaluations on popular MTL benchmarks show that the recommended architectures could achieve competitive task accuracy and computation efficiency compared with state-of-the-art MTL methods.

1 Introduction

Multi-task learning (MTL) aims to solve multiple tasks simultaneously. Compared to independently learning tasks, it is an effective approach to improve task performance while reducing computation and storage costs. However, over-sharing information between tasks can cause task interference (Sener and Koltun, 2018; Maninis et al., 2019) and accuracy degradation. The major challenge in designing a multi-task architecture is thus to identify an intermediate state between over-shared and independent architectures (i.e., a partially-shared architecture), which not only preserves the benefits of lower computation cost and memory overhead, but also avoid task interference as much as possible to guarantee acceptable task accuracy. Such a partially-shared architecture is also called *a tree-structured multi-task architecture*. Its shallow network layers are shared across tasks like tree roots, whereas deeper ones gradually grow more task-specific like tree branches (Vandenhende et al., 2019). Identifying the best tree-structured multi-task architecture needs to determine where to branch out for each task to optimize for both computation efficiency and task accuracy.

Previous works opted for the simplest strategy of sharing the initial layers of a backbone model, after which all tasks branch out simultaneously (Ruder, 2017; Nekrasov et al., 2019; Suteu and Guo, 2019; Leang et al., 2020). Since the point at which the branching occurs is determined manually, they call for domain expertise when tackling different tasks and usually result in unsatisfactory solutions due to the enormous architecture design space. To automate architecture design, one line of work deduced the layer sharing possibility based on measurable task relatedness (Lu et al., 2017; Vandenhende et al., 2019; Standley et al., 2020) and minimized the total task dissimilarity when designing multi-task architectures. However, they ignore task interactions that could bring the potential generalization improvement and positive inhibition of overfitting when multiple tasks are trained together (Ruder, 2017; Vandenhende et al., 2020). Another line of work attempted to learn how to branch a network such that the overall multi-task loss is minimized via differentiable neural architecture search (Bruggemann et al., 2020; Guo et al., 2020). Such end-to-end frameworks integrated the architecture search with the network training process, which easily leads to sub-optimal multi-task architectures (Choromanska et al., 2015; Sun et al., 2020) due to training difficulties. Besides, the learned multi-task architectures cannot guarantee to meet a user-defined computation budget since these methods are like a black box where users cannot control the exploring process.

In this paper, we overcome the aforementioned limitations and propose a tree-structured multi-task model recommender. It takes as inputs an arbitrary convolutional neural network (CNN)

Submitted to AutoML Conference 2022

backbone model and a set of tasks in interest, and then predicts the top-k tree-structured multi-task architectures that achieve high task accuracy while meeting a user-specified computation budget. Our basic idea is to build a *task accuracy estimator* that can predict the task accuracy of each multi-task model architecture in the design space without performing model training. The task accuracy estimator captures task interactions by leveraging the task performance of well-trained two-task architectures instead and enables ranking of all multi-task architectures with more than two tasks using their predicted task accuracy. The recommender can then *enumerate the design space* and identify the multi-task models with the highest predicted task accuracy. Unlike differentiable neural architecture search-based approaches, the recommender is a white-box that allows users to easily control the computation complexity of the multi-task architectures. The basic idea poses three major research questions:

- **RQ 1**: how to build an *accurate* task accuracy estimator that enables a faithful ranking of the multi-task architectures in the design space based on their estimated task performance?
- **RQ 2**: how to *represent* multi-task model architectures such that a recommender can *completely* enumerate the design space for estimating their performance?
- **RQ 3**: how to *automatically* support various CNN backbone models?

To answer **RQ 1**, our task accuracy estimator predicts the task accuracy of a multi-task architecture by averaging the task accuracy of associated well-trained two-task architectures. A ranking score of the multi-task architecture is calculated as the weighted sum of the tasks' accuracy, where the weight of each task is determined by quantified accuracy variance to ensure faithful ranking. To answer **RQ 2**, we propose a novel data structure called *Layout* to represent a multi-task architecture and an operation called *Layout Cut* to derive multi-task architectures. We further propose a *cut-based recursive algorithm* that is proved to be able to enumerate the design space completely. To answer **RQ 3**, we design a *branching point detector* to automatically separate a CNN backbone model into a sequence of computation blocks where each block corresponds to a possible branching point.[1] The detector saves manual efforts in applying the recommender to an arbitrary CNN architecture.

Experiments on popular MTL benchmarks, NYUv2 (Silberman et al., 2012) and Tiny-Taskonomy (Zamir et al., 2018), using different backbone models, Deeplab-ResNet34 (Chen et al., 2017) and MobileNetV2 (Sandler et al., 2018), demonstrate that the recommended tree-structured multi-task architectures achieve competitive task accuracy compared with state-of-the-art MTL methods under specified computation budgets. Our empirical evaluation also demonstrates that ranking of the multi-task architectures using estimated task accuracy without training has a high correlation (Pearson's γ is $0.5 \sim 0.85$) with the oracle ranking after training for different CNN architectures.

2 Related Works

Multi-task learning (MTL) is commonly categorized into either hard or soft parameter sharing (Ruder, 2017; Vandenhende et al., 2020). In hard parameter sharing, a set of parameters in the backbone model are shared among tasks. In soft parameter sharing (Misra et al., 2016; Ruder et al., 2019; Gao et al., 2019), each task has its own set of parameters. Task information is shared by applying regularization on parameters during training, such as enforcing the weights of the model for each task to be similar. In this paper, we focus on hard parameter sharing as it produces memory- and computation-efficient multi-task models.

Early works on multi-task architecture design rely on domain expertise to decide which layers should be shared across tasks and which ones should be task-specific (Long et al., 2017; Nekrasov

[1] A branching point usually corresponds to a micro-architecture such as a residual block in ResNet50, following prior works (Vandenhende et al., 2019; Guo et al., 2020; Bruggemann et al., 2020).

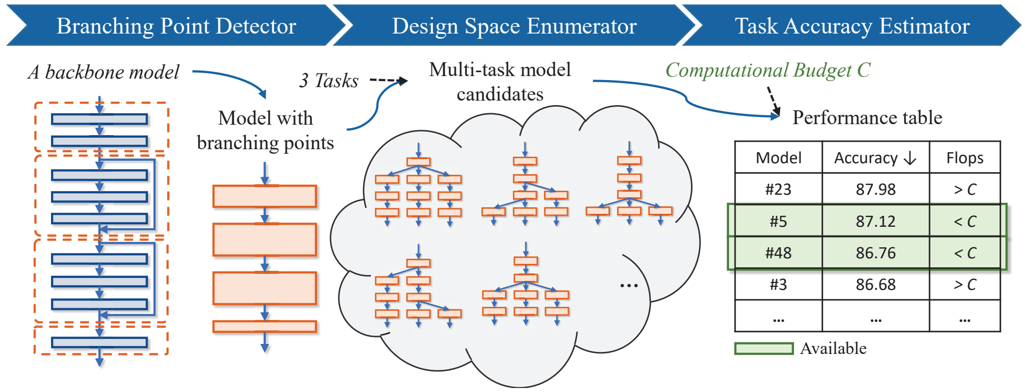


Figure 1: Tree-structured multi-task model recommender workflow.

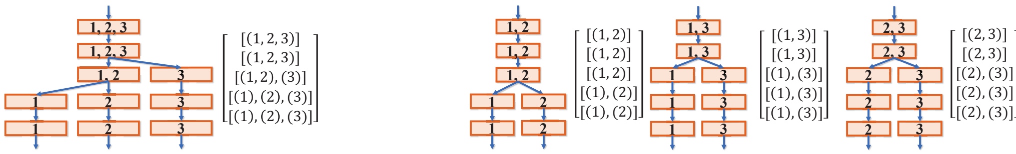
et al., 2019; Suteu and Guo, 2019; Leang et al., 2020). Due to the enormous design space, such approaches are difficult to find an optimal solution.

In recent years, researchers attempt to automate the procedure of designing multi-task architectures. Deep Elastic Network (DEN) (Ahn et al., 2019) uses reinforcement learning (RL) to determine whether each filter in convolutional layers can be shared across tasks. Similarly, AdaShare (Sun et al., 2019) and AutoMTL (Zhang et al., 2021) learn task-specific policies that select which layers to execute for a given task. Some other works (Gao et al., 2020; Wu et al., 2021) adopt NAS techniques to explore feature fusion opportunities across tasks. Their primary goal is to improve task accuracy instead of computation efficiency by minimizing the overall multi-task loss. Thus there is no guarantee that the searched multi-task model architectures will meet the computation budget. Also, their architecture search procedure requires substantial search time and is usually hard to converge since the sharing strategy and network parameters generally prefer the alternating training principle to stabilize the training process (Xie et al., 2018; Sun et al., 2019; Wu et al., 2019).

Our work pays more attention to balancing task accuracy and computation efficiency through recommending branching structures for multi-task models. There also exist several interesting methods under this direction. FAFS (Lu et al., 2017) starts from a thin network where tasks initially share all layers and dynamically grows the model in a greedy layer-by-layer fashion depending on task similarities. It computes task similarity based on the likelihood of input samples having the same difficulty level. What-to-Share (Vandenhende et al., 2019) measures the task affinity by analyzing the representation similarity between independent models for each task. It recommends the multi-task architecture with the minimum total task dissimilarity. However, because the task dissimilarity between two tasks is always non-negative, the theoretical optimal multi-task architecture would be always independent models whose total task dissimilarity is zero. In contrast to pre-computing the task relatedness, BMTAS (Bruggemann et al., 2020) and Learn-to-Branch (Guo et al., 2020) utilize differentiable neural architecture search to construct end-to-end trainable frameworks that integrate the architecture exploration with the network training process. These learning-based methods easily lead to the suboptimal multi-task model (Choromanska et al., 2015; Sun et al., 2020) due to difficulties in training and cannot guarantee the resulting multi-task architecture to obey a user-defined computation budget.

3 Proposed Approach

Given a backbone model with B branching point and a set of T tasks, our goal is to build a recommender that, when deployed, predicts k tree-structured multi-task architectures that achieve a high task accuracy while meeting a user-specified computation budget C. Figure 1 illustrates the offline building process and the online usage of the recommender. During the offline building process, users provide an arbitrary CNN-based backbone model and a set of tasks. A *branching point*



(a) A multi-task model for three tasks. (b) Associated two-task models.

Figure 2: A multi-task architecture and the related two-task architectures. The average of task accuracy in (b) is a good indicator of the task accuracy in (a).

detector will automatically identify the sequential computational blocks in the backbone model, each of the blocks corresponding to a viable branching point. A *task accuracy estimator* is then built based on the given set of tasks and the identified branching points to predict the performance of all the tree-structured multi-task architectures in the design space, which are explored using a *design space enumerator.* The performance of these multi-task architectures including their other attributes such as model size, FLOPs etc. could be stored in a performance table to facilitate online queries. When deployed, the recommender takes a user-specified computation budget C as input, and suggests multi-task architectures by looking up the performance table on the fly. We next elaborate the three major components, *task accuracy estimator*, *design space enumerator*, and *branching point detector* in detail.

3.1 Task Accuracy Estimator

Task accuracy estimator predicts the task accuracy of a tree-structured multi-task architecture without performing actual model training. The problem is challenging because predicting a single-task architecture's accuracy is already non-trivial and multi-task architectures introduce more complexities due to task interactions and interference. Task accuracy estimator addresses the problem by leveraging well-trained two-task architectures to quantify task accuracy and interactions and predict the performance of a multi-task architecture. Specifically, for any tree-structured multi-task architecture, the estimator predicts its task accuracy by averaging the task accuracy of all the *associated* two-task architectures. A two-task architecture is considered associated if it meets both conditions: (1) the two tasks are a subset of the tasks in the multi-task architecture; (2) the two tasks have the identical branching point as in the multi-task architecture. The algorithm pseudocode for identifying associated two-task architectures is in the Appendix Section A.

Figure 2 illustrate the basic idea. Figure 2(a) shows a multi-task architecture constructed from a backbone model with five branching points for three tasks and Figure 2(b) shows the three associated two-task architectures. The numbers inside each block indicate among which tasks the block is shared. Tasks 1 and 2 branch out after the third block, which is the same branching point as the first two-task architecture in Figure 2(b). Similarly, tasks 1 and 3 branch out after the second block, which is the same branching point as the second two-task architecture in Figure 2(b). We estimate task 1 accuracy of the multi-task architecture by averaging task 1 accuracy of the first and second two-task architectures, task 2 accuracy from those of the first and third two-task models, and task 3 accuracy from those of the second and third two-task models.

The ultimate goal of the task accuracy estimator is to enable ranking of multi-task architectures based on their estimated task accuracy. Due to the noise in training two-task architectures (Pham et al., 2020), an estimated task accuracy of a multi-task architecture could suffer from some accuracy variance and lead to an inaccurate ranking. A *ranking score* is thus calculated as the *weighted* sum of the tasks' performance. Tasks with higher accuracy variance have lower *task weight.*

To quantify accuracy variance and task weight, we adopt the Singular Value Decomposition Entropy (SVDE) (Li et al., 2008; Jelinek et al., 2019) to measure the regularity of each task t_i's performance in its $B+1$ two-task architectures with another task t_j. SVDE reflects the number of orthogonal vectors contributed to a task performance sequence $(\Delta t_i^{(0)}, \ldots, \Delta t_i^{(B)} | t_i, t_j)$, where $\Delta t_i^{(b)}$

is t_i's performance in a two-task model that branches at b-th point. Higher entropy indicates lower regularity and thus higher variance. The *task weight* of t_i is the average of the negative entropy over all possible two task combinations $(t_i, t_j), \forall j \neq i$:

$$w_i = \frac{1}{T-1} \sum_{j \in \mathcal{T}, j \neq i} -SVDE(\Delta t_i^{(0)}, \ldots, \Delta t_i^{(B)} | t_i, t_j), \tag{1}$$

where $\mathcal{T}$ is the set of tasks and $T = |\mathcal{T}|$ is the number of tasks. The final ranking score of a multi-task architecture is:

$$S = normalize([w_1, \ldots, w_T])^T [\Delta t_1, \ldots, \Delta t_T], \tag{2}$$

where $normalize([w_1, \ldots, w_T])$ is the normalized task weights so that their sum is equal to one.

Estimating the accuracy of a multi-task architecture requires training of its associated two-task architectures. Given T number of tasks and B number of branching points, the total number of two-task architectures to train is $C_T^2 \cdot (B+1)$, where C_T^2 is the number of two task combinations. The training overhead of the two-task architectures is much less than training all the multi-task architectures whose number is $O([(2^{T-1}-1) \cdot B]^{T-1})$. Our experiments in Section 4.3 demonstrate that the ranking of multi-task architectures using estimated task accuracy without training has a high correlation (Pearson's γ is $0.5 \sim 0.85$) with the oracle ranking from training for different CNN architectures.

3.2 Design Space Enumerator

Design space enumerator formalizes the representation of tree-structured multi-task architectures so that the recommender can completely enumerate the design space. It introduces a data structure called *Layout* and an operator called *Layout Cut* to derive multi-task architectures. Based on the two core concepts, we propose a cut-based recursive algorithm to enumerate all possible architectures.

Definition 3.1 (Layout). *A layout is a symbolized representation of a tree-structured multi-task architecture. Formally, for T tasks and a backbone model with B branching points, a layout $\mathbf{L} = [L_1, L_2, \cdots, L_B]$, where L_i is a list of task sets at the i-th branching point. Task sets in $L_i = [L_i^1, L_i^2, \cdots]$ are subsets of tasks $\mathcal{T}$ and satisfy two conditions: (1) $L_i^1 \cup L_i^2 \cup \cdots = \mathcal{T}$, and (2) $L_i^p \cap L_i^q = \emptyset, \forall L_i^{\{p,q\}} \in L_i$.*

A task set L_i^p means the set of tasks in L_i^p sharing the i-th block. Figure 2 illustrates the layouts of a multi-task architecture and three two-task architectures. We define the initial layout as $\mathbf{L}_0 = [\underbrace{[\mathcal{T}], \cdots, [\mathcal{T}]}_{B}] = [\underbrace{[\{t_1, \ldots, t_T\}], \cdots, [\{t_1, \ldots, t_T\}]}_{B}]$, which means all the tasks share all the blocks in the multi-task model.

Definition 3.2 (Layout Cut). *A layout cut is an operator that transforms one layout to another layout by selecting an available task set and dividing it into two task sets. (The complete definition can be found in the Appendix Section C)*

Based on Definition 3.2, we propose a cut-based algorithm to enumerate all possible layouts, namely tree-structured multi-task architectures. The main idea is to recursively apply layout cuts on the initial layout $\mathbf{L}_0$ and all the generated layouts until no new layout is generated. The pseudocode of the layouts enumerator is included in the Appendix Section D.

Theorem 3.1. *The cut-based layout enumeration algorithm could explore the design space of tree-structured multi-task models completely.*

This completeness theorem is proved by induction as demonstrated in the Appendix Section E.

3.3 Branching Point Detector

Branching point detector allows the recommender to support an arbitrary CNN backbone model without manual reimplementation. Its design is motivated by the observation that common CNN backbone models are a sequence of *computation blocks*, such as residual blocks in ResNet50 (He et al., 2016) and bottleneck blocks in MobileNetV2 (Sandler et al., 2018). These computation blocks are typically treated as branching points in MTL (Vandenhende et al., 2019; Bruggemann et al., 2020) and satisfy two requirements. (1) They contain trainable parameters so that whether they are shared across tasks is likely to make a difference in task accuracy. (2) They are connected to each other sequentially–that is, there is no link across non-sequential blocks. The two requirements inspire us to design a two-stage branching point detector.

The first stage is to identify groups of operators called *candidate blocks* in a given backbone model. Each candidate block is a subgraph in the computation graph of the backbone model that takes only one input tensor and produces only one output tensor. The branching point detector leverages the Cut Theorem[2] in the Graph Theory to partition the original computation graph of the backbone model into candidate blocks. A subgraph can be divided into two subgraphs if the size of the minimum cut is one; otherwise, the subgraph can be no longer partitioned and is a candidate block. Because a candidate block could contain operators that have no parameters at all, the second stage is to merge candidate blocks that contain only unparameterized layers (e.g., ReLU, Pooling) and normalization layers (e.g., Batch Normalization) with adjacent candidate blocks (e.g., Convolution Layer) to generate final computation blocks (e.g., ConvBNReLU). Each computation block corresponds to a viable branching point. The detailed algorithm and pseudocode are in the Appendix Section F.

The proposed branching point detector enables the recommender to automatically parse the backbone model and produce two-task architectures and multi-task architectures based on a layout. It saves manual efforts in generalizing multi-task architecture search across different backbone models. We also allow users to flexibly add or remove branching points to adjust the architecture search space.

4 Experiments

4.1 Experiment Settings

Datasets and Tasks. Our experiments are conducted on two popular datasets in multi-task learning, **NYUv2** (Silberman et al., 2012) and **Tiny-Taskonomy** (Zamir et al., 2018). The NYUv2 dataset consists of RGB-D indoor scene images and three tasks, 13-class semantic segmentation, depth estimation, and surface normal prediction. Tiny-Taskonomy contains indoor images and five tasks: semantic segmentation, surface normal prediction, depth estimation, keypoint detection, and edge detection. The data splits follow prior works (Sun et al., 2019; Zhang et al., 2021).

Loss Functions and Evaluation Metrics. In NYUv2, Semantic segmentation uses a pixel-wise cross-entropy loss for each predicted class label, and is evaluated using mean Intersection over Union and Pixel Accuracy (mIoU and Pixel Acc, the higher the better). Surface normal prediction uses the inverse of cosine similarity between the normalized prediction and ground truth, and is evaluated using mean and median angle distances between the prediction and the ground truth (the lower the better), and the percentage of pixels whose prediction is within the angles of 11.25°, 22.5° and 30° to the ground truth (Eigen and Fergus, 2015) (the higher the better). Depth estimation uses the L1 loss, and the absolute and relative errors between the prediction and the ground truth are computed (the lower the better). In Taskonomy, all the tasks are trained using the same loss as in NYUv2 and directly evaluated by the task-specific loss. Since tasks have multiple evaluation metrics and their value can also be at different scales, we compute a single **relative performance**

[2] `https://en.wikipedia.org/wiki/Minimum_cut`

Table 1: Performance of top-5 recommended architectures on NYUv2 using Deeplab-ResNet34.

Model	FLOPs (%) ↓	#Params (%) ↓	Semantic Seg.			Surface Normal Prediction						Depth Estimation						Δt ↑
			mIoU ↑	Pixel Acc. ↑	Δt_1 ↑	Error ↓		θ, within ↑			Δt_2 ↑	Error ↓		δ, within ↑			Δt_3 ↑	
						Mean	Median	11.25°	22.5°	30°		Abs.	Rel.	1.25	1.25^2	1.25^3		
Ind. Models	-	-	**26.50**	**58.20**	-	17.70	16.30	29.40	72.30	**87.30**	-	0.62	0.24	57.80	85.80	96.00	-	-
#0	-66.67	-66.66	25.23	57.69	-2.8	**17.14**	15.15	35.85	72.20	85.54	**6.0**	**0.55**	0.23	63.85	89.38	97.03	5.8	**3.0**
#45	-36.56	-35.45	25.18	57.36	-3.2	17.26	**14.93**	**36.33**	72.27	85.16	6.4	0.58	**0.22**	62.70	88.79	96.93	5.5	2.9
#50	-46.87	-46.13	24.72	56.71	-4.6	17.24	15.13	32.17	**72.66**	85.75	3.6	0.56	0.23	63.87	88.72	96.81	5.6	1.5
#37	-34.87	-33.70	26.30	57.94	**-0.6**	17.24	15.16	35.78	71.90	85.43	5.7	0.61	**0.22**	60.09	87.18	96.31	2.6	2.6
#49	-46.87	-46.13	25.56	57.62	-2.3	17.77	15.70	33.18	70.99	84.64	2.3	**0.55**	**0.22**	**64.62**	**89.78**	**97.55**	**7.8**	2.6

metric following (Maninis et al., 2019; Sun et al., 2019). The overall performance is the average of the relative performance over all tasks, namely $\Delta t = \frac{1}{T}\sum_{i=1}\Delta t_i$. The units for relative performance Δt_i and Δt are percentage (%).

Baselines for Comparison. Our baselines include both tree-structured MTL methods and general MTL approaches. For state-of-the-art tree-structured MTL methods, we compare with **What-to-Share**[3] (Vandenhende et al., 2019), **BMTAS**[4] (Bruggemann et al., 2020), **Learn-to-Branch**[5] (Guo et al., 2020), **Task-Grouping**[5] (Standley et al., 2020). For general MTL approaches, we compare with following baselines: the **Single-Task** baseline where each task has its own model and is trained independently, popular MTL methods (e.g., **Cross-Stitch** (Misra et al., 2016), **Sluice** (Ruder et al., 2019), **NDDR-CNN** (Gao et al., 2019), **MTAN** (Liu et al., 2019)), and state-of-the-art NAS-based MTL methods (e.g. **DEN** (Ahn et al., 2019), **AdaShare** (Sun et al., 2019), **AutoMTL** (Zhang et al., 2021)).

We use the same backbone model in all baselines and in our approach for fair comparisons. We use Deeplab-ResNet34 (Chen et al., 2017) and MobileNetV2 (Sandler et al., 2018) as the backbone model and the Atrous Spatial Pyramid Pooling (ASPP) architecture as the task-specific head. Both of them are popular architectures for pixel-wise prediction tasks. The branching points of Deeplab-ResNet34 are generated by our branching point detector and then further customized to be five according to He et al. (2016) to reduce search space, each computation block corresponding to one ConvBNReLU block or one Residual Block. Similarly, MobileNetV2 is split into separate Inverted Blocks firstly and then its branching points are defined by merging adjacent blocks into five larger ones with similar computation cost measured by FLOPs.

4.2 Performance of Recommended Tree-Structured Multi-Task Models

Table 1~2 report the real task performance of the recommended tree-structured multi-task models after training using Deeplab-ResNet34. It reports both absolute values of all evaluation metrics and the relative performance. Results on MobileNetV2 can be found in the Appendix Section H. The first column "Model" lists the index of the recommended models. The specific model structures are shown in the Appendix Section I. Overall, the recommendation of our framework is consistent with the common belief that MTL can achieve higher task accuracy and improved efficiency for each task by leveraging commonalities across related tasks (Caruana, 1997; Ruder, 2017).

The superiority of our recommender can be observed more clearly in Table 2. With different computation budgets (specified by the number of backbone models in column "Com. Budget"), our recommender could always recommend multi-task architectures with high task performance within the computation constraint. Unlike prior works (Sun et al., 2019; Bruggemann et al., 2020), which have to re-train the whole architecture searching framework when the computational requirement changes, there is no extra effort for our framework to re-predict the top architectures.

[3] We implemented the algorithm ourselves since the work is not open-sourced.

[4] It has implementation on MobileNetV2 only.

[5] Its tree-structured multi-task model for Taskonomy is implemented based on the architecture reported in the paper by ourselves since the work is not open-sourced.

Table 2: Performance of top-1 recommended architectures on Taskonomy using Deeplab-ResNet34 under different computation budgets.

Models	Com. Budget	FLOPs (%) ↓	#Params (%) ↓	Semantic Seg. Abs. ↓	Semantic Seg. Δt_1 ↑	Normal Pred. Abs. ↑	Normal Pred. Δt_2 ↑	Depth Est. Abs. ↓	Depth Est. Δt_3 ↑	Keypoint Det. Abs. ↓	Keypoint Det. Δt_4 ↑	Edge Det. Abs. ↓	Edge Det. Δt_5 ↑	Δt ↑
Ind. Models	-	-	-	0.5217	-	0.8070	-	0.0220	-	0.2024	-	0.2140	-	-
#353	w/o	-11.31	-7.90	0.5168	0.9	0.8745	8.4	0.0195	11.4	0.2003	1.0	0.2082	2.7	4.9
#958	4 Models	-22.05	-21.27	0.5268	-1.0	0.8744	8.4	0.0202	8.2	0.1887	6.8	0.2159	-0.9	4.3
#1046	3 Models	-41.93	-41.27	0.5368	-2.9	0.8723	8.1	0.0201	8.6	0.1987	1.8	0.2118	1.0	3.3
#817	2 Models	-60.00	-60.00	0.5891	-12.9	0.8725	8.1	0.0200	9.1	0.1915	5.4	0.2105	1.6	2.3
#0	1 Model	-80.00	-80.00	0.5994	-14.9	0.8390	4.0	0.0265	-20.5	0.1947	3.8	0.2072	3.2	-4.9

The recommender can suggest top architectures on the fly by filtering out architectures that do not satisfy the given requirement.

4.3 Evaluation of the Task Accuracy Estimator

Our recommender can get a **predicted ranking** of all the multi-task architectures based on their estimated task performance from the task accuracy estimator. To evaluate the predicted ranking, we also get an **oracle ranking**, by actual training the multi-task architectures. We use the Pearson correlation coefficient (Pearson's γ) (Benesty et al., 2009) of the *predicted ranking* and the *oracle ranking* to evaluate the efficacy of the task accuracy estimator component. We compare with the correlation of **What-to-Share** (Vandenhende et al., 2019), the only existing branched MTL method which could sort the architectures according to task dissimilarity scores. Table 3 reports the correlation results. For reproducibility, the random seed of the experiments is set as 10. For NYUv2 on MobileNetV2, we also conduct the same experiment with seed 20. The range of γ is $[-1, 1]$. The larger the value of γ is, the stronger the positive correlation, and the better the predicted ranking. Overall, our estimated architecture ranking has a moderately high correlation (i.e., $0.4 \leq \gamma < 0.7$) or even very strong correlation (i.e., $0.7 \leq \gamma < 0.9$) with the oracle ranking according to the interpretation of Pearson's γ (Akoglu, 2018), which demonstrates the reliability of the task accuracy estimator and the effectiveness of our recommender. In contrast, What-to-Share produces negative correlations, indicating that their estimations from task dissimilarity are unreliable. Compared with What-to-Share, our recommender improves the correlation significantly.

Table 3: Pearson's γ between the predicted ranking and the oracle ranking.

Method	Deeplab-ResNet34 NYUv2	Deeplab-ResNet34 Taskonomy	MobileNetV2 NYUv2	MobileNetV2 Taskonomy
What-to-Share	-0.478	-0.147	-0.4901	-0.754
Ours	0.699	0.768	0.504 / 0.772	0.836

4.4 Comparison with State-of-the-Art MTL Methods

Table 4 summarizes the comparisons with state-of-the-art MTL methods for Taskonomy on Deeplab-ResNet34. Results on other datasets and backbone models are included in the Appendix Section J. Generally, the best multi-task architectures suggested by our recommender could achieve competitive or even higher overall task performance as indicated by the Δt columns.

Our work is closest to **What-to-Share** which also ranks all the candidate multi-task architectures and outperforms it by 4.8% in terms of the overall task performance. **Task-Grouping** focuses on deciding how to split the tasks into groups according to the given computation budget so that one group will share the entire backbone model. Compared to **Task-Grouping**, our recommender yields better branching models under the same budget. For instance, when the budget is three models, our top-1 multi-task architecture could achieve higher task performance (3.3% *vs* 2.4%) with lower computation cost (-41.93% *vs* -40%) and number of parameters (-41.27% *vs* -40%) than Task-Grouping.

Table 4: Comparison with state-of-the-art MTL methods for Taskonomy using Deeplab-ResNet34.

Models	FLOPs (%) $\downarrow$	#Params (%) $\downarrow$	Semantic Seg.		Normal Pred.		Depth Est.		Keypoint Det.		Edge Det.		$\Delta t \uparrow$
			Abs. $\downarrow$	$\Delta t_1 \uparrow$	Abs. $\uparrow$	$\Delta t_2 \uparrow$	Abs. $\downarrow$	$\Delta t_3 \uparrow$	Abs. $\downarrow$	$\Delta t_4 \uparrow$	Abs. $\downarrow$	$\Delta t_5 \uparrow$	
Ind. Models	-	-	0.5217	-	0.807	-	0.022	-	0.2024	-	0.214	-	-
What-to-Share	-0.13	-0.01	0.5378	-3.1	0.8696	7.8	0.0233	-5.9	0.2019	0.2	0.2113	1.3	0.1
Task-Grouping	-40.00	-40.00	0.5388	-3.3	0.8743	8.3	0.0202	8.2	0.2037	-0.6	0.2151	-0.5	2.4
Cross-Stitch	0.00	0.00	0.57	-9.3	0.779	-3.5	0.021	4.5	0.199	1.7	0.217	-1.4	-1.6
Sluice	0.00	0.00	0.596	-14.2	0.795	-1.5	0.023	-4.5	0.196	3.2	0.207	3.3	-2.8
NDDR-CNN	8.38	8.20	0.599	-14.8	0.8	-0.9	0.022	0.0	0.196	3.2	0.203	5.1	-1.5
MTAN	-10.55	-9.80	0.621	-19.0	0.787	-2.5	0.022	0.0	0.197	2.7	0.206	3.7	-3.0
Learn-to-Branch	**-68.11**	**-67.67**	0.5214	0.1	0.8503	5.4	0.0235	-6.8	0.2021	0.1	0.2171	-1.4	-0.5
DEN	2.15	-77.60	0.737	-41.3	0.786	-2.6	0.026	-18.2	0.192	5.1	0.203	5.1	-10.4
AdaShare	-5.42	-71.20	0.562	-7.7	0.802	-0.6	0.022	0.0	0.191	**5.6**	0.200	6.5	0.8
AutoMTL	-3.85	-50.10	0.536	-2.7	0.873	8.2	0.021	4.5	0.191	**5.6**	0.197	**7.9**	4.7
Top-1 w/o budget	-11.31	-7.90	0.5168	**0.9**	0.8745	**8.4**	0.0195	**11.4**	0.2003	1.0	0.2082	2.7	**4.9**
Top-1 within 3 models	-41.93	-41.27	0.5368	-2.9	0.8723	8.1	0.0201	8.6	0.1987	1.8	0.2118	1.0	3.3

Compared with manually-design multi-task architectures, **Cross-Stitch**, **Sluice**, **NDDR-CNN**, and **MTAN**, which usually consist of separate networks for each task and define a mechanism for feature sharing between independent networks, our recommended architectures perform higher task performance (4.9% *vs* -1.6%/-2.8%/-1.5%/-3.0%) with computation cost (-11.31% *vs* 0%/8.38%/-10.55%) and the number of parameters reduction (-7.90% *vs* 0%/8.20%/-9.80%).

We also compare with NAS-based methods, including NAS-based branched MTL methods such as **Learn-to-Branch** and **BMTAS**, and NAS-based general MTL approaches such as **DEN**, **AdaShare**, and **AutoMTL**. Learn-to-Branch and BMTAS explore the same tree-structured architecture design space as our recommender. However, since they resort to integrating space searching with network training, the searched multi-task models are usually sub-optimal. Instead, our recommender could overcome the limitation to identify multi-task architectures with higher task performance, 5.4% higher than Learn-to-Branch, and 0.3%/4.1% higher than BMTAS on NYUv2 and Taskonomy using MobileNetV2 as shown in Table 10 and 11 in the Appendix. When comparing to DEN, AdaShare, and AutoMTL, our recommender identifies multi-task architectures with competitive task performance (4.9% *vs* -10.4%/0.8%/4.7%), even though the search space of those methods are larger and more complex than our tree-structured multi-task model space.

5 Conclusion

This paper proposes a tree-structured multi-task model recommender that predicts the top-k architectures with high task performance given a set of tasks, an arbitrary CNN backbone model, and a user-specified computation budget. Our recommender consists of three key components, a branching point detector that automatically detects branching points in any given CNN backbone model, a design space enumerator that enumerates all the multi-task architecture in the design space, and a task accuracy estimator that predicts the task performance of multi-task architectures without performing actual training. Experiments on popular MTL benchmarks demonstrate the superiority and reliability of our recommender compared with state-of-the-art approaches.

Limitations and Broader Impact Statement. Our research facilitates the adoption of deep learning techniques to solve many tasks at once in resource-constraint scenarios. It also promotes the leverage of multi-task learning to increase task performance and computation efficiency. It has a positive impact on applications that tackle multiple tasks such as environment perceptions for autonomous vehicles and human-computer interactions in robotic, mobile, and IoT applications. The negative social impact of our research is difficult to predict since it shares the same pitfalls with general deep learning techniques that suffer from dataset bias, adversarial attacks, fairness, etc.

References

Ahn, C., Kim, E., and Oh, S. (2019). Deep elastic networks with model selection for multi-task learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6529–6538.

Akoglu, H. (2018). User's guide to correlation coefficients. *Turkish Journal of Emergency Medicine*, 18(3):91–93.

Benesty, J., Chen, J., Huang, Y., and Cohen, I. (2009). Pearson correlation coefficient. In *Noise Reduction in Speech Processing*, pages 1–4. Springer.

Bruggemann, D., Kanakis, M., Georgoulis, S., and Van Gool, L. (2020). Automated search for resource-efficient branched multi-task networks. *arXiv preprint arXiv:2008.10292*.

Caruana, R. (1997). Multitask learning. *Machine Learning*, 28(1):41–75.

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., and Yuille, A. L. (2017). Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):834–848.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2015). The loss surfaces of multilayer networks. In *Artificial Intelligence and Statistics*, pages 192–204. PMLR.

Eigen, D. and Fergus, R. (2015). Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2650–2658.

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 *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11543–11552.

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 *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3205–3214.

Guo, P., Lee, C.-Y., and Ulbricht, D. (2020). Learning to branch for multi-task learning. In *International Conference on Machine Learning*, pages 3854–3863. PMLR.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778.

Jelinek, H. F., Donnan, L., and Khandoker, A. H. (2019). Singular value decomposition entropy as a measure of ankle dynamics efficacy in a y-balance test following supportive lower limb taping. In *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 2439–2442. IEEE.

Leang, I., Sistu, G., Bürger, F., Bursuc, A., and Yogamani, S. (2020). Dynamic task weighting methods for multi-task networks in autonomous driving systems. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8. IEEE.

Li, S.-y., Yang, M., Li, C.-c., and Cai, P. (2008). Analysis of heart rate variability based on singular value decomposition entropy. *Journal of Shanghai University (English Edition)*, 12(5):433–437.

Liu, S., Johns, E., and Davison, A. J. (2019). End-to-end multi-task learning with attention. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1871–1880.

Long, M., Cao, Z., Wang, J., and Philip, S. Y. (2017). Learning multiple tasks with multilinear relationship networks. In *Advances in Neural Information Processing Systems*, pages 1594–1603.

Lu, Y., Kumar, A., Zhai, S., Cheng, Y., Javidi, T., and Feris, R. (2017). Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5334–5343.

Maninis, K.-K., Radosavovic, I., and Kokkinos, I. (2019). Attentive single-tasking of multiple tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1851–1860.

Misra, I., Shrivastava, A., Gupta, A., and Hebert, M. (2016). Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3994–4003.

Nekrasov, V., Dharmasiri, T., Spek, A., Drummond, T., Shen, C., and Reid, I. (2019). Real-time joint semantic segmentation and depth estimation using asymmetric annotations. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7101–7107. IEEE.

Pham, H. V., Qian, S., Wang, J., Lutellier, T., Rosenthal, J., Tan, L., Yu, Y., and Nagappan, N. (2020). Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 771–783.

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

Ruder, S., Bingel, J., Augenstein, I., and Søgaard, A. (2019). Latent multi-task architecture learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4822–4829.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520.

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*, pages 746–760. Springer.

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 *Proceedings of the International Conference on Machine Learning*, pages 9120–9132.

Sun, R., Li, D., Liang, S., Ding, T., and Srikant, R. (2020). The global landscape of neural networks: An overview. *IEEE Signal Processing Magazine*, 37(5):95–108.

Sun, X., Panda, R., Feris, R., and Saenko, K. (2019). Adashare: Learning what to share for efficient deep multi-task learning. *arXiv preprint arXiv:1911.12423*.

Suteu, M. and Guo, Y. (2019). Regularizing deep multi-task networks using orthogonal gradients. *arXiv preprint arXiv:1912.06844*.

Vandenhende, S., Georgoulis, S., De Brabandere, B., and Van Gool, L. (2019). Branched multi-task networks: deciding what layers to share. *arXiv preprint arXiv:1904.02920*.

Vandenhende, S., Georgoulis, S., Proesmans, M., Dai, D., and Van Gool, L. (2020). Revisiting multi-task learning in the deep learning era. *arXiv preprint arXiv:2004.13379*, 2:3.

Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. (2019). Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10734–10742.

Wu, B., Li, C., Zhang, H., Dai, X., Zhang, P., Yu, M., Wang, J., Lin, Y., and Vajda, P. (2021). Fbnetv5: Neural architecture search for multiple tasks in one run. *arXiv preprint arXiv:2111.10007*.

Xie, S., Zheng, H., Liu, C., and Lin, L. (2018). Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*.

Zamir, A. R., Sax, A., Shen, W., Guibas, L. J., Malik, J., and Savarese, S. (2018). Taskonomy: Disentangling task transfer learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3712–3722.

Zhang, L., Liu, X., and Guan, H. (2021). Automtl: A programming framework for automated multi-task learning. *arXiv preprint arXiv:2110.13076*.

A Associated Two-Task Models Identifier

As introduced in Section 3.1, to estimate the task accuracy for a multi-task model with three or more tasks, we first need to identify its associated two-task models. To achieve the goal, we propose an iterative algorithm. The pseudocode of the algorithm is shown in Algorithm 1. The layout representation proposed in Section 3.2 is used to refer to the multi-task models for simplicity.

Algorithm 1 Associated Two-Task Layout Identifier

Input: A multi-task layout $\mathbf{L}$
Output: A dictionary D containing associated two-task layouts

$D \leftarrow \{\}$

for $t_1 \in \mathbf{L}$ **do** ▷ For each task in $\mathbf{L}$
  $D[t_1] \leftarrow []$ ▷ Store the associated two-task layouts for t_1

  for $t_2 \in \mathbf{L}$ **do** ▷ Identify for all possible task combinations
    ▷ Skip identical task
    if $t_1 = t_2$ **then**
      continue
    end if

    ▷ Initiate the branching out point to the maximum possible branching point
    $b = B + 1$
    for $i = 1 \to B$ **do** ▷ Check every possible branching point
      $flag \leftarrow$ **false** ▷ Store if t_1 and t_2 share at the i-th branching point
      for $L_i^j \in \mathbf{L}$ **do** ▷ Check for each task set at the i-th branching point
        if $t_1 \in L_i^j$ **and** $t_2 \in L_i^j$ **then**
          $flag \leftarrow$ **true** ▷ There exists a task set has both t_1 and t_2
          break
        end if
      end for
      if $flag$ **is false then** ▷ t_1 and t_2 don't share at the i-th branching point
        $b = i$
        break
      end if
    end for

    ▷ Store the associated two-task layouts for t_1 and t_2 based on b
    $\mathbf{L}' \leftarrow [[\{t_1, t_2\}]_1, \ldots, [\{t_1, t_2\}]_{b-1}, [\{t_1\}, \{t_2\}]_b, \ldots, [\{t_1\}, \{t_2\}]_B]$ ▷ t_1 and t_2 separate at b-th branching point
    $D[t_1]$.append($\mathbf{L}'$)
  end for
end for

B Property of Layout

This section introduces the properties of the proposed layout definitions. Given a layout $\mathbf{L} = [L_1, \ldots, L_i, L_{i+1}, \ldots, L_B]$, for $L_{i+1} = [L_{i+1}^1, L_{i+1}^2, \cdots]$, we have,

$$\forall L_i^q \quad \exists \{L_{i+1}^{a_1}, L_{i+1}^{a_2}, \cdots\} : L_i^q = \bigcup L_{i+1}^{a_p}$$
$$\forall L_{i+1}^p \quad \exists L_i^q : L_{i+1}^p \subseteq L_i^q$$

In other words, these are two properties for successive lists of task sets in a layout.

- Any task set in the i-th list of task sets is the union of some task sets in the $i+1$-th list.
- Any task set in the $i+1$-th list of task sets is a subset of one task set in the i-th list.

The above two properties can be derived directly from the fact that a layout describes a tree.

C Complete Definition of Layout Cut

We first define available branching points of a layout as follows.

Definition C.1 (Available Branching Point). *Given a layout, a branching point is available if the list of task sets in the branching point is the same as that in all the subsequent branching points, and there exists a task set that contains at least 2 tasks.*

From the definition, we could identify available branching points in a given layout by comparing the list of task sets from the B-th branching point up to its previous ones until reaching a different list, then checking whether there exists an eligible task set with at least 2 elements. As shown in Figure 3, the available branching points of the left layout are the last three branching points.

Then the complete definition of *Layout Cut* for enumerating all possible layouts is defined as follows.

Definition C.2 (Layout Cut). *A layout cut is an operator that transforms one layout to another layout by selecting a task set containing at least two tasks from an available branching point and dividing it into two task sets.*

Formally, given a layout $\mathbf{L}$*, the format of a cut* $\mathbf{C}$ *would be like,*

$$\mathbf{C} = \{i, L_i^j, [L_{i\,1}^j, L_{i\,2}^j]\}(|L_i^j| \geq 2)$$

where L_i^j *is the selected task set containing at least 2 tasks from the i-th branching point (available) of* $\mathbf{L}$ *to be divided into 2 sub task sets* $L_{i\,1}^j$ *and* $L_{i\,2}^j$*.*

When applying the cut $\mathbf{C}$ *on the layout* $\mathbf{L}$*, the selected task set* L_i^j *will be divided into* $L_{i\,1}^j$ *and* $L_{i\,2}^j$ *in the new layout* $\mathbf{L}'$*.*

Notice that the i-th branching point must be an available branching point of $\mathbf{L}$*. Then according to Definition C.1, we know that the list of task sets in the i-th branching point and its subsequent branching points are the same. In other words, the selected task set* L_i^j *also exists in the subsequent branching points. Therefore we define that when applying the cut* $\mathbf{C}$ *on the layout* $\mathbf{L}$*, all the task sets in the subsequent branching points that are the same as* L_i^j *will be divided into* $L_{i\,1}^j$ *and* $L_{i\,2}^j$ *in the new layout* $\mathbf{L}'$ *as well.*

For example in Figure 3, there are two different cuts applied on the left given layout which has 4 tasks and 5 branching points. The first cut is $\mathbf{C} = \{3, (3,4), [(3), (4)]\}$, which divides the task set $(3,4)$ into task sets (3) and (4) at the third branching point as well as all its subsequent branching points. Similarly, the second one, $\mathbf{C} = \{4, (1,2), [(1), (2)]\}$, divides the task set $(1,2)$ into (1) and (2) at the fourth and fifth branching points. The effects of the defined cuts are just like the red and orange dashed lines applied on the left layout and the generated new layouts are illustrated on the right side. It's worth mentioning that the available branching points in the new layout $\mathbf{L}'$ may need to be updated as the second example in Figure 3. Specifically, the available branching points of the generated layout $\mathbf{L}'$ would be all the follow-up branching points of the selected branching point of the cut $\mathbf{C}$ and the selected one itself.

Corollary C.2.1. *The maximum number of cuts that can be applied to the initial layout* $\mathbf{L}_0$ *is* $T-1$*, where* T *is the number of tasks.*

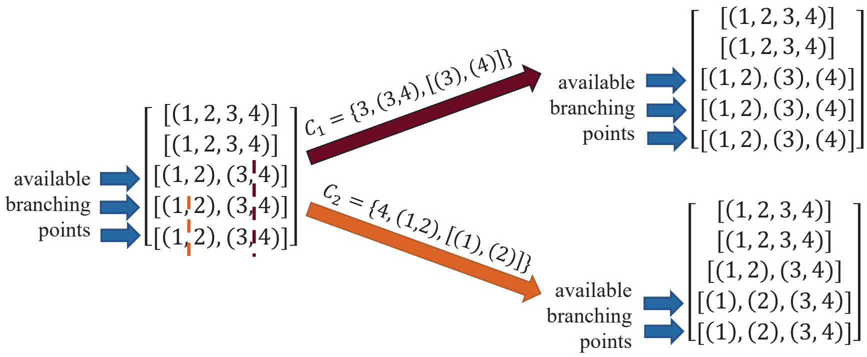


Figure 3: Examples of applying a cut on a given layout.

Proof of Corollary C.2.1. The corollary can be proved by contradiction easily.

If we apply T cuts on the initial layout $\mathbf{L}_0$, $[\{t_1, \dots, t_T\}]$ at the B-th branching point will be divided into $T+1$ task sets, since the task sets in the subsequent branching points are also influenced when applying a cut according to Definition C.2, which means the task sets in B-th branching point will be divided into subsets by all the T cuts. However, it is impossible that $[\{t_1, \dots, t_T\}]$ is divided into $T+1$ task sets since there are only T tasks in it. $\square$

D Layouts Enumerator

As introduced in Section 3.2, we propose a cut-based algorithm to enumerate all possible layouts in the design space fully in a recursive way.

Algorithm 2 Layouts Enumerator

Input: T tasks and a backbone model with B branching points
Output: A set of all possible layouts S

function ENUMERATOR($\mathbf{L}$)
    $S \leftarrow set()$
    ▷ Exit Case: The number of cuts applied to $\mathbf{L}$ is $T-1$
    if $\mathbf{L}.num\_cut = T-1$ **then**
        return S
    end if

    ▷ Enumerate all possible layout cuts on $\mathbf{L}$
    for $i \in \mathbf{L}.avail\_bp$ **do** ▷ Cut for every available branching points
        for $L_i^j \in \mathbf{L}$ **do** ▷ Cut for each task set at the i-th branching point
            if $|L_i^j| = 1$ **then** ▷ If the selected task set has only 1 task, no more cut applied
                continue
            end if
            for $[L_{i\,1}^j, L_{i\,2}^j] \in \text{partition}(L_i^j)$ **do** ▷ For every possible partition of L_i^j
                $\mathbf{C} \leftarrow \{i, L_i^j, [L_{i\,1}^j, L_{i\,2}^j]\}$
                $\mathbf{L}' \leftarrow \text{apply\_cut}(\mathbf{L}, \mathbf{C})$
                $\mathbf{L}'.num\_cut{+}= 1$ ▷ Update the number of cuts applied
                $\mathbf{L}'.avail\_bp \leftarrow [i, \cdots, B]$ ▷ Update the available branching points
                $S.\text{append}(\mathbf{L}')$
                $S' \leftarrow$ENUMERATOR($\mathbf{L}'$) ▷ Enumerate cuts for $\mathbf{L}'$ recursively
                $S {+}= S'$
            end for

23: **end for**
24: **end for**
25: **return** S
26: **end function**
27:
28: $\mathbf{L}_0 \leftarrow [\underbrace{[\{t_1, \ldots, t_T\}], \cdots, [\{t_1, \ldots, t_T\}]}_{B}]$
29: $\mathbf{L}_0.num\_cut \leftarrow 0$
30: $\mathbf{L}_0.avail\_bp \leftarrow [1, \cdots, B]$
31: $S \leftarrow$ ENUMERATOR($\mathbf{L}_0$)

Considering the recursive tree of the enumerator, the time complexity of this algorithm is $O([(2^{T-1} - 1) \cdot B]^{T-1})$, where $T - 1$ is the depth of the recursive tree according to Corollary C.2.1 and $O((2^{T-1} - 1) \cdot B)$ is the branching factor of it. Here $O(2^{T-1} - 1)$ is the time complexity of partitioning a set into two subsets (line 14).

E Proof of Theorem 3.1

Proof of Theorem 3.1. We want to show that for T tasks and a backbone model with B branching points, the layouts enumerated by the cut-based algorithm can cover all tree-structured multi-task architectures. We can prove it by induction.

Base Case: When $B = 1$, all possible tree-structured multi-task architectures can be considered as dividing the T tasks into n groups $\{\{\Gamma_1, \Gamma_2, \cdots, \Gamma_n\} | \bigcup_{i=1}^{n} \Gamma_i = \mathcal{T}, n \leq T\}$ where Γ_i is a task set, indicating the sharing pattern across tasks at the only branching point. Then each tree-structured multi-task model can be represented as a layout $\mathbf{L} = [[\Gamma_1, \Gamma_2, \cdots, \Gamma_n]]$ and such a layout can be generated by applying $n - 1$ cuts on the initial layout $\mathbf{L}_0 = [[\mathcal{T}]]$ inductively.

The first cut is $\mathbf{C}_1 = \{1, \mathcal{T}, [\Gamma_1, \bigcup_{i=2}^{n} \Gamma_i]\}$, which divides the initial task set $\mathbf{T}$ into Γ_1 and $\bigcup_{i=2}^{n} \Gamma_i$. By applying $\mathbf{C}_1$ on $\mathbf{L}_0$, we can get $\mathbf{L}_1 = [[\Gamma_1, \bigcup_{i=2}^{n} \Gamma_i]]$. Then the second cut is $\mathbf{C}_2 = \{1, \bigcup_{i=2}^{n} \Gamma_i, [\Gamma_2, \bigcup_{i=3}^{n} \Gamma_i]\}$, and we can get $\mathbf{L}_2 = [[\Gamma_1, \Gamma_2, \bigcup_{i=3}^{n} \Gamma_i]]$. Similarly the k-th cut is $\mathbf{C}_k = \{1, \bigcup_{i=k}^{n} \Gamma_i, [\Gamma_k, \bigcup_{i=k+1}^{n} \Gamma_i]\}$, and we can get $\mathbf{L}_k = [[\Gamma_1, \Gamma_2, \cdots, \Gamma_k, \bigcup_{i=k+1}^{n} \Gamma_i]]$. Finally, the $(n-1)$-th cut is $\mathbf{C}_{n-1} = \{1, \Gamma_{n-1} \bigcup \Gamma_n, [\Gamma_{n-1}, \Gamma_n]\}$, and we can get $\mathbf{L}_{n-1} = [[\Gamma_1, \Gamma_2, \cdots, \Gamma_{n-1}, \Gamma_n]] = \mathbf{L}$, which is any layout we want. The overall chain would be like,

$$
\begin{aligned}
\mathbf{L}_0 = [[\mathcal{T}]] &\xRightarrow{\mathbf{C}_1 = \{1, \mathcal{T}, [\Gamma_1, \bigcup_{i=2}^{n} \Gamma_i]\}} \mathbf{L}_1 = [[\Gamma_1, \bigcup_{i=2}^{n} \Gamma_i]] \\
&\vdots \\
&\xRightarrow{\mathbf{C}_k = \{1, \bigcup_{i=k}^{n} \Gamma_i, [\Gamma_k, \bigcup_{i=k+1}^{n} \Gamma_i]\}} \mathbf{L}_k = [[\Gamma_1, \Gamma_2, \cdots, \Gamma_k, \bigcup_{i=k+1}^{n} \Gamma_i]] \\
&\vdots \\
&\xRightarrow{\mathbf{C}_{n-1} = \{1, \Gamma_{n-1} \bigcup \Gamma_n, [\Gamma_{n-1}, \Gamma_n]\}} \mathbf{L}_{n-1} = [[\Gamma_1, \Gamma_2, \cdots, \Gamma_{n-1}, \Gamma_n]] = \mathbf{L}
\end{aligned}
$$

In summary, when $B = 1$, any tree-structured multi-task model can be generated by the cut-based layout enumeration algorithm through the cuts chain above.

Inductive Hypothesis: Any tree-structured multi-task model for T tasks based on a backbone model with B branching points can be enumerated through the cut-based layout enumeration algorithm completely.

Inductive Goal: The completeness also holds for any backbone model with $B+1$ branching point.

Inductive Steps: Suppose we have a tree-structured multi-task model M built on a backbone model with B branching point whose layout representation is $\mathbf{L} = [L_1, L_2, \cdots, L_B]$ where L_i is a list of task sets. We can derive a new layout $\mathbf{L}' = [L_1, L_2, \cdots, L_B, L_{B+1}]$ with $B+1$ branching point from $\mathbf{L}$ easily by keeping the first B levels unchanged and adding one more level L_{B+1}, which is equivalent to adding a new level with leaf nodes to M. There are two cases about this new level:

(1) $L_{B+1} = L_B$. Since according to Definition C.2, a cut will influence all the subsequent branching points from the selected one, the cuts we used to generate the layout $\mathbf{L}$ from the initial layout $\mathbf{L}_0$ can generate the new layout $\mathbf{L}'$ as well.

(2) $L_{B+1} \neq L_B$. Suppose we have,

$$L_B = [\Gamma_1, \cdots, \Gamma_n]$$
$$L_{B+1} = [\Gamma'_1, \cdots, \Gamma'_m]$$

Then according to the two properties of a layout introduced in Section B, we have,

$$\forall \Gamma_i \exists \{\Gamma'_{a_1}, \Gamma'_{a_2}, \cdots, \Gamma'_{a_k}\} : \Gamma_i = \bigcup_{j=1}^{k} \Gamma'_{a_j}$$

For any Γ_i with such a set $|\{\Gamma'_{a_1}, \Gamma'_{a_2}, \cdots, \Gamma'_{a_k}\}| > 1$, we could apply $k-1$ cuts whose selected branching points are $B+1$ on the layout $\hat{\mathbf{L}}$ which is derived from the layout $\mathbf{L}$ with one more level $L_{B+1} = L_B$ to divide $\Gamma'_i = \Gamma_i$ at the $B+1$ level into k task sets $\Gamma'_{a_1}, \Gamma'_{a_2}, \cdots, \Gamma'_{a_k}$. This process is the same as the process that we divide the task set $\mathcal{T}$ into n groups in the base case. Therefore if we have r such Γ_i at the B-th level in the target layout $\mathbf{L}'$, we can apply $r(k-1)$ cuts on the layout $\hat{\mathbf{L}}$ inductively to generate $\mathbf{L}'$.

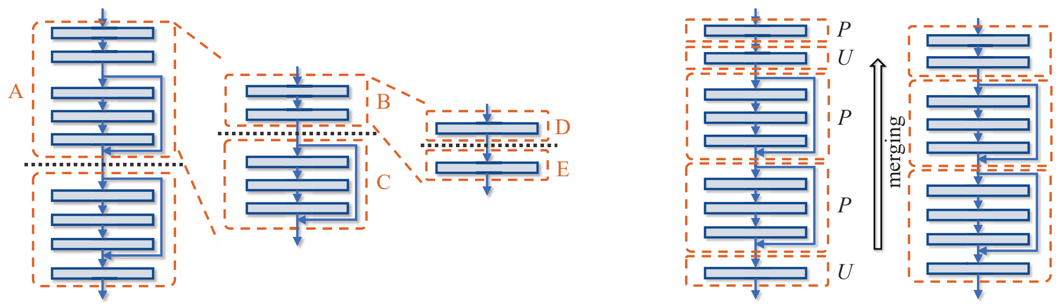
In summary, if any tree-structured multi-task model for T tasks based on a backbone model with B branching point can be enumerated through the cut-based layout enumeration algorithm completely, we can achieve it for any backbone model with $B+1$ branching point as well. Together with the base case, we have proved that the search space of tree-structured multi-task models can be explored by the cut-based layout enumeration algorithm completely. $\square$

F Branching Point Detector

As in described in Section 3.3, we propose a 2-stage branching point detector to partition the user-provided backbone into sequential blocks.

An example is illustrated in Figure 4. In the first stage, if the size of the minimum cut of the computation graph is 1 as in Figure 4(a), the graph can be divided into two subgraphs with only one link between them according to the definition of the minimum cut. Then for each subgraph, the same division occurs if it has a minimum cut of size 1 (e.g., blocks A and B). However, if the size of the minimum cut in the subgraph is greater than 1 (e.g., block C) or the subgraph has only one node (e.g., blocks D and E), the subgraph can no longer be partitioned. After the first stage, each candidate block is assigned a label to reveal its property for merging. If a block contains only unparameterized layers and normalization layers, it is labeled as U, otherwise it is labeled as P. Then the merging carries out from bottom to the top. If the label of a block is U, it should be merged with its next blocks until reaching a block labeled P, otherwise no action is required.

The whole process is illustrated in the following pseudocode.



(a) Stage 1: Divisor

(b) Stage 2: Merger

Figure 4: Process of the branching point detector.

Algorithm 3 Branching Point Detector

Input: A backbone model M
Output: A set of subgraphs S' indicating sequential blocks

function DIVISOR($G = (V, E)$)
  $S \leftarrow set()$
  ▷ Case 1: The graph has only 1 node
  if $|G.V| = 1$ **then**
    S.append(G)
    return S
  end if

  $C \leftarrow min\_cut(G)$ ▷ Find the minimum cut $C = (A, B)$ of G
  ▷ Case 2: The size of the minimum cut is greater than 1
  if $|C.E| > 1$ **then** ▷ $C.E = \{(u, v) \in G.E | u \in A, v \in B\}$
    S.append(G)
  else
    ▷ Case 3: The size of the minimum cut is 1
    $S_A \leftarrow$DIVISOR($G_A = (A, E_A)$) ▷ $E_A = \{(u, v) \in G.E | u \in A, v \in A\}$
    $S_B \leftarrow$DIVISOR($G_B = (B, E_B)$) ▷ $E_B = \{(u, v) \in G.E | u \in B, v \in B\}$
    S.append(S_A, S_B)
  end if
  return S
end function

function MERGER(S)
  S.reverse() ▷ Handle merging from bottom to the top
  for $B \in S$ **do**
    if B.label **is** P **then**
      continue
    else ▷ B is a block with only unparameterized layers and normalization layers
      $S[B] \leftarrow B + S[B+1]$ ▷ Merge B with $B+1$
      if $S[B+1]$.label **is** P **then** ▷ Update the label of new B according to the label of $B+1$
        $S[B]$.label $\leftarrow P$
      else
        $S[B]$.label $\leftarrow U$

end if
end if
end for
end function

$G \leftarrow CG(M)$ ▷ Convert the model to its computation graph (CG)
$S \leftarrow$ DIVISOR(G) ▷ Stage 1: Divide the model into candidate blocks
$S' \leftarrow$ MERGER(S) ▷ Stage 2: Merge auxiliary blocks with adjacent primary blocks

G Hyper-Parameters Settings

Table 5 summarizes the hyper-parameters used in 2-task and multi-task model training. The settings are chosen by experience in previous works (Sun et al., 2019; Zhang et al., 2021).

Table 5: Hyper-parameters for training NYUv2, and Tiny-Taskonomy.

Dataset	lr	lr decay	epoch
NYUv2	0.001	0.5/4,000 iters	20,000
Tiny-Taskonomy	0.0001	0.3/10,000 iters	50,000

H More Recommended Multi-Task Models

Table 6 and 7 report more recommended multi-task models on MobileNetV2. Table 8 is a detailed version of Table 2 to include the top-5 recommended models within different computation budgets.

Table 6: Performance of top-5 recommended models for NYUv2 using MobileNetV2.

Model	FLOPs (%) ↓	#Params (%) ↓	Semantic Seg.			Surface Normal Prediction						Depth Estimation						Δt ↑
			mIoU ↑	Pixel Acc. ↑	Δt_1 ↑	Error ↓		θ, within ↑			Δt_2 ↑	Error ↓		δ, within ↑			Δt_3 ↑	
						Mean	Median	11.25°	22.5°	30°		Abs.	Rel.	1.25	1.25^2	1.25^3		
Ind. Models	-	-	20.36	49.44	-	18.17	16.62	28.37	70.20	85.58	-	0.77	0.28	47.92	78.46	92.81	-	-
#0	-66.67	-66.67	19.36	48.97	-2.9	**17.99**	16.02	**31.43**	70.41	84.65	**2.9**	0.61	**0.23**	60.02	**86.89**	**96.34**	**15.7**	**5.2**
#7	-33.16	-33.41	19.60	48.33	-3.0	18.00	**15.96**	30.39	71.15	85.22	2.6	0.61	**0.23**	60.03	86.78	96.31	15.7	5.1
#11	-30.83	-27.12	20.04	48.82	-1.4	18.21	16.38	30.37	68.95	84.45	1.0	**0.60**	**0.23**	**60.37**	84.04	96.16	15.3	5.0
#10	-22.89	-8.24	19.42	48.49	-3.3	18.06	16.07	29.48	71.12	85.26	1.8	0.63	0.24	58.23	85.57	95.45	13.2	3.9
#9	-16.35	-3.60	**21.09**	**50.25**	**2.6**	18.10	16.73	26.85	**72.13**	**87.11**	-0.2	0.64	0.25	57.10	85.07	95.32	11.6	4.7

Table 7: Performance of top-5 recommended models for Taskonomy using MobileNetV2.

Models	FLOPs (%) ↓	#Params (%) ↓	Semantic Seg.		Normal Pred.		Depth Est.		Keypoint Det.		Edge Det.		Δt ↑
			Abs. ↓	Δt_1 ↑	Abs. ↑	Δt_2 ↑	Abs. ↓	Δt_3 ↑	Abs. ↓	Δt_4 ↑	Abs. ↓	Δt_5 ↑	
Ind. Models	-	-	1.0096	-	0.7662	-	0.0277	-	0.2395	-	0.2681	-	-
#3221	-53.99	-44.86	0.9770	3.2	0.7625	-0.5	0.0277	0.0	0.2232	6.8	0.2519	6.0	3.1
#3220	-52.73	-41.17	1.0120	-0.2	0.7624	-0.5	0.0273	1.4	0.2269	5.3	0.2443	8.9	3.0
#2947	-60.26	-60.05	1.0179	-0.8	0.7510	-2.0	0.0275	0.7	0.2117	11.6	0.2547	5.0	2.9
#3215	-53.99	-44.86	1.0066	0.3	0.7620	-0.5	0.0274	1.1	0.2250	6.1	0.2552	4.8	2.3
#3261	-59.01	-56.27	1.0359	-2.6	0.7489	-2.3	0.0273	1.4	0.2060	14.0	0.2477	7.6	**3.6**

Table 8: Performance of top-5 recommended models for Taskonomy using Deeplab-ResNet34 under different computation budgets.

Models	FLOPs (%) ↓	#Params (%) ↓	Semantic Seg.		Normal Pred.		Depth Est.		Keypoint Det.		Edge Det.		Δt ↑
			Abs. ↓	Δt_1 ↑	Abs. ↑	Δt_2 ↑	Abs. ↓	Δt_3 ↑	Abs. ↓	Δt_4 ↑	Abs. ↓	Δt_5 ↑	
Ind. Models	-	-	0.5217	-	0.8070	-	0.0220	-	0.2024	-	0.2140	-	-
No computation budget													
#353	-11.31	-7.90	0.5168	0.9	0.8745	8.4	0.0195	11.4	0.2003	1.0	0.2082	2.7	**4.9**
#352	-9.43	-1.49	0.5166	1.0	0.8741	8.3	0.0200	9.1	0.1992	1.6	0.2116	1.1	4.2
#958	-22.05	-21.27	0.5268	-1.0	0.8744	8.4	0.0202	8.2	0.1887	6.8	0.2159	-0.9	4.3
#480	-22.83	-21.48	0.5168	0.9	0.8734	8.2	0.0210	4.5	0.2018	0.3	0.2146	-0.3	2.7
#360	-22.05	-21.27	0.5178	0.7	0.8735	8.2	0.0206	6.4	0.2003	1.0	0.2126	0.7	3.4
computation budget: 4 Models													
#958	-22.05	-21.27	0.5268	-1.0	0.8744	8.4	0.0202	8.2	0.1887	6.8	0.2159	-0.9	**4.3**
#480	-22.83	-21.48	0.5168	0.9	0.8734	8.2	0.0210	4.5	0.2018	0.3	0.2146	-0.3	2.7
#360	-22.05	-21.27	0.5178	0.7	0.8735	8.2	0.0206	6.4	0.2003	1.0	0.2126	0.7	3.4
#1037	-41.93	-41.27	0.5300	-1.6	0.8725	8.1	0.0212	3.6	0.1888	6.7	0.2192	-2.4	2.9
#962	-28.24	-27.68	0.5124	1.8	0.8739	8.3	0.0204	7.3	0.1920	5.1	0.2184	-2.1	4.1
computation budget: 3 Models													
#1037	-41.93	-41.27	0.5300	-1.6	0.8725	8.1	0.0212	3.6	0.1888	6.7	0.2192	-2.4	2.9
#1046	-41.93	-41.27	0.5368	-2.9	0.8723	8.1	0.0201	8.6	0.1987	1.8	0.2118	1.0	**3.3**
#943	-40.13	-40.01	0.5308	-1.7	0.8746	8.4	0.0208	5.5	0.1998	1.3	0.2101	1.8	3.0
#1063	-48.11	-47.68	0.5488	-5.2	0.8730	8.2	0.0210	4.5	0.1897	6.3	0.2207	-3.1	2.1
#479	-40.91	-40.22	0.5407	-3.6	0.8724	8.1	0.0198	10.0	0.2040	-0.8	0.2132	0.4	2.8
computation budget: 2 Models													
#817	-60.00	-60.00	0.5891	-12.9	0.8725	8.1	0.0200	9.1	0.1915	5.4	0.2105	1.6	**2.3**
#562	-60.00	-60.00	0.6216	-19.1	0.8713	8.0	0.0202	8.2	0.1976	2.4	0.2011	6.0	1.1
#4697	-60.13	-60.01	0.5985	-14.7	0.8734	8.2	0.0202	8.2	0.1983	2.0	0.2085	2.6	1.3
#6539	-60.91	-60.21	0.5911	-13.3	0.8705	7.9	0.0217	1.4	0.1959	3.2	0.2062	3.6	0.6
#1	-60.00	-60.00	0.6205	-18.9	0.8692	7.7	0.0205	6.8	0.2018	0.3	0.2150	-0.5	-0.9
computation budget: 1 Model													
#0	-80.00	-80.00	0.5994	-14.9	0.8390	4.0	0.0265	-20.5	0.1947	3.8	0.2072	3.2	-4.9

I Recommended Tree-Structured Multi-Task Model Architectures

As introduced in Section 4.1, we conduct experiments on NYUv2 (3 tasks) and Taskonomy (5 tasks) with Deeplab-ResNet34 (5 branching points) and MobileNetV2 (5 branching points). Figure 5 and 6 show the specific structures of the multi-task models recommended by our framework. For simplicity, the model architectures are depicted by their equivalent layouts.

J More Comparisons with State-of-the-Art Methods

Table 9 ~ 11 report more comparisons with state-of-the-art MTL methods. Compared with general MTL methods, our recommended model outperforms existing works in computation cost and the number of parameters with competitive task performance. Compared with branched MTL methods, What-to-Share, BMTAS, and Task-Grouping, in terms of the overall task performance in the Δt columns, our top-1 multi-task architectures could achieve higher results (5.2% *vs* 4.8%/4.9%, 3.1%, *vs* -0.5%/-1.0%/0.9%), which indicates that our recommender has the ability to search out predominant architectures in the tree-structured multi-task model design space.

$$
\underset{\#0}{\begin{bmatrix}[(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\end{bmatrix}}
\underset{\#45}{\begin{bmatrix}[(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\\ [(1,2),(3)]\\ [(1,2),(3)]\end{bmatrix}}
\underset{\#50}{\begin{bmatrix}[(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\\ [(1,2),(3)]\end{bmatrix}}
\underset{\#37}{\begin{bmatrix}[(1,2,3)]\\ [(1,2,3)]\\ [(1,2),(3)]\\ [(1,2),(3)]\\ [(1,2),(3)]\end{bmatrix}}
\underset{\#49}{\begin{bmatrix}[(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\\ [(1,3),(2)]\end{bmatrix}}
$$

(a) Corresponding to Table 1 on Deeplab-ResNet34.

$$
\underset{\#0}{\begin{bmatrix}[(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\\ [(1,2,3)]\end{bmatrix}}
\underset{\#7}{\begin{bmatrix}[(1,3),(2)]\\ [(1,3),(2)]\\ [(1,3),(2)]\\ [(1,3),(2)]\\ [(1,3),(2)]\end{bmatrix}}
\underset{\#11}{\begin{bmatrix}[(1,3),(2)]\\ [(1,3),(2)]\\ [(1,3),(2)]\\ [(1,3),(2)]\\ [(1),(3),(2)]\end{bmatrix}}
\underset{\#10}{\begin{bmatrix}[(1,3),(2)]\\ [(1,3),(2)]\\ [(1,3),(2)]\\ [(1),(3),(2)]\\ [(1),(3),(2)]\end{bmatrix}}
\underset{\#9}{\begin{bmatrix}[(1,3),(2)]\\ [(1,3),(2)]\\ [(1),(3),(2)]\\ [(1),(3),(2)]\\ [(1),(3),(2)]\end{bmatrix}}
$$

(b) Corresponding to Table 6 on MobileNetV2.

Figure 5: Recommended multi-task models on NYUv2.

$$
\underset{\#353}{\begin{bmatrix}[(1),(2,3,4,5)]\\ [(1),(2,3),(4,5)]\\ [(1),(2,3),(4),(5)]\\ [(1),(2,3),(4),(5)]\\ [(1),(2),(3),(4),(5)]\end{bmatrix}}
\underset{\#958}{\begin{bmatrix}[(1,4),(2,3,5)]\\ [(1,4),(2,3),(5)]\\ [(1,4),(2,3),(5)]\\ [(1,4),(2),(3),(5)]\\ [(1,4),(2),(3),(5)]\end{bmatrix}}
\underset{\#1046}{\begin{bmatrix}[(1,4),(2,3,5)]\\ [(1,4),(2,3,5)]\\ [(1,4),(2,3,5)]\\ [(1,4),(2,5),(3)]\\ [(1,4),(2,5),(3)]\end{bmatrix}}
\underset{\#817}{\begin{bmatrix}[(1,4),(2,3,5)]\\ [(1,4),(2,3,5)]\\ [(1,4),(2,3,5)]\\ [(1,4),(2,3,5)]\\ [(1,4),(2,3,5)]\end{bmatrix}}
$$

(a) Corresponding to Table 2 on Deeplab-ResNet34.

$$
\underset{\#3221}{\begin{bmatrix}[(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,4),(2,5)]\\ [(3),(1,4),(2,5)]\end{bmatrix}}
\underset{\#3220}{\begin{bmatrix}[(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,5),(2,4)]\\ [(3),(1),(2,4),(5)]\end{bmatrix}}
\underset{\#2947}{\begin{bmatrix}[(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\end{bmatrix}}
\underset{\#3215}{\begin{bmatrix}[(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,5),(2,4)]\\ [(3),(1,5),(2,4)]\end{bmatrix}}
\underset{\#3261}{\begin{bmatrix}[(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,2,4,5)]\\ [(3),(1,2,4),(5)]\end{bmatrix}}
$$

(b) Corresponding to Table 7 on MobileNetV2.

Figure 6: Recommended multi-task models on Taskonomy.

K Reproducibility Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] See Section 1.

Table 9: Comparison with state-of-the-art MTL methods for NYUv2 using Deeplab-ResNet34.

Model	FLOPs (%) ↓	#Params (%) ↓	Semantic Seg.			Surface Normal Prediction						Depth Estimation						Δt ↑
			mIoU ↑	Pixel Acc. ↑	Δt_1 ↑	Error ↓		θ, within ↑			Δt_2 ↑	Error ↓		δ, within ↑			Δt_3 ↑	
						Mean	Median	11.25°	22.5°	30°		Abs.	Rel.	1.25	1.25^2	1.25^3		
Ind. Models	-	-	26.50	**58.20**	-	17.70	16.30	29.40	72.30	87.30	-	0.62	0.24	57.80	85.80	96.00	-	-
What-to-Share	-1.54	-0.38	25.66	57.64	-2.1	17.75	16.38	29.15	**73.02**	**87.44**	-0.1	0.60	0.22	60.49	87.45	96.55	3.7	0.5
Cross-Stitch	0.00	0.00	25.4	57.6	-2.6	17.2	14.0	41.4	67.7	80.4	8.7	0.58	0.23	61.4	88.4	95.5	3.9	3.3
Sluice	0.00	0.00	23.8	56.9	-6.2	17.2	14.4	38.9	69.0	81.4	7.1	0.58	0.24	61.9	88.1	96.3	3.3	1.4
NDDR-CNN	6.44	5.00	21.6	53.9	-12.9	**17.1**	14.5	37.4	70.9	83.1	7.0	0.66	0.26	55.7	83.7	94.8	-4.4	-3.5
MTAN	22.11	3.70	26.0	57.2	-1.8	17.2	13.9	**43.7**	70.5	81.9	**11.5**	0.57	0.25	62.7	87.7	95.9	2.9	4.2
DEN	10.81	-62.70	23.9	54.9	-7.7	**17.1**	14.8	36.0	70.6	83.4	5.6	0.97	0.31	22.8	62.4	88.2	-36.3	-12.8
AdaShare	-6.24	**-66.67**	24.4	57.8	-4.3	17.7	**13.8**	42.3	68.9	80.5	9.3	0.59	**0.20**	61.3	88.5	96.5	6.2	3.8
AutoMTL	-0.45	-45.10	**26.6**	**58.2**	**0.2**	17.3	14.4	39.1	70.7	83.1	8.0	**0.54**	0.22	**65.1**	**89.2**	**96.9**	**7.8**	**5.3**
Top-1	**-66.67**	**-66.67**	25.23	57.69	-2.8	17.14	15.15	35.85	72.20	85.54	6.0	0.55	0.23	63.85	89.38	97.03	5.8	3.0

Table 10: Comparison with Branched MTL methods for NYUv2 using MobileNetV2.

Model	FLOPs (%) ↓	#Params (%) ↓	Semantic Seg.			Surface Normal Prediction						Depth Estimation						Δt ↑
			mIoU ↑	Pixel Acc. ↑	Δt_1 ↑	Error ↓ Mean	Error ↓ Median	θ, within ↑ 11.25°	22.5°	30°	Δt_2 ↑	Error ↓ Abs.	Rel.	δ, within ↑ 1.25	1.25^2	1.25^3	Δt_3 ↑	
Ind. Models	-	-	20.36	**49.44**	-	18.17	16.62	28.37	70.20	85.58	-	0.77	0.28	47.92	78.46	92.81	-	-
What-to-Share	-8.41	-0.30	**21.10**	49.03	**1.4**	17.82	**15.71**	30.61	72.60	86.08	**3.9**	0.67	0.25	54.71	83.25	94.97	9.3	4.8
BMTAS	-64.46	-33.41	18.98	48.40	-4.4	**17.71**	16.09	29.74	**72.70**	**86.90**	3.1	**0.60**	0.24	**60.73**	**87.25**	96.33	**15.9**	4.9
Top-1	**-66.67**	**-66.67**	19.36	48.97	-2.9	17.99	16.02	**31.43**	70.41	84.65	2.9	0.61	**0.23**	60.02	86.89	**96.34**	15.7	**5.2**

Table 11: Comparison with Branched MTL methods for Taskonomy using MobileNetV2.

Models	FLOPs (%) ↓	#Params (%) ↓	Semantic Seg. Abs. ↓	Δt_1 ↑	Normal Pred. Abs. ↑	Δt_2 ↑	Depth Est. Abs. ↓	Δt_3 ↑	Keypoint Det. Abs. ↓	Δt_4 ↑	Edge Det. Abs. ↓	Δt_5 ↑	Δt ↑
Ind. Models	-	-	0.5217	-	0.807	-	0.022	-	0.2024	-	0.214	-	-
What-to-Share	-5.02	-0.18	1.0283	-1.9	0.7656	-0.1	0.0275	**0.7**	0.2417	-0.9	0.2688	-0.3	-0.5
BMTAS	**-78.47**	**-76.32**	1.0239	-1.4	0.7511	-2.0	0.0322	-16.2	0.2202	**8.1**	0.2508	**6.5**	-1.0
Task-Grouping	-25.11	-5.21	0.9965	1.3	0.7678	**0.2**	0.0287	-3.6	0.2323	3.0	0.2591	3.4	0.9
Top-1	-53.99	-44.86	0.977	**3.2**	0.7625	-0.5	0.0277	0.0	0.2232	6.8	0.2519	6.0	**3.1**

(b) Did you describe the limitations of your work? [Yes]

(c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 5.

(d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...

   (a) Did you state the full set of assumptions of all theoretical results? [Yes] See Section 3.1 and Appendix Section B and C.

   (b) Did you include complete proofs of all theoretical results? [Yes] See Appendix Section E.

3. If you ran experiments...

   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results, including all requirements (e.g., `requirements.txt` with explicit version), an instructive `README` with installation, and execution commands (either in the supplemental material or as a URL)? [Yes] We describe data, models, and experiments in detail. [No] We have not shared code when this answer is written, but plan to open-source the code to assist future research.

   (b) Did you include the raw results of running the given instructions on the given code and data? [N/A] We will include them in the future public source.

   (c) Did you include scripts and commands that can be used to generate the figures and tables in your paper based on the raw results of the code, data, and instructions given? [N/A] We will include them in the future public source.

   (d) Did you ensure sufficient code quality such that your code can be safely executed and the code is properly documented? [N/A] We will ensure it in the future public source.

   (e) Did you specify all the training details (e.g., data splits, pre-processing, search spaces, fixed hyperparameter settings, and how they were chosen)? [Yes] See Section 4.1.

(f) Did you ensure that you compared different methods (including your own) exactly on the same benchmarks, including the same datasets, search space, code for training and hyperparameters for that code? [Yes] See Section 4.1 and Appendix Section G.

(g) Did you run ablation studies to assess the impact of different components of your approach? [N/A] Our approach consists of three components and cannot work without any one of them. In the paper, we verified the effectiveness of each component through experiments or theoretical proofs.

(h) Did you use the same evaluation protocol for the methods being compared? [Yes] See Section 4.1.

(i) Did you compare performance over time? [No]

(j) Did you perform multiple runs of your experiments and report random seeds? [Yes] See Section 4.3.

(k) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] Instead we conduct correlation experiments to show the reliability of our recommender for each dataset with different backbone models and random seeds.

(l) Did you use tabular or surrogate benchmarks for in-depth evaluations? [No]

(m) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [No] Our algorithms can be executed on any computing resources. In this paper, we train the associated 2-task models on NVIDIA 1080ti and the recommended multi-task models on NVIDIA m40 in parallel.

(n) Did you report how you tuned hyperparameters, and what time and resources this required (if they were not automatically tuned by your AutoML method, e.g. in a NAS approach; and also hyperparameters of your own method)? [N/A] We conduct experiments with fixed hyperparameters following prior works.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets…

(a) If your work uses existing assets, did you cite the creators? [Yes] See Section 4.1.

(b) Did you mention the license of the assets? [N/A] Our experiments were conducted on publicly available datasets.

(c) Did you include any new assets either in the supplemental material or as a URL? [No] We did not introduce new datasets.

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [No] Our experiments were conducted on publicly available datasets.

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No] We are not aware of relevant issues in the data we use.

5. If you used crowdsourcing or conducted research with human subjects…

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] We didn't use crowdsourcing.

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] We didn't use crowdsourcing.

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] We didn't use crowdsourcing.