DIANNE: Distributed Artificial Neural Networks for the Internet of Things

Elias De Coninck Tim Verbelen Bert Vankeirsbilck

Steven Bohez Sam Leroux Pieter Simoens

Ghent University - iMinds
Department of Information Technology
Gaston Crommenlaan 8/201
9050 Gent, Belgium
elias.deconinck@intec.ugent.be

ABSTRACT

Nowadays artificial neural networks are widely used to accurately classify and recognize patterns. An interesting application area is the Internet of Things (IoT), where physical things are connected to the Internet, and generate a huge amount of sensor data that can be used for a myriad of new, pervasive applications. Neural networks' ability to comprehend unstructured data make them a useful building block for such IoT applications. As neural networks require a lot of processing power, especially during the training phase, these are most often deployed in a cloud environment, or on specialized servers with dedicated GPU hardware. However, for IoT applications, sending all raw data to a remote back-end might not be feasible, taking into account the high and variable latency to the cloud, or could lead to issues concerning privacy. In this paper the DIANNE middleware framework is presented that is optimized for single sample feed-forward execution and facilitates distributing artificial neural networks across multiple IoT devices. The modular approach enables executing neural network components on a large number of heterogeneous devices, allowing us to exploit the local compute power at hand, and mitigating the need for a large server-side infrastructure at runtime.

Categories and Subject Descriptors

C.2 [**Computer-communication Networks**]: Distributed Systems—*Distributed applications*; I.5 [**Pattern Recognition**]: Models—*Neural nets*

Keywords

Distributed Artificial Neural Networks, Internet of Things, Middleware

M4IoT 2015, December 07-11, 2015, Vancouver, BC, Canada

DOI: http://dx.doi.org/10.1145/2836127.2836130

1. INTRODUCTION

The Internet of Things (IoT) is a popular paradigm that refers to the interconnection, wired or wireless, between all kinds of physical objects or "things", which generate information about the environment (sensors) and/or interact with the environment (actuators), resulting in a plethora of new applications in the domains of smart cities, healthcare, transportation, and so forth [1]. In order to create a truly smart environment, the goal of many IoT applications is to process these large amounts of sensor data in real-time, classify this information and come to a set of resulting actions to execute.

Neural networks lend themselves naturally to process large amounts of unstructured data and are commonly used for many classification problems [21] such as recognizing objects [15], traffic signs [6], speech [10] and handwritten digits [5], as well as for more complex tasks such as obstacle avoidance [4] or robot steering [19]. Similarly, neural networks can also be used for generic IoT applications, in its simplest form consisting of an input layer that takes various sensor data as input, one or more hidden layers and an output layer which maps to a number of concrete actions. For example setting the temperature of a thermostat, which depends on humidity, sunlight, room temperature, user preference, etc.

Currently, the Cloud is often the natural choice to train and evaluate neural networks, benefiting from the huge compute power and scalability [20]. However, for IoT applications with sensors sending a continuous stream of data, the Cloud introduces additional complications. First, a connection to the Cloud is required at all times, having to deal with limitations in bandwidth and a high and variable latency. Second, sending sensor data to the Cloud may introduce security holes and privacy issues. Therefore, our approach is to distribute neural networks on local compute power in the various devices at hand of the IoT ecosystem.

In this paper the DIANNE middleware framework is presented, that models, trains and evaluates neural networks distributed across multiple devices. The framework runs on a multitude of heterogeneous devices, ranging from small embedded devices such as the Raspberry Pi, up to enter-

prise server machines, as well as specialized GPU accelerated hardware such as the NVIDIA Jetson TK1. By splitting a neural network into separate functional components, parts of the neural network can be distributed among multiple devices at runtime, and one can overcome the memory and/or processing power limitations of the IoT devices at hand. While most frameworks are optimized for training with batches of samples, the DIANNE middleware is optimized for fast *execution* of *one input sample* at a time. This is important in an IoT context where the neural network continuously has to process incoming sensor data.

The remainder of this paper is organised as follows. The next section presents the related work in scope of distributed neural networks and current frameworks for building neural networks. Section 3 gives an overview of the modular approach of the DIANNE middleware. Section 4 and 5 expand on the design and implementation choices while Section 6 shows the feasibility of this framework by comparing with existing solutions. Section 7 concludes this paper and provides pointers for future work.

2. RELATED WORK

Current frameworks for building, training and executing neural networks mostly focus on running a neural network on a single machine, often with support for GPU acceleration.

The University of Montreal's Theano [2] compiler for mathematical expressions written in Python is often used for designing neural networks. The developer defines a neural network as a set of mathematical expressions written in a high-level description language, which are then optimized and translated into native C++ (or CUDA for GPU), and compiled into dynamic Python libraries and automatically loaded by the framework. "Lasagne" [11] further enhances Theano by creating an easy to use library of neural network layers to build and train neural networks.

The deep learning framework Caffe [12] written in C++ from UC Berkeley allows neural networks to be defined as plain text schemas instead of code. It also has interfaces to Python and Matlab.

NYU's Torch7 [7] is a scientific computing framework with wide support for machine learning algorithms and provides an easy to use interface via the LuaJIT scripting language. The Torch7 *nn* package provides modules for building neural networks, where each neural network layer can be composed of a number of Torch modules. This modular approach makes it easy to compose and build neural networks.

None of these frameworks support the distribution of a single neural network across multiple devices. Software distribution is only used to some extent to speed up the training phase. Krizhevsky et al. [16] showed how a larger neural network can be trained by spreading the net across two GPUs. In [13], the authors show that scaling up further to 8 GPUs can lead to a speed up factor of 6.16. Dean et al. [8] presented the DistBelief framework for parallel distributed training of deep neural networks. By adopting new training algorithms they can distribute the training procedure on a large number of CPU nodes, for example achieving a speed up of more than 12 using 81 machines.

For DIANNE, a similar modular approach as Torch7 was followed, treating a neural network as a composition of individual modules. However, in addition to Torch7, a neural network module in DIANNE is also a unit of deployment, allowing us to distribute modules of a single neural network across different devices. Like Caffe, DIANNE also has support for building neural networks in a declarative way, using a JSON format. This allows to easily create and import neural networks using a web UI builder.

3. DIANNE ARCHITECTURE

Typical feed-forward neural networks are composed of an input layer, one or more hidden layers and a single output layer. The output layer provides for example a vector classifying the input data. Such a vector has one value for each classification class, between 0 and 1, depicting the probability that the input can be classified as such.

In DIANNE, each neural network layer is represented by one or more modules. DIANNE modules are the basic building blocks of neural networks, which provide two flows of information: a forward pass and a backward pass. The forward pass, required for neural network execution, in which input data is transformed in some way to give an output. The backward pass, required for training neural networks, that takes in the gradient on the output of the previous forward pass and calculates the corresponding gradient on the input. Each module can have one (or more) *next* modules to forward its output to, and one (or more) *previous* modules to propagate the gradient on the input to.

A neural network can be constructed by chaining a number of modules. Starting with a special *Input* module, which simply forwards the input data, and ending with another special *Output* module, which collects the output. Besides the Input and the Output module, DIANNE supports a number of other types to build up neural networks. A *Lin-*

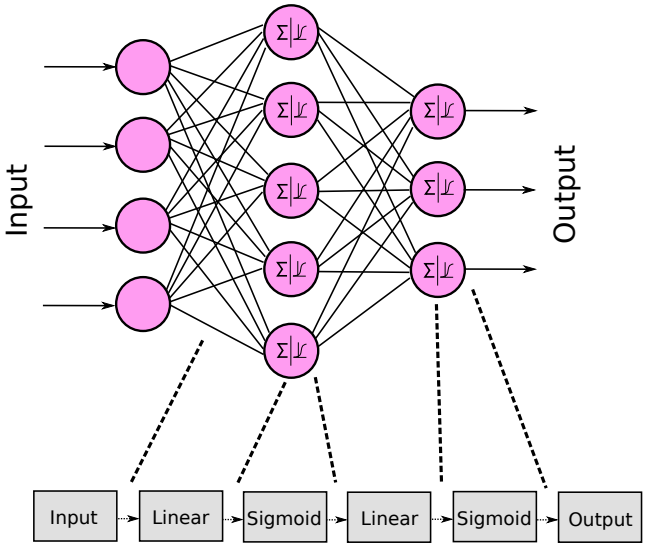

Figure 1: A feed-forward fully connected neural network example with a single hidden layer is split up as a chain of DIANNE modules.

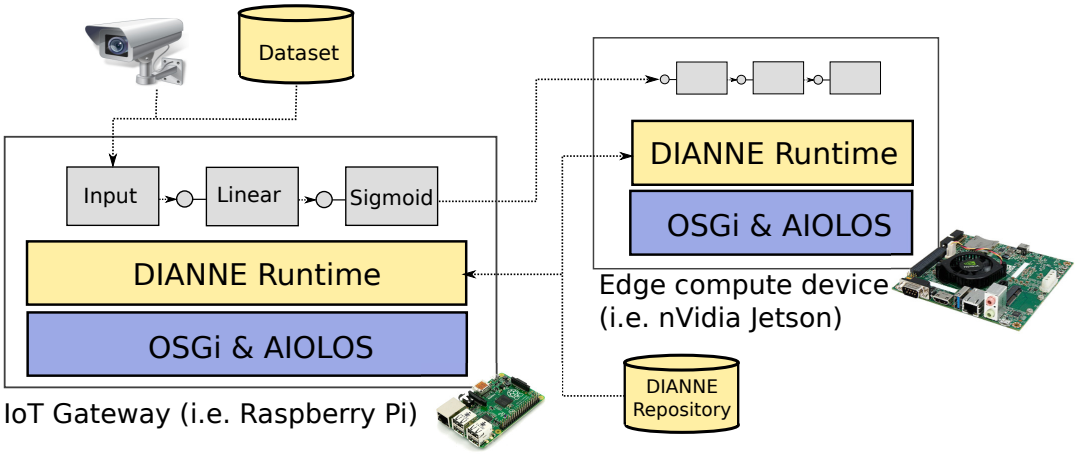

Figure 2: Each node runs the DIANNE runtime which is able to create and configure module instances from the centralized DIANNE Repository. This repository is an OSGi service which is deployed on an AIOLOS node. Datasets can be used to forward samples through the neural network and evaluate or train the network. AIOLOS and OSGi are used to enable distribution and remote calling of DIANNE modules.

ear module calculates a weighted sum of the input and can be combined with an activation module (currently *Sigmoid*, *Tanh* and (Parameterized) Rectified Linear Units (*(P)ReLU*) modules are supported) to model a fully connected neural network layer. For classification, a *Softmax* module is added before the Output which converts the output to classification probabilities. In Figure 1 we show how a sample fully connected neural network with one hidden layer is split up in DIANNE modules.

Besides modules for fully connected neural networks, we provide *Convolution* and *MaxPooling* modules which are frequently used in state-of-the-art convolutional neural networks. Various split and combine modules allow to create parallel paths in a neural network, which makes it more suitable for offloading parts of the network to other devices. These special split and combine modules are the only modules which can have multiple *next* and *prevous* modules. An example of such a distributed network is the cascade of neural network layers described in [18]. Finally, a number of preprocessing modules are provided, for example to normalize the input samples.

The DIANNE framework proposes a uniform method for defining modules with which neural networks can be composed. The framework facilitates the distribution of these modules, by letting the user choose where to deploy modules to. Both manual deployment, as well as automatic deployment using optimisation algorithms are supported.

4. DESIGN AND IMPLEMENTATION

An illustration of a DIANNE deployment is shown in Figure 2. As is already clear from Section 3, the central entity in DIANNE is the neural network module. Each device in the network is provided with the DIANNE runtime. The runtime is able to create new module instances from a module configuration description. Each module is uniquely identified by a 128 bit UUID and also states the identifiers of the *next* and *previous* modules in the neural network. A module instance is created based on the module type and a set of module-specific properties depending on that type. A neural network is then defined as a collection of modules that are interconnected. The JSON format is used to easily configure a neural network as shown in Listing 1.

```
{
  "name": "mnist",
  "modules": {
    "46a5b20b-...": {
      "id": "46a5b20b-...",
      "type": "Input",
      "next": "3753d189-..."
    },
    "3753d189-...": {
      "id": "3753d189-...",
      "type": "Linear",
      "next": "f124137f-...",
      "prev": "46a5b20b-...",
      "output": "20",
      "input": "784"
    },
    "f124137f-...": {
      "id": "f124137f-...",
      "type": "Sigmoid",
      "next": "b5ffb82d-...",
      "prev": "3753d189-..."
    },
    ...
}
```

Listing 1: The partial DIANNE neural network configuration file of the network shown in Figure 1. The modules' UUIDs are shortened for clarity.

These configurations can be stored together with previously trained parameters in the centralized DIANNE Repository, such that trained neural networks can be easily (re)deployed. Note that the configuration file contains duplicate information on the *next* and *previous* modules. This is done on purpose, as it allows to deploy a single module on a runtime without explicit knowledge of the complete neural network

structure. Of course this makes it a tedious task to create such a neural network configuration. Therefore, we also provide a web UI builder that reads or generates a JSON configuration as explained in Section 5.

To enable distributed module deployment this modular neural network approach is combined with the OSGi-based platform AIOLOS [23], which allows for transparently deploying software components on multiple devices. By implementing the DIANNE runtime as an OSGi bundle, and exposing the DIANNE modules as OSGi services, modules can be seamlessly deployed and redeployed to different devices. AIOLOS will discover and import remote module services that are required as *next* and/or *previous* modules of the current local DIANNE module, and make these available through remote method calls.

For the DIANNE modules, three different implementations are supported to account for the heterogeneity of devices to be deployed to. Besides our own pure Java implementation, there is also support for a native C implementation based on the Torch tensor library that uses BLAS, as well as a native GPU accelerated implementation using CUDA via JNI. This way, although the framework is OSGi and hence Java based, the middleware performance is on par with existing neural network frameworks. Consequently, depending on the device hardware and architecture, the most suitable native library is automatically loaded at runtime with fall back to the Java implementation if none apply.

Besides creating, deploying and executing a neural network, DIANNE also has limited support for training and evaluating a neural network. Commonly used datasets such as MNIST [17], CIFAR [14] and ImageNet [9] are made available as an OSGi service. Currently, only a basic stochastic gradient descent training algorithm is implemented, but more complex training techniques can be easily implemented using the API exposed by the DIANNE runtime.

5. DIANNE GUI

In order to facilitate the construction and configuration of neural networks, the DIANNE runtime is equipped with a web-based GUI as shown on Figure 3. In the *build* tab, all

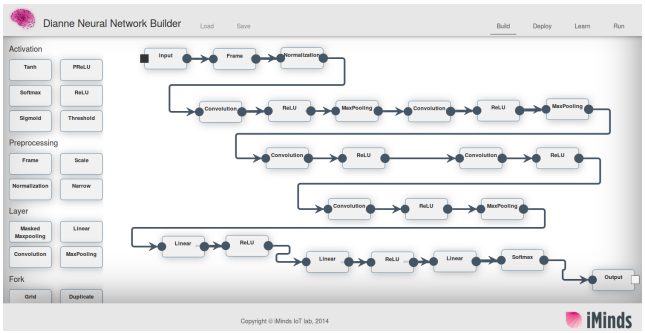

Figure 3: DIANNE GUI for constructing and configuring neural networks. On the left you can find all supported modules that can be dragged onto the canvas and connected to create a neural network.

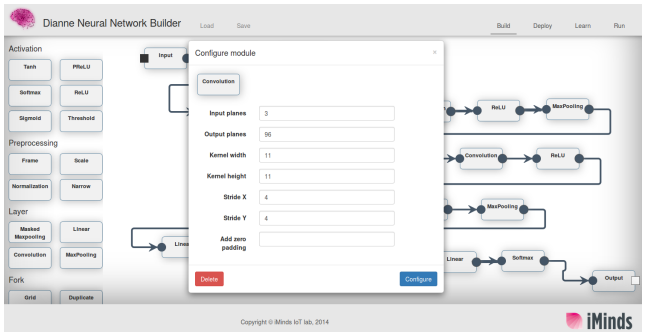

Figure 4: Double clicking on a module opens a configuration dialog to configure the module.

building blocks are available from a toolbox on the left of the screen and can be dragged and dropped onto the canvas and connected to other modules to form a neural network. Each module can be configured using a configuration dialog by double clicking on the module, as shown on Figure 4. The configuration can be saved to a JSON formatted string such as the one in Listing 1 and will be stored in the DIANNE repository for later reuse.

The UI has three more tabs: *deploy*, *learn* and *run*. In the *deploy* tab, all devices running the DIANNE runtime are listed and the user can manually select which device each module should be deployed to. In the *learn* tab the neural network can be trained by selecting one of the available datasets in combination with a suitable trainer (e.g. a basic stochastic gradient descent). The chosen dataset can also be dynamically split into a train, validation and test set to assess the accuracy of the neural network. Finally in the *run* tab the user can directly couple the neural network to actual input and output devices. For example, a camera can serve as the input for the network, while the output can be used to open a door.

6. EXPERIMENTAL RESULTS

Our experimental results were conducted on three devices shown in Table 1 connected with a wired gigabit LAN: a laptop without GPU support, a Jetson TK1 embedded device with a specialized low power GPU and a powerful server from iLab.t [3] equipped with a desktop grade GPU. Take into account that there is a huge difference between the Kepler GPU of the Jetson TK1, which has 192 CUDA cores, and the GTX 980 GPU of this server, which has 2048 CUDA cores with a higher base clock frequency. Each device runs the CUDA tensor library if possible, otherwise the native C library is used. The Java library was not used during these experiments. The pre-trained fast OverFeat [22] network was used, which classifies an image in 1000 categories with

Table 1: Hardware specifications.

name	arch.	CPU	GPU	RAM
Laptop	x86	i7 4500U	NA	4GB
Jetson TK1	ARM	Cortex-A15	Tegra K1	2GB
iLab.t [3]	x86	E5-2620v3	GTX 980	32GB

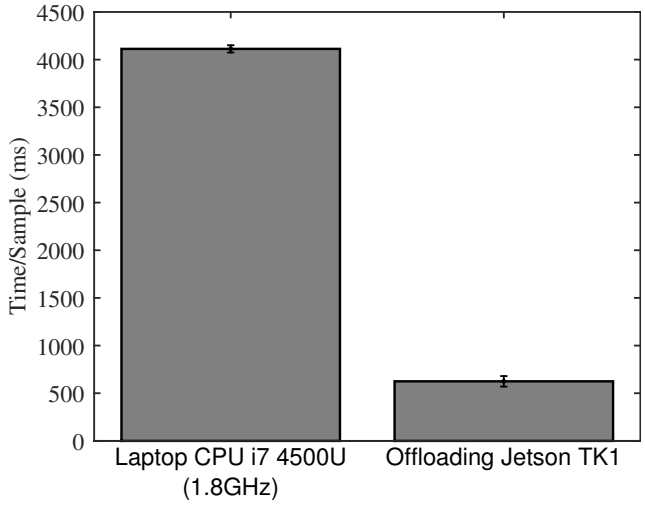

Figure 5: Fast OverFeat [22] network on two deployment setups. On the left deployed on a single laptop and on the right the network was partially deployed on the laptop and partially offloaded to the Jetson.

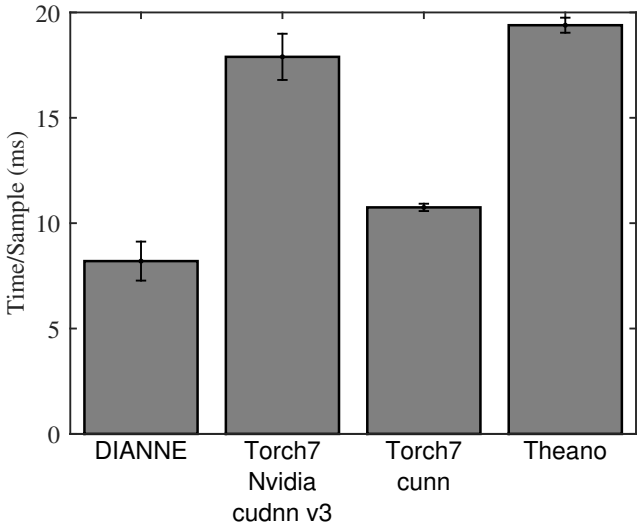

Figure 6: Comparing the single sample feed-forward time on the DIANNE middleware and popular deep learning frameworks. All test were conducted on the iLab.t server with a GTX 980 GPU (Table 1).

a classification error of 16.39% on the ImageNet test sample set. The accurate OverFeat model reaches a classification error of 14.18% on the same test sample set, but it needs nearly twice as many connections which results in a longer execution time.

In a first experiment we compare two deployments of the fast Overfeat neural network: (1) the complete neural network deployed on the laptop and (2) the first 5 of the 8 layers offloaded to the Jetson TK1. Executing OverFeat entirely on a single Jetson is impossible because the GPU memory is too limited to fit the complete neural network. In each scenario samples were randomly selected from the ImageNet dataset and were one by one executed on the neural network. The average execution times can be seen in Figure 5. Offloading convolutional modules to a low power GPU device lowers the execution time drastically even if the modules need to communicate over the network. Also note that the biggest share of the time in the offloading scenario is due to the part that is still executed on the laptop, so additional speedup can be achieved when having multiple local GPU devices.

In order to compare DIANNE to the other popular frameworks depicted in Section 2, we conducted an experiment on the powerful iLab.t server [3] (see Table 1 for specifications). During this test we used the same OverFeat network with ImageNet samples. The results of Figure 6 show that DIANNE actually performs on par or better than the other frameworks, while Torch7 with *cudnn* and Theano perform roughly the same. This is expected since Torch7 with *cudnn* and Theano both use version three of the *NVIDIA cudnn* CUDA implementation, while the DIANNE CUDA code is largely based on *cunn*, the standard CUDA back-end from Torch7. Additionally, the other frameworks are optimized for processing samples in batch in the training phase, instead of a single sample feed-forward.

The first experiment's results prove that large neural networks, which can not fit on small embedded devices, can benefit from distributing the slow convolutional modules to other devices in the IoT environment preferable equipped with GPU acceleration. The second experiment shows that the DIANNE middleware performs excellently on GPU accelerated devices, outperforming all tested frameworks when only a single image is forwarded through the network.

In the future more embedded devices will incorporate low power cost efficient GPU chips, like the Kepler GPU in the Jetson TK1. While this would improve the local execution time of neural networks they still have far less available memory capacity than their desktop variants, which limits the neural network size. By using DIANNE middleware we can distribute large neural networks on a range of devices without the need for costly infrastructure. Even with powerful cloud infrastructure there is a hard limit to the size of a neural network on a single device. When we move to datasets with more input parameters from all kinds of IoT devices the distribution of neural networks will be a must.

7. CONCLUSION AND FUTURE WORK

In this paper the DIANNE framework is introduced, which facilitates easily modelling neural networks and deploy them across multiple devices supporting the DIANNE runtime. The framework has the following features: (i) model neural networks as a set of interconnected, modular and configurable components, (ii) discover all targetable devices with a DIANNE runtime, (iii) deploy all or specific modules to these devices, (iv) connect input data (e.g. camera, dataset samples, etc.) and execute the neural network, (v) load or save trained networks from the JSON-format. As an illustration the pre-trained fast OverFeat neural network was loaded and deployed on multiple devices, which allows for classification of ImageNet samples to 1000 classification classes.

Future work includes designing algorithms to let the framework automatically select and deploy the neural network modules to the pool of devices according to various metrics (such as minimal neural network output latency or specific interconnection link bandwidth minimization). This automatic selection could take into account the available CPU/GPU compute power, memory limitations and network speeds, or even be more dynamic to support on the fly reorganisation of the neural network when devices connect or disconnect from the IoT environment.

8. ACKNOWLEDGMENTS

Part of the work was supported by the iMinds IoT research program. Steven Bohez is funded by Ph.D. grant of the Agency for Innovation by Science and Technology in Flanders (IWT). We would also like to acknowledge NVIDIA for providing us with the Jetson TK1 board.

9. ADDITIONAL AUTHORS

Bart Dhoedt and Piet Demeester (Ghent University - iMinds).

10. REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010.

[2] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[3] S. Bouckaert, P. Becue, B. Vermeulen, B. Jooris, I. Moerman, and P. Demeester. Federating wired and wireless test facilities through Emulab and OMF: the iLab.t use case. In *8th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, Proceedings*, pages 1–16. Ghent University, Department of Information technology, 2012.

[4] K.-H. Chi and M. Lee. Obstacle avoidance in mobile robot using neural network. In *Consumer Electronics, Communications and Networks (CECNet), 2011 International Conference on*, pages 5082–5085, 2011.

[5] D. Ciresan, U. Meier, L. Gambardella, and J. Schmidhuber. Convolutional neural network committees for handwritten character classification. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 1135–1139, 2011.

[6] D. Ciresan, U. Meier, J. Masci, and J. Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333 – 338, 2012.

[7] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[8] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[10] L. Deng, G. Hinton, and B. Kingsbury. New types of deep neural network learning for speech recognition and related applications: an overview. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8599–8603, May 2013.

[11] S. Dieleman and Others. Lasagne. https://github.com/Lasagne/Lasagne.

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, and Others. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[13] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[14] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 2009.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[17] Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits, 1998.

[18] S. Leroux, S. Bohez, T. Verbelen, B. Vankeirsbilck, P. Simoens, and B. Dhoedt. Resource-constrained classification using a cascade of neural network layers. In *International Joint Conference on Neural Networks*, 2015.

[19] Z. Miljković, M. Mitić, M. Lazarević, and B. Babić. Neural network reinforcement learning for visual control of robot manipulators. *Expert Systems with Applications*, 40(5):1721 – 1736, 2013.

[20] P. Parwekar. From internet of things towards cloud of things. In *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*, pages 329–333, Sept 2011.

[21] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, January 2015.

[22] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. OverFeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[23] T. Verbelen, P. Simoens, F. D. Turck, and B. Dhoedt. Aiolos: Middleware for improving mobile application performance through cyber foraging. *Journal of Systems and Software*, 85(11):2629 – 2639, 2012.