Preprints.org

Article

Not peer-reviewed version

Analyzing Convolutional Neural Network Performance on an Edge TPU with a Focus on Transfer Learning Adjustments

Christian DeLozier * , Justin Blanco , Ryan Rakvic , James Shey

Posted Date: 25 October 2023

doi: 10.20944/preprints202310.1612.v1

Keywords: machine learning; IoT; performance; energy; neural networks

Preprints.org is a free multidiscipline platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Analyzing Convolutional Neural Network Performance on an Edge TPU with a Focus on Transfer Learning Adjustments

Christian DeLozier [1,*,†], Justin Blanco [1], Ryan Rakvic [1] and James Shey [1]

[1] United States Naval Academy; delozier@usna.edu, blanco@usna.edu, rakvic@usna.edu, shey@usna.edu
* Correspondence: delozier@usna.edu;
† Current address: 105 Maryland Ave, Annapolis, MD, USA

Abstract: Transfer learning has proven to be a valuable technique for deploying machine learning models on edge devices and embedded systems. By leveraging pre-trained models and fine-tuning them on specific tasks, practitioners can effectively adapt existing models to the constraints and requirements of their application. In the process of adapting an existing model, a practitioner may make adjustments to the model architecture, including the input layers, output layers, and intermediate layers. In this study, we examine the effects of these adjustments on the runtime and energy performance of an edge processor performing inferences. Based on our observations, we make recommendations for how to adjust convolutional neural networks during transfer learning to maintain runtime performance. We observe that the Edge TPU is generally more efficient than a CPU at performing inferences on convolutional neural networks, and continues to outperform a CPU as the depth and width of the convolutional network increases. We explore multiple strategies for adjusting the input and output layers of an existing model and demonstrate important performance cliffs for practitioners to consider when modifying a convolutional neural network model.

Keywords: machine learning; IoT; performance; energy; neural networks

---

0. Introduction

Convolutional Neural Netorks (CNNs) are a powerful tool for solving a variety of problems using deep learning techniques, demonstrating exceptional performance in tasks such as image classification, object detection, and segmentation. Their ability to automatically learn hierarchical features from raw data has revolutionized various industries, from healthcare to autonomous vehicles. However, training deep CNNs from scratch demands vast amounts of labeled data and computational resources, making applying them difficult for many real-world applications.

Transfer learning addresses this challenge by leveraging models pre-trained on large datasets and adapting them for specific tasks with limited labeled data. This approach not only significantly reduces the data requirements but also accelerates convergence during training. During the process of applying transfer learning to a CNN, a practitioner may wish to tweak the neural network architecture to further fit the targeted application. Changes to the neural network architecture may affect both the accuracy of the network and the runtime performance of the network executing on a device.

To meet the intense processing demands of neural networks, Google developed the Tensor Processing Unit (TPU), an integrated chip designed specifically for machine learning with neural networks. The TPU is powered by a matrix multiply unit, allowing it to run parallel computations. The original TPU was designed for data centers and was not optimized for energy efficiency [7]. The Coral Edge TPU is a low power option intended for embedded system machine learning inference. The *Edge* component of the name conveys that the device does not need to rely on a cloud server and instead is able to process data locally [2]. It is not as fast as Google's original Cloud TPU, but well-suited to on-device machine learning. While the device is capturing data, it is able to analyze and process the data at the source, instead of sending it off-device to be processed [1]. Many applications, including IoT security [4] and wildlife behavior monitoring [11], benefit from edge machine learning.

© 2023 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

We utilized Google's Edge TPU for the experiments described in this paper. Specifically, we used the Coral Development (Dev) Board, which, in addition to a TPU, incorporates a CPU, sensors, and devices that can be utilized for edge machine learning applications. Our setup can be seen in Figure 1 and it consists of a Laptop running Linux, a Gen7i data acquisition system [15], two external power supplies, and the Coral Dev Board. In this paper, we analyze the runtime performance and energy usage of the Edge TPU compared to a mobile CPU on a set of neural network models designed to highlight the strengths and weaknesses of the Edge TPU.

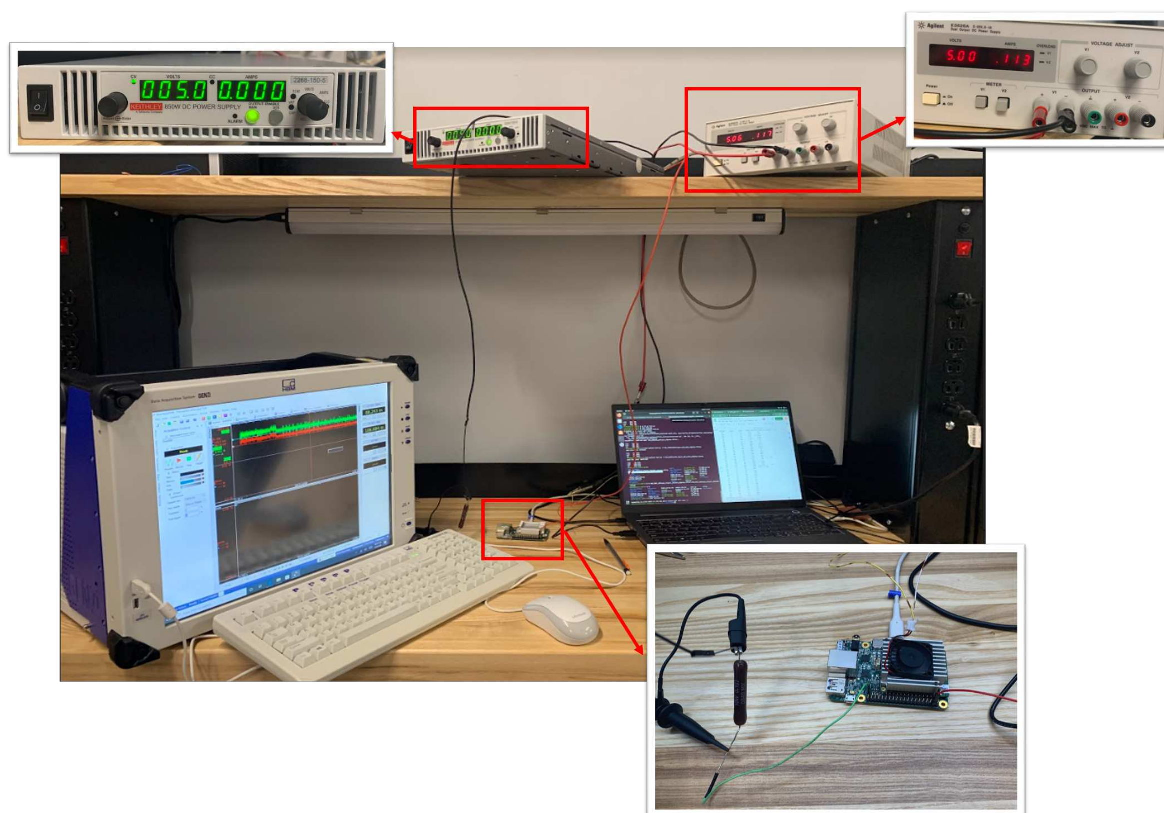

Figure 1. Experimental setup including (counterclockwise from top right to bottom right) a 5V, 1A power supply to power the development board's fan, a 5V, 3A power supply to power the development board, a data recorder, the Coral development board, and a laptop for running commands on the development board.

We evaluated the performance of the Edge TPU compared to a mobile CPU with specific interest in convolutional neural networks that have been modified as part of the process of transfer learning. We started by evaluating the runtime and energy performance of the Edge TPU compared to the mobile CPU on a set of baseline convolutional neural networks. We then evaluated modified versions of a subset of the convolutional neural networks to model the tweaks that might be made by a practitioner during the process of applying transfer learning.

This paper makes the following contributions:

- Proposes a methodology for determining the limits of neural network performance on edge devices
- Analyzes the performance, both runtime and energy, of an edge TPU on both fully connected and Convolutional Neural Networks as compared to a mobile CPU
- Assesses the performance impact of modifications made to convolutional neural networks as part of transfer learning

The remainder of this paper is organized as follows. Section 1 presents background material and related work on deep neural networks and tensor processors. Section 2 discusses the methodology

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

for our experiments. Section 3 presents the results of the experiments that we conducted. Section 4 provides practical recommendations for neural network designs targeting Edge tensor processors. Finally, Section 5 concludes the paper.

1. Background and Related Work

In the following sections, we discuss the relevant background on neural network architectures and tensor processing units. We also highlight related work.

1.1. Deep Neural Networks

Deep neural networks offer promise as flexible, nearly "off-the-shelf" solutions to machine learning problems that can perform adequately even for non-expert users or those who lack significant technical domain knowledge. [18,19]. By using deep neural networks, less experienced practitioners can apply machine learning to their domain-specific problem. Common deep neural network structures include fully connected neural networks and convolutional neural networks.

Neural networks (a.k.a. artificial neural networks) are a type of machine learning model inspired by the biological neurons in the human brain [20]. They comprise multiple layers of interconnected nodes, known as neurons, that work together to process information and make predictions. In a feed-forward neural network, information flows in one direction, from the input layer through one or more hidden layers, to the output layer. In a fully connected feed-forward neural network, each neuron in a layer receives input from all of the neurons in the previous layer, performs a calculation using weights and biases, and then passes the result to the neurons in the next layer. By adjusting the weights and biases of the neurons, the network can learn to recognize patterns in data and make accurate predictions. Fully connected neural networks have been successfully applied to a wide range of tasks, including image recognition, speech recognition, and natural language processing.

Convolutional neural networks (CNNs) are a type of deep (i.e., large number of layers) neural network model that excels in image processing tasks [21], among others. The name convolutional comes from the use of convolutional kernels. A kernel is feature map that represents each node in a given layer as its weighted inputs from the same number and arrangement of neurons in the previous layer. In other words, the inputs to each node differ only by the shifting of a common weight vector (and bias term) at the previous layer. CNNs were inspired by the way the visual cortex in the brain processes visual information. A typical CNN consists of multiple layers of interconnected neurons, including convolutional layers, pooling layers, and fully connected layers. For an image processing CNN, in a convolutional layer, a set of filters is applied to the input image, the effect of which is to extract features such as edges and textures. The output of the convolutional layer can then be passed through a pooling layer to reduce the dimensionality of the features and make the model more efficient. Finally, the output of the pooling layer is typically fed into one or more fully connected layers, which perform classification or regression on the extracted features. CNNs have been shown to be very effective at a variety of image processing tasks, including object recognition, face detection, and image segmentation. They have also been applied in other domains such as natural language processing and speech recognition.

Transfer learning is a technique used in machine learning where a pre-trained model is used as a starting point for a new task [17]. Instead of training a model from scratch, which can be computationally expensive and require large amounts of labeled data, a pre-trained model is fine-tuned to the new task by updating its weights and biases using a smaller dataset. The intuition behind transfer learning is that the features learned by the pre-trained model on a large dataset are likely to be useful for a new, related task, even if the input datasets are not identical in nature. Transfer learning has been successfully applied in a wide range of domains, including computer vision, natural language processing, and speech recognition. It is particularly useful when there is limited labeled data available for the new task or when training from scratch would take too long. Transfer learning is commonly performed by replacing the input and output layers of the pre-trained model with layers

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

that fit the target problem. Small changes may also be made to the existing model in an attempt to improve accuracy on the target problem.

1.2. Tensor Processing

Tensor Processing Units are domain-specific hardware specifically designed for neural networks, with extensive optimization for matrix multiplication [5–7,12,14]. For this reason, they often outperform CPU and GPU architectures in speed while minimizing power consumption. When compared against the Haswell CPU and NVIDIA K80 GPU, for example, the TPU used the lowest power per die but had the highest energy per area, while the CPU, although it had the highest power usage, had the best energy proportionality. The TPU had a 14-16 times better performance per watt than the NVIDIA K80 GPU and 17-34 times better than the Haswell CPU [7]. Furthermore, algorithms have been proposed to utilize the embedded processor with lower voltage and frequency without compromising runtime performance [12].

To perform inferences using neural networks, the primary task of the Edge TPU is matrix processing. Therefore, the chip design features thousands of multiply-accumulate units in a so-called *systolic array* [1]. In contrast, a CPU, even with vector instructions, can only execute a small number of add or multiply instructions per cycle. For convolutional neural networks, which generally require a large number of multiply-adds per inference, the TPU architecture greatly accelerates the computation.

The Coral Edge TPU provides on-device, low-power hardware for AI applications that can perform around four trillion operations per second using 0.5 W for each tera-operation per second [6,14]. Benchmarking tests of the Coral Dev board on various model architectures show that the on-board TPU inference time far outperforms its CPU counterpart by a magnitude of at least 5 times, ranging up to 100 times faster for certain network models [6,10].

Given the high interest in edge machine learning, prior work has studied the performance of edge tensor processors. In [14], the authors evaluate the performance of convolutional networks from the NASBench-101 benchmark suite on three Edge tensor processors with the goal of training a performance and energy model for exploring new tensor processor architectures. In a similar study [35], the authors evaluate fully-connected and convolutional neural networks on an edge TPU. DeepEdgeBench [10] evaluates five edge processors on the MobileNtV2 benchmark. Prior work has also evaluated the performance of edge tensor processors in the context of specific applications, including network intrusion detection [36], animal activity classification [34], and object classification [37]. Our work differs from prior studies by generating modified network models based on real-world examples and evaluating transfer learning techniques.

2. Methodology

We measured the runtime performance and the energy usage of the Coral Edge TPU for various deep neural network architectures. We compared the runtime performance of the Coral Edge TPU to the mobile CPU included with the Coral development board. The specifications of these devices can be found in Table 1.

Table 1. Comparison of specifications for the processors used in this study.

	Coral TPU [8]	Coral CPU [8,9]
Processor	Google Edge TPU	Cortex-A53 Quad-core
Frequency	480 MHz	3.01 GHz
RAM	4 GB DDR4	4 GB DDR4
Operation Type	Fixed Point	Floating Point
Operations/s	4 Trillion (8-bit)	32 Billion (32-bit)

For runtime performance measurements, we recorded the runtime of the `interpreter.invoke()` method from the `tflite` Python library using Python's `time.perf_counter_ns()` method. This

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

performance measurement captures the runtime performance of an inference using the machine learning model. For all experiments, we recorded the runtime of 10,000 inferences, and we report the average runtime for a single inference in each graph.

For energy measurements, we powered the Coral development board with a 5V, 3A power supply. We connected the ground pin via a 0.1 ohm resistor to a Gen7i data acquisition system that includes a high-resolution oscilloscope (as seen in Figure 1). The data acquisition system measured the voltage across the resistor at a sampling rate of 100 kHz. The data acquisition system begins taking samples based on a trigger from a GPIO pin on the development board that is set to high just prior to invoking the inference. The GPIO pin is set to low as soon as the inference ends. The energy required to complete an inference, E, is then computed as in (1), where v_S is the supply voltage, R is the resistance, v_R is the voltage across the resistor, f_s is the sampling rate, and K is the duration, in samples, required to complete the inference.

$$E = \sum_{k=1}^{K} (v_S v_R[k] - v_R^2[k]) \frac{1/f_s}{R} \quad (1)$$

Figure 2 shows sample voltage traces for the CPU and TPU on the *MobileNet1.0* convolutional neural network. These traces are processed using a Matlab script that outputs the total energy (J) recorded during the inference.

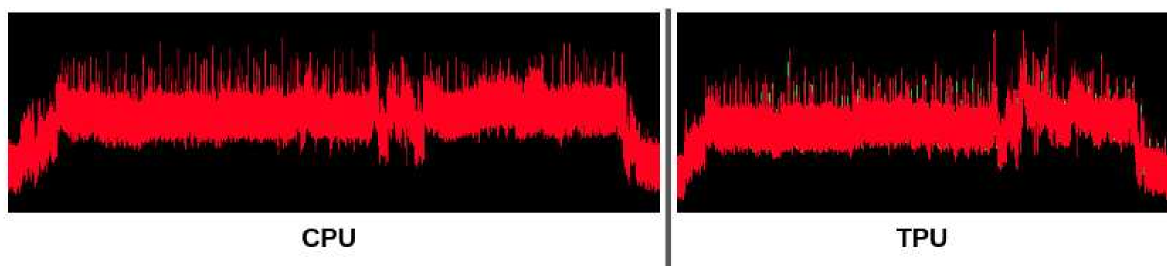

Figure 2. Sample power traces for the CPU and TPU on the MobileNet1.0 CNN model.

2.1. Convolutional Neural Networks

For baseline experiments on convolutional neural networks, we started with a set of CNN models, described in Table 2, built for the edge TPU [16]. These models use deep neural networks to assist in image classification, object detection, and semantic segmentation. Many of these models are modified versions of the same base models: EfficientNet, Inception, and MobileNet.

Table 2 provides metrics that give insight into the performance impact of performing an inference with these neural networks. *GFLOP* shows the total number of floating point operations required to execute the model. *GFLOP* indicates how much total work must be performed by a processor to perform an inference, from input to output, with each model. We calculated the floating-point operations per model by executing the model, profiling the runtime per layer, and multiplying the runtime by the floating-point operations per second for the processor. *GFLOP* is not a perfect analog of the total runtime required to perform an inference with each model because, as shown in Figure 3, models differ in terms of how many layers can be executed in parallel throughout the model. For example, the `MobileNet1.0` computes an inference using a serial chain of layers, while the `InceptionV1` model computes an inference using multiple layers in parallel. These parallel layers may use different filters to extract different characteristics from the input data. The amount of parallel work in a model also depends on the sizes of the inputs, filters, and other parameters for each layer. For their Cloud TPUs, Google recommends tiling data into 128x8 chunks [3]. If the computation does not exactly fit that chunk size, the compiler will pad the tensors to match. This can lead to increases in the amount of memory required to store a tensor.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

Table 2. Characteristics of Baseline CNN Models. FNN240L810N is provided as a comparison point for fully connected networks.

Model Name	Input	Output	GFLOP	Layers	% Parallel Layers
EfficientDet320 [39]	320x320x3	90	2,323.1	266	36%
EfficientDet384 [39]	384x384x3	90	4,272.3	321	31%
EfficientDet448 [39]	448x448x3	90	6,806.7	356	29%
EfficientDet512 [39]	512x512x3	90	13,117.4	423	29%
EfficientDet640 [39]	640x640x3	90	23,671.8	423	29%
EfficientNetS [40]	244x244x3	1000	2,991.1	66	0%
EfficientNetM [40]	240x240x3	1000	4,598.3	86	0%
EfficientNetL [40]	300x300x3	1000	11,752.0	97	0%
InceptionV1 [41]	244x244x3	1000	2,167.2	83	53%
InceptionV2 [32]	244x244x3	1000	2,708.2	98	46%
InceptionV3 [32]	299x299x3	1000	7,347.8	132	47%
InceptionV4 [28]	299x299x3	1000	15,666.1	205	32%
MobileDetSSDLite [42]	320x320x3	90	2,437.3	136	32%
MobileDetV1 [44]	300x300x3	90	1,929.1	75	44%
MobileDetV2Coco [42]	300x300x3	90	1,494.4	110	30%
MobileDetV2Face [42]	320x320x3	90	1,524.6	132	25%
TF2MobileDetV1 [46]	640x640x3	90	67,482.4	104	56%
TF2MobileDetV2 [46]	300x300x3	90	1,407.5	101	24%
DLV3DM05MobileNet [47]	513x513x3	20	2,276.7	72	0%
DLV3MobileNet [47]	513x513x3	20	5,343.5	72	0%
KerasMobileNet128 [45]	128x128x3	37	1,350.8	76	10%
KerasMobileNet256 [45]	256x256x3	37	5,390.1	76	10%
MobileNet0.25 [44]	128x128x3	1000	37.8	31	0%
MobileNet0.5 [44]	160x160x3	1000	155.1	31	0%
MobileNet0.75 [44]	192x192x3	1000	412.1	31	0%
MobileNet1.0 [44]	224x224x3	1000	912.6	31	0%
MobileNetV2Bird [42]	224x224x3	900	652.3	65	0%
MobileNetV2Plant [42]	224x224x3	2000	658.6	65	0%
MobileNetV2 [42]	224x224x3	1000	652.3	66	0%
TF2MobileNetV1 [46]	224x224x3	1000	840.3	33	0%
TF2MobileNetV2 [46]	224x224x3	1000	614.8	68	0%
TF2MobileNetV3 [46]	224x224x3	1000	1,280.9	79	0%
FNN240L810N [38]	*100x1*	*9*	*565.3*	*242*	*0%*

The *Layers* column shows the total number of high-level Tensorflow operations performed by each model, and the *% Parallel Layers* column shows the number of these operations, or layers, that can be executed in parallel. We calculated the number of Parallel Layers by traversing the graph and counting the steps required to execute the entire graph under the assumption that if the inputs to a layer were ready, the layer could be executed. We note that this calculation assumes an infinitely large matrix multiplication unit that can fit the entire calculations required for multiple layers concurrently. For example, in Figure 3, the `InceptionV1` model could execute as follows. First, the *MaxPool2D* layer executes. Once the *MaxPool2D* layer finishes, the next three *Conv2D* operations and the next *MaxPool2D* operation can execute in parallel using the output from the first *MaxPool2D* layer. At this point, the output from the leftmost chain is ready for the *Concatenation* operation, but the rest of its inputs are not ready, so it must wait. The three remaining *Conv2D* operations can execute, and, finally, the *Concatenation* operation can execute once all of its inputs are ready. In total, the 9 layers in this part of the model will execute in 4 steps. Therefore, we would calculate that this part of the model has $(9-4)/9 = 55.6\%$ Parallel Layers. In Table 2, we see that the total *% Parallel Layers* for `InceptionV1` is slightly lower, at 53%, because other parts of the model have less parallel work available. We also note that speculative execution techniques may be able to exploit additional parallelism not considered by this calculation.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

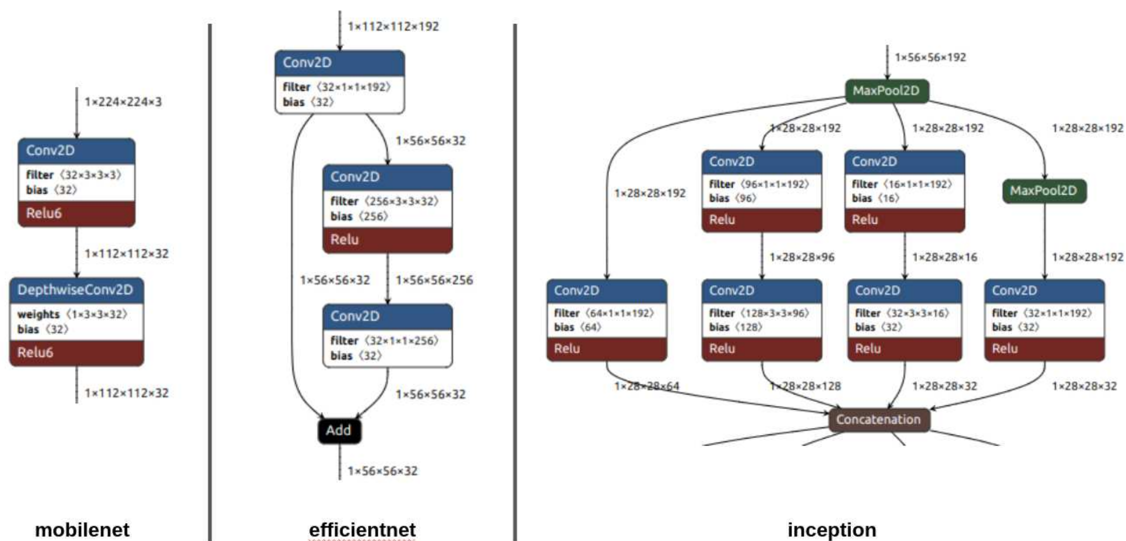

Figure 3. Main subgraphs for EfficientNetS, InceptionV1, and MobileNet1.0

In combination, the *Layers* and *% Parallel Layers* columns indicate how *deep* or *wide* the baseline models are. A *deeper* model requires more serial steps to perform an inference. For example, `MobileNet1.0` requires 31 steps, and `EfficientDet320` requires $266 \times .36 \approx 96$ steps. Therefore, we would consider `EfficientDet320` to be a deeper model than `MobileNet1.0`. A *wider* model performs more work per step. This can be derived from both *% Parallel Layers*, which shows how many of the layers can be executed in parallel, and by dividing *GFLOP* by *Layers* to find, on average, how many floating-point operations are performed per layer. For example, `DLV3MobileNet` is wider than `DLV3DM05MobileNet` because it requires more floating-point operations for the same number of total layers, which indicates that the layers must perform more work. This difference is due to the use of 2x larger filters in `DLV3MobileNet`.

Overall, the baseline models that we examined cover a variety of input and output sizes, total number of floating-point operations required to perform an inference, and model architectures in terms of serial versus parallel work. As a reference point, we also provide these metrics for a fully-connected feed-forward neural network with 240 layers and 810 nodes per layer (*FNN240L810N*). In general, performing an inference with this feed-forward network requires fewer floating-point operations and has less parallel work available, compared to the CNN models.

2.2. Exploring Adjustments to CNN Models

We analyzed the structure of the CNN models to identify common modifications to produce different versions of the same model. In many cases, the baseline model features a repeated subgraph of convolution operations, as shown in Figure 4. Deeper versions of the model repeat this subgraph in order to extend the model. Wider versions of the model add convolution or other operations to the subgraph. Aside from additional convolutions, models may also add a fully-connected layer at the end of the CNN.

Starting with a subset of the CNN models, we generated deeper, wider, and otherwise modified versions of these CNN models to evaluate the performance impact of such modifications. For our experiments, we used the *EfficientNetS*, *InceptionV1*, and *MobileNet1.0* models as a baseline.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

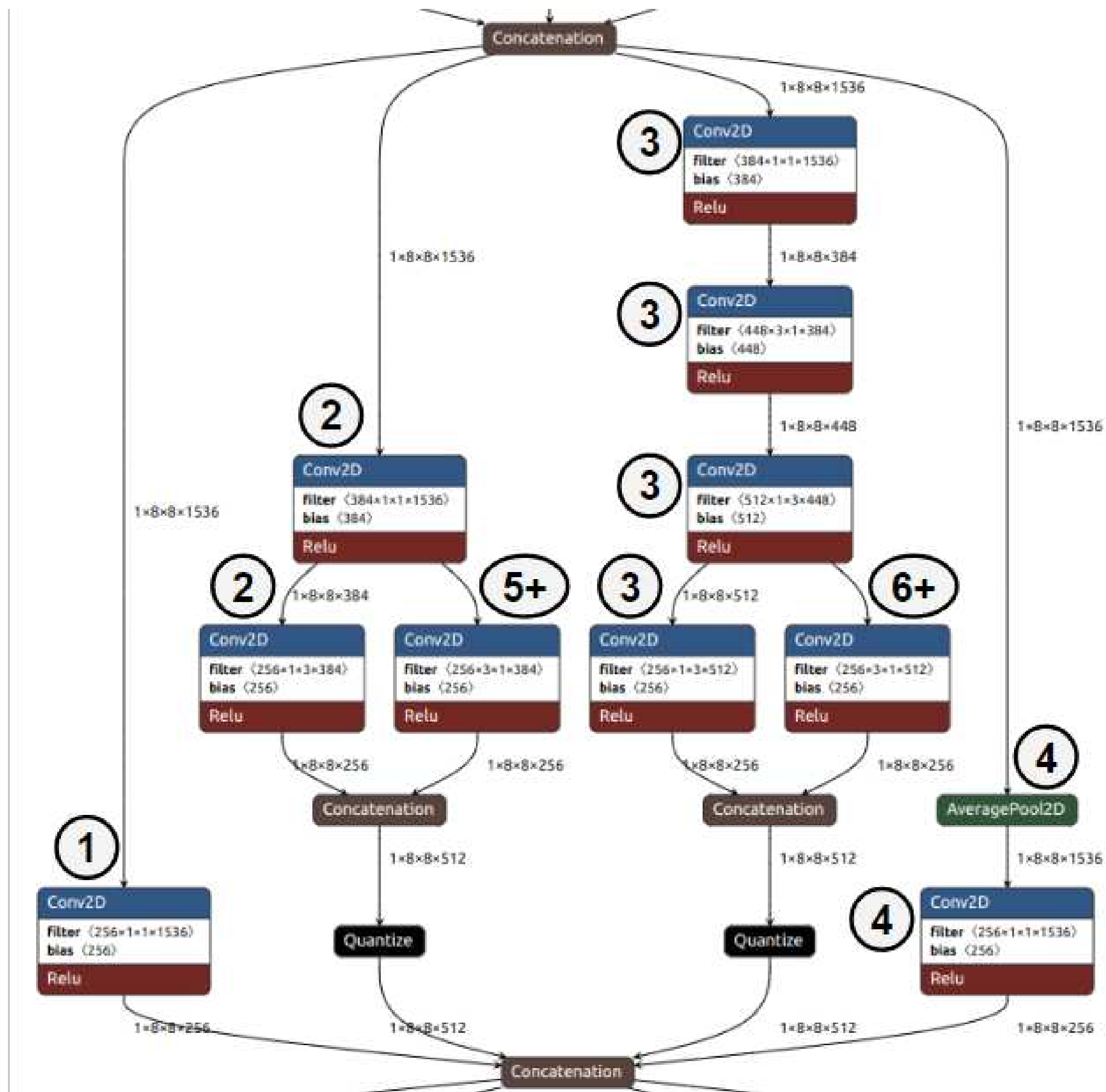

Figure 4. Structure of the main subgraph for InceptionV4. The numbers indicate which layers were added expansions to subgraphs for our transfer learning experiments.

2.2.1. Extracting the Baseline Models from Tensorflow Lite

To create deeper and wider CNNs for performance analysis, we first needed to extract the baseline models into a modifiable format because pre-existing Tensorflow Lite models, the model format required by the Edge TPU, are not easy to modify. In practice, a model designer will generate a Tensorflow Lite model from a Tensorflow model or by converting from another model format. We extract a modifiable model from the Tensorflow Lite model in two steps. First, we run `Analyzer.analyze` from the Tensorflow Lite Python library to extract the model architecture. This tool provides both the order and types of layers in the model and the input and output tensor sizes for each layer. However, this tool does not provide all of the required information to reproduce the model, including the filter sizes and strides. Next, we run `flatc`, which is the FlatBuffer compiler, to produce a `json` file with specific model parameters, including filters and strides. We combine these two sources of information in a Python script that generates a new model using Keras to match the input Tensorflow Lite model. For each extracted model, we verified that the extracted model's performance and energy characteristics match the original model.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

2.2.2. Generating Deeper Models

Once we had the extracted model, we created deeper models by identifying the main repeated subgraph within the original model and further repeating that subgraph. We created shallower models by removing repetitions of the subgraph. In practice, both the shallow and deep versions of each model were created in a single run of our Python script that removed all of the repeated subgraphs from the original model and then re-added one subgraph at a time to generate models with zero to N repeated subgraphs.

To create shallower and deeper versions of these models, we used the subgraphs shown in Figure 3. In *EfficientNetS*, the main subgraph is a 2D convolution that is added to the result two further 2D convolutions. The main subgraph in *IncepionV1* performs parallel 2D convolutions with different filter sizes. In one of the parallel branches, a 2D max pooling operation is performed. For *MobileNet1.0*, the main subgraph is a 2D convolution followed by a depthwise 2D convolution.

2.2.3. Generating Wider Models

We explored multiple avenues for generating wider models from the baseline CNNs by both increasing the number of layers that could be executed in parallel and increasing the total amount of work per layer.

To increase the number of layers, we drew inspiration from the evolution of the Inception model from `InceptionV1` to `InceptionV4`. Figure 4 shows the main subgraph of `InceptionV4`. `InceptionV1`, `InceptionV2`, and `InceptionV3` all use portions of this subgraph. Each layer in the subgraph attempts to derive additional information from the data by using different bias and filter sizes.

For experiments on increasing the number of parallel layers, we created wide versions of the `EfficientNetS` and `MobileNet1.0` models because these models have 0% parallel layers in the original model. We did not expand the other baseline model from previous experiments (`InceptionV1`) using this methodology because it already has multiple parallel layers. We expanded existing 2D convolution layers in the numerical order shown in Figure 4. To further explain, the baseline version of the model only had the original 2D convolution (1). The first expansion of the layer adds two 2D convolutions in parallel with the original 2D convolution (2). The second expansion adds a chain of four 2D convolutions in parallel (3). The third expansion adds an average pooling operation followed by a 2D convolution (4). Finally, additional expansions add parallel layers in the two middle subgraphs (5+ and 6+).

We also examined the performance impacts of increasing the total amount of work per layer by expanding the dimensions of the 2D convolutions throughout the baseline models. We attempted to align these experiments with common transfer learning techniques. First, we widened the model at two points: after the input layer and before the output layer. We also performed an experiment with wider layers at both the input and output layers. Second, we scaled up the width of the entire model. We performed these experiments on all three of the baseline models (`EfficientNetS`, `MobileNet1.0`, and `InceptionV1`).

3. Experimental Results

We performed a number of experiments in order to analyze the runtime performance and energy usage of the CPU and TPU on convolutional neural networks. We first examine the runtime and energy performance of the baseline CNNs on the CPU and TPU and characterize the performance based on the structure of the network. We then examine the performance impact of modifications to a set of CNNs that might be applied to a model as part of the process of transfer learning. We examine transformations such as the input and output sizes, adding a fully-connected layer after the input, and making the models deeper and wider. All experiments were performed on the Coral Edge TPU development board.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

3.1. Convolutional Neural Networks

We measured the runtime performance of the Edge CPU and Edge TPU using the baseline models described in Table 2. Figure 5 shows the runtime speedup for a single inference on the Coral development board compared to the single core baseline. The baseline measurement was performed on a single CPU core. We then measured the runtime performance on an inference using 4 CPU cores and the TPU. As shown, the TPU consistently outperforms the CPU, even with 4 CPU cores, on all of the CNN models. The rightmost bars (*geomean*) show the geometric mean of all speedups for the 4-core CPU and the TPU. On average, these CNNs execute an inference on the 4-core CPU in 33% of the time it takes to execute the same model on a 1-core CPU. On the TPU, it takes 10% of the time it takes to execute the same model on a 1-core CPU.

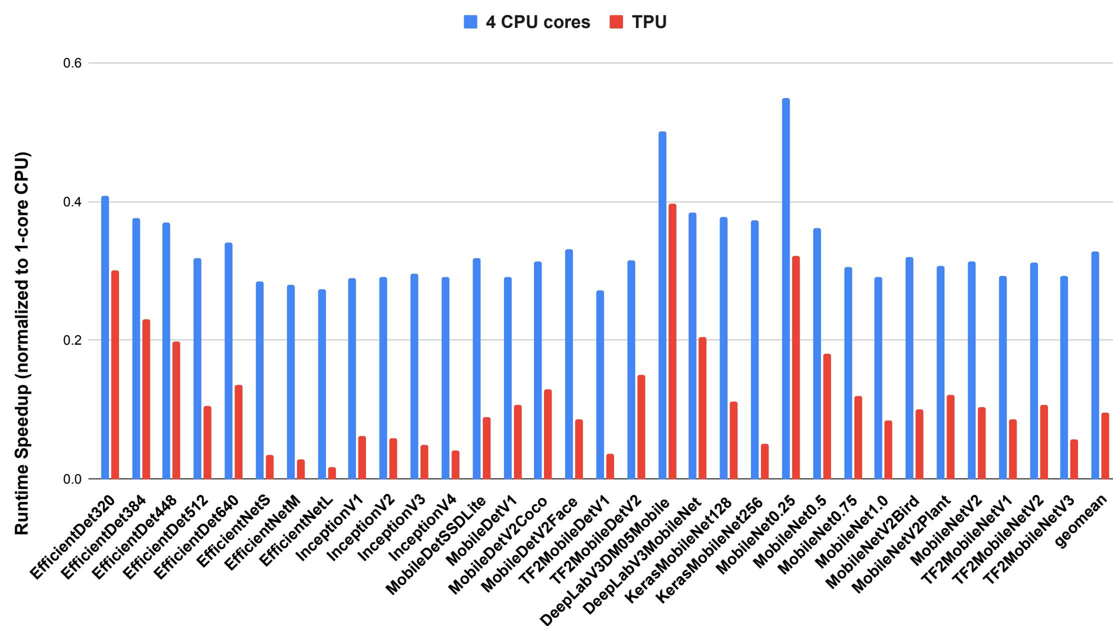

Figure 5. Runtime performance (normalized to 1-core baseline) of convolutional neural networks on the edge CPU with 1 and 4 cores and on the edge TPU

We also compared energy per inference for the CNN models. The results of this experiment are shown in Figure 6. The energy results are similar to the runtime results for the CNN models, with the TPU consistently outperforming both the 1 core and 4 core CPU. One notable difference is that the 4-core CPU only uses 45% less energy to perform an inference than the 1-core CPU. Given that the 4-core CPU executes an inference in 33% of the time it take to execute on the 1-core CPU, the power consumption of 4 cores slightly outweighs the runtime speedup from running on 4 cores. However, the significant improvement in runtime performance still leads to lower energy usage per inference. In low-power environments, using a 1 core CPU will provide better long-term energy usage if inferences are being run frequently. The TPU uses 10% of the energy required to perform an inference on the 1-core CPU.

Figure 7 breaks down the performance speedup of the TPU over the single core CPU with reference to the floating-point operations per parallel layer for each model. In general, more available parallel work leads to a larger performance improvement on the TPU. However, other factors prevent this speedup from being monotonic compared to the available parallel work. The TPU's matrix multiplication unit may not be completely utilized at all times due to the fixed hardware structure (128x128 on TPU version 3) and various sizes of filters used in these CNNs. Furthermore, the TPU must load model parameters from memory, which takes additional time, especially for larger models. Other

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

hardware factors, such as cache line sizes, associativity, and prefetching, may also impact performance. Overall, we find that floating-point operations per parallel layer is a reasonable, though certainly not perfect, indicator of the runtime performance of a CNN on the Edge TPU.

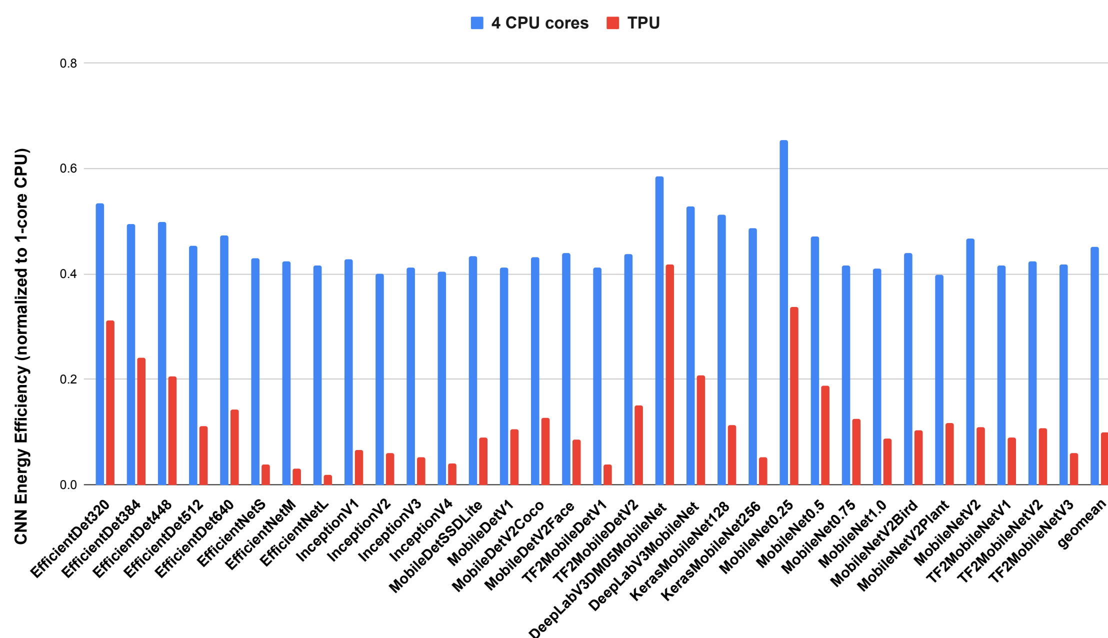

Figure 6. Energy per inference of convolutional neural networks on the edge CPU with 1 and 4 cores and on the edge TPU

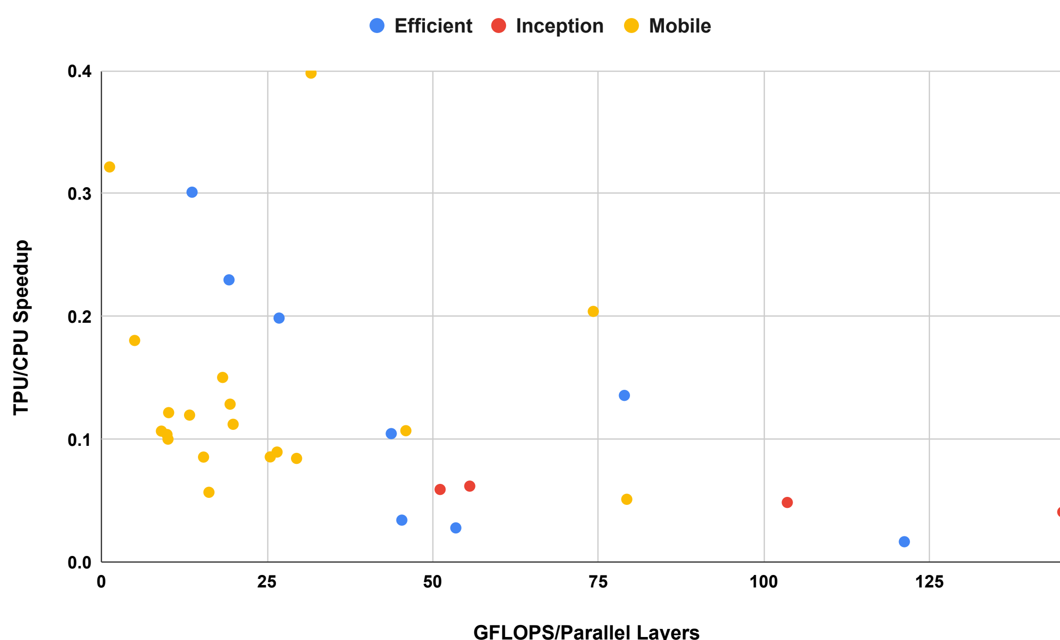

Figure 7. TPU runtime speedup compared to floating-point operations per parallel layer

3.2. Transfer Learning

Transfer learning involves applying an existing model, potentially with small modifications, to a new problem. This may require modifying the input and output layers to match the new problem's

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

inputs and outputs. In some cases, practitioners may wish to make small modifications to the existing model to improve its accuracy on the new problem. We are interested, specifically, in the runtime performance impact on inferences performed with modified CNN models. We do not evaluate the accuracy of such models or the impact on training time. In the following sections, we evaluate the runtime performance metrics of modifications that might be made to a model while applying transfer learning on the edge TPU.

3.3. Input and Output Size

Transfer learning often requires changing the input size of the machine learning model to match the target problem's input characteristics. We perform two experiments to assess the performance impact of modifying the input size. For problems with image inputs, we simply vary the size of the input by resizing the image. We chose to scale the image inputs by factors of 2 (.25x, .5x, 1x, 2x, and 4x). Figure 8 shows the performance impact of resizing the input images to the CNN models. As the image input size grows, the TPU speedup over the CPU decreases.

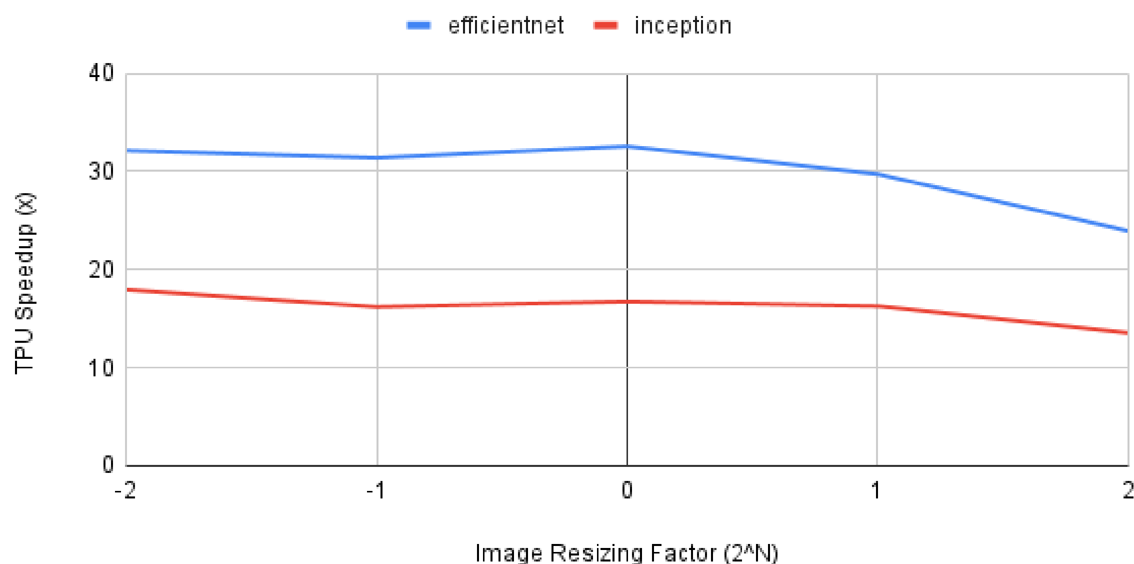

Figure 8. Performance impact of varying image input size to CNNs

For problems with 1 dimensional inputs, we apply a fully-connected layer to expand the number of input parameters to more closely match the expected number of inputs from an image. We then reshape the 1 dimensional data into 2 dimensional data with 3 channels. Finally, we resize the shaped inputs to match the expected image size for the model. Figure 9 demonstrates this procedure. With powerful enough hardware to train the models, it may be possible to skip the resize operation and simply reshape the output of the fully-connected layer. On our hardware, we were unable to produce a working Tensorflow Lite model with a fully-connected layer that could be reshaped to the 224x224x3 input of the models [1].

[1] $224 \times 224 \times 3 = 150528$ nodes in the fully-connected layer

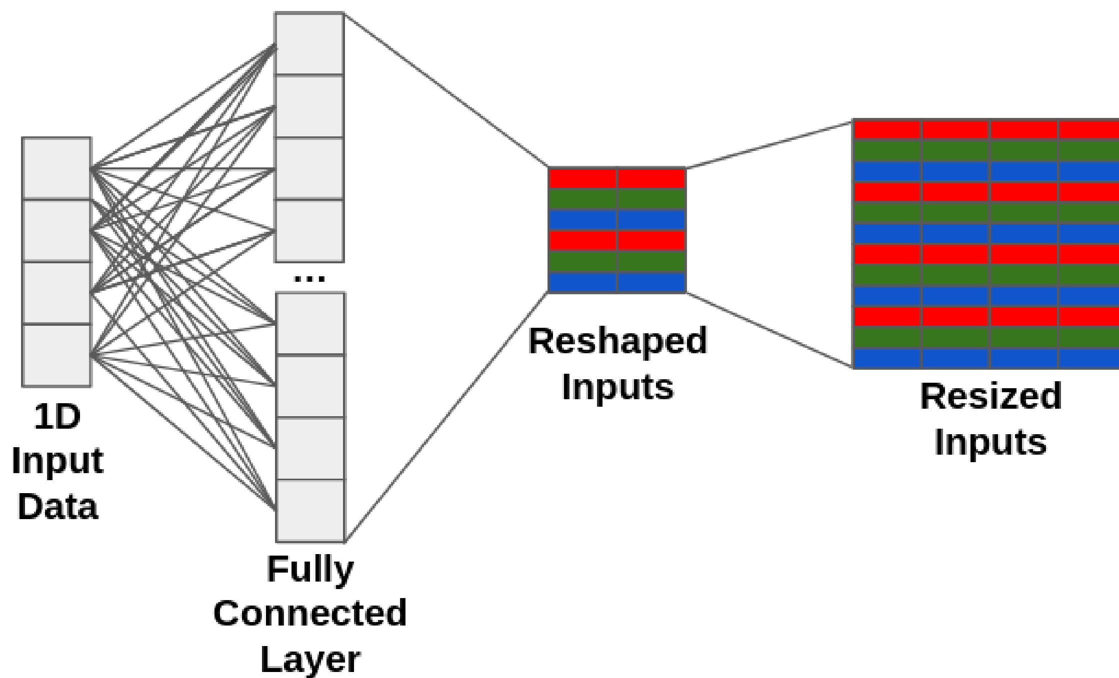

Figure 9. Adding a fully-connected layer to map 1D input data to a 2D CNN

Table 3 shows the speedup of the TPU over the CPU on the CNN models using a fully-connected layer to expand the 1D inputs into the 2D image size expected by the CNN. As shown, with a small fully-connected layer, the performance benefit of the TPU on a CNN outweighs the work required to execute the fully-connected layer. As the size of the fully-connected layer increases, the CPU begins to outperform the TPU, despite the TPU's performance advantage on the CNN.

Table 3. TPU Speedup over CPU on CNN model execution with a Fully-Connected layer to expand input size. All models expect a 224x224x3 image input.

Model	TPU Speedup (14 FC Nodes)	TPU Speedup (224 FC Nodes)
efficientnet	33.0x	0.99x
inception	17.9x	0.94x
mobilnet	13.3x	0.85x

Figure 10 shows the on-chip and off-chip memory assigned to parameters for the mobilenet CNN model with a fully-connected layer of size N used to expand the 1D inputs into a 2D image. For this experiment, we explored fully connected input sizes of M where $14 \leq M \leq 50176$ and the original CNN input size is 224x224x3. These bounds were derived from the original input size of 224x224x3 using $14 \approx \sqrt{224}$ and $50176 = 224 \times 224$. We compiled each generated model with the edgetpu compiler and recorded the amount of memory used for on-chip and off-chip model parameters. For up to 900 nodes in the fully-connected layer, the edgetpu compiler uses only on-chip memory. At 3025 nodes in the fully-connected layer, we notice the first instance in which the edgetpu compiler only uses off-chip memory for model parameters. In the graph, we can see that this phenomenon occurs semi-regularly when the on-chip memory falls to zero and there is a sharp spike in the off-chip memory. Above 32,041 fully-connected nodes, the edgetpu compiler no longer uses on-chip memory for model parameters. In short, it may be beneficial to test multiple potential sizes for a fully-connected layer to determine which fits best into on-chip memory.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

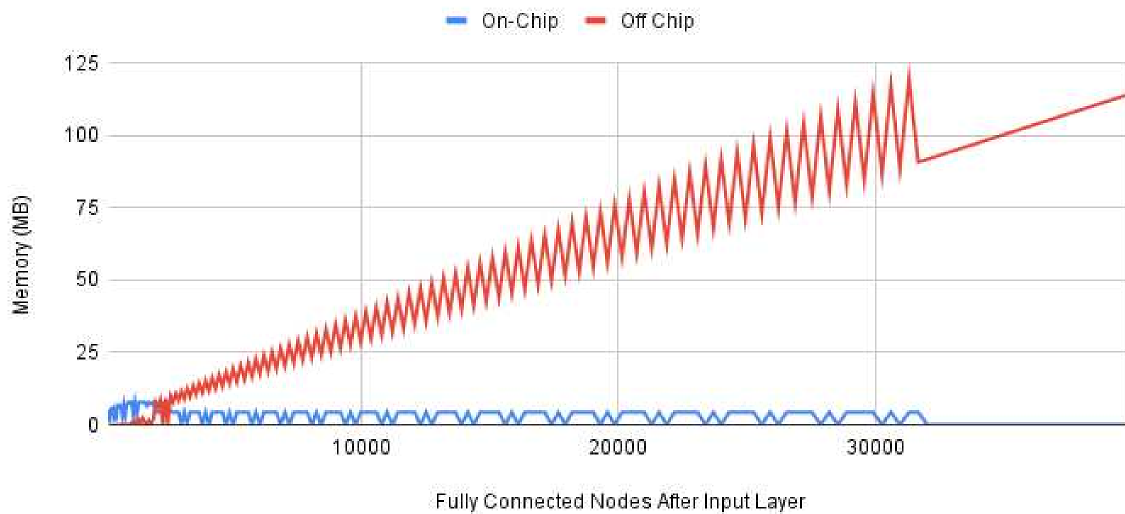

Figure 10. On-chip and off-chip memory used for adding a fully-connected layer to expand 1D inputs to a 2D image

3.3.1. Depth Extensions

To further evaluate the performance implications of transfer learning on an edge TPU, we generated altered CNNs, based on the original models for efficientnet, inception, and mobilenet, with more or fewer subgraphs of the main computational component of the network. Figure 4 demonstrates the main subgraph of `InceptionV1`. For each of the three models, we identify the main subgraph, extract that subgraph, and generate models with 1 to 50 repetitions of that subgraph. Each of the generated models also reproduces the rest of the original model.

Figure 11 shows the results of this experiment. As shown, the performance gap between the CPU and TPU decreases as the subgraph is repeated more, but the gap remains at over 10x on these CNN models. On both the CPU and TPU, adding repeated subgraphs may be a potential avenue to improve the accuracy of the model for a problem in transfer learning without significantly impacting performance. The difference between the worst performing and best performing generated models on the TPU was 2.5x, 4.7x, and 2.24x, respectively, for `EfficientNetS`, `InceptionV1`, and `MobileNet1.0`. For the CPU, the difference between the worst performing and best performing generated models was 1.06x, 1.15x, and 1.38x, respectively. We expect that cache locality and data transfer explain the CPU's relative efficiency as the depth of the model increases, but we would need to investigate further to definitively show this. We leave a more in-depth study of the performance of deep models to future work.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

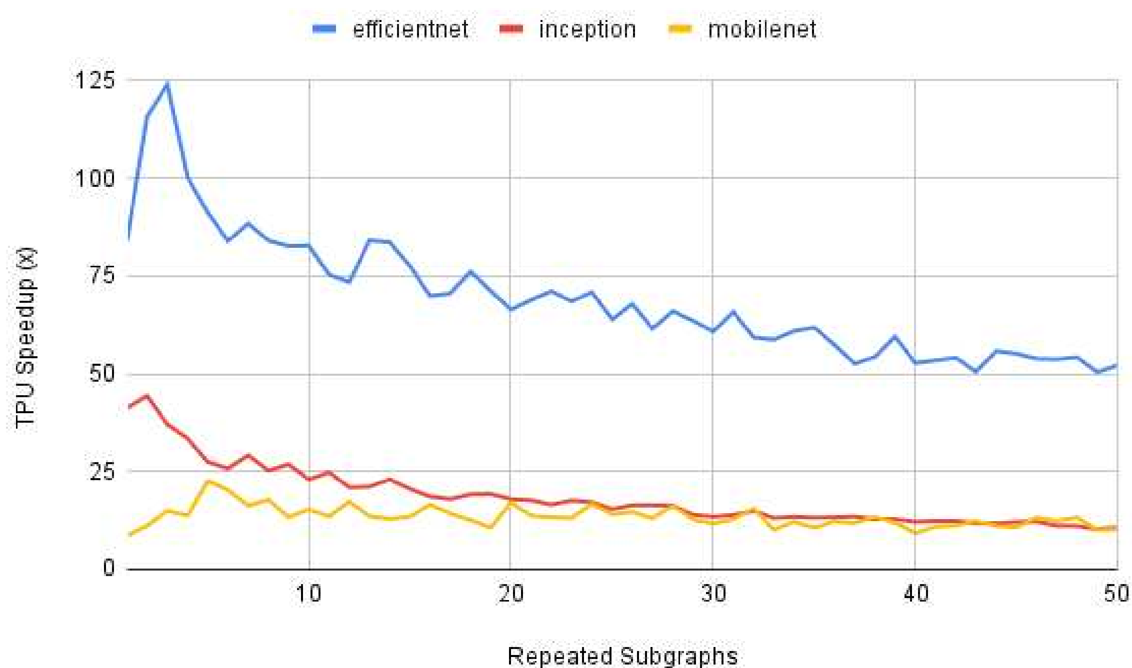

Figure 11. TPU Speedup on generated deep CNN models with repeated subgraphs

Figure 12 shows similar results for the energy efficiency of the TPU on generated deep CNN models. The TPU consistently outperforms the CPU, but the gap gets smaller as the depth of the model increases.

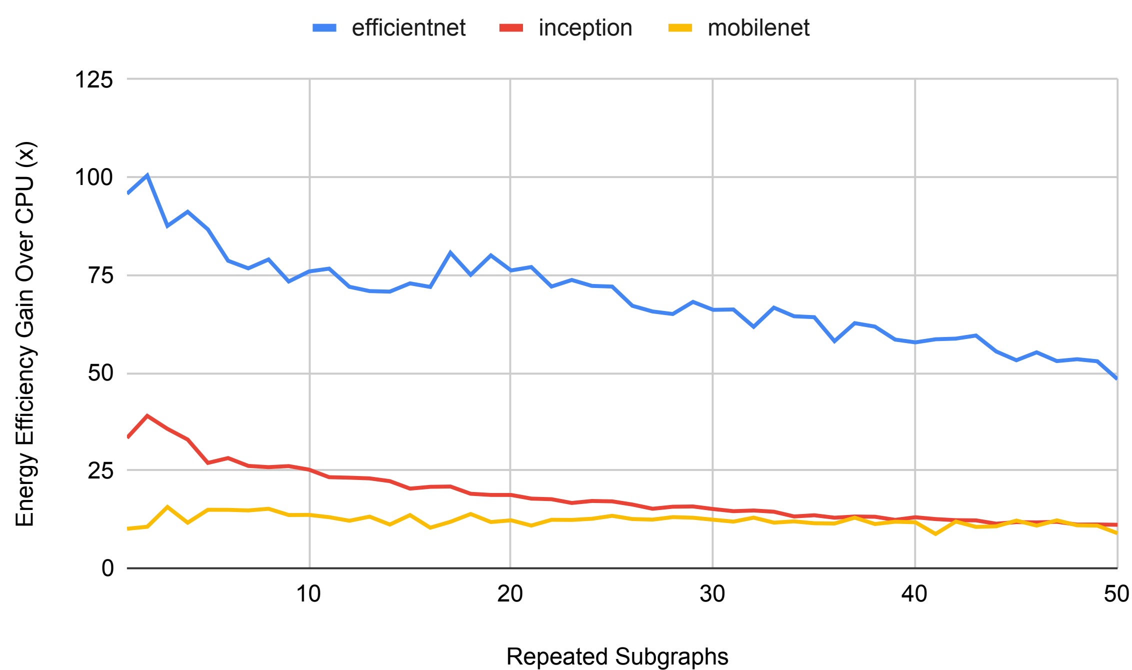

Figure 12. TPU energy efficiency per inference on generated deep CNN models with repeated subgraphs

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

3.3.2. Wide Extensions

We also generated wider versions of efficientnet, inception, and mobilenet, and we analyzed the runtime performance of these generated models. We explored two procedures for generating wider models.

In the first procedure, we widened the models by scaling up the bias of convolution operations in the models based on a factor from 1.1x to 5x, which was the limit of generating Tensorflow Lite models on our training hardware. Increasing the bias provides more parameters for the model to learn and increases the amount of parallel work available when performing inference with the model. We examined the effects of scaling at the input and output layers because practitioners of transfer learning may increase the model size to account for a different number of inputs and outputs for their targeted problem.

Figure 13 shows the runtime performance of the TPU compared to the CPU with scaled input size. In this graph, we can see that the TPU's speedup compared to the CPU increases as the width of the model increases. We note that scaling the width of the input layer increases the width of the entire model as the larger output tensor from the scaled layer serves as an input to the rest of the model, which we scaled accordingly. We observe similar results when comparing the energy efficiency of performing an inference on the TPU as compared to the CPU as the model gets wider.

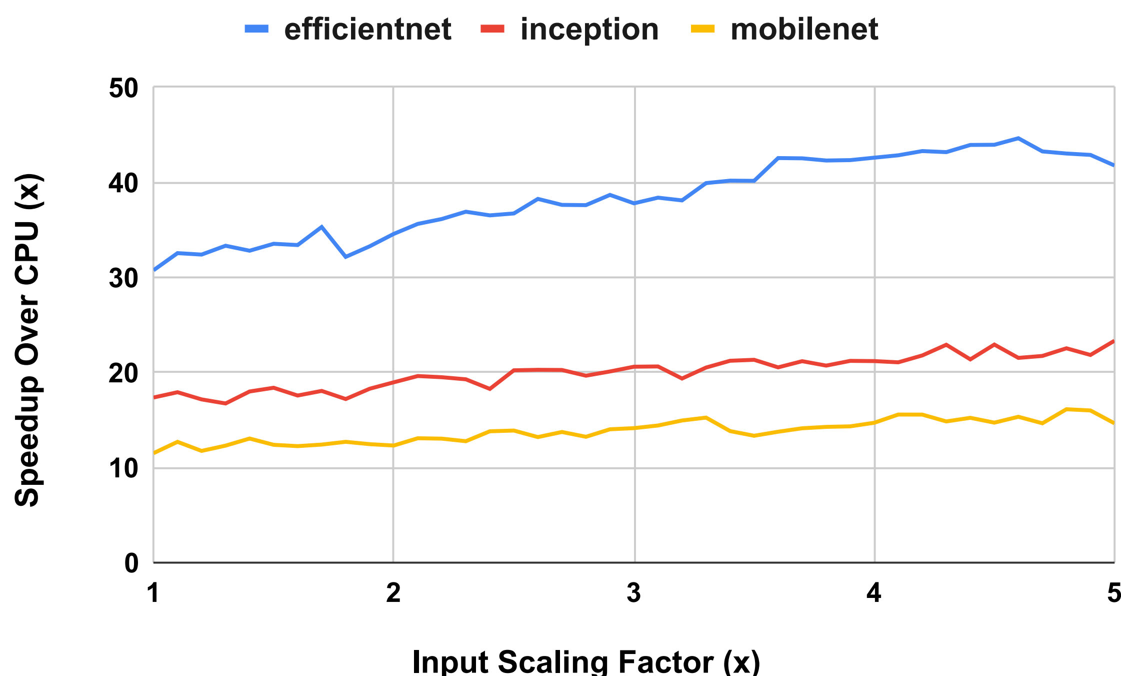

Figure 13. TPU runtime performance speedup over CPU on models widened by scaling up input size

Figure 14 shows the runtime performance of the TPU compared to the CPU with scaled output size. As the width of the model was increased at the output layers, the gap decreased between the runtime performance on the TPU compared to the CPU. We expect that this occurs due to the increased size of the input to the `SoftMax` operation at the end of each model. Compared to scaling the width of the input, there is less parallel work for the TPU to take advantage of throughout the model, leading to a declining performance benefit for using the TPU. We observe similar results for energy efficiency in that the TPU's energy efficiency benefit over the CPU decreases as the width of the output layer increases.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

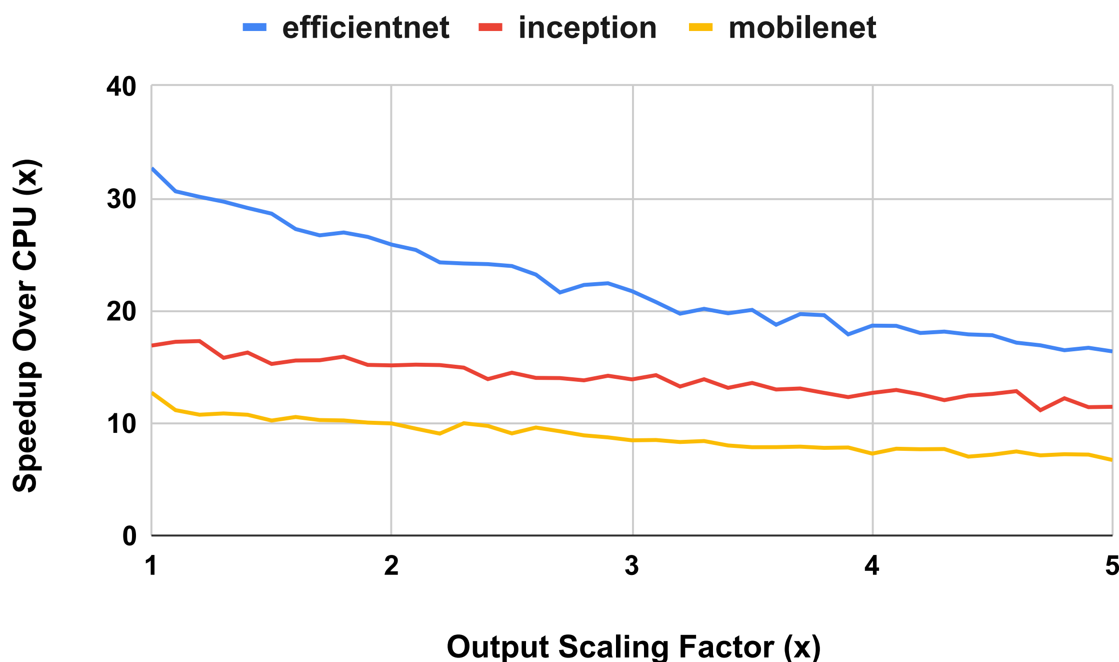

Figure 14. CPU and TPU Runtime Performance on models widened by scaling up output size

Figure 15 shows the runtime performance of the TPU compared to the CPU with scaled output size. From the graph, it appears that the effect of scaling up the output size slightly outweighs the effect of scaling up the input size. For all three benchmarks, the TPU maintains a similar runtime speedup over the CPU at all scaling factors.

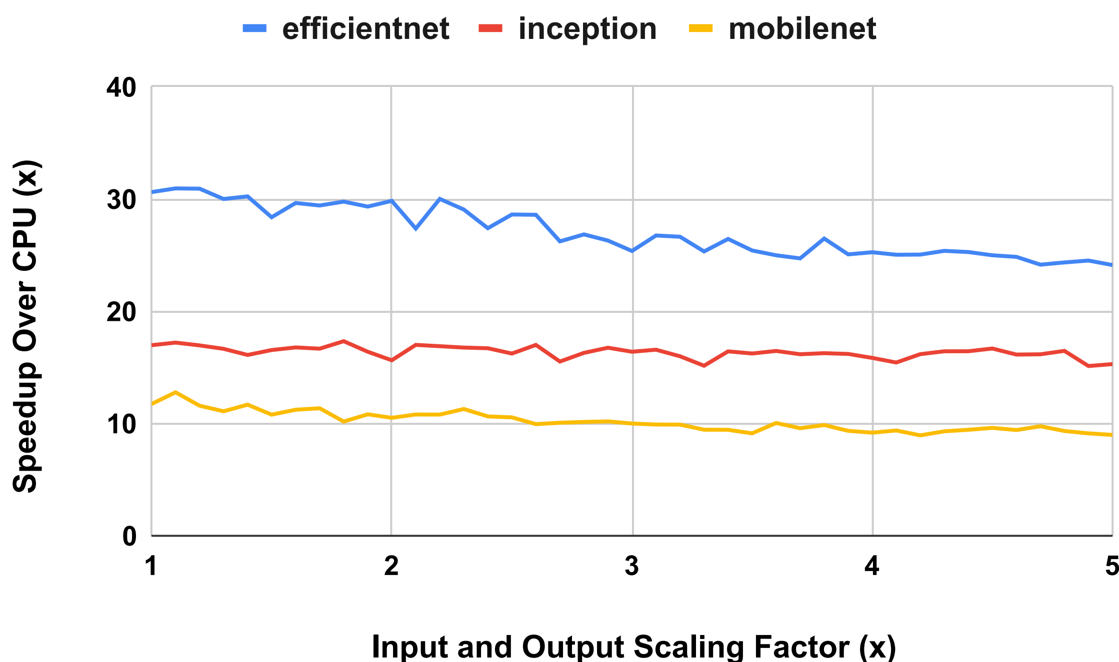

Figure 15. CPU and TPU Runtime Performance on models widened by scaling up input and output size

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

In the second procedure, we increased the width of the models by expanding convolution layers based on insights from the main subgraph in the Inception V1-V4 models, as described in figure 4. We performed this expansion on the `EfficientNetS` and `MobileNetV1` models. We excluded the `InceptionV1` model because we already had data on expanding the Inception model from versions 1 through 4 in our baseline experiments. Figure 16 shows the results of this experiment on widening the model by adding parallel layers. As parallel layers are initially added to the model to increase it's width, the TPU provides an increased performance gain over the CPU. However, as more parallel layers are added, the TPU reaches the limit of its ability to exploit parallel work, and the speedup over the CPU reaches a steady state. Similar to prior experiments, the energy efficiency results mirror the runtime performance results.

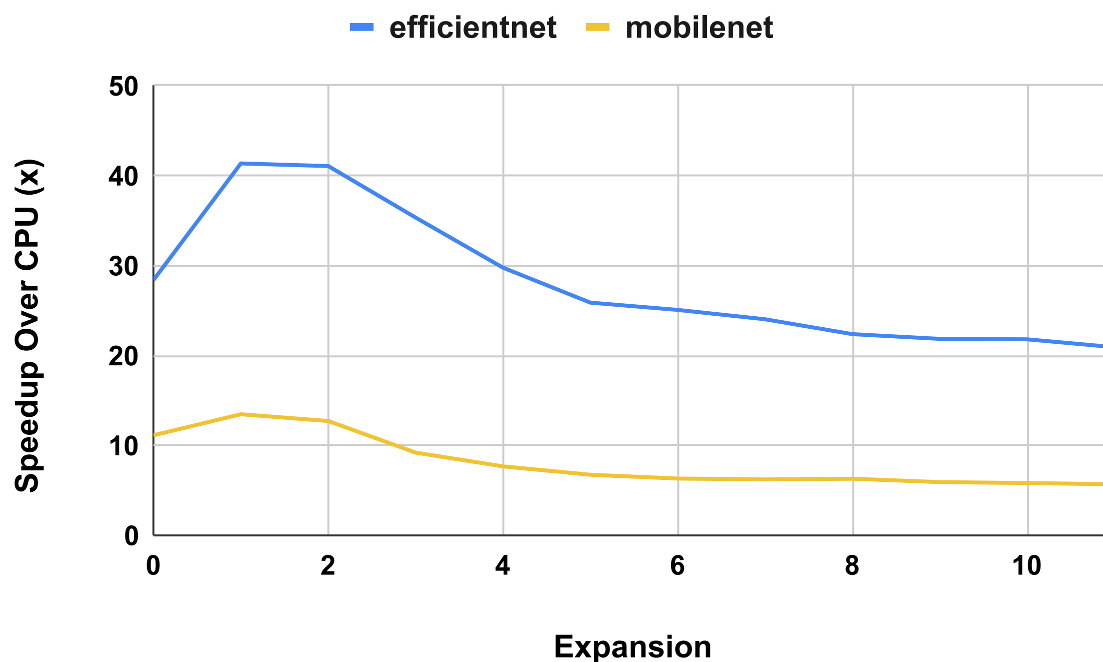

Figure 16. CPU and TPU Runtime Performance on models widened by adding parallel layers

3.3.3. Off-Chip Memory

As previously observed for fully-connected, feed-forward neural networks [38], the runtime performance of convolutional neural networks is also affected by the percentage of parameters stored off-chip. Figure 17 shows the TPU runtime performance per inference compared to the percentage of off-chip memory used to store model parameters. We perform this experiment on the models generated from the baseline mobilenet CNN model with scaled filter sizes to increase the width of the model. As the width of the model increases, so does the amount of memory required for parameters, and therefore the amount of off-chip memory used to store parameters also increases. Compared to the results demonstrated in prior work on feed-forward neural networks, CNN runtime scales linearly, instead of in a stepwise manner, due to the increased amount of computation required for CNNs that hides the memory cost of storing parameters off chip.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

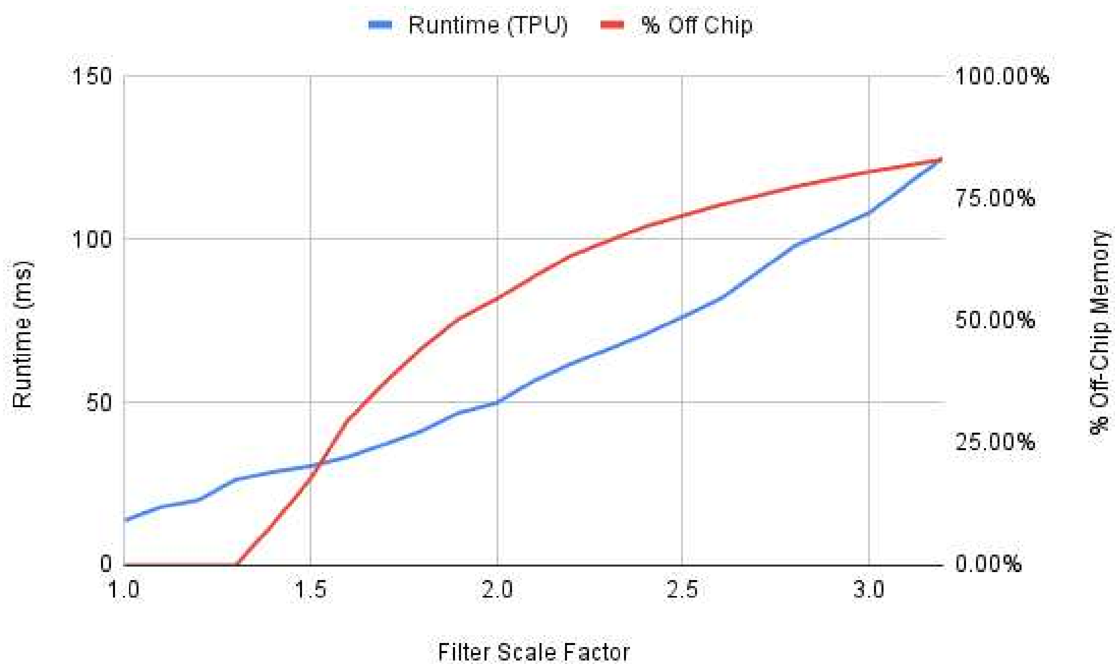

Figure 17. Runtime compared to percentage of weights stored off-chip for scaled width `InceptionV1`

4. Discussion

Building from the data in Section 3, we have the following practical recommendations for designing machine learning models for edge devices.

4.1. Prefer Single-Core for Long Term Energy Efficiency but Multi-Core for Energy Efficiency Per Inference

On the baseline CNNs, using a 4-core CPU provided a runtime performance and energy efficiency advantage compared to the 1-core CPU. However, the increase in energy used per inference outweighed the decrease in runtime. Therefore, if inferences will be run continuously in an edge environment, a 1-core CPU will be more energy efficient over time. For bursts of inferences, the 4-core CPU should provide better energy efficiency due to the decrease in runtime.

4.2. Prefer a TPU for Convolutional Neural Networks

For all of the convolutional neural networks evaluated in this paper, the TPU outperformed both the single core and 4 core CPU in both runtime performance and energy efficiency. As we scaled the CNN models, both in depth and in width, the TPU continued to consistently outperform the CPU. The only case in which the CPU outperformed the TPU occurred when we added a large fully connected layer to map 1 dimensional inputs to the 2 dimensional image input expected by the CNN for transfer learning. This case concurs with our findings on the performance degradation of fully connected neural networks on the TPU as the percentage of off-chip memory usage increases.

4.3. For Edge TPUs, prefer model depth when possible for convolutional networks

Due to the width of convolution operations, there is more parallelism inherent to CNNs, and further increasing the width may overload the hardware's capacity. Table 2 provides a comparison point to a fully connected network with 240 layers and 810 nodes per layer, and all but the smallest CNNs require more floating-point operations to perform an inference.

To mitigate this challenge, it is advisable to focus on expanding the depth of the network instead. By increasing the depth, the network can effectively capture complex hierarchical features [30]. Early

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

convolutional neural networks [29] used as few as 5 layers, but more recent convolutional neural networks, such as Inception-Resnet at 572 layers [28] and the Residual Attention Network at 452 layers, have become significantly deeper. With more data available to train networks, deeper networks can be well-supported by the edge TPU.

4.4. *Avoid performance cliffs for transfer learning with fully-connected input layers*

As shown in our experiments on transfer learning, the size of a fully-connected layer for mapping 1 dimensional input data to a 2 dimensional image may significantly affect performance. As the number of nodes in the fully-connected layer increases, the edgetpu compiler may choose to place all of the model parameters in off-chip memory, leading to degraded performance. In our dataset, these performance cliffs were not easy to predict based on the number of nodes in the fully-connected layer. However, the performance cliffs occurred infrequently enough that testing the performance of a few fully-connected layer sizes should be sufficient to avoid them.

Performance cliffs are common pitfalls for hardware accelerators. NVIDIA provides an occupancy calculator for general-purpose GPU applications (GPGPU) [33] to help application developers choose the correct number of threads and amount of memory to use. We recommend that edge TPU designers provide similar tools to help application developers choose the parameters for their neural networks. However, we note that this is a more complex issue to solve due to the process of developing a neural network model for execution on the edge TPU. The application developer must design the higher level model in a framework like Keras or Tensorflow, then convert the model to Tensorflow Lite, and finally compile the model using the edgetpu compiler. It may be difficult to develop a calculator that accounts for the nuances of this entire process.

4.5. *Scale Width at Input Layers to Exploit Parallelism on the Edge TPU*

Scaling the width of a Convolutional Neural Network (CNN) at the input layers can be a strategic choice to exploit parallelism and enhance the network's performance. By increasing the width, or the number of channels or filters in the initial convolutional layers, the network gains the ability to capture a broader range of low-level features and patterns from the input data. This enables the CNN to distribute the processing of different features across multiple parallel pathways. Consequently, scaling the width of the network at the input layers can lead to more efficient and effective feature extraction, making it a beneficial strategy for enhancing CNN performance. We observe that the Edge TPU effectively exploits the increased parallelism found in the model by increasing its width at the input layers. Therefore, scaling the width at the input layers may increase model accuracy and can be efficiently executed by the Edge TPU.

4.6. *Limitations*

The experiments outlined in this study were performed on the Coral development board. This development board provides features for developing and testing IoT applications, but it is not optimized for deployed applications. For example, the development board runs Mendel Linux, which allows the developer to run programs in a familiar command-line environment. In a deployed IoT application, functionality, like the operating system, that is provided for developer convenience would not be included, leading to performance and energy improvements. We attempted to factor out the energy cost of these convenience features by measuring the resting energy used by the development board, but this may not perfectly model the energy usage of a custom IoT device using a tightly integrated TPU. For custom devices, the methodology presented in this paper can serve as a guide for analyzing the runtime and energy performance of the device on potential network models.

We analyze a wide variety of neural network models in this paper, but there are infinitely many ways to structure a neural network. This paper does not evaluate recurrent neural networks such as long short-term memory models, or transformer networks, or other commonly used models. The runtime and energy performance evaluation of additional model structures is left to future work.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

We have not evaluated the machine learning algorithm accuracy of these modifications to CNNs. The accuracy of the algorithm is highly dependent on the problem for which machine learning is being applied and on the availability of training data. We leave experimentation on accuracy up to practitioners with a specific problem to solve and hope that the guidance on runtime and energy performance in this paper can assist them in finding an efficient CNN model.

5. Conclusions

In conclusion, this paper has provided an evaluation of the runtime and energy performance of Convolutional Neural Networks (CNNs) when executed on an Edge TPU. Our findings underscore the remarkable efficiency gains achieved by leveraging TPUs over traditional CPU architectures. Notably, we have demonstrated that extending the depth of a CNN has a comparatively limited effect on runtime performance in contrast to expanding its width. This insight can guide practitioners in optimizing their model architectures for TPU deployment, emphasizing the potential benefits of deeper networks.

We have analyzed various adjustments to CNNs that might be made by practitioners during the process of applying transfer learning. We find that simply resizing an image input to a different size has little impact on the runtime performance. However, adding a fully-connected layer to bridge the gap from a small number of real inputs to the larger number of expected inputs for a CNN may have a significant performance impact on an edge processor.

Furthermore, our investigation has shed light on the role of off-chip memory storage in CNN performance on TPUs. In line with our expectations, the impact of off-chip memory storage appears to be less consequential in the context of convolutional neural networks. This observation highlights the substantial computational requirements inherent to convolution operations, which tend to dominate the overall execution time.

In light of these findings, it is evident that the Edge TPU stands as a compelling platform for deploying CNNs, offering not only improved runtime efficiency but also energy savings. As the demand for efficient edge computing solutions continues to rise, our research contributes valuable insights that can aid in the development of optimized models and hardware configurations for real-world applications, especially for practitioners considering an application of transfer learning.

Author Contributions: Conceptualization, C.D.; methodology, C.D., J.B., R.R., and J.S.; software, C.D. and J.B.; validation, C.D., J.B., R.R., and J.S.; formal analysis, C.D., J.B., and J.S.; investigation, C.D. and J.B.; resources, R.R. and J.S.; data curation, C.D.; writing—original draft preparation, C.D.; writing—review and editing, C.D., J.B., and R.R.; visualization, C.D and J.B.; supervision, C.D. and J.B.; project administration, C.D. and J.B.; funding acquisition, R.R. All authors have read and agreed to the published version of the manuscript.

Funding: Equipment for this project was purchased through funding from the United States Naval Academy Cybersecurity Fund, and additional support was provided by the Program Executive Office for Integrated Warfare Systems.

Acknowledgments: We would also like to thank Dr. Mike Painter and LT Andrew Smith for their insights on this project.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CPU	Central Processing Unit
TPU	Tensor Processing Unit
CNN	Convolutional Neural Network

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

References

1. "What makes TPUs fine-tuned for deep learning?" Google. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning/. [Accessed: 07-May-2022].
2. "Frequently asked questions," Coral. [Online]. Available: https://coral.ai/docs/edgetpu/faq/what-is-the-edge-tpu/. [Accessed: 07-May-2022].
3. "Cloud TPU Performance Guide," Google. [Online]. Available: https://cloud.google.com/tpu/docs/performance-guide. [Accessed: 01-Sept-2023].
4. D. Xu, M. Zheng, L. Jiang, C. Gu, R. Tan, and P. Cheng. "Lightweight and Unobtrusive Data Obfuscation at IoT Edge for Remote Inference," IEEE Internet of Things Journal, vol 7, 2020, pp. 9540-9551.
5. A. Yazdanbakshsh, K. Seshadri, B. Akin, J. Laudon, R. Narayanaswami. "An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks," 2020, https://arxiv.org/abs/2102.10423
6. "Edge TPU performance benchmarks," Coral. [Online]. Available: https://coral.ai/docs/edgetpu/benchmarks/. [Accessed: 08-May-2022].
7. N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp. 1-12, doi: 10.1145/3079856.3080246.
8. "Dev Board Datasheet," Coral. [Online]. Available: https://coral.ai/docs/dev-board/datasheet/. [Accessed: 10-May-2022].
9. "Arm Cortex-A53 MPCore Processor Technical Reference Manual," ARM. [Online]. Available: https://developer.arm.com/documentation/ddi0500/latest/. [Accessed: 10-May-2022].
10. Baller, S.P.; Jindal, A.; Chadha, M.; Gerndt, M. "DeepEdgeBench: Benchmarking Deep Neural Networks on Edge Devices." In Proceedings of the 2021 IEEE International Conference on Cloud Engineering (IC2E), San Francisco, CA, USA, 4–8 October 2021; pp. 20–30.
11. Dominguez-Morales JP, Duran-Lopez L, Gutierrez-Galan D, Rios-Navarro A, Linares-Barranco A, Jimenez-Fernandez A. Wildlife Monitoring on the Edge: A Performance Evaluation of Embedded Neural Networks on Microcontrollers for Animal Behavior Classification. Sensors. 2021; 21(9):2975. https://doi.org/10.3390/s21092975
12. B. Kim, S. Lee, A. R. Trivedi and W. J. Song, Energy-Efficient Acceleration of Deep Neural Networks on Realtime-Constrained Embedded Edge Devices, in IEEE Access, vol. 8, pp. 216259-216270, 2020, doi: 10.1109/ACCESS.2020.3038908.
13. "Energizer L91 Ultimate Lithium Product Datasheet," Energizer. [Online]. Available: https://data.energizer.com/pdfs/l91.pdf. [Accessed: 10-May-2022].
14. A. Yazdanbakhsh, K. Seshadri, B. Akin, J, Laudon, and R. Narayanaswami, "An evaluation of Edge TPU accelerators for convolutional neural networks," arXiv preparing arXiv:2102.10423, 2021.
15. "Gen7i Transient Recorder and Data Acquisition System," Durham Instruments. [Online]. Available: https://disensors.com/product/gen7i-transient-recorder-and-data-acquisition-system/. [Accessed: 20-June-2022].
16. "Trained TensorFlow models for the Edge TPU," Coral. [Online]. Available: https://coral.ai/models/. [Accessed: 29-Jan-2022].
17. F. Zhuang et al., "A Comprehensive Survey on Transfer Learning," in Proceedings of the IEEE, vol. 109, no. 1, pp. 43-76, Jan. 2021, doi: 10.1109/JPROC.2020.3004555.
18. Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. 2018. A Survey on Deep Learning: Algorithms, Techniques, and Applications. ACM Comput. Surv. 51, 5, Article 92 (September 2019), 36 pages. https://doi.org/10.1145/3234150.
19. Alom, M.Z., Taha, T.M., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M.S., Hasan, M., Van Essen, B.C., Awwal, A.A.S., Asari, V.K. A State-of-the-Art Survey on Deep Learning Theory and Architectures. Electronics 2019, 8, 292. https://doi.org/10.3390/electronics8030292.
20. G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," in IEEE Potentials, vol. 13, no. 4, pp. 27-31, Oct.-Nov. 1994, doi: 10.1109/45.329294.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

21. Z. Li, F. Liu, W. Yang, S. Peng and J. Zhou, "A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects," in IEEE Transactions on Neural Networks and Learning Systems, vol. 33, no. 12, pp. 6999-7019, Dec. 2022, doi: 10.1109/TNNLS.2021.3084827.
22. Jaehoon Lee, Yasaman Bahri, Roman Novak, Sam Schoenholz, Jeffrey Pennington, and Jascha Sohl-dickstein. Deep neural networks as gaussian processes. In International Conference on Learning Representations, 2018.
23. Roman Novak, Lechao Xiao, Jaehoon Lee, Yasaman Bahri, Greg Yang, Jiri Hron, Daniel A. Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Bayesian deep convolutional networks with many channels are gaussian processes. In International Conference on Learning Representations, 2019.
24. Jaehoon Lee, Lechao Xiao, Samuel Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide Neural Networks of Any Depth Evolve as Linear Models Under Gradient Descent. In NeurIPS 2019.
25. Mavrovouniotis, M., Yang, S. Training neural networks with ant colony optimization algorithms for pattern classification. Soft Computing 19, 1511–1522 (2015). https://doi.org/10.1007/s00500-014-1334-5
26. Beheshti, Z., Shamsuddin, S.M.H., Beheshti, E. et al. Enhancement of artificial neural network learning using centripetal accelerated particle swarm optimization for medical diseases diagnosis. Soft Computing 18, 2253–2270 (2014). https://doi.org/10.1007/s00500-013-1198-0
27. Ashraf Mohamed Hemeida, Somaia Awad Hassan, Al-Attar Ali Mohamed, Salem Alkhalaf, Mountasser Mohamed Mahmoud, Tomonobu Senjyu, and Ayman Bahaa El-Din. Nature-inspired algorithms for feed-forward neural network classifiers: A survey of one decade of research. Ain Shams Engineering Journal, Volume 11, Issue 3, 2020, Pages 659-675, ISSN 2090-4479, https://doi.org/10.1016/j.asej.2020.01.007.
28. Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, inception-ResNet and the impact of residual connections on learning. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17). AAAI Press, 4278–4284.
29. Yann LeCun and Yoshua Bengio. 1998. Convolutional networks for images, speech, and time series. The handbook of brain theory and neural networks. MIT Press, Cambridge, MA, USA, 255–258.
30. Eugene Charniak. 2019. Introduction to Deep Learning. The MIT Press.
31. F. Wang et al., "Residual Attention Network for Image Classification," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 2017, pp. 6450-6458, doi: 10.1109/CVPR.2017.683.
32. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 2818-2826, doi: 10.1109/CVPR.2016.308.
33. Nsight Compute Occupancy Calculator. [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator. [Accessed: 9-Jun-2023].
34. Seyedehfaezeh Hosseininoorbin, Siamak Layeghy, Brano Kusy, Raja Jurdak, Marius Portmann. Exploring Edge TPU for deep feed-forward neural networks. Internet of Things, Volume 22, 2023, 100749, ISSN 2542-6605, https://doi.org/10.1016/j.iot.2023.100749.
35. Y. Ni, Y. Kim, T. Rosing and M. Imani, "Online Performance and Power Prediction for Edge TPU via Comprehensive Characterization," 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 2022, pp. 612-615, doi: 10.23919/DATE54114.2022.9774764.
36. Seyedehfaezeh Hosseininoorbin, Siamak Layeghy, Mohanad Sarhan, Raja Jurdak, Marius Portmann. Exploring edge TPU for network intrusion detection in IoT. Journal of Parallel and Distributed Computing, Volume 179, 2023, 104712, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2023.05.001.
37. A. A. Asyraaf Jainuddin, Y. C. Hou, M. Z. Baharuddin and S. Yussof, "Performance Analysis of Deep Neural Networks for Object Classification with Edge TPU," 2020 8th International Conference on Information Technology and Multimedia (ICIMU), Selangor, Malaysia, 2020, pp. 323-328, doi: 10.1109/ICIMU49871.2020.9243367.
38. C. DeLozier, F. Rooney, J. Jung, J. A. Blanco, R. Rakvic and J. Shey, "A Performance Analysis of Deep Neural Network Models on an Edge Tensor Processing Unit," 2022 International Conference on Electrical, Computer and Energy Technologies (ICECET), Prague, Czech Republic, 2022, pp. 1-6, doi: 10.1109/ICECET55527.2022.9873024.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 October 2023 doi:10.20944/preprints202310.1612.v1

39. M. Tan, R. Pang and Q. Le, "EfficientDet: Scalable and Efficient Object Detection," in 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 2020 pp. 10778-10787. doi: 10.1109/CVPR42600.2020.01079
40. M. Tan, Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in Proceedings of the 36th Internation Conference on Machine Learning, Long Beach, CA, PMLR 97, 2019.
41. C. Szegedy, et al., "Going deeper with convolutions," in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, 2015 pp. 1-9. doi: 10.1109/CVPR.2015.7298594
42. Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 4510–4520, 2018.
43. Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In Proceedings of the IEEE International Conference on Computer Vision, pages 1314–1324, 2019.
44. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR, abs/1704.04861. Retrieved from http://arxiv.org/abs/1704.04861
45. MobileNet, MobileNetV2, and MobileNetV3. [Online]. Available: https://keras.io/api/applications/mobilenet/. [Accessed: 28-Sept-23].
46. Tensorflow 2 Detection Model Zoo. [Online]. Available: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md. [Accessed: 28-Sept-23].
47. Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation. Liang-Chieh Chen and Yukun Zhu and George Papandreou and Florian Schroff and Hartwig Adam. arXiv:1802.02611, 2018.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.