GMorph: Accelerating Multi-DNN Inference via Model Fusion

Qizheng Yang
University of Massachusetts Amherst
Amherst, MA, USA
qizhengyang@umass.edu

Tianyi Yang
University of Massachusetts Amherst
Amherst, MA, USA
tianyiyang@umass.edu

Mingcan Xiang
University of Massachusetts Amherst
Amherst, MA, USA
mingcanxiang@umass.edu

Lijun Zhang
University of Massachusetts Amherst
Amherst, MA, USA
lijunzhang@cs.umass.edu

Haoliang Wang
Adobe Research
San Jose, CA, USA
hawang@adobe.com

Marco Serafini
University of Massachusetts Amherst
Amherst, MA, USA
marco@cs.umass.edu

Hui Guan
University of Massachusetts Amherst
Amherst, MA, USA
huiguan@cs.umass.edu

Abstract

AI-powered applications often involve multiple deep neural network (DNN)-based prediction tasks to support application-level functionalities. However, executing multi-DNNs can be challenging due to the high resource demands and computation costs that increase linearly with the number of DNNs. Multi-task learning (MTL) addresses this problem by designing a multi-task model that shares parameters across tasks based on a single backbone DNN. This paper explores an alternative approach called model fusion: rather than training a single multi-task model from scratch as MTL does, model fusion fuses multiple task-specific DNNs that are pre-trained separately and can have heterogeneous architectures into a single multi-task model. We materialize model fusion in a software framework called GMorph to accelerate multi-DNN inference while maintaining task accuracy. GMorph features three main technical contributions: graph mutations to fuse multi-DNNs into resource-efficient multi-task models, search-space sampling algorithms, and predictive filtering to reduce the high search costs. Our experiments show that GMorph can outperform MTL baselines and reduce the inference latency of multi-DNNs by 1.1-3× while meeting the target task accuracy.

CCS Concepts: • **Computing methodologies** → **Neural networks**.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04.
https://doi.org/10.1145/3627703.3650074

Keywords: machine learning inference, multi-task inference, deep learning systems

ACM Reference Format:
Qizheng Yang, Tianyi Yang, Mingcan Xiang, Lijun Zhang, Haoliang Wang, Marco Serafini, and Hui Guan. 2024. GMorph: Accelerating Multi-DNN Inference via Model Fusion. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3627703.3650074

1 Introduction

Deep learning has had a profound impact on the field of artificial intelligence (AI) by enabling machines to make highly accurate predictions. As deep learning becomes increasingly prevalent in tackling various prediction tasks, modern AI-powered applications such as home robots, augmented reality, and self-driving cars require the execution of multiple deep neural networks (DNNs) to support complex application-level functionalities that involve several related prediction tasks.

Executing multi-DNN inference on resource-constrained devices can be challenging due to the high computation costs of DNNs. Model compression techniques [11, 25, 27, 35] can improve the efficiency of single DNNs. However, the cost of using a separate DNN for each task still increases linearly with the number of DNNs involved, as existing compression techniques cannot leverage commonalities across tasks.

Multi-task learning (MTL) addresses this problem by sharing the parameters of a *single backbone DNN* across multiple tasks [6]. Traditional deep MTL approaches [9, 39, 44, 46, 49, 54, 60, 62, 78] manually design multi-task models with shared parameters, but this can lead to dramatic accuracy loss due to improper feature sharing. More recent work [58, 59, 76, 77] automatically searches for how to share parameters across tasks in the backbone DNN to minimize accuracy loss.

This paper explores an alternative approach: rather than training a single multi-task model from scratch, we propose *fusing multiple task-specific DNNs*, which are *pre-trained separately* and can have *heterogeneous architectures* with no shared backbone, into a single multi-task model. This approach, which we call *model fusion*, is more flexible and easily applicable than MTL because it can fuse any set of pre-trained task-specific models. Furthermore, MTL usually requires a training dataset with labels for each of the tasks of interest, which can be costly to obtain since it is more common to develop task-specific datasets. Model fusion does not have this requirement because it leverages the outputs of the pre-trained DNNs as ground truth. Besides ease of use, our evaluation shows that our model fusion approach outperforms the state-of-the-art MTL approach in terms of inference time and accuracy.

We propose GMorph, a model fusion framework aimed at accelerating multi-DNN inference while maintaining task accuracy. GMorph targets applications where multiple tasks operate on the same data stream, which is a common scenario [47, 71]. For instance, home robotics and augmented reality applications leverage DNNs to perform tasks such as image classification, object detection, semantic segmentation, and face detection on the input vision stream to understand the environment and analyze user behaviors [38, 40].

GMorph explores opportunities for fusing multiple independent task-specific DNNs. It is based on the observation that a DNN is a sequence of computation blocks, such as residual blocks in ResNets [65] or convolution layers in VGGs [57]. A computation block produces tensors, also called *intermediate features*, that are consumed by subsequent computation blocks. GMorph automatically explores *intermediate feature sharing* opportunities where the input features of a computation block of one DNN can be re-used as inputs of a block of another DNN. To enable feature sharing, GMorph fuses the two DNNs into a multi-task model so that shared features are computed only once, leading to significant computation savings.

To identify a multi-task model that meets a given task accuracy requirement while having a low inference cost, GMorph employs two main components: *mutation optimization* and *performance estimation*. The former generates multi-task model candidates by mutating input DNNs to allow for feature sharing across DNNs with different architectures. The latter filters out non-promising multi-task models and fine-tunes the remaining ones to evaluate their task accuracy. GMorph fine-tunes multi-task models using outputs from the well-trained task-specific DNNs through knowledge distillation [24], eliminating the need for task labels.

GMorph makes three main technical contributions to tackle the new model fusion problem. First, it introduces a novel technique called *graph mutation* to fuse multiple DNNs into more resource-efficient multi-task models. GMorph represents multiple DNNs as an *abstract graph*, a novel data structure designed to encode the search space of multi-task models and facilitate graph mutation. It then incrementally generates mutations of the graph that share more and more features across DNN computation blocks using a set of *mutation rules* we introduce. We show empirically that sharing features across computation blocks with *similar* input feature shapes is more likely to preserve accuracy. Based on this insight, GMorph only shares similar input features when generating mutations.

The second technical contribution of GMorph is an algorithm to efficiently explore the search space of multi-task models. Exhaustively searching all possible sharing configurations is impractical because the number of configurations increases exponentially with the number of DNNs and their computation blocks. To explore the search space efficiently, we develop a simulated-annealing-based *search-space sampling algorithm* that first generates multi-task models with diverse sharing patterns and then gradually focuses on mutating promising candidates to further increase feature sharing and reduce their inference costs. This approach is based on the insight that a mutation of a previously evaluated promising candidate is more likely to achieve high efficiency and accuracy. Furthermore, we can reuse the well-trained weights of the promising candidate in their mutations, which reduces the cost of fine-tuning. The resulting algorithm strikes a balance between exploration and exploitation.

Third, GMorph employs novel *predictive filtering* mechanisms to reduce the high cost of accuracy evaluation. Fine-tuning model mutations is necessary to recover task accuracy, but it can be time-consuming. Predictive filtering addresses this challenge by identifying and eliminating non-promising mutations early in the process. We observe that if a mutation cannot meet an accuracy requirement, mutations with more aggressive feature sharing are also likely to fail, so we prune them before fine-tuning. In addition, during fine-tuning, predictive filtering predicts whether the current mutation will likely reach the target accuracy. If not, it terminates the fine-tuning for that mutation.

We evaluate GMorph on seven benchmarks, each consisting of two or three DNNs. Our results show that GMorph is able to reduce the inference latency of the original models on PyTorch [52] by 1.11-2.23×, without any accuracy drop. When allowing for a maximum 2% accuracy drop, GMorph is able to further reduce the inference latency by up to 3.06× on PyTorch and 3.25× on TensorRT. These results demonstrate that GMorph is a complementary optimization approach to existing DNN compilers. Compared to using manually-optimized baselines or TreeMTL [77], the state-of-the-art MTL approach, model fusion does not suffer from significant accuracy drops due to over-sharing or limited speedups due to under-sharing.

Our major contributions are summarized as follows.

- We propose model fusion, a new approach to speed up multi-DNN inference. Given a set of separately pre-trained task-specific models, model fusion automatically generates an efficient multi-task model that preserves the target accuracy.
- We propose a novel graph mutation technique to transform multiple separate DNNs into more efficient multi-task models that share features across DNNs. Graph mutation builds on a novel data structure called abstract graph that encodes the search space of feature sharing and supports mutation rules.
- We design a simulated-annealing-based search-space sampling policy to efficiently explore a vast space of multi-task models. We also propose predictive filtering to filter out unpromising candidates. These techniques reduce the search cost by up to 68-90%.
- We implement these techniques in the GMorph software framework. We evaluate GMorph to demonstrate its effectiveness in accelerating multi-DNN inference with no or only minor accuracy drops.

2 Motivations and Challenges

This section discusses the potential and challenges of cross-DNN feature sharing to accelerate multi-DNN inference.

2.1 Motivation

Multi-DNN inference on the same input stream is quite common in various applications such as autonomous driving [74], augmented reality [38], robotics [30], VR classroom [37], etc. These scenarios have the potential to benefit from sharing features across tasks. Table 1 lists three multi-task applications, *Lifelogging* and *Vision Support* from [40, 47] and *General Language Understanding* from [45], and their tasks, each using a separate task-specific DNN for prediction. Lifelogging records and archives one's daily life by detecting objects and predicting the number of salient objects in the scene using object detection and saliency prediction models. Vision Support examines face images and executes multi-DNNs that predict age, gender, ethnicity, and emotional expression. General Language Understanding consists of tasks from the GLUE benchmark [66], which makes certain predictions given input sentences.

It is possible to share features between DNNs to reduce their computation costs without compromising task accuracy. For example, age and gender prediction could share low-level features extracted from an input face image, instead of having task-specific DNNs to compute these features from scratch. To do so, we can allow a computation block of one DNN to reuse the features computed by some computation block of another DNN.

Figure 1 illustrates how model fusion impacts task accuracy and inference speedups by sharing features across DNNs. In Figure 1(a), three task-specific VGG-16s are utilized in the vision support application to predict emotion, gender, and ethnicity, respectively. In Figure 1(b), two ResNets (ResNet-18 and ResNet-34) are utilized in the lifelogging application to predict the objects and the saliency of each object, respectively. In both sub-figures, we randomly select a computation block of a DNN to share the features of some randomly selected computation block of another DNN. If there are three DNNs, we perform the action twice so that all three models can have some feature sharing. To ensure that the shared features are compatible, we apply re-scaling to the dimensions of the tensors before they are reused by a computation block of another DNN. We sample a total of 400 multi-task models with different feature-sharing patterns across DNNs and measure the inference speedups and accuracy drops after fine-tuning. The inference speedup is the ratio of time spent on executing the task-specific DNNs and a multi-task model. The accuracy drop is the maximum accuracy drop among the tasks. Detailed experiment settings are provided in Section 6.

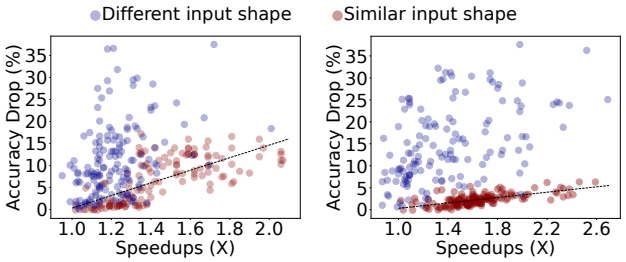
Application	Prediction tasks
Lifelogging	1. *Object detection*: Detecting various objects present in an image 2. *Saliency prediction*: predicting the existence and the number of salient objects in an image
Vision Support	1. *Age prediction*: predicting the age of a person from an input image 2. *Gender prediction*: predicting the gender of a person from an image 3. *Ethnicity prediction*: predicting the ethnicity of a person from an image 4. *Emotion prediction*: recognize the emotion on a person's face
General Language Understand	1. *CoLA task*: predicting whether an English sentence is grammatically plausible 2. *SST-2 task*: determine whether the sentiment of a sentence from movie reviews is positive or negative

Table 1. Example applications and their DNN-based tasks.

The results of Figure 1 show that model fusion by proper feature sharing can achieve 1.1-1.4× inference speedups without compromising task accuracy, and higher speedups can be achieved if a minor accuracy drop is acceptable. Making the right model fusion choices, however, is challenging. Sharing the wrong features can result in significant accuracy drops, which can be larger than 35%. We now discuss the challenges of model fusion and these results in depth.

2.2 Challenges of Model Fusion

Model fusion requires identifying an optimal feature-sharing configuration across DNNs that maximizes multi-DNN inference speedups while maintaining task accuracy. The problem is challenging due to several factors, including the differences in DNN weights and architectures, the large search space of



(a) Three VGG16 (b) ResNet18+ResNet34

Figure 1. Accuracy drop *vs.* inference speedups. Each point is a well-trained multi-task model. Red represents sharing between computation blocks with similar input shapes while blue represents sharing with completely different input shapes. The dotted line interpolates the red points.

possible multi-task models, and the high cost of evaluating the accuracy of candidate multi-task models.

2.2.1 Challenge 1: Fusing Multiple Heterogeneous DNNs without Accuracy Loss.

Model fusion takes as inputs multiple task-specific DNNs that are developed and trained independently. DNNs for different tasks can be heterogeneous and differ in their model weights and architectures for various reasons: DNNs trained on data from different tasks have distinct model weights; the best architecture for one task may not be optimal for another [5]; smaller models are preferred for tasks with less training data to prevent overfitting [14]; different tasks result in varied architectures after optimizations such as model compression [3, 25, 27], even when starting with the same architecture.

Sharing features across DNNs with different weights and architectures is challenging because none of the computation blocks in one DNN may produce identical features as those of the other DNNs. Allowing any computation block of one DNN to reuse features from another DNN could cause a drop in task accuracy, which may not be avoidable even with fine-tuning.

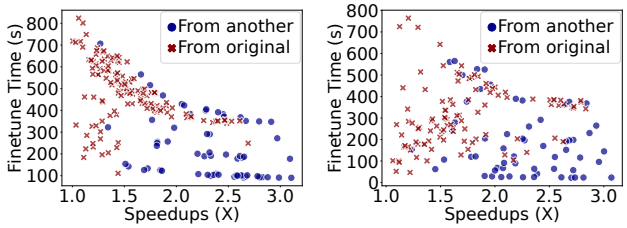
Insight. Our empirical study suggests that sharing features between DNN computation blocks that consume features of *similar* shapes is more likely to recover task accuracy via fine-tuning compared to sharing features between computation blocks with vastly different feature shapes. A similar shape means that any or all of the width, height, and channel dimensions are the same. Figure 1 illustrates this finding. The red points correspond to multi-task models resulting from feature sharing between DNN computation blocks that consume features of similar shapes. The blue points correspond to feature sharing between computation blocks that consume completely different feature shapes, i.e., none of the dimensions are the same. The red points dominate the Pareto frontier between accuracy drop and speedups. This observation motivates us to develop a *graph mutation technique* that focuses on feature sharing between computation blocks with similar input shapes. The formal problem definition is detailed in Section 4.1.

2.2.2 Challenge 2: Exploring the Large Search Space.

Even after restricting feature sharing to similar input shapes, the search space of possible feature-sharing configurations grows exponentially with the number of DNNs and the number of computation blocks in each DNN. For example, in our experiment using three VGG-16 models, we iterate over all the possible mutations of the models and find more than 350K multi-task model variants. These variants differ in which features are shared between which pair of DNNs, leading to a broad range of accuracy drops (e.g., 0%-17% in Figure 1 for red points) and inference speedups.

Insight. Our insight to address this challenge is to efficiently sample the configuration space by leveraging the dependencies among multi-task model variants. Specifically, we can mutate a promising multi-task model into another by increasing the level of feature sharing. This gives two advantages over mutating the original multi-DNNs. First, it is more likely to result in a multi-task model that can achieve higher speedups. Second, the new multi-task model mutations inherit the well-trained weights of the promising candidate and thus need a much shorter fine-tuning time to recover accuracy. Figure 2 illustrates the two benefits. The multi-task models are derived from three VGG-13 that predict *Age*, *Gender* and *Ethnicity* on the *UTKface* [79] dataset.

Based on this insight, we designed a simulated-annealing-based search-space sampling policy to reduce the search cost. The policy first explores the search space by mutating the input multi-DNNs to generate diverse multi-task model variants and then exploits the search results by further mutating satisfactory variants to identify more efficient solutions.



(a) Accuracy Drop = 1% (b) Accuracy Drop = 2%

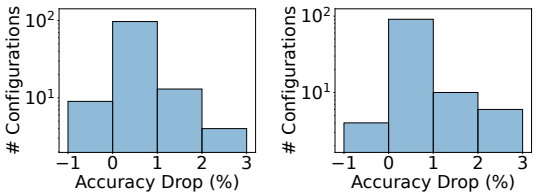
Figure 2. Fine-tuning time *vs.* inference speedups. Each point is a well-trained multi-task model mutated either from original multi-DNNs (*From original*) or another promising multi-task model candidate that meets the target accuracy (*From another*). Mutations of promising candidates are more likely to achieve higher speedups and require significantly less fine-tuning time.

2.2.3 Challenge 3: The High Cost of Accuracy Evaluation.

After mutating new multi-task model candidates,

it is necessary to train them in order to estimate their accuracy. Mutations inherit their weights from another trained candidate or input DNNs, so fine-tuning is sufficient. Fine-tuning is faster than training from scratch but it is still time-consuming, especially when evaluating a large number of candidates. Unfortunately, predicting the task accuracy of a candidate based only on the model architecture, without fine-tuning, is not feasible. Two candidates with the same architecture may have different weight initialization, depending on which multi-task model they are mutated from, resulting in different task accuracy.

Figure 3 illustrates this problem. Each sub-figure considers a different multi-task model architecture. We train each architecture with different weight initialization and measure how this impacts the model accuracy compared to the input task-specific DNNs. The multi-task models are derived from two VGG-13s that predict *Age* and *Gender* on the *UTKface* dataset, respectively. When considering 123 and 111 different weight initializations, respectively, the accuracy drop varies from −1% (accuracy increase) to 3%.



(a) Architecture 1 (b) Architecture 2

Figure 3. Impact of different initialization configurations on the accuracy drop of two model architectures.

Insight. To mitigate this problem, we develop predictive filtering mechanisms based on two key observations. First, we observe that a multi-task model that exhibits more aggressive feature sharing than a non-promising one from previous trials is likely to be non-promising as well. As shown by the interpolated dotted curve in Figure 1, the higher the speedups achieved through feature sharing, the higher the accuracy drops. This observation motivates us to design filtering strategies that identify non-promising candidates without undergoing the fine-tuning process based on the accuracy of their counterparts with less aggressive feature sharing. Second, the test accuracy of a DNN usually converges to an asymptotic value as the number of iterations increases. By using a sequence of test accuracies to estimate the convergence rate and then extrapolating the learning curve, we can approximate the final accuracy of a multi-task model and predict whether the model has the potential to reach the target accuracy. This enables us to terminate non-promising candidates early.

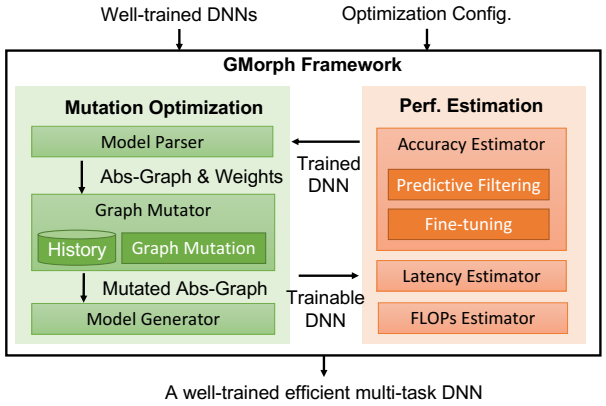


Figure 4. Overview of GraphMorph Framework

3 Overview of GMorph

This section gives an overview of GMorph. GMorph is a software framework that automatically enables feature sharing across DNNs to accelerate multi-DNN inference. As Figure 4 shows, its input has two parts:

- A set of well-trained DNNs, which in our current implementation are stored in PyTorch checkpoint (.pt, .pth) format. These DNNs consume the same input but could have different weights and architecture.
- A configuration file for the graph mutation optimization. The set of configurations includes (1) the metric to be optimized (i.e., latency or FLOPS) and the acceptable task accuracy threshold, (2) representative DNN inputs for multi-task model fine-tuning, (3) testing data and scripts to evaluate task accuracy, (4) optimization hyperparameters such as the learning rate for model fine-tuning, the maximum fine-tuning epochs, and the maximum rounds or time for the graph mutation optimization.

The output is a well-trained multi-task model that is more efficient in terms of the optimized metric (e.g., latency) while meeting a user-specified task accuracy threshold. It is worth noting that GMorph depends solely on representative inputs to fine-tune the multi-task model, thus eliminating the requirement for training datasets with task labels.

The GMorph framework consists of two main components, as illustrated in Figure 4. The *Mutation Optimization* component takes as input either the multi-DNNs provided by the user or a well-trained multi-task DNN from the performance estimation component. It generates a *trainable multi-task DNN* with features sharing, which is then evaluated by the *Performance Estimation* component to produce a well-trained multi-task DNN. The well-trained multi-task DNN is subsequently utilized in the mutation optimization in the next round. Algorithm 1 outlines the graph mutation optimization algorithm in GMorph.

Algorithm 1 Graph Mutation Optimization

Require: T_{acc} ▷ Threshold for accuracy drop.
Require: $\{DNN_i\}$ ▷ The set of input DNNs.
Require: N ▷ Total number of rounds.
1: $\mathcal{G}_{input}.graph, \mathcal{G}_{input}.weights$ = parse($\{DNN_i\}$)
2: $elites \leftarrow \emptyset$ ▷ Initialize elite candidates.
3: $evaluated \leftarrow \emptyset$ ▷ Initialize evaluated candidates.
4: $best\_model \leftarrow \{DNN_i\}$ ▷ Initialize the best model.
5: **for** $iter = 1, 2, \cdots, N$ **do**
6: /* Step 1: sample a new candidate and mutate it. */
7: $\mathcal{G}_{base} \leftarrow$ sampleBaseGraph($\mathcal{G}_{input}$, $elites$)
8: $node\_pairs \leftarrow$ sampleNodePairs($\mathcal{G}_{base}.graph$)
9: $\mathcal{G}_m.graph \leftarrow$ mutate($\mathcal{G}_{base}.graph$, $node\_pairs$)
10: $trainable\_model$ = generate($\mathcal{G}_m$, $\mathcal{G}_{base}.weights$)
▷ Generate the trainable multi-task model; initialize the model with well-trained weights from the base model.
11: /* Step 2: evaluate the multi-task model. */
12: $trained\_model \leftarrow eval(trainable\_model, evaluated)$
13: $\mathcal{G}_m.graph, \mathcal{G}_m.weights \leftarrow$ parse($trained\_model$)
14: **if** $trained\_model.accuracy >= T_{acc}$ **then**
15: $elites$.add($\mathcal{G}_m$)
16: updateBestModel($best\_model$, $trained\_model$)
17: **end if**
18: $evaluated$.add($\mathcal{G}_m$)
19: **end for**
20: **return** $best\_model$

Mutation Optimization. Mutation optimization comprises three main modules executed in sequential order: (1) The *Model Parser* transforms the input multi-DNNs or a well-trained multi-task DNN into an **abstract graph** and extracts model weights (Lines 1 and 13). The abstract graph is the core data structure for representing multi-DNNs and multi-task DNNs to automate graph mutation optimization. (2) The *Graph Mutator* saves abstract graphs and model weights in its *History Database*. It then selects an abstract graph $\mathcal{G}_{base}$ from the History Database and applies a *graph mutation* pass on it to create a new abstract graph $\mathcal{G}_m$ (Lines 7-9). $\mathcal{G}_{base}$ is called *base abstract graph* while $\mathcal{G}_m$ is called *mutated abstract graph*. The graph mutator uses a simulated annealing-based sampling policy to select the abstract graph and the mutation operations applied to it. (3) The *Model Generator* materializes a trainable multi-task DNN based on the mutated abstract graph and initializes its weights using the well-trained weights of the base abstract graph (Line 10).

Performance Estimation. Performance estimation computes several commonly-used performance metrics including latency, FLOPs, and accuracy (Line 12). The *Latency Estimator* measures the inference time by executing the trainable DNN on the targeted devices. The *FLOPs Estimator* counts the total number of Floating-Point Operations. The *Accuracy Estimator* is the most time-consuming estimator since it fine-tunes the trainable DNN to check whether it can meet the target accuracy threshold. To reduce the model fine-tuning costs, the accuracy estimator uses *predictive filtering* to filter out non-promising models as early as possible. Fine-tuning adjusts the weights of the multi-task model to generate output features similar to those of the input multi-DNNs, thereby eliminating the need for task labels.

We next explain in detail the two components in GMorph.

4 Mutation Optimization

This section first gives a formal definition of the abstract graph data structure and the mutation optimization problem and then elaborates on the modules.

4.1 Abstract Graph

An abstract graph is a data structure generated by the model parser that represents multi-DNNs with the same input or a multi-task model in a linearized format. Feature sharing between two DNNs would lead to a tree-structured model that consists of some shared computation blocks and two branches after the shared computation blocks. Therefore, an abstract graph represents each DNN as a sequence of computation nodes and the multi-task model as a tree of computation nodes to enable graph mutation optimization.

Definition 1 (Abstract Graph)**.** *An abstract graph (abs-graph) is a tree variant of a Directed Acyclic Graph (DAG),* $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{D})$*, where*

- *each node* $n \in \mathcal{V}$ *is an operator (e.g., Conv2d, ReLU, Linear, etc) or a block of operators (e.g., Residual Module in ResNet) in a DNN that transforms input features to output features,*
- *each edge* $e \in \mathcal{E}$ *represents the data dependency between two nodes,*
- *the root of the tree is a placeholder for the input tensor shared by all the DNNs, and*
- $\mathcal{D}$ *is a feature shape dictionary that maps a specific feature shape to the set of nodes in the abstract graph that can take features of that shape as inputs.*

Specifically, a node in an abs-graph is represented as a tuple *(task_id, op_id, op_type, input_shape, capacity, parent, children)*. The *task_id* indicates which task or DNN the node comes from. The *op_id* represents the order of the node in the DNN based on a topological sort. The *op_type* is the operator or module type (e.g., CONV or Linear) in the DNN model. The *input_shape* stores the shape of the input features taken by the node. The *capacity* indicates the number of parameters of each node. The *parent* and *children* contain the node before it and the nodes after it respectively.

Some nodes in the abstract graph cannot share input features with other nodes without negatively affecting accuracy. Recall that our empirical study in Section 2.2.1 suggests that sharing features between DNN computation blocks that consume features of similar shapes is more likely to recover task accuracy via fine-tuning compared to vastly different feature shapes. Based on the finding, graph mutation optimization

focuses on feature sharing between nodes with similar input feature shapes, which we refer to as *input-shareable node pairs*.

Definition 2 (Input-Shareable Node Pairs)**.** *In an abstract graph, two nodes n and m are considered input-shareable if they accept input features that have compatible shapes in at least one dimension. If this condition is met, the two nodes can form an input-shareable node pair,* (n, m) *indicating that m reuses n's input features, or* (m, n) *indicating the opposite.*

When a node m reuses another node n's input features, the part of the computation producing m's input features can be removed, leading to computation savings. If nodes n and m have input features with different shapes, an additional *re-scale* operator is inserted before node m. For convolutional neural networks, this operator resizes the width and height of the features using interpolation techniques and adjusts the channel dimension using a 1x1 convolution layer. For transformer-based models, the channel dimension corresponds to the token length (i.e., sequence length for text inputs and number of patches for image inputs).

The abs-graph built from the input multi-DNNs defines a configuration space for the graph mutation optimization. It could have many input-shareable node pairs, each resulting in different computation savings and effects on task accuracy. Formally, we define the optimization as follows:

Definition 3 (Graph Mutation Optimization)**.** *Graph mutation optimization aims to identify the best set of input-shareable node pairs,* $\{(n_i, m_i)\}$ *from the abstract graph of multi-DNNs, that optimizes inference efficiency while meeting target task accuracy by making each pair share input features.*

Exhaustively searching for the global optimal solution is time-consuming. Instead, GMorph aims to identify a local optimum under a time budget. To achieve the goal, Mutation Optimization effectively samples the configuration space to reduce the number of configurations to evaluate while Performance Estimation efficiently measures the performance of each sampled configuration. We next explain each module in the Mutation Optimization component. Performance Estimation is elaborated in Section 5.

4.2 Model Parser

The *Model Parser* converts a set of DNN models from the user or a trained multi-task DNN from the Performance Estimation component into an abs-graph and corresponding weights. It allows GMorph to automatically support a wide range of deep learning models as inputs. Its design is motivated by the observation that common DNNs are a sequence of *computation blocks*, such as residual blocks in ResNet18 [28] and convolution blocks in VGG16 [57]. These computation blocks are nodes in abs-graphs.

The parsing procedure is as follows. We trace the computation graph of input models where each basic operator (e.g., Conv2d, Linear, etc) is a node in an abs-graph. If an input model consists of a sequence of customized modules (e.g., Residual Block), each module will be mapped into a node. The weights of the DNNs are saved as key-value pairs, where each key is the *(task_id, op_id)* of a node in the abs-graph and the value is the parameters of the operator or the group of operators. The abs-graph and weights later are passed and stored into the *History Database* in the Graph Mutator.

4.3 Graph Mutator

Graph Mutator first samples an abs-graph as a *base abs-graph* and a set of input-sharable node pairs from the base abs-graph. It then makes each pair share the input tensor by applying a graph mutation pass to produce a *mutated abs-graph*. We next explain the sampling policy, mutation operations, and the graph mutation pass.

4.3.1 Sampling Policy. The sampling policy uses a variant of the simulated annealing algorithm [61] to balance between exploitation and exploration. During graph mutation optimization, the sampling policy maintains a list of *elite candidates*, which are the abs-graphs of multi-task models after graph mutations and meet the accuracy requirement. It also records the abs-graph of the user-provided input DNNs (called the *original abs-graph*).

In each graph mutation optimization iteration, the sampling policy takes an elite candidate as the base abs-graph with a probability p, or takes the original abs-graph with a probability $1 - p$. It then randomly selects a set of input-sharable node pairs from the base abs-graph to perform a graph mutation pass (See Algorithm 1 Lines 7-8). The mutated abs-graph is added to the elite candidate list if it meets the target accuracy requirement, or dropped otherwise.

GMorph gradually prioritizes sampling from elite candidates over the original graph to improve sampling effectiveness. In the early iterations of optimization, the policy tends to take the original graph as the base abs-graph, to explore more potential elite candidates that meet the accuracy constraint, whereas in the later iterations, the policy tends to find base abs-graphs from the elite candidates. The probability p of sampling an elite candidate is updated as:

$$p = \left(1 - \exp\left(-\frac{1-\Delta}{T_c \times T_i}\right)\right) \times \sqrt{\frac{N_c}{N_i}},$$

where Δ is the accuracy drop after finetuning, T_c is the current temperature, T_i is the initial temperature, N_c is the current number of elite candidates, and N_i is the max number of elite candidates to record. Current temperature T_c is updated with $T_c = T_i \times \alpha^{iter}$, where α is a constant for reducing temperature, and $iter$ is the current number of iteration. In our experiment, we set α to 0.99, N_i to 16, T_i to 90.

4.3.2 Mutation Operations. Depending on each sampled pair of input-shareable nodes, Graph Mutator selects one of the five pre-defined mutation operations. Figure 5 illustrates

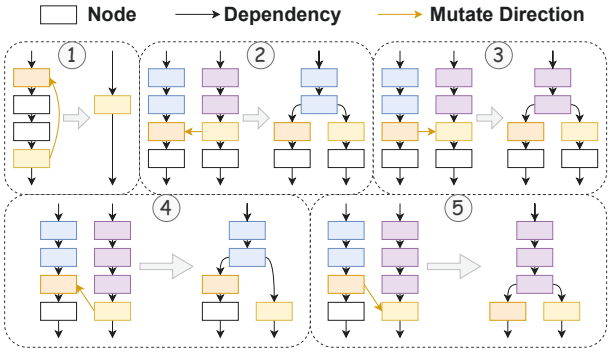


Figure 5. Different mutation operations. Inside each panel, the left graph is the part of a base abs-graph while the right graph is the part of the mutated abs-graph after applying a mutation operation.

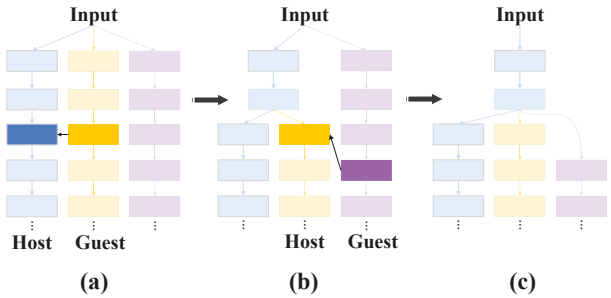


Figure 6. A graph mutation pass that performs two mutation operations. (a) The base abs-graph has three branches (left, middle, and right) corresponding to three tasks. (b) The intermediate abs-graph. (c) The mutated abs-graph.

the mutation operations, which fall into two types, in-branch mutation (①) and cross-branch mutation (② ③ ④ ⑤).

In-branch mutation is applied when the pair of input-shareable nodes reside in the same branch of the base abs-graph and thus the nodes belong to the same task. In ①, in-branch mutation removes the nodes between the pair, reducing the computation for the corresponding task.

Cross-branch mutation is applied when the pair of input-shareable nodes lie at different branches of the abs-graph– that is, the two nodes belong to different tasks. Given an input-sharable node pair (n, m), the node n' branch is the *host* while the node m's branch is the *guest*. The mutation operation will make the node m share the input tensor of n and remove other nodes on the guest branch whose topological order is lower m. In Figure 5, the left branch in panels ② and ④ is the host while the right branch in panels ③ and ⑤ is the host.

Cross-branch mutation can reduce computation costs in two ways. First, it allows two tasks to share nodes whose order is lower than the input-sharable nodes. In ② and ③, the first two nodes are shared across the two tasks. Second, for an input-sharable node pair (n, m), when $m.op\_id > n.op\_id$, then the node m's task uses less number of nodes after the mutation, implying the potentially reduced computation cost for the task. For example, in ④, the total number of nodes for the task on the right side (guest branch) is reduced from 4 (3 purple + 1 yellow) to 3 (2 blue + 1 yellow).

4.3.3 A Graph Mutation Pass. Given the sampled base abs-graph and a sequence of input-sharable node pairs, a graph mutation pass applies mutation operations to the base abs-graph to produce a mutated abs-graph. Figure 6 illustrates a graph mutation pass that performs two mutation operations. The first mutation operation makes the deep yellow node in the middle branch share the input tensor of the deep blue node in the left branch. The second mutation operation makes the deep purple node use the input tensor of the deep yellow node from the middle branch.

4.4 Model Generator

The *Model Generator* converts a mutated abs-graph into a multi-task model that is ready to fine-tune. It will fetch the well-trained weights of the base abs-graph from the History Database and use these weights to initialize the mutated abs-graph. Each node of a mutated abs-graph will be mapped back to a module of PyTorch.

5 Performance Estimation

The *Performance Estimation* component evaluates various performance metrics of a multi-task model including the FLOPs, latency, and accuracy. We focus on the accuracy estimator as it is the most time-consuming module. The accuracy estimator uses predictive filtering to reduce the fine-tuning costs and distillation-based fine-tuning to address the lack of task labels.

5.1 Predictive Filtering

We designed two predictive filtering approaches to reduce the long fine-tuning time of multi-task models. The first approach *rule-based filtering* filters out trainable models that are unlikely to meet the accuracy constraint before fine-tuning. The second approach *predictive early termination* predicts the final accuracy of the training model by extrapolating the learning curve and early terminating the fine-tuning if the predicted accuracy is not promising.

Rule-based Filtering. The rule is based on the capacity of a trainable multi-task model: *when a mutated abs-graph is trained and shown to be non-promising, then all mutated abs-graphs that are more aggressive in feature sharing are also non-promising.* The capacity refers to the number of weights. A mutated abs-graph is more aggressive if all the following conditions are met: (1) it has fewer total capacities; (2) it has fewer total capacities for each task; (3) It has fewer task-specific capacities for each task; and (4) It has more

shared capacities between tasks. During the graph mutation optimization, GMorph extracts the capacity of each mutated abs-graph in the Model Parser, and applies the rule in the accuracy estimator. If the mutated abs-graph is more aggressive compared with existing non-promising abs-graphs, it is skipped without fine-tuning.

Predictive Early Termination. GMorph predicts whether a training model is promising based on the *convergence rate* [56]. Specifically, we measure the test accuracy once every $delta$ epochs. We use four sequentially measured test accuracy $(f(x), f(x+\delta), f(x+2\delta), f(x+3\delta))$ to estimate a convergence rate α, where x is the x-th epoch. Based on α, we can estimate the future test accuracy $f(x+4\delta)$, $f(x+5\delta)$, etc. Iteratively, we can estimate the final accuracy after T epochs $f(T)$. The formula for computing the rate of convergence is:

$$\alpha = \frac{\log(|f(x+2\delta) - f(x+3\delta)|) - \log(|f(x+\delta) - f(x+2\delta)|)}{\log(|f(x+\delta) - f(x+2\delta)|) - \log(|f(x) - f(x+\delta)|)},$$

If the predicted test accuracy $f(T)$ is not promising, we early terminate the fine-tuning process. Otherwise, we fine-tune the model for the next δ epochs and predict again until the fine-tuning process ends.

5.2 Distillation-based Fine-Tuning

Fine-tuning the weights of a multi-task model is necessary to recover its task accuracy. As it is common for tasks to have their separate training data, there is no ground truth label available for all tasks given each input data point. Inspired by DNN knowledge distillation [29], GMorph fine-tunes a multi-task model to produce similar output features as the original task-specific DNNs, forgoing the need for task labels. The optimization objective is the weighted sum of the ℓ_1 loss from all tasks, where each loss is the ℓ_1 distance between the multi-task model's output features and the single-task model's output features. To reduce the fine-tuning time, we set an early-stopping condition such that the fine-tuning stops once the test accuracy meets the requirement.

6 Evaluation

We conduct a set of experiments to evaluate the efficacy of model fusion using GMorph. Our experiments are designed to answer the following major questions: 1) How much inference speedup we could get from GMorph given different task accuracy targets? 2) How effective are the sampling policy and the predictive filtering in reducing the search cost? We first describe the experiment settings in Section 6.1, then report our experiment results in Sections 6.2-6.5.

6.1 Experiment Settings

Tasks and Models. We adopt the tasks from the three applications described in Table 1. Table 2 lists the DNNs and datasets per benchmark. The DNN models and the datasets are used in previous work [18, 41, 42, 45, 47, 75]. B1/2/3 correspond to *Vision Support*, B4/5/6 correspond to *Lifelogging*, and B7 correspond to *General Language Understanding*. The goal of B1-B2 is to show that GMorph can achieve better performance compared to MTL even with DNNs with the same architecture. B3-B5 test the support of GMorph for convolution-based DNNs with heterogeneous architectures. In particular, B5 considers DNN models from different families. B6-B7 test the performance of GMorph on transformer models [13, 16] trained on vision and NLP tasks, which have different numbers of layers and hidden sizes.

B1 uses the *UTKFace* [79] dataset, which consists of images with annotations of age, gender, and ethnicity. **B2** and **B3** use the *FER2013* [23] dataset for EmotionNet and the *Adience* [17] dataset for AgeNet and GenderNet. **B4/5/6** use *PASCAL VOC2007* [18] for ObjectNet and *SOS* [75] for SalientNet. **B7** uses *CoLA* and *SST-2* datasets from the GLUE benchmark[66]. The detailed information on the datasets and the accuracy of each task evaluated on the datasets are reported in Appendix A.

Benchmark	Tasks and Models
B1	AgeNet: VGG-13; GenderNet: VGG-13; EthnicityNet: VGG-13
B2	EmotionNet: VGG-16; AgeNet: VGG16; GenderNet: VGG-16
B3	EmotionNet: VGG-13; AgeNet: VGG16; GenderNet: VGG-11
B4	ObjectNet: ResNet-34; SalientNet: ResNet-18
B5	ObjectNet: ResNet-34; SalientNet: VGG-16
B6	ObjectNet: $\text{ViT}_{\text{Large}}$; SalientNet: ViT_{Base}
B7	CoLANet: $\text{BERT}_{\text{Large}}$; SSTNet: $\text{BERT}_{\text{Base}}$

Table 2. Tasks and Models.

Optimization Parameters. We set the configuration file for GMorph as follows. (1) We set inference latency as the metric to optimize. For B1/2/3, the accuracy metric is the classification accuracy, while for B4/5/6, the accuracy metric is mean average precision (mAP). In B7, the accuracy metric for CoLANet is Matthews correlation coefficient while for SSTNet is the classification accuracy. (2) The representative input data is sampled from all the datasets in a benchmark, with a size of 20K (10K for B4 to B7). (3) We use the same testing data split to evaluate the multi-task model's quality. (4) For each generated mutated graph in B1, B4, and B5, the number of epochs for fine-tuning is 35, with a batch size of 64; for those in B2 and B3, the number of epochs is 40, with a batch size of 128; in B6 and B7, the number of epochs is 16 with a batch size of 32. The validation accuracy is measured every 5 epochs for B1-B5 and every 2 epochs for B6-B7. We use the adam optimizer [36].

The initial learning rate for fine-tuning a multi-task model is the same as the initial learning rate to train the input DNNs. If the input DNNs are trained with different learning rates, then we use the minimum between all the models. The detailed learning rate settings are reported in Appendix A. Early-stopping condition is enabled for all the baselines. We use three task accuracy targets corresponding to accuracy drop thresholds of 0%, 1%, and 2%. The graph mutation optimization uses a total number of 200 iterations.

Baselines for Comparison. We compare GMorph with the following baselines.

- **Original.** The input DNNs given to GMorph without any feature sharing. The models are rewritten as one computation graph where tasks share the input tensor.
- **All-shared.** This is the most commonly used multi-task architecture where all identical layers are shared across tasks [54]. When DNNs are of different architectures, they bring no (or limited) speedups because there are no (or a limited number of) identical layers.
- **TreeMTL.** This is the state-of-the-art multi-task learning (MTL) approach [77] that automatically generates efficient multi-task models with high accuracy. MTL requires that each input has task labels of all the tasks. However, since there is no such large-scale dataset for benchmarks B2-B7, we use the distillation-based fine-tuning method in GMorph to train the multi-task model recommended by TreeMTL. The number of epochs and learning rate in the training process are the same as those in GMorph.
- **GMorph.** We consider three variants of GMorph, to evaluate the performance of components in the framework. 1) **GMorph**: the basic GMorph consisting of only the Simulated Annealing-based sampling policy. 2) **GMorph w P**: the GMorph equipped with predictive early termination. 3) **GMorph w P+R**: the GMorph equipped with both rule-based filtering and predictive early termination.

Inference Engines and Hardware. GMorph is implemented in *PyTorch* [52]. By default, we use *PyTorch* to fine-tune models and evaluate inference. We also compile and evaluate the baselines and the multi-task models using *TensorRT* [64], a production DNN compiler using graph optimizations such as operator fusion and CUDA multi-stream execution. These experiments aim to show that our model fusion techniques are complementary to existing compiler optimizations. All the experiments are conducted and evaluated on an NVIDIA Quadro RTX 8000 GPU.

6.2 Inference Time Reduction with Model Fusion

Model fusion using GMorph can substantially reduce the inference latency. Figure 7 reports the speedups obtained using the three variants of GMorph compared to the original multi-DNNs using PyTorch. More detailed experimental results and the visualizations of the multi-task models from the benchmarks are reported in Appendix B. Overall, without any accuracy drop, GMorph reduces the inference latency of the original models by 1.11-2.23× using PyTorch for all benchmarks. When 1% − 2% accuracy drop is allowed, GMorph further reduces the inference latency by up to 3.06×.

In *B1*, GMorph achieves 1.56× speedups without any accuracy drop because GMorph discovers a multi-task model that

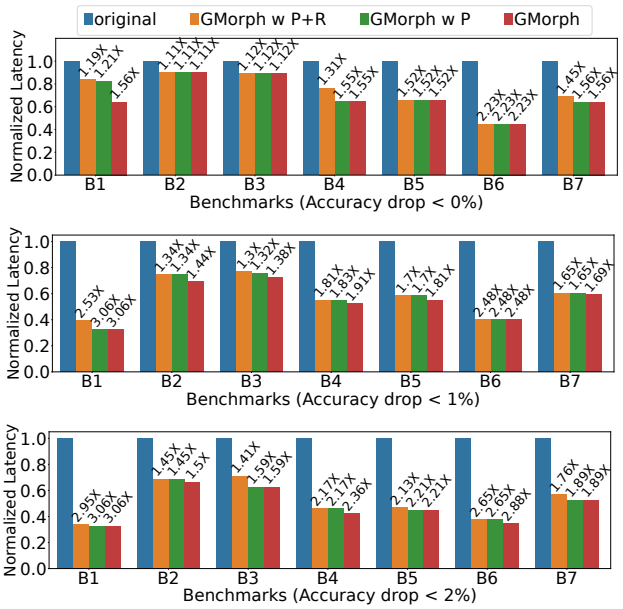


Figure 7. Normalized Latency and speedup (×) of GMorph on PyTorch compared to baselines in each benchmark (B1-B7). The numbers on each bar are the speedup.

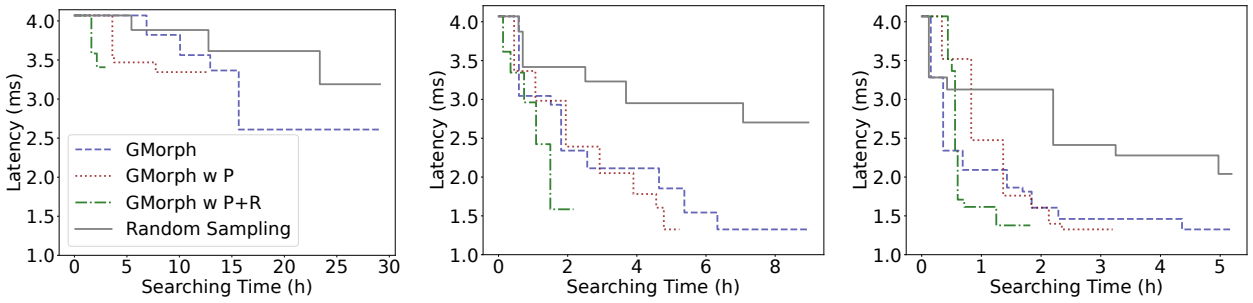
allows only two of the three tasks (GenderNet and EthinicityNet) to share features from shallow layers. When 1 − 2% accuracy drop is allowed, GMorph achieves 3.06× speedups. This is because GMorph not only discovers that it is possible to share features from the last layer across all tasks, but it also reduces the total number of layers using in-branch mutation, which further reduces computation costs.

In *B2*, GMorph found that sharing features from shallow layers across the three tasks ensures high task accuracy and thus results in 1.11× speedup without accuracy drop. Within 2% accuracy drop, GMorph gives 1.5× speedups mostly because of feature sharing between AgeNet and GenderNet.

B3, *B4* and *B5* have DNNs with different architectures. When allowing 1% accuracy drop, *GMorph* is able to reduce the latency by up to 1.38×, 1.91×, and 1.81×, respectively. The reason for this is that GMorph selects the appropriate layers from each task to form a common backbone that can be shared across various tasks, which results in large speedups without violating the accuracy requirements.

In *B6*, GMorph achieves up to 2.23× speedup without accuracy drop. This is because some layers from ViT_{Base} are shared between the two tasks and the total number of layers in $\text{ViT}_{\text{Large}}$ is reduced. When 2% accuracy drop is allowed, GMorph can achieve 2.88× speedup since some shared layers are further removed due to the in-branch mutation. In *B7*, GMorph gives 1.56× speedup without accuracy drop by sharing all the layers from $\text{BERT}_{\text{Large}}$, while gives 1.89× speedup within 2% accuracy drop via in-branch mutation.

GMorph w P+R and *GMorph w P* produce multi-task models with slightly lower speedups in some benchmarks compared



(a) Accuracy drop < 0% (b) Accuracy drop < 1% (c) Accuracy drop < 2%

Figure 8. Inference latency of multi-task models from GMorph variants during search. Results use B1.

Benchmark	PyTorch			TensorRT		
	Original	GMorph	**Speedup**	Original	GMorph	**Speedup**
B1	4.07	1.33	**3.06**	1.27	0.48	**2.65**
B2	13.04	8.68	**1.5**	8.88	5.89	**1.51**
B3	10.47	6.58	**1.59**	7.38	5.05	**1.46**
B4	9.36	3.97	**2.36**	3.44	1.45	**2.37**
B5	9.46	4.28	**2.21**	4.87	1.69	**2.88**
B6	52.81	18.36	**2.88**	23.21	7.14	**3.25**
B7	55.45	29.34	**1.89**	14.22	6.93	**2.05**

Table 3. Latency (ms) and speedup (×) comparisons between GMorph and Original models compiled on PyTorch and TensorRT, with accuracy drop < 2%.

Benchmark	All-shared		TreeMTL		GMorph	
	Acc Drop	**Speedup**	Acc Drop	**Speedup**	Acc Drop	**Speedup**
B1	0.87%	2.31	0.87%	2.31	1%	**3.06**
B2	12.55%	2.16	2.79%	**1.56**	1%	1.44
B3	0.96%	1.16	0.81%	1.08	1%	**1.38**
B4	1.72%	1.08	1.72%	1.08	1%	**1.91**
B5	-	-	-	-	1%	**1.81**
B6	-	-	-	-	1%	**2.48**
B7	-	-	-	-	1%	**1.69**

Table 4. Comparisons of accuracy drop and speedup (×) between multi-task learning baselines and GMorph.

to GMorph because the predictive filter mechanisms could filter out some promising candidates and lead to sub-optimal multi-task models. However, as we will discuss in Section 6.5, the predictive filter mechanisms can significantly reduce the search time cost.

Combination with DNN Compilers. We also report the latency and speedups using the TensorRT inference engine. Table 3 compares the original task-specific DNN models with the multi-task DNN produced by GMorph on both PyTorch and TensorRT with less than 2% accuracy drop. GMorph achieves speedups not only on PyTorch (1.5-3.06×) but also on TensorRT (1.46-3.25×). These results demonstrate that the model fusion techniques of GMorph can complement existing graph optimizations in production DNN compilers.

6.3 Model Fusion vs. Multi-Task Learning

We now show that the multi-task models produced by GMorph by fusing separate task-specific DNNs can outperform models obtained with multi-task learning (MTL). Table 4 shows the comparison of accuracy drop and latency speedup between the MTL baselines and GMorph. Note that the comparison of the accuracy drop favors the baselines, as the accuracy drop for *All-shared* and *treeMTL* are collected after models are trained to converge. In GMorph, fine-tuning stops once the user-specified accuracy target (e.g., 1%) is met, but accuracy could be even higher if the fine-tuning continues.

Overall, GMorph achieves similar or higher speedups with less accuracy drop compared to MTL baselines. In *B1*, both the baseline and GMorph discovered that all three tasks can share the entire backbone within 1% accuracy drop. The higher speedup from GMorph is because of the in-branch mutation that further reduces the computation costs of the shared backbone. In *B2*, *treeMTL* shares features from deeper layers than GMorph's model but suffers from a significant accuracy drop (e.g., 2.79%) due to over-sharing.

In *B3* and *B4*, where tasks use DNNs with different backbone architectures, performance improvements from MTL approaches are limited to 1.08-1.16×. It is due to the fundamental limitation of MTL which allows two DNNs to share only their common parts. Since the common part between VGG-16 and VGG-11/13 in *B3* has only one layer (the first convolution layer of each model) and that of ResNet-34 and ResNet-18 has only 5 convolution layers, sharing these common layers brings only limited speedups. In contrast, GMorph allows feature sharing in deeper layers, leading to much higher speedups (i.e., 1.38-1.91×).

In *B5/6/7*, where tasks use models with entirely different backbones or hidden sizes, MTL methods are not able to find feature-sharing opportunities across tasks. In comparison, GMorph demonstrates significant speedups (i.e., 1.69-2.48× within 1% accuracy drop).

Benchmark	Accuracy Drop <0%					Accuracy Drop <1%					Accuracy Drop <2%				
	GMorph	GMorph w P		GMorph w P+R		GMorph	GMorph w P		GMorph w P+R		GMorph	GMorph w P		GMorph w P+R	
	ST	ST	**saving**	ST	**saving**	ST	ST	**saving**	ST	**saving**	ST	ST	**saving**	ST	**saving**
B1	29.11	12.69	56%	2.96	90%	8.96	5.23	42%	2.17	76%	5.19	3.2	38%	1.82	65%
B2	386.37	289.3	25%	95.32	75%	337.94	278.27	18%	115.39	66%	313.31	224.43	28%	103.3	67%
B3	363.36	221.57	39%	114.59	68%	352	244.69	30%	115.2	67%	288.31	220.48	24%	93.93	67%
B4	211.59	120.61	43%	28.93	86%	208.25	118.68	43%	57.48	72%	192.26	119.17	38%	51.96	73%
B5	233.94	130.54	44%	34.33	85%	214.37	117.47	45%	46.3	78%	182.86	115.22	37%	38.81	79%
B6	285.28	131.29	54%	53.7	81%	255.55	123.94	52%	48.1	81%	251.34	119.87	52%	43.1	83%
B7	171.77	91.66	47%	37.88	78%	161.19	90.05	44%	36.41	77%	159.1	86.79	45%	32.67	79%

Table 5. Search time (ST) in hours and search time savings of GMorph with approaches of predictive filtering. P - Predictive early termination. R - Rule-based filtering.

6.4 Effectiveness of the Sampling Policy

The sampling policy of GMorph explores the search space of fused models by generating new mutated abstract graphs from previous promising candidates (i.e., the elite candidates). This helps the search process converge faster. We show this by comparing different variants of GMorph with a random sampling policy that builds candidates by sharing random features from the original DNNs, as discussed in Section 2.1.

Figure 8 demonstrates that the sampling policy gradually finds better candidates with lower inference latency during the search for all GMorph variants. In comparison, random sampling converges slower and yields less optimal multi-task models given a fixed search budget. Specifically, random sampling takes 31.6 hours, 21.5 hours, and 11.2 hours to finish 200 iterations for accuracy drop threshold of 0%, 1%, and 2% respectively, resulting in 1.56×, 2.75×, and 2.75× speedups, which are worse than GMorph's best speedup (3.06×). As Figure 2 in Section 2.1 shows, random sampling tends to generate candidates with statistically less aggressive feature sharing. Random sampling also requires more time to fine-tune each candidate, since candidates do not inherit the weight of previous elite candidates and show less aggressive computation reuse.

6.5 Effectiveness of Predictive Filtering

We now show that predictive filtering can reduce the search time of model fusion. Table 5 compares the search time between the basic version of GMorph (*GMorph*), GMorph with predictive early termination (*GMorph w P*), and GMorph with both predictive early termination and rule-based filtering (*GMorph w P+R*). Overall, predictive early termination (*GMorph w P*) reduces the search time by up to 28%-56% for all benchmarks. It can typically terminate the fine-tuning of non-promising candidates after a few epochs based on their test accuracy. Note that we measure the test accuracy every 5 epochs for predictive early termination in *B1-B5* while every 2 epochs in *B6-B7*. The reduction of searching time could be larger by setting a smaller evaluation interval. *GMorph w P+R* combines predictive early termination and rule-based filtering, saving up to 68%-90% of the search time. Rule-based filtering can further eliminate non-promising multi-task models before fine-tuning, and concentrate the fine-tuning time on more promising models.

Figure 8 further shows how, as the search progresses, the inference latency of best multi-task models found so far for *B1* decreases. Thanks to predictive filtering, *GMorph w P* and *GMorph w P+R* can converge much faster. Especially, given a limited amount of search time (e.g., 2 or 3 hours), variants of GMorph using those techniques can find better candidates.

7 Discussion

The Applicability of GMorph. There are two main application scenarios where the benefits of GMorph can outweigh the expenses associated with fine-tuning and candidate searching. First, in performance-critical applications (e.g., robotics [30] and extended reality [38]) running on resource-limited devices, the speedups from GMorph can be one of the necessary optimization steps during DNN deployment to ensure the delivery of real-time responses. Second, GMorph can be applied to optimize multi-DNNs in model serving systems to improve serving throughput, which is measured as queries per second. By paying the one-time cost of model searching and fine-tuning offline, GMorph can fuse multi-DNNs into a resource-efficient multi-task model to achieve lower latency and thus improve the throughput during online model serving.

Reducing Search Time. In the current GMorph prototype, model search and fine-tuning are executed sequentially on a single GPU. We can reduce search time by leveraging PyTorch's distributed training library. We can also distribute the training workload across multiple GPUs to reduce the training time. In addition, our current implementation samples only one multi-task model at a time, which limits the efficiency of the iterative process. We can accelerate this process by sampling multiple models in parallel or adopting parallel simulated annealing algorithms [53].

8 Related Work

Graph Optimization. Graph optimization is crucial in deep learning compilers [43], with common techniques like operator fusion, graph substitutions, layout transformation, etc. TensorRT [64] and TVM [8] use rule-based strategies

for operator fusion while DNNFusion [50] follows a more principled approach to determine beneficial fusion patterns. MetaFlow [34] allows users to define functionally-equivalent graph substitutions, TASO [33] automates the generation of graph substitutions and provides correctness verification support, and PET [67] identifies optimization opportunities via partially equivalent transformation. Unlike existing graph optimizations which focus on graph rewriting and increasing parallelism, GMorph explores a complementary direction where intermediate features across well-trained single-task models can be shared to directly reduce computation.

Multi-DNN Inference. Multi-DNN inference focuses on efficiently running mixed DNN workloads [73] by maximizing hardware utilization (GPUs, CPU, Mobile, etc) and avoiding contentions to reduce the execution time. Studies have explored parallelism for concurrent execution through graph-level space and time sharing [15, 31, 32, 51] or resource scheduling [1, 10, 12, 19, 47, 70–72]. Unlike GMorph, these approaches do not reduce the resource demands of multi-DNNs via feature sharing.

Multi-Task Learning. Multi-task learning (MTL) [7], a sub-field in machine learning, focuses on sharing parameters of a backbone model across tasks to improve accuracy. However, it is insufficient for our problem as it assumes that tasks use the same DNN architecture and that input data have all task labels. MTL uses hard or soft parameter sharing [2, 6, 22, 48, 54, 55, 63, 78], with the former sharing a set of parameters in the backbone model among tasks and the latter sharing the task information by enforcing the similarity of the model weights for each task. Various approaches have been proposed for multi-task model design, including manual design and task grouping [9, 20, 39, 44, 46, 49, 58, 60, 62], and NAS-based approaches [4, 21, 26, 59, 69, 76, 77]. In contrast, this work deals with how to share features across tasks given a set of well-trained DNNs with diverse architectures and weights, and separate training data for each task. Besides, automating model fusion at the system level, as done by GMorph, has the advantage of being fully automated and friendly to non-ML experts.

9 Conclusion

This work proposed model fusion, a novel approach to accelerate multiple pre-trained task-specific DNNs execution by sharing features across them without sacrificing accuracy. We implemented model fusion in a software framework, called GMorph, that iteratively samples multi-task models and evaluates their efficiency and accuracy, using a simulated annealing-based sampling policy and predictive filtering to lower search costs. Evaluations on seven benchmarks show that GMorph reduces the inference latency of DNNs by up to 3× while meeting target task accuracy goals. It also shows that model fusion can outperform multi-task learning, and is complementary to DNN compilation optimizations.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. CNS-2312396, CNS-2338512, CNS-2224054, and DMS-2220211. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Part of the work is also supported by Adobe gift funding.

References

[1] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. A multi-neural network acceleration architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 940–953, 2020.

[2] Jonathan Baxter. A model of inductive bias learning. *Journal of Artificial Intelligence Research*, 12:149–198, 2000.

[3] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

[4] David Bruggemann, Menelaos Kanakis, Stamatios Georgoulis, and Luc Van Gool. Automated search for resource-efficient branched multi-task networks. *arXiv preprint arXiv:2008.10292*, 2020.

[5] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[6] Rich Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, 1997.

[7] Richard Caruana. Multitask learning: A knowledge-based source of inductive bias. In *Proceedings of the 10th International Conference on Machine Learning*, pages 41–48. Morgan Kaufmann, 1993.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[9] Sumanth Chennupati, Ganesh Sistu, Senthil Yogamani, and Samir A Rawashdeh. Multinet++: Multi-stream feature aggregation and geometric loss strategy for multi-task learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.

[10] Y. Choi and M. Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 220–233. IEEE Computer Society, 2020.

[11] Tejalal Choudhary, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, 53:5113–5155, 2020.

[12] Bart Cox, Robert Birke, and Lydia Y. Chen. Memory-aware and context-aware multi-dnn inference on the edge. *Pervasive and Mobile Computing*, 83:101594, 2022.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[14] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM computing surveys (CSUR)*, 27(3):326–327, 1995.

[15] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. IOS: Inter-Operator Scheduler for CNN Acceleration. In *Proceedings of Machine Learning and Systems*, volume 3, 2021.

[16] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for

image recognition at scale. In *International Conference on Learning Representations*, 2021.

[17] Eran Eidinger, Roee Enbar, and Tal Hassner. Age and gender estimation of unfiltered faces. *IEEE Transactions on Information Forensics and Security*, 9(12):2170–2179, 2014.

[18] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88:303–308, September 2009. Printed version publication date: June 2010.

[19] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 115–127, 2018.

[20] Chris Fifty, Ehsan Amid, Zhe Zhao, Tianhe Yu, Rohan Anil, and Chelsea Finn. Efficiently identifying task groupings for multi-task learning. *Advances in Neural Information Processing Systems*, 34:27503–27516, 2021.

[21] Yuan Gao, Haoping Bai, Zequn Jie, Jiayi Ma, Kui Jia, and Wei Liu. 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, 2020.

[22] Yuan Gao, Jiayi Ma, Mingbo Zhao, Wei Liu, and Alan L Yuille. 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, 2019.

[23] Ian J. Goodfellow, Dumitru Erhan, Pierre Luc Carrier, Aaron Courville, Mehdi Mirza, Ben Hamner, Will Cukierski, Yichuan Tang, David Thaler, Dong-Hyun Lee, Yingbo Zhou, Chetan Ramaiah, Fangxiang Feng, Ruifan Li, Xiaojie Wang, Dimitris Athanasakis, John Shawe-Taylor, Maxim Milakov, John Park, Radu Ionescu, Marius Popescu, Cristian Grozea, James Bergstra, Jingjing Xie, Lukasz Romaszko, Bing Xu, Zhang Chuang, and Yoshua Bengio. Challenges in representation learning: A report on three machine learning contests. *Neural Networks*, 64:59–63, 2015. Special Issue on "Deep Learning of Representations".

[24] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129:1789–1819, 2021.

[25] Hui Guan, Xipeng Shen, and Seung-Hwan Lim. Wootz: A compiler-based framework for fast cnn pruning via composability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 717–730, 2019.

[26] Pengsheng Guo, Chen-Yu Lee, and Daniel Ulbricht. Learning to branch for multi-task learning. In *International Conference on Machine Learning*, pages 3854–3863. PMLR, 2020.

[27] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[29] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Dark knowledge. *Presented as the keynote in BayLearn*, 2(2), 2014.

[30] Hochul Hwang, Tim Xia, Ibrahima Keita, Ken Suzuki, Joydeep Biswas, Sunghoon I Lee, and Donghyun Kim. System configuration and navigation of a guide dog robot: Toward animal guide dog-level guiding work. *arXiv preprint arXiv:2210.13368*, 2022.

[31] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehana Durrani, Alexey Tumanov, Joseph E. Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference. *ArXiv*, abs/1901.00041, 2019.

[32] Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, Yunseong Lee, and Byung-Gon Chun. Accelerating multi-model inference by merging dnns of different weights, 2020.

[33] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[34] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. *Proceedings of Machine Learning and Systems*, 1:27–39, 2019.

[35] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[36] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[37] Arik Kurniawati, Ari Kusumaningsih, and Imam Hasan. Class vr: Learning class environment for special educational needs using virtual reality games. *2019 International Conference on Computer Engineering, Network, and Intelligent Multimedia (CENIM)*, pages 1–5, 2019.

[38] Hyoukjun Kwon, Krishnakumar Nair, Jamin Seo, Jason Yik, Debabrata Mohapatra, Dongyuan Zhan, Jinook Song, Peter Capak, Peizhao Zhang, Peter Vajda, et al. Xrbench: An extended reality (xr) machine learning benchmark suite for the metaverse. *arXiv preprint arXiv:2211.08675*, 2022.

[39] Isabelle Leang, Ganesh Sistu, Fabian Bürger, Andrei Bursuc, and Senthil Yogamani. 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, 2020.

[40] Seulki Lee and Shahriar Nirjon. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 175–190, 2020.

[41] Gil Levi and Tal Hassncer. Age and gender classification using convolutional neural networks. In *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 34–42, 2015.

[42] Gil Levi and Tal Hassner. Emotion recognition in the wild via convolutional neural networks and mapped binary patterns. In *Proc. ACM International Conference on Multimodal Interaction (ICMI)*, November 2015.

[43] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[44] Wei-Hong Li and Hakan Bilen. Knowledge distillation for multi-task learning. In *European Conference on Computer Vision Workshop*, pages 163–176. Springer, 2020.

[45] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4487–4496, Florence, Italy, July 2019. Association for Computational Linguistics.

[46] Mingsheng Long, Zhangjie Cao, Jianmin Wang, and S Yu Philip. Learning multiple tasks with multilinear relationship networks. In *Advances in Neural Information Processing Systems*, pages 1594–1603, 2017.

[47] Akhil Mathur, Nicholas D Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 68–81, 2017.

[48] Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3994–4003, 2016.

[49] Vladimir Nekrasov, Thanuja Dharmasiri, Andrew Spek, Tom Drummond, Chunhua Shen, and Ian Reid. Real-time joint semantic segmentation and depth estimation using asymmetric annotations. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7101–7107. IEEE, 2019.

[50] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[51] NVIDIA. Nvidia Multi Process Service. https://docs.nvidia.com/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020.

[52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[53] D Janaki Ram, TH Sreenivas, and K Ganapathy Subramaniam. Parallel simulated annealing algorithms. *Journal of parallel and distributed computing*, 37(2):207–212, 1996.

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

[55] Sebastian Ruder, Joachim Bingel, Isabelle Augenstein, and Anders Søgaard. Latent multi-task architecture learning, 2018.

[56] Jonathan R Senning. Computing and estimating the rate of convergence, 2007.

[57] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[58] Trevor Standley, Amir Zamir, Dawn Chen, Leonidas Guibas, Jitendra Malik, and Silvio Savarese. Which tasks should be learned together in multi-task learning? In *Proceedings of the International Conference on Machine Learning*, pages 9120–9132. PMLR, 2020.

[59] Ximeng Sun, Rameswar Panda, Rogerio Feris, and Kate Saenko. Adashare: Learning what to share for efficient deep multi-task learning. *arXiv preprint arXiv:1911.12423*, 2019.

[60] Mihai Suteu and Yike Guo. Regularizing deep multi-task networks using orthogonal gradients. *arXiv preprint arXiv:1912.06844*, 2019.

[61] Peter JM Van Laarhoven and Emile HL Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[62] Simon Vandenhende, Stamatios Georgoulis, Bert De Brabandere, and Luc Van Gool. Branched multi-task networks: deciding what layers to share. *arXiv preprint arXiv:1904.02920*, 2019.

[63] Simon Vandenhende, Stamatios Georgoulis, Wouter Van Gansbeke, Marc Proesmans, Dengxin Dai, and Luc Van Gool. Multi-task learning for dense prediction tasks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 2021.

[64] Han Vanholder. Efficient inference with tensorrt. In *GPU Technology Conference*, volume 1, page 2, 2016.

[65] Andreas Veit, Michael J Wilber, and Serge Belongie. Residual networks behave like ensembles of relatively shallow networks. *Advances in Neural Information Processing Systems*, 29:550–558, 2016.

[66] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics.

[67] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021.

[68] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[69] Bichen Wu, Chaojian Li, Hang Zhang, Xiaoliang Dai, Peizhao Zhang, Matthew Yu, Jialiang Wang, Yingyan Lin, and Peter Vajda. Fbnetv5: Neural architecture search for multiple tasks in one run. *arXiv preprint arXiv:2111.10007*, 2021.

[70] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405, 2019.

[71] Juheon Yi and Youngki Lee. Heimdall: Mobile gpu coordination platform for augmented reality applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20. Association for Computing Machinery, 2020.

[72] Fuxun Yu, Shawn Bray, Di Wang, Longfei Shangguan, Xulong Tang, Chenchen Liu, and Xiang Chen. Automated runtime-aware scheduling for multi-tenant dnn inference on gpu. *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[73] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xiang Chen. A survey of multi-tenant deep learning inference on gpu, 2022.

[74] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE Access*, 8:58443–58469, 2020.

[75] Jianming Zhang, Shuga Ma, Mehrnoosh Sameki, Stan Sclaroff, Margrit Betke, Zhe Lin, Xiaohui Shen, Brian Price, and Radomír Měch. Salient object subitizing. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[76] Lijun Zhang, Xiao Liu, and Hui Guan. Automtl: A programming framework for automated multi-task learning. *arXiv preprint arXiv:2110.13076*, 2021.

[77] Lijun Zhang, Xiao Liu, and Hui Guan. A tree-structured multi-task model recommender. *arXiv preprint arXiv:2203.05092*, 2022.

[78] Yu Zhang and Qiang Yang. A survey on multi-task learning. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[79] Zhifei Zhang, Yang Song, and Hairong Qi. Age progression/regression by conditional adversarial autoencoder. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2017.

A Detailed Experiment Settings

In this section, we report the detailed experiment settings omitted from Section 6.1 due to space constraints. The tasks we adopt for the benchmarks and the corresponding scores are listed in Table 6.

Tasks and datasets. **B1** uses the *UTKFace* [79] dataset, which consists of over 20K images with annotations of age, gender, and ethnicity. **B2** and **B3** use the *FER2013* [23] dataset for EmotionNet and the *Adience* [17] dataset for AgeNet and GenderNet. *FER2013* contains approximately 30K facial RGB images of 7 different types of expressions. *Adience* contains about 26K photos of the real world with labels of age and gender. **B4/5/6** use *PASCAL VOC2007* [18] for ObjectNet and *SOS* [75] for SalientNet. *VOC2007* consists of 9963 images containing over 24k annotated objects of 20 classes. *SOS* contains about 7K images, where each image is annotated with the number of salient objects in the image. **B7** uses *CoLA* and *SST-2* datasets from the GLUE benchmark[66], which have about 10k and 70k sentences respectively with labels of binary classes.

For B1-B3, the score is the number of instances classified correctly (i.e., classification accuracy), while for B4-B6, the score is Mean Average Precision. For B7 which is a General Language Understanding task, the score for CoLANet is Matthews Correlation Coefficient, while for SSTNet the score is classification accuracy.

Benchmark	Models	Datasets	Scores
B1	AgeNet: VGG-13	UTKFace	0.495
	GenderNet: VGG-13		0.895
	EthnicityNet: VGG-13		0.763
B2	EmotionNet: VGG-16	FER2013	0.698
	AgeNet: VGG-16	Adience	0.662
	GenderNet: VGG-16		0.791
B3	EmotionNet: VGG-13	FER2013	0.681
	AgeNet: VGG-16	Adience	0.662
	GenderNet: VGG-11		0.760
B4	ObjectNet: ResNet-34	VOC2007	0.884
	SalientNet: ResNet-18	SOS	0.701
B5	ObjectNet: ResNet-34	VOC2007	0.884
	SalientNet: VGG-16	SOS	0.693
B6	ObjectNet: $\text{ViT}_{\text{Large}}$	VOC2007	0.894
	SalientNet: ViT_{Base}	SOS	0.766
B7	CoLANet: $\text{BERT}_{\text{Large}}$	CoLA	0.603
	SSTNet: $\text{BERT}_{\text{Base}}$	SST-2	0.917

Table 6. Models and datasets.

Optimization Parameters. We report the learning rate setting for training single-task DNNs in Table 6 as follows. In B1, the learning rate lr to train all task-specific models and to fine-tune multi-task models is 0.0005. In B2, lr is 0.001 for all models. In B3, lr is 0.001 for EmotionNet and 0.005 for the others, so GMorph set $lr = 0.001$. In B4 and B5, as lr are 0.001 and 0.0001 for ObjectNet and SalientNet, so GMorph sets $lr = 0.0001$. In B6 and B7, lr is 0.00005 for all models.

Methods	Accuracy Drop <0%						
	Original	GMorph		GMorph w P		GMorph w P+R	
	Latency	Latency	**speedup**	Latency	**speedup**	Latency	**speedup**
B1	4.07	2.61	1.56	3.36	1.21	3.42	1.19
B2	13.04	11.79	1.11	11.79	1.11	11.79	1.11
B3	10.47	9.32	1.12	9.32	1.12	9.38	1.12
B4	9.36	6.04	1.55	6.04	1.55	7.16	1.31
B5	9.46	6.23	1.52	6.23	1.52	6.23	1.52
B6	52.81	23.68	2.23	23.68	2.23	23.68	2.23
B7	55.45	35.47	1.56	35.47	1.56	38.24	1.45

Table 7. Latency (ms) and speedup (×) of GMorph with accuracy drop < 0%.

Methods	Accuracy Drop <1%						
	Original	GMorph		GMorph w P		GMorph w P+R	
	Latency	Latency	**speedup**	Latency	**speedup**	Latency	**speedup**
B1	4.07	1.33	3.06	1.33	3.06	1.61	2.53
B2	13.04	9.07	1.44	9.73	1.34	9.73	1.34
B3	10.47	7.58	1.38	7.94	1.32	8.07	1.3
B4	9.36	4.91	1.91	5.12	1.83	5.18	1.81
B5	9.46	5.23	1.81	5.57	1.7	5.57	1.7
B6	52.81	21.29	2.48	21.29	2.48	21.29	2.48
B7	55.45	32.87	1.69	33.61	1.65	33.61	1.65

Table 8. Latency (ms) and speedup (×) of GMorph with accuracy drop < 1%.

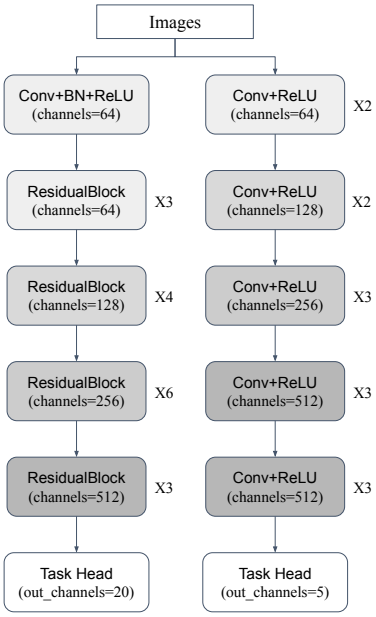
Methods	Accuracy Drop <2%						
	Original	GMorph		GMorph w P		GMorph w P+R	
	Latency	Latency	**speedup**	Latency	**speedup**	Latency	**speedup**
B1	4.07	1.33	3.06	1.33	3.06	1.38	2.95
B2	13.04	8.68	1.5	9	1.45	9	1.45
B3	10.47	6.58	1.59	6.58	1.59	7.4	1.41
B4	9.36	3.97	2.36	4.32	2.17	4.32	2.17
B5	9.46	4.28	2.21	4.28	2.21	4.49	2.13
B6	52.81	18.36	2.88	19.93	2.65	19.93	2.65
B7	55.45	29.34	1.89	29.34	1.89	31.55	1.76

Table 9. Latency (ms) and speedup (×) of GMorph with accuracy drop < 2%.

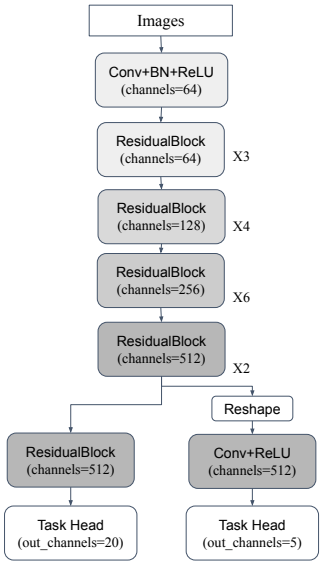
B Detailed Experimental Results

In this section, we report the latency of multi-task models generated from GMorph and single-task DNNs and visualize some of these models.

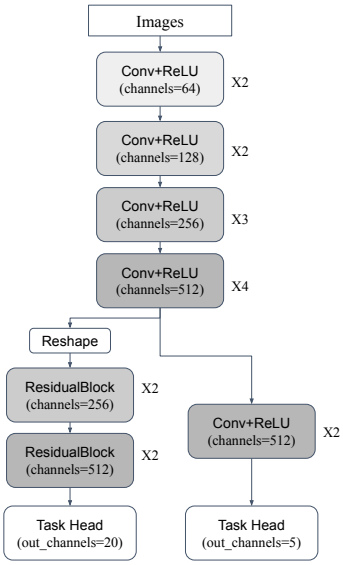
Latency results. Figure 7 presents the speedups obtained using three different versions of GMorph compared to the original multi-DNNs using Pytorch with the normalized latency. We present the corresponding latency in each experiment in Tables 7, 8 and 9.



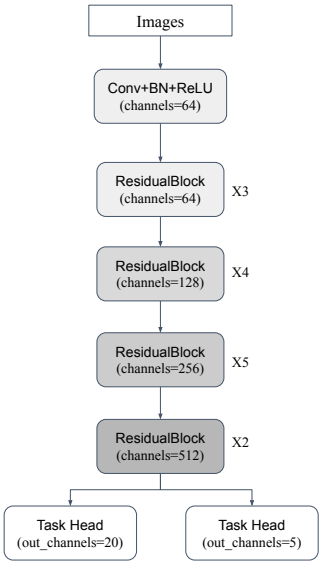
(a) Original multi-task model



(b) Mutated model 1



(c) Mutated model 2



(d) Mutated model 3

Figure 9. Visualization of examples of mutated models from benchmark-5 with accuracy drop < 1%.

Visualization of multi-task models. In Section 6.2, we analyze the speedups obtained by GMorph and the reason why the inference latency of the mutated multi-task models is reduced. In Figure 9, we present the visualizations of some examples of the multi-task models discovered by GMorph in the benchmark-5 for more straightforward comparisons. Figure 9(a) shows the architecture of the original model which consists of a ResNet-34 and a VGG-16, and Figure 9(b)-(d) show three mutated models discovered by GMorph within 1% accuracy drop. We omit operators such as maxpooling layers to make the visualization simpler. Model(b) and model(c) inherit layers and weights from ResNet and VGG respectively to form the common backbone but keep some independent layers for each task. Model(d) shares the entire backbone of ResNet while removing some layers via in-branch mutation, which provides more speedup and is selected as the best candidate (i.e., 1.81× as shown in Table 8).

A Artifact Appendix

A.1 Abstract

This artifact contains the codes of GMorph, the codes of all the benchmarks listed in Section 6.1 and Table 2, and the codes to evaluate all the benchmarks.

A.2 Description & Requirements

A.2.1 How to access. The artifact is public and available on GitHub: https://github.com/qizhengyang98/GMorph.git. A *readme* document is also included in the repository. Codes are under the *master* branch.

The artifact is also submitted to Zenodo with link: https://doi.org/10.5281/zenodo.10783786.

A.2.2 Hardware dependencies. CPU with 6 cores, 16G RAM, and a GPU (10G vRAM is needed for benchmark-1,4,5, 20G is recommended for benchmark-2,3,6,7).

A.2.3 Software dependencies. Linux OS, Python = 3.8, and several Python packages listed in the *requirements.txt*. A Conda environment is recommended for installation.

A.2.4 Benchmarks. For benchmark-1, the dataset and pre-trained single-task models are included in the repository. For benchmarks 2-6, datasets and pre-trained single-task models can be downloaded from this link: https://drive.google.com/drive/folders/1Dtvd5eIDeDiseCAwCrj3_wrqjWsy3bq3?usp=sharing. For benchmark-7, which has two GLUE tasks, the datasets will be downloaded automatically when this benchmark is executed for the first time. For benchmark-6 and -7, we use the Huggingface Transformers library [68].

A.3 Set-up

Setting up the environment takes three main steps.

Step 1: Clone the repository from GitHub via the link listed above.

Step 2: Download the datasets and models used for each benchmark and put them under the corresponding folders. You can simply run the script *prepare_ds_mod.sh* which prepares all datasets and models automatically. To do so, run the script after installing *gdown* package by:

- pip install gdown

Alternatively, you can do the following to prepare them manually:

- Put *datasets.zip* under *GMorph/* and unzip it. There should be four folders under *datasets*: *adience*, *ESOS*, *fer2013*, *VOCDetection*;
- Put *salientNet.model*, *salientNet_vgg16.model*, *objectNet.model* under */test_metamorph/scene/pre_models*. Make the directory if it does not exist;
- Put *EmotionNet.model*, *EmotionNet_vgg13.model*, *ageNet.model*, *genderNet.model*, *genderNet_vgg11.model* under */test_metamorph/face/pre_models*;
- Put *age_gender.gz* under *metamorph/data*;
- Put *toy_vgg13.pt* under *metamorph/model*;
- Put *cola.zip*, *sst2.zip*, *multiclass.zip*, *salient.zip* under */test_metamorph/transformer_model* and unzip them.

Step 3: Under the *GMorph* folder, set up the conda environment:

- conda create -n gmorph python=3.8
- conda activate gmorph
- pip install -r requirements.txt
- cd test_metamorph/transformers/
- pip install -e .
- cd ../..
- pip install metamorph/

With the above three steps, all the necessary dependencies should be installed. To do a simple test, go to the folder *metamorph/test* and run

- python test.py

If the computation graph of models is printed successfully, then the set-up is done.

A.4 Evaluation Workflow

A.4.1 Major claims.

- *(C1): GMorph can substantially reduce the inference latency of models with different architectures with minor accuracy drops. This is proven by the experiment (E1) in Section 6.2 (Figure 7), and also experiment (E2) in Section 6.3 (Table 4).*
- *(C2): Predictive filtering can reduce the search time of model fusion. This is proven by the experiment (E1) in Section 6.5 (Figure 8 and Table 5).*
- *(C3): The model fusion techniques of GMorph can complement existing graph optimizations in production DNN compilers like TensorRT. This is proven by the experiment (E3) in Section 6.2 (Table 3).*

A.4.2 Experiments. In this section, we provide instructions on how to execute experiments using the scripts in the artifact.

Experiment (E1): [5 human-minutes + various compute-hour for different benchmarks]: Run GMorph for different benchmarks and generate well-trained multi-task models. The estimated search time (compute-hour) for each benchmark is reported in Table 5.

[Preparation] Under the *GMorph* folder, there are several shell scripts named *submit_xxx.sh*, which are used to evaluate different benchmarks in this experiment. We will execute the shell scripts with proper arguments. The script *figure7table5.sh* is used to reproduce the results in Figure 7, and the script *figure8.sh* is used to reproduce the results in Figure 8 and Table 5.

Under the *benchmark_scripts* folder, there are also separate scripts provided to run all the experiments for each benchmark without manually changing arguments, and which script corresponding to which experiment is written in the script *figure7table5.sh*.

We explain the meaning of each configuration/argument here:

- *policy_select:* set *SimulatedAnnealing* when testing GMorph, set *LCBased* when testing GMorph w P and GMorph w P+R.
- *log_name:* the name of the log file, which saves useful intermediate information when GMorph is running.
- *acc_drop_thres:* the threshold of accuracy drop. Can be set to 0, 0.01, or 0.02 to test the results shown in Section 6.2 and Figure 7.
- *enable_filtering_rules:* whether or not to enable rule-based filtering. Add this flag when testing GMorph w P+R, and remove this flag when testing GMorph w P. This flag is useful only when *policy_select=LCBased*.

Other arguments and flags do not need to be changed during evaluations. Note that the arguments of *batch_size* and *num_workers* can be smaller if GPU memory is not enough.

[Execution] For benchmark-1, run *submit_b1.sh* or the commands in this file. Modify the flags or arguments listed above when doing different evaluations. Benchmark-1 with GMorph w P+R should take the least computation time, which can be tested first.

- *policy_select=SimulatedAnnealing*, *log_name=b1_SA_t0*, *acc_drop_thres*=0, *enable_filtering_rules* added or removed;
- *policy_select=LCBased*, *log_name=b1_LC_t001*, *acc_drop_thres*=0.01, *enable_filtering_rules* removed;
- *policy_select=LCBased*, *log_name=b1_LC_t002_R*, *acc_drop_thres*=0.02, *enable_filtering_rules* added;
- .....

For different benchmark-n, run *submit_bn.sh*, and modify the flags or arguments as shown above. To run experiments without manually changing flags or arguments, go to the *benchmark_scripts* directory and run corresponding scripts.

To reproduce results shown in Figure 7, 8 and Table 5, run scripts *figure7table5.sh* and *figure8.sh* accordingly. Note that running these scripts can be time-consuming, which basically runs all the experiments for all the benchmarks, so an alternative way is to run each experiment separately given the comments in the scripts.

Note that since GMorph uses simulated annealing, which introduces randomness, the outcomes from the model searching may be similar but not exactly the same between different runs. It would be better to run each benchmark multiple times to minimize the influence of randomness.

[Results] The shell script will create a log file under the directory: *GMorph/results/log*.

For each run of the script, there will be a total of 200 iterations, and in each iteration, a multi-task model will be generated and trained. The log records the architecture of the model, the accuracy and latency of the model, and the overall search time at the end of each iteration. These numbers will match the numbers in Figure 7 and Table 5. It is likely that the results look slightly different from the results reported in the paper due to the randomness. Evaluating each benchmark two or three times will minimize the impact of the randomness of the search algorithm.

The search process of GMorph can be time-consuming. To address the problem, we provide some of the logs we pre-generated in the *log/examples* folder. Those logs can also be used as references when new logs are generated.

Experiment (E2): [1 human-minute + 5 compute-minute]: Test the latency of all-shared multi-task models and multi-task models found by TreeMTL.

[Preparation] None

[Execution and Results] Simply run the shell script *table4.sh*, the latency of all-shared models and multi-task models found by TreeMTL in benchmark 1-4 will be printed. The latency of models generated by GMorph is reported in the generated logs of *E1*. It does not need to finish E1 in order to do E2.

Experiment (E3): [1 human-minute + 30 compute-minute]: Test the latency of multi-task models generated by GMorph on both PyTorch and TensorRT.

[Preparation] None

[Execution and Results] Run the shell script *table3.sh*, the model architectures found by GMorph will be compiled by both PyTorch and TensorRT automatically, and the results, which are the latency of the models, will be printed. 12G GPU memory is needed for benchmark-6 and 15G is needed for benchmark-7.