COMBINATORIAL REINFORCEMENT LEARNING BASED SCHEDULING FOR DNN EXECUTION ON EDGE

Anonymous authors
Paper under double-blind review

ABSTRACT

The past half-decade has seen unprecedented growth in machine learning with deep neural networks (DNNs) that represent state-of-the-art in many real-world applications. However, DNNs have substantial computational and memory requirements, in which the compilation of its computational graphs has great impact in resource-constrained (e.g., computation, I/O, and memory bounded) edge computing systems. While efficient execution of its computational graph leads to high-performance and energy-efficient execution, generating an optimal computational graph schedule is known as *NP-hard* problem. The complexity of scheduling the DNNs computational graphs will further increase on pipelined multi-core system considering memory communication cost, as well as the increasing size of DNNs. This work presents a reinforcement learning based scheduling framework, which imitates the behaviors of optimal optimization algorithms at the speed of inference, and compiles arbitrary DNNs computational graphs without re-training. Our framework has demonstrated up to $\sim 2.5\times$ runtime speedups over the commercial Edge TPU compiler, using ten popular ImageNet models, on physical Google Edge TPUs system. More importantly, compared to the exact optimization methods solved by heuristics and brute-force, the proposed RL scheduling improves the scheduling runtime by several orders of magnitude.

1 INTRODUCTION

Deep neural networks (DNNs) represent state-of-the-art in many applications, but introduce substantial computational and memory requirements, which greatly limit their training and deployment in resource-constrained (e.g., compute and memory resources) environments. To efficiently deploy DNNs on the hardware platforms, it usually requires designated compilers that take in front-end DNN models and map them to the platforms. As the size of DNN models rises, it becomes more challenging to deploy the models onto edge devices with small on-chip buffer size using static and heuristic-based execution scheduling methods, specifically for edge computing ecosystems, such as Google Edge, Microsoft Azure ML, etc. To efficiently utilize those hardware platforms, scheduling algorithms implemented in deep learning compilers are critical in deploying such hyper-dimensional computationally-intensive workloads, which is a classical *NP-hard* combinatorial optimization problem (Lenté et al. (2014); Kuchcinski (2003)). Mostly, vendor-specific libraries such as Nvidia cuBLAS, TVM, and TF-Lite (Chen et al. (2018); Abadi et al. (2016); Sanders & Kandrot (2010)), rely on hand-crafted domain-specific heuristics to optimize the executions, which trades the execution performance for scheduling runtime. Specifically, the limitations of existing DNNs computational graph scheduling can be summarized as follows: **(1)** existing algorithms are either heuristics that lack in quality of optimization, or exact/brute-force algorithms lack in scalability. The challenges for large-scale deep learning executions are still rising up in particularly for edge devices. **(2)** Hand-crafted heuristics can be efficient but the development process requires high engineering efforts and domain knowledge in compilation and hardware systems. **(3)** There have recently seen ML-based frameworks for scheduling. However, they are either limited to certain graph structure/sizes, or requires online training or retraining(Mao et al. (2019); Chen & Shen (2019); Sheng et al. (2021)).

This work presents a novel combinatorial reinforcement learning framework that imitates the behaviors of graph combinatorial algorithms for scheduling DNNs computational graphs. This framework aims to perform near-optimum scheduling on edge computing platforms without retraining, while

deploying arbitrary sized DNNs computational graphs. Specifically, the proposed framework imitates the algorithmic behaviors of existing optimal scheduling algorithm, while the training process is fully conducted on synthetic sampling. For experimental evaluations, we build a physical multi-stage pipelining Edge TPU system (Yazdanbakhsh et al. (2021); Boroumand et al. (2021)). The experimental results conducted on this physical computing platform demonstrates significant runtime speedups over the commercial Edge TPU compiler with ten popular ImageNet DNNs.

2 BACKGROUND

Combinatorial optimization problems are fundamental to a wide range of research communities. Many solutions to those problems rely on handcrafted heuristics, such as compiler optimizations and scheduling, that guide their search procedures to find sub-optimal solutions efficiently. Similar to many other combinatorial optimization problems, the development of efficient compiler optimization heuristics requires extensive domain-specific knowledge in algorithm and targeted hardware platform. In contrast, reinforcement learning methods are applicable across many tasks. They can discover their own heuristics, thus requiring less hand-engineering, and more importantly, can break through the inertia of traditional R&D methods. With the recent success in neural architectures and optimization algorithms Kingma & Ba (2014); Vaswani et al. (2017); Wu et al. (2016); Ruder (2016), RL has been successfully applied to explore and discover new heuristics and rules for many classic combinatorial problems, such as mixed integer linear programming, traveling salesman problem (Song et al. (2019); Cappart et al. (2019); Gambardella & Dorigo (1995); Khalil et al. (2017); Chen & Tian (2019); Nazari et al. (2018); Lodi & Zarpellon (2017); Hottung et al. (2020); Karapetyan et al. (2017)). However, there are few works that leverage RL in accelerating combinatorial optimizations on directed graphs. Moreover, existing RL/ML based scheduling frameworks (Mao et al. (2019); Chen & Shen (2019); Sheng et al. (2021)) are highly limited to domain-specific platforms and specific upper bound of graph size, and have not demonstrated on real-world physical computing platforms.

3 APPROACH

3.1 PRELIMINARIES

Computational Graphs of DNNs In the modern deep learning frameworks, machine learning algorithms are represented as computational graphs, where each graph is a directed graph $G(V, E)$ with nodes V describing operations. The edges E represent the dataflows that connect the operators and input/output tensors. In particular, while deploying DNNs on hardware accelerators, the computational graphs are mostly represented as directed acyclic graphs (DAG) while the acyclic paths are unrolled to maximize the hardware performance. The computational graphs are mostly generated with static compilation.

Specifically, the optimization objectives can be defined as follows: **Given:** *(1)* A DAG $G(V, E)$ where V represents the set of operations in the DNNs computational graphs, and E represents the set of edges; *(2)* A set of scheduling constraints, which may include dependency constraints (E), resource constraints, execution time, memory allocations, etc. **Objective:** Construct an exact optimal schedule $S = s_0, s_1, ...s_n$, where V will be allocated to S (where $n \leq |V|$) that satisfies all scheduling constraints. For example, in a multi-stage pipelined Edge TPU system in Figure 1, resulted schedule assigns computation node N_i to s_0 (Edge TPU:0), N_k to s_1 (Edge TPU:1), N_l and N_o to s_2 (Edge TPU:2), N_j, N_p and N_q to s_3 (Edge TPU:3), etc.

DNNs Computational Graph Embedding For the favor of learning purpose, computational graph has to be mapped into vector space. Considering the performance of scheduling on edge devices, there are three critical attributes of the computational graphs should be included in the embedding: **(1)** the connectivity of computational graphs (relative coordinates of vertices), **(2)** topological order of operators (absolute coordinates), and **(3)** memory consumption for operator. Specifically, as shown in Figure 1, absolute coordinates embedding represents the topological order of every node, while relative position are encoded using both the absolute coordinates and node IDs. Intuitively, relative coordinates maintain the dependency constraint, and absolute coordinates mimics the feasible scheduling space for a given node. For example, in Figure 1, the dependency prerequisite for

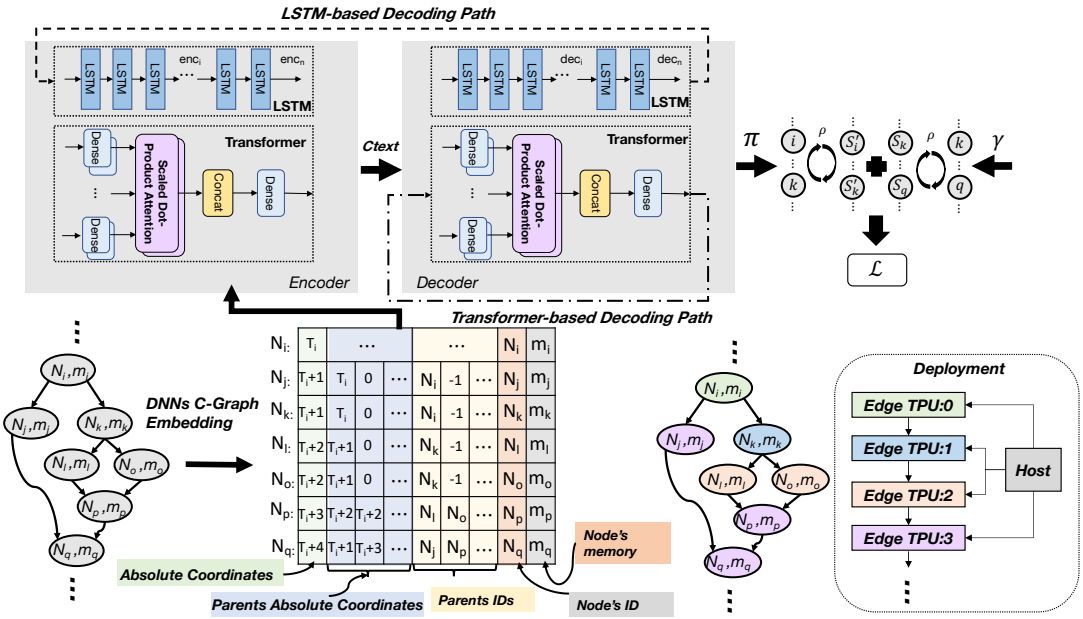


Figure 1: Overview of the proposed RL framework, including the DNNs computational graph embedding, RL agent neural architecture, and inference and deployment illustration.

N_p to operate is the complete scheduling of N_l and N_o while N_j and N_p can be scheduled together due to free of dependence if permitted computation resource, where all information are encoded in the embedding vectors. Besides, the absolute coordinates of node N_j and its parent node N_i are 2 and 1, the available schedule locations for N_j can be calculated, i.e., any free locations between the absolute coordinates of N_i and N_j $\{S_0, S_1, S_2, S_3\}$. Finally, cache and memory locality is one of the most critical metric in DNNs acceleration (Jouppi et al. (2017)). Thus, we add the memory consumption of each operator in the last embedding column.

Specifically, as shown in Figure 1, the embedding of computational graph consists of four components: **1)** *absolute coordinates* generated based on topological ordering. Note that there are many variants topological orders of a given DAG. In this work, we use As-Soon-As-Possible (ASAP) ordering, where each node is ordered as close as to the source node; **2)** *relative coordinates* consist of parent nodes absolute coordinates and IDs. Note that the source nodes absolute coordinates are set to 0, and their IDs are set to -1; **3)** node IDs are unique integers generated by hashing all the operator names; **4)** memory consumption of the operator. This embedding vector of graphs will be the input of the RL agent. While collecting real-world DNNs computational graphs execution data is very time consuming, we propose to train the RL framework with full synthetic graphs.

3.2 Formulations and Neural Architecture

As discussed earlier, brute-force and exact methods scheduling algorithms can generate the best scheduling solutions search(Lenté et al. (2014); Kuchcinski (2003)) at the cost of long runtime, which fails in scaling up to large problems. While heuristic algorithms optimize the schedules more efficiently at scale but suffers in quality of results. Thus, we aim to develop a RL framework that imitates the algorithmic behaviors of any optimal scheduling algorithm (e.g., exact or brute-force), such that it performs polynomial time scheduling with near-optimal quality of results at inference runtime. Currently, Reinforcement Learning(RL) catches more and more attention in solving graph combinatorial problems.

RL Formulation for DNNs Computational Graph Scheduling Given a computational graph as DAG $G(V, E)$, RL-agent is trained to develop a policy π, that picks computation node in the same order as specific algorithm does. We define sequence order figured out by our method as $\pi(i), i \in |V|$, by some given deterministic scheduling method as $\gamma(i), i \in |V|$. Thus, the reward function is

Algorithm 1: Pointing Mechanism Decoding Flow Illustration

Initialization: $d \leftarrow$decoder input; $C = \{ctext_i\}_{i=1}^{n}$; dec_i; $\{emb_i\}_{i=1}^{n}$; θ, ω, β (trainable parameters);
for $i = 1$ *to* n **do**
  h, $dec_i \leftarrow$ Dec(d, dec_{i-1});
  h $\leftarrow$ glimpse(C$*\boldsymbol{\theta_g}$, $\boldsymbol{\omega_g}$·h+$\boldsymbol{\beta_g}$);
  $P^i \leftarrow$ pointer(tanh(C$*\boldsymbol{\theta_p}$, $\boldsymbol{\omega_p}$· h + $\boldsymbol{\beta_p}$));
  $idx_i \leftarrow$ argmax $\frac{\exp P_i^i}{\sum_{j=1}^{n} \exp P_j^i}$;
  d $\leftarrow emb_{idx_i}$
end
$S' \leftarrow \rho\,(\pi(i), s_i)$; S $\leftarrow \rho(\gamma(i), s_i), i \in [1, n]$;
Loss $\leftarrow \sum_{i=1}^{n} P^i \cdot (1 - \frac{\sum S(i) \cdot S'(i)}{\max(\sqrt{\sum S(i)^2} \cdot \sqrt{\sum S'(i)^2}, \epsilon)} - b(G))$

designed as the similarity comparison between them:

$$R = \frac{\sum \pi(i) \cdot \gamma(i)}{\max(\sqrt{\sum \pi(i)^2} \cdot \sqrt{\sum \gamma(i)^2}, \epsilon)} \tag{1}$$

Let $N_i \rightarrow s_i, i \in [1, n]$ be the solution of scheduling node N_i at s_i. Let $S' = \{s_0, s_i, ...s_n\}$ be the sequence produced by our policy as $\pi(i)$, i.e., the output of the RL agent for a given computational graph. The targeted scheduling solution generated by given deterministic scheduling algorithm is the ground truth label sequence, denoted as S.

$$S' = \rho(\pi(i), s_i); \quad S = \rho(\gamma(i), s_i), i \in [1, n] \tag{2}$$

Thus, to optimize the RL agent to imitates the behaviors of the targeted deterministic algorithm, the reward is designed using `cosine similarity`, with the generated label sequence S' and the ground truth sequence S as inputs (Equation 3).

$$R = \frac{\sum S(i) \cdot S'(i)}{\max(\sqrt{\sum S(i)^2} \cdot \sqrt{\sum S'(i)^2}, \epsilon)}, i \in [1, n] \tag{3}$$

While maximizing the reward function R, parameters of the stochastic policy $p(\pi|G)$ will be optimized to assign high probabilities to sequence order closer to the target sequence. The chain rule utilized to factorize the sequence probability distribution can be expressed as:

$$p(\pi|G) = \prod_{i=1}^{n} p(\pi(i)|\pi(< i), G) \tag{4}$$

RL Agent Architecture Here, we extent pointer network (PtrNet) architecture (Bello et al. (2016)) as the RL agent. PtrNet excels in finding path with target objective to be optimized. It achieves huge success in solving some combinatorial problem over graphs, such as Traveling Salesman Problem (Bello et al. (2016)). With the advantage of attention mechanism (Vaswani et al. (2017)), PtrNet reinforces the dependency constraints among nodes and overcomes the limitations of learning combinatorial graph algorithms with fixed size of graph inputs. However, there are two novel challenges in scheduling DNNs computational graph: 1) scheduling computational graph is equivalent to generating node permutation in a directional fashion; 2) in order to fully evaluate the RL scheduling performance on physical edge devices, the results need to satisfy domain-specific hardware execution requirements.

The RL agent architecture is an encoder-decoder based PtrNet, consisting of encoding and decoding components, shown in Figure 1. Specifically, each component is configurable to either Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber (1997)) or Transformer (Vaswani et al. (2017)). The proposed architecture introduces four different configurations of encoding-decoding. **Encoder network** – The encoder network digests the encoding input queue q of nodes and transforms it into a sequence of context $\{ctext\}_{i=1}^{n}$ representing the information of each node, where $ctext_i \in R^d$, d is the hidden dimension of RNNs. The concatenation of all contexts will be used as reference matrix C in nodes pickup during decoding step, $C(i) = ctext_i, i \in [1, n]$. Up to this step, encoding

process works the same for using either LSTM or Transformer as encoder. While using LSTM as encoder, additionally, it produces a sequence of latent memory states $\{enc_i\}_{i=1}^{n}$ recording the propagation encoding message from first node to current one along with the contexts, where the final state enc_n is the initial latent memory state for decoding. **Decoder network** – While using LSTM as decoder network, it generates its own latent memory state $dec_i \in R^d$ at each decoding step i to update the previous one. We illustrate the detailed decoding procedure in Algorithm 1. With context reference matrix C from encoder, decoder will produce a selection probability distribution over candidate computation nodes using pointing mechanism proposed in (Bello et al. (2016)). Once a computation node is picked, its embedding will be passed as the input to LSTMs during the next decoding step. The input to the first decoding step is a trainable parameter sharing same dimensions of node embedding. For Transformer-based decoder, it generates its selection reference matrix $\{K_i\}, K_i \in R^{n \times d}, i \in [1, n]$ at each decoding step i. With selection reference matrix K_i, decoder will produce a selection distribution probability over computation nodes using linear transformation. Updated selection reference matrix K_{i+1} will behave as input in next decoding step. The initial selection reference matrix is the context reference matrix from encoder $K_1 = C$. In Section 4.1, we provide comprehensive empirical studies of the four configurations introduced by the proposed architectures.

RL Training We use model-free policy-based RL training method to optimize the parameters of a pointer network denoted as θ. The learning objective is the expected similarity of node distribution. Given an input graph G, the optimization objective can be defined in Equation 5, where it maximizes the cosine similarity reward.

$$J(\theta|G) = \mathbb{E}_{\pi \sim p_\theta(\cdot|G)}(1 - R(\pi|G)) \tag{5}$$

In this work, we deploy policy gradient methods and stochastic gradient descent to optimize the parameters (Williams (1992)). Reward as score to evaluate the resulted distribution probability is applied as coefficient in gradient calculation, such that the gradient of (5) is constructed as follows:

$$\nabla_\theta J(\theta|G) = \mathbb{E}_{\pi \sim p_\theta(\cdot|G)}[(1 - R(\pi|G) - b(G))\nabla_\theta \log p_\theta(\pi|G)] \tag{6}$$

where $b(G)$ represents a baseline recording the highest score in previous training, applied in loss function to reduce the variance of the gradients. Note that unlike existing RL-based scheduling works, our framework imitates the behaviors of any scheduling algorithm, such that the resulted RL agent can perform as generic as the imitated deterministic algorithm. In addition, there is a critical advantage to use synthetic data sampling during RL training: synthetic graph sampler has full control of graph complexity, memory attributes of operators, which offers much better data coverage over real-world DNN computation graphs. In this work, we integrate a DAG sampler in our RL training framework, where we can control the complexity of the sampled graphs by limiting the graph degrees, memory range of each operator, and total topological levels, which mimics the structural and properties of DNNs computational graphs. Detailed settings of sampling are discussed in result sections. In addition, an evaluation-only metric `mismatch` is introduced for evaluating the performance of the RL agent at inference stage (Equation 7). Specifically, `mismatch` counts the number of level distribution differing in generated label sequence and its ground truth one. In other words, it measures the absolute similarity between the output of RL agent and ground truth, where `cosine similarity` measures the normalized similarity. While it is possible to deploy `mismatch` directly as the RL reward function, the training performance is much worse than using the normalized similarity. `Mismatch` (M) can be defined as:

$$M = \sum_{i=1}^{n} [(S(i) - S'(i)) \neq 0] \tag{7}$$

Post-Inference Processing Unlike algorithms such as TSP, DNNs computational graph scheduling needs to satisfy domain-specific constraints to successfully deploy the scheduled graphs on hardware platforms. Thus, a post-inference processing procedure is added in our RL framework, which is executed at the deployment stage, which takes the inferenced output S' as inputs. To be specific for our evaluation Edge TPU platforms, this procedure corrects the dependency violation by simply pushing the node involved forward, which is a deterministic step with minimum changes to the RL solution. Besides, Edge TPU hardware requires children's nodes of any node to be in same pipeline. In this case, the post-inference procedure assigns these nodes to the earliest predicted stage among theirs.

4 EXPERIMENTAL RESULTS

In this section, we will first provide comprehensive studies of the proposed RL approaches from the aspects of neural architecture, data preparation and embedding, and leveraging hardware domain-specific knowledge in the training setups. Specifically, we analyze the approaches with various configurations, including **1)** architecture configurations of the encoding-decoding neural architecture using Transformer and LSTMs, **2)** variations and coverages of dataset and embedding, and **3)** comparisons of training w and w/o TPU accelerator domain-specific knowledge. Second, we experimentally demonstrate the effectiveness of the proposed RL-based scheduling approaches on physical computing platforms built with Google Edge TPUs. Specifically, we build a central-hosted pipelined Edge TPU system to evaluate the real-world computation performance improvements, with options of 4-stage, 5-stage, and 6-stage pipelining setup for DNNs inference execution. All runtime results included in this section are obtained in this physical computing platforms and are compared to the commercial Edge TPU compiler. The experimental results show that the proposed RL scheduler can consistently outperforms commercial Edge TPU compiler in all three different pipeline settings. Training, inference and explorations of the proposed RL methods are conducted on one Nvidia 2080 Ti with Intel Xeon Gold 6230 x20 CPUs.

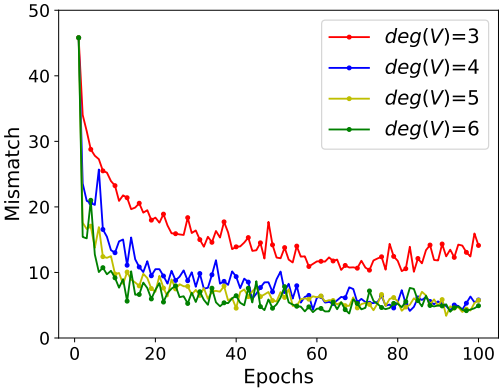
4.1 EXPLORATIONS OF TRAINING AND NEURAL ARCHITECTURES

To optimize the performance of RL-based scheduling framework for deploying real-world DNNs on edge devices, we find several critical empirical evidence that the training setups and architecture configurations of PtrNet are critical. In this work, training setups specifically refer to the synthetic training dataset constructions. Note that all the RL models used in this work are trained with synthetic dataset only and testing computation graphs are generated with real-world DNNs models. As discussed in Section 3, the main reason is that there is very limited number of computation graphs for training the RL models, which restricts the generalizability of RL in solving graph-based combinatorial optimization problems. Besides, to simultaneously minimize the training efforts, the training dataset needs to balance between data coverage and graph size (i.e., $|V|$ and $|E|$ in $G(V, E)$). In this work, our goal is to limit the training dataset with $|V| \leq 50$ where $|E|$ is constrained with $|V|$ and degree of graphs $deg(V)$ in training, such that the trained RL model is generalizable for scheduling arbitrary DNNs computation graphs. Thus, in the rest of this section, all experiments conducted in this section are trained with 100 epochs with learning rate a 10^{-4} and batch size as 128, using Adam optimization algorithm if not specified.

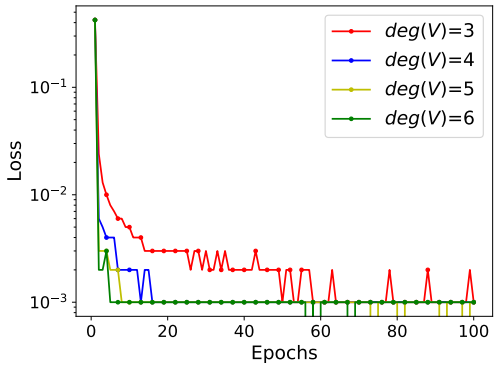
Graph Complexity of Training Dataset While all the graphs in the training datasets with $|V| = 30$, the complexity of the training graphs can vary with different graph degrees, i.e., $deg(V)$ is the maximum number of incoming edges connecting to the vertices V in graph $G(V, E)$. In practice, DNNs computation graphs can have very different graph structures, e.g., the TF-Lite compiled computation graph of ResNet152 has $|V| = 517$ and $deg(V) = 2$, and Inception-ResNet-V2 has $|V| = 782$ and $deg(V) = 4$. In order to train the RL model to be generalizable to wide range of DNNs models, the model should be trained with awareness of various graph degree sizes. In Figure 2a, we evaluate the testing performance with RL models trained with different degree sizes $deg(V)$=$\{$3,4,5,6$\}$, while testing graphs have $deg(V) = 6$ and $|V| = 50$. Note that the number of graphs in the training dataset are the same for all the four tests. The results have demonstrated the effectiveness of leveraging higher degree graphs in training dataset. For example, the testing `mismatch` obtained using RL model trained with $deg(V) = 3$ is about 50% higher than the training with $deg(V) = \{4, 5, 6\}$. However, there are two drawbacks if training graph degree increases: (a) the vector space embedding dimension increases as the degree increases, which increases the connection/complexity of the graph and training efforts; (b) as the upper bound degree of graphs increases, the number of all possible graphs increases dramatically; thus, while fixing the number of graphs in training set, the coverage/generalizability of the training dataset could decrease significantly. In practice, the popular DNNs models mostly have computation graphs degrees $deg(V) \leq 6$, in particularly for the models prepared for Edge TPU devices[1]. Therefore, in Section 4.2, the RL model used for Edge TPU scheduling is trained with $deg(V) = 6$ dataset.

Directional Graph CO Learning Explorations There are two traditional scheduling or pipelining formulations that solve the combinatorial problems in different directions of execution, namely *as*

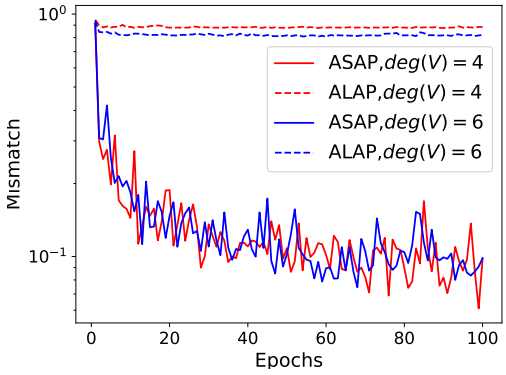
[1] `https://coral.ai/models/image-classification/`



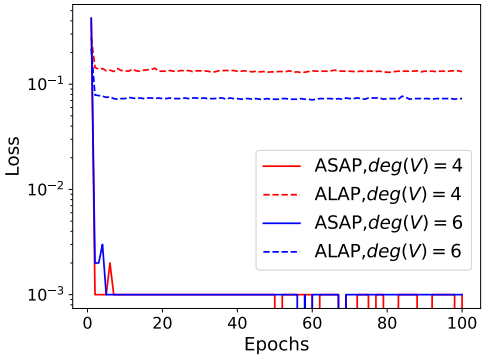
(a) Testing mismatch with different training graph in-degree constrains $deg(V)$={3,4,5,6}



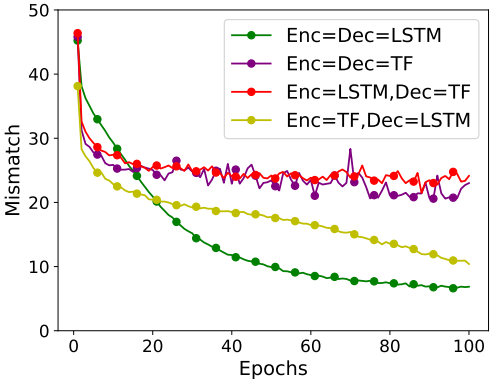
(b) Testing loss while training with different graph in-degree constrains $deg(V)$={3,4,5,6}



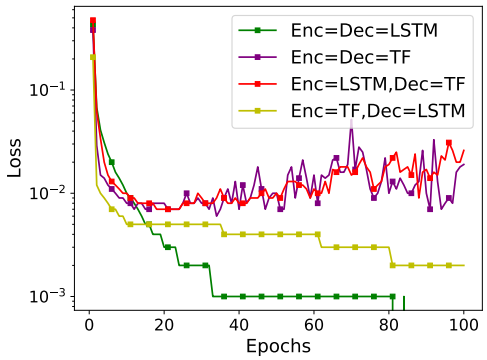
(c) Testing mismatch comparisons between training with ASAP and ALAP scheduling as targeted labels



(d) Testing loss comparisons between training with ASAP and ALAP scheduling as targeted labels



(e) Testing mismatch with four different configurations of PtrNet (lr=10^{-6})



(f) Testing loss with four different configurations of PtrNet (lr=10^{-6})

Figure 2: Explorations of training setups and neural architecture configurations by measuring the testing performance using `mismatch` and loss values, specifically covering aspects of training graphs complexity, scheduling types, and encoder-decoder architectures.

soon as possible (ASAP) and *as late as possible* (ALAP) scheduling. ASAP scheduling firstly maximizes the number of pipeline stages that are allowed, and then schedules operations in the earliest possible control step, subject to satisfying the hardware and data dependency (DAG dependency) constrains. On the other hand, ALAP will schedule the operations in the latest possible control steps with the same maximum pipeline stages. Mostly, the performance of classic ASAP and ALAP scheduling algorithms differs in computation resource utilization. Here, we leverage the same concepts in RL training, where the labels of "ASAP" and "ALAP" style learning approaches are in the

Table 1: Statistics of DNNs models and their computational graphs used for evaluating inference runtime on Pipelined Edge TPU Systems

	Xception Chollet (2017)	ResNet50 He et al. (2016a)	ResNet101 He et al. (2016a)	ResNet152 He et al. (2016a)	DenseNet201 Huang et al. (2017)
$\|\mathbf{V}\|$	134	177	347	517	709
$\mathbf{deg(V)}$	2	2	2	2	2
Depth	125	168	338	508	708
	DenseNet121 Huang et al. (2017)	ResNet101v2 Yu et al. (2018)	ResNet152v2 He et al. (2016b)	DenseNet169 Huang et al. (2017)	InceptionResNetv2 Szegedy et al. (2017)
$\|\mathbf{V}\|$	429	379	566	597	782
$\mathbf{deg(V)}$	2	2	2	2	4
Depth	428	371	558	596	571

original ground truth order and its reversed order, respectively (Figures 2c and 2d). We can see that the ALAP RL training setup does not converge, regardless of the graph complexity of the training dataset (testing degree same as training). Thus, in the rest of this section, all models are trained w.r.t ASAP algorithm.

Neural Architecture Configurations While PtrNet is mainly an encoder-decoder neural architecture that learns the behaviors of a given CO graph algorithm, the common architectures for encoder/decoder are attention-based networks such as Transformer (TF) and LSTMs. Specifically, the TF configurations are: number of layers and heads for encoder as 1, for decoder as 2; and LSTMs configurations are: hidden dimension as 256 with 1 glimpse. Embedding dimension for both is fixed at 256. Here, we empirically evaluate the configurations by configuring the encoder/decoder using either TF or LSTM. As shown in Figures 2e and 2f, in learning pipelining optimizations for edge computing systems, we conclude that **(1)** decoder has to be built with LSTM, **(2)** if decoder is built with LSTMs, both TF and LSTM encoders will converge, and **(3)** the best configuration of PtrNet is having both encoder and decoder built with LSTMs since training and tuning TF based PtrNet is more computational expensive.

4.2 Experimental Studies on Pipelined Edge TPUs

Based on all experiments and discussed in Section 4.1, we train and deploy the RL model to our Google Edge TPU system, trained with 1) dataset degrees $deg(V) = 6$ with $|V| = 30$, 2) ground truth labeling follows rules of ASAP scheduling algorithm, and 3) neural architecture with both encoder and decoder built with LSTMs. Training is conducted on 300 epochs with learning rate as 10^{-4} and batch size as 128.

Experimental setups on Edge TPU runtime Our comparison baseline for computational graph compilation is the commercial version of Edge TPU compiler[2]. The runtime comparisons are evaluated using **ten** medium-scale popular image classification models shown in Figure 3, including ImageNet models listed in Table 1. While TPUs can only execute INT8 quantized neural networks, we perform INT8 quantization using TF-Lite with Tensorflow embedded pre-trained models. These models are the inputs to Edge TPU compiler and our RL scheduling framework.

To minimize the impacts of runtime variations in executing DNNs on Edge TPU system, the results included in Figure 3 are the mean runtime of 10 rounds of 5,000 ImageNet inference, using the ten models. In Figure 3, the horizontal axis represents the runtime results normalized w.r.t runtime obtained with commercial Edge TPU Compiler, and vertical axis shows the deployed models. The results have clearly demonstrated that the proposed RL scheduler can consistently outperform commercial Edge TPU compiler. Specifically, our approach improves the runtime on 4, 5, and 6-stage pipelining Edge TPU 5.5%, 5.8%, and 28.4%, on average. Moreover, we find that the improvements of the same model are not consistent over different pipeline stages. In other words, the performance of RL scheduler over Edge TPU compiler is not strictly caused by the graph structures, but also the available computing resources in the system. For example, ResNet152 in 4-stage and 5-stage Edge TPU, the RL improves the runtime by 1.42%, and improves 56.3% in 6-stage.

[2] `https://coral.ai/docs/EdgeTPU/compiler/`

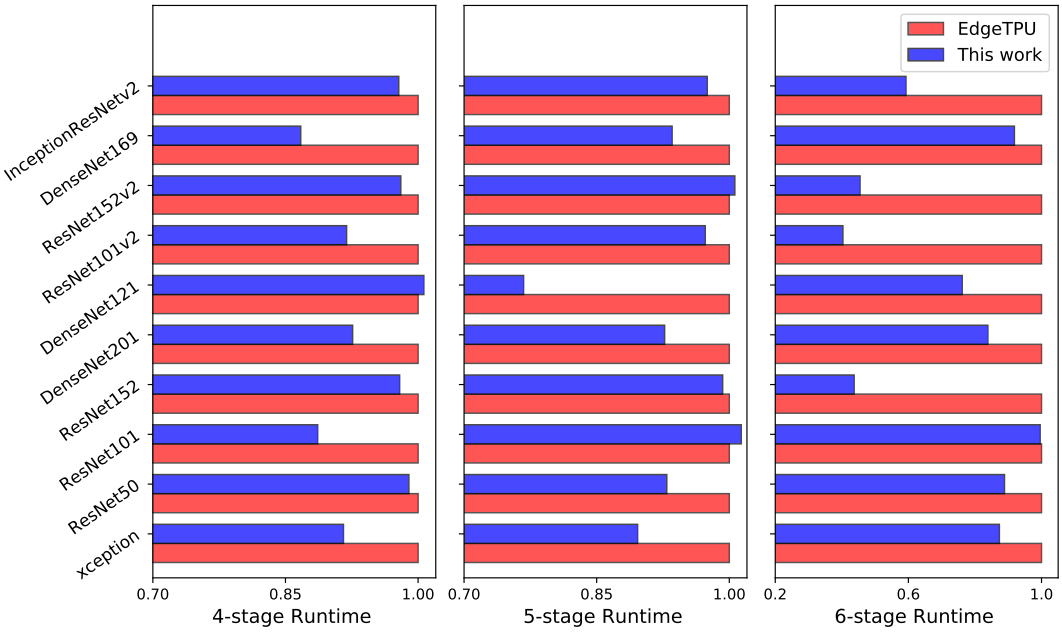


Figure 3: Multi-stage pipelined Edge TPUs inference runtime comparisons between the proposed RL methods and commercial Edge TPU compiler (baseline scale=1). The runtime performance has been consistently improved over commercial compiler with 4, 5, and 6-stage pipelined Edge TPU system e.g., ResNet101v2 and ResNet152 execute $\sim 2.5\times$ faster than Edge TPU compiler.

Explainable domain knowledge learned by RL To understand the runtime differences, we analyze the post-compiled execution graphs to extract explainable knowledge learned by the reinforcement process. Note that TPU architecture is a template-based machine learning accelerator including a 2D array of processing elements (PE), where all PEs share memory one dedicated over the core. In other words, while deploying the computation graph operators to the cores, it is critical to balance the memory and compute. In addition, for the multi-TPU pipelining with a central CPU host, the communication from stage-to-stage could significantly dominate the runtime performance, since it communicates via slow I/O interface (e.g., USB 3.0). These have been confirmed by the two observations by comparing the compilations between Edge TPU compiler and our RL scheduler: (1) our RL model further optimizes the memory utilization for each stage, i.e., minimizing off-chip DRAM utilization and maximizing cache locality; (2) with more compute and memory resources (e.g., 6-stage), the communication cost becomes more critical since the models can be simply deployed with cache-only computation.

Generalizability analysis Note that the entire training process is fully conducted on synthetic graph sampling, where the training graph size is $|V| = 30$. The experimental results shown in Table 1 and Figure 3 also conclude the generalizability of the proposed approach. Specifically, as shown in Table 1, the size of the DNNs computational graphs varies from 134 to 782, which are all far beyond the training graph size $|V| = 30$. Besides, the degree size of the deployed DNNs models varies as well, e.g., $deg(V) = 4$ in InceptionResNetv2 and $deg(V) = 2$ in other models. The proposed RL framework consistently outperform commercial Edge TPU compiler in scheduling large computational graphs, even with graph that is $26\times$ larger than the synthetic training samples ($|V| = 782$ in InceptionResNetv2 versus $|V| = 30$ in training).

5 Conclusion

This work presents a reinforcement learning based scheduling framework, which imitates the behaviors of optimal optimization algorithms in inference quality. Our framework has demonstrated up to $\sim 2.5\times$ runtime speedups over the commercial Edge TPU compiler, using ten popular ImageNet models on three different physical pipelined Google Edge TPU systems. More importantly, compared to the exact optimization methods solved by heuristics and brute-force, the proposed RL scheduling improves the scheduling runtime by several orders of magnitude.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, 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.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. Mitigating edge machine learning inference bottlenecks: An empirical study on accelerating google edge models. *arXiv preprint arXiv:2103.00768*, 2021.

Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 1443–1451, 2019.

Hongzheng Chen and Minghua Shen. A deep-reinforcement-learning-based scheduler for fpga hls. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8. IEEE, 2019.

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)*, pp. 578–594, 2018.

Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, pp. 6281–6292, 2019.

François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258, 2017.

Luca M Gambardella and Marco Dorigo. Ant-q: A reinforcement learning approach to the traveling salesman problem. In *Machine Learning Proceedings 1995*, pp. 252–260. Elsevier, 1995.

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*, pp. 770–778, 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pp. 630–645. Springer, 2016b.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Andre Hottung, Shunji Tanaka, and Kevin Tierney. Deep learning assisted heuristic tree search for the container pre-marshalling problem. *Computers & Operations Research*, 113:104781, 2020.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

Daniel Karapetyan, Abraham P Punnen, and Andrew J Parkes. Markov chain methods for the bipartite boolean quadratic programming problem. *European Journal of Operational Research*, 260(2):494–506, 2017.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pp. 6348–6358, 2017.

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

Krzysztof Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(3):355–383, 2003.

Christophe Lenté, Mathieu Liedloff, Ameur Soukhal, and Vincent t'Kindt. Exponential algorithms for scheduling problems. 2014.

Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *Top*, 25(2):207–236, 2017.

Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pp. 270–288. 2019.

Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pp. 9839–9849, 2018.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

Shuran Sheng, Peng Chen, Zhimin Chen, Lenan Wu, and Yuxuan Yao. Deep reinforcement learning-based task scheduling in iot edge computing. *Sensors*, 21(5):1666, 2021.

Jialin Song, Ravi Lanka, Yisong Yue, and Masahiro Ono. Co-training for policy learning. *arXiv preprint arXiv:1907.04484*, 2019.

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Amir Yazdanbakhsh, Kiran Seshadri, Berkin Akin, James Laudon, and Ravi Narayanaswami. An evaluation of edge tpu accelerators for convolutional neural networks. *arXiv preprint arXiv:2102.10423*, 2021.

Bo Yu, Lu Yang, and Fang Chen. Semantic segmentation for high spatial resolution remote sensing images based on convolution neural network and pyramid pooling module. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 11(9):3252–3261, 2018.

A APPENDIX

CODE AND DATASET

5 folders: 4 for running code, 1 for dataset;

Folder:

LSTM-LSTM: encoder as LSTM, decoder as LSTM;
LSTM-Transformer: encoder as LSTM, decoder as Transformer;
Transformer-LSTM: encoder as Transformer, decoder as LSTM;
Transformer-Transformer: encoder as Transformer, decoder as Transformer

Running code:
Entering into each running folder:
run:

```
CUDA_VISIBLE_DEVICES=0 python run.py
--train_dataset_path
    train.pt (generate training dataset in dataset folder)
--eval_dataset_path
    eval.pt (generate testing dataset in dataset folder)
```

For other experiments, please generate related dataset using:

```
dataset/dataset_generator.py (ASAP)
dataset/dataset_generator_scheduling_reversed.py (ALAP)
```

The command in dataset generator is:

```
myDataset = TopoSortDataset(size=50, num_samples=10240, in_degree_fixed=3,
in_degree_total=6, resource_constraint_level=eval_lvl, level_range=[16, 40],
weight_multiply=5., weight_constraint=35.)

size: number of nodes;
num_samples: number of graphs in the dataset;
in_degree_fixed: incoming edges per node;
in_degree_total: maximum incoming edges embedding can hold;
resource_constraint_level: default
level_range: depth of graphs;
weight_multiply: memory for each node;
weight_constraint: pipeline memory constraint.
```

x