Analysis of the effect of caching convolutional network layers on resource constraint devices

Wouter van Lil , Lydia Chen , Bart Cox , Masoud Ghiassi
TU Delft
wvanlil@student.tudelft.nl

Abstract

Using transfer learning, convolutional neural networks for different purposes can have similar layers which can be reused by caching them, reducing their load time. Four ways of loading and executing these layers, bulk, linear, DeepEye and partial loading, were analysed under different memory constraints and different amounts of similar networks. When there is sufficient memory, caching will decrease the loading time and will always influence the single threaded bulk and linear mode. On the multithreaded approaches this only holds when the loading time is longer than the execution time. This depends largely on what network will be run. When memory constraints are applied caching can be a way to still run multiple networks without much increased cost. It can also be opted to use less memory on a device and use transfer learning with caching to still get the same results.

Keywords

Deep learning, partial loading, caching, transfer learning

1 Introduction

The devices we use in our everyday life are getting smarter and faster. As this trend is increasing and their capabilities grow, we can start using them for more difficult tasks. Examples of fields in which we can use these edge devices are cognitive assistance, physical analytics, mobile health, memory augmentation and lifelogging. Some of these tasks use mobile vision, which take visual data as input and compute the desired application result using a convolutional neural network. Although these devices are improving, they are still not able to perform costly tasks such as training a network or sometimes even running an inference. Many edge devices overcome their limitations by sending their data to the cloud to be computed elsewhere. This puts the main constraint a device has as the connection speed. The data sent can contain sensitive information which imposes a privacy risk. Another drawback to using this method is that a high bandwidth is required when the data becomes large. Being able to do the computations locally has its benefits. One of the places to increase the capabilities of edge devices is with caching. This paper will look into what strategies benefit most from caching on executing convolutional neural networks and what the impact of it is.

2 Related Work

On these resource constrained devices it is even more crucial to find clever ways to avoid doing unnecessary instructions. Previous works have made progress in this field. DeepEye makes an early analysis of the input picture to detect whether or not an inference on a similar image has been done recently [7]. If so, the cached result of this inference is returned and the entire inference can be omitted. The idea is to prevent having to do the work twice if the outcome will most likely be the same anyway. The problem with this is how image similarity is defined. Images with a stationary background and a moving foreground might still inadvertently classify as similar, even though the main subject has been altered and the result of the inference would be different. DeepMon recognizes this problem and improves upon this by comparing small blocks of the image, and caching the results of applying the convolutional layers to these [4]. In the previously stated example the background blocks have been cached and can be reused, while the changed foreground elements are checked again. The similar blocks in the image do not have to be computed again, and with the changed blocks being recomputed the outcome of the entire image can be reconstructed from the separate parts. This is especially advantageous as a large part of the execution time comes from these convolutional layers. Later work has enhanced this by matching these blocks to different locations on the cached image by using diamond search, a technique used in video compression [10; 11]. This ensures that even if the camera has moved sections might still match and can be retrieved from the cache.

In many fields there is a lack of training data available. A solution that has been found for this is to use Transfer Learning. An existing convolutional neural network that has been trained to do a certain task, can have specific layers retrained to fit the new task at hand [8]. This makes use of the abstraction levels introduced by the original network and prevents overfitting of the network due to a lack of training data. Previous work has shown that these networks trained for very different scenarios can yield positive results when applied in

a different field [3; 1; 2]. It should be noted that this does not always hold as there are cases where the network is better when retrained [9]. A side effect of this is that different networks might have similar layers which can be cached so that the loading time of the inference is decreased. The approach of caching layers has been to take the largest layers possible and cache those [7]. If the similarities between layers are frequently occurring this solution is suboptimal.

Another approach to optimising networks with similarities has been proposed by Jiang et al. They present Mainstream, when two inferences from different networks share layers, these layers only need to be computed once [6]. Whenever the networks branch off in different directions, the result up to that point is used for both individual computation of unshared layers. This makes sure that the earlier shared layers of a network only need to be executed once. As a side effect of this, the layers only need to be loaded once as well. This would eliminate the need for caching similar layers as they only need to be loaded once anyway. Important to note here is that the specific case this works is when the inferences are done on the same input image and at the same time. If this is not the case this method fails to get any improved results.

3 System Scenarios

3.1 Deep Learning Framework

As a choice for handling the operations of the convolutional networks, Caffe is used. This is a framework which provides all tools necessary to use and manipulate deep learning. It offers a broad variety of layer types and pre-trained networks through a standardised format. Caffe is designed with expression, speed, modularity, openness and community in mind, aiming at reproducibility and academic research [5]. Caffe reduces the convolutional computation problem to matrix multiplication and makes use of BLAS libraries, which are highly optimised at this for almost all platforms.

Built on top of this is EdgeCaffe, which focuses on splitting a deep neural network into separate layers and being able to run it partially, meaning independently layer by layer. This provides many possibilities for improvements as there is more control over how the networks are loaded and executed. Aside from having previously used strategies implemented, it also provides its own novel way of loading and executing. This is discussed in the next section on partial loading.

Every layer of the network is divided into two tasks that need to be completed in order for the layer to be considered finished. First, the loading task, where the layer has to be loaded into memory from the disk. The second task is the execution task, where the layer type dependent operations are executed. This tasks has some prerequisites that need to be fulfilled before the execution can begin, namely the layer has to be loaded into memory, and the previous layer needs to have completed its execution in order for there to be input to the current layer. Once all of the layers are done executing the network is done with the inference and the result has been computed.

3.2 Modes of operation

The splitting of the network into layers allows for different strategies of the ordering of tasks and the use of other features such as multithreading. These different modes of operations each provide a unique method with its own advantages and disadvantages. When analysing the added benefit of using caching, the four modes provided by EdgeCaffe are tested and compared.

Bulk

Bulk mode can be seen as the most traditional approach on running convolutional neural networks. The entire network is loaded into memory first. When this is done, all the layers will execute one after the other. The memory used by this mode is high, as layers are in memory that are not currently used. This mode does not gain anything out of the fact that the network is split into layers.

Linear

With linear mode, the tasks of loading and executing are alternated. First a layer is loaded, after which it is immediately executed. This is done in a linear fashion, as a layer cannot be executed before its previous layer is finished. By having the network split into separate layers, once a layer is executed it can be released from memory as it is no longer needed for the current inference. Layers that are not handled yet have not been loaded, which means the memory they would take up is also free to use.

DeepEye

The method used in DeepEye is described as runtime interleaving [7]. The executing of convolutional layers and the loading of fully connected layers takes up most of the time of an inference. Two threads are created, one for each of these tasks. While the *convolution-execution* thread is loading and executing the convolutional layers of the network, the *data-loading* thread is simultaneously loading fully connected layers. Once the *convolution-execution* thread has calculated its output by executing all its layers, the *data-loading* thread will take this as input and can start to execute the fully connected layers to classify the inference input. This essentially splits the network up into two parts, with one thread for each segment.

Partial

The last mode used in the analysis is uniquely from EdgeCaffe. Partial loading uses multiple threads like DeepEye, but does not have specific tasks for each of these *worker* threads. Instead, task pools are made where every available task is stored. These tasks are either the loading or the execution of a layer. All the loading tasks are available immediately. Before a layer can be executed though, it has to be loaded and the previous layer has to have finished its execution, since the current layer needs input. Once these conditions are met, the task is placed in the task pool and the *worker* threads can execute them. Two *worker* threads can have the same interleaving effect described in DeepEye, but partial loading has the advantage of being able to do so in loading all layers instead of just the fully connected ones. Another benefit of partial loading is that less memory is required. Instead of starting

with loading the fully connected layers and having to keep them in memory, other layers can be loaded and executed, enabling this memory to be freed after the execution.

3.3 Memory

Running an inference on a network will yield different results depending on the available memory. When the available amount of memory is exceeded, the system makes use of a swap file. This is virtual memory that is stored on disk, meaning the access time will be slower than when accessing physical memory. As more memory is available, the swap file will be accessed less, decreasing the time taken to fetch the data. Since the inference is not done on an actual constraint device, this is simulated by using control groups. Control groups is a feature from the Linux kernel which allows the user to set a custom memory limit. After this size is exceeded, the system will make use of the swap file, which can also be altered in size. If the swap file is exceeded the program is terminated as continuing without sufficient memory will lead to unpredictable results.

4 Caching

Different approaches can be taken to optimise the inferences based on what the conditions are. The circumstances in Mainstream require the input image to be the same and the arrival time of the inferences to be before the moment where the branches are diverging.

This paper will look at a very different case. First, the inferences can, but are not required for improvement, have different input images. An important use case for this is if there are multiple cameras on the device that all need to run the same networks simultaneously. Further more the arrival time can vary. Having multiple images taken seconds apart and running the same inference on each of these can increase the certainty of this being the right result.

Convolutional neural networks take up most of the time executing the convolutional layers and loading the fully connected layers. The caching of layers will allow the program to fetch them from memory instead of looking them up from disk, saving time on the loading part. The caching of the convolutional layers will have a smaller impact but also requires less space in memory. The impact of this will depend on the strategy used.

To determine whether or not two layers are similar, a hash value is used. This ensures that knowing if it is already stored does not require loading the entire network. These hashes can be generated offline, meaning the overhead is only in comparing the hashes, not the calculation of it. If the layer has already been seen and is loaded into memory, the cached version can be used preventing costly loading from disk.

With transfer learning, the idea is to use a pre-trained network and alter it slightly for the specific task. This is often done by freezing all layers except for the last softmax layer, which outputs the probability for each class label. By doing so the entire network except for the last layer stays the same. When running the tests this structure is applied and when networks were required to be similar all layers except for this last one corresponded to the layers and hashes in different networks.

EdgeCaffe provides a clear division in how the tasks are handled. The code of the tasks where layers have to be loaded is extended. Before the layer would immediately be loaded from disk. As of now the check is first done whether or not the hash of the layer has been seen already. This check is done through a single unique map. On the first inference no layers have been cached yet. The first layers will be loaded from disk as usual, and a pointer will be stored to the location of these individual layers. After this the layers will be separately loaded into the network that requires it. Whenever a layer with the same hash is requested later on, its memory location from the cache can be retrieved from the map. The layer can then be loaded into the network. As some of the modes of operation use multithreading, it is important to make sure these do not interfere. When accessing this map, mutexes are used, but should only be held for this duration, and not for the time loading the memory from disk. This ensures that the multithreaded approach suffers minimal timeloss when multiple threads want to access this at the same time.

5 Analysis of Caching Policies

5.1 Testbed and workloads

As a start, four networks are compared. These are AgeNet, FaceNet, GenderNet and SoS GoogleNet. As AgeNet appeared to be most useful for testing caching, the later results are done with this network. AgeNet consists of 3 convolutional layers and 3 fully connected layers. The Caffe model is 45.7MB in size, with 86.8% of the size being the fully connected layers. Every statistic presented in the evaluation section consists of 20 to 25 runs.

With the tests run, some of the layers of networks would be considered similar and thus reusable from cache. Whenever two networks are similar, this means that all their layers are similar, except for the last fully connected layer. Why this is the case is argued for in section 4. In the figures presented in this section, the amount of similar networks is denoted by (X/Y), where X is the amount of networks that were similar out of Y networks total. As an example, $(3/5)$ means that in total five networks were run, with three networks sharing similarities. One of these needs to be loaded from disk after which the others can use the cached data. The other two networks not similar required loading from disk. If the notation is not stated or is $(1/Y)$, none of the networks share similarities and everything needs to be loaded from disk.

Partial loading can be run with different amounts of threads. In the test cases two *worker* threads are used. This was chosen as the the main advantage of multithreading is to be achieved from having the execution and loading side by side.

The machine the tests were run on uses an Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz processor. The memory with a capacity set by the control groups is DDR3. A hard disk with 6.0Gb/s SATA was used.

5.2 Results

First, it is important to look at how the time spent on each part of the network is distributed. All of the networks were run a total of 20 times on a single thread, with the average

	AgeNet	FaceNet	GenderNet	SoS Google
LoadTime	421.81	1009.22	339.01	278.31
CachedTime	164.87	576.22	118.00	196.63
ExecTime	120.03	157.63	790.04	4170.08
RunTime	642.05	1505.83	1256.02	4898.02
RunTime*	423.14	1019.07	1108.02	4772.26

Table 1: A comparison of the load, cache retrieval, execution and runtime of four different networks. The runtime indicated with an asterisk is the runtime with caching applied.

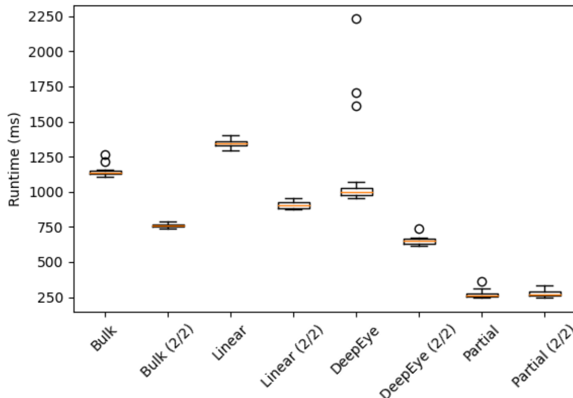

Figure 1: AgeNet is run twice with 1G of memory available on the different modes of operation

of each statistic taken. The results of this is shown in table 1. The *LoadTime*, *CachedTime* and *ExecTime* labels are the time taken for solely the layers, while the *RunTime* displays the aggregate of the entire run with overhead of the program included. The *LoadTime* and *CachedTime* are mutually exclusive and for a run only one applies, as the layer needs to be loaded from either memory or disk. One thing that is clear to see is that the time it takes to retrieve a layer from cache is significantly faster than retrieving it from the disk. This works for all networks, with the best decrease in time taken being 65.19% and the average being 49.59%.

When looking at the total runtime of the networks, SoS GoogleNet is the one that is relatively changing the least. This can be explained by the unbalanced distribution of the time taken by loading or retrieving from cache, and executing layers. Applying any form of acceleration to the loading time of this network will not yield significant results to the overall time. From this example it is already clearly visible that the caching of layers does not have a great impact on every network, and in order to get the most from this technique the right networks should be used. The other three networks have shown a good response to caching. AgeNet is the one that is closest to having the time taken for the loading tasks and execution tasks equal. In theory this should provide the best results as the tasks can run simultaneous.

In figure 1 we can see the modes are run with sufficient memory supplied so that the swap file does not need to be

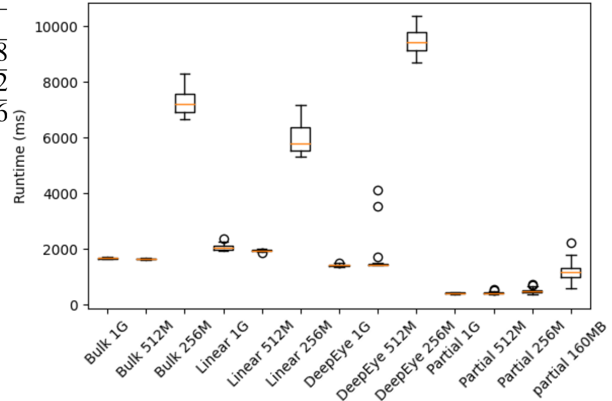

Figure 2: Three networks are run with varying amounts of memory. The networks are not cached.

accessed. A clear improvement in runtime can be seen in most of the modes, with the exception being partial loading. With bulk and linear mode, the improvement seems obvious. Since these modes are single threaded, the time waited for the disk access has been shortened by using the in memory layer instead, which will decrease the total runtime by the same amount. With the DeepEye mode the two threads do not interfere with eachother and operate on different layers. The same result can be seen here. With partial the effect does not seem to have an impact. An important note about the validity of the results of partial loading in this figure is made in section 6. The only reason the runtime does not decrease would be if it was dominated by the execution time. As seen in table 1 this is not the case.

The results of testing different amounts of memory can be seen in figure 2. As long as there is enough memory, which is at 1G and 512M in this case, the runtime does not differ. As soon as we reach a critical amount, the runtime increases rapidly. For bulk and DeepEye mode this can be explained by the loading of layers that had to be put in swap memory as the total amount did not have enough capacity to keep them in memory. After the initial loading of the layer from disk, it was written back to swap memory on disk, after which it was retrieved again when it was ready to be executed. These modes scale terribly when the amount and size of networks grow and the memory decreases to a demanding level.

Linear mode is not affected too much by this. Its approach to immediately use the layer that was loaded does not have the issue of loading it twice from swap memory. It can be seen though that at 256M of memory the runtime is also vastly increased. This is likely due to the execution of the layers not having enough memory to properly work without using the swap file. This also affects the other modes of operation. Again, the results for partial loading are dubious because of the aforementioned reasons. In figure 6 the correct results are displayed and here it can be seen partial loading reacts in a similar fashion, as both the issue of loading early and not having enough memory for the execution take place.

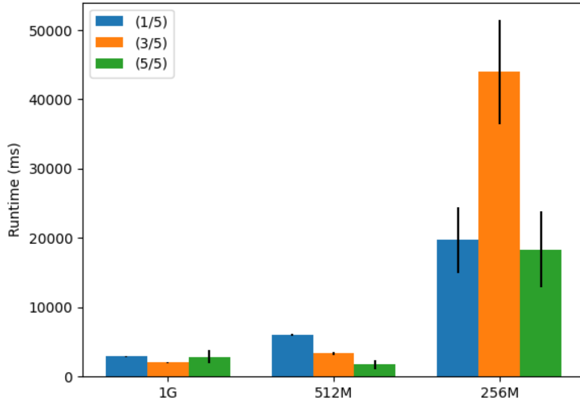

Figure 3: Five networks are run in bulk mode with varying amounts of memory. Different amount of networks are considered similar.

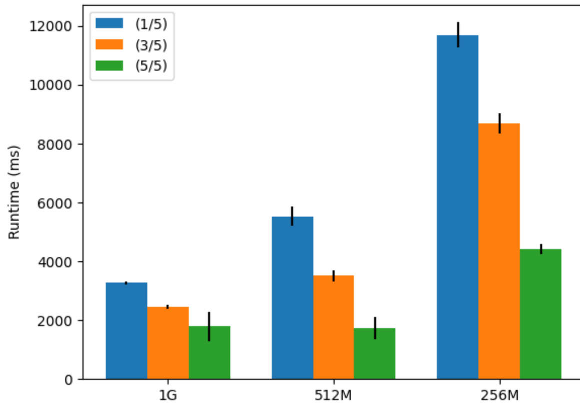

Figure 4: Linear mode is used when running the five networks. Caching is applied in the cases where applicable.

The results for bulk mode have ended surprisingly. In figure 3 we can see the results of using 1G of memory where having 5 similar networks does not yield better results than having only 3. The expected results can be seen when using 512M of memory, where a gradual decrease in runtime is presented by having more similar layers. When using only 256M of memory the results seem unstable as well. Reasons for this could be the operating system working on other processes, or when the swapping of the wrong data when exceeding the memory limit.

With linear mode in figure 4 the outcomes behave intuitively. The more memory available, the faster the inferences run up to a certain threshold. As more cached layers can be reused, the loading time decreases and so does the overall execution time as it is single threaded. Linear is a very robust mode that suffers the least from having too little memory. The most it needs to have in memory at a single time is one layer

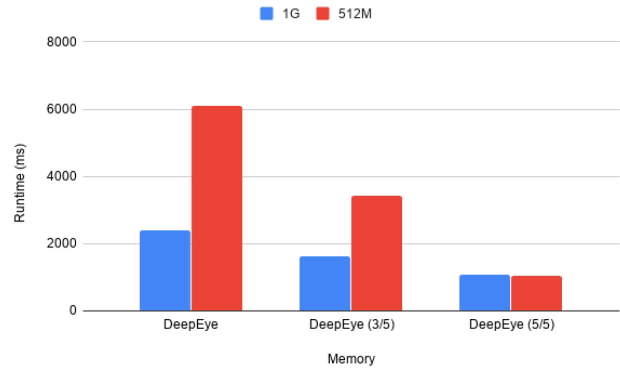

Figure 5: The different runtimes are plotted against the amount of cached layers. Five networks are tested.

of the network, whereas other modes often have more than this. The slowdown on execution time due to the memory constraint is similar to that of other modes.

figure caption more extensive, figure 5 (also applies to others)

In figure 5, the axis and labels are swapped, which makes it clear how changing the memory compares to each other and affects the runtime. The proportions of running with 1G or 512M of memory are getting more equal as more of the networks become similar. When all networks are similar the advantage of running with extra memory appears to be gone. This indicates an important decision to be made when designing the device for the application. A trade-off can be made, with on one side possibly decreasing the accuracy by using networks trained with transfer learning and on the other side investing more in better hardware.

The change in memory is especially visible with the DeepEye mode. As this mode loads the fully connected layers on a different thread from the start, the memory quickly becomes full and the swap file will be used. The reusing allows for the memory difference to not be noticeable.

Figure 6 depicts the runs performed with partial loading. The same trend can be seen where more caching results in a lower runtime. When running the inferences with partial loading, 512M of memory and not being able to reuse any layers, the runtime has a small spike. This is the brim of where the runtime will start growing rapidly if the memory is slightly decreased or the workload is increased. This is also indicated by the error margin being larger. With caching 3 out of 5 networks the problem is solved and a low steady runtime is achieved again. Partial loading works the fastest of any mode when there is enough memory available, but does not scale as well as linear when this is not the case.

The differences in load and execution time spent on the layers is shown in figure 7. The test is run on partial mode. As expected the execution times of the layers remains roughly the same, with the only exception being 512M with no reusable layers, which was described earlier as a shortage of memory for the execution part of the layer. The time on loading the layers is much higher still than that of execution. For the non cached runs this seems appropriate, but for

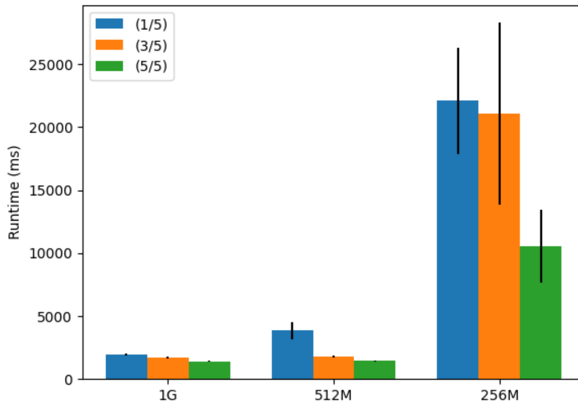

Figure 6: The results of running five networks with partial loading. Varying amounts of networks share similarities.

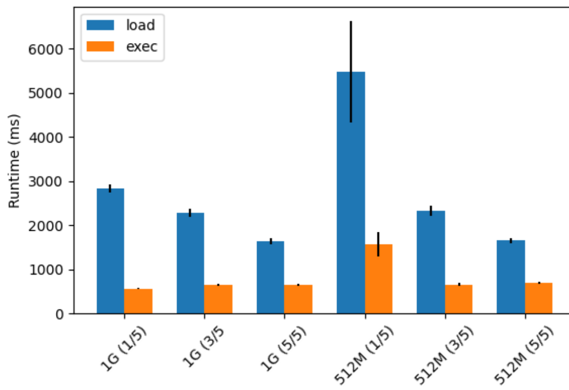

Figure 7: The time for loading tasks and executing tasks for five networks. The time taken is from start to finish, two threads are used.

the ones where caching takes place the differences are quite large. With these tests the total time taken on each layer is taken. This means that with two threads, if they are both trying to load a layer, the time taken for accessing the disk of one thread is also taken in the loading time of the other. The time a thread is waiting on another thread diminishes the efficiency of using a multithreaded approach.

6 Responsible Research

6.1 Reproducibility

The code that was used to gather these results is from Edge-Caffe, and will become open source in the fall of 2020 when the paper from the author is released. The caching discussed in this paper is an extension of this. Whether this is supplied as well is unsure, but the framework provides easily modifiable code. The methods used to implement it were described in section 4. The specifications of the hardware used is given, but since this is not any common standardised hardware results may be difficult to get this exact hardware and it may vary on different devices.

The results presented in section 5 are from different versions of the code. Although the results are similar to that of other versions, changes in the code might have changed the outcome slightly. This is due to code added later on that made more statistics about the inferences available, which also included other updates such as not needing to initialise every network from the start but being able to do so during an execution. The relative differences visible in the images are however still valid, and prove the point even if the results are not exact. In figure 1 and figure 2 the partial loading results are most likely from a faulty version, but due to time constraint these remain in the image. In the other parts of the analysis, the partial code results are correct and are up to date.

Networks stored in hdf5 file format have not been tested as the underlying code in loading these is less adjustable. This is the reason why SoS AlexNet is not used. The four other networks that are compared seem diverse enough that this should not be a huge problem, but analysing more different cases is always better. That this network is stored in hdf5 format should not matter for the caching.

6.2 Ethics

In itself, the analysis of caching on edge devices is not related to ethics. Any improvements on being able to run neural networks on these resource constrained devices will stop this data from being sent over the internet, and will therefore remove this privacy concern. These improvements do not lead to any new cases that could not have been done before. What can be done with running deep neural networks on edge devices in general seems outside of the scope of this research.

7 Discussion

Many different looks on caching have been researched in the past. Each with their own set of constraints and strengths. Many of these techniques can be combined to fit a certain use case. If for example we use networks that are trained with transfer learning, and multiple inferences need to be run on

one single image, all previous mentioned techniques can apply. We can apply the splitting of the intermediate result from Mainstream when the networks branch off, and combine this with the caching of small blocks of the image that DeepMon introduced.

When the images do not arrive at the same time or the images are different, the caching of these similar layers will decrease the loading time and can be used in combination with the DeepMon approach, which lowers the execution time. With partial loading making the most of an even spacing between loading and execution time, the use of both systems can be balanced given a limited amount of memory to optimally make use of the multithreaded approach. What the optimal balance is also depends on the networks run, as the ratio between the load and exection time varies for each.

8 Conclusion and Future Work

As seen, caching can be used to decrease the runtime in certain cases, but also provides methods for decreasing the hardware requirements of running inferences. For inferring in different modes, the caching techniques can vastly increase performance or decrease the amount of memory needed while still obtaining the same runtime.

For DeepEye and bulk, the caching has less of an effect. These modes require much space for keeping pre-loaded layers in memory. Linear shows steady results with caching having a solid benefit. Partial loading is the fastest of the modes and also has much to gain with caching.

The capabilities of resource constrained devices increase due to technological advancement and improvements on how we run deep neural networks. This paired with transfer learning makes running inferences much more accessible, as the networks can be both trained and used afterwards more easily.

References

[1] Yaniv Bar, Idit Diamant, Lior Wolf, and Hayit Greenspan. *Chest Pathology Detection Using Deep Learning with Non-Medical Training*.

[2] Kasthurirangan Gopalakrishnan, Siddhartha K. Khaitan, Alok Choudhary, and Ankit Agrawal. Deep Convolutional Neural Networks with transfer learning for computer vision-based data-driven pavement distress detection. *Construction and Building Materials*, 157:322–330, December 2017.

[3] Shin Hoo-Chang, Holger R. Roth, Mingchen Gao, Le Lu, Ziyue Xu, Isabella Nogues, Jianhua Yao, Daniel Mollura, and Ronald M. Summers. Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning. *IEEE transactions on medical imaging*, 35(5):1285–1298, May 2016.

[4] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95, Niagara Falls New York USA, June 2017. ACM.

[5] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[6] Angela H Jiang, Daniel L-K Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A Kozuch, Padmanabhan Pillai, David G Andersen, and Gregory R Ganger. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 29–42, 2018.

[7] Akhil Mathur, Nicholas D. Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. DeepEye: Resource Efficient Local Execution of Multiple Deep Vision Models using Wearable Commodity Hardware. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 68–81, Niagara Falls, New York, USA, June 2017. Association for Computing Machinery.

[8] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. CNN Features Off-the-Shelf: An Astounding Baseline for Recognition. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 512–519, Columbus, OH, USA, June 2014. IEEE.

[9] Nima Tajbakhsh, Jae Y. Shin, Suryakanth R. Gurudu, R. Todd Hurst, Christopher B. Kendall, Michael B. Gotway, and Jianming Liang. Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning? *IEEE Transactions on Medical Imaging*, 35(5):1299–1312, May 2016. arXiv: 1706.00712.

[10] Mengwei Xu, Xuanzhe Liu, Yunxin Liu, and Felix Xiaozhu Lin. Accelerating Convolutional Neural Networks for Continuous Mobile Vision via Cache Reuse. page 13.

[11] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. DeepCache: Principled Cache for Mobile Deep Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, pages 129–144, New Delhi, India, October 2018. Association for Computing Machinery.