Analysis of Loading Policies for Multi-model Inference on the Edge

Yohan Runhaar[1] , **Bart Cox**[1] , **Amirmasoud Ghiassi**[1]
[1]TU Delft
{y.f.runhaar, b.a.cox}@student.tudelft.nl, {s.ghiassi}@tudelft.nl

Abstract

The increasingly growing expansion of the Internet of Things (IoT) along with the convergence of multiple technologies such as the arrival of next generation wireless broadband in 5G, is creating a paradigm shift from cloud computing towards edge computing. Performing tasks normally done by the cloud directly on edge devices would ensure multiple benefits such as latency gains and a more robust privacy of data. However, edge devices are resource-constrained and often do not possess the computational and memory capabilities to perform demanding tasks. Complex algorithms such as the training and inference of a complete Deep Neural Network (DNN) is often not feasible on these devices.

In this paper we perform a novel empirical study of the various ways that multiple inference tasks of deep learning models can be loaded on these edge devices. We analyse the run time gain, under different resource limits, of various DNN layer loading policies that aim to optimize the overall run time of consecutive inference tasks. We combine this with further research in the memory usage and swapping behaviour when performing these inference tasks. Using these results, we show that if the memory overhead becomes too large, loading and executing DNN layers in an interleaved manner provides significant gains in run time. This is achieved trough multiple experiments in our specially made evaluation environment EdgeCaffe which is presented in this paper as well.

Keywords
Deep learning, Inference, Edge devices, Edge computing, Memory constrained

1 INTRODUCTION

The variety and amount of computing devices that are interrelated with each other and make up the system known as the Internet of Things (IoT), are expanding at an astonishing rate. Indeed, the convergence of multiple technologies as well as the ever-growing interest and demand in “Smart products” has sparked a revolution in the production of such devices, considerably expanding the range of the Internet of Things. A report published by Fortune Business Insights [1] indicates that the global IoT market valued at US$ 190.0 Bn in 2018, is anticipated to reach US$ 1,102.6 Bn by 2026, showcasing the burgeoning future of IoT. This rapid innovation is causing a paradigm shift from cloud computing, where the computation is done by a centralized server, towards the newer computing paradigm known as edge computing, where the computation happens at the ‘edge’ of the IoT network. This is due to the fact that in cloud computing the devices situated at the edge of the network, also known as edge devices, gather a large amount of data from their various sensors but do not do anything with it. This data would be sent to the cloud for further processing. Edge computing would remove the network latency between the devices and the cloud as well as the increasing bandwidth and load on the centralized servers, allowing for real-time computation and a decrease in cost pressures of IoT. Furthermore, edge computing limits the amount of data sent to the cloud for processing, therefore ensuring a more robust privacy of input data.

This paradigm shift brings a new prospect of performing artificial intelligence (AI) at the edge. Indeed, AI applications such as machine learning can benefit immensely from the shorter latency as well as the privacy advantages. However, one of the challenges that is faced by the industry is the limited resources offered by these edge devices. The task of performing computational and memory heavy algorithms on these devices is one that has yet to be solved. This makes the training and inference jobs of complete Deep Neural Networks (DNN) on these devices not feasible most of the time. According to a chief scientist at Baidu’s Silicon Valley Lab [2], training one of Baidu’s Chinese speech recognition models requires four terabytes of training data as well as 20 billion billion math operations across the entire training cycle. For this reason, training a DNN on a device such as a smartphone is virtually impossible to run. The focus of this paper will thus be on inference.

Stahl, Zhao, Mueller-Gritschneder, Gerstlauer, and Schlichtmann [11] propose a solution to this issue by describing a memory and communication aware approach that enables edge devices to partition and fuse consecutive fully connected and convolutional deep neural network layers. The combination of the partitioning and fusing methods would allow full deep neural network applications to be executed on resource-constrained edge clusters in a fully distributed manner. While this technique allows for the optimization of the loading and processing of deep inference jobs on a cluster of edge devices, it is not designed to be used on a single device. In contrast to this, the DeepEye paper [9], describes a method applicable to a single edge device. The authors propose a novel approach to the issue by presenting a "match-box sized wearable camera capable of running multiple could-scale deep learning models locally on the device". The paper describes the design of their inference pipeline incorporated in their machine. This execution pipeline includes a number of new optimization techniques, including their take on the loading of the DNN layers in such a resource constrained environment. This technique – known as "Runtime Interleaving", is built on the idea of interleaving the loading of fully connected layers with the execution of convolution layers. In combination with other techniques, the Runtime Interleaving technique makes the proposed inference pipeline one of the many required innovations to support this paradigm shift. Although the authors of the DeepEye paper argue that the optimization techniques described in their paper will operate successfully for any mobile platform that target uses of deep learning, it is designed to be used for image-based models.

The aim of this paper is to find the most optimal way that multiple concurrent deep learning inference jobs can be scheduled in a resource constrained environment. In order to achieve this, we outline and compare different given loading and executing methods for DNN layers meant to optimize the overall run time of multiple inference jobs. We determine how effective each loading policy is by conducting run time measurements in *EdgeCaffe*, our specially made evaluation environment built around Caffe [5]. These evaluations are undertaken with regard to both Convolutional Neural Networks (CNN) layers as well as fully connected layers by using different deep learning models, with different types and amounts of layers: AgeNet and GenderNet [7], AlexNet [6], FaceNet [3] and GoogleNet [12]. We combine the gathered results with further research in other potential factors of the overall inference time such as the impact of memory swapping. Using these evaluations, we will showcase why loading and executing a layers in a linear manner is the most desirable loading technique to use in order to optimize the inference time in a resource constrained environment. Using this technique allows for run time gains up to 1.63x compared to the normal bulk loading.

2 BACKGROUND

In this section we discuss the methodology undertaken to answer the research question and the motivation behind the choices we made. First, our specially made evaluation environment built around Caffe [5] is described. This is followed by a discussion of the motivation behind the project as well as the possibilities it offers. And finally, the experimental setup is described.

2.1 EdgeCaffe

Our evaluation environment known as *EdgeCaffe* aims to run the Caffe [5] Deep Learning framework on edge devices. Caffe is a deep learning framework which takes into account expression, modularity and speed. It is tailored towards the classification and segmentation of images all the while supporting both convolutional and fully-connected neural network designs. The extensibility of the Caffe framework made it the perfect project to built our evaluation environment around. The EdgeCaffe code is targeted for Ubuntu 18.04 for both x86-64 and Raspberry Pi machines (ARMv8-A). C++ is the main language used throughout the project while a few Python bindings are made.

Tasks are the core of the EdgeCaffe project. According to the chosen schedule policy, the availability of workers, the arrival distribution as well as other factors, the core system will execute a set of tasks in a specific order. In essence, what EdgeCaffe does is that it monitors and organizes the tasks to be executed. In order to achieve this, EdgeCaffe stores the tasks in three different types of task pools: the *pool of waiting tasks*, the *pool of ready tasks* and the *pool of finished tasks*. Figure 1 helps visualizing the organization of these different task pools which can be described as follows:

- *Pool of waiting tasks*: These tasks depend and thus await the execution of preceding tasks before they can be executed. The *Orchestrator* is in charge of verifying if a task is ready to be executed and can be moved to the ready pool.
- *Pool of ready tasks*: These tasks are awaiting to be selected by a worker to be executed. This is done in a First Come First Serve manner.
- *Pool of finished tasks*: This pool contains all the tasks that have been performed. Once all the tasks are present in this pool, the inference job is finished and the DNN can be cleaned up.

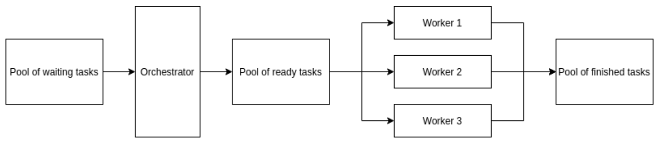


Figure 1: EdgeCaffe task pools

The EdgeCaffe project distinguishes three types of tasks:

- *LoadingTask*: This task loads the current layer parameters from disk to memory.
- *ExecutionTask*: This task utilizes the parameters that a loading task stored in memory to calculate the values for the next layer.
- *DummyTask*: This task task is used as a mock task in order to simulate networks. When a network is generated randomly, the dummy task will estimate the execution time of a task by keeping the worker busy. The execution time only is stalled, it does not actually consume any memory. This type of task is not used for normal networks, other already trained deep models such as

AgeNet [7] or FaceNet [3] only use loading tasks and execution tasks.

Built around Caffe [5], EdgeCaffe supports any deep neural model geared towards image classification or segmentation. In order for a model to be processed by the EdgeCaffe core it needs to be split beforehand by the *ModelSplitter*. This specially made tool splits a .caffemodel file in smaller model files. The main idea behind the EdgeCaffe project is the *partial execution* of layers of a deep neural network. By splitting a deep neural model into smaller parts, swapping can be prevented from happening which will improve the inference time in resource constrained environment. The idea of partial execution is that each split layer of the model has to be loaded into memory and executed. But by scheduling this process, a dependency graph can be created of the tasks that need to be executed before the current task can be ran.

Once a model is split, it can be submitted individually or as a set of multiple models to EdgeCaffe. The set of networks will then be processed according to the chosen policy. Currently, EdgeCaffe offers four scheduling policies to choose from: *Bulk*, *DeepEye*, *Linear* and *Partial*. A more detailed description and analysis of the methods that will be compared can be found in section 3. After the set of networks is executed, the used memory is deallocated and the end-to-end times are outputed to a csv file.

2.2 Experimental Setup

A series of experiments is conducted around the EdgeCaffe environment. Using five different image-based deep models, a series of pipelines is generated, each varying in length and computational complexity. These pipelines are created by randomly combining these five deep models. In order to evaluate the benefits of the different scheduling policies, the numerous generated pipelines are executed according to each of these policies. The amount of pipelines to be generated has to be specified to the program. After running all the generated pipeline, the program formats the measured data into an csv output file.

These evaluations are undertaken with regard to both Convolutional Neural Networks (CNN) layers as well as fully connected layers by using different deep learning models, with different types and amounts of layers. These models are all open source and available to download as a pre-trained caffemodel[1]. Here is a description of each used model, for more details see Table 1 which contains more detailed specifications of each caffemodel:

- *AgeNet* and *GenderNet* [7]: Convolutional neural network model for age and gender prediction based on an input image of a person.
- *AlexNet* [6]: Winner of the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) that popularized CNNs, it is a deep convolutional neural network able to classify high resolution images.
- *FaceNet* [3]: Convolutional neural network model for multi-view face detection.
- *GoogleNet* [12]: Winner of the 2014 ILSVRC competition, it is a convolutional neural network able to classify high resolution images with significant improvements over *AlexNet* [6] and other previously proposed CNNs.

Caffemodel	**Size (in MB)**	**# of conv. layers**	**# of fc layers**
AgeNet	45.7	12	8
AlexNet	227.6	16	6
FaceNet	227.5	16	8
GenderNet	45.6	12	8
GoogleNet	23.9	10	142

Table 1: Specifications of the caffemodels used during evaluation

All experiments are run on a virtual machine using VirtualBox [10] developed by Oracle. The guest virtual machine runs a 64-bit version of Ubuntu 18.04.4 as operating system. It has a single core Intel processor (Intel Core i7-7700HQ CPU @ 2.80 GHz), 40GB of disk storage, 4096MB of usable RAM and 16MB of video memory. In order to simulate the resource limitations met on an edge device, the experiments are performed under different memory limitations. This is achieved by using control groups (cgroups), a feature provided by the Linux kernel allowing us to manage and restrict groups of processes [8]. Using cgroups, the memory usage is limited and the experiments are ran under the following byte limits: 1GB, 512MB and 256MB. The maximum amount for the sum of memory and swap usage is limited to 8GB.

3 DEEP NEURAL NETWORK LAYERS LOADING POLICIES

In this section we discuss the different loading policies for multiple model deep inference jobs that we will be comparing in our empirical research. Each loading technique is described in as much details as possible in order to provide a high-level understanding of them. These descriptions are then discussed in order to hypothesize how each technique is to perform during the evaluation. In order to better showcase these different loading policies, dependency graphs of the AgeNet [7] model are generated for each of the policies. These graphs can be found in appendix B.

3.1 Bulk

The *Bulk* loading policy can also be described as the "Normal" mode. When selecting this policy, all the EdgeCaffe loading tasks inferred from the first network in the set of submitted networks are moved to the *pool of ready tasks* first by the orchestrator. Once all the layers are fully loaded, meaning that all the *loading tasks* that were in the *pool of ready tasks* are now in the *pool of finished tasks*, the orchestrator starts to move the *execution tasks* from the *pool of waiting tasks* to the *pool of ready tasks*. All the execution tasks are then performed in order until the network is finished and its memory deallocated. This process is then repeated for every network in the set of networks. Figure 5 is the dependency graph of the tasks that need to be executed when the AgeNet [7] model is submitted using the *Bulk* policy.

[1]Caffe Model Zoo: https://caffe.berkeleyvision.org/model_zoo.html

When performing inference tasks on resource-constrained devices, the computational overhead is dominated by the executing of convolutional operation while the memory overhead is dominated by the loading of fully connected layer parameters. The *Bulk* loading policy does not optimize this in any way as it loads all the layer parameters before executing them. This policy is expected to perform the worst between all the loading policies, and the difference in inference time to be increasingly bigger as the complexity of the inference pipeline increases as well. This policy will be used as baseline in the evaluations.

The *Bulk* loading policy loads all the layers upfront before executing them.

3.2 DeepEye

The *DeepEye* loading policy is based on the ”Runtime Interleaving” technique proposed in the DeepEye paper [9]. The authors of DeepEye base this technique on the idea of interleaving the loading of fully connected layers with the execution of convolution layers. They separate the layers of all the models into a convolutional layers pool and a fully connected layers pool. They then spawn two threads, a *convolution-execution* thread and a *data-loading* thread. The *convolution-execution* thread loads all the convolution filter parameters of all the layers in the pool into the memory and starts performing convolution operations on the pre-processed input data for each model. It does so in a pipelined manner, meaning one model after the other. In parallel, the *data-loading* thread loads all the fully connected layer parameters for each model into the memory in the same manner. The goal of the *convolution-execution* thread is to finish all the convolution operations and pass the result of the final convolution layers of each model to the *data-loading* thread. The objective of the *data-loading* thread is to finish loading all the fully connected layer parameters for a model and to then proceed to get the final output by using the pre-computed convolution outputs provided by the other thread.

In order to reproduce this concept on EdgeCaffe, two task pools are spawned, one for the convolutional layers, and one for the fully connected layers. The orchestrator organizes the tasks between two workers such that one of the workers, known as the convolution execution worker, is to perform all convolutions on the input image, and and pass the results of the final convolution layer of each model to the second worker, known as the data-loading worker. When the data-loading worker finishes loading the fully connected layer parameters for a model, it can use the precomputed convolution outputs from the convolution execution tasks and proceed to obtain the final classification results. This is depicted in Figure 6, showcasing the dependency graph of the tasks that need to be executed when the AgeNet [7] model is submitted using the *DeepEye* policy.

Since the loading of fully connected layers is interleaved with the execution of convolutional layers, a significant gain in inference time can be expected by using this loading policy. Indeed, the interleaving gain is equal to the maximum execution time between the two workers. If the time to run the convolutional operations is equal to the time taken to load the fully connected layers, a maximized interleaving gain of 2 can be achieved. But as the memory resource get more and more scarce, the loading of fully connected layers will start to dominate the inference time and this technique might not perform as well.

The *DeepEye* loading policy interleaves the loading of fully connected layers with the execution of convolutional layers.

3.3 Linear

The *Linear* loading policy, as the name states, executes layers in a straight sequential manner. When selecting this policy, all the EdgeCaffe tasks are performed in order, one at a time. The *loading task* of the first layer of the first network is performed first, followed by the *execution task* of the same layer. This process is repeated in order for every layer of the submitted deep learning models. Figure 7 depicts the dependency graph of the tasks that need to be executed when the AgeNet [7] model is submitted using the *Linear* policy.

Since the loading and the execution of layers is always alternated, this loading method is expected to perform better than the *Bulk* policy as it won't reach the memory overhead dominated by loading all the layer parameters in a sequence. Instead, the memory usage is limited due to the fact that the layer is executed immediately after it has been loaded. In a more and more memory constrained environment, the *Linear* loading policy might then perform significantly better in comparison to the other policies.

The *Linear* loading policy loads and executes a layer at a time.

3.4 Partial

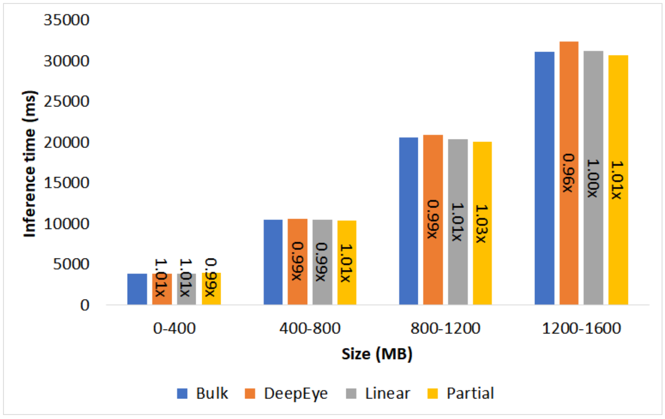
The *Partial* loading policy is a multi-threaded version of the *Linear* loading policy. The orchestrator loads and executes DNN layers in the same order for both policies. The difference with the *Linear* loading policy is that the *loading tasks* can be split over multiple workers, while the *execution tasks* are executed by a single worker. Thus, the loading of the parameters of each layer can be distributed and accelerated. Figure 8 shows the dependency graph of the tasks that need to be executed when the AgeNet [7] model is submitted using the Partial policy.

Since the *loading tasks* are distributed over multiple workers, the *Partial* loading technique will in theory perform better than the *Linear* loading policy. But this might not be the case if the memory resources are too scarce to hold all the parameters of the DNN layers. In that case the *Linear* policy might work better as the loading and executing of layers is always interleaved.

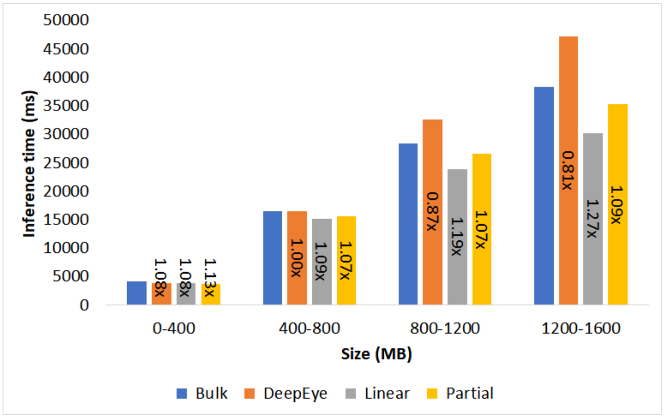
The *Partial* loading policy loads all the layers greedily over multiple workers, while a single worker executes the loaded layers.

4 ANALYSIS

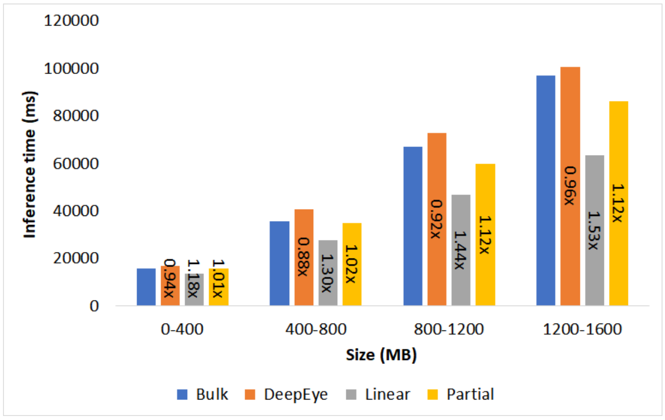
In this section, we present the multiple evaluations we performed in order to compare the different loading policies and



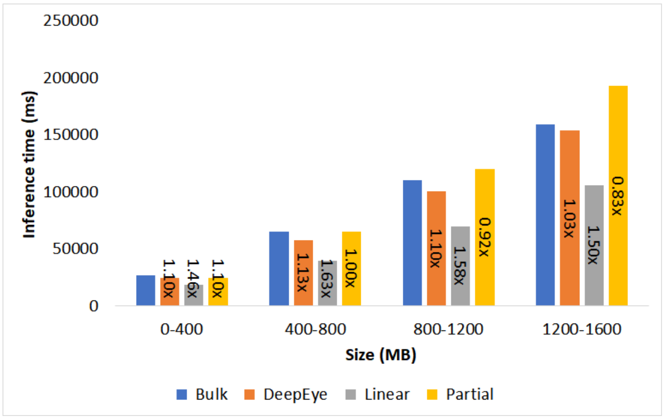
(a) No additional memory limit



(b) Memory limit of 1GB



(c) Memory limit of 512MB



(d) Memory limit of 256MB

Figure 2: Average inference time in ms per size of the pipelines for each loading technique based on the measurements of 100 differing runs of the EdgeCaffe pipeline under different memory limitations

determine how effective each of these loading policies are under different environments.

4.1 Impact on Inference Time

Here we evaluate the performance of the different loading policies without setting any additional memory limitations as well as under different memory limitations.

Setup. In total, 100 different pipelines are generated with various length and complexities. Every pipeline is to be executed and measured for each of the four loading policies. We perform this evaluation under different memory limitations to see if this has an impact on the performance of the different loading techniques. We thus perform the same script under the following byte limits: 1GB, 512MB, and 256MB.

Results. Figure 2a illustrates the performance of the different scheduling techniques ordered by the size of each pipeline under no additional memory limit. This means that the full 4096MB of memory is available to use. The results are obtained by averaging the results that correspond to pipelines within the same size range of 400MB. The run time gain is displayed for each loading policy in comparison to the *Bulk* policy which is used as baseline for the evaluation. Figure 2a shows that as the run time memory available on the device is not additionally limited, that all the four policies perform approximately the same. Indeed, the pipelines achieved similar inference times for each of the policies for the ranges of size with run times gains not exceeding the 1.01x mark.

Figure 2b shows the results under a memory limit of 1GB. In this figure, the results start to differ from the previously observed ones. The four policies still perform similarly for smaller pipelines that have a size varying from 0 to 800 MB. As the size of the pipeline increases, some policies start to outperform others. An interesting observation here is the fact that the difference in run time between the policies increases with the length of the pipeline. A bigger gain in inference time can be observed with the *Linear* loading policy over the other policies with a maximum run time gain of 1.27x. The *Partial* policy achieves a maximum run time gain of 1.09x. Interestingly, the *DeepEye* policy performs worse than the baseline with a maximum run time loss of 0.81x. In comparison to the previous figure, where no additional memory limits were set, the overall inference time increases considerably for all the policies. The baseline *Bulk* policy observes a maximum inference time of approximately 53 seconds in comparison to the observed 35 seconds inference time in Figure 2a which is 1.5x the inference time.

Figure 2c shows the results under a memory limit of 512MB. A bigger difference in inference time can be observed here between the policies. Similarly to the previous memory limitations, the four policies achieve very close inference times for the smaller pipelines. But in an identical manner to the Figure 2b, as the size of the pipeline increases, the difference in run time gains between the policies also increases. The performance hierarchy between the loading policies also remains the same. The *Linear* loading policy does still achieve the best inference times with a maximum run time gain of 1.53x. The *Partial* policy achieves a maximum run time gain of 1.12x which is significantly lower than the run time gains achieved by the *Linear* policy. As discussed in section 3, this is more than likely due to a memory overhead created by the greedy execution of consecutive *loading tasks* by the *Partial* policy in contrast to the loading and executing interleaving by the *Linear* policy. The *DeepEye* policy performs very similarly to the *Bulk* policy. Again here, the overall inference time increases compared to the previous memory limitations. In this instance, the *Bulk* policy achieves a maximum inference time of approximately 129 seconds.

Figure 2d shows the results under a memory limit of 256MB. In such a memory constrained environment, compelling changes appear in the performance hierarchy of these different loading policies. As the size of the pipelines increases, the *Partial* loading policy starts to achieves the worst inference times in this environment with a maximum run time loss of 0.83x. This could be due to the fact that as the memory overhead caused by the greedy loading of layers increases due to the pipeline being larger, the *Partial* policy experiences significantly higher memory swapping which would slow down the whole process. The *Linear* policy remains the finest with a maximum run time gain of 1.63x. This further pushes the idea that as the memory overhead becomes larger, the Linear policy becomes more and more advantageous to use in comparison to the other techniques. The overall inference time increases drastically under this environment. The *Bulk* policy achieves a maximum inference time around 191 seconds.

As such, the findings above can be summarized as follows. We have found that in an environment with sufficient memory - above 1GB - the *DeepEye*, the *Linear* and the *Partial* loading policies provide similar run time gains in comparison to the *Bulk* policy which is used as the baseline of the experiment. But as the environment becomes more and more scarce and the size of the pipeline larger, meaning that the memory overhead is increasing, the *Linear* policy starts to outperform all other loading policies. The overall inference time also increases drastically as the memory limit becomes more and more constrained.

4.2 Impact on Memory Usage

In this section we investigate the impact on the memory usage and swapping behaviour the different loading policies have when performing inference tasks under different memory limitations.

Setup. We combined the previous evaluation with the calculation of important memory information such as the maximum and average usage of both physical and swap memory throughout the execution of each pipeline. We will be analyzing the measured memory information for the 1GB and 512MB memory constrained environments.

Results. The most interesting results from this experiment could be found by analyzing the average RAM usage and the average swap space usage in these two environments.

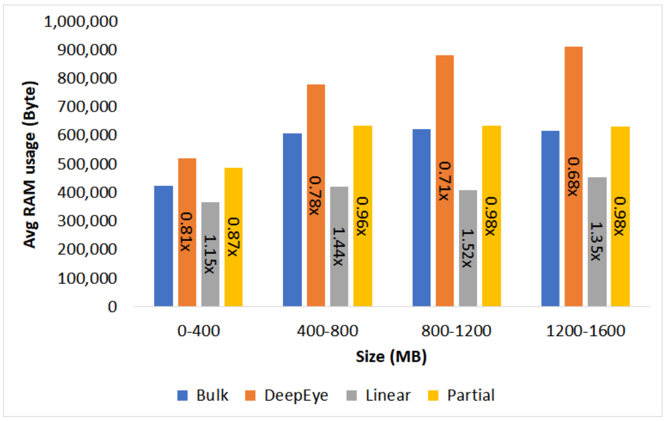
Figure 3a shows the average RAM usage under a memory limit of 1GB. Indeed the RAM usage here does not exceed the 1GB mark for any of the four policies. As expected, the memory usage increases as the size of the pipelines increases. Another interesting observation is the fact that the usage differs significantly between each loading policy. The *DeepEye* and the *Linear* policy both use an approximately similar amount of memory, the most between the four policies. The *Linear* policy is the most resource friendly policy under this environment, with a memory usage not exceeding 600MB, 1.68x better than the baseline. Figure 3c shows the average swap space usage under the same memory limit of 1GB. The memory consumption here is approximately similar for all the policies despite the size of the pipeline. The four policies have a swap space usage varying between 900 and 1100 MB. The *Linear* policy does however uses slightly less swap space than the other policies.

Figure 3b shows the average RAM usage under a memory limit of 512MB. In Figure 3a, smaller sized pipelines would not require the full 1GB of memory to be used and the memory usage would increase with the size of the pipelines. Here, the memory usage for each policy are vaguely similar to each other despite the size of the pipeline. The RAM usage does not exceed the 500MB mark for any of the four policies but the *DeepEye* policy borders it. The *Linear* policy uses the least memory under this memory limit as well, with a memory consumption fluctuating around 320MB. Figure 3d shows the average swap space usage under a memory limit of 512MB. The swap space usage is remarkably higher than under the 1GB memory limit. In Figure 3c the swap space usage wouldn't exceed the 1100 MB mark, while in Figure 3d all the policies do exceed it. Thus, as the RAM memory limit is exceeded, the swap space usage increases drastically. Indeed, the *DeepEye* policy which bordered the maximum available RAM, uses the most swap space with a consumption up to 1800MB for the largest pipelines.

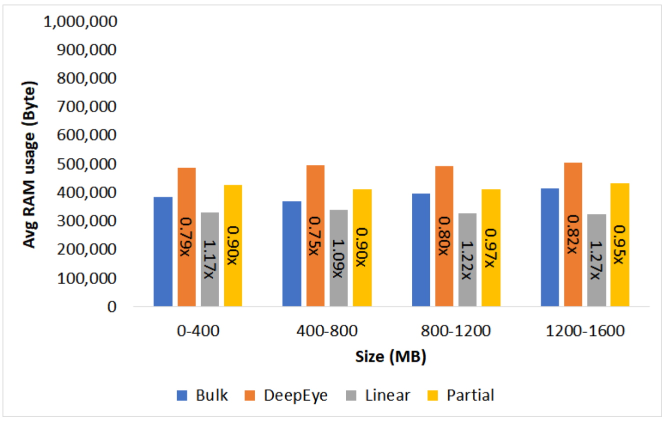
We have found that as the maximum available RAM usage is reached, the swap space usage increases significantly. Thus, as the available memory is more and more constrained, the pipeline is more likely to require a higher amount of swap space. The *DeepEye* loading policy uses the most memory while the *Linear* policy is the most resource friendly of the four policies.

4.3 Impact of Number of Loaded Layers

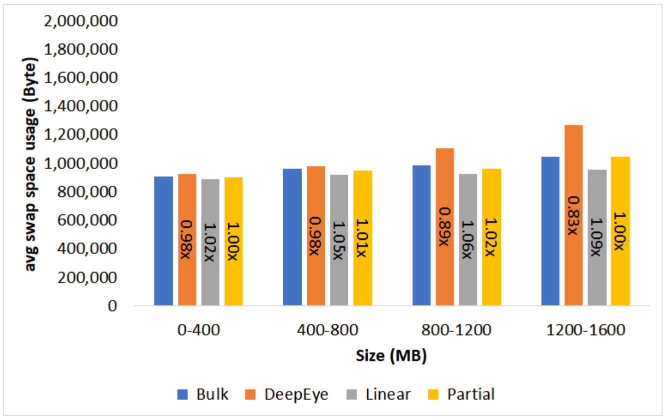
Here we evaluate the impact of loading multiple layers at the same time. We propose two alternate versions of the *Partial* loading policy. The first variant which we will name the *Double* loading policy, is a *Partial* policy with two workers. The second variant, namely the *Triple* policy, is a *Partial* policy with three workers.



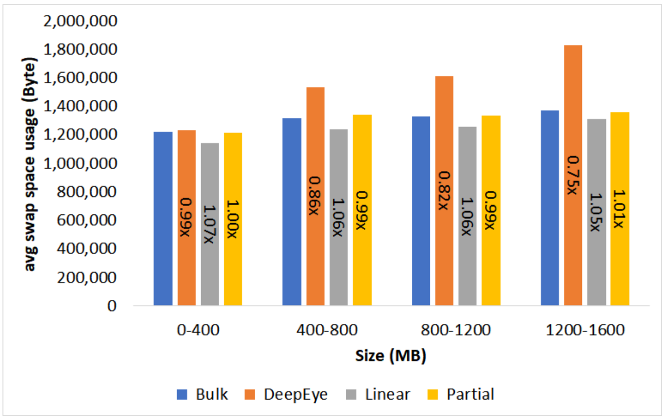
(a) Average RAM usage under the memory limit of 1GB



(b) Average RAM usage under the memory limit of 512MB



(c) Average swap space usage under the memory limit of 1GB



(d) Average swap space usage under the memory limit of 512MB

Figure 3: Average RAM usage and average swap space usage per size of the pipelines for each loading technique based on the measurements of 100 differing runs of the EdgeCaffe pipeline under different memory limitations

Setup. In total, 100 different pipelines are generated with various length and complexities. Every pipeline is to be executed and measured for the *Linear* policy as well as the two variants of the *Partial* policy. The evaluation is undertaken under a memory limit of 512MB.

Results. Figure 4a shows the average inference time of the three loading policies. A clear observation can be made from this figure. As the number of layer that are loaded at the same is bigger, the inference time becomes lengthier. This coincides with the observations made in Figure 2c and Figure 2d, where it is noticeable that the *Partial* loading policy is affected by the memory overhead caused by the greedy loading of layers. Here, the more layers can be greedily loaded, the more the inference time is affected. Figure 4b shows the average RAM usage of the loading policies. The results from this figure confirms that the memory overhead increases significantly, as the number of greedily loaded layers grows.

We have found that as the number of available workers for the *Partial* policy increases, the increased number of greedily loaded layers cause a larger memory overhead, ultimately slowing down the inference tasks significantly.

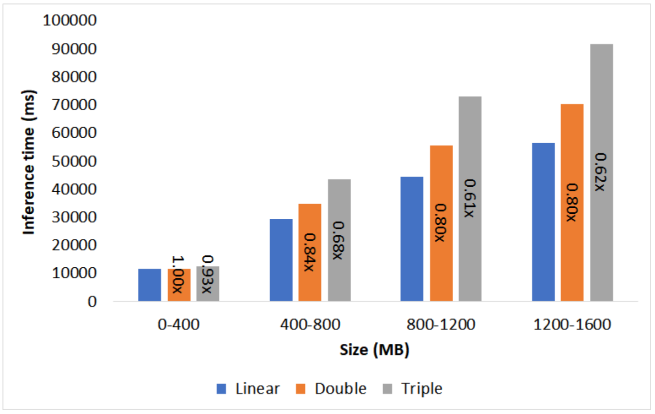
5 Discussion

In this section, we discuss the results of our empirical research. We provide a reflection on the achieved results for each scheduling policy as well as how they can be explained.

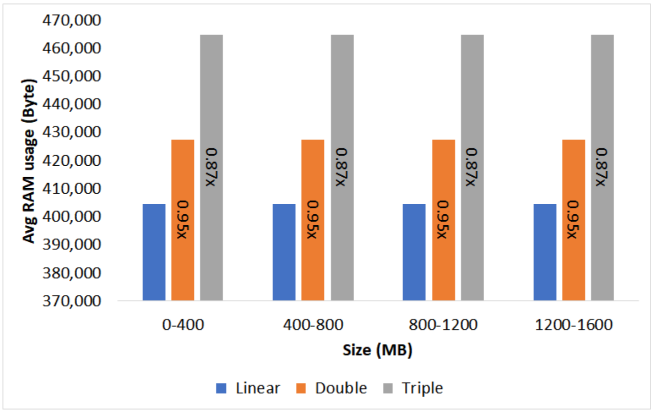
Our first experiment has shown that as the available memory becomes more and more constrained, the inference time increase drastically. Analyzing the memory consumption during the running of these pipelines unveiled that this also causes the average swap space usage to rise significantly. Furthermore, when the memory overhead caused by the loading of layer parameters grows, the *Linear* policy outperforms the other policies with an increasing difference in inference time. Figure 3c and Figure 3d have shown us that as the memory overhead becomes larger, that the usage of swap memory increases.

Swap memory is used to provide processes with additional memory space when all the main memory (RAM) is used up. But accessing the disk where the swap space is located, requires a lot more time than accessing RAM. The speed scale of RAM is in the area of 100ns while accessing data on a rotating disk requires 10ms, 100.000 times the RAM [4].

The *Linear* policy loads and executes a layer a time, limiting the memory usage. This is visible in Figure 3a and Fig-



(a) Average inference time



(b) Average RAM usage

Figure 4: Average inference time and RAM usage usage per size of the pipelines for each the *Linear* and alternate *Partial* policies based on the measurements of 100 differing runs of the EdgeCaffe pipeline under a 512MB memory limit

ure 3b, where the *Linear* policy uses significantly less RAM on average in comparison to the other policies. It is for this reason that the *Linear* policy performs remarkably better in resource constrained environments.

The *Partial* policy suffered deeply from greedily loading layers in resource constrained environments. The results observable in Figure 4a showcase the fact that the more layers are greedily loaded, the slower the inference time. Figure 4b shows how this performance loss is related to the memory overhead caused by loading layers in such a manner. Although the *Partial* policy does not provide significant run time gains in resource constrained environment, it can still perform very well if the available memory is large enough to handle the greedy loading.

Surprisingly, the *DeepEye* policy performed the worst between all four policies, even achieving inference time longer than the *Bulk* policy. This is due in parts because of the high RAM usage under this policy caused by the pre loading of fully connected layers during the loading and execution of convolutional layers. Another factor for this surprising performance is that in the DeepEye paper [9], a specially made hardware unit with a custom integrated carrier board was build to run their experiments on, which could have provided better results.

Knowing that the memory overhead caused by the loading of the DNN layers is the cause behind the biggest inference time loss, it would be interesting to propose a unified system that could determine the loading policy to use according to a number of factors such as : the length of the pipeline, the rate of fully-connected layers, the rate of convolutional layers and the available main memory.

6 RESPONSIBLE RESEARCH

In this section, we review the integrity of our conducted research. We conduct an epistemic review of our research in order to assess the degree of validation and reproducibility of our experiment and its results. For ethical concerns related to our research, view appendix A.

Reliable. In order to assess a confident degree of reliability of our research, we made sure the evaluations we performed produced consistent results. Indeed, we used a script to generate a large number of pipelines of various complexities and sizes by combining five different deep models in various ways. The results are then averaged over a range of pipelines, ensuring that the accuracy of the results.

Reproducible. In order to make sure our research and all the evaluations we performed are reproducible, we focused on transparency and automation. First of all, we only used data that was publicly available. Indeed, the deep learning models used for our research are all pre-trained caffemodels available to download on website such as Model Zoo [13]. Our evaluation environment, EdgeCaffe, is built around Caffe [5], an open-source deep learning framework. It will also be made publicly available in the near future. Secondly, in order to avoid any human error during our evaluation setup, we used multiple scripts to run the experiments. They will also be made available along with the EdgeCaffe project.

7 CONCLUSION

In this paper, we performed an empirical study on the different scheduling techniques for the loading and execution of deep neural network (DNN) layers. In a first place, we described our specially made evaluation environment EdgeCaffe that aims to run the Caffe Deep Learning framework on edge devices. We then showed that the loading and executing of DNN layers could be scheduled in several ways. We used the evaluation environment to conduct several experiments surrounding these scheduling techniques that we were investigating and determined the conditions under which each of these methods are the most desirable to use. In other words, the conditions for which they provided the biggest run time gain during the execution of multiple concurrent inference tasks. We highlighted the negative impact memory swapping had on the overall run time of multiple inference jobs when the memory overhead caused by the loading of DNN layer parameters became to large. We also showed that interleaving the loading and execution of DNN layers in a linear manner could provide significant gains in run time when coming across this issue.

References

[1] Global internet of things (iot) market expected to rise at 24.7% cagr, rising demand for iot solutions in bfsi. *Fortune Business Insights*, 2019.

[2] Michael Copeland. What's the difference between deep learning training and inference? *NVIDIA*, 2016.

[3] Sachin Sudhakar Farfade, Mohammad Saberian, and Li-Jia Li. Multi-view face detection using deep convolutional neural networks. 2015.

[4] Christian Horn. Do we really need swap on modern systems?

[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] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. volume 60, New York, NY, USA, may 2017. Association for Computing Machinery.

[7] G. Levi and T. Hassncer. Age and gender classification using convolutional neural networks. In *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 34–42, 2015.

[8] Arch Linux. cgroups - archwiki, May 2020.

[9] Akhil Mathur, Nicholas Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. pages 68–81, 06 2017.

[10] Oracle. Oracle vm virtualbox, May 2020.

[11] Rafael Stahl, Zhuoran Zhao, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. *Fully Distributed Deep Learning Inference on Resource-Constrained Edge Devices*, pages 77–90. 08 2019.

[12] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[13] Model Zoo. Model zoo - caffe deep learning code and models, May 2020.

Appendices

A ETHICAL CONCERNS

In this section, we discuss some of the major ethical concerns related to the execution of deep inference jobs on edge devices and the impact our research might have on them.

Discrimination. A major issue with deep learning and artificial intelligence in general, is that these algorithms can be biased. This issue know as algorithmic bias can be the cause of many discriminating outcomes Our research on the other hand, does not affect this issue. Indeed, the deep neural models used in our empirical study are all pre-trained. The different inference scheduling policies do not affect the accuracy of these models.

Humanity. The ever-growing interest in "Smart products" as well as the integration of various innovations has sparked a revolution in the production of edge devices, considerably expanding the range of the IoT. Reinforcing this revolution is the paradigm shift from cloud to edge computing, which only gives these devices more and more capabilities. Our research on the partial loading of deep network layers further supports this paradigm shift and allows for more demanding inference jobs to be executed on these devices. Which raises the question of how far machines will affect our behaviour and interaction?

Privacy and security concerns. One of the biggest controversies surrounding cloud computing is the privacy and security concerns about personal and sensitive data. By sending the data to the cloud for further processing, this data becomes at risk of being abused by the cloud provider itself, a hacker or another third party. By moving deep learning to the edge, we are essentially keeping the data that would otherwise be send to the cloud, as we use it immediately for processing. After the inference jobs are terminated, only the inference output is necessary and the input data doesn't have to be retained, therefore ensuring a more robust privacy of data. However, other then having to ensure virtual privacy and security, IT companies are going to have to ensure physical security as well. Indeed, edge devices are required to physically protect the data that is stored on them. If for example an edge device is hacked while collecting data, the attacker could manipulate the device to misunderstand the data it collected. As for the privacy concerns surrounding the EdgeCaffe environment, the data used as input for our inference tasks is not retained, only the output is stored in a csv file.

B DEPENDENCY GRAPHS

In this section, the generate dependency graphs of the AgeNet [7] model are displayed for each of the policies. The following figures are shown:

- Figure 5: AgeNet network dependency graph for the Bulk policy
- Figure 6: AgeNet network dependency graph for the DeepEye policy
- Figure 7: AgeNet network dependency graph for the Linear policy
- Figure 8: AgeNet network dependency graph for the Partial policy

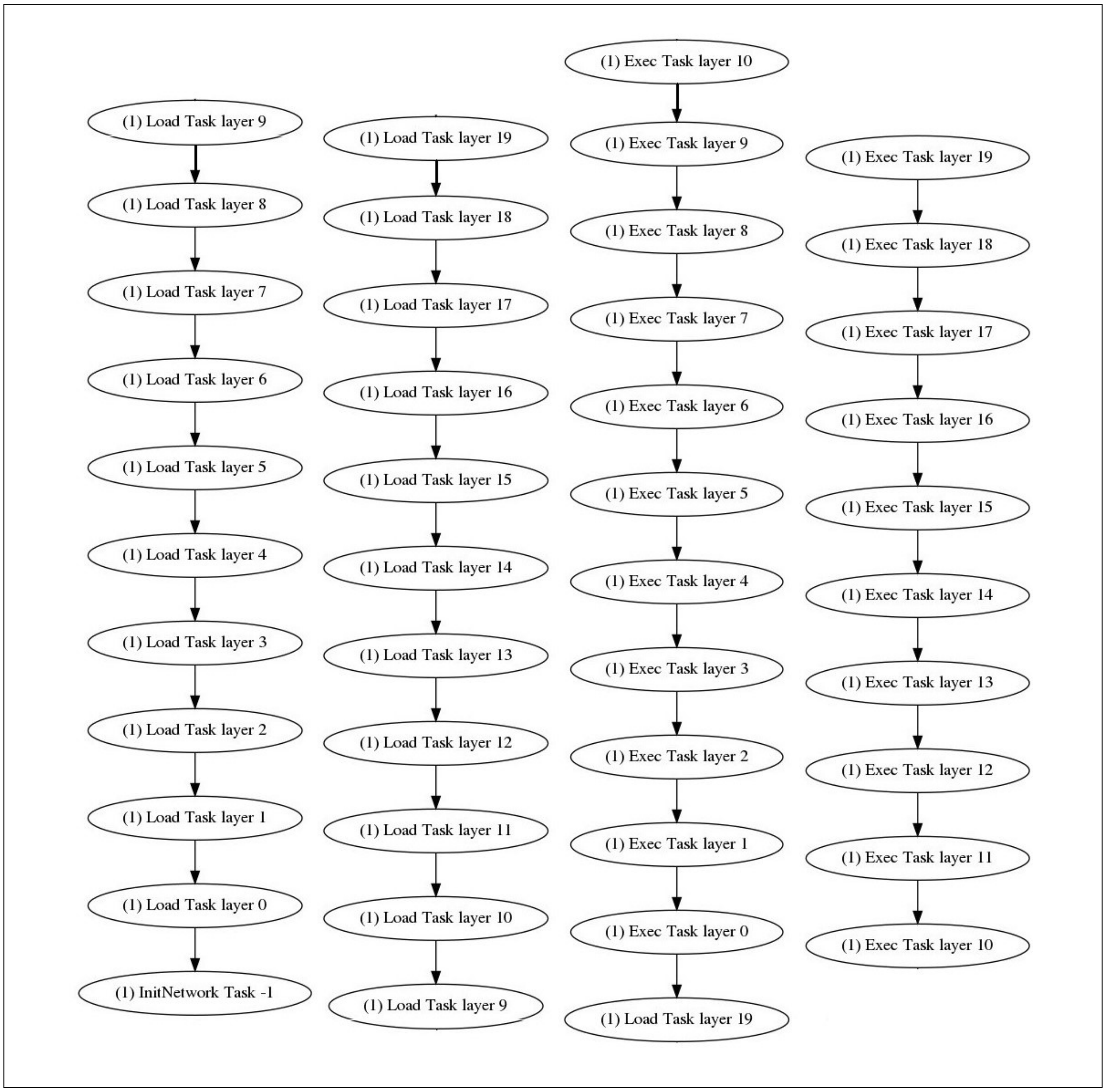


Figure 5: AgeNet network dependency graph for the Bulk policy

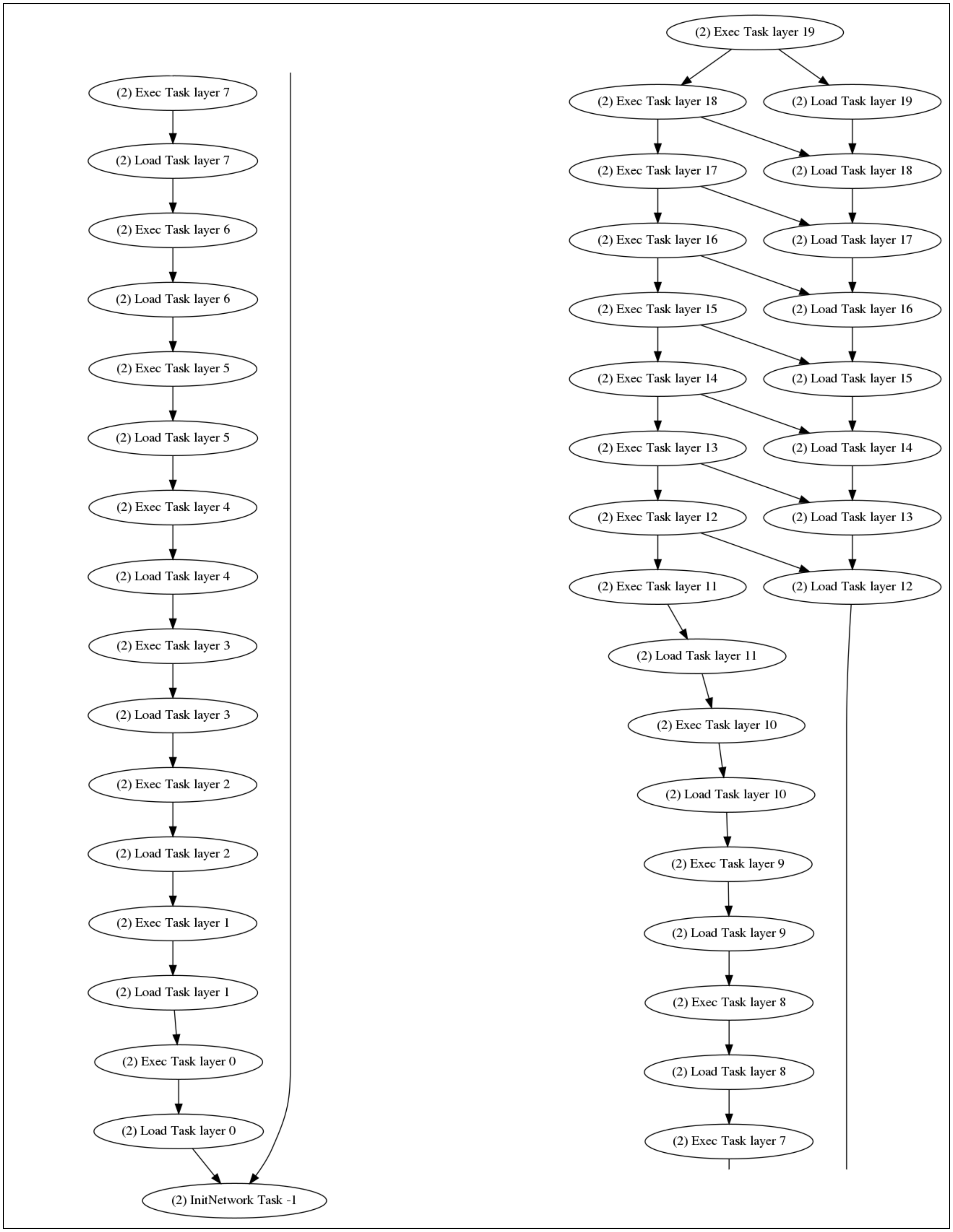


Figure 6: AgeNet network dependency graph for the DeepEye policy

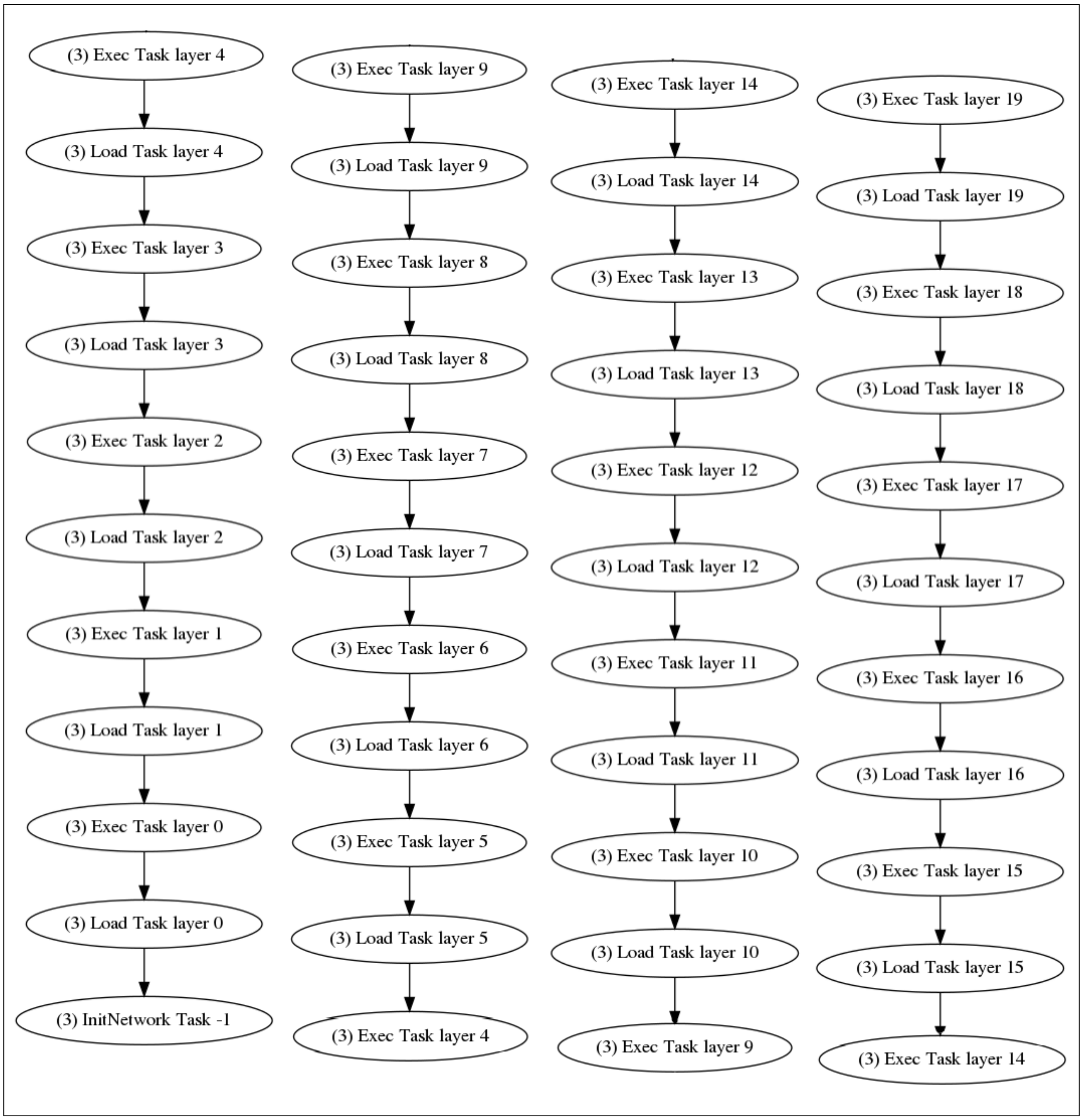


Figure 7: AgeNet network dependency graph for the Linear policy

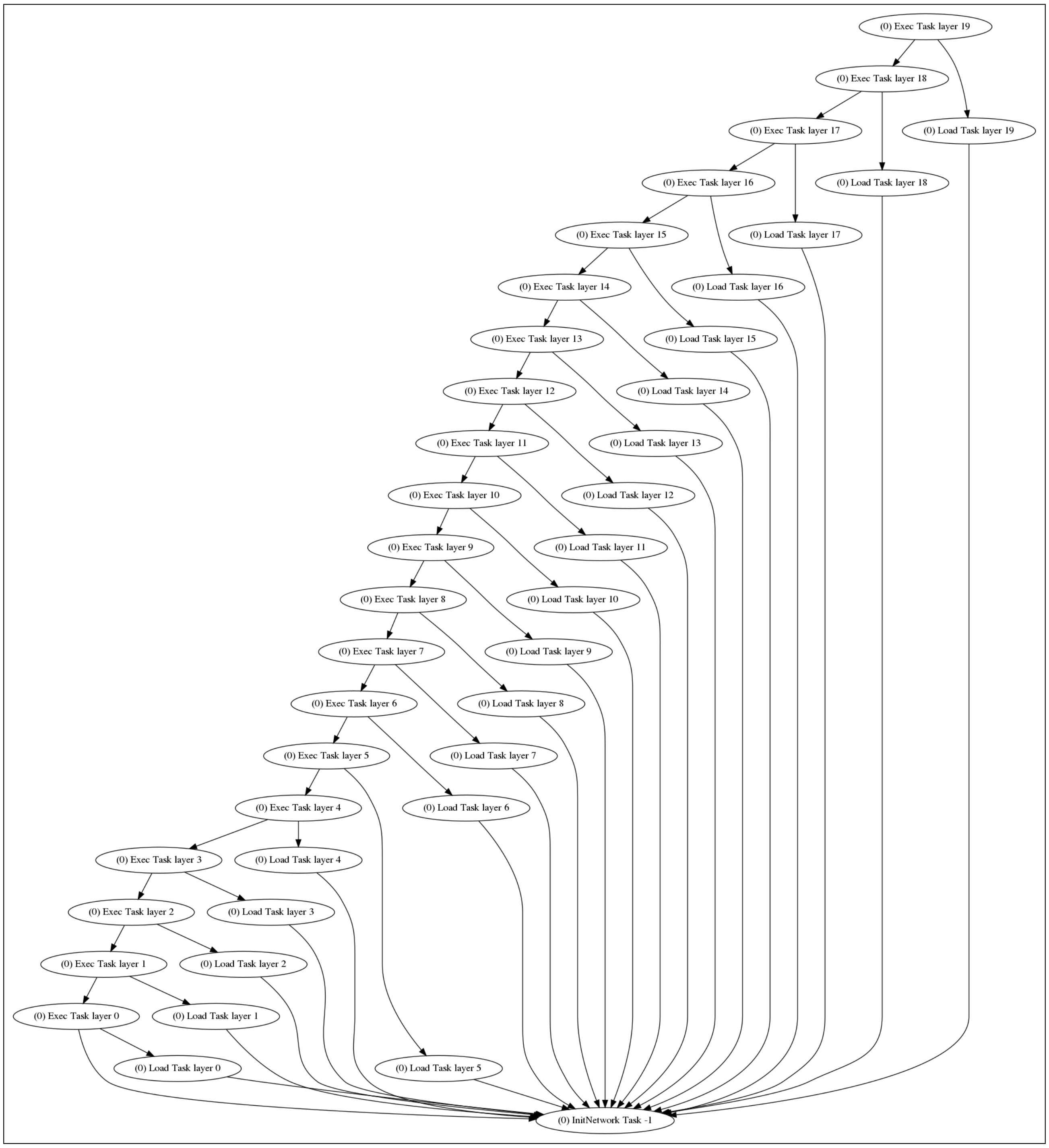


Figure 8: AgeNet network dependency graph for the Partial policy