DIANNE: a modular framework for designing, training and deploying deep neural networks on heterogeneous distributed infrastructure.

Elias De Coninck[a,*], Steven Bohez[a], Sam Leroux[a], Tim Verbelen[a], Bert Vankeirsbilck[a], Pieter Simoens[a], Bart Dhoedt[a]

[a] *Ghent University - imec, IDLab, Department of Information Technology, Technologiepark-Zwijnaarde 15, B-9052 Ghent, Belgium*

Abstract

Deep learning has shown tremendous results on various machine learning tasks, but the nature of the problems being tackled and the size of state-of-the-art deep neural networks often require training and deploying models on distributed infrastructure. DIANNE is a modular framework designed for dynamic (re)distribution of deep learning models and procedures. Besides providing elementary network building blocks as well as various training and evaluation routines, DIANNE focuses on dynamic deployment on heterogeneous distributed infrastructure, abstraction of Internet of Things (IoT) sensors, integration with external systems and graphical user interfaces to build and deploy networks, while retaining the performance of similar deep learning frameworks.

In this paper the DIANNE framework is proposed as an all-in-one solution for deep learning, enabling data and model parallelism though a modular design, offloading to local compute power, and the ability to abstract between simulation and real environment.

Keywords: Artificial Neural Networks, Distributed Applications, Machine Learning, Internet of Things

*Corresponding author

Email addresses: elias.deconinck@ugent.be (Elias De Coninck), steven.bohez@ugent.be (Steven Bohez), sam.leroux@ugent.be (Sam Leroux), tim.verbelen@ugent.be (Tim Verbelen), bert.vankeirsbilck@ugent.be (Bert Vankeirsbilck), pieter.simoens@ugent.be (Pieter Simoens), bart.dhoedt@ugent.be (Bart Dhoedt)

1. Introduction

In recent years, deep learning [1, 2] has revolutionized many areas of computer science, from computer vision and natural language processing to control systems. As state-of-the-art neural networks generally have millions of parameters, training such models requires an equally large amount of training data and computation time. Distributing the training procedure on a cluster of compute nodes has become a necessity to achieve feasible training times. Recent work has studied the trade-offs of various algorithms for distributed training [3, 4, 5]. Most deep learning frameworks, however, do not provide distributed training out-of-the-box, or do not do so in a user-friendly and intuitive way, requiring additional development. This is especially the case for distributing custom training routines.

In practical applications of deep learning such as in robotics, often high-dimensional sensor data from multiple sources needs to be combined and processed, making it impossible to avoid a distributed setting. In the Internet of Things (IoT) world there exist a plethora of networked devices, sensors or actuators, with or without compute capabilities which could be used in deep learning as inputs (sensors), outputs (actuators), training capacity or training performance. In such cases, combining deep learning with principles from edge computing to provide on-demand acceleration becomes appealing.

In this paper we present DIANNE (DIstributed Artificial Neural NEtworks), a modular deep learning framework implemented in Java, built from the ground up with a clear focus on dynamic distributed training and deployment of deep neural networks on a multitude of heterogeneous devices. The targeted devices range from small embedded devices, like the Raspberry Pi, up to cloud infrastructure with specialized learning hardware. While most frameworks focus on batch learning speed, DIANNE is optimized for ease-of use during application development and interaction with robots and IoT environments through input/output abstraction.

A getting-started guide and documentation are available on the DIANNE webpage [1]. Source code is hosted on GitHub [2] under a GPLv3 License. Additionally, automatically tested and compiled binaries for various platforms

[1] `http://dianne.intec.ugent.be`

[2] `http://github.com/ibcn-cloudlet/dianne`

are available on the download page of the DIANNE website, as well as a number of pre-trained models.

The remainder of this paper is structured as follows. In the next section we discuss the related work in scope of scaling and managing deep neural network learning, and other comparable frameworks. Section 3 gives more detail on the requirements of the framework by introducing common use cases, which are later expanded upon in the experiments in Section 6. Section 4 describes the design details of the architecture mapping the components to the requirements, while Section 5 gives the implementation details required to understand how we conducted our experiments in Section 6.

2. Related work

Many deep learning frameworks exist, but are mainly targeted to data scientists, who prefer quick prototyping in scripting languages such as Python or Lua. Google's Tensorflow [6] is perhaps the best known framework available. It has a substantial community, several programming interfaces amongst which Python, C++ and Java, although the Python interface is mostly used. Tensorflow also supports distribution on multiple devices through their Borg cluster management system [7]. However, this requires all operations to be defined in the so called Tensorflow graph, which becomes tedious for non-trivial use cases.

Other frameworks like Caffe [8] and Theano [9] offer a Python interface and focus on single device deployments with highly optimized CPU and GPU code. Similarly, Torch7 [10] was developed in Luascript, but recently evolved into PyTorch, again written in Python, with a focus on creating neural networks in a dynamic and modular way. Deeplearning4j (DL4J) [11] is a Java based framework, which can use Apache Spark for distributed processing.

DIANNE follows more the Torch7/PyTorch model, where a neural network is composed of modules that are deployed on the fly at runtime. Also, each module can be easily deployed on any device in the network, providing a fine grained distribution mechanism.

With the ever increasing depth and size of neural networks there is a definite need for alternative ways to distribute training and evaluation of neural networks. Krizhevsky et al. [12] presented work on model parallelism splitting up a large neural network across two GPUs with shared memory, limiting the communication overhead. In [13], they further enhanced this work spreading the network across 8 GPUs leading to a factor 6.16 boost in

learning speed. Dean et al. [14] presented the DistBelief framework which focuses on model parallelism on CPU hardware. They introduced new large scale training algorithms using Downpour SGD, a highly asynchronous variant of SGD, which was able to train modestly sized networks significantly faster on 2000 CPU cores than on a single GPU, removing the GPUs' upper limit of the network size.

In Alsheikh et al. [15] the focus shifts to Mobile Big Data (MBD) from smartphones and IoT gadgets. They propose a scalable learning framework that leverages Apache Spark, where each Spark worker is responsible for learning, through an iterative MapReduce process, a partial deep model on a portion of the overall MBD. The entire deep model is later reconstructed by averaging the parameters of all trained partial models.

In order to support such advanced flavors of both model and data parallelism, DIANNE is designed as loosely coupled components that can each be deployed on a variety of heterogeneous devices and can be scaled at will. In addition, DIANNE also offers components for the increasingly important field of reinforcement learning, integrates well with external systems and provides web-based user interfaces for both creating and monitoring neural networks.

3. Requirements

The goal of the proposed framework is to facilitate the integration of deep learning algorithms into external systems, deploy them on a wide variety of devices and to support a broad range of applications. In order to define requirements of our framework, we first discuss three main scenarios in which deep learning can be applied, and list the specific set of requirements for each case.

3.1. Supervised learning

The first scenario is the most common one for a deep learning framework: train a neural network model for a classification or regression task. The input of a neural network can be anything, such as raw image data, text, sensor input, etc. In the case of classification, the output of the neural network resembles the class, often encoded as a one-hot vector. In the case of regression, the output can again be any type.

In supervised learning, the model is fitted to learn input-output behavior from a labeled dataset. The model is trained by forwarding a batch of data from the dataset through the neural network, evaluate a certain loss function

on the output, and updating the model weights by calculating the gradients of the weights to decrease the loss, so called stochastic gradient descent (SGD).

Such a dataset typically contains a large number of input samples with corresponding labels, and is fixed in size. When training a model, the dataset is often split into three parts: a large train set, on which the model is trained, a small validation set that is used to evaluate the model performance during training, and a small test set that is only used to evaluate the performance of the final, fully trained model. The accuracy on the validation set during training can for example be interpreted to avoid overfitting to the training data, i.e. when the accuracy on the validation set starts to diminish. Hence our framework should at least be able to import data from a dataset, feed it into a neural network and update the neural network weights according to a certain loss function. The framework also has to allow for various splits of the datasets, and evaluate the model on both the validation set (during training) and the test set.

A widely used dataset is ImageNet [16], which is a manually annotated dataset containing more than ten million images labelled into 1000 categories. This dataset is used in the annual research contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [17], which is a well known benchmark for object detection and recognition. In order to achieve state of the art performance on such a task, one needs to design a model that is sufficiently deep (Inception-v4 [18]) often more than 100 layers and has millions of parameters to train. Although the calculations can be sped up by benefiting from highly optimized operation implementations on GPU hardware, one is still pushing the limits of the capabilities of current GPU hardware, both in time and memory. By enabling training across multiple nodes we can further reduce this required training time. 'Data parallelism' and 'model parallelism' are two methods of distributing an algorithm on multiple nodes. Data parallelism is realized by sending batches of samples to different nodes and aggregating the results into an overall estimation over all batches [19]. Model parallelism is achieved by cutting the neural network into multiple subnetworks which each can be distributed across multiple nodes (AlexNet [12]).

Also, the accuracy of the model is determined by a huge number of hyperparameters, both in terms of the neural network architecture (how many layers, their size, ...) as the training algorithm (learning rate, batch size, ...). In order to find the best model, often a hyperparameter search is done by training and evaluating many models in parallel. Therefore, a key requirement is to support running many experiments in parallel on a cluster

infrastructure.

In summary the framework must meet the following requirements to support supervised learning experiments:

1.1 Fetch samples from a labelled dataset
1.2 Model evaluation on validation and test set
1.3 Scaling out using model and data parallelism
1.4 Manage experiments: jobs, resources and results
1.5 Save, update and restore trained parameters
1.6 Custom definition of supervised learning routines

3.2. Distributed reinforcement learning

An increasingly important application of neural networks is in reinforcement learning (RL) [20]. Unlike supervised learning, in reinforcement learning the goal is to design an agent that interacts with a certain environment. Based on the agent interactions, it receives rewards, which it has to maximize. In this case, deep neural networks can be used as function approximators, which can either represent a policy π to map observations to actions directly, or learn an action-value function $Q(s, a)$, which represents the expected discounted return or Q-value for taking a specific action a in the current state s and following policy π afterwards. Recently reinforcement learning has gained popularity for training robotic tasks from raw sensor input such as locomotion [21], grasping [22], manipulation [23], autonomous driving [24, 25], etc.

A reinforcement learning problem consists of an agent and an environment. The agent interacts with the environment through actions and can receive a partial or full state observation s_t from the environment. For each action a_t taken by the agent it receives a reward r_t, and results in the environment transitioning to state s_{t+1}. The goal of the agent is to take a sequence of actions that maximizes the future reward over time. An important class of reinforcement learning algorithms store each (s_t, a_t, r_t, s_{t+1}) tuple in a so called experience pool, which is used to train a neural network to optimize the policy. When the experience pool is full, it usually replaces the oldest samples from the pool such as a FIFO queue.

An environment can represent many different systems, both physical or simulated. Training directly on a physical environment can lead to problems, e.g. servos can brake down, safety concerns, uncontrolled/unwanted external inputs. To address these issues, one often starts training in simulation, and

later transfers the learned model to the physical system [26]. Therefore, the framework should enable to easily switch between both simulated and physical environments.

Also, simulating a complex environment is often very resource-intensive, for example learning robotics tasks requires realistic physics simulation. This limits the rate at which experience samples can be generated. Hence, we need to be able to easily scale up the number of environments and agents that can act in parallel. Supporting this scenario adds the following requirements to the framework:

2.1 Easy integration of environments
2.2 Scaling agents/environments
2.3 Collect, store and sample from experience
2.4 Custom definition for agent interactions
2.5 Custom definition of reinforcement learning routine

3.3. Neural networks in IoT applications

The number of IoT devices is predicted to sky rocket in the coming years with estimations of 50 billion devices by 2020 (not including smartphones). Also here deep neural networks can be exploited to process the data coming from these devices, such as high-dimensional anomaly detection [27], human activity recognition through wearables [28, 29], home automation [30],etc. An abstraction of these heterogeneous devices (e.g. camera, lidar, temperature sensor, etc.) is required to create a bridge between neural networks and things.

The major problem these devices bring is that they have limited resource capabilities to handle large neural networks, e.g. Raspberry Pi, Nvidia Jetson, Intel Edison, smartphone or other embedded devices with limited memory, constrained compute power and/or battery operated. The available RAM in a CPU or GPU also limits the number of weights that can be deployed on a single device. The solution is using sequential model splitting, splitting a neural network into smaller subnetworks, and distributing these subnetworks across multiple devices. Moreover, offloading certain parts of a network to a more suited device can increase the overall performance.

Other use case examples are: (1) The input comes from multiple devices already preprocessing their inputs to later combine on an edge device with specialized hardware. The inputs processing is therefore done in parallel and should increase the performance. (2) Dynamically offload neural networks

(partially or completely) when specialized hardware comes available in the environment to increase performance or reduce battery consumption with wireless devices (e.g. smartphones). This adds the last requirements of the framework:

3.1 Deploy on resource restrictive hardware for inference
3.2 Distribute part of neural network to optimize performance
3.3 Inputs and outputs abstraction to connect to IoT sensors

4. DIANNE, modular framework for distributed deep learning

This section elaborates on the design choices, made for the DIANNE modular framework for distributed deep learning, that constitute the flexible architecture that enables developers to create a broad range of applications while ensuring the requirements derived from the use cases described in Section 3 are fully met. An overview is given in Table 1 at the end of Section 4.1.

4.1. System component model

The DIANNE framework adopts a microservice architecture, consisting of a number of loosely coupled components that communicate through well defined services. Depending on the concrete scenario, a different set of components can be deployed based on the requirements. Also, multiple instances of each component can be deployed in a system in order to offer various ways of parallelism. Sections 4.1.1 to 4.1.10 introduce each component in relation to the requirements presented in Section 3.

4.1.1. Neural network modules

In DIANNE, a neural network layer is treated as a directed graph of neural network modules. Each *Module* provides a $forward$ method which

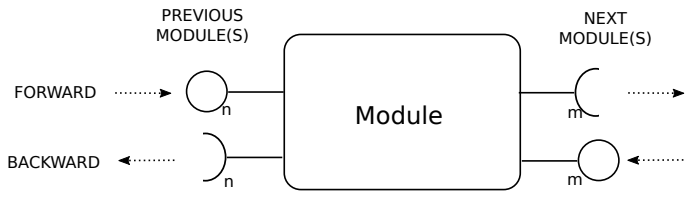
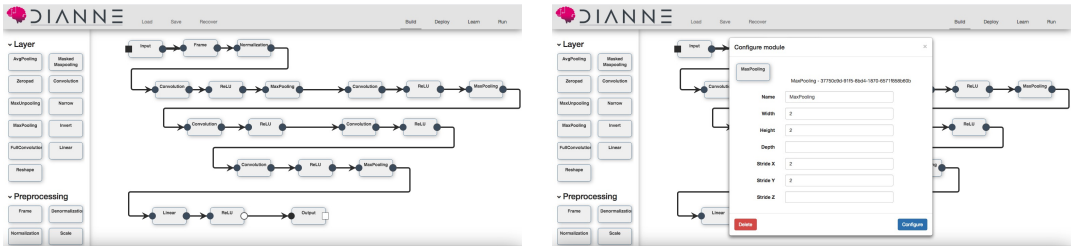


Figure 1: A DIANNE *Module* has references to its predecessors and successors in the graph for forwarding input and backpropagating gradient information.



(a) (b)

Figure 2: A small convolutional neural network build using the drag-and-drop web interface. The *Module*'s dimensions, strides, padding, etc. are configured through a pop-up dialog.

represents an operation transforming the input into an output. Each *Module* also provides a *backward* method which enables to back propagate gradients in the network during training. Hence a *Module* has references to its predecessors and successors in the graph for the forward and backward pass as shown in Figure 1. Besides a functional unit, a neural network module in DIANNE is also a unit of deployment enabling model distribution (Requirement 1.3) on multiple DIANNE *Runtime*s (Section 4.1.2). The framework supports a wide range of *Module*s, such as fully-connected layer, convolutional layer, activation functions, batch normalization, input pre-processing units, etc. A special *Module* is the memory *Module*, which stores its input for one timestep, allowing to create recurrent neural networks. One can also add his own *Module*s, or define aggregate *Module*s by combining the low level ones.

A neural network can be defined programmatically using a builder pattern API, and using a visual drag-and-drop web user interface as shown on Figure 2, which are both serialized to a JSON format. Each *Module* is uniquely identified by an id, and can have one or more next and previous *Module* ids listed, to which data has to be forwarded or back propagated. Dedicated *Input* and *Output* modules provide the entry and exit points of the network, through which input data resp. output gradients can be passed through the model.

4.1.2. Runtime

The DIANNE *Runtime* is responsible for actually deploying the neural network modules on a device, allocating memory for the weights and provides

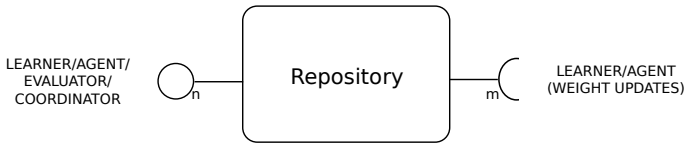


Figure 3: The *Repository* handles persistence of neural network descriptions and *Module* weights. *Learner*s and *Agent*s can also get callbacks whenever weights are updated.

implementations for the operations. To this end, the *Runtime* defines a single native interface that should be implemented in order for all current modules to be deployed. By providing implementations of this interface for various native platforms, we can run DIANNE on a wide variety of heterogeneous devices (Requirement 3.1 and 3.2).

The *Runtime* also binds each *Module* to its specified next and previous *Module*s, and asynchronously propagate activations and gradients between *Input*s and *Output*s. For each neural network it also provides a separate *NeuralNetwork* service, which offers a nice, asynchronous API that can be used by external applications to feed data into the model (Requirement 3.3).

4.1.3. Repository

The *Repository* handles the persistence of neural network descriptions and their weights. Every neural network description specified is stored in the JSON format, and for each *Module* weights can be stored by their unique id. Weights can also be tagged with user defined tags, which enables to store and retrieve multiple versions of weights for the same neural network. For example, the *Learner*s (Section 4.1.6) can tag the weights during training time with the number of epochs passed, or tag the network weights that achieve the best accuracy on a validation set. These features satisfy Requirement 1.5.

During training, learners publish a delta on the weights to the *Repository*, which adds up this delta. When sharing a single *Repository* between multiple learners, this can introduce data parallelism by merging weight updates from multiple learners (Requirement 1.3). For scalability, one could also deploy multiple *Repositories*, where each is responsible for gathering weight updates for different *Module*s.

By saving the weights by *Module* id instead of the complete neural network, we can also easily share *Module*s and their weights between multiple neural networks. This is an important feature for weight sharing between networks [31] or transfer learning [26].

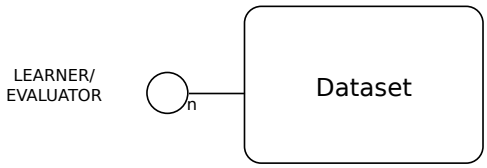


Figure 4: A *Dataset* provides access to samples and their corresponding labels (if applicable) to the *Learner*s and *Evaluator*s.

*Learner*s, *Agent*s, *Evaluator*s and the *Coordinator* access the *Repository* to fetch neural network definitions and their weights as shown in Figure 3. *Learner*s and *Agent*s can also get callbacks whenever weights got updated in the *Repository*.

4.1.4. Datasets

Data is provided by the *Dataset* component, which provides methods for querying samples and their corresponding labels (if any) to the *Learner*s and *Evaluator*s (Figure 4). DIANNE supports various popular image datasets out of the box, such as ImageNet [12], MNIST [32], CIFAR-10/100 [33], etc., and adding new datasets is as trivial as implementing a single method to fetch a new sample.

Requirement 1.1 and Requirement 1.2 are accomplished by this component and the *Learner* (Section 4.1.6) component. During training learners query the *Dataset* to fetch new samples, stored in memory or on disk, and forward the sample through the deployed network. In the end the sample's label can be compared with the output of the network.

Datasets can also be decorated by so called dataset adapters, which for example can split the dataset into a train, validation and test set. . The adapters can also adapt the samples on the fly as they are queried from the dataset. This mechanism can for example be used for online data augmentation, where random crops and rotations are provided from the original data sample.

In case the underlying data has a time dimension, the samples are stored in a *SequenceDataset*, which extends the *Dataset* to keep the order of the samples. In this case, a complete sequence of samples can be fetched from the dataset.

There are three ways to access samples from datasets, each with different memory requirements and access times. (1) Completely loading in all data from a dataset into memory, making access as fast as possible with a high

Figure 5: Like a *Dataset*, an *Experience pool* provides access to experience samples to *Learner*s and *Evaluator*s, and provides an additional interface towards *Agent*s to add new experience samples.

memory demand. (2) Read in the data into a memory mapped file, slowing down random access but allowing for loading in larger datasets without increasing the memory requirement. (3) Managing the dataset in a folder with a certain file structure allowing to load in samples as needed directly from file. This slows down learning significantly but the dataset's size limit is increased to the available disk space. Pre-fetching samples asynchronously can help lower the impact of file access time.

4.1.5. Experience pool

In case of reinforcement learning, a fixed dataset does not suffice, as new experience will be generated over time (Requirement 2.3). The *Experience pool* component extends the *SequenceDataset* with extra information per sample. Each sample, uploaded by the *Agent*, has a state s_t, action a_t, next state s_{t+1}, reward r and a terminal indicator τ. Also, agents in the system can add additional samples to the *Experience pool* (Figure 5). Each *Experience pool* has a configurable maximum size, and by default the pool uses a First-In First-Out (FIFO) sample replacement strategy.

4.1.6. Learner

The learning routines (Requirement 1.6 and 2.5) are performed by the *Learner* components, which take one (or more) neural network(s) and a *Dataset* or *Experience pool*, and start updating the weights of the neural network as shown in Figure 6. The learning is initiated by the coordinator, which provides a configuration dictionary with parameters like the learning rate, batch size, noise, etc., and a *LearningStrategy* object that defines the actual learning procedure to use. While a learner will take care of tasks such as synchronizing with the repository and monitoring the stop criteria, the *LearningStrategy* evaluates the loss function and calculates the weight updates. The default *LearningStrategy* supports supervised learning, but

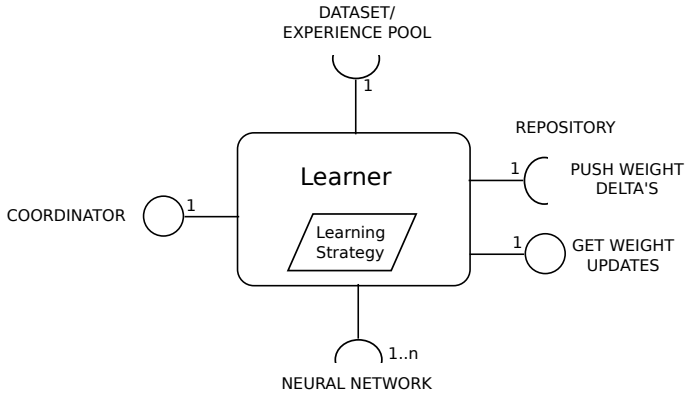


Figure 6: The *Learner* is triggered by the *Coordinator* and trains a neural network using data from a *Dataset* or an *Experience pool*. The delta's of the weight updates are pushed to the *Repository*, and the learner is notified of other weight updates (i.e. in case multiple *Learner*s are training the same network in parallel).

out of the box we also provide *LearningStrategies* for training more complex models such as Variational Autoencoders [34] (VAE), Generative Adverserial models [35] (GAN) and various reinforcement learning procedures.

The *Learner* component can easily be scaled up in order to achieve (Requirement 1.2 and 1.3) data parallelism, which can speed up the training process. Multiple *Learner*s can work in parallel asynchronously, and sync their weights at a configurable interval with the central repository by sending their gradient update and fetching the latest weights. A higher sync interval reduces the communication overhead at the risk of multiple learners diverging.

4.1.7. Evaluator

Evaluating the performance of a network is done using a separate *Evaluator* component and an *EvaluationStrategy*, which e.g. report the mean error, confusion matrix and the evaluation time of a neural network on a certain dataset (Figure 7). The *Evaluator* is most often used after training to evaluate the model performance on a test set, or during training to asses the model performance on a validation set to counter overfitting.

4.1.8. Agent

In the reinforcement learning setting, a neural network is used by an *Agent* in order to interact with an *Environment* according to a certain policy (Figure 8). The internals of the policy is implemented in the *ActionStrategy*.

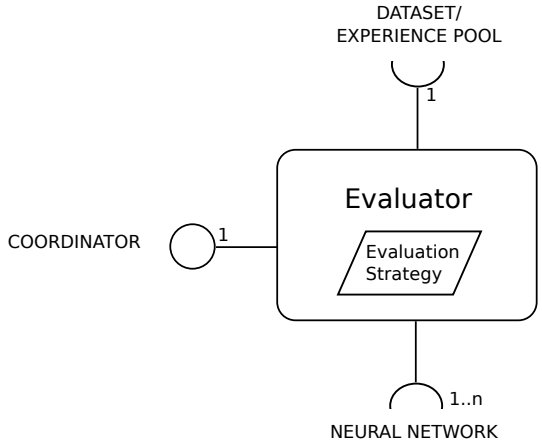


Figure 7: The *Evaluator* evaluates a neural network performance on a certain *Dataset* according to the provided *EvaluationStrategy*.

The *Agent* follows a strict interaction loop with the *Environment*: (1) reset to initial or random state s_0, (2) periodically update the policy π, (3) fetch observation s_t from the *Environment*, (4) execute action a_t based on policy π (Requirement 2.4), (5) receive reward r_t, next state s_{t+1} and terminal indicator τ, (6) collect samples $(s_t, a_t, r, s_{t+1}, \tau)$ and send sequence of samples to *Experience pool*.

This interaction loop of the *Agent* is sequential in nature and is the most prominent bottleneck in a reinforcement learning problem. Such problems

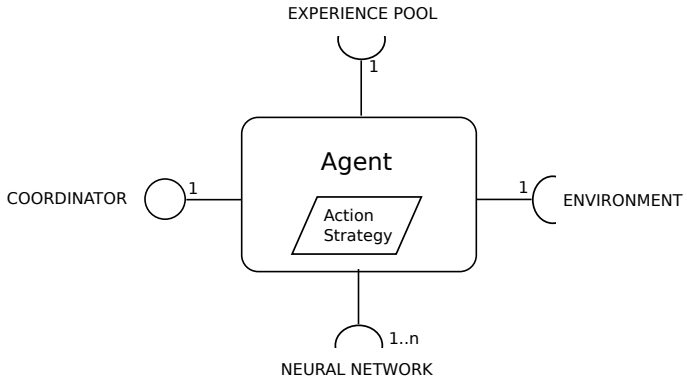


Figure 8: An *Agent* interacts with an *Environment*, taking actions based on the output of a neural network according to a given *ActionStrategy*. The generated state-action samples are uploaded to an *Experience pool*.

require lots of data to train the policy to realize a desired behavior. But most realistic simulation models are slow and require high-end computers to run. To overcome this problem the user is able to deploy multiple *Agent*s and *Environment*s on multiple devices increasing the sample rate generation.

4.1.9. Environment

An *Environment* component runs a simulation of the environment or functions as a proxy for an external environment (e.g. a physical robot). The *Environment* provides two main methods, one to fetch the current state of the environment, and one to execute a certain action on the environment, which returns a scalar reward.

At the time of writing we support the following reinforcement learning environments:

Arcade Learning Environment [36] (ALE) is collection of 2600 Atari games where the score is extracted as the reward function of these games. It is possible to train on pixel data, RAM data, or higher level observations.

OpenAI-Gym [37], a collection of reinforcement learning environments such as board games, classic control problems or physics based environments based on the Mujoco or Box2D.

KuKa Youbot A custom environment to interact with a physical Kuka Youbot robot or the simulated version in V-Rep simulator [38].

Erle Rover A custom environment to interact with a physical Erle Rover robot or the simulated version in the Gazebo simulator [39].

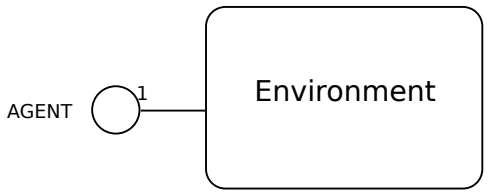
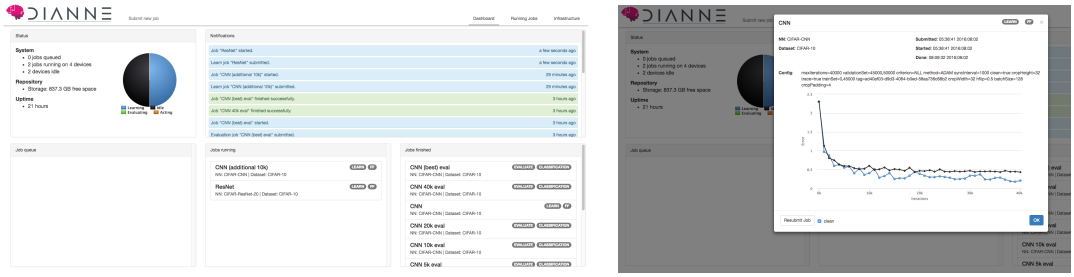


Figure 9: The *Environment* provides access to the state of the system, and allows an agent to interact with it.



(a) (b)

Figure 10: The DIANNE dashboard view (a) gives an overview of all queued, running and finished jobs. One can monitor job progress and inspect results in the job view (b).

4.1.10. Coordinator

The DIANNE *Coordinator* component coordinates all *Runtime* components deployed in the network. It manages and keeps track of all *Learner*s, *Agent*s and *Evaluator*s, making it possible to add additional learning, evaluating and acting runtimes on the fly.

It also acts as a training job scheduler, assigning incoming jobs to the different *Learner*s, *Evaluator*s and *Agent*s available. If all resources are in use the job scheduler queues all incoming jobs until more resources are deployed or a job of the same type finishes. The final returned weights and other results are saved to disk for later reuse or inspection.

Using the *Coordinator*'s Java API the user is able to write an extra component, which generates a number of jobs with custom parameters and a stop criteria to perform a parameter sweep easing the search for optimized hyperparameters (e.g. grid search, random search, Bayesian search, etc.).

Besides submitting jobs through the Java API or the Command-Line Interface (CLI), a separate JSON-RPC interface is available, allowing submission from external systems. Additionally, a web UI, using the JSON-RPC interface, is also available for submitting jobs and monitoring job progress, as shown on Figure 10. Custom *Strategy* implementations can even be uploaded using the UI, and are dynamically loaded on the required runtimes. This allows one to easily test out new routines without needing to take down the DIANNE cluster.

Submitting a job though one of the supported APIs requires information about this job: 1. A name for easy reference. 2. One of the supported job types. 3. The name of the network, designed in the builder, that needs

to be deployed for the job. 4. The entire dataset (available as a service). 5. Key-value pair options which are different for each job type. At the time of writing the *Coordinator* supports three job types:

Learn Used for training models. This requires at least one idle *Learner* service (Section 4.1.6) to be available in the cluster.

Evaluate Used for evaluating trained models. This requires at least one idle *Evaluator* service(Section 4.1.7). A Learn job can be configured in combination with an Evaluate job to evaluate the model on a validation set (range configuration on Learn job) to prevent overfitting to the training data.

Act Used for reinforcement learning. This requires an idle *Agent* service (Section 4.1.8) to interact with an environment to generate experience. While collecting new experience the deployed model can synchronize the model's weights at preconfigured intervals with the *Repository*.

Table 1: Requirements to components mapping.

Requirements	Components	Modules	Runtime	Repository	Dataset	Experience pool	Learner	Evaluator	Agent	Environment	Coordinator
1.1Labelled dataset samples					✓						
1.2Model evaluation					✓						
1.3Model and data Parallelism		✓		✓							
1.4Manage experiments											✓
1.5Manage trained parameters				✓							
1.6Supervised learning routines							✓				
2.1Integration of new environments										✓	
2.2Scaling agents and environments			✓								
2.3Manage experience samples						✓					
2.4Agent interactions									✓		
2.5Reinforcement learning routines							✓				
3.1Cross-platform			✓								
3.2Split neural network deployment			✓								
3.3Inputs and outputs abstraction			✓								

In general this component is responsible for managing every aspect of an experiment starting from connecting resources, managing jobs, allocation of correct resources and ending with the aggregation of results, therefore satisfying Requirement 1.4.

4.2. Extensibility

DIANNE is a flexible framework which provides a Java API, CLI and a JSON-RPC interface to manage the framework as well as the ability to add new components which interact with the system. As an example we look at hyperparameters tuning through cross validation and model selection. This mechanism is used to find optimal parameters and select the best performing model.

A popular and well known cross validation strategy is K-fold. The main idea is to break the dataset up into k subsets with the same dimensions. On the first iteration the network is trained on $k-1$ subsets (training set) and tested on the remaining subset (test set) to get that models accuracy. This process is repeated until all subsets have been used as the test set and the average of all test set accuracies is taken to measure the performance of the current model with the configured hyperparameters.

The best model architecture and hyperparameters can then be selected by comparing the model's performance after repeating the K-fold cross validation with other hyperparameters (found through e.g. random search, grid search, Bayesian search, etc.) and different model architectures.

Implementing this workflow in DIANNE can be accomplished by writing a *Dataset* adapter splitting the entire dataset into equally sized folds and an extra component (using the Java or JSON-RPC API) which manages creating new Learn and Evaluate jobs. This component is then responsible for changing hyperparameters, selecting model structures (can be pre-build), selecting the correct cross validation folds, starting new jobs and stopping jobs based on overfitting behaviour of the Evaluate job.

4.3. User interface

In order to facilitate the construction, configuration, deployment and training of neural networks, the DIANNE runtime is equipped with a web-based graphical user interface (GUI) as shown in Figures 2 and 10. Each tab providing a different functionality to design or control a neural network.

The "Build' tab provides a mechanism to drag and drop modules onto the canvas to construct neural networks. The toolbox on the left of the

screen shows all available building blocks. Connecting these building blocks creates layer connections. Before deploying the module it has to be configured by opening the module's configuration dialog (Figure 2(b)) and setting the dimensions, strides, padding, etc.

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. Selecting a device initiates the weight transfer to this particular node. In the last tab the user can couple the input and output modules to real devices. For example, a camera can serve as the network's input, while the output can be used to control a robot.

Lastly, a dashboard is provided, which gives the user the ability to learn neural networks by starting a training procedure and manually search for hyperparameters by testing different values and analyzing the graphs. This interface is heavily connected with the *Coordinator* and this interface enables the user to submit and configure (1) act jobs, generating new experience, (2) learn jobs, updating the pre-build network's weights, and (3) evaluate jobs, visualising the performance of the network and stop the learn job before overfitting.

5. Design and implementation

The DIANNE framework is implemented using the OSGi specification [40], the de facto standard for modularization in Java. Each DIANNE component is implemented as an OSGi bundle, exposing the service interfaces as OSGi services. To allow distributed deployment of the DIANNE components, we built upon AIOLOS [41] platform which discovers, connects and binds remote services.

The DIANNE *Runtime* is the base component of the DIANNE framework and can be deployed on different computer architectures (currently supports ARM, x86 and x86_64). We provide multiple implementations for the native interface and the optimal available native library is automatically loaded. The native implementation is built on the Torch7 tensor library, which provides a BLAS backend to support CPU only devices and a GPU accelerated backend using cuDNN. Figure 11 shows an example deployment of a neural network on two devices, each having a different native backend.

In order to integrate with external systems, DIANNE provides several ways to interface. First of all, every component is available as a well defined OSGi service in the Java runtime. Each deployed neural network can

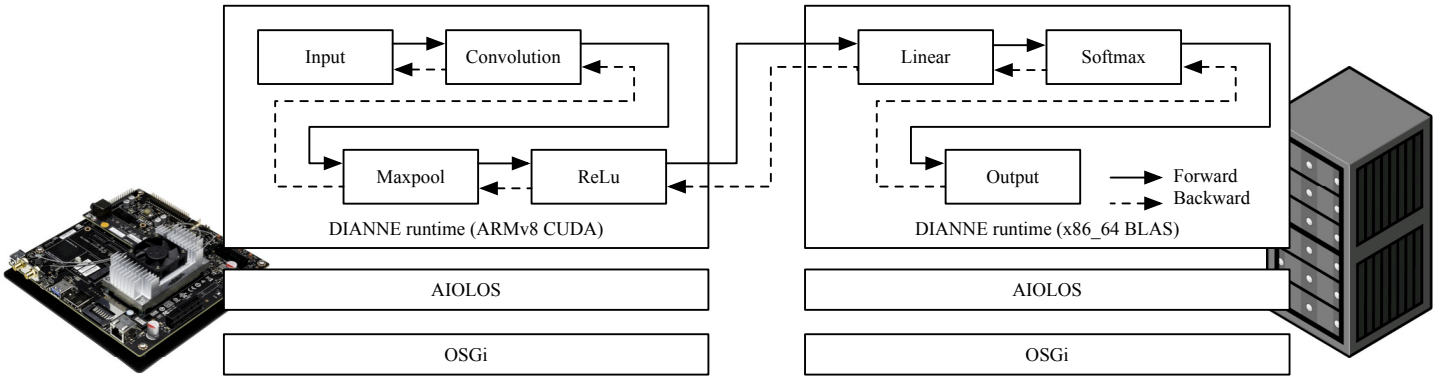


Figure 11: A neural network is constructed as a directed graph of modules that can be deployed on different DIANNE *Runtime*s.

be called via its corresponding *NeuralNetwork* service. For example, DIANNE can be used in combination with device abstraction layers that expose IoT sensors as OSGi services, in order to process sensor information in real time [42]. External programs can interface with DIANNE via the JSON-RPC interface, which exposes the core functionality of the framework.

For the integration of the various reinforcement learning environments, the Environment interface has to be implemented. We integrate with the OpenAI Gym environments using Java Embedded Python (JEP) [43], which embeds CPython in Java through JNI. For our robotics environments, we communicate via the Robot Operating System (ROS) [44], which also provides interfaces to simulators such as V-Rep and Gazebo.

In order to facilitate the deployment of a DIANNE setup, especially when integrating with multiple external programs such as a simulator, the DIANNE build system allows to compile and construct Docker [45] containers for various configurations. This enables developers to create environment optimized Docker containers to deploy on any kind of infrastructure, including cloud providers.

Table 2 gives an overview of deep learning framework features, comparing DIANNE to other popular frameworks mentioned in the introduction. DIANNE provides researcher the ability to test out models and training parameters without writing a single line of code.

Table 2: Deep learning framework comparison.

Features	Software	TensorFlow	Torch7	PyTorch	Caffe2	Theano	DL4J	DIANNE
Built-in multi node support		✓		✓	✓		✓	✓[a]
Multi GPU support		✓	✓	✓	✓	✓	✓	✓[a]
Train models without writing any code					✓			✓
Dynamic deployment								✓
Dynamic computation graphs			✓	✓	✓		✓	✓
Step-by-step debugging				✓	✓		✓	✓
Graphical neural network builder								✓
Connecting sensors and actuators (GUI)								✓
Built-in visualization dashboard		✓					✓	✓
Training job management dashboard								✓
Automatic differentiation		✓		✓	✓	✓	✓	[b]

[a] No programming required.
[b] Future work.

6. Experiments

To illustrate that the framework supports the cases described in Section 3 we conducted three experiments focusing on the features of each case individually.

6.1. Overhead of DIANNE in image classification case

We compared the performance of DIANNE to a number of similar frameworks: Deeplearning4j (DL4J) [11], Torch7 [46] and TensorFlow [6]. We measured 1000 forward passes of the OverFeat (fast) [47] model, using variable-length batches of random data. Where applicable, we measured the performance with and without cuDNN support enabled. Benchmarking was performed on the iLab.t [48] hardware described in Table 3. The results are shown in Figure 12 indicating the mean evaluation time of a single sample in order to indicate how each framework scales with batch size.

Without cuDNN support, DIANNE performs almost identical to Torch7, showing that the service-level abstractions do not impose any significant overhead. With cuDNN support enabled, the performance of DIANNE is on par

Table 3: Hardware specifications.

name	arch.	CPU	GPU	RAM	VRAM
Raspberry Pi 2	ARMv7	Cortex-A7 (900MHz)	NA	1GB	NA
Jetson TX1	AArch64	Cortex-A57 (1.9GHz)	Nvidia Tegra X1 T210	4GB	
iLab.t [48]	x86	E5-2620v3 (2.4GHz)	Nvidia Tesla K40c	32GB	12GB

with TensorFlow. Torch7 performs even better, indicating some headroom for improvements in the cuDNN implementation of DIANNE. Torch7 has its own implementation for cuDNN written in Lua, making it impracticable for DIANNE to include Torch7's cuDNN implementation as a library, with an extra benchmarking option to find the fastest convolution algorithm. This benchmark option was enabled during this experiment. DL4J suffered from a very high variance during benchmarking. While mean times were higher than other frameworks, median values were on par.

To further compare the frameworks we investigated CPU and GPU memory consumption of forwarding samples through the OverFeat (fast) model. We measured the average memory consumption when forwarding 1000 individual random samples in order to remove the memory requirement of loading in a large dataset. Loading in a dataset can be very memory consuming, depending on the implementation. Table 4 shows the results indicating that DIANNE uses the least amount of memory, comparable to Tensorflow, which

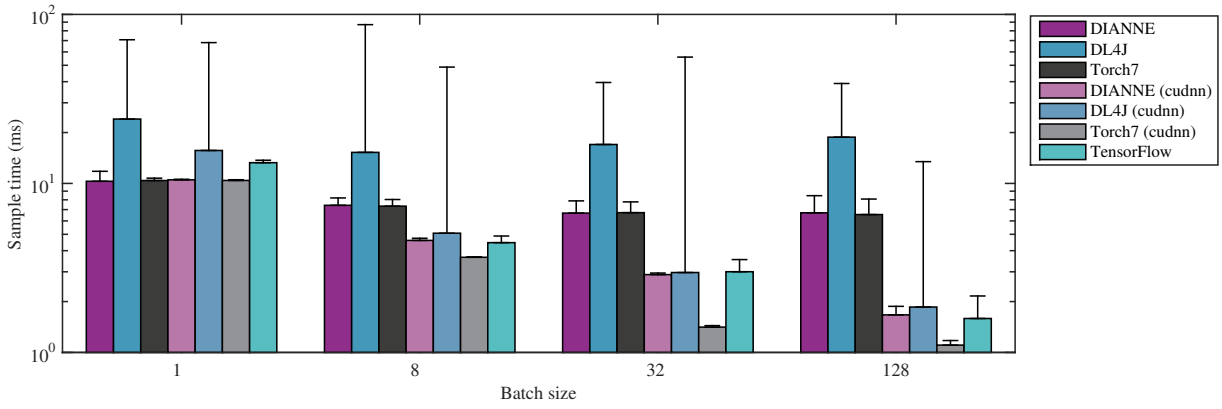


Figure 12: Performance results of the OverFeat (fast) model for different batch sizes. Results show the mean evaluation time of a single sample. Lower values are better and bars indicate standard error.

Table 4: Memory usage results of the OverFeat (fast) model. Results show the mean memory usage while forwarding a single sample through the model. RAM values are private and shared memory combined.

		RAM (MB)	VRAM (MB)
DIANNE	CPU	728	NA
	CUDA	425	714
	cuDNN	557	758
DL4J	CPU	1655	NA
	CUDA	2292	2436
	cuDNN	2368	2590
Torch7	CUDA	1307	1408
	cuDNN	1575	3819
Tensorflow	cuDNN	643	3826

is mostly the memory requirements of the OverFeat model (558MiB).

DIANNE manages to lower the memory requirements by allocating the necessary data only in the specific native memory. A neural network model is directly loaded into native memory (CPU or GPU) through a buffer. In contrast, DL4J and Torch7 hold copies of the tensors in native memory and on the GPU. DIANNE and DL4J are both programmed in Java with native bindings to C++ code, but use a different native strategy.

DL4J uses JavaCPP to wrap the native code, which handles memory management of native objects inside the Java heap space. DIANNE has a custom written interface that allocates objects outside this heap space allowing object allocations to extend beyond Java's configured heap space size. With DIANNE the maximum heap space size can be configured as low as $16MiB$, if no dataset loading is required, which allows all remaining RAM to be used for other allocations. In our tests DL4J could not be started with a configuration less than $1024MiB$, while it only required $5MiB$ during sample feed forward removing a big chunk of memory unused.

In the case of CUDA and cuDNN the CPU memory requirement is even further lowered with a comparable memory usage on the GPU. Tensorflow performs comparable to DIANNE, but uses a different default strategy for VRAM allocation. DIANNE only allocates what the model and batch require, while Tensorflow pre-allocates all GPU memory for the current process.

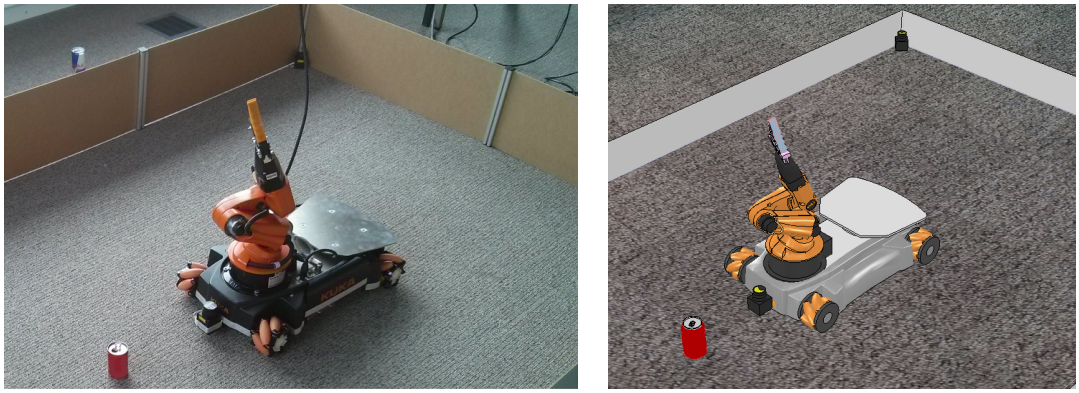
For restrictive devices it is important to lower the maximum heap space

size and the overall memory consumption of deep learning frameworks. From Table 3 and Table 4 we can conclude that Torch7 and DL4J are unable to run on the Raspberry Pi. Even the Jetson would struggle as this memory is shared between CPU and GPU.

6.2. Increase learning speed with multi agent deployments

As a experiment for reinforcement learning, we created an environment for a search and pick task. Our setup is depicted in Figure 13(a) [49]. The environment is constructed as a rectangular search area in which a Kuka YouBot is placed. The robot can move in any direction with its omni-directional wheels and has a Hokuyo URG-04LX-UG01 1D lidar sensor mounted on the base. This sensor is configured with a 180 degree field of view. Both the robot and sensor are controlled and monitored through ROS. For the search target we selected soda cans, which makes a recognisable indent in the laser input. For processing, we equipped the Kuka Youbot with an Nvidia Jetson TX1 embedded GPU board. This board needs to handle the robot controller, laser inputs and neural network processing at inference time on the shared memory and limited processing power.

For training, this environment was replicated in simulation as shown in Figure 13(b), using the V-Rep simulator [38]. Within the simulator, the robot and sensors expose exactly the same interface via ROS. This allows us to control the simulated environment in the same way as the real world.



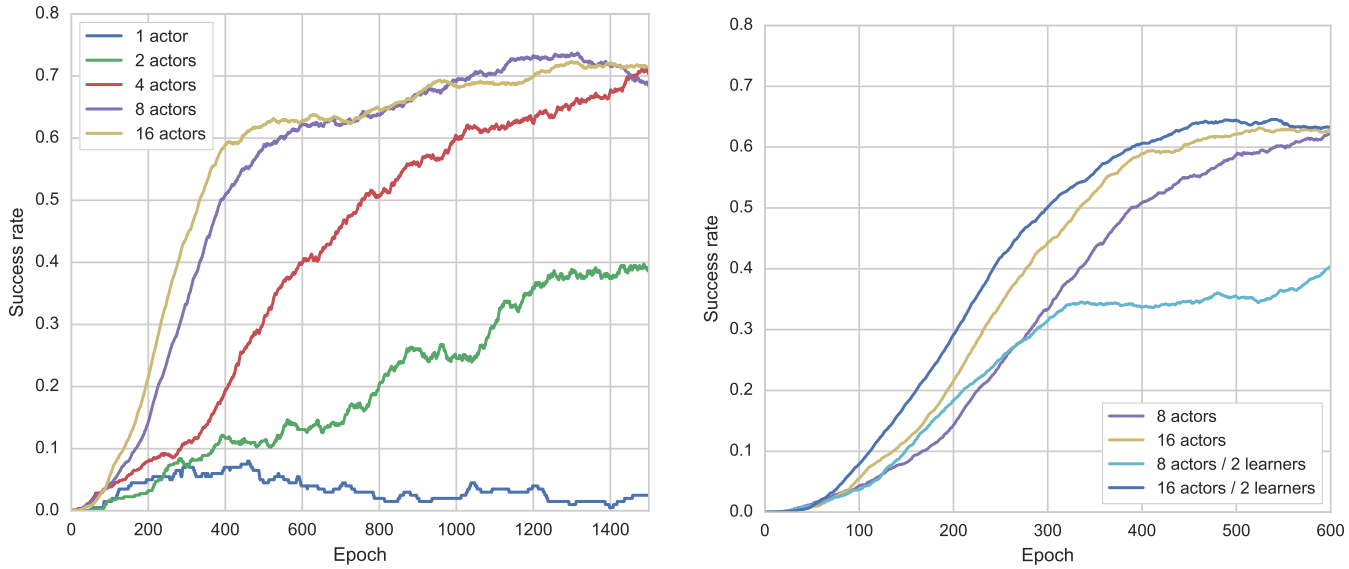
(a) (b)

Figure 13: Experimental setup (a) with a Kuka YouBot in a rectangular arena, equipped with a Hokuyo lidar. We replicated the same setup in the V-Rep simulator (b) for training.

For the pick can use case we will use Deep Q-Network (DQN) [50] approximate and maximize the action-value function Q and derive the robot's policy. The DQN network gets input from the mounted lidar sensor, and outputs Q values for each of 7 discretized actions to control the robot: move left/right, move forward/backward, rotate clockwise/counterclockwise and execute a (hard-coded) grip. As reward function we provide the negative distance of the robot's target position to the can, normalized between -1 (farest) and 0 (succesful pickup location). When the robot collides the reward is set to -1 and a reward of 1 is provided for a succesful grip.

With this experiment we aim to assess the effect of scaling the number of agents sending experience to the same experience pool. We compare the success rate of picking up the object with increasing number of agents in function of the number of epochs, where one epoch is 1000 batch weight updates. Our results are shown in Figure 14(a), concluding that adding more agents benefits the learning speed significantly.

Training the DQN network on a single agent fails to learn this task,



(a) Training with increasing number of actors on a single *Learner* component.

(b) Comparing multiple configurations of variable number of actors and learners.

Figure 14: Learning to fetch a can with increasing number of agents. Plotting the success rate in function of number of epochs with a window size of 200 epochs.

while 2 or more agents converge to the same success rate. With 16 actors we do not get a big improvement any more because the new bottleneck is the computational bottleneck of the learner. By adding extra learners (see Figure 14(b)) or deploying the learner to a more powerful machine we could benefit more from extra agents. Figure 14(b) is a zoomed in view of the left figure that shows adding learners does not directly decrease convergence time and as for 8 agents the learning process is slowed down in the middle with 2 learners.

This phenomenon can be explained by the sync interval at which *Learner*s update the repository's weights. *Learner*s send their gradients to the repository and updates the weights by combining both *Learner*s deltas, making the gradient update bigger and move faster in the right direction. At some point the training starts to diverge as the combined deltas overshoot the corrections slowing down the learning time. In order to make these updates meaningful we need enough samples to learn from. One could investigate further to use an adaptive sync interval, e.g. using an annealing scheme, but this is left as future work. With 16 agents and 2 learners we do see an overall improvement as it starts to converge faster and keeps maintaining its momentum. Specifically for this environment and hardware combination we see that a 1:8 ratio between learners and agents is optimal. We also tested with 4 learners (1:4 ratio) but than the *Learner*s diverged.

Learning with multiple learners (distributed or local) is complex and adds extra criteria to the learner's configuration. Batch size, synchronous or asynchronous weight updates, weight update interval and maximum and minimum experience pool size are all parameters which can be tuned to enhance training on multiple learners. The communication overhead should be taken into account as a constraint for adding more learners [51].

6.3. Offload neural networks on IoT devices

In the last experiment we evaluate with offloading a part of the network, using sequential model splitting during inference time, to specialized hardware without the ability to completely fit the entire network on it. For this experiment we used a Raspberry Pi 2 lacking GPU hardware and a Nvidia Jetson TX1, which is a low budget embedded device with a specialized low power GPU (see Table 3 for specifications). For each DIANNE runtime the optimal native support is deployed, meaning a CUDA tensor library for the Jetson and a BLAS implementation from Torch7 for the Raspberry Pi.

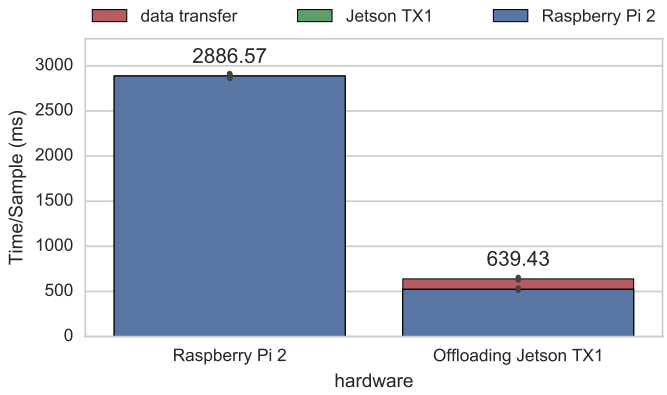


Figure 15: Comparing OverFeat [47] (Fast) network single machine network deployment to a distributed deployment. On the left deployed on a single Raspberry Pi 2 and on the right the network was partially deployed on the Pi and partially offloaded to the Jetson. The part executed on the Jetson is too small to see (4.5ms).

In an IoT environment there are many devices with limited computing power and they could benefit from adding specialized hardware. We compare two deployments of the OverFeat (fast) neural network: (1) the complete neural network deployed on the Raspberry Pi and (2) all the convolutional layers, which are the first 5 of the 8 layers, offloaded to the Jetson TX1.

OverFeat is a large network and requires a lot of GPU or CPU memory. Because the Jetson TX1 has only 4GB memory, which is shared between CPU and GPU, we are unable to fit the entire network on this single device. In this case we can offload only a part of the network running on the laptop to the edge (with a high speed network connection). To compare both scenarios we randomly generate samples with the same dimensions as ImageNet samples and these samples were executed one by one on the neural network. The average execution time per device including data transfer and communication overhead can be seen in Figure 15.

In order to achieve this speed increase we offloaded only the five convolutional layers of the OverFeat (fast) network. The two remaining fully connected layers were still deployed on the Pi. By offloading the compute intensive convolutional modules we get a 4.5 times faster execution time even though the random sample (625KiB) needs to be transferred to the first convolutional layer and the last convolutional layer has to forward its output (144KiB) over the network to the first fully connected layer. The total data

transfer per sample takes 117.5ms on average. Also note that the biggest share of the time in the offloading scenario is due to the part that is still executed on the Raspberry Pi with an average of 521.9ms compared to 4.5ms for the convolutional layers. Therefore, additional speedup can be achieved when having multiple local GPU devices. Further improvements can be made by splitting the network in places where there is less data transfer required, lowering the transfer overhead.

7. Conclusion

We presented DIANNE, a modular framework for deep learning applications. By splitting a neural network into its elementary operations and implementing these as services that can be linked on the fly, neural networks can be distributed across multiple, heterogeneous devices. Moreover, by also offering multiple learning and evaluation routines as services, it supports various flavors of model and data parallelism out of the box. In addition, components are available for supporting reinforcement learning use cases, with integration points to various external systems.

In Section 6 we validated the operation of the framework in three distinct cases: (1) Performance and memory comparison with other deep learning frameworks on a supervised learning problem. (2) Data parallelism by deploying multi agent reinforcement learning with abstraction for simulated and real environment. (3) Model splitting by distributing computation intensive tasks to edge GPU devices. Each experiment showcases the specific features of the framework mentioned in the case studies (See Section 3).

Acknowledgements

This 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 gratefully acknowledge the support of NVIDIA Corporation with the donation of GPUs used for this research.

References

[1] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.

[2] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016, http://www.deeplearningbook.org.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, A. Y. Ng, Large scale distributed deep networks, in: Advances in Neural Information Processing Systems 25, 2012, pp. 1232–1240.

[4] D. Povey, X. Zhang, S. Khudanpur, Parallel training of DNNs with Natural Gradient and Parameter AveragingarXiv:1410.7455.
URL http://arxiv.org/abs/1410.7455

[5] N. Strom, Scalable distributed dnn training using commodity gpu cloud computing, in: Sixteenth Annual Conference of the International Speech Communication Association, 2015.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, et al., TensorFlow: Large-scale machine learning on heterogeneous systems, software available from tensorflow.org (2015).
URL http://tensorflow.org/

[7] D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at google with borg, in: Proceedings of the Tenth European Conference on Computer Systems, 2015.

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

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

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

[11] Deeplearning4j Development Team, Deeplearning4j: Open-source distributed deep learning for the JVM, Apache Software Foundation License 2.0.
URL http://deeplearning4j.org

[12] A. Krizhevsky, I. Sutskever, G. E. Hinton, ImageNet classification with deep convolutional neural networks, in: Advances in neural information processing systems, 2012, pp. 1097–1105.

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

[14] 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, 2012, pp. 1223–1231.

[15] M. A. Alsheikh, D. Niyato, S. Lin, H. p. Tan, Z. Han, Mobile big data analytics using deep learning and apache spark, IEEE Network 30 (3) (2016) 22–29. doi:10.1109/MNET.2016.7474340.

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

[17] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Others, ImageNet large scale visual recognition challenge, International Journal of Computer Vision (2015) 1–42doi:10.1007/s11263-015-0816-y.
URL http://dx.doi.org/10.1007/s11263-015-0816-y

[18] C. Szegedy, S. Ioffe, V. Vanhoucke, Inception-v4, inception-resnet and the impact of residual connections on learning, CoRR abs/1602.07261.
URL http://arxiv.org/abs/1602.07261

[19] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, Accurate, large minibatch sgd: Training imagenet in 1 hour, arXiv preprint arXiv:1706.02677.

[20] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, Vol. 1, MIT press Cambridge, 1998.

[21] J. Schulman, S. Levine, P. Abbeel, M. Jordan, P. Moritz, Trust Region Policy Optimization, in: Proceedings of the 32nd International Conference on Machine Learning (ICML), 2015, pp. 1889–1897.

[22] I. Lenz, H. Lee, A. Saxena, Deep Learning for Detecting Robotic Grasps, International Journal of Robotics Research 34 (4-5) (2015) 705–724.
URL http://dx.doi.org/10.1177/0278364914549607

[23] S. Gu, E. Holly, T. P. Lillicrap, S. Levine, Deep Reinforcement Learning for Robotic Manipulation, CoRR abs/1610.00633.
URL http://arxiv.org/abs/1610.00633

[24] R. Hadsell, A. Erkan, P. Sermanet, M. Scoffier, U. Muller, Y. LeCun, Deep belief net learning in a long-range vision system for autonomous off-road driving, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2008, pp. 628–633. doi:10.1109/IROS.2008.4651217.

[25] S. Thrun, Making cars drive themselves, in: Hot Chips 20 Symposium (HCS), 2008 IEEE, IEEE, 2008, pp. 1–86.

[26] R. Raina, A. Battle, H. Lee, B. Packer, A. Y. Ng, Self-taught learning: transfer learning from unlabeled data, in: Proceedings of the 24th international conference on Machine learning, ACM, 2007, pp. 759–766.

[27] S. M. Erfani, S. Rajasegarar, S. Karunasekera, C. Leckie, High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning, Pattern Recognition 58 (2016) 121–134.

[28] N. Y. Hammerla, S. Halloran, T. Ploetz, Deep, convolutional, and recurrent models for human activity recognition using wearables, arXiv preprint arXiv:1604.08880.

[29] Y. Guan, T. Ploetz, Ensembles of deep lstm learners for activity recognition using wearables, arXiv preprint arXiv:1703.09370.

[30] V. Gokul, P. Kannan, S. Kumar, S. G. Jacob, Deep q-learning for home automation, International Journal of Computer Applications 152 (6).

[31] S. J. Nowlan, G. E. Hinton, Simplifying neural networks by soft weight-sharing, Neural computation 4 (4) (1992) 473–493.

[32] Y. LeCun, C. Cortes, C. J. Burges, The MNIST database of handwritten digits (1998).

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

[34] D. P. Kingma, M. Welling, Auto-encoding variational bayes, arXiv preprint arXiv:1312.6114.

[35] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative adversarial nets, in: Advances in neural information processing systems, 2014, pp. 2672–2680.

[36] M. G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, The arcade learning environment: An evaluation platform for general agents, Journal of Artificial Intelligence Research 47 (2013) 253–279.

[37] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym (2016). arXiv:arXiv:1606.01540.

[38] M. F. E. Rohmer, S. P. N. Singh, V-REP: a Versatile and Scalable Robot Simulation Framework, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2013.

[39] N. Koenig, A. Howard, Design and use paradigms for gazebo, an open-source multi-robot simulator, in: Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on, Vol. 3, IEEE, pp. 2149–2154.

[40] OSGi Alliance, Osgi service platform, release 3, IOS Press, Inc., 2003.

[41] T. Verbelen, P. Simoens, F. D. Turck, B. Dhoedt, Aiolos: Middleware for improving mobile application performance through cyber foraging, Journal of Systems and Software 85 (11) (2012) 2629 – 2639. doi:10.1016/j.jss.2012.06.011.

[42] E. De Coninck, T. Verbelen, B. Vankeirsbilck, S. Bohez, S. Leroux, P. Simoens, Dianne: Distributed artificial neural networks for the internet of things, in: Proceedings of the 2Nd Workshop on Middleware for Context-Aware Applications in the IoT, M4IoT 2015, 2015, pp. 19–24. doi:10.1145/2836127.2836130.
URL http://doi.acm.org/10.1145/2836127.2836130

[43] Jep: Java embedded python, https://github.com/ninia/jep, accessed: 2017-09-26.

[44] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, ROS: an open-source Robot Operating System, in: ICRA Workshop on Open Source Software, 2009.

[45] D. Merkel, Docker: Lightweight linux containers for consistent development and deployment, Linux J. 2014 (239).
URL http://dl.acm.org/citation.cfm?id=2600239.2600241

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

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

[48] S. Bouckaert, P. Becue, B. Vermeulen, B. Jooris, I. Moerman, 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, Ghent University, Department of Information technology, 2012, pp. 1–16.

[49] S. Bohez, T. Verbelen, E. De Coninck, B. Vankeirsbilck, P. Simoens, B. Dhoedt, Sensor fusion for robot control through deep reinforcement learning, arXiv preprint arXiv:1703.04550.

[50] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529–533.

[51] J. Keuper, F.-J. Preundt, Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability, in: Proceedings of the Workshop on Machine Learning in High Performance Computing Environments, MLHPC '16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 19–26. doi:10.1109/MLHPC.2016.6.
URL https://doi.org/10.1109/MLHPC.2016.6