Acknowledgements

We would like to thank Blake Hechtman, Ryan Sepassi, and Tong Shen for their help with software changes needed to make our algorithms to work well on TPUs. We would also like to thank Berkin Akin, Okan Arikan, Yiming Chen, Zhifeng Chen, Frank Chu, Ekin D. Cubuk, Matthieu Devin, Suyog Gupta, Andrew Howard, Da Huang, Adam Kraft, Peisheng Li, Yifeng Lu, Ruoming Pang, Daiyi Peng, Mark Sandler, Yonghui Wu, Zhinan Xu, Xin Zhou, and Menglong Zhu for helpful feedback and discussions.

A. Variance of ProxylessNAS-Mobile Model

For stand-alone model training, we estimated the variance in accuracy across runs by starting five identical runs of the ProxylessNAS-Mobile model. We repeated the experiment in two configurations. In the first configuration, we trained for 90 epochs and then evaluated the models on our validation set; the resulting validation set accuracies were $[76.1\%, 76.4\%, 76.4\%, 76.5\%, 76.2\%]$. In the second configuration, we trained for 360 epochs and then evaluated on our test set; the resulting test set accuracies were $[75.0\%, 75.0\%, 74.9\%, 74.9\%, 75.0\%]$.[7]

B. Average vs Argmax Inference Time

For the absolute value reward function with a constant RL learning rate, we argued that the reason models didn't converge to the target inference time was because of differences between the inference times of *randomly sampled* models vs. the *argmax* model taken by selecting the most likely choice for each categorical decision. To test this hypothesis, we compared the two at the end of a search. We obtained an average time from randomly sampled models using an exponential moving average with a decay rate of 0.9 which was updated every 100 training steps. While average times consistently converged to the desired inference time target, the inference time from the argmax could differ by 8 ms or more.

β	Avg. Time	ArgMax Time
-0.01	134.2 ms	128.8 ms
-0.02	104.3 ms	92.2 ms
-0.05	84.2 ms	75.8 ms
-0.10	83.1 ms	78.3 ms
-0.20	84.0 ms	82.7 ms
-0.50	84.2 ms	83.4 ms
-1.00	83.2 ms	85.5 ms

Table 11: Average vs. argmax inference times when using the absolute value reward function without a learning rate schedule for the RL controller.

[7] Accuracies on the validation set are typically a few percentage points higher than on the test set.

C. Rematerialization

If op warmup is implemented naively then the activation memory required to train the shared model weights grows linearly with the number of possible operations in the search space. If many different possible operations can be simultaneously enabled at each position in the network, the model will be unable to fit in memory. We use rematerialization to address this issue. During the forward pass, we apply N different operations to the same input, and average their outputs. Rather than retaining the intermediate results of these operations for use during the backwards pass, we throw them away. During the backwards pass, we recompute the intermediate results for one operation at a time. In practice, this leads to a large decrease in memory requirements, as we only need to retain a single input tensor and a single output tensor for each choice block. For example, in our reproduction of the original ProxylessNAS search space with a per-core batch size of 128, rematerialization decreases the memory needed to train a reproduction of the original ProxylessNAS search space from 29.5 GiB to 4.8 GiB. This memory-saving technique, which allowed us to scale to larger search spaces, came at the cost of roughly a 30% increase in search times to perform a second forward pass for each of the N possible operations.

Finally, we note that although this rematerialization trick was developed with our version of op rampup in mind, it could also be used to reduce the memory requirements of a method such as DARTS [24] which requires us to evaluate every possible operation in the search space at every training step.

D. Discussion of Absolute Value Reward

We now contrast a typical architecture search workflow with the MnasNet-Soft reward function against a workflow with our new Absolute Value reward function. For the MnasNet-Soft reward function, the first step when using a new search space or training configuration is to tune the RL controller's cost exponent β to obtain inference times which are reasonably close to our target latency. In our early experiments, we found that grid searching over $\beta \in \{-0.03, -0.04, -0.05, -0.06, -0.07, -0.08, -0.09\}$ worked well in practice. However, running this grid search increased the cost of architecture search experiments by a factor of 7.

Even after we fixed the value of β, the latencies and accuracies of architectures found by a search could vary significantly from one run to the next. For example, in our reproduction of the ProxylessNAS search space with $\beta = -0.07$, five identical architecture search experiments returned latencies which ranged from 74ms to 82ms. We also saw a wide variance in accuracies across the different architectures, ranging from 75.8% to 76.4% on the valida-

tion set and 74.2% to 75.1% on the test set. Larger models generally had better accuracies, indicating that the problem stemmed from our inability to precisely control the latency.

This helped motivate our Absolute Reward function, which allowed the RL controller to reliably find architectures whose latencies were close to the target. For example, the low variance of searched TuNAS model latencies in Tables 4, 5, 6, and 12 shows we can reliably find high-quality architectures within 1 ms of the target across several different search configurations, even when we reuse the same search hyper-parameters between different setups.

As an alternative to the absolute value reward function, we also considered searching for an architecture close to the inference time target, and then uniformly scaling up or down the number of filters in every layer. While this helped reduce the variance of searched model accuracies, it did not remove the need to tune the RL cost exponent, since we needed to find a model that was already close to the inference time target to get good results. Furthermore, finding the right scaling factor to hit a specific inference time target added an extra step to experiments in this setup. The absolute value reward function gave us high-quality architectures with a more streamlined search process.

E. Experimental Setup

E.1. Standalone Training for Classification

During stand-alone model training, each model was trained using distributed synchronous SGD on TensorFlow with a Cloud TPU v2-32 or Cloud TPU v3-32 instance (32 TPU cores) and a per-core batch size of 128. Models were optimized using RMSProp with momentum = 0.9, decay rate = 0.9, and epsilon = 0.1. The learning rate was annealed following a cosine decay schedule without restarts [25], with a maximum value of 2.64 globally (or 0.0825 per core). We linearly increased the learning rate from 0 over the first 2.5% of training steps [11]. Models were trained with batch normalization with epsilon = 0.001 and momentum = 0.99. Convolutional kernels were initialized with He initialization [12],[8] while bias variables were initialized to 0. The final fully connected layer of the network was initialized from a random normal distribution with mean 0 and standard deviation 0.01. We applied L2 regularization with a strength of 0.00004 to all convolutional kernels except the final fully connected layer of the network. All models were trained with ResNet data preprocessing and an input image size of 224×224 pixels. When training models for 360 epochs, we applied a dropout rate of 0.15 before the final fully connected layer for models from MobileNetV2 search spaces and 0.25 for models from MobileNetV3 search spaces. We did not apply dropout when training models for 90 epochs.

As is standard for ImageNet experiments, our test set accuracies were obtained on what is confusingly called the ImageNet validation set for historical reasons. What we refer to as validation set accuracies were obtained on a held-out subset of the ImageNet training set containing 50,046 randomly selected examples.

[8]TensorFlow's default variable initialization heuristics, such as `tf.initializers.he_normal` are designed for ordinary convolutions, and can overestimate the fan-in of depthwise convolutional kernels by multiple orders of magnitude; we corrected this issue in our version.

E.2. Architecture Search for Classification

Architecture search experiments are performed using Cloud TPU v2-32 or Cloud TPU v3-32 instances with 32 TPU cores and a per-core batch size of 128.

For training the shared model weights, we use the same hyper-parameters as for stand-alone model training, except that the dropout rate of the final fully connected layer is always set to 0. When applying L2 regularization to the traininable model variables, we only regularize parameters which are used in the current training step. Because batch norm statistics can potentially vary significantly from one candidate architecture to the next, batch norm is always applied in "training" mode, even during model evaluation.

For training the RL controller, we use an Adam optimizer with a base learning rate of 3e-4, $\beta_1 = 0$, $\beta_2 = 0.999$, and ϵ =1e-8. We set the learning rate of the RL controller to 0 for the first 25% of training. If using an exponential schedule, we set the learning rate equal to the base value 25% of the way through training, and increase it exponentially so that the final learning rate is 10x the base learning rate. If using a constant schedule, we set the learning rate equal to the base learning rate after the first 25% of training.

E.3. Object Detection

Our implementation is based on the Tensorflow Object Detection API [16]. All backbones are combined with SSDLite [33] as the head. Following MobileNetV2 [33] and V3 [13], we use the last feature extractor layers that have an output stride of 16 (C4) and 32 (C5) as the endpoints for the head. In contrast with MobileNetV3 + SSDLite [13], we do *not* manually halve the number of channels for blocks between C4 and C5, since in our case the number of channels is automatically learned by the search algorithm. All experiments use 320×320 input images.

For standalone training, each detection model is trained for 50K steps from scratch on COCO train2017 data using a Cloud TPU v2-32 or TPU v3-32 instance (32 TPU cores) with a per-core batch size of 32. We use SGD to optimize the shared model weights with a momentum of 0.9. The (global) learning rate is warmed up linearly from 0 to 4 during the first 5K steps and then decayed to 0 following a cosine schedule [25] during the rest of the training process.

For architecture searches, training configurations for the model weights remain the same as for standalone training.

We split out 10% of the data from COCO train2017 to compute the reward during an architecture search. The training setup of the RL controller is the same as for classification, except that the base learning rate of the Adam optimizer is set to 5e-3. Whereas classification models are evaluated based on accuracy, detection models are evaluated using mAP (mean Average Precision). To obtain results in Table 7, architecture searches were carried out in the MobileNetV3-Like search space with a target inference cost of 106ms to match the simulated latency of MobileNetV3 + SSDLite.

To obtain the test-dev results, each model is trained over the combined COCO train2017 and val2017 data for 100K steps. Other settings remain the same as those for standalone training and validation.

E.4. Simulated Inference Times

In early experiments, we found that if we benchmarked the same model on two different phones, the observed latencies could differ by several milliseconds. To ensure that our results were reproducible – and to mitigate the possibility of random hardware-specific variance across runs – we estimated the latencies of our models using lookup tables similar to those proposed by NetAdapt [42]. These lookup tables let us estimate the latency of each individual operation (e.g., convolution or pooling layer) in the network. The overall latency of a network architecture was estimated by summing up the latencies of all its individual operations.

We validated our use of simulated latencies by sampling 100 random architectures and comparing the simulated numbers against on-device numbers measured on a real Pixel-1 phone. Figure 4 shows that the two are well-correlated.

F. Cost of Random Search vs Efficient NAS

Training a single architecture for 90 epochs on ImageNet requires about 1.7 hours using a Cloud TPU v3-32 instance (32 cores), whereas a single architecture search run takes between 8 and 13 hours, depending on the search space. This means that for the cost of a single 90-epoch search, we can evaluate 4-8 random models. In some cases, we found that the cost of an efficient architecture search could be further improved by increasing the number of search epochs from 90 to 360. For the cost of a single 360-epoch search, we can evaluate 15 - 30 random models. We provide a generous budget of 20 - 50 models for our random search experiments in order to demonstrate that efficient architecture search can outperform random search even if each random search experiment is more compute-intensive.

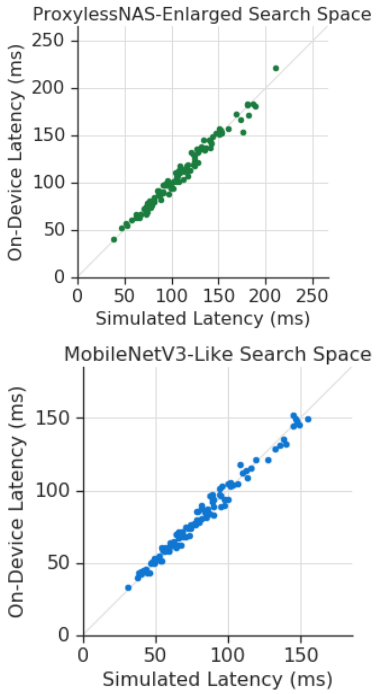

Figure 4: On-device vs. simulated latencies in the ProxylessNAS-Enlarged and MobileNetV3-Like search spaces. Each plot is based on 100 random architectures in the given space.

Agg. Sharing	Valid Acc (%)	Test Acc (%)	Latency
No	76.4 ± 0.1	$\mathbf{75.0 \pm 0.1}$	84.1 ± 0.4
Yes	76.3 ± 0.2	$\mathbf{75.0 \pm 0.1}$	84.0 ± 0.4

Table 12: Effect of aggressive weight sharing (abbreviated as "Agg Sharing" in the table above) on the quality of searched architectures. Each search is run for 90 epochs on the ProxylessNAS search space with op and filter warmup enabled.

G. Quality of Aggressive Weight Sharing

To verify the quality impact of aggressive weight sharing, we ran architecture searches on the original ProxylessNAS search space both with and without aggressive sharing. The results (Table 12) indicate that aggressive weight sharing does not significantly affect searched model accuracies in this space. Our other two search spaces (ProxylessNAS-Enlarged and MobilenetV3-Like) were too large us to run searches without aggressive weight sharing.