Performance Analysis of Machine Learning Model Training on Low Power Devices

Group 4: Saurabh Agarwal, Derek Paulsen, Derek Hancock

1 Introduction

Low power devices are being increasingly used for machine learning inference but are rarely (if ever) used for machine learning training. This problem has been explored little but could result in incredible benefits. If these devices could be leveraged for machine learning training, it would open up billions of devices for training which would allow for more customized solutions in the field, reduced network traffic, and cloud system overhead.

Previous work has explored how to efficiently do inference on these low power devices by creating more efficient and smaller models, resulting in networks like Mobilenet[3]. These works however don't consider training on low power devices, which has substantially different requirements in terms of latency, throughput, and memory. Very little work has studied the bottlenecks of model training on edge devices and what can be done to make training on edge devices more feasible.

Therefore, the goal of this project is to investigate the performance of machine learning model training on low power devices. We do this through three main avenues. First, we look at using memory profiling tools to profile common convolutional neural networks (CNN's) and find the memory bottlenecks of those networks. Second, we run micro benchmarks on a single convolution operation after identifying the convolution operation as the main computation bottleneck in a CNN. We show that micro benchmarks should be used to profile the target device, as each device has different sweet spots for efficient model training and inference. Finally, we explore training modifications such as random layer training in order to make training more feasible on an edge device, because these modifications have the potential to reduce the memory and computation time required to train a network.

2 Background

In recent years there has been a lot of work around building machine learning solutions to run on the edge. Most of the work is targeted around inference of CNN's. In our work we wanted to figure out how training tends to work on these low power devices. The majority of machine learning workloads try to minimize the loss functions using gradient descent optimizers. Gradient descent is one of the most popular way to to optimize a loss function defined by $J(\theta)$. θ being the parameters of the model. Most of the contemporary machine learning models are trained using mini-batch stochastic gradient descent. The gradients are calculated for each of the n parameters in a batch but the update is done once per batch. The update equation for mini-batch SGD is given by

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

For the purpose of this work we worked exclusively with with convolution neural networks. Convolution neural networks are de facto models for image recognition tasks and have now made inroads in both in natural language processing and speech recognition problems. Convolutions are memory efficient though they are quite compute intensive. In recent years specialized hardware like graphics processing units (GPUs) and specific accelerator like Google's tensor processing units provide special hardware units suitable for doing millions of convolution operations. To run these compute intensive operations on low power device there has been quite a lot of work for inference not much has been done on training. For the purpose of this study we used established data sets like MNIST, CIFAR-10 and MINC [1] to evaluate it. We also evaluated existing work like Resnet [2] and MobileNet [3] popular models for computer vision tasks. We performed study both on memory and computation requirements of these models.

3 Experiment Design

We decided to focus on convolutional neural networks for two reasons. First, most low power devices have very limited RAM, meaning that many long-short-term memory neural networks and multi-layer perceptron networks would not fit in memory. Second, we wanted to examine when compute is the bottle neck versus when memory is the bottleneck, which is much easier to do with a CNN.

3.1 Memory Profiling

When attempting to run a CNN on a low power device, it immediately becomes clear that memory is precious resource. Hence we decided to profile the memory usage of a two smaller CNNs to compare them in terms of memory usage. Specifically, we choose to profile Mobilenet V1 and Resnet 18 because they are relatively comparable in terms of size and performance on common machine learning benchmarks. As the name implies, Mobilenet V1 was intended for doing fast inference on mobile devices, which is achieved by taking some of the convolutional layers and splitting them into multiple smaller layers with smaller filters. Resnet 18 was designed to run on a normal server for general computer vision applications, hence we hypothesized that Mobilenet V1 than Resnet 18 would be easier to run on a lower power device, such as the Jetson TX2.

We found that there was a lack of memory profiling tools for both TensorFlow and PyTorch. The former having little to no documentation as to what the profiler outputs and the latter completely lacking tools to profile memory. This being the case we decided to use valgrind, and nvidia-smi to profile the memory albeit, at a coarse granularity.

3.2 Micro Benchmarks

The next challenge of training on a low power device is the lack of compute power present. This being the case the neural net design must use the limited resources as efficiently as possible. To determine what the most efficient way to use the resources are, we created micro benchmarks of a single convolution layer and varied the batch size, and filter size and timed how long it took for the operation to run.

We decide to mainly look at profiling (and reducing the run-time of) the convolution operation because the convolution operation dominates the computation time of a CNN model. We ran 15 steps of Mobilenet V1 and Resnet 18 on a Dell XPS 9570 and looked at the total execution time taken for each operation. In figure 1, we graph the top 2 dominating operations for both networks. In both cases, the time to perform a convolution dominates the second most expensive operation, batch normalization, by at least a factor of two. Thus, we can focus our efforts on understanding and reducing the convolution computation time.

3.3 Training Variations

After using micro benchmarks to guide our network design, we wanted to see if we could find a faster way to train it. Our first idea was to only train a subset

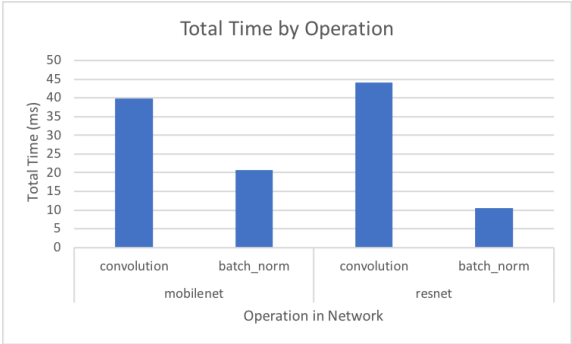

Figure 1: CPU Time by Operation in a CNN

of the filters in the network. Taking cue from [5], we ran experiments to randomly train parts of neural network. Our goal with this experiment was to have the network converge to an accuracy comparable to training the whole network while reducing the total training time and memory required. We varied the number of filters being trained on Resnet-18 by either training a random subset of layers (e.g. only update the filters in layer 3,4,7,etc.) or training a random subset of the filters in each layer e.g. (only 20 percent of the filters in each layer). Of course this could affect the convergence properties of the network, so for each of these experiments we measured both the time per step as well as the accuracy that the network converged to. We ran experiments for training from scratch on MNIST and CIFAR-10. In addition to changing the way that the training updates were done, we also experimented with transfer learning, which previous work has investigated for reducing the total time taken to train a neural network [6]. For this we followed the typical approach of taking a network that was pretrained on one data set and then retraining it on a new data set. The intuition behind this technique is that the pretrained network will already be extracting useful information from the new data set and will only require some tuning to converge to a high accuracy on the new data set.

4 Evaluation

In this section we present our findings for the memory usage, micro benchmarks, and training variations of convolutional neural networks. Our implementation of Mobilenet V1, Resnet 18, and LeNet can be found in our project github repository *https://github.com/dereklh4/project_744.git*

4.1 Memory Usage

The first experiment we ran was to try and determine what operations in the network were taking the most memory. Due to the memory constraints on the Jetson TX2, we had to conduct these experiments on the XPS 9750. Figure 2,3, 4, 5 and show the resulting massif profiles produced by valgrind.

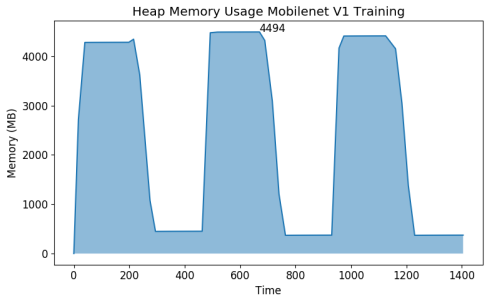

Figure 2: Heap memory usage for Mobilenet V1 Training

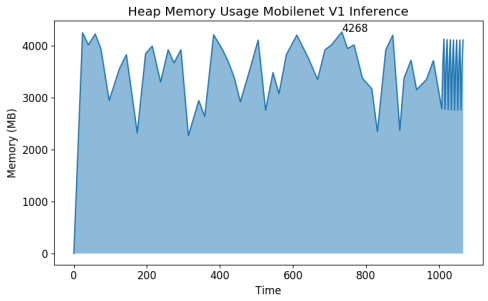

Figure 3: Heap memory usage for Mobilenet V1 Inference

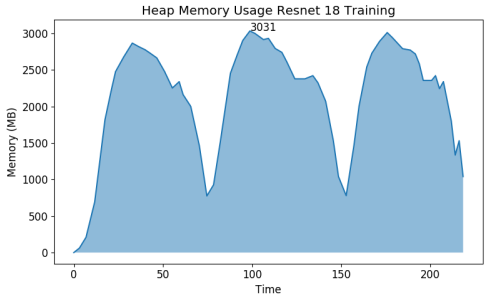

Figure 4: Heap memory usage for Resnet 18 Training

Examining the stack traces for these graphs, we found, not surprisingly, that the convolution operations were taking the majority of the memory used by the process and batch norm was taking the second largest. What

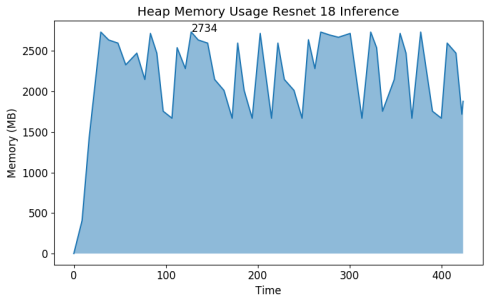

Figure 5: Heap memory usage for Resnet 18 Inference

we did not expect was to find that Mobilenet V1 actually used more memory than Resnet 18 for both training and inference. We initially thought this was a bug in PyTorch, however we got the same result running on TensorFlow. We then suspected that the CPU implementation may be to blame for this discrepancy, however we found the same result running on the GPU of the laptop as well as the figure 6 shows.

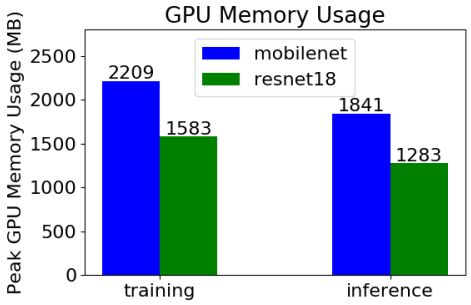

Figure 6: Peak GPU memory usage for Mobilenet V1 and Resnet 18

We exclude the peak heap (DRAM) memory usage from the plot because they we exactly the same for both networks with a peak of 1.8 GB, leveling off to 1.7 GB, which we attributed to the PyTorch run time. Comparing the GPU memory usage to the CPU memory usage, Mobilenet has a net decrease of 0.4 GB in total memory usage (4.4GB to 4.0GB), while Resnet has a net *increase* of 0.3GB (3.0GB to 3.3GB). Additionally, we found that the memory usage for the GPU implementation was steady over the course of training, building up to a peak and then never differing by more than 100 MB. In contrast, the CPU implementation of both networks has clear peaks and troughs. We hypothesize that this difference in memory usage patterns is due to the GPU implementations being optimized by calculating the memory needed to run the entire network ahead of time and then reusing the tensors which are allocated, while the CPU

implementations allocate and deallocate the memory on a per function call basis. We attempted to replicate these results on the Jetson TX2, however we were unable to run Mobilenet V1, due to a bug in PyTorch which affected the 1 by 1 filters in the convolutional layers.

4.2 Micro Benchmarks

After considering the memory constraints, we wanted to see how the optimize the run time for both training and inference in the CNN. As figure 1 shows, the convolutions were the most expensive operation in the networks, ergo we decided to benchmark a single convolution operation varying the batch size and kernel size. Figure 7 plots of the time it takes to do the convolution operation for inference (the forward pass) and training (forward and backward pass), for all plots the number of input channels and output channels were fixed at 32 and each batch consisted of 256 by 256 tensors of full precision (32 bit) floats. The inference benchmarks used a batch size of 4 as online inference is typically done with a very small batch size, the training benchmarks are done with batch size 32 which is be more typical of training.

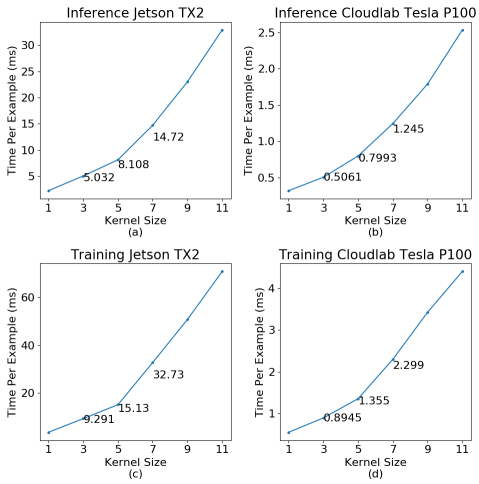

Figure 7: (a) and (b) where ran with batch size 4, (c) and (d) were ran with batch size 32

We note that there appears to be a penalty for increasing the convolution filter size beyond 5 by 5 as shown by the increase in slope from kernel size 5 to 7 on plots (a) and (c). Moreover, for the Jetson, performing a 5 by 5 convolution followed by a 3 by 3 convolution layer (which has the same receptive field as 7 by 7) takes less time than a single 7 by 7 layer, even with the overhead of cuda call. Interestingly, this penalty doesn't seem to be as pronounced when running the Cloudlab instance P100 GPU. This difference is clearly due to the hardware but we find no obvious characteristic of the hardware that explains our results.

Batch size is an important consideration while training as well as testing. The common belief is that a larger batch size provides better utilization of hardware. While this is true for the GPU's, this assumption doesn't really work for CPU. As illustrated by Figure 8 and 9,in our experiments we found that increasing batch size on a CPU doesn't produce significantly better speed but on the other hand increasing batch size on the GPU ends up.

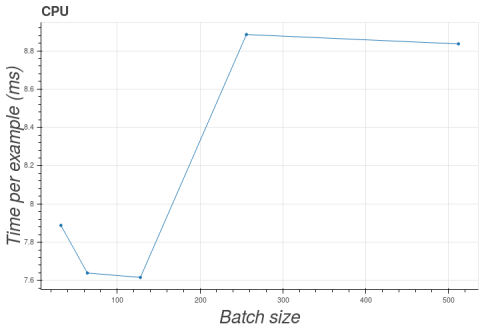

Figure 8: Time taken per image for a fixed batch size on a CPU

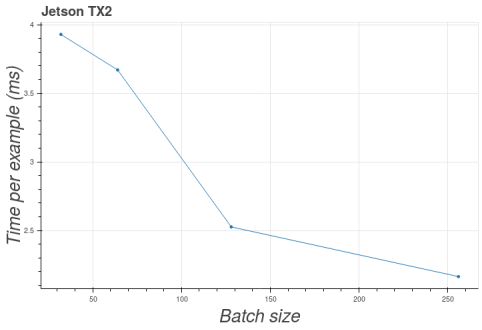

Figure 9: Time taken per image for a fixed batch size on Jetson

We believe these results underline the importance of profiling on the device which the training or inference is being done on whenever possible.

4.3 Training Variations

For our last set of experiments we wanted to explore ways to train the network faster. We tried multiple training variations on a Cloudlab instance with a P100 GPU. We opted to use the Cloudlab instance for our experiments because we were concerned with accuracy as well as run time, ergo we needed to run all experiments to

convergence, which takes significantly longer on a Jetson TX2.

Our first idea for a training variation was to only train a subset of the filters. We tested how feasible this technique was by applying it to LeNet on the MNIST data set. We chose LeNet because of it has a simple architecture and performs very well on the MNIST data set. The results of this experiment are shown in figure 10.

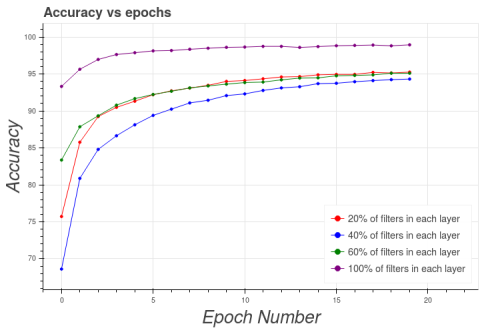

Figure 10: Accuracy of Training variations on MNIST

Our initial results were quite encouraging as there was only a slight decrease in accuracy, but significant reduction in training times. Comparing training the entire network to 60% of the network, there was only a 2-3% decrease in accuracy but more than a 50% decrease in training time when the batch size was set to 64. Interestingly we found that there was only a modest decrease memory usage of 14% between training the entire network compared to 60%. From these results we conclude that training a subset of the filters has potential to significant decrease the time it takes to train the network, but is unlikely to significantly decrease the memory usage during training.

Our next round of experiments we attempted to apply the same technique to Resnet 18 with harder data sets. Figure 11 shows the results for CIFAR-10, which is a moderately difficult data set. Variation a (red) is when we train hundred percent of the network (i.e. unmodified training), variation b (blue) corresponds to training twenty percent of the network, and variation c (green) corresponds to forty percent of the network.

Unfortunately, only training a subset of the layers with CIFAR-10 and Resnet 18 resulted in a 10 - 15% drop in accuracy which is far too large to make the technique feasible. We believe that the relatively small drop in performance on the MNIST data set was due in large part to how easy the data set is. With more difficult data sets this technique needs more tweaking to have the network converge to accuracy comparable to training the entire network.

During this evaluation we realized that a lot training for applications is done using the transfer learning approach [6]. In transfer learning the usual approach is to train the last layer but it is not uncommon for people to train the whole pretrained network. We tested this approach on the MINC data set [1], with Resnet 18 pretrained on the imagenet data set. The results are depicted in Figure 12.

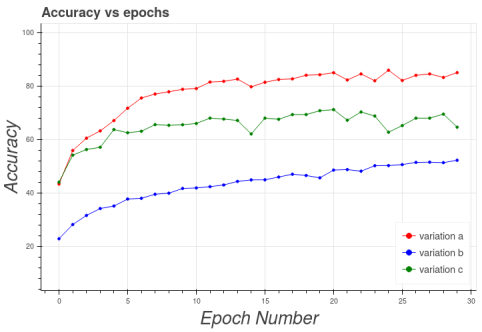

Figure 11: Accuracy of Training variations on CIFAR

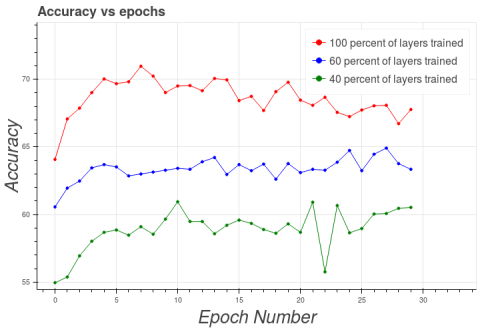

Figure 12: Transfer learning on MINC

Similar to the CIFAR-10 experiment, using transfer learning and training a subset of the layers each step resulted in a significant drop in accuracy when compared to training the entire network. We still think that in some cases transfer learning might an effective tool for reducing the amount of time it takes to train for a low power device. However, more work needs to be done to refine how the transfer learning is done to obtain the same accuracy as would be achieved by training the network from scratch.

5 Related Work

In our preliminary literature survey we didn't find any literature addressing the problem of training on the edge. However, there are some related papers exploring efficient model inference as we mention in the follow sections.

5.1 Quantization

Quantization is a common optimization of converted floating point tensors in the network into 8 bit integers before running inference on the input [8]. The advantage of quantization is it greatly reduces the memory footprint of the network, allow for more of the network to fit in the processor cache. Multiple schemes can be used quantization, including using predefined intervals and taking the min and max of the floats, which quantization scheme is used largely determined by the knowledge about the data or lack there of.

5.2 Efficient Reduction in size of Network

We looked at both Mobilenet V1 [3] and Mobilenet V2 [4] papers. They perform clever use of the idea that the complexity of convolution operation increases non-linearly, e.g. smaller convolution filters have fewer parameters so they are faster to compute. The other key insight in Mobilenet V2 paper is about using of residual connection between bottleneck layers. They also point out that residual connections between bottleneck layers give better performance than residual connections between expansion layers and they are still low on resources.

Mobilenet V1 [3] introduces the idea termed as Depthwise separable convolutions, so instead of using a single 3D convolution they use a 2D convolution followed by several 1x1 convolutions.

5.3 Binary Filters for Training

In [7], the authors come up with the idea of using binary filters instead of floating point numbers. The authors borrow the idea of Orthogonal Variable Spreading Factor (OVSF) code generation from the Wireless Communication community. Using the OVSF technique they generate a complete set of binary filters for a given size which forms a basis set. During the training process each filter in the convolution neural network is learned as a weighted average of the basis filters. So all that is needed to be stored is the weight for each binary filter, which is considerable smaller in size. This leads to huge reduction is memory at the time of inference as well as training.

6 Conclusion

Our first major challenge in trying to do training on low power devices was memory. Not only did we find that many networks required more memory that was available on many low power devices (i.e. greater than 8 GB), but even trying to profile the memory usage was a challenge. Neither TensorFlow nor PyTorch had adequate profiling tools for memory and currently cuDNN (which both TensorFlow and PyTorch use for GPU implementation) doesn't expose any memory usage information. Given the memory constraints on low power devices, optimizing for memory usage is a vital step in efficiently training on low power devices. We believe that improvements in memory profiling would greatly aid in designing networks that are intended for low power devices.

The next challenge of doing training on low power devices is the restrictions on compute resources. We began to explore this issue by examining the interaction between convolution filter size and time per example. Here we noticed that the run time versus filter size curve differed between machines, and for the Jetson TX2, filter greater than 5 by 5 are particularly inefficient. We expect that such differences will also be present in other operations in the network. We conclude that profiling should be done whenever possible on the device that the network is intended to run on.

Our final set of experiments was variations on training which attempted to reduce the total training time as well as the memory requirements for training. Our first variation of only training a subset of the layers we found to be effective for an easy data set (i.e. MNIST), however it lead to a substantial drop in accuracy for harder data sets. We attempted to ameliorate this problem by applying transfer learning with the same technique and found similar results as training the network from scratch. While our attempts failed to achieve the same accuracy as training the entire network, there are substantial gains to be made with these approaches and therefore these techniques which are worth investigating further.

7 Future work

As stated in the experimental design, we chose to focus on CNN's and did not address LSTM's or MLP's. CNN's are not well suited for many machine learning tasks which someone might want to perform on a low power device, such as translating speech to text, which LSTM's are far better suited for. The resource requirements for LSTM's and MLP's differ significantly from CNN's, and therefore we would expect that our profiling techniques would yield drastically different results when applied to these networks. Future work could apply the same profiling techniques which we present in the paper, possibly expanding them as well, and use them to optimize LSTM's and MLP's for running on low power devices.

Another aspect of network design that we did not address in this paper is how much power the training and inference are using on the devices we tested on. One of the draws of low power devices is being able to de-

ploy multiple devices for the same cost as a single server, hence while the power draw per device is far less than a traditional server, the power usage can still be significant. Reducing the power needed to do training and inference on these devices can then translate to significant savings in electricity cost given that enough are deployed. Additionally, in some settings the low power devices run on a battery for the lifetime of the device. Reducing the power usage has the potential to extend the life of these remote devices and/or reduce the size of the battery needed to run them, which reduces the cost of the deploying and maintaining the devices. Future work could further improve our approach by profiling the power usage and the trade off between power, memory and compute of the networks. Such information would allow users to further tune the network for their specific use case.

The most expensive operations in a CNN are the convolutions, however the convolutions are certainly not the only component of the network worth profiling and optimizing. Our preliminary exploration suggested that the gradient optimizer can have a significant impact on the memory usage and run time of the networks. In fact, for some networks switching from a more sophisticated optimizer (e.g. AdaGrad) to basic stochastic gradient descent was the difference from between being able to run the network and not. Further profiling is needed to determine what the trade off are between the number of epochs to convergence, accuracy of the network, run time, and memory footprint of various optimizers. We expect that many optimizers can themselves be optimized for running on low power devices.

Finally, we restricted our scope to running on GPU's, however many low power devices only contain a CPU. In our initial exploration we noticed that the CPU implementations for PyTorch and Tensorflow in some cases showed drastically different behavior from the GPU implementation of the same networks. We believe that these observations are in part due to the GPU implementations being far more optimized, but also as a result of the hardware itself. Future work could apply our profiling techniques to the CPU implementations of neural networks to try and optimize them for running on low power devices as well as elucidate the source of the differences between the CPU and GPU implementations.

References

[1] S. Bell, P. Upchurch, N. Snavely, and K. Bala. Material recognition in the wild with the materials in context database. *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[3] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[4] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2018.

[5] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[6] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu. A survey on deep transfer learning, 2018.

[7] V. W.-S. Tseng, S. Bhattacharya, J. F. Marqus, M. Alizadeh, C. Tong, and N. D. Lane. Deterministic binary filters for convolutional neural networks. pages 2739–2747. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[8] P. Warden. Building mobile applications with tensorflow, 2017.