MTLSys: Latency-aware Pruning for Multi-task Learning

Abhigyan Khaund [*1] **Joseph Miano** [*1]

Abstract

We present MTLSys, which is a latency and Multi-task Learning (MTL) aware inference serving system for supervised machine learning. The core idea of MTLSys is to integrate MTL model variants and MTL-awareness into an inference serving system by using task-aware pruning. In this paper, we explain how we developed our MTLSys prototype, including model variant generation and inference serving logic. Furthermore, we characterize various aspects of the performance of our system, including: accuracy-latency trade-offs when compared to a single-task-system baseline, the proportion of SLO mishits in the MTL-aware system vs. the single-task system, and results around the impact of pruning and task-specific pruning on performance. Overall, we found that MTLSys is able to extend the Pareto frontier of accuracy and latency when compared to a single-task-only baseline. Furthermore, for multi-task requests including accuracy and latency constraints, we found that our system violated SLOs 1.6x less often when compared to the single-task baseline. We also found that task-aware pruning enabled MTL model variants to be specialized to single tasks and achieve single task outputs with over 20% decrease in latency on GPU inference for task heads with 15 layers when compared to complete, unpruned MTL models; the benefits of task-aware pruning appear to increase with the complexity of task-specific heads. Our system could be applied across a variety of domains, including natural language processing (NLP) and computer vision, where multi-task outputs (i.e., several outputs for the same input) are required.

Link to Team GitHub: https://github.gatech.edu/cs8803smr/repo_team14

1 Introduction

Machine Learning (ML) models are everywhere. There are a large number of applications in a breadth of industries that rely on results from ML models. The number of applications and industries is only expected to grow more. As more and more ML models are getting used to solve difficult tasks, the complexity and size of these models are also growing at a tremendous speed. These large models incur huge training and inference costs.

ML applications have diverse requirements. Applications can have queries that have different latency, cost, output tasks, accuracy for similar types of inputs. For example, for a simple face recognition model social media application can have higher accuracy requirements, while an intruder detection system can work with a lower accuracy threshold. However, an intruder detection system may need the results faster than a social media application. Some applications are required to produce different outputs for the same input data. For instance, in a self-driving car, the ML model is required to produce outputs such as segmentation, object-detection, classification on the same image snapshot at a point of time. This can be enabled using Multi-Task Learning (MTL) models (Ruder, 2017). It is also possible that applications do not require all the outputs all the time, which would require them to have separate models for those specific tasks or do unnecessary computations on an MTL model. Furthermore, a model could have several variants based on training hyper-parameters, quantization, etc., which may differ in size, memory, latency and accuracy. Single or multiple applications may have requirements to use any one of these variants for different tasks, hardware, or workloads.

These different applications requirements create a huge challenge for model-variant generation and inference serving systems. Serving systems like Clipper (Crankshaw et al., 2017) and TensorFlow Serving (Olston et al., 2017) are able to optimize single model serving without taking into account application requirements. Newer serving systems like like InferLine (Crankshaw et al., 2020) build on top of these to incorporate provisioning and planning of pipelines to navigate in the latency and cost search space. INFaaS (Romero et al., 2019) abstracts out model-variant generation and allows developer specify application queries with per-

[*]Equal contribution [1]College of Computing, Georgia Institute of Technology, Atlanta, GA. Correspondence to: Abhigyan Khaund <abhigyan@gatech.edu>, Joseph Miano <jmiano@gatech.edu>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2022.

formance and accuracy metrics. However, none of these systems satisfy the task specific requirements of an applications in a multi-task setting. To the best of our knowledge, no system exists today that automates model-variant generation specialized for multiple task requirements and navigates the search space of these model-variants for developers subject to specified tasks, cost, end-to-end latency performance, and accuracy metrics of applications across varied hardware configurations. Hence, the goal of our work is to incorporate task-awareness into inference serving systems satisfying latency-accuracy requirements.

We propose MTLSys – a latency-aware inference system for automated model-less multi-task learning. With MTLSys, we generate model-variants from a trained MTL model by pruning to meet different performance requirements for multiple tasks as well optimized for single tasks. MTLSys also consists of a serving system that navigates the generated model-variant search space to select and decide the best model that meets applications each inference query requirement consisting of output task, end-to-end latency, and minimum accuracy.

We apply MTLSys to push the Pareto curve for accuracy-latency tradeoff with constraints on output tasks. We generate 40 model-variants from an MTL model by pruning the MTL model down to smaller sizes and also by pruning down task heads to specialize them for specific tasks. We evaluate the performance of MTLSys against a baseline system consisting of model combinations trained for single tasks as well as MTL model systems which are not optimized to serve single tasks. Compared to these widely used existing inference systems, we achieve higher performance metrics across multiple and single task requirements and violate SLOs 1.6x less often for all-task queries.

The specific hypotheses we sought to test when evaluating our system are as follows:

1. Multi-task networks enable better accuracy-latency trade-offs for multi-task outputs than using several single-task networks.

2. Pruning a multi-task network down to single tasks will improve the accuracy-latency trade-off for those tasks compared to keeping all task heads.

3. Multi-task network encoders are more robust to the negative accuracy impacts of pruning when compared to single-task networks.

2 Background and Motivation

Sharing representations between related tasks to better generalize models is referred to as Multi-Task Learning. This is similar to how humans often apply the knowledge we have acquired to learn new and related tasks. From an application's perspective, it could require several similar task results from its data. For example, a social media application could be generating various results like face detection, age estimation, gender identifier, face count, etc., on an image. All these tasks are related, derived from the same image and could share many representations. These applications could use multi-task learning models (Caruana, 1997) to achieve all tasks at once. In the context of Deep learning, multi-task Learning is done either by hard parameter sharing or soft parameter sharing of hidden layers. Hard parameter sharing involves sharing a set of hidden layers for all task followed by having task-specific heads for each task. In soft parameter sharing, each task has its own model and parameters but are made similar to constraining some layers to be closer by regularization. Hard parameter sharing can thus give multiple outputs for different tasks on a single input, thus greatly reducing training and inference time for applications that require multiple outputs.

However, MTL models can be computationally heavy and may require much more training data (Zhang & Yang, 2017) when compared to their single-task counterparts. When an application requires only a single task in the query, it is unnecessary and expensive to run the entire MTL model. In MTLSys, we explore how we can transform multi-task models to gain the latency benefits of their single-task counterparts for single tasks. Our model-variant generation pipeline prunes off the unused task heads to reduce MTL models into multiple single-task models while maintaining the performance accuracy of the remaining task.

Apart from reducing MTL models to single task variants, it is also necessary for the model-generator to create other compressed model variants that be used by the serving system to meet varied latency deadlines. Such model variants can be generated by layer fusion, quantization (Wu et al., 2020), graph optimizers (Amazon SageMaker Neo, Tensor RT , (Vanholder, 2016)) or pruning (Molchanov et al., 2016). Pruning techniques have been shown to work on both MTL and single task models to compress models with low accuracy loss. Existing serving systems (Romero et al., 2019) use one or more of the above techniques to generate model-variants for single task models to achieve latency deadlines. However, as MTL models are more generalized due to parameter sharing and learning from related tasks, we hypothesize that they are more resilient to accuracy loss when pruned as compared to single task models. Hence, it would be much better to use compressed MTL model's single task variants to achieve higher accuracy on lower latency SLOs. This is a a large number of model-variants to generate, train and maintain across hardware profiles, and is a time-consuming task for developers. MTLSys involves generating these model-variants and profiles their performance on deployed hardware to be used by the serving system.

Having such a large number of model variants makes it hard for developers to specify which model-variant to use for different tasks. Existing inference systems are not task-aware and do not take into account the tasks as a parameter while working with a multi-task model and could end up being expensive and missing tight latency SLOs. A task-aware serving system is able to select and deploy the right variant for each prediction query, which could have different task requirements, ranging from a variety of single tasks to multiple tasks in one query; it is challenging for existing serving systems to navigate this search space. Automatically being able to serve such type of queries saves developer effort specify per query model-variant and meets more accuracy-latency SLOs with the task constraint and saves cost.

3 Related Work

3.1 Multi-task Learning

Multi-task learning (MTL) refers to using a shared representation to learn more than one task in parallel (Caruana, 1997). Within neural networks, specifically, there are two main methods to achieve MTL: hard parameter sharing and soft parameter sharing (Ruder, 2017). In our work, we focus on hard parameter sharing, which refers to having layers of shared weights followed by task-specific heads (Ruder, 2017), with one head for each task. Thus, the network needs to learn a shared representation across tasks in its shared layers, which can then be specialized for each task in the task-specific layers. Though MTL is similar to transfer learning in that performance from one task can help with performance of another task (Pan & Yang, 2009), in transfer learning, the tasks are learned sequentially. However, in the MTL setting, the network is trained for all its tasks at once using a loss function that incorporates all of the relevant tasks.

MTL has been shown to able to achieve model accuracy results on par with or greater than training individual single-task models across a variety of domains. Yang et al. (Yang et al., 2017) showed how a multi-task learning model could be jointly trained for segmentation and classification of skin lesions, and that training these tasks together improved both the segmentation and classification performance over individual models. Another common domain where MTL sees successful use is natural language processing (NLP); for example, Sogaard and Goldberg showed that by incorporating part-of-speech (POS) tagging as an MTL objective into the task of phrase-chunking, their models outperformed those that trained only for the phrase-chunking objective (Søgaard & Goldberg, 2016). The intuition behind why MTL training is able to outperform single-task training on single tasks is that related tasks enable models to learn features that can be useful for the other tasks, especially in low-data settings.

3.2 Pruning

Pruning is a technique whereby some weights are removed from a neural network (Blalock et al., 2020), with the goal often being to reduce latency or network storage size. Pruning is relevant to our project because we use pruning as the method to generate model variants across the accuracy-latency-tradeoff space. Two broad categories of pruning include structured and unstructured pruning, where structured pruning refers to pruning away individual parameters and structured pruning removes parameters in groups (Blalock et al., 2020); from the perspective of a convolutional network, structured pruning would refer to removing whole filters from convolutional layers.

One popularly-used technique for neural network pruning is iterative pruning, in which a network is trained, pruned, and then fine-tuned (Blalock et al., 2020); such a pruning scheme helps to gain the latency/size reduction benefits of pruning, whilst mitigating the accuracy losses associated with pruning away parameters. To determine which parameters should be pruned at each iteration of pruning, a score can be issued to each parameter, with the lowest-scoring parameters then being pruned (Blalock et al., 2020). One example of a simple scoring scheme is to use the parameter weights as the scores, and thus prune away some percentage of the parameters with the smallest weights.

Several papers have leveraged pruning techniques in MTL settings, including Chen et al. (Chen et al., 2020) and Dellinger et al. (Dellinger et al., 2021). Chen et al. focus on channel (i.e., structured) pruning of convolutional neural networks for segmentation and classification (Chen et al., 2020), and Dellinger et al. prune a network that can perform object detection, segmentation, and soiling detection (i.e., dirt and water on the camera) in the context of autonomous vehicles (Dellinger et al., 2021). However, neither of these papers perform task-specific pruning, whereby an unused task head would be pruned off when it is not being used.

3.3 Inference-serving Systems

One of the first commercial inference serving systems was TensorFlow Serving (Olston et al., 2017) which supported inference pipelines for models built with TensorFlow (Abadi et al., 2016). It provides a production environment to deploy TensorFlow models and support continuous training and serving of ML pipelines, but does not support any performance constraints. Clipper (Crankshaw et al., 2017) advances inference systems by generalizing support to other frameworks, using containers to individually manage different models. Several other systems like InferLine and Nexus (Shen et al., 2019) have been built on top of Clipper to optimize serving pipelines to achieve higher throughput and SLO attainments. InferLine (Crankshaw et al., 2020)

focuses on efficiently provisioning prediction pipelines to to meet latency SLOs for different loads of application queries. Clockwork (Gujarati et al., 2020) attempts to meet inference latency deadlines by ordering queries in the pipeline based on their SLOs and running them sequentially.

Some newer serving systems aim to meet both accuracy and latency requirements of application queries. (Halpern et al., 2019; Zhang et al., 2020) allows the switching of models with different performance metrics to trade accuracy off for latency. INFaaS extends this by generating models using optimizers to improve latency and resource usage, automatically navigating this model search space to meet accuracy and latency requirements. However, none of these systems are task-aware and do not meet task specific application requirements in multi-task model settings. MTLSys, on the other hand, makes decisions to choose a model from the variant search space optimum to meet the task, accuracy, and latency requirements of queries.

4 METHODS

Overall, we developed MTLSys in two main parts: model variant development and inference-serving system development. In order to measure the benefits of MTL-awareness, we built a version of our system that includes MTL models and a version that includes only single-task models.

4.1 Dataset

Although MTLSys is flexible to work with many different datasets and tasks for supervised learning and inference, we generated the prototype of our system by leveraging the UTKFace dataset (Zhang & Qi, 2017), which is a dataset of 23,708 cropped and aligned human faces with labeled age, gender, and ethnicity. The faces range from age 1 to 116, male and female gender, and 5 different ethnicities. Thus, this dataset serves as a good case-study from an MTL perspective because we can develop a shared encoder to understand faces, with task-specific heads to perform regression on age and classification on gender and ethnicity.

4.2 Model Development

In order to serve inference outputs in response to user input queries, MTLSys first trains, prunes, and fine-tunes several model variants to meet various task/latency/accuracy constraints. Each of these model variants is derived from a standardized parent model, which consists of a shared encoder and 3 task heads (1 for each task). For the encoder, we leverage a pretrained resnet18 from the PyTorch library (Paszke et al., 2019), which we then fine-tune during the variant generation process. Our task-specific heads are each 3-layer ReLU feed-forward neural networks, which take in the shared representation output by the encoder and produce task-specific outputs. One benefit of this structure is that the specific pretrained encoder (i.e., the resnet18) and the task-specific heads could be customized to meet the needs of the system end-user.

The model variant generation process we use in our system is consistent across the MTL and single-task versions. **In the MTL-version** of our system, we generate 40 model variants by training the full MTL-model for 10 epochs, pruning it by an amount between 0% and 90%, and then fine-tuning it on 10 more epochs. Of the 40 total model variants, 10 are optimized for MTL, 10 are optimized to predict age, 10 are optimized to predict gender, and 10 are optimized to predict ethnicity. For the MTL training, we use a linear combination of the losses for each task such that they are on a similar scale (specifically, a coefficient of 0.004 for the age MSE Loss, 2 for the gender Cross Entropy Loss, and 1 for the ethnicity Cross Entropy Loss). After the MTL pretraining, the model is pruned by one of the following amounts: {0%, 10%, 20%, 30%, ..., 90%}, which yields 10 variants according to their amount of pruning. Next, the pruned models are fine-tuned for their specific task(s), using either the same combined loss for the MTL models or the task-specific loss for the single-task variants. Another important component of our pruning process is that we fully prune off the unused task heads for the single-task models, which enables them to benefit from the MTL pretraining and task-specific fine-tuning, without suffering the inefficiency to latency that would come with keeping unnecessary task heads. **In the single-task version** of our system, we follow the same 3-step procedure of training, pruning, and fine-tuning, except we do not include the MTL pretraining or MTL models. In the single-task system, there are 30 model variants, 10 each for each of the 3 tasks, which are generated as follows: pretrain a task-specific model for 10 epochs, then prune by an amount in {0%, 10%, 20%, 30%, ..., 90%}, and then fine-tune to the same specific task for 10 more epochs.

Across both the MTL and single-task model variant development, we leveraged the Adam optimizer (Kingma & Ba, 2014) with the default PyTorch hyperparameters, including a learning rate of 0.001, which we found worked well for our tasks. Once generated, the model variants are evaluated to determine their performance characteristics (accuracy, task, and latency) and then saved to disk, to be loaded and used by the system for inference serving at run-time. The model variants were generated using a local Windows Desktop PC with an NVIDIA GeForce GTX 1080Ti GPU, 32GB of RAM, and an Intel Core i7-8700K CPU.

4.3 System Development

In this section we describe the inference system development of the MTLSys.

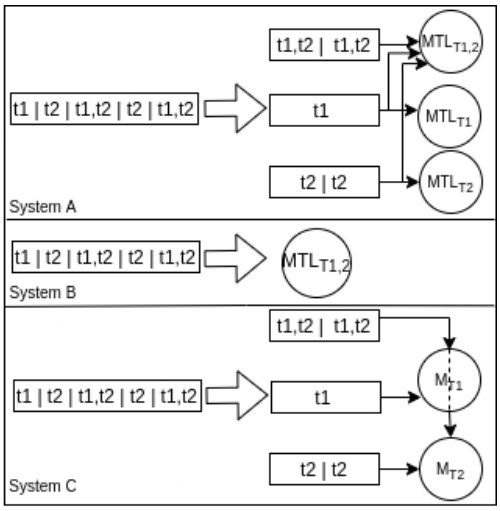

Figure 1. Different system types to handle multi-task queries {t1 ,t2}. System A 9is proposed MTLSys, System B is a traditional MTL model only system and System C is combination of single-task models.

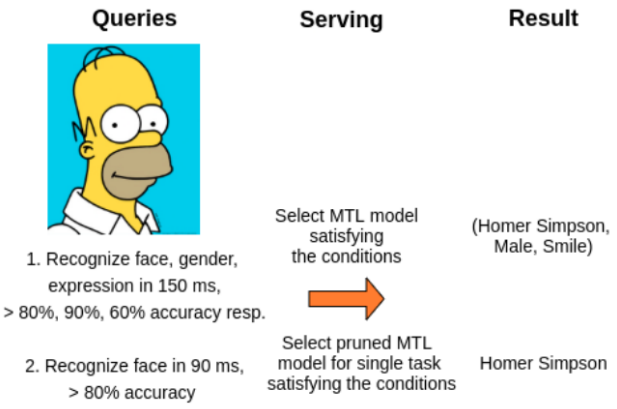

Figure 2. Sample types of queries and serving system pipeline

Model-variant profiler: Once the models are generated and made available on disk from the process in section 4.2, the profiler is the first step that the system runs. The profiler does a one-time profiling of the generated models on the deployed hardware configurations and measures performance of the models (accuracy), as well as loading and inference latency. These parameters are recorded in the memory of the system and named the Metadata Store. The total profiling time on the machine that we ran the inference system is a few minutes. This is a one-time start-up cost on a hardware and traded off with the flexibility to deploy in any kind of hardware configuration without additional effort.

Model selection policy: MTLSys determines which model-variant to select for each incoming query. It assumes they are ordered on the basis of priority and attempts to select the best performing model-variant that satisfies the query constraints. When a query arrives, the selection policy goes through the metadata store and iterates on the model-variants that meets the task constraints of the queries to find the model with that satisfies the latency with its combined loading and inference time. Among those models, it chooses the highest accuracy model if there are no accuracy constraints, or chooses the fastest model that meets the accuracy threshold of the query if there are accuracy constraints. If there is no model-variant that can meet the query requirements, it flags that as a mishit and logs it. The selection policy is designed to support systems with general MTL models, which do no have pruned task-head variants designated for single tasks, and MTL models that have such variants, as well as systems with multiple single-task-only models for those set of tasks.

For queries with multi-task requirements, the system can check for two types of accuracy metrics. For n number of tasks it can use n separate accuracy performance metrics and tries to find an MTL model or model combination that meets the metrics for all tasks. Fig. 5 is an experimental result with such a metric type. The other accuracy metric is the average of the performance metrics of the all the n tasks. The type of metric to use can be part of the query. For systems with only single-task model-variants, the inference system runs each model for a task sequentially. For fairness, it assigns a equal latency deadline by the dividing latency requirement equally for all tasks while searching for a model-variant of that task. Fig. 3 is an experimental result using this approach.

Metadata Store: The Metadata Store is an in-memory store that enables efficient access to the profiled performance and task data of the model-variants. It is a hash-table type data structure that ensures fast access (O(1)) to minimize minimize decision making time of the system. The information in Metadata Store consists of accuracy (from disk) and profiled loading and inference latency of the model-variants categorized by tasks.

The combination of profiler and selection policy, along with the Metadata Store, allows an interface for users to specify tasks, performance, and latency deadlines for each queries for any deployed hardware and eliminates need for the user to identify model-variants to serve varied application queries.

5 Experimental Evaluation

Experimental Setup : We use a different system for inference experiments than model-variant generation. The inference system runs on a 16-core Ryzen 7 5800H CPU and a NVIDIA 3050 TI GPU for CPU and GPU results respectively.

In order to test our hypotheses, we conducted several experiments to characterize the performance of MTLSys and its various components. In Figure 3, we show how our MTL-

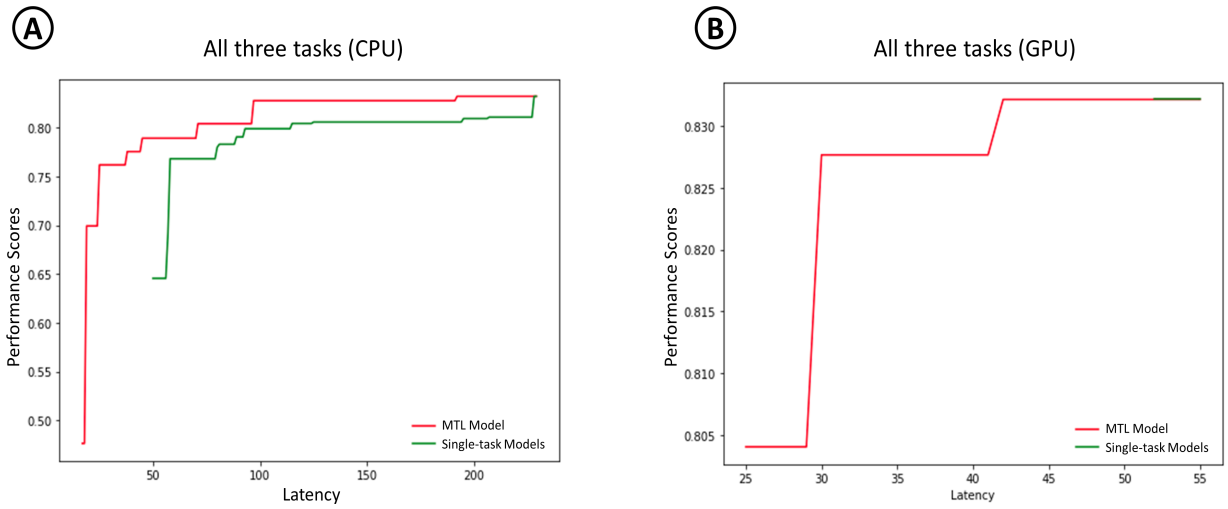

Figure 3. Pareto-frontier of accuracy and latency for the MTL model vs. the system with single-task models. In A, the best simple average performance score is returned for each latency using CPU inference. In B, the same experiment is conducted on the GPU.

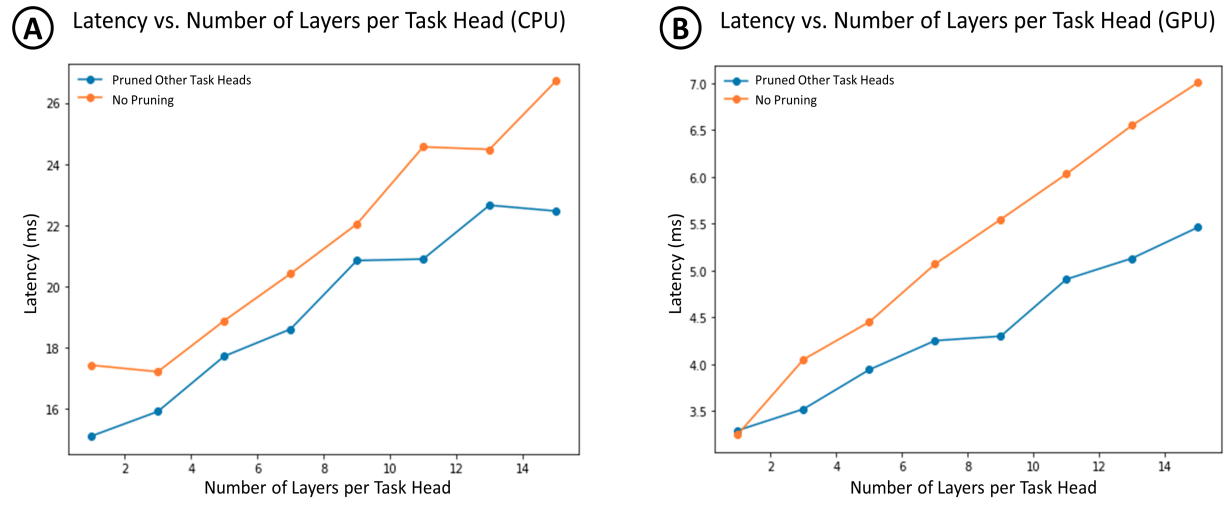

Figure 4. Evaluation of the effect of number of layers per task-specific head on latency. On the left, in A, we measure the inference latency on CPU for MTL models where the unused task heads are pruned off (in blue), vs. where they are left on. On the right, in B, the same experiment is conducted using GPU inference.

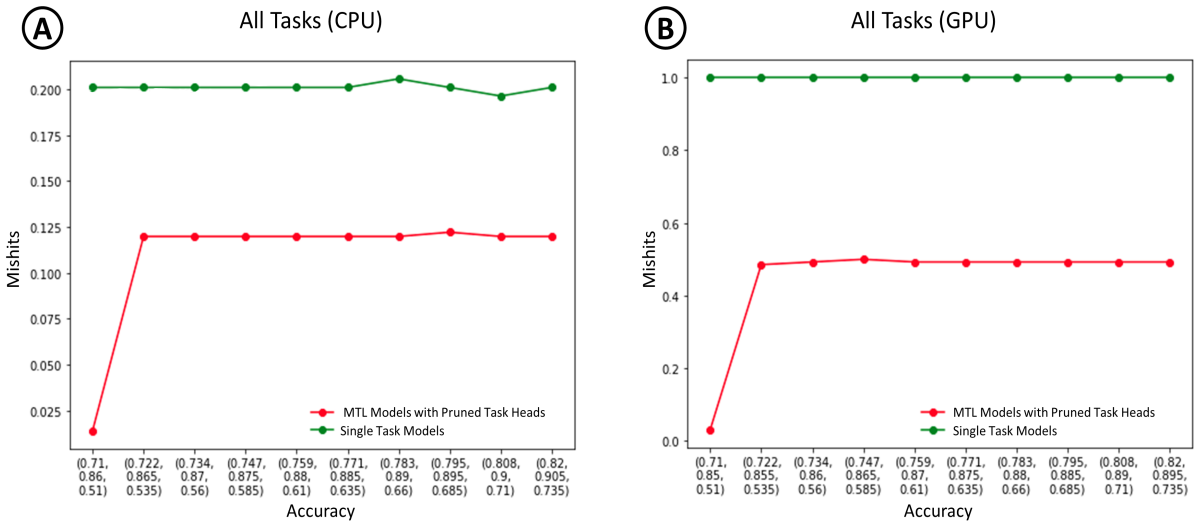

Figure 5. Evaluation of the number of mishits as a function of accuracy requirements on requests. The x-axis represents accuracy requirements for the (age, gender, ethnicity) tasks, and the y-axis represents the proportion of mishits occurring across the range of latencies (from min to max) for the models in the MTL system. The red line represents the MTL system and the green line represents the single-task system.

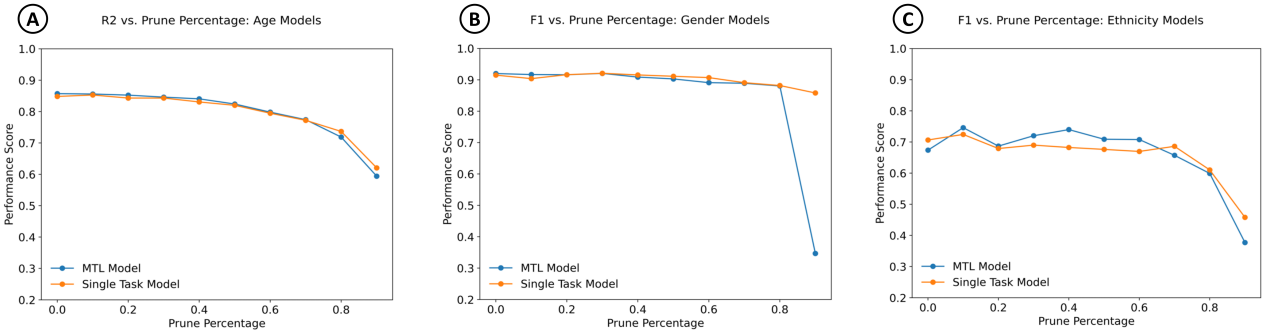

Figure 6. Evaluation of the effect of pruning on the accuracy of single-task models vs. MTL pretrained and single-task fine-tuned models. On the left, in A, we show the performance score against the prune percentage for the MTL-pretrained model in blue and the single-task model in orange for the age task. In B and C, respectively, we show the same results for the gender models and the ethnicity models.

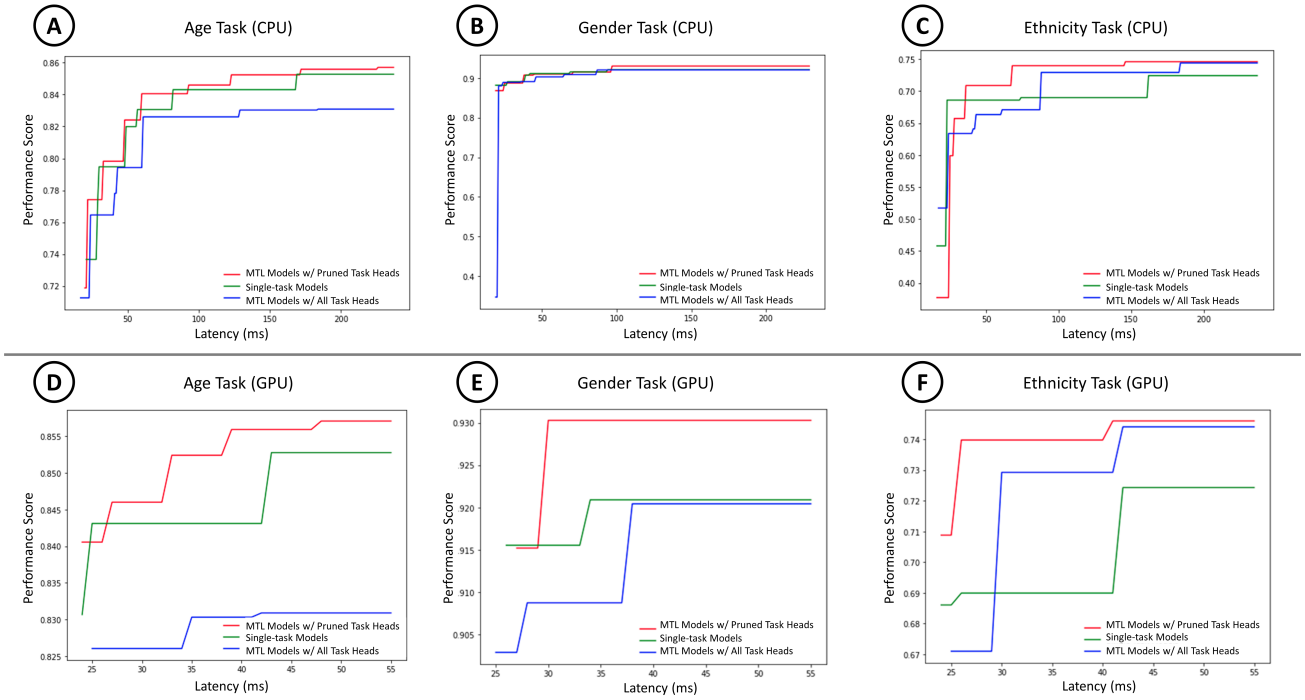

Figure 7. Pareto-frontier of accuracy and latency for MTL models with pruned unused task heads (red), single-task models (green), and MTL models with unpruned task heads (blue). On the x-axis of each plot, we show the latency in ms, and on the y-axis, the performance score for each respective task. Across the top are all the results for CPU inference, and on the bottom are the results for GPU inference. A and D show results for the age task, B and E for the gender task, and C and F for the ethnicity task.

system pushes out the Pareto frontier of accuracy-latency when compared to the single-task system. In Figure 3 A, we observe that the simple average performance score across all three tasks is higher for the MTL model system at each level of latency for CPU-inference. In Figure 3, we conducted the same experiment using GPU-inference, in which we observe the the MTL system is actually able to serve models at latencies lower than the single-task system is capable of, even given arbitrary accuracy.

In Figure 4, we measured the effects on latency of pruning the unused task-heads compared with leaving them unpruned, for single-task queries. Overall, we found that pruning off the unused task heads does indeed enable lower latencies, with a more pronounced effect as the number of layers in each task head grows, which we use as a simple way to increase the task-head complexity. For the GPU-inference latency at 15 task heads, we observed a latency decrease of more than 20% on the task-specialized pruned MTL model compared with the full MTL model.

In Figure 5, we measure the proportion of mishits for the MTL and single-task system across both CPU and GPU inference. Here, requests to the system contain both an accuracy and latency constraint; we run requests through the system across the spectrum of latencies from lowest to highest (according to the min and max latency of MTL-system models) and plot the proportion of mishits on the y-axis. As we can see from 3 A and B, the MTL system is consistently able to achieve lower mishits across varying accuracy requirements. In fact, the GPU-inference single-task system is not able to return any outputs meeting the accuracy and latency constraints for all tasks, which the MTL system is able to handle with its more efficient processing due to its shared encoder weights. Figure 5 shows that MTLSys achieves 1.6x fewer SLO mishits than a single-task model only system on CPU.

In Figure 6, we wanted to test whether models that were pretrained on the multi-task objective would be more robust to pruning when fine-tuned to perform single tasks. Overall, the results here were inconclusive–it appears that the MTL-pretrained models had slightly higher accuracy for low amounts of pruning across the 3 tasks, but that their performance degraded more than the single-task models for high levels of pruning.

Figure 7 is similar to Figure 3 in that we are showing the Pareto frontier of accuracy and latency. However, in Figure 7, we show this Pareto frontier for each task individually, and include an additional line for MTL models with pruned unused task heads. Overall, we observe that the MTL mod-

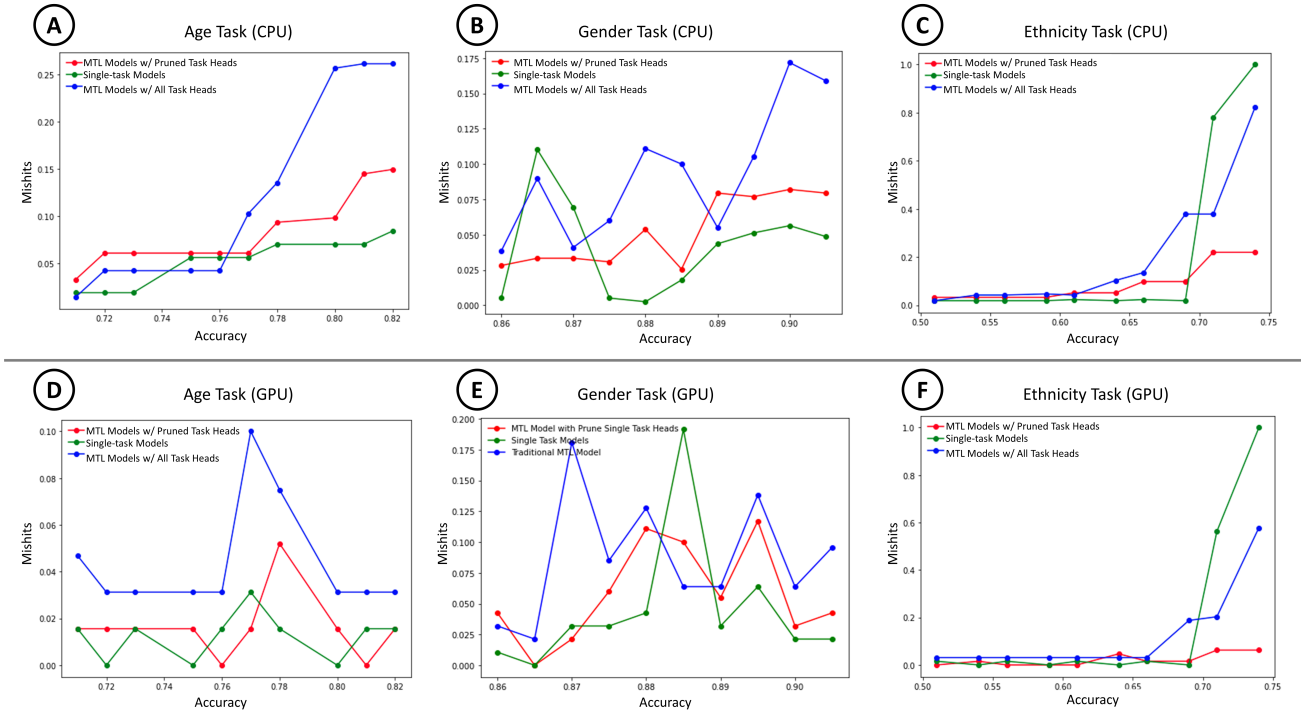

Figure 8. Plots of mishits against accuracy requirements for individual tasks. Across the top in A, B, and C are the CPU-inference results, and on the bottom in D, E, and F are the GPU-inference results. For each task, we measure the proportion of mishits over the range of latency requirement values, which span from the minimum to the maximum model latency of the MTL system. The blue line represents MTL models with all their task heads, the green represents single-task models, and the red represents MTL models with pruned unused task heads.

els with pruned unused task heads perform best for most levels of latency across both CPU and GPU, with the single-task models and unpruned MTL models performing in second or third place, depending on the task.

Finally, in Figure 8, we show results similar to those of Figure 5, in that we are measuring system mishits across different accuracy and latency requirements, by simulating queries to the system at a range of latencies between the min and max achievable latency for the MTL-system models for each accuracy requirement. However, here in Figure 8, we show these mishits across each specific task and include a line for the MTL models with pruned task heads. Overall, these results are less conclusive, but the MTL models with pruned task heads appear to perform either best or second best across most tasks and accuracy constraints.

6 Discussion

Overall, our results show that incorporating MTL awareness into our system enables it to improve the performance at each level of latency for both CPU and GPU and achieve fewer mishits. Furthermore, our experiments in Figure 4 demonstrated the benefits of task-aware pruning, with which we generated task-specific model variants to go along with the full MTL variants in the system.

The MTL system likely performs better than the baseline single-task system for multi-task requests because the MTL system is able to process multi-task requests much more efficiently due to its shared encoder. Instead of passing multiple inputs through the single-task models sequentially (one for each task), the MTL model can use its encoder once to get a shared representation, which can be leveraged by the task-specific heads to get task-specific outputs.

Some assumptions and limitations of our current prototype include that the number of tasks is known in advance, labeled data is available for each task, and already-sorted requests are sent sequentially to the system. Because our system starts from a parent network and generates model variants via pruning and fine-tuning from that, tasks cannot be added to existing models, and new variants would need to be created and retrained if tasks were to be added. Furthermore, the current prototype of our system does not leverage distributed training or inference and instead processes requests sequentially; this framing suited our experiments to directly test the benefits of MTL-awareness in an inference-serving system, but if MTLSys were to be further developed into a production-grade system, support for dis-

tributed training and inference, as well as smart queuing and task prioritization would be a necessity.

7 CONCLUSION

Incorporating MTL awareness into inference serving systems is a potential avenue to improve accuracy/latency trade-offs and reduce costs. In this paper, we presented our prototype of MTLSys, which demonstrates that incorporating MTL-awareness can reduce training time, extend the accuracy-latency Pareto frontier, and reduce mishits when compared to the same system that does not incorporate MTL models. Furthermore, we showed that pruning is a viable method to generate and maintain MTL model variants to meet latency/task/accuracy constraints, and that in our system, task-specific-pruned models perform as well or better than purely single-task models.

Some potential future directions include explicitly analyzing the cost benefits of our system, building out support for multi-node/distributed inference, and testing MTLSys across more task domains. From a cost perspective, MTLSys enables multi-task processing with better trade-offs between accuracy and latency when compared to using several single-task networks sequentially; as a next step, it would be interesting to explicitly look into the potential cost savings when comparing single nodes that are MTL-enabled with multi-task parallel inference using single-task networks. Furthermore, MTLSys could be developed to benefit from parallelization and distributed computing, for which MTL-specific prioritization and queuing schemes could be developed. From the perspective of further task domains to test, natural language processing (NLP) could be a domain of interest, for which a pre-trained encoder (e.g., BERT (Devlin et al., 2018)) could serve as the shared-task layers, upon which task-specific heads could be attached.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

Blalock, D., Ortiz, J. J. G., Frankle, J., and Guttag, J. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033*, 2020.

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

Chen, X., Wang, Y., Zhang, Y., Du, P., Xu, C., and Xu, C. Multi-task pruning for semantic segmentation networks. *arXiv preprint arXiv:2007.08386*, 2020.

Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 613–627, 2017.

Crankshaw, D., Sela, G.-E., Mo, X., Zumar, C., Stoica, I., Gonzalez, J., and Tumanov, A. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 477–491, 2020.

Dellinger, F., Boulay, T., Barrenechea, D. M., El-Hachimi, S., Leang, I., and Bürger, F. Multi-task network pruning and embedded optimization for real-time deployment in adas. *arXiv preprint arXiv:2101.07831*, 2021.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., and Mace, J. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 443–462, 2020.

Halpern, M., Boroujerdian, B., Mummert, T., Duesterwald, E., and Reddi, V. J. One size does not fit all: Quantifying and exposing the accuracy-latency trade-off in machine learning cloud service apis via tolerance tiers. *arXiv preprint arXiv:1906.11307*, 2019.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., and Soyke, J. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

Pan, S. J. and Yang, Q. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10): 1345–1359, 2009.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

Romero, F., Li, Q., Yadwadkar, N. J., and Kozyrakis, C. Infaas: A model-less inference serving system. *arXiv e-prints*, pp. arXiv–1905, 2019.

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

Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., and Sundaram, R. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 322–337, 2019.

Søgaard, A. and Goldberg, Y. Deep multi-task learning with low level tasks supervised at lower layers. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 231–235, 2016.

Vanholder, H. Efficient inference with tensorrt, 2016.

Wu, H., Judd, P., Zhang, X., Isaev, M., and Micikevicius, P. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*, 2020.

Yang, X., Zeng, Z., Yeo, S. Y., Tan, C., Tey, H. L., and Su, Y. A novel multi-task deep learning model for skin lesion segmentation and classification. *arXiv preprint arXiv:1703.01025*, 2017.

Zhang, J., Elnikety, S., Zarar, S., Gupta, A., and Garg, S. Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

Zhang, Zhifei, S. Y. and Qi, H. Age progression/regression by conditional adversarial autoencoder. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2017.

Zhang, Y. and Yang, Q. A survey on multi-task learning. *arXiv preprint arXiv:1707.08114*, 2017.