StereoPhonic: Depth from Stereo on Phones

Robert Konrad*
Stanford University
CS 231N

Nitish Padmanaban*
Stanford University
CS 231N

Abstract

Estimating depth of scenes using stereo image pairs is a core computer vision task, which has benefited greatly in recent years from the application of neural networks. Depth estimation is also extremely important for mobile applications, for example, in virtual and augmented reality (VR/AR), but many mobile devices lack the same computational clout as the servers traditionally used to run networks. We make use of the ARM Compute Library to implement an efficient neural network–based depth estimation algorithm with GPU acceleration on a phone capable of taking stereo images.

1. Introduction

The problem of estimating three dimensional geometry from stereo imagery is a core computer vision problem that has applications in many areas such as robotics, self-driving cars, and most recently mobile augmented and virtual reality systems. While various types of 3D sensors have been used, cameras remain attractive due to their cost effectiveness and robustness to various environments. In this project, we focus on capturing stereo images and computing high quality depth maps on a mobile platform, the Huawei Mate 9. The phone has a dual f/2.2 camera, where one module has a 20 MP RGB sensor while the other has a 12 MP monochrome sensor.

The underlying problem for the depth from stereo task is finding the disparity between corresponding pixels of a rectified image pair captured with cameras, which can then be converted to metric depth with camera calibration. Despite decades of research, estimating depth from stereo pairs remains an open problem due to difficulties with textureless areas, reflective surfaces, thin structures and repetitive patterns.

Many stereo algorithms aim to mitigate failures caused by such cases by aggregating information from spatially local patches. Methods such as cost aggregation, semi-global block matching, and Markov random fields have been used with some success. However, each of these cost functions require hand crafted parameters or only learning a linear function of the features.

Figure 1. The device on which we conduct our final tests. It has RGB and monochrome sensors mounted with an 11 mm stereo baseline, and an ARM Mali GPU.

In recent years, the entire area of computer vision underwent a fundamental change by learning deep representations directly from raw pixel data as opposed to previously used feature based methods. These new algorithms, specifically convolutional networks, significantly improve performance on many high-level scene understanding tasks such as image classification, segmentation, and detection. Even more recently, convolutional networks have been exploited to learn disparity from stereo image pairs.

2. Related Work

Several approaches exist in the literature to address the problem of obtaining depth maps from stereo image pairs using deep learning algorithms. Kendall et al. [1] recently introduced a state-of-the-art framework that formulates the problem using a cost volume. The approach uses 3D convolutional layers to learn contextual information, which al-

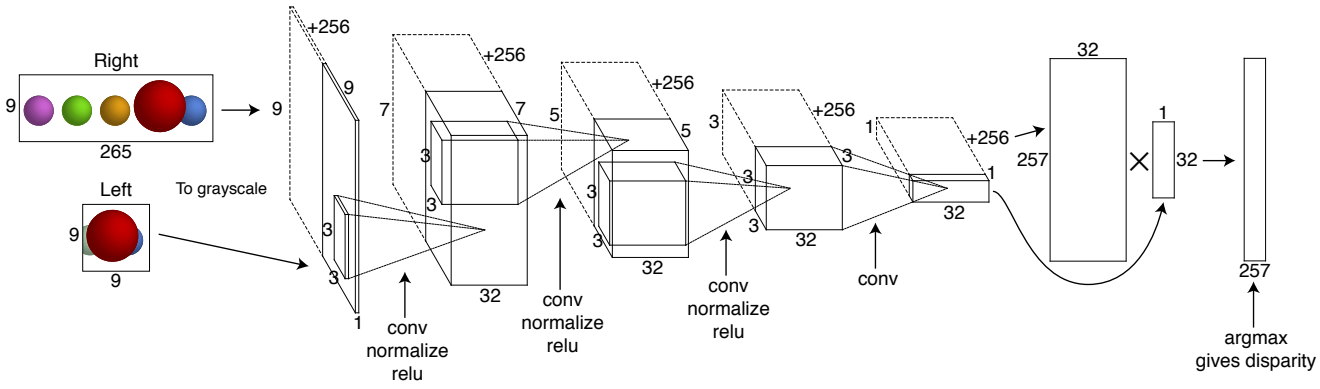

Figure 2. Diagram of the Siamese network architecture, based on Luo et al.'s work. Both left and right image patches are passed through 4 convolutional layers, and then the disparity is selected to be the index with the highest inner product between the two.

lows them to directly obtain a stereo map that can be used without need for smoothing or other post-processing. However, this represents a relatively complex network that may not adapt well to use on a phone. Another interesting approach used optical flow to aid in the calculation of disparity [3]. This implementation achieves competitive performance with a runtime of less than a tenth of a second, but requires a dense ground truth depth map to train, which required synthetic data.

However, these implementations work on the entire input image pair and learn more spatial and semantic information. This leads to them having millions of trained parameters. The Huawei GPU lacks dedicated main memory, with only at most 2MB of L2 cache on the GPU. For computational efficiency, we find it necessary to forgo these approaches.

There are two approaches we consider here. First is work by Žbontar et al [5]. They use a Siamese network architecture that operates on a patch-level. They propose both a fast and slow, but more accurate, implementation. The fast version computes disparities based on a dot product of the output of the left and right patch's sub-layers. The slow version instead concatenates the outputs of the sublayers and follows those with several fully-connected layers. Unfortunately, the slower network also takes about a minute to run, making it unsuitable for mobile environments. Luo et al. [2] implemented a similar approach with a patch-based Siamese network that terminates with a simple dot product. Their implementation is however, faster than that of Žbontar et al., and more accurate. The number of features is also relatively small, coming in at only about 100,000 for their 19×19 receptive field model, though we use a smaller 9×9 receptive field to allow for faster computation.

3. Dataset

We use the KITTI 2015 dataset [4] which comprises of 400 dynamic scenes with 4 color images per scene, captured from a moving vehicle with ground truth supplied by LIDAR. This has the result that not every pixel has a ground truth depth due to the way LIDAR scans work, so our selection of training patches must take this into account. Furthermore, as described in the methods section we only train on monochrome images, hence we convert the RGB images to grayscale first. Because the images are 370×1224, which is comparable to the size of the images we will be operating with on the phone, we will train on the full-sized images. Since we use a patch-based method to predict each pixel, the 200 images in the training set represent several million patches to learn.

Interestingly, there is a difference in the baseline between stereo images in the dataset we train on and the stereo images from the phone (11 mm). Due to the patch based architecture of the network, we believe these features will transfer from the dataset to the captured images. In fact, due to the reduced baseline there are less views that appear in one camera but not the other, making the correspondence matching problem significantly easier. This comes at the trade-off of a potentially enlarged depth error estimate at farther depths.

4. Model and Design

We adapt the Siamese network architecture proposed by Luo et al. for our disparity estimator. In our case (Figure 2), we use 4 convolutional layers in the sub-networks. Each layer has 32 filters of size 3×3, and is followed by a spatial batch normalization and ReLU. Our left input patch size is 9×9, to match the receptive field of the sub-network. The left sub-network outputs a vector of length 32. The right input is 9×265, to allow for 257 discrete disparity values, and results in a 257×32 output from its sub-network. These two are combined with a simple inner product, resulting in a vector $\hat{p} \in \mathbb{R}^{257}$. The loss for example i is then computed as the cross-entropy of this probability estimate against a

ground-truth probability based on the known pixel disparity,

$$L_i = p_i^T \log \hat{p}_i. \quad (1)$$

Here, the ground truth p_i is defined as 0.5 in element location corresponding to the correct disparity offset, 0.2 if off by 1 pixel, and 0.05 if off by 2 pixels. All other ground truth probabilities are defined as 0.

For the training phase, we convert all the images to grayscale (required here in order to maintain a Siamese network for the final phone implementation because one of the cameras only captures grayscale information), and use input patches of size 9×9 from the left image, and 9×265 from the right, based on our receptive field size and number of disparities. The patches were centered around pixels with ground truth disparity values, and chosen in random minibatches across all training images. For efficiency during the test phase, the sub-network can be run in parallel on all pixels in the image, using a single pass through the convolutional layers. The resulting $\hat{p}$ vectors can be reused as needed for the inner products [2].

The training and validation are done server-side in TensorFlow[1] to take full advantage of the higher computing power. However, due to the benefits of having a mobile implementation mentioned earlier, we implemented the forward pass of the network on the Huawei Mate 9. We used the recently released open source ARM Compute Library, provided by ARM, for low level optimizations of many commonly used machine learning and image processing operations, including certain neural network layers. While using a low-level library on a mobile device makes for a less-than-ideal workflow, recent announcements such as Caffe2Go, TensorFlow Lite, and Core ML may introduce higher-level and streamlined APIs for porting desktop-trained networks to mobile.

In the following sections we investigate hyper-parameter tuning to search for the best model taking into account the limited memory constraint of the phone, as well as performance optimizations in order to reduce computation time to minimum.

4.1. Hyperparameter Tuning

For our network, we primarily experimented with tuning the learning rate, gradient step optimizer, and the batch size. TensorFlow's default initializers in conjunction with batch normalization was sufficient to obviate the need for much tuning of the weight initialization. Furthermore, as seen in Figure 4 for our final network, we do not suffer from overfit to the training data. This was true for every network we trained, including 19×19 receptive field network, with 64 filters in each of the 9 layers, likely to due to the large number of patches in the dataset. Therefore, we found it unnecessary to apply regularization. A summary of some representative results[2] of various hyperparameters can be found in Table 1. Our chosen architecture for running the network on a phone was 4 layers, with 32 filters per layer, but we also experimented with 16 or 64 filters per layer, and 2, 3, or 9 layers, in case the phone proved more or less capable than we expected. All following results tended to hold regardless of network size, though we did not run all experiments on sizes besides 4 layer with 32 filters. We conclude from these experiments that the loss function for this network is extremely well behaved, with similar results across a wide range of parameters.

Table 1. Architecture and Hyperparameters

Architecture		Hyperparameters			Training		Validation	
Layers	*Filters/layer*	*Learning Rate*	*Optimizer*	*Batch size*	*Loss*	*% > 3px*	*Loss*	*% > 3px*
9	32	0.1	Adam	512	5.5491	100	n/a	n/a
4	32	0.01	Adagrad	512	3.4358	34.23	3.3875	32.6
4	32	0.01	RMSProp	512	3.2485	30.39	3.2319	29.43
4	32	0.001	Adam	512	3.2577	30.77	3.2208	29.17
4	32	0.01	Adam	128	3.2449	30.17	3.226	28.85
4	32	0.01	Adam	1024	3.1834	29.25	3.1968	28.78
4	32	0.01	Adam	256	3.197	29.43	3.1969	28.69
4	32	0.01	Adam	512	3.2002	29.5	3.1833	28.39
4	64	0.001	Adam	512	3.0807	27.9	3.0658	26.59
9	32	0.01	Adam	512	2.6154	19.58	2.7334	19.82
9	64	0.01	Adagrad	128	2.2317	11.59	2.1731	9.63

4.1.1 Learning Rate

We first attempted to determine a range of learning rates for which the network converged best. At one extreme, a learning rate of 10^{-1} caused exploding errors, and at the other, learning rates 10^{-4} or lower tended to converge observably slower. While the 10^{-3} learning rate trained slightly slower in general, it was not significantly different after an epoch or two. The 10^{-3} learning rate didn't seem to find a lower eventual minimum, so we turned to learning rate decay schedule.

Here, we tried various decay step rates ranging from after 1000 to 5000 iterations, at a rate of 0.9 or 0.95. Despite running this for over 100 thousand iterations, we failed to see any significant further reductions in loss or increases in accuracy. Thinking that it was possible that the learning rate had decayed too much, we loaded a warm-start of the network after 3 epochs (78500 iterations) of training with decay, then re-started training at a rate of either 10^{-2} or 10^{-3} with decay. This also failed to improve results.

[1] Code for `run_model` based on assignment 2 TensorFlow notebook: http://cs231n.github.io/assignments2017/assignment2/

[2] The table was generated with loss and accuracy using the disparity mapped into the reference frame of the right instead of left image, which we did not discover until after extensive tuning attempts. A few experiments showed similar, but improved results with the correct mapping, so we did not re-attempt the full suite of tuning experiments.

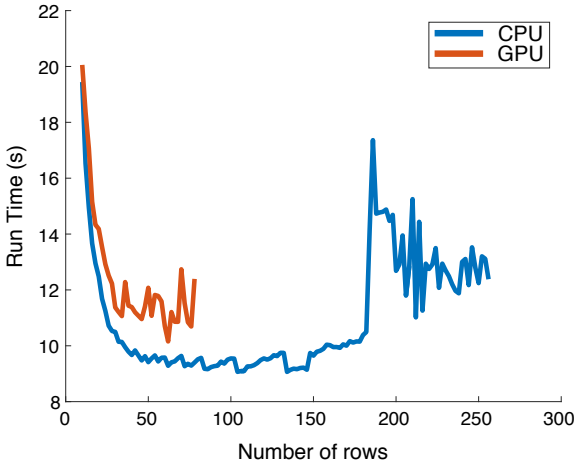

Figure 3. Run time of a single branch of the Siamese network on a full 370×1224 image as a function of the number of rows that the the image is blocked into per run. The largest block size is limited by the hardware capacity.

4.1.2 Gradient Step Optimizer

While we primarily used the Adam step optimization for its general high performance in learning, we also briefly compared results with other gradient step optimizers, particularly RMSProp and Adagrad. While Adagrad took longer to converge than the other two, the results were largely similar for all of them.

4.1.3 Mini-batch Size

We initially chose the batch size of 512, since it was the largest we could use in training based on the memory requirements of the 9 layer, 32 filter architecture. However, since our target was 4 layers, it was possible for us to try increasing the batch size as high as 2048 to perhaps get smarter steps. Unfortunately, this failed to improve final performance. We also tried smaller batches to perhaps get "lucky" with the more random updates, but this also did not affect results.

4.2. Forward Pass Phone Optimizations

In order to maximize performance of the forward pass on the phone we investigated multiple optimizations and algorithms at various points in the architecture. Just using a small network was not enough, as explained in the results section.

4.2.1 Siamese Network Optimizations

Removing Batch-Normalization We note that all the batch normalization layers in the network immediately follow convolutional layers. Since the mobile network only needs to run the forward pass, it doesn't benefit from the explicit structure of a batch normalization layer, meaning that it does us no harm to simply fold the batch normalization into the convolutional layer. Let k and b be the kernel and bias for a single channel of the convolutional layer output, and let μ, σ^2, β, and γ be the moving average, moving variance, center offset, and scale parameters for that channel's normalization, respectively. With this, we can compute the equivalent convolution kernel and bias, k' and b', as

$$x' = k * x + b \tag{2}$$

$$\begin{aligned} x'' &= \left(\frac{x' - \mu}{\sqrt{\sigma^2 + \varepsilon}}\right)\gamma + \beta \\ &= \left(\frac{k * x + b - \mu}{\sqrt{\sigma^2 + \varepsilon}}\right)\gamma + \beta \\ &= \left(\frac{\gamma}{\sqrt{\sigma^2 + \varepsilon}}k\right) * x + \left(\frac{(b-\mu)\gamma}{\sqrt{\sigma^2 + \varepsilon}} + \beta\right) \\ &= k' * x + b' \end{aligned} \tag{3}$$

$$\begin{aligned} k' &= \frac{\gamma}{\sqrt{\sigma^2 + \varepsilon}}k \\ b' &= \frac{(b-\mu)\gamma}{\sqrt{\sigma^2 + \varepsilon}} + \beta \end{aligned} \tag{4}$$

where x, x', and x'' are the input, intermediate before normalization, and output, and ε is some small constant. We simplify our trained weights and biases using this approach before loading them onto the device, reducing the complexity of the mobile network.

Image Cropping The limited memory of the phone prevented us from running an entire image through the Siamese network at once. Instead, taking advantage of the flexibility of the patch based method, we are able to run subsections of the image through the network at a time and then stitch the outputs into the full feature map. To minimize cache misses, we copy over multiple rows of the image at once due to the data being stored in contiguous memory. The results of performance tests run on a 370x1224 input image with varying crop sizes is shown in Figure 3 for both CPU and GPU implementations. Due to the even further reduced GPU memory, we were only able to load up to 80 rows at a time for the GPU implementation. Clearly, the CPU implementation outperforms the GPU implementation across the board, which we assume to be due to the slow memory copy to GPU memory and the GPU not having many cores (32 GPU cores vs 4 CPU core in the Mate 9). The spike in run time around the 185 row mark for the CPU implementation is interesting, and we reason to be due to the fact that a large portion of the last batch has wasted compute cycles for a 370 row image.

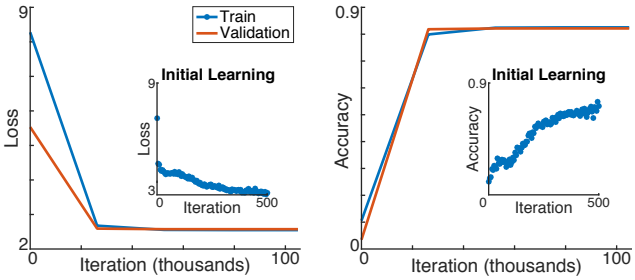

Figure 4. The loss and accuracy curves for our final trained network. Accuracy is defined based on percent with over 3 px of error in the disparity estimate. We see that the validation and the training results are extremely similar in both cases, indicating no overfit. The insets show the initial results from the first 500 iterations, after which the network is almost done training.

4.2.2 Inner Product Optimizations

We also consider the impact of different methods when computing the final inner product layer. In this layer the core computation is the inner product between the 32 features for a position in the left image with the features of each of the possible 256 pixel disparities from the right image, followed by an argmax over the resulting 256 dimensional vector (Figure 2).

Our initial approach was to parallelize the computation as much as possible: take the entire 370×1224 left image, pointwise multiply with a 370×1224 region of the padded right image, add to compute the dot product, and then shift and repeat while keeping track of the argmax at each pixel. Due to memory constraints, we performed this operation on 37 rows at a time.

Despite the operations being implemented with ARM Compute Library functions on the GPU, this implementation took 4 minutes for 10% of the image (the full image was 370x1224) – clearly too long. We suspected this to be the result of wasteful copies to the from the GPU. To investigate the impact of these memory copies we implemented the naïve nested loop CPU implementation, which is very memory efficient, but computationally inefficient. We observed a decrease in run time for the 37x1224 image to 3 minutes. With these initial results from the two extreme scenarios we concluded that both memory inefficiencies and computational inefficiencies are capable of bottlenecking the system when completely ignored. We therefore investigated various methods that trade these two off to varying degrees:

1. The first method was based on the theory that the memory issues were due not to simply copying it too often, but rather that the copying occurred over several non-contiguous blocks of memory; since the right image is padded extra to allow for keeping a parallelizable implementation at the edge pixels, reading multiple rows of the image at once likely causes more cache misses than otherwise, only exacerbated by having to read more and more rows. Therefore, we tried to do the same operation as before, but for a single row at a time.

2. The second method assumes that the problem is primarily caused by too many memory moving instructions, despite fully 256 of the 257 columns in the buffer already needing to be reused. While this seems like it could be solved by treating the right image's columns as a circular buffer, that would unfortunately then require the *left* image to have its columns shifted in memory to match up for the next disparity. We need some way to load each pixel's features once for a single GPU operation, and never again. However, there isn't an operation that allows us to simultaneously load all the pixels, and then apply the inner product for only the left/right pairs we desire. Therefore, we take the closest approximation, choosing to waste computation in exchange for having only a single memory copy: a matrix multiplication of an entire 32×1224 row of the left image features with the padded 32×1480 row of the right. This will compute all the requisite inner products, but also 5 times as many extra ones. The matrices in question may be small enough to be fully parallel on the GPU however, mitigating some of the temporal cost. We also note that the extra computations would be reduced when switching to images captured by the phone, which will only be 480 px wide after downsampling. It is also important to note that this method also benefits from the increased memory locality desired by first, since the matrix multiply forces a single row at a time. However, if memory locality and not redundant copying is the root cause of slow operations, the extra computation required here should show this method to be worse.

3. The third method follows the rationale of the first, but also considers whether any possible improvements of the second method may be caused by the matrix multiplication implementation simply being faster than pointwise multiply and sum. We apply the matrix multiplication as before, but only to a single 32 feature vector in the left image with the corresponding 257 vectors of the right image as a 257×32 matrix. This is the least parallelized method, doing only a single output pixel at a time, but wastes no computation.

Our results showed that the second method was best, indicating that our concern over redundant, repeated memory access and copying were warranted. The resulting runtime for 37 rows was merely 6.5 seconds.

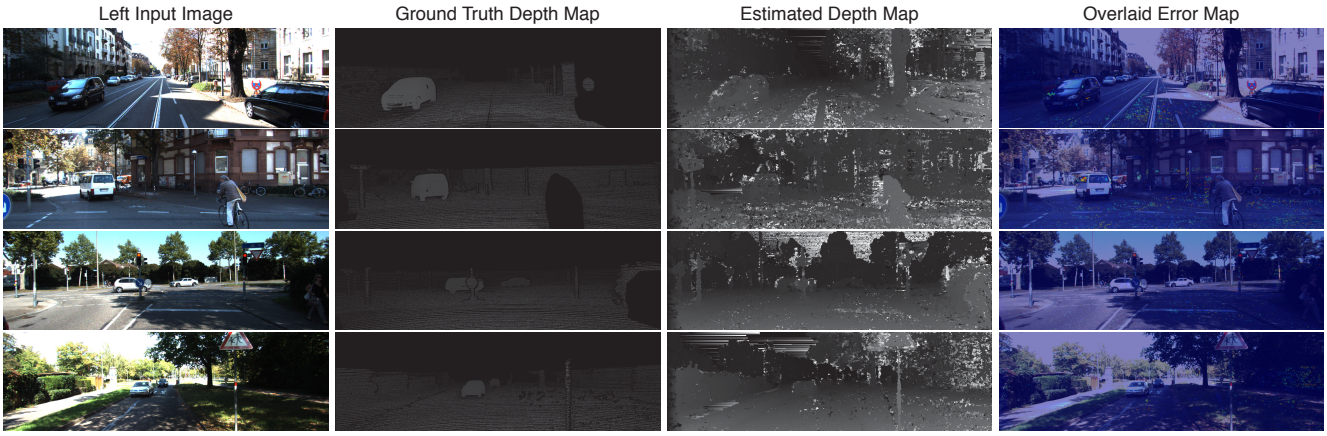

Figure 5. Some example results from our trained neural network. The columns, in order from left to right, contain the original left view input image, the ground truth depth map, our network's estimated depth map, and the error overlaid on the image (blue low error to red high error). It can be seen that the most common errors occur in regions of repetitive or no texture, similar to most patch-based algorithms for stereo estimation. Note that the ground truth map is 0 (black) where there are no ground truth estimates.

5. Results

5.1. Trained Performance

We chose as our best network the 4-layer, 32-filter one with highest validation accuracy. This corresponded to a learning rate of 10^{-2} with decay 0.9 per 1000 iterations, a batch size of 512, and the Adam optimizer. As seen in Figure 4, our network converges quickly and does not overfit. We attain a training accuracy of 82.5% of pixels with an error of 3 px or less, compared to 82.1% on our validation set and 80.3% on our test set. This corresponds to an average error of 7.5 px from the true disparity estimate for all cases.

We also note that when using the 9-layer, 64-filter network, we achieve 90.4% accuracy on our validation set. When comparing this to the un-processed output of Luo et al.'s network, we see that they achieved 91.1% accuracy. These are similar enough that the difference may be caused by a slightly different validation set, meaning that we can be confident in having accurately reproduced their work.

5.2. Phone Runtime

While our initial attempt took 3 minutes for only 10% of the image, our optimizations proved useful, with the final phone implementation taking only 56 seconds on a full KITTI dataset image for the optimized CPU implementation, and 58 seconds for the optimized GPU implementation. While this is certainly 120 times slower than the model on our server-side GPUs, which take 0.46 sec for their computation per KITTI image, it is within reason for a phone with far fewer resources at its disposal. We were unable to implement the full pipeline from taking the photo to outputting the depth map due to only one of the stereo cameras being visible to the Android SDK. A special Huawei SDK would be needed to access RAW data coming off of the stereo camera module to implement the full pipeline.

5.3. Qualitative Evaluation

The images that result from running our trained network over the KITTI dataset can be seen for some examples in Figure 5. The efficiency of the forward model of our network comes in part from the use of an inner product to combine the left and right images. However, since this is similar to the process used by classical patch-based stereo matching approaches, we suffer from many of the same failure cases. As can be seen in the figures, the most common locations of errors tend to be places with occlusions, with spatially repeated textures, or simply flat color regions.

Many of these issues can be addressed using post-processing, such as simple smoothing or more sophisticated algorithm. This was done by the by Luo et al., and dramatically improved accuracy on average [2]. However, since our primary objective was to have a neural-network based stereo matching model running on a phone and evaluate the performance of the network itself, we did not apply any post-processing.

5.4. Discussion

This network does well given that it uses a patch-based approach with only a 9×9 receptive field. However, we were conservative in our choice of network architecture, and thus, it may be possible to load larger, better-performing networks onto the phone, or different, complex, architectures entirely. It would also be useful to see the best CPU–GPU optimization across the functions, since some parts of the model, such as a single branch of the Siamese network, seem to work better on the CPU.

6. Conclusion

We were able to successfully port a neural network trained in Tensorflow on a server, and transfer it to a mobile architecture using a low-level mobile GPU library. Mobile computing is becoming, if not already, among the most important modes of computation. Though many learned models can be queried remotely, many mobile applications, especially in spaces such as VR and AR, require reliable and fast results, which must be done on-device. Given the recent dual trends toward mobile computing and neural networks, their combined used, as evidenced by recently announced frameworks such as Caffe2Go, TensorFlow Lite, and Core ML, is inevitable. We investigate the potential for one such use case, and achieve promising results, though there is certainly more work to be done to find the optimal balance between computation time, network size, and accuracy.

References

[1] A. Kendall, H. Martirosyan, S. Dasgupta, P. Henry, R. Kennedy, A. Bachrach, and A. Bry. End-to-end learning of geometry and context for deep stereo regression. *arXiv preprint arxiv:1703.04309*, 2017. 1

[2] W. Luo, A. Schwing, and R. Urtasun. Efficient deep learning for stereo matching. In *CVPR*, June 2016. 2, 3, 6

[3] N. Mayer, E. Ilg, P. Husser, P. Fischer, D. Cremers, A. Dosovitskiy, and T. Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *CVPR*, June 2016. 2

[4] M. Menze and A. Geiger. Object scene flow for autonomous vehicles. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. 2

[5] J. Žbontar and Y. LeCun. Stereo matching by training a convolutional neural network to compare image patches. *Journal of Machine Learning Research*, 17(1-32):2, 2016. 2