Open access • Journal Article • DOI:10.1007/S10115-017-1029-1

The cascading neural network: building the Internet of Smart Things

Sam Leroux, Steven Bohez, Elias De Coninck, Tim Verbelen ...+3 more authors

Institutions: Ghent University

Published on: 01 Sep 2017 - Knowledge and Information Systems (Springer London)

Topics: Deep learning, Cloud computing, Telecommunications network, Mobile device and The Internet

Knowledge and Information Systems manuscript No.
(will be inserted by the editor)

The Cascading Neural Network: Building the Internet of Smart Things

Sam Leroux · Steven Bohez · Elias De Coninck · Tim Verbelen · Bert Vankeirsbilck · Pieter Simoens · Bart Dhoedt

Received: Sep 22, 2015
Revised: Jan 04, 2017
Accepted: Jan 14, 2017

Abstract Most of the research on deep neural networks (DNNs) so far has been focused on obtaining higher accuracy levels by building increasingly large and deep architectures. Training and evaluating these models is only feasible when large amounts of resources such as processing power and memory are available. Typical applications that could benefit from these models are however executed on resource constrained devices. Mobile devices such as smartphones already use deep learning techniques but they often have to perform all processing on a remote cloud. We propose a new architecture called a Cascading network that is capable of distributing a deep neural network between a local device and the cloud while keeping the required communication network traffic to a minimum. The network begins processing on the constrained device and only relies on the remote part when the local part does not provide an accurate enough result. The Cascading network allows for an early stopping mechanism during the recall phase of the network. We evaluated our approach in an Internet Of Things (IoT) context where a deep neural network adds intelligence to a large amount of heterogeneous connected devices. This technique enables a whole variety of autonomous systems where sensors, actuators and computing nodes can work together. We show that the Cascading architecture allows for a substantial improvement in evaluation speed on constrained devices while the loss in accuracy is kept to a minimum.

Ghent University - iMinds
Gaston Crommenlaan 8/201
B-9050 Ghent, Belgium
E-mail: first.lastname@intec.ugent.be

Keywords Neural Networks · Internet of Things (IoT) · Deep learning · Distributed systems and applications · Cloud computing · Mobile systems · Ubiquitous and pervasive computing

1 Introduction

In the past years deep artificial neural networks have proven to be exceptionally powerful for various machine learning tasks. Deep learning techniques are currently the state of the art for various machine learning tasks such as image and speech recognition or natural language processing [1]. While extremely capable, they are also resource demanding, both to train and to evaluate. Most of the research on deep learning focuses on training these deep models. Increasingly deep and complex networks are constructed to be more accurate on various benchmark datasets. Crucial for training these huge models are Graphical Processing Units (GPUs). High-end GPUs were once reserved for 3D modelling and gaming but their parallel architecture makes them also remarkably suitable for deep learning. The majority of the operations within a deep neural network are matrix multiplications and additions, two types of operations for which a GPU is orders of magnitude faster than a Central Processing Unit (CPU).

Training a deep neural network is computationally very expensive but efficient (distributed) GPU implementations now make it feasible to train a model considered too difficult to train in the past [2]. The time needed to train a deep neural network is in most cases not very critical. The evaluation of a trained model however can be extremely time sensitive. When the network is used to guide a robot or to interpret voice commands from a user, it should be able to operate in real-time. Any delay will result in poor user experience or possibly in dangerous situations when a robot or drone is involved. While training the network is often done on a high-performance system, once trained, the network has to be used in a real-world environment. The resources available to systems in these environments are much more limited.

In this paper, we focus on image classification problems using deep neural networks. The techniques presented here are however not limited to this domain but can be extended to all deep learning classification tasks. Possible applications include home automation and security systems, smart appliances and household robots. We want to use deep neural networks on constrained devices that are unable to evaluate the entire network due to limitations in available memory, processing power or battery capacity. Current wireless technologies are fast and affordable enough to consider offloading all the computations to a cloud back-end as a solution. This of course introduces an extra latency (10-500 ms) and makes the devices dependent on the network connection. This dependency may be unacceptable in some cases. A robot, for example, would become inoperable when the server can not be reached.

In this paper we strike a middle ground. A neural network consists of sequential layers where each layer transforms the output from the previous layer to a representation suitable for the next layer. Each layer extracts more complex features from its input. The last layer uses the high level features to classify the input. We exploit the inherent sequential design of a neural network to enable

an early stopping mechanism. We use the layers of a pretrained network as stages in a cascade. Each layer is able to capture additional complexity but also requires additional resources such as computing time and memory to store the parameters. Every stage classifies the input and returns a confidence value. We cease the evaluation of deeper layers once a certain required confidence threshold is reached. The choice of this threshold value allows us to trade-off accuracy and speed.

We proposed the concept of a Cascading network before in a conference paper [3]. Here, we extend this work by including a much more thorough evaluation on three typical IoT devices. We also include a validation of the architecture on a distributed neural network trained on real-world large color images (Imagenet dataset [4]).

The remainder of this paper is organized as follows. Section 3 introduces the Cascading architecture. Section 4 illustrates what kind of problems can be solved by this architecture. A thorough evaluation of the Cascading technique can be found in section 5 where our approach is tested on three well known datasets and on three types of resource-constrained devices. We begin in Section 2 with an overview of the related previous work and the differences with our approach.

2 Related work

2.1 Neural networks and deep learning

The basic architecture of neural networks dates back to the 1950s and the essence has not changed much since. A neural network contains interconnected layers of neurons. The knowledge of the network is stored in the weights of the connections between the nodes. In the 1980s it was proven that neural networks with a single hidden layer are universal approximators [5]. This theorem states that these simple neural networks can represent every possible function when given appropriate weights; it does however not state how to find these parameters or how many weights are needed.

Around 2006, interest in neural networks was renewed thanks to the advent of deep learning [6]. Advances in technology such as efficient GPU implementations and the availability of huge (labelled) datasets allowed to train increasingly deeper and complex network architectures. Currently (extremely) deep networks are the state of the art technique for image and speech recognition [7]. For a more in-depth overview of the history of neural networks and deep learning, we refer to [6].

2.2 Resource constrained machine learning

Both neural networks and other machine learning algorithms and techniques require vast amounts of resources, especially memory and processing power. The training phase of a neural network is the most computationally expensive. The gradient descent algorithm [8] used to tune the weights of the network needs multiple passes over the training set and each iteration requires multiple matrix multiplications and additions. Much of the research on distributed neural networks has thus been focused on architectures for the distributed training of deep networks on huge amounts of data. The most famous example of this is the Google DistBelief [9] system, capable of training extremely large neural networks on 1000s of machines and 10000s of cpu cores.

While the resources available when training a network are almost unlimited, the evaluation of the trained network is often done on a budget. We sometimes want to add the intelligence of a deep neural network to a constrained device. Here, intrinsic restrictions on battery capacity, processing power and memory, limit the size and complexity of the network. Various works have proposed techniques to minimise the cost when evaluating a machine learning model [10][11].

The use of a cascade architecture in a machine learning model has been proposed before [12][13]. In [14], the authors present various topologies in which machine learning models can be combined to minimise the cost when evaluating the models. They describe how to construct a tree of classifiers where samples can follow an individual path. Each path looks at specific features of the input data. A cascade can be seen as a special case of a tree topology. The technique we present here differs from previous uses of a cascade topology in a machine learning model. Our cascade does not contain a set of independent feature extractors but is trained as a whole, as one big model. By including an early stopping mechanism in the form of intermediate output layers, we are able to reuse parts of the big model as a smaller model.

Recently, various techniques have been proposed to compress a trained neural network, making it more suitable for resource constrained devices such as smartphones, robots or drones. In [15] and [16], the authors show that a shallow network can learn to mimick a large, deep network, effectively compressing the deep architecture in a small network with similar properties. This allows the small network to obtain an excellent performance at a much lower cost, both in memory required to store the weights and in processing power needed to evaluate the network. It is also possible to compress an ensemble of neural networks into one network [17]. The technique proposed here (Knowledge Distillation: KD) trains a *student* network based on the output of an ensemble of *teacher* networks.

State-of-the-art networks are usually deep (number of layers) and wide (number of neurons per layer). In [18], a technique similar to the previous com-

pressing techniques is used to train very thin but deep networks based on large powerful networks. The depth of the networks is crucial since it encourages the reuse of features, and leads to more abstract and invariant representations at higher layers [19].

In [20] the authors present a network architecture called HashedNets. They exploit the redundancy inherent in neural networks to achieve reductions in model sizes, thereby making it possible to store the networks on devices with limited memory. The hashing technique is elegantly simple: a hash function is used to group weights in buckets. Every connection grouped in the same bucket shares a weight value. A similar result can be obtained when using reduced precision parameters in the network [21][22].

Deep neural network architectures contain thousands of neurons. A large improvement in runtime speed may be obtained by pruning the network. Optimal Brain Damage [23] uses second order derivatives to remove unimportant weights from the network. More recently, a technique to reduce the computational cost of convolutional neural network layers was proposed [24]. The *Perforated Convolutional Layer* introduced here only calculates a subset of the output exactly. The other outputs are approximated through interpolation.

Our cascading architecture also makes deep neural networks suitable for constrained devices but does it in a fundamentally different way. Our resulting model is not a compressed variant of the original network, in fact, the cascade model is even slightly larger than the original model since there are extra parameters required for the additional output layers. We make a model more suitable for distributed evaluation by introducing an early-stopping mechanism. The major advantage of this technique is that it allows for a runtime trade-off between accuracy and speed. A suitable threshold can be selected based on the required accuracy and on the available resources instead of having one network with a fixed accuracy and computational cost. The time needed to process one image depends on the complexity of the image whereas a normal implementation of a neural network uses the exact same steps for each image regardless of the different complexities. This concept of conditional computation has been recently proposed in other works as well. The most relevant of these approaches are the *Big-little neural networks* [25] where a *little*, fast to execute network is used to try to classify an input sample. The *big* network is only used when the confidence of the little network is less than a predefined threshold.

The Cascading architecture could be seen as a special case of a Big-little network where a part of the *big* network is used as the *little* network, therefore avoiding the overhead of storing two completely independent networks. Another advantage of the Cascade compared to the Big-little architecture is that the computations done by the first stage in the cascade are used by the latter stages when needed. The *Big* network in the Big-little architecture on the other hand needs to start again from scratch when the *little* network is unable

to classify the input. We compare the Cascade and the Big Little approach in section 5.1.

3 Architecture

We want to evaluate a trained deep neural network on a constrained device unable to hold all the parameters in memory or unable to perform the calculations in the required time. Instead of offloading the entire network to a cloud backend, we offload only a part of the network. The first layers are evaluated locally and the remote part is only required when these layers are unable to classify a sample with sufficient confidence. This early-stopping mechanism during the recall phase of the network makes sure that we only communicate with the cloud backend when it is absolutely required. By avoiding unnecessary data transfers to the cloud, we can reduce the average latency and cost when evaluating the network.

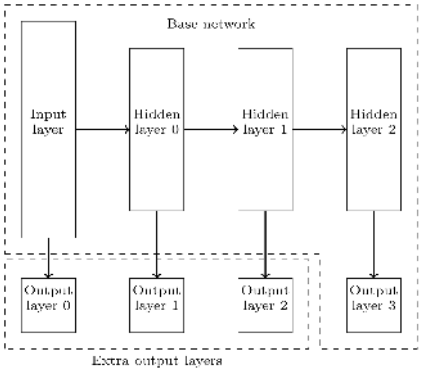


Fig. 1: The cascading architecture. The three additional output layers allow for an early-stopping mechanism when evaluating the network.

We slightly modify the standard architecture of a feed forward neural network to enable the early-stopping mechanism. Instead of one output layer (a softmax classifier) after the last hidden layer, we train multiple output layers: one directly on the raw input data and one after every hidden layer in the network. This allows to stop propagating a sample through the network once a sufficiently confident result is obtained. We use an interesting property of neural network classifiers stating that they provide outputs which estimate Bayesian a posteriori probabilities [26], meaning the outputs can be interpreted as confidence measures (i.e. how confident is the network that a certain sample belongs to a certain class ?).

This approach is shown in Figure 1 for a neural network with three hidden layers. The technique used to propagate a sample through the network is illustrated in Algorithm 1. The network consists of n hidden layers and $n+1$ output layers.

Algorithm 1 Propagating a sample through the cascade network: Keep evaluating the hidden layers until a confident result is obtained.

```
1: procedure FPROP(x)
2:     i ← 0
3:     y ← output_layer_i(x)
4:     while max(y) < threshold_i  &  i < n do
5:         x ← hidden_layer_i(x)
6:         i ← i + 1
7:         y ← output_layer_i(x)
8:     return y
```

3.1 Training

A Cascade network is trained as follows. We append additional output layers (softmax classifiers) after all or after a subset of the hidden layers and use standard backpropagation to train the layers. It is possible to train all the layers at once. The error backpropagated to a certain parameter is the (weighted) average of the error of every output layer for that parameter.

It is also possible to reuse a pre-trained off-the-shelf network. Research has shown that the features learned by the first layers of a deep neural network are often not specific to one problem but can be generalized over different datasets [27]. A popular approach to train a powerful network is to reuse the first layers of a publicly available pre-trained network and to replace the layers at the end of the network. The network as a whole is then fine-tuned on the problem specific dataset. This technique makes it possible to train a complex network on a relatively small amount of data since the first layers of the network already are suitable feature extractors.

Converting a completely trained traditional network to a cascade network can be done very fast at a small cost when keeping the weights fixed. We propagate the training set data once through the network and store the internal representations after every hidden layer. We then train softmax output layers to classify the stored representations. This second approach is used in all our experiments.

4 Use cases

The principal use case aims at evaluating a large neural network on a device unable to hold all the parameters in memory or unable to do the required calculations in the given time window. Instead of offloading the entire network to the cloud, we run a part of the network locally and only rely on the cloud server when absolutely necessary.

The delay introduced by offloading the computations to a server in a datacenter may be unacceptable for real-time applications such as a control system for a robot. An interesting idea is to bring the cloud closer. Fog computing [28] aims at reducing the physical distance between the user and the cloud. Local computation nodes (cloudlets [29]) can be used as a substitute for remote cloud servers. Technological advancements allow for ever more powerful systems in a smaller, more energy efficient package but these local systems will always fall behind the remote cloud servers where space and energy is abundant.

In most cases, neural networks are simulated in software on general purpose hardware. While extremely flexible, this paradigm is not the most efficient way to evaluate a neural network. Neuromorphic chips [30] are hardware components, specially designed to accommodate a neural network. They require less power to run and are able to generate an output faster. They are still expensive and hard to obtain at the moment and the amount of neurons they can contain is relatively small for any real-world network. The cascade architecture however would allow for a potentially very powerful hybrid network. The first layers are evaluated on the fast neural network hardware. The deeper layers, simulated in software, are only needed when the first layers were unable to classify the sample confidently. A similar architecture could incorporate Field Programmable Gate Arrays (FPGAs) to evaluate the first layers. The potential of FPGAs as a hardware accelerator for deep neural networks has been well documented [31] but practical applications are still rather uncommon.

The Cascading paradigm also allows for a more robust fault-tolerant system. Internet connectivity can be unstable in many practical situations. The Cascade network divides the neural network into different parts. One part is always evaluated locally so the system will still be able to operate when the Internet connection drops, although the accuracy will be lower.

The Cascade network decides whether to accept or to reject a classification based on the threshold value. This value is not hard-coded into the network but can be passed as an argument at runtime, independent for each sample. This can be useful in many practical situations since it allows a trade-off between accuracy and speed. Similarly, the threshold could depend on other measurements such as network latency or the cost associated with the network connection (WiFi vs mobile connections).

A possible architecture enabled by the Cascade network is shown in Figure 2. The first layers are evaluated on the robot, either by an on board neuromorphic

chip or by the embedded CPU or GPU. Offloading the computations is only needed when these layers are unable to classify the input. A local computation node (cloudlet) is used for the intermediate layers. The cloudlet can be reached by a local low latency network connection. Sending data to the cloud introduces a higher latency and is only required when the deeper layers are needed.

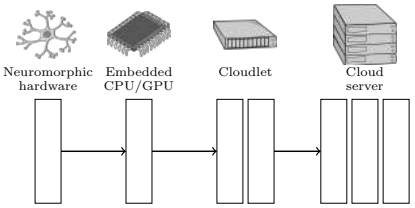


Fig. 2: A deep neural network with the layers distributed between devices. The first two hidden layers are evaluated on the robot. The intermediate representations can be transferred to the cloudlet or even to the cloud when needed.

5 Experimental results

In this section, we present the results obtained on three well known image classification datasets (MNIST, CIFAR10 and ImageNet 1K). These datasets represent increasingly difficult tasks that require increasingly complex networks and amounts of training data. All experiments described here were performed using the Theano framework [32].

We used an Nvidia GTX980 and an Nvidia Tesla K40 GPU for training. We used three devices typical for an IoT-context to validate our approach.

Each experiment was performed on a different device. A summary of the system specifications can be found in Table 1.

The Raspberry Pi[1] was originally developed to teach basic programming skills in schools. It quickly became a favourite platform for developers to build Internet of Things (IoT) systems because of the small physical size and affordability. The Intel Edison[2] was, in contrast to the Raspberry Pi, specially designed with IoT applications in mind. The Edison includes a 500 MHz Atom processor together with WiFi and Bluetooth connectivity in a package half the size of the Raspberry Pi. Its size and typical power consumption of less than 1W make it even suitable for wearable applications. The Nvidia Jetson TK1[3] finally is

[1] https://www.raspberrypi.org/

[2] http://www.intel.com/content/www/us/en/do-it-yourself/edison.html

[3] http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html

(a) Raspberry Pi 2

(b) Intel Edison

(c) Nvidia Jetson TK1

Fig. 3: Resource constrained devices used for testing

Table 1: Summary of the device specifications

	Raspberry Pi 2	**Intel Edison**	**Nvidia Jetson Tk1**
CPU	900MHz quad-core ARM Cortex-A7	500MHz Dual-core Atom processor MCU	NVIDIA 2.32GHz ARM quad-core Cortex-A15
GPU	Broadcom VideoCore IV @ 250 MHz	N/A	NVIDIA Kepler GPU 192 SM3.2 CUDA cores
Memory	1GB (shared with gpu)	1GB	2GB (shared with gpu)
Dimensions	85mm x 56mm	60mm x 29mm	127mm x 127mm
Power	$\approx$ 3W	$\approx$ 1W	$\approx$ 12W

a very powerful (considering its size and price) single board computer. The Jetson includes a Kepler GPU with 192 CUDA cores which makes it perfect for deep learning. The TK1 is especially suited for robotics and automotive applications. These three devices are shown in Figure 3.

5.1 MNIST

The MNIST dataset [33] is arguably one of the most common benchmark datasets for image recognition. It consists of a 60,000 sample training set and a 10,000 sample test set. The samples are 28 by 28 pixel black and white images of handwritten digits. While this dataset is a relatively easy task for most state-of-the-art models, it is still interesting as a first evaluation of new techniques since the amount of data is relatively small. The human performance on this dataset is estimated at an error rate of 0.2% [34]. Deep (convolutional) neural networks are able to achieve similar performance levels [35]. Some typical examples of the digits in this dataset are shown in Figure 4.

We trained the basic fully-connected architecture shown in Figure 5 to obtain an error rate of 0.69% on the MNIST dataset. All neurons are Rectified

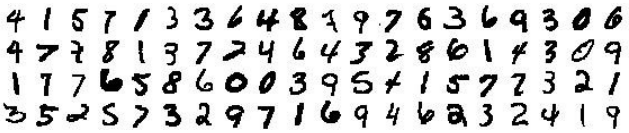


Fig. 4: The MNIST dataset consists of 28 by 28 pixel black and white images of handwritten digits.

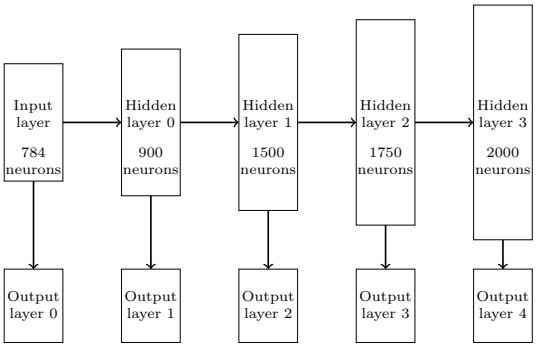


Fig. 5: 4 layer fully connected MNIST Cascade network. Four additional outputlayers were trained.

Linear units (ReLUs) [36]. A fixed momentum [37] value of 0.9 was used during training.

Dropout [38] and L2 regularization proved to be essential in training this network. We used the infimnist code[4] [39] to generate additional training samples by applying pseudo-random deformations and translations to the original MNIST training set.

Table 2 shows the accuracy of the different output layers in the network and the corresponding runtime on the Raspberry Pi 2. These results confirm the premise that deeper neural networks are usually capable of more accurate classification than shallow ones. This also proves that it is indeed possible to have a hidden layer that functions as an input for another hidden layer and simultaneously for a softmax output layer. While additional hidden layers are able to improve the classification accuracy, they also increase the computational cost and memory requirements of the network.

The Softmax output layer trained directly on the raw input data is still able to achieve a 91.29% accuracy rate. This suggests that the greater part of the network is only needed for a minority of the data samples. The cascading architecture allows us to exploit this property by providing an early-stopping mechanism.

[4] http://leon.bottou.org/projects/infimnist

Table 2: Accuracy and runtime on the Raspberry Pi 2 of the network at varying depths.

Layer number	Test error rate	Average time (in ms) needed to process one test sample on the Raspberry Pi 2
0	8.71%	0.85 ± 0.01
1	2.46%	8.76 ± 0.09
2	1.02%	22.03 ± 0.15
3	0.75%	48.17 ± 0.17
4	0.69%	80.11 ± 0.26

Table 3: Accuracy and runtime of the cascade using varying thresholds, evaluated on the Raspberry Pi 2.

Threshold	Test error rate	Average time (in ms) needed to process one test sample on the Raspberry Pi 2
0.5	5.37%	1.32 ± 0.02
0.7	2.44%	2.71 ± 0.02
0.9	1.03%	8.40 ± 0.06
0.95	0.82%	12.89 ± 0.06
0.99	0.72%	28.84 ± 0.11
0.995	0.69%	34.16 ± 0.11
0.999	0.69%	53.97 ± 0.18

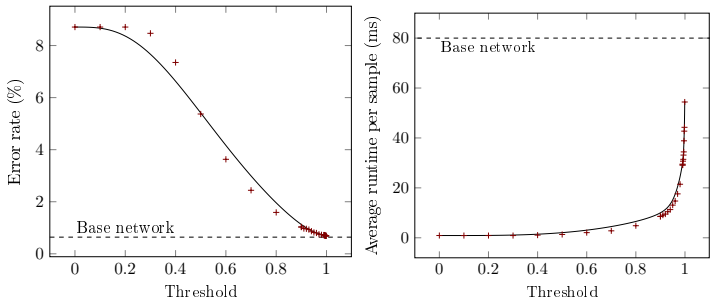
The test error rate and the corresponding runtime of the cascade on the Raspberry Pi 2 are presented in Table 3. These results are also graphically summarized in Figure 6. The same threshold is used for every layer. This experiment confirms the advantages of the Cascade network. The Cascade is able to achieve the same error rate as the base network while the required runtime is less than half the time needed for the base network.

Some random samples classified by each layer are shown in Table 4. This gives a qualitative idea of what type of samples are classified by each layer. These images confirm our intuitive expectations, the uncomplicated samples are classified by the early layers while the harder samples are left for the deeper layers.

We can distinguish the harder from the easier classes in a similar way. Table 5 shows for each class and for each layer the percentage of the samples of that class that are classified by the layer. Images of a handwritten zero are relatively easy to classify, over a third of these samples are classified by the first output layer, trained directly on the raw input data. The digit one on the other hand poses more of a challenge to the network. Two possible explanations for the difficulty of this class are the different styles of handwritten ones and the fact that a vertical pen stroke is also present in other classes such as four or seven.

Table 6 reveals the total percentage of samples classified by each layer. While the first output layer is capable of an accurate classification in 91.29% of

the samples, only 16.47% are classified by this layer because of the threshold imposed by the cascade.



(a) Error rate (b) Runtime

Fig. 6: Accuracy and runtime (measured on the Raspberry Pi 2) of the MNIST cascade network using varying thresholds. A larger threshold requires the network to be more confident of the result. The error rate is lower but the computational cost is higher. The accuracy and the runtime of the base network are indicated by the dashed horizontal line.

Table 4: Typical images classified by different layers. The easier samples are classified by the first layers while the harder samples are left for the deeper layers. (threshold=0.99)

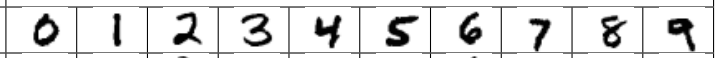
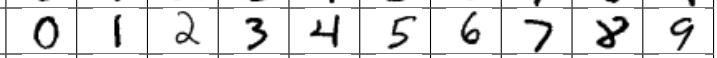
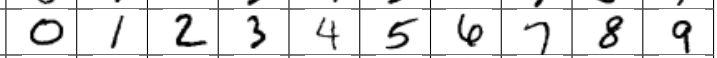
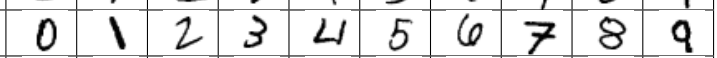
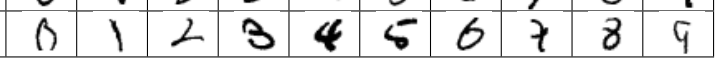
output layer	Typical samples classified by this layer									
0										
1										
2										
3										
4										

Table 5: Percentage of the test samples classified by each layer (threshold = 0.999)

Layers	Classes				
	0	1	2	3	4
0	37.86%	2.56%	29.94%	17.82%	12.32%
1	23.57%	24.58%	17.54%	25.54%	20.77%
2	22.04%	57.09%	22.77%	33.76%	41.04%
3	12.14%	10.57%	21.71%	15.05%	16.50%
4	4.39%	5.20%	8.04%	7.82%	9.37%
	5	6	7	8	9
0	4.48%	29.85%	25.00%	4.83%	0.69%
1	37.33%	18.37%	24.61%	8.42%	1.98%
2	28.70%	29.54%	17.41%	42.81%	50.84%
3	19.28%	15.24%	20.62%	33.78%	35.08%
4	10.20%	6.99%	12.35%	10.16%	11.40%

Table 6: Total percentage of the test samples classified by each layer (threshold = 0.99)

Layer	Total percentage of samples classified
0	16.47%
1	20.17%
2	34.91%
3	19.90%
4	8.55%

5.2 CIFAR10

While the MNIST dataset contained relatively uncomplicated images of numeric digits, the CIFAR10 dataset [40] contains images of complex types of objects. This dataset consists of 60,000 32 by 32 pixel color images in 10 classes. Some of the classes include: airplane, car, truck, cat and dog. Human level performance is estimated at an accuracy of 94% [41], the current state-of-the-art models are able to achieve human performance (93.57%)[42]. Some typical samples are shown in Figure 7.

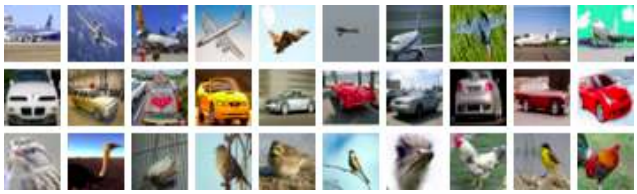


Fig. 7: The CIFAR-10 dataset contains 32 by 32 pixel color images of ten classes such as car, truck, cat and dog.

We trained the convolutional architecture shown in Figure 8 to obtain an accuracy of 84.26%. The network consists of three convolutional layers with 64 5 by 5 filters each and one fully connected layer with 1024 neurons at the end. The non-linearities are all Rectified Linear Units (ReLU) [36]. We used stochastic gradient descent with a fixed momentum value of 0.9 to train these layers. Dropout [38] with probability of 0.5 was used on the fully connected layer. The input image data was rescaled to have zero mean and unit variance but no other preprocessing or data augmentation techniques were used.

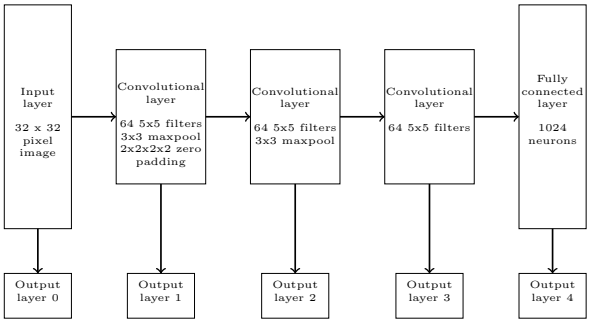


Fig. 8: 4 layer convolutional CIFAR10 network.

The Intel Edison was chosen as the test platform for this experiment. Table 7 shows the error rate that can be obtained by the different subnetworks in the cascade and the corresponding runtime on the Edison. We also include the accuracy when each path is trained completely from scratch. This to investigate the impact of training softmax output layers on the intermediate representations. We found that the penalty of using these already trained layers is small. The complexity of the images included in the CIFAR10 dataset poses more of a challenge than the MNIST digits. Yet, a single softmax classifier trained on the raw pixel data is still able to classify 41.85% of the test set correctly. This suggest that the Cascade could also allow for a speed-up on this more complicated dataset.

Table 8 shows the obtained test error rate and the required runtime on the Intel Edison using various thresholds. The Cascade again allows for a speed up, although less spectacular than the MNIST cascade. The average runtime of the Cascade with a threshold of 0.95 is 25% less than the runtime of the base network at a marginal increase in error rate (15.97% instead of 15.74%).

Even though the Cascade allows for a gain in speed when evaluating the network on one machine, this is not the main goal of this architecture. The Cascade is even more advantageous when it is used to distribute the layers over different machines, as described in the following experiment.

Table 7: Accuracy and runtime of the CIFAR10 network at varying depths, evaluated on the Intel Edison.

Layer number	Test error rate	Test error rate (from scratch)	Average time (in ms) needed to process one test sample on the Edison
0	58.15%	58.15%	0.99 ± 0.01
1	29.31%	28.33%	13.90 ± 0.02
2	18.11%	18.05%	37.62 ± 0.02
3	16.24%	16.05%	39.72 ± 0.03
4	15.74%	15.74%	56.70 ± 0.03

Table 8: Accuracy and runtime of the cascade using varying thresholds, evaluated on the Intel Edison.

Threshold	Test error rate	Average time (in ms) needed to process one test sample on the Edison
0.5	28.57%	14.34 ± 0.02
0.7	20.13%	25.72 ± 0.02
0.8	18.13%	31.63 ± 0.02
0.9	16.40%	36.54 ± 0.03
0.95	15.97%	41.89 ± 0.03
0.99	15.74%	47.62 ± 0.03

The Cascading approach exploits the fact that not all possible input samples are equally hard to classify and that even a small network is able to capture enough information to allow a correct classification. In the worst case all layers of the network are used but the amortized cost over all samples should be lower. A similar approach is also presented in [25]. Here the authors propose a mechanism with two independent networks. First a "little network" is used. This is a low cost, fast to execute model. The second ("Big") network is only used for those input samples where the "little" network is not confident in the output. Our cascading technique could be seen as a special case where we do not have two completely independent models but where instead we provide an early-stopping mechanism in the network. Our "little" network is part of the "Big" network. This allows us to reduce the memory footprint of the system and this also allows us to build upon the computations of the first stage when the deeper layers are needed (compared to starting over from scratch in the Big-little technique). The little network in the Big-little technique however is not forced to be useful as a part of the Big network which means that the architecure of both networks can be optimised independently, something that is not possible for the Cascade.

We have implemented a basic version of the Big-little technique to compare against the Cascade. We used the Cifar10 network described before as the Cascade. We based the Big-little version on the same network. The "little" network consists of one convolutional layer (64 5*5 filters) and a softmax output layer. The "Big network" is the same network as used in the Cascade. The results are shown in Figure 9. The reported runtime is measured on the

Intel Edison. This graph shows that the Cascade allows for more flexibility to trade-off accuracy and speed since the cascade has multiple decision points compared to just one in the Big-little architecture. We also find that for these networks the Cascade approach is able to obtain the same accuracy level at a lower computational cost.

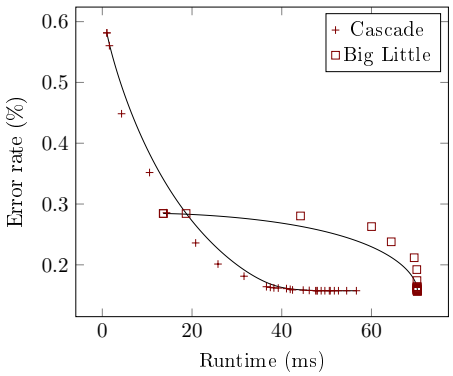


Fig. 9: Accuracy as a function of runtime for both the Cascade and the Big Little architecture, measured on the Intel Edison.

5.3 ImageNet

The previous two datasets are excellent default benchmark datasets but do not really capture the complexity of real-world high-resolution images.

The ImageNet dataset [4] contains millions of images, organized following the WordNet [43] hierarchy. Wordnet can be seen as a linked database of English words grouped in sets of synonyms (synsets). ImageNet contains manually labelled high resolution images for a subset of these words. At the moment of writing (September 2015), ImageNet contains 14,197,122 images in 21841 synsets for an average of 650 images per synset.

A subset of the data is used in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [44]. This challenge has been run annually since 2010 and every year, new state-of-the-art results were obtained. The 2014 dataset contained 1,281,167 training images, a validation set of 50,000 images and a 100,000 test set. There were 1000 classes and each class had at least 732 training images. Some typical examples of the images included in this dataset are shown in Figure 10.

The accuracy on this challenge is most often measured using the top-5 test error rate (the model is allowed to guess 5 times). The human performance on

this dataset is hard to measure but is estimated at an error rate of 5.1% [44]. Recently, a deep convolutional neural network outperformed humans when it achieved a 4.94% top-5 test error rate [45].

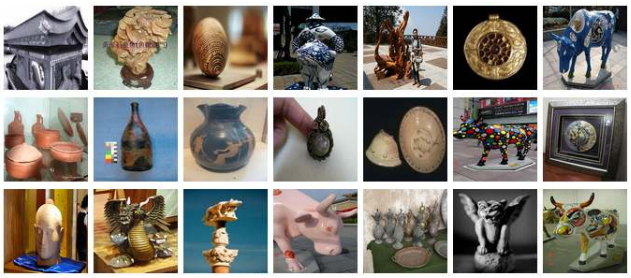


Fig. 10: Sample ImageNet images

The strength of the ImageNet dataset is its size but this size also makes training a model very challenging. For our experiments, we choose not to train a network from scratch but re-used a pre-trained network. We used the Overfeat network [46]. Overfeat was designed for the 2013 ILSVRC contest where it obtained very competitive results.

There are two versions available for download, a fast version and an accurate version. Both have a similar architecture. The fast network achieves a 16.39% top-5 error rate on the ILSVRC 2013 test set while the accurate network obtains a 14.18% top-5 error rate [46].

The Overfeat network contains 5369 million connections, requiring 144 million weights [46]. Every weight is a 32 bit floating point number, this means that at least 576 MB of memory is required just to store the weights. Even more memory is temporarily needed when using the network. These memory requirements, combined with the needed processing power makes it practically impossible to evaluate a network of this size on most embedded devices.

We transformed the pretrained Overfeat network into a Cascade by training two additional output layers after the second and the fourth convolutional layer. The intermediate representations after these layers are large (respectively 57600 and 115200 elements). We applied an eight by eight max pooling operation just before the softmax layers to reduce the dimensionality and to make it easier for the softmax layers to learn a suitable classification. Figure 11 shows the components of the Overfeat network and the extra cascading layers. Traditional stochastic gradient descent with a momentum value of 0.9 was used to train the output layers. Dropout (with probability=0.5) proved to be crucial to reliably train these layers. The weights of the base network were kept fixed.

Table 9 summarizes the results that can be obtained by the different output layers in the network. The first output layer is able to achieve a top-5 accuracy of 33.83% which is impressive for a network with only two convolutional layers (a random guess would yield a top-5 accuracy of 0.5%). The next two convolutional layers are able to improve this result to a top-5 accuracy rate of 51.7%.

The last output layer is the pretrained Overfeat Softmax layer and is able to obtain a top-5 accuracy of 81.59%. All calculations were performed on the Nvidia GTX980 GPU. Each sample was processed one at a time by the GPU to simulate an environment where each image has to be processed as soon as it becomes available.

We then evaluated the required runtime and the obtained accuracy of the cascade with varying thresholds. Table 10 shows that the cascading architecture even allows for a small speed-up when evaluating the network on a GPU.

The real strength of this architecture however becomes apparent when we distribute the neural network between devices. To demonstrate this, we built an experimental set-up where the network is distributed between the Jetson TK1 board and a GPU server (GTX980 GPU) in the cloud. The network connection between the two nodes was throttled to simulate real-world network

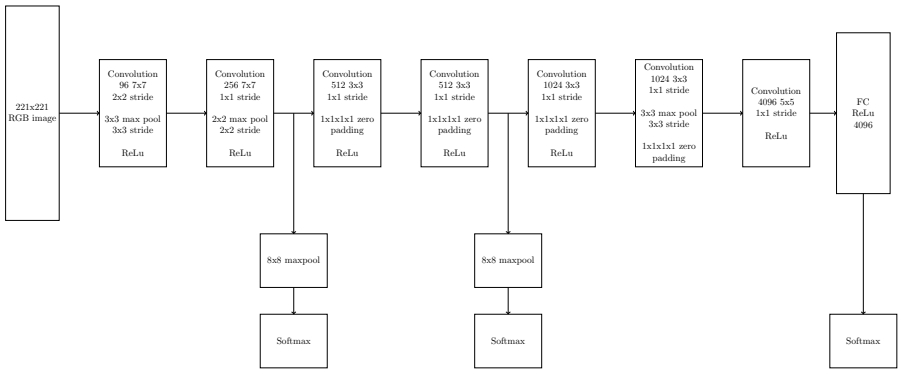


Fig. 11: The adapted Overfeat network with two extra output layers. Max-pooling is used to reduce the dimensionality before applying the additional softmax layers. A larger version of this image is included in Appendix A.

Table 9: Accuracy and runtime of the output layers in the Overfeat network when evaluated on an Nvidia GTX980 GPU

Outputlayer	**Top-1 accuracy**	**Top-5 accuracy**	**Runtime (ms)**
1	17.95%	33.83%	3.6 ± 0.001
2	29.49%	51.7%	7.2 ± 0.004
3	59.95%	81.59%	36 ± 0.018

Table 10: Accuracy and runtime of the Overfeat cascade when evaluated on an Nvidia GTX980 GPU

Threshold	**Top-1 accuracy**	**Top-5 accuracy**	**Runtime (ms)**
0.9	58.14%	79.79%	30 ± 0.02
0.99	59.73%	81.27%	34 ± 0.02
0.999	59.95%	81.59%	35 ± 0.02

Table 11: Local evaluation on the Jetson TK1 compared to full offload to the cloud with varying network bandwidth and latency.

Bandwidth (Mbit/s)	**RTT (ms)**	**Cloud (ms)**	**Local (ms)**
1	10	4853 ± 15	1110 ± 4
	100	4944 ± 16	
10	10	551 ± 1.5	
	100	639 ± 1.6	
100	10	121 ± 0.16	
	100	211 ± 0.65	

connections. For each architecture, we measured the required runtime with a network bandwidth of 1, 10 and 100 Mbit/s and a Round Trip Time (RTT) of 10 and 100 ms.

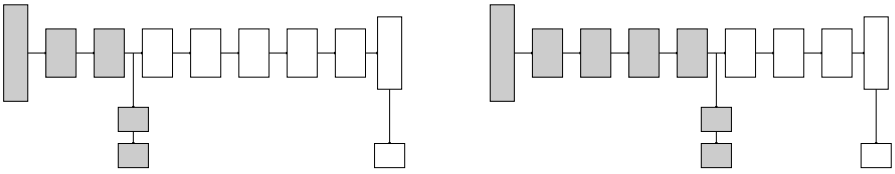
The two traditional options (local evaluation and full offload) are compared in Table 11. When all calculations needed by the Overfeat network are performed locally on the Jetson TK 1 GPU, it takes 1110 ms to process one image.

The alternative approach is to offload all the computations to the GPU server in the cloud. The time required by this technique will depend on the bandwidth and latency of the network connection. Table 11 shows that a complete offload to the cloud takes less time than the local computation except in the case of very limited bandwidth (1 Mbit/s).

The time needed to serialise and to transfer the data can quickly outweigh the time needed to do the actual calculations. The cascading architecture avoids sending data over the network when a confident classification can be made by the local part of the network. We evaluated the cascading network on the same machines using the same network parameters.

We compared two possible cascades, one with two local convolutional layers (and one maxpool + softmax layer) and one with four local convolutional layers (and one maxpool + softmax layer). These networks are illustrated in Figure 12.

Table 12 shows the required runtime of the first cascade with varying network bandwidth and latency. In the case of very limited bandwidth (1 Mbit/s), it takes over two seconds to process one image. The Jetson board is able to evaluate the entire network in just over 1 second so in this case it is less time consuming to do all the calculations locally. This however is only possible because the Jetson TK1 can hold the entire network in memory. On other



(a) 2 local layers (b) 4 local layers

Fig. 12: The two cascade networks, gray blocks are evaluated locally

Table 12: Accuracy and runtime of the Overfeat cascade with two local convolutional layers when using a threshold value t of 0.9,0.99 and 0.999

Bandwidth (Mbit/s)	**RTT (ms)**	t**=0.9 (ms)**	t**=0.99 (ms)**	t**=0.999 (ms)**
1	10	2143 ± 16	2390 ± 14	2490 ± 12
	100	2220 ± 17	2477 ± 14	2579 ± 12
10	10	299 ± 2	329 ± 3	341 ± 3
	100	375 ± 3	414 ± 3	430 ± 3
100	10	114 ± 1	123 ± 1	127 ± 1
	100	190 ± 2	208 ± 2	215 ± 2
Top 1 Accuracy		58.12%	59.67%	59.92%
Top 5 Accuracy		79.73%	81.33%	81.56%

devices, with less memory, offloading to the cloud would be unavoidable. The cascade network would allow for a 2X speed-up compared to a full offload in these cases.

A full offload in the case of a 10 Mbit/s connection with 10 and 100 ms RTT takes respectively 551 and 639 ms. The cascade with threshold 0.99 requires only 329 and 414 ms respectively. A speed-up of 40% while the drop in top-5 accuracy is negligible (-0.3%).

A high speed network connection (100 Mbit/s) makes offloading to the cloud less time consuming. The runtime of the cascade is statistically the same as a full offload in this case. The cascade could still be useful however since it provides a redundancy against network failure and could avoid costs related with wireless network connections.

We repeated the experiment but now with a larger local part. The first four convolutional layers are evaluated locally. The cascade offers little to no improvement in this case since the local computations take much longer and the data that needs to be transferred over the network is twice as large as the data sent over the network in the previous cascade. This to illustrate that the performance of the cascade will strongly depend on the choice of the local and the remote part.

Table 13: Accuracy and runtime of the overfeat cascade with four local convolutional layers.

Bandwidth (Mbit/s)	**RTT (ms)**	**t=0.9 (ms)**	**t=0.99 (ms)**	**t=0.999 (ms)**
1	10	4211 ± 20	4700 ± 20	4899 ± 20
	100	4285 ± 20	4785 ± 20	4986 ± 21
10	10	559 ± 5	614 ± 4	637 ± 4
	100	636 ± 6	702 ± 6	728 ± 6
100	10	191 ± 2	203 ± 2	210 ± 2
	100	268 ± 3	290 ± 2	297 ± 2
Top 1 Accuracy		58.14%	59.73%	59.95%
Top 5 Accuracy		79.79%	81.27%	81.59%

6 Conclusion

We presented a novel architecture called a Cascade network to avoid redundant calculations when evaluating a deep neural network model. In addition, this technique also allows for an elegant offloading mechanism where network communication is avoided when it is not absolutely necessary. The performance gain depends on the neural network architecture and on the hardware specifications. We evaluated our approach on three well known benchmark datasets (MNIST, CIFAR10 and Imagenet) and were able to speed up the evaluation of three standard network architectures while keeping the loss in accuracy to a minimum. The measurements were performed on three typical IoT devices, simulating real-world environments. For the MNIST network we are able to reduce the computational cost by half while keeping the same level of accuracy. On the CIFAR10 dataset we have a speedup of 20% with a marginal loss of accuracy. For the Imagenet dataset we distributed the well known Overfeat network. The network was evaluated partially on a local device and partially offloaded to the cloud. We measured the performance for different bandwidth and round trip times and found that we were able to reduce the average runtime by up to 40% depending on the network characteristics.

Acknowledgements Part of this work was supported by the iMinds IoT Research Program. Steven Bohez is funded by a Ph.D. grant of the Agency for Innovation by Science and Technology in Flanders (IWT). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU and the Jetson TK1 used for this research.

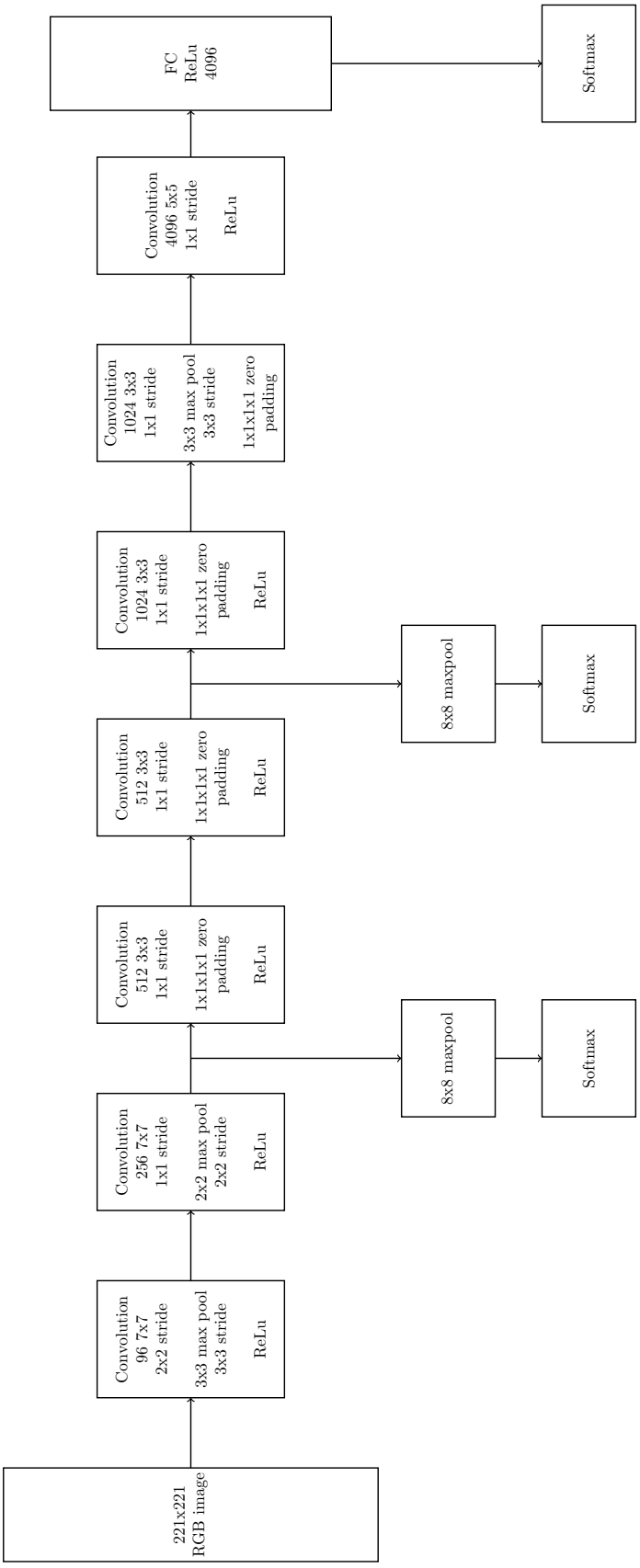
References

1. G. Hinton, Y. LeCun *et al.*, "Guest editorial: Deep learning," *International Journal of Computer Vision*, vol. 113, no. 1, pp. 1–2, 2015.
2. A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of the 30th international conference on machine learning*, 2013, pp. 1337–1345.

3. S. Leroux, S. Bohez, T. Verbelen, B. Vankeirsbilck, P. Simoens, and B. Dhoedt, "Resource-constrained classification using a cascade of neural network layers," in *IJCNN 2015*, 2015.
4. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 248–255.
5. K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.
6. J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.
7. Y. Bengio, "Learning deep architectures for ai," *Foundations and trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.
8. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, 1988.
9. J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1223–1231.
10. Z. Xu, K. Weinberger, and O. Chapelle, "The greedy miser: Learning under test-time budgets," *arXiv preprint arXiv:1206.6451*, 2012.
11. K. Singer, "Online classification on a budget," *Advances in neural information processing systems*, vol. 16, p. 225, 2004.
12. P. Viola and M. J. Jones, "Robust real-time face detection," *International journal of computer vision*, vol. 57, no. 2, pp. 137–154, 2004.
13. L. Lefakis and F. Fleuret, "Joint cascade optimization using a product of boosted classifiers," in *Advances in neural information processing systems*, 2010, pp. 1315–1323.
14. Z. E. Xu, M. J. Kusner, K. Q. Weinberger, M. Chen, and O. Chapelle, "Classifier cascades and trees for minimizing feature evaluation cost," *Journal of Machine Learning Research*, vol. 15, pp. 2113–2144, 2014.
15. C. Bucilu, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 535–541.
16. J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *Advances in Neural Information Processing Systems*, 2014, pp. 2654–2662.
17. G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *NIPS 2014 Deep Learning Workshop*, 2014.
18. A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for thin deep nets," *arXiv preprint arXiv:1412.6550*, 2014.
19. Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 8, pp. 1798–1828, 2013.
20. W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," *arXiv preprint arXiv:1504.04788*, 2015.
21. S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *arXiv preprint arXiv:1502.02551*, 2015.
22. M. Courbariaux, Y. Bengio, and J.-P. David, "Low precision arithmetic for deep learning," *arXiv preprint arXiv:1412.7024*, 2014.
23. Y. LeCun, J. S. Denker, S. A. Solla, R. E. Howard, and L. D. Jackel, "Optimal brain damage." in *NIPs*, vol. 89, 1989.
24. M. Figurnov, D. Vetrov, and P. Kohli, "Perforatedcnns: Acceleration through elimination of redundant convolutions," *arXiv preprint arXiv:1504.08362*, 2015.
25. E. Park, D. Kim, S. Kim, Y.-D. Kim, G. Kim, S. Yoon, and S. Yoo, "Big/little deep neural network for ultra low power inference," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2015 International Conference on*. IEEE, 2015, pp. 124–132.
26. M. D. Richard and R. P. Lippmann, "Neural network classifiers estimate bayesian a posteriori probabilities," *Neural computation*, vol. 3, no. 4, pp. 461–483, 1991.
27. J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in Neural Information Processing Systems*, 2014, pp. 3320–3328.

28. F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.
29. T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Cloudlets: Bringing the cloud to the mobile user," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM, 2012, pp. 29–36.
30. K. Boahen, "Neuromorphic microchips," *Scientific American*, vol. 292, no. 5, pp. 56–63, 2005.
31. K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, 2015.
32. F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," *arXiv preprint arXiv:1211.5590*, 2012.
33. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
34. Y. LeCun, L. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard *et al.*, "Learning algorithms for classification: A comparison on handwritten digit recognition," *Neural networks: the statistical mechanics perspective*, vol. 261, p. 276, 1995.
35. L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 2013, pp. 1058–1066.
36. V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 807–814.
37. B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.
38. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.
39. G. Loosli, S. Canu, and L. Bottou, "Training invariant support vector machines using selective sampling," in *Large Scale Kernel Machines*, L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, Eds. Cambridge, MA.: MIT Press, 2007, pp. 301–320. [Online]. Available: http://leon.bottou.org/papers/loosli-canu-bottou-2006
40. A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Computer Science Department, University of Toronto, Tech. Rep*, vol. 1, no. 4, p. 7, 2009.
41. A. Karpathy, "Lessons learned from manually classifying cifar-10," 2011.
42. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.
43. G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.
44. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *arXiv preprint arXiv:1409.0575*, 2014.
45. K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *arXiv preprint arXiv:1502.01852*, 2015.
46. P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," in *International Conference on Learning Representations (ICLR 2014)*. CBLS, April 2014.

A The Overfeat Cascading architecture



The adapted Overfeat network with two extra output layers. Maxpooling is used to reduce the dimensionality before applying the additional softmax layers. This is a larger version of Figure 11.

Author Biographies

Sam Leroux received his M.Sc degree in Information Engineering Technology from Ghent University, Belgium in July 2014. In September of that year, he joined the Department of Information Technology at Ghent University, where he is active as a Ph.D. student. His main research interests are machine learning, neural networks, deep learning and cloud computing. He is also active as a teaching assistant for various courses in both the bachelor and master of Science in Information Engineering Technology program.

Elias De Coninck received his M.Sc. in Information Engineering Technology from University College Ghent, Belgium in August 2012. He is now working on a Ph.D. at Ghent University - iMinds on hybrid cloud systems.

Steven Bohez received his M.Sc. degree in Computer Science from Ghent University, Belgium in June 2013. He is working on a Ph.D. at Ghent University - iMinds and is focusing on advanced mobile cloud applications that are distributed between mobile devices and the cloud.

Tim Verbelen received his M.Sc. degree in Computer Science from Ghent University, Belgium in June 2009. In July 2013, he received his Ph.D. degree with his dissertation "Adaptive Offloading and Configuration of Resource Intensive Mobile Applications". Since August 2009, he has been working at the Departement of Information Technology (INTEC) of the Faculty of Engineering at Ghent University, and is now active as postdoctoral researcher. His main research interests include mobile cloud computing and adaptive software. Specifically he is researching adaptive strategies to enhance real-time applications such as Augmented Reality on mobile devices.

Bert Vankeirsbilck received a M. Sc. Degree (2007) and a Ph.D. Degree (2013) in Computer Science Engineering from Ghent University. Since June 2013, he has been active as a post-doctoral research at the dept of Information Technology at the same university. From a Ph.D. topic on optimization of quality of experience for mobile thin client systems, the focus broadened towards resource constrained computing and distributed intelligence, mostly supported by software design based on edge cloud architectures.

Pieter Simoens received his M.Sc. degree in Electronic Engineering (2005) and Ph.D. degree (2011) from the Ghent University, Belgium. During his Ph.D. research, he was funded by the Fund for Scientific Research Flanders (FWO-V). In 2012, he was a visiting researcher at the School of Computer Science of Carnegie Mellon University, USA. Currently, he is assistant professor affiliated with the Department of Information Technology of the Ghent University and with iMinds. He is teaching courses on Mobile Application Development and Software Engineering.

His main research interests include mobile cloud offloading, service-oriented networking, edge/fog computing paradigms, and service engineering for advanced mobile applications. In these fields, he is author and co-author of more than 70 papers published in international journals or in the proceedings of international conferences. He has also been involved in several national and European research projects (FP6 MUSE, FP7 MobiThin, H2020 FUSION).

Bart Dhoedt received a Masters degree in Electro-technical Engineering (1990) from Ghent University. His research, addressing the use of micro-optics to realize parallel free space optical interconnects, resulted in a Ph.D. degree in 1995. After a 2-year post-doc in opto-electronics, he became Professor at the Department of Information Technology.

Bart Dhoedt is responsible for various courses on algorithms, advanced programming, software development and distributed systems. His research interests include software engineering, distributed systems, mobile and ubiquitous computing, smart clients, middleware, cloud computing and autonomic systems. He is author or co-author of more than 300 publications in international journals or conference proceedings..