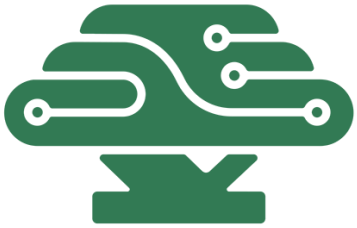

BONSEYES

ARTIFICIAL INTELLIGENCE MARKETPLACE

Deliverable D3.1

Initial Deep Learning Toolbox

Point of Contact	**Amos Storkey**
Institution	**UEDIN**
E-mail	**a.storkey@ed.ac.uk**
Phone	**+44 131 651 1208**

Project Acronym	**BONSEYES**
Project Title	**Platform for Open Development of Systems of Artificial Intelligence**
Grant Agreement No.	**732204**
Topic	**H2020-ICT-2016 Smart Cyber-Physical Systems**
Project start date	**1 December 2016**
Nature	**Report**
Dissemination level	**Public**
Due date	**M12**
Date of delivery	**M14**
Lead partner	**UEDIN**
Contributing partners	**UCLM, ICCS, TCD**
Authors	**Amos Storkey (UEDIN)**
Reviewers	**Tim Llewellynn (NVISO)**

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Economic Affairs,
Education and Research EAER
State Secretariat for Education,
Research and Innovation SERI

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732204 (Bonseyes).

This work is supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 16.0159.

The opinions expressed and arguments employed herein do not necessarily reflect the official views of these funding bodies.

Revision History

Version	Date	Author	Comment
0.1	10 Oct 2017	Amos Storkey	First draft
0.2	15 December		Second draft including partner contributions
	19. Jan 2017	Tim Llewellynn	Review
1.0	19. Jan 2017	Peter Ulrich	Final formatting, formal issues

Contents

Abbreviations, Participant short names and Glossary **v**

Abbreviations v

Participant short names v

Glossary v

List of Tables **vi**

List of Figures **vi**

Summary **1**

1. Introduction: Deep Learning Toolbox **2**

2. Architecture Sensitive Deep-Learning Methods **3**

2.1 Network Model Converters (UEDIN lead) 3

2.2 Efficient Implementations of Neural Network Computations (TCD lead) 4

2.2.1 Classic im2col Algorithm 4

2.2.2 Key takeaways 7

2.3 Analysis of model compression for network size reduction (UEDIN lead) 7

2.4 Plugins to LPDNN for efficient CPU use (ICCS lead) 9

3. Cost-Sensitive, Distributed and Transfer Learning Methods. **12**

3.1 Exploiting characteristics of heterogeneous systems for DNN inference optimization (UEDIN lead) 12

3.2 Latency sensitive methods (UCLM lead) 14

3.3 Data Augmentation Generative Adversarial Networks (UEDIN Lead) 14

3.4 Parameter cost reduction in RNN and Embeddings (TCD lead) 15

References **16**

Abbreviations, Participant short names and Glossary

Abbreviations

AI	**Artificial Intelligence**
CPS	**Cyber-Physical Systems**
LPDNN	**Low Power Deep Neural Network**

Participant short names

NVISO	**nViso SA**
UCLM	**Universidad de Castilla – La Mancha**
TCD	**Trinity College Dublin**
UEDIN	**University of Edinburgh**
FHNW	**Fachhochschule Nordwestschweiz**
TUM-Med	**Klinikum rechts der Isar der Technischen Universität München**
ICCS	**Institute of Communication and Computer Systems**
SYNYO	**Synyo GmbH**
HES-SO	**Haute Ecole Spécialisée de Suisse Occidentale**
ARM	**ARM Limited**
ZFFNAG	**ZF Friedrichshafen AG**
RTRK	**Institut RT-RK**
SCIPROM	**SCIPROM Sàrl**
BTH	**Blekinge Tekniska Högskola**

Glossary

The Bonseyes glossary is available on https://www.bonseyes.com/glossary/.

List of Tables

Table 1: Summary of methods 7

List of Figures

Figure 1: Im2Col is the standard form for image representation in convolutional neural networks 4

Figure 2: Using a sum of scaled matrices to perform convolutions 5

Figure 3: GEMM-accumulating algorithm 6

Figure 4: The Grouped and Pointwise Block G(g) substitutes all full convolutions with a grouped followed by pointwise convolution. A pointwsie Bottleneck can also be introduced to reduce parameters further. 8

Figure 5: Test Error vs No. Parameters for student networks learnt with attention transfer on CIFAR-10. The legend lists different compression families. 9

Figure 6: VGG-16 implementation in Caffe running on ARM big.LITTLE architecture (8 cores) 13

Figure 7: Comparison between hand coded version of VGG-16 and Caffe in OpenCL running on Odroid-XU4 board (CPU+GPU). The baseline is the CPU version of Caffe with 1 thread. 13

Summary

The Deep Learning Toolbox is designed to enable users who have developed non-embedded deep learning methods to transform those into methods and code that do work in embedded scenarios and can be transformed to code that will run as effectively as possible on developer platforms. The toolbox is required to work at many levels. First, it must provide implementations for standard deep learning models. Second it must provide methods for transforming standard or provided deep learning models into models that are more optimized for constrained systems, through an accuracy versus resource payoff that can be defined by the user. The learning process for such models should be specified. Third it must transform a learnt model into suitable code for constrained systems (this is a substantial effort – the code generator is shared with WP4). Fourth it must be able to optimize the compilation of that code for heterogeneous platforms.

This document summarizes the methods, code and documentation developed as part of the initial development of the Deep Learning toolbox. These methods include:

- Provision of hand-coded implementations of standard networks (alexnet, VGG networks, resnet) for benchmarking against.
- Moonshine: a method for distillation of standard networks into considerably cheaper networks (in memory and computation terms) with little loss of performance.
- Low Power Deep Neural Networks (LPDNN): a code generation suite, in development. LPDNN will provide full code generation facility for the deep learning toolbox. At this stage it can generate a number of standard networks, and do full equivalence verification. Equivalence verification is vital to ensure that implemented networks do the same as those trained using some other framework: most frameworks are incompatible with one another, for example in how they provide padding for convolutions. We do this via an augmented Caffe network definition model.
- TriNNity: a toolbox of code generation components that choose optimal matrix handling structures for full and sparse matrices. The TriNNity transformations have been tested on hand coded networks for CPU implementations on different platforms. Optimal choices of matrix handling provide substantial speed gains in CPU Neural Network implementation.
- CPU/GPU tests on a number of embedded devices that implement compile-time scheduler choices for neural network implementation. Again such compile optimization can produce substantial difference in performance.
- Data Augmentation Methods for generalising models to unseen domains.

Altogether substantial progress has been made at all levels of the toolbox. At this stage, the LPDNN needs substantial further development; however a full pipeline of the development is now possible for any network structure via a compiled Caffe implementation, and for select network structures via LPDNN.

1. Introduction: Deep Learning Toolbox

Deep Learning Toolbox: The method and tools developed in WP3 will contribute to open source projects and provide, for the first time, open, componential deep learning tools for constrained architectures and embedded systems. Data rich technologies, from on device speech recognition and translation, vehicle warning, navigation and driving systems etc. rely on efficient and flexible machine learning. We will track use and benefit as part of the deep learning toolbox, through a distributed reward system: as part of the toolbox, we will assess the added value of individual components. Hence, we can and will not only assess, and report, on how often components are used, feedback about that component, but also measured added value from those components. In this section we summarise the primary goals on the 12 month deliverable and what has been achieved against each of these. The core tool that is being developed is the Low Power Deep Neural Network package (LPDNN), but other packages have also been released in their own right.

2. Architecture Sensitive Deep-Learning Methods

The following components are the primary targets from 3.1 Architecture Sensitive Deep-Learning Methods.

- **Converters from CAFFE format to pytorch networks and vice versa (UEDIN)**

  Bidirectional converters from/to pytorch and tensorflow have been made and provided as part of the LPDNN.

- **Encapsulated structure for networks, data etc (NVISO)**

  The LPDNN includes a docker specification for network, network parameters, data etc. This enables networks and data to be transactable quantities, which are fully specified and unambiguous. It also provides a means of ensuring benchmarking tests are properly matched to an actual instantiation, preventing a mismatch between the code and data used for a benchmark and that provided on a marketplace.

- **Definitions of standard networks in CAFFE format, along with a number of more efficient implementations (TCD)**

  Since the LPDNN tooling supports Caffe models out of the box, we have focused our efforts on well-known and widely used networks. In particular we have considered the performance of AlexNet, VGG-B, VGG-C, VGG-E, and GoogleNet. TCD have developed many efficient implementations for different sizes and shapes of convolutions, and have open-sourced our DNN primitive library containing 70+ different convolution routines, including FFT and Winograd convolutions (https://bitbucket.org/STG-TCD/trinnity).

- **Standard Benchmarking tools for network performance, including robustness to input degradations (UEDIN, TCD)**

  TCD have made publically available their benchmark suite for DNN inference, which includes benchmarks of per-layer performance in AlexNet, VGG-B, VGG-C, VGG-E, and GoogleNet versus Caffe (https://bitbucket.org/STG-TCD/trinnity-benchmarks). We plan to add more networks and more competitors (e.g. TensorFlow) in the near future.

- **Analysis of model compression methods for reduced bit depth and network size reduction (UEDIN)**

  We detail the analysis of initial compression methods below. We find that significant compression of standard networks in possible with only small reduction in performance.

- **Plugins to LPDNN for efficient CPU use (ICCS)**

2.1 Network Model Converters (UEDIN lead)

We anticipated it would be necessary to provide a way to convert models between different frameworks during this project. Typically, we plan to work with a small set of reference architectures, and develop methods on each of these. So, we developed a bridge between frameworks guaranteed to work with these architectures. The current shortlist is:

- AlexNet
- GoogleNet
- Network in Network
- ResNet50
- VGG16
- SqueezeNet

All of these are famous architectures that can be trivially implemented in all major deep learning frameworks; such as Tensorflow, Caffe, PyTorch etc. However, given a particular configuration of the

weights trained in any of these it is difficult to transfer that configuration to a different framework. Some work has been done on making this easier, and there are some large industry pushes to this end, such as ONNX. At this point, none of these solutions are applicable in our project.

Another approach to model conversion is to parse the code in each framework for the basic operations and then rebuild a new computational graph automatically in the new framework. This produces model definitions that are not useable.

We wanted a method that would provide uniform model definitions regardless of the framework, and to have them defined ahead of time. We call the solution a *bridge* because it transparently moves the weights back and forth between frameworks without changing the model definition.

The code is published in the Bonseyes bitbucket repository. It has been tested on the VGG16 architecture and converts back and forth between Caffe and PyTorch. We are ready to extend it to test on the other model definitions and provide a further bridge to Tensorflow, when this becomes a bottleneck during research.

2.2 Efficient Implementations of Neural Network Computations (TCD lead)

A number of more efficient convolution algorithms have been designed and tested, these were published in ASAP 2017 (http://ieeexplore.ieee.org/document/7995254) which provides further details. TCD have extended the paper with more algorithms for publication in a Journal in 2018 (https://arxiv.org/abs/1709.03395).

2.2.1 Classic im2col Algorithm

The classic way that convolutional neural networks implement the convolutional processing is to organise images column-wise in memory and do convolutions with this structure. This process is illustrated in Figure 1. Im2Col, as this organisation is called, needs lots of memory for the patch matrix: precisely $C \times H \times W \times K^2$ space, where C is the number of channels in a layer, and H and W are the height and width of a layer, and K is the convolution size.

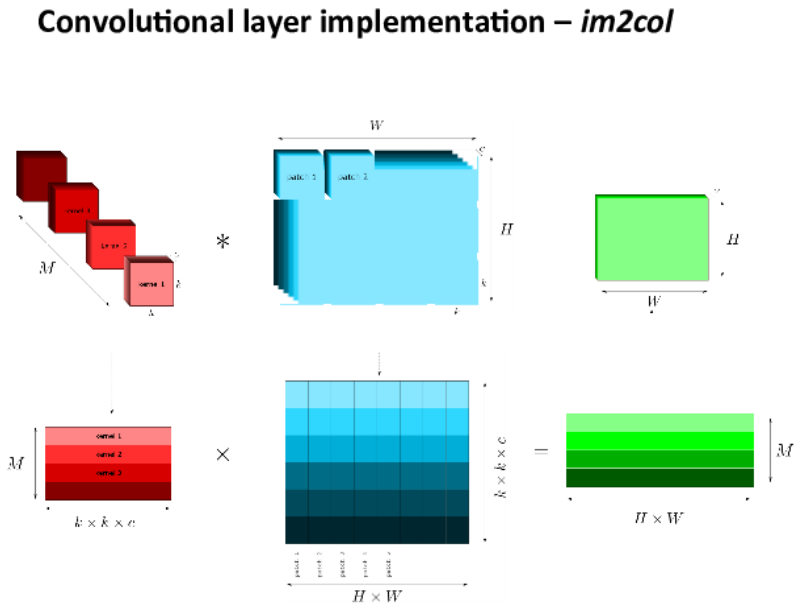

Figure 1: Im2Col is the standard form for image representation in convolutional neural networks

For a deep learning toolbox, it is critical that we find another algorithm for convolution that

- Uses GEMM to achieve high speeds
- Does not build a patch matrix

In this work, a family of new GEMM-based algorithms is proposed, that are based on sums of convolutions, and do not need a patch matrix. Figure 2 illustrates one approach for achieving this via a sum of scaled matrix method. The figure illustrates the fundamental differences between the in-memory organisation of this approach. In addition the use of matrix operations enables optimized standard generalised matrix libraries to be used.

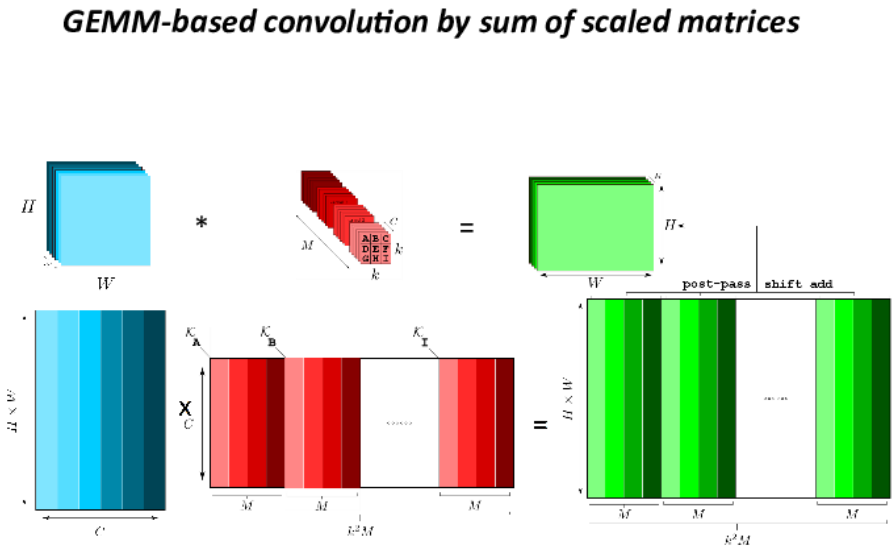

Figure 2: Using a sum of scaled matrices to perform convolutions

We can extend our sum of scaled matrices algorithm to input with multiple channels by

- Replacing matrix scaling with 1x1 DNN convolution
- Computing KxK DNN convolution as the sum of K^2 1x1 DNN convolutions
- Implementing 1x1 DNN convolution with matrix-matrix multiplication (GEMM)

The result of this is that no extra patch matrices needed. The primary downside is that there would be more GEMM calls

- We do K^2 GEMM calls versus just one GEMM call for im2col
- Output matrix size **increases** K^2-fold

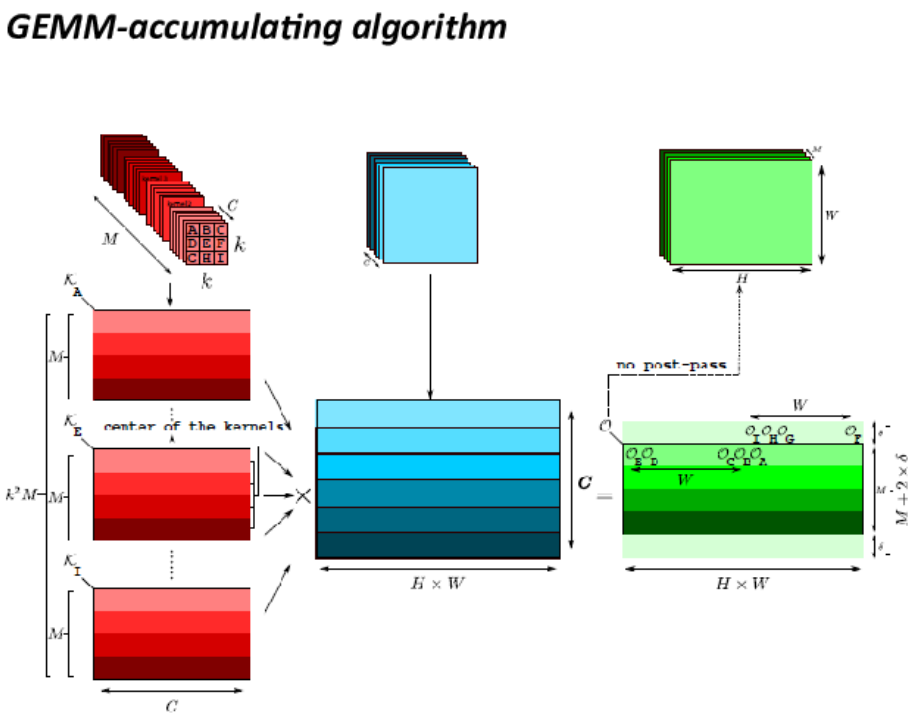

Figure 3: GEMM-accumulating algorithm

Various things are worth noting about this approach, illustrated in Figure 3.

- BLAS GEMM is already an accumulating algorithm
  - Takes an optional matrix parameter to accumulate to
  - So we can do the accumulation as part of the GEMM call
  - Potentially faster than a post-pass loop
- There are significant complications
  - We shift the result matrices when accumulating
  - How should we manage pixels at the boundaries of images?
- Convolutions stop at the edge of images
  - All convolution algorithms deal with boundaries as special cases
  - But we are building our sum of 1x1 convolutions with GEMM
- We are misusing the GEMM accumulate
  - At boundaries we spill into and over-write the next row
  - Lots of wrong values in results matrix
- Two strategies
  - Post-pass fix-up of values
  - Dynamically modify input matrix with carefully-placed zeros

We can summarise the requirements of each of these methods in the following Table 1:

Table 1: Summary of methods

- **Input image of size CHW**
  - C channels, H pixels high, W pixels wide
- **Kernels**
  - KxK size, C channels, M kernels
  - K is typically 1, 3 or 5

Algorithm	#GEMM calls	Ops/GEMM call	Extra space
Im2col	1	K^2CHWM	$O(K^2CHW)$
Aravind accumulating	K^2	CHWM	O(CHW)
GEMM accumulating	K^2	CHWM	O(KW+HC+WC)

2.2.2 Key takeaways

We can choose between input-explosion, output-explosion, or compromise for each layer of the network

- Use compromise when input is large and output is large (early in net)
- Use output explosion when input is large and output is small (middle of net)
- Use input explosion when output is large and input is small (late in net)

- DNN convolution can leverage optimized GEMMs libraries
- Im2col is fast but needs lots of extra space
- Our GEMM-accumulating approach offers
  - Similar performance
  - At a fraction of the additional space

- To find out more
  - A. Anderson, A. Vasudevan, C. Keane and D. Gregg. Low-memory GEMM-based convolution algorithms for deep neural networks. arXiv:1709.03395v1
  - Code open sourced on bitbucket

2.3 Analysis of model compression for network size reduction (UEDIN lead)

We have explored methods to reduce the parameter cost of neural networks while keeping their performance intact. For this, we devised a simple strategy where one transforms a standard network by replacing its convolutional blocks with cheaper, low-parameter alternatives. This requires no architectural change to the original network. We show that when these smaller networks are trained using distillation techniques they are able to perform similarly well to the original network with a significant reduction in parameters, and outperform reduced architectures with standard convolutional blocks.

Our cheap replacement blocks utilise a combination of (i) splitting each full convolution kernel into a set of smaller, grouped convolutions and (ii) contracting the number of channels a convolution is performed across through a bottleneck.

Figure 4 illustrates the G and BG blocks we have designed for this purpose.

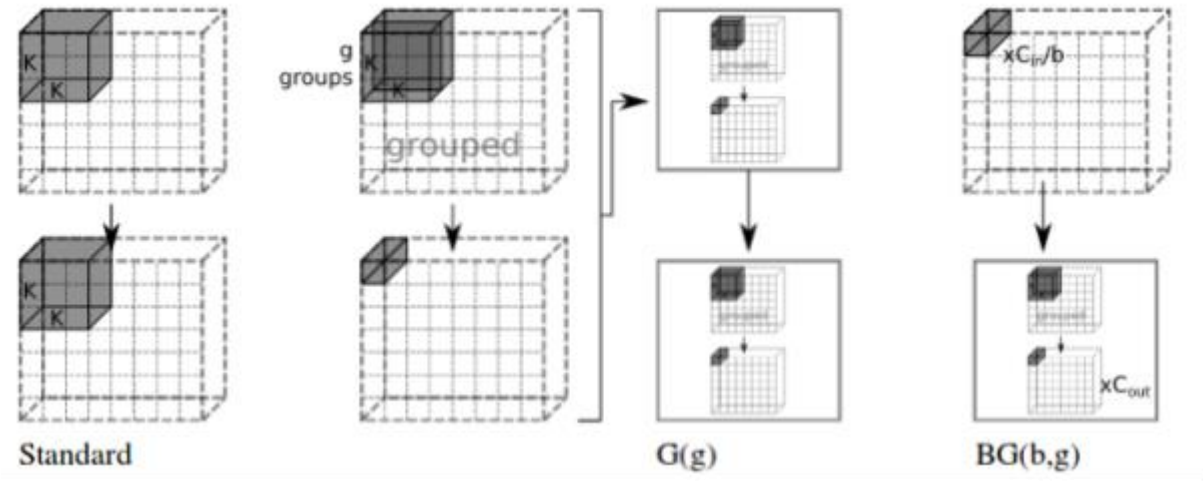

Figure 4: The Grouped and Pointwise Block G(g) substitutes all full convolutions with a grouped followed by pointwise convolution. A pointwsie Bottleneck can also be introduced to reduce parameters further.

Trained from scratch, networks with these cheaper blocks perform markedly worse than the original. However, modern distillation methods (particularly attention transfer) allow us to use the original network as a *teacher* to aid in the training of a *student* network with cheap convolutions.

We train a multitude of student networks with different blocks on the CIFAR-10 image classification dataset. The performance on the test set vs. parameter cost for each network is shown in Figure 5. The original (teacher) network is also shown on this plot. It is clear that all the networks with cheap convolutions outperform standard networks with an equivalent number of parameters.

Some of the results are remarkable: G(N/8) has an error very close to that of the teacher but has just over a fifth of the total parameters. BG(2,M/8) has less than a tenth of the parameters of the teacher for a cost of just over a percent error.

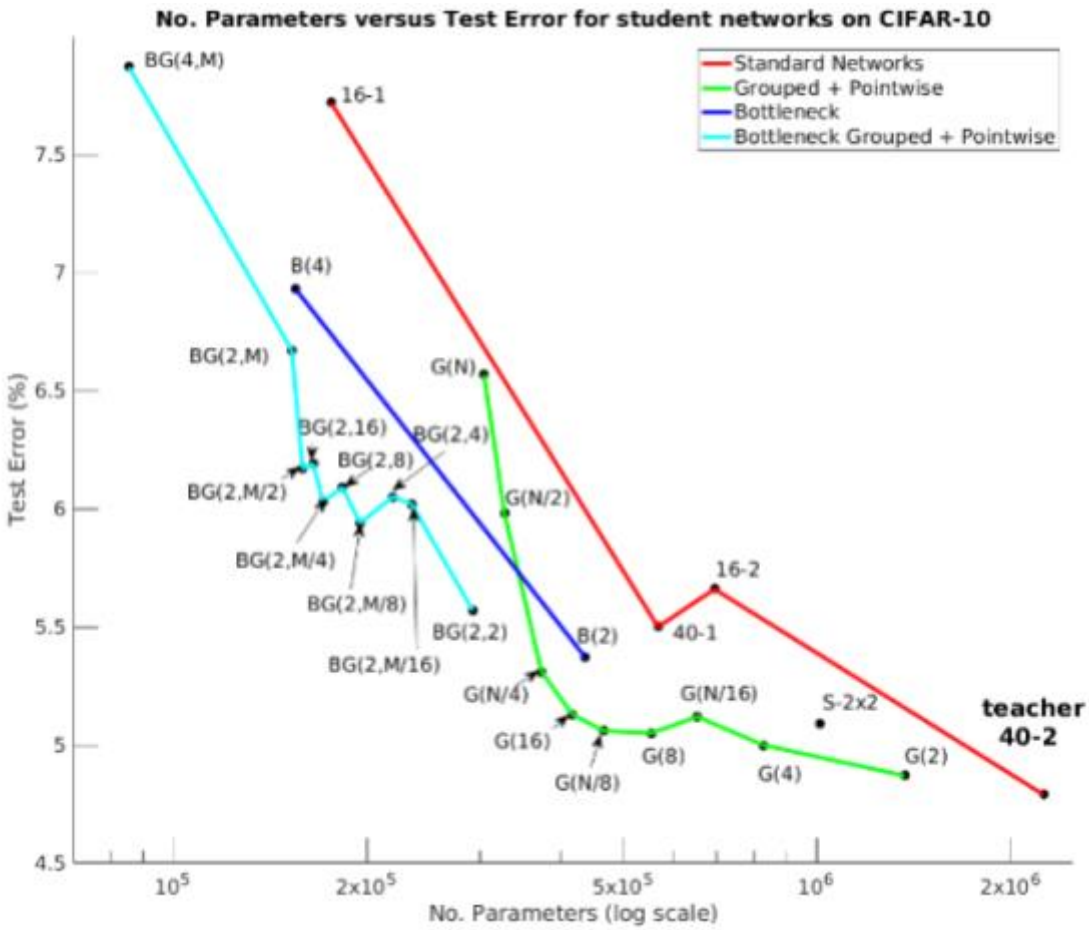

Figure 5: Test Error vs No. Parameters for student networks learnt with attention transfer on CIFAR-10. The legend lists different compression families.

The paper for this work is available on arXiv https://arxiv.org/abs/1711.02613. The source code is available on github for the consortium and will be later be made public.

2.4 Plugins to LPDNN for efficient CPU use (ICCS lead)

We have introduced two new plugins to the Low-Power Deep Neural Network (LPDNN) inference engine that is being developed within the Bonseyes project. The first plugin includes the integration of the high-performance NNPACK acceleration package for Neural Network computations, which is publicly available here and is optimised for ARMv7 processors with the NEON instruction set, ARMv8 (AArch64) processors and x86-64 processors with the AVX2 instruction set. Since the hardware platforms that will be used for Bonseyes' demonstrators will include ARMv8 processors, it seemed fruitful to incorporate this acceleration package within LPDNN. The second plugin includes multiple implementations of sparse convolutional and fully-connected layers, which can be used to compress the model representation in memory when the model has been pruned with techniques that introduce unstructured sparsity in the model. The implementations follow the typical approach of lowering the tensors involved in the respective computations to matrices and rely on standard sparse BLAS routines, including the sparse matrix - dense matrix multiplication for the convolutional layer and the sparse matrix - dense vector multiplication for the fully-connected layer. In the convolutional layer, the original $K \times C \times H_k \times W_k$ 4D weight tensor is lowered into a $K \times CH_kW_k$ 2D matrix, while the original $C \times H_{in} \times W_{in}$ 3D input activation tensor is lowered into a 2D $CH_kW_k \times H_{in}W_{in}$ matrix. A multiplication of the lowered sparse weight matrix with the lowered dense input matrix is performed and the resulting matrix can be directly used by the subsequent layer without any modification. We have currently experimented with the sparse BLAS routines available in the Intel MKL library for x86 architectures. The sparse weight matrix is represented with either one of the de facto

standard sparse matrix storage formats, including the Coordinate (COO), Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats. Even though these are not state-of-the-art formats for sparse matrix computations, they are useful for an initial evaluation. In our initial experiments we have tested single-threaded performance of the sparse layer implementations using multiple sparsity levels on an Intel Pentium T3400 CPU, and compared them to the corresponding dense implementations. Specifically, we benchmark the following implementations:

- Convolutional layer:
  - Im2Col lowering + dense × dense matrix multiply (BLAS, Intel MKL)
  - Im2Col lowering + sparse × dense matrix multiply (sparse BLAS, Intel MKL)
    - Using the COO format for the sparse matrix
    - Using the CSR format for the sparse matrix
  - Direct sparse, i.e., no transformation to the input activation tensor (hand-written)
    - Using the CSR format for the sparse matrix
- Fully-connected layer:
  - dense matrix × dense vector multiply (BLAS, Intel MKL)
  - sparse matrix × dense vector multiply (sparse BLAS, Intel MKL)
    - Using the COO format for the sparse matrix
    - Using the CSR format for the sparse matrix

We test the third convolutional ("conv3") and the first fully-connected ("fc6") layer of the AlexNet network with increasing sparsity levels in the range {60-90}%, i.e., 60-90% of the weights are zero.

	Relative performance over Im2Col + dense-gemm		
	sparse-gemm-coo	sparse-gemm-csr	sparse-direct-csr
alexnet_conv3_60%	0.162	0.164	0.158
alexnet_conv3_70%	0.219	0.221	0.207
alexnet_conv3_80%	0.324	0.333	0.311
alexnet_conv3_90%	0.657	0.657	0.605
	Relative performance over dense-gemv		
alexnet_fc6_60%	0.527	1.195	
alexnet_fc6_70%	0.690	1.581	
alexnet_fc6_80%	1.043	2.333	
alexnet_fc6_90%	2.130	4.900	

	sparse matrix format compression ratio	
sparsity level (%)	COO	CSR
60	0.83	1.25
70	1.11	1.66
80	1.66	2.49
90	3.33	4.99

Concerning the sparse convolutional layer, we notice that no execution time speedups are gained through sparsity using either of the evaluated implementations. However, this is a limitation of the specific implementations, which are not optimal (GEMM-based implementations of Intel MKL) or optimised (direct hand-written implementation). The performance gap can be significantly reduced through extensive optimizations that can be found in the literature and are currently work in progress. Some useful

observations can be made, however, even with these initial results. First, concerning the sparse GEMM-based implementations, we see no performance difference between the two sparse matrix storage formats, even though the CSR format achieves higher compression ratios. This is because the sparse × dense matrix multiply is a compute-bound kernel and, thus, it does not benefit from data compression. It will actually benefit from optimizations that are typically used to optimise dense matrix multiply, e.g. cache blocking, tiling, etc. Also, one should notice that the direct sparse implementation achieves similar performance to the sparse GEMM-based ones, without requiring additional memory for lowering the input activation tensor. Concerning the fully-connected layer, here the sparse implementations are quite efficient even for low sparsity levels. This is due to the fact that the sparse matrix dense vector multiplication kernel is a memory-bandwidth-bound kernel and, thus, benefits from data compression techniques. Since the execution time of the majority of convolutional neural networks is dominated by the convolutional layers, the focus of our current work is on providing high-performance sparse implementations for these layers.

3. Cost-Sensitive, Distributed and Transfer Learning Methods.

The following components are the primary targets for 3.2 Cost-Sensitive, Distributed and Transfer Learning Methods.

- **Bayesian optimization of model compression against cost/constraints (UEDIN). Joint optimization of compiler options and network options (TCD, UEDIN)**

The within component optimization sits within the inner loop of the hyper-parameter search that was used for the analysis of compression in WP1. It needs to be developed further for the full pipeline joint optimization of models and compile-options that forms part of the Full Deep Learning Toolbox.

- **Benefit assessment of particular network elements or transformations to enhance suitability for embedded devices (TCD)**

Half of the experimentation in TCDs forthcoming CGO 2018 paper concerns the ARM Cortex A-57 processor, which is present in many embedded and automotive development kits, such as the NVIDIA Jetson TX1. They assessed the suitability of over 70 different convolution algorithms for inference on this embedded processor, in both single-threaded and multi-threaded execution modes.

- **Benchmarking of hand coded versus generated network code (UEDIN, TCD)**

Two benchmarking suites have been developed by the two partners. The first targets network decisions, used in producing the Pareto curves in Figure 5. This involves systematic efficient hyper-parameter search for each possible compression option and network size. The second publicly released benchmark suite works at the lower level and contains a number of hand coded strategies for network implementations, which we benchmark against an optimal arrangement produced from a PBQP solver (https://bitbucket.org/STG-TCD/trinnity-benchmarks). We have also benchmark standard Caffe developments and hand-coded versions of VGG-16 for CPU (OpenMP) and CPU+GPU (OpenCL) for comparison purposes and to give a point of reference for the performance of our generated network code.

- **Learning approaches for ensuring robustness against changes at deployment.**

We have developed a novel Generative Adversarial Mechanism for augmenting small data availability in a deployment domain. This effectively associates data points with whole manifolds, and enables much more data-efficient learning methods. It gives state of the art performance on one-shot learning tasks and for data augmentation.

3.1 Exploiting characteristics of heterogeneous systems for DNN inference optimization (UEDIN lead)

We investigated the limitations of some deep learning frameworks on heterogeneous systems and identified key points for improvement on top of OpenCL.

We explored the popular methods employed to achieve parallelism in the widely used deep learning frameworks (Caffe, TensorFlow), which can be utilised on heterogeneous systems (multi-core CPU, GPU). Our first observation is that parallelism is achieved naively by current frameworks. Plotting the inference time of VGG-16 with Caffe on an ARM big.LITTLE architecture (Cortex A15 Quad + A7 Quad), we can see (Figure 6) that parallelism is beneficial only when cores are of the same characteristic (1-4) on the “big” architecture.

Adding an additional core with the “LITTLE” architecture has the effect of penalising the inference time by 60% instead of contributing to speedup. This clearly indicates that current deep learning frameworks are designed for uniform computation splits, more common on larger platforms (servers), and not very useful on systems with heterogeneous computation resources.

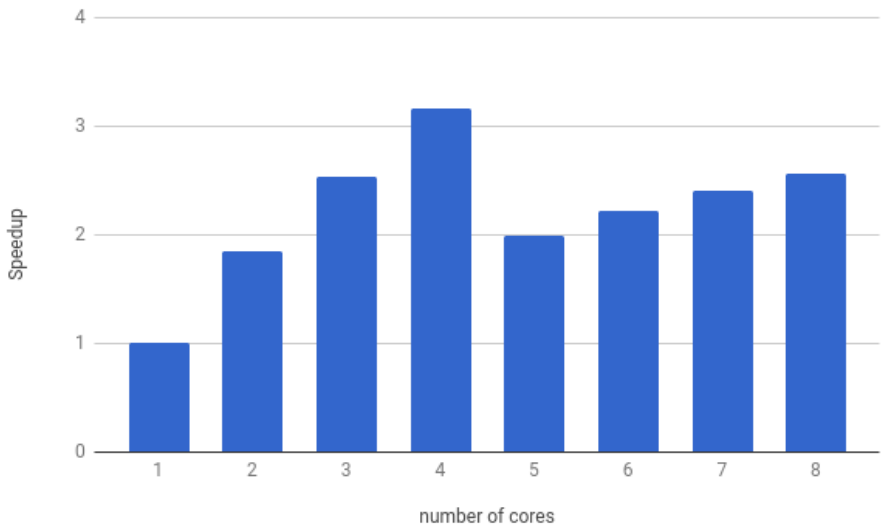

Figure 6: VGG-16 implementation in Caffe running on ARM big.LITTLE architecture (8 cores)

On the other hand, if we observe (Figure 7) the inference time of our implementation of VGG-16 on the same big.LITTLE architecture, we see how the speedup increases as we increase the number of threads (cores). The reason for this is the finer granularity of the threads created in our implementation. It is important to notice that values in Figure 7 are normalised to the first column in Figure 6 (that is, 1 core), which is a way of comparing the performance of both implementations of VGG-16. Based on that, we see that our implementation provides much slower inference times, but the comparison is not completely fair, as our current version is not yet using any BLAS (Basic Linear Algebra Subprograms) library, while Caffe does. Therefore, there is a clear scope for better results when we adopt the BLAS operations available in common libraries.

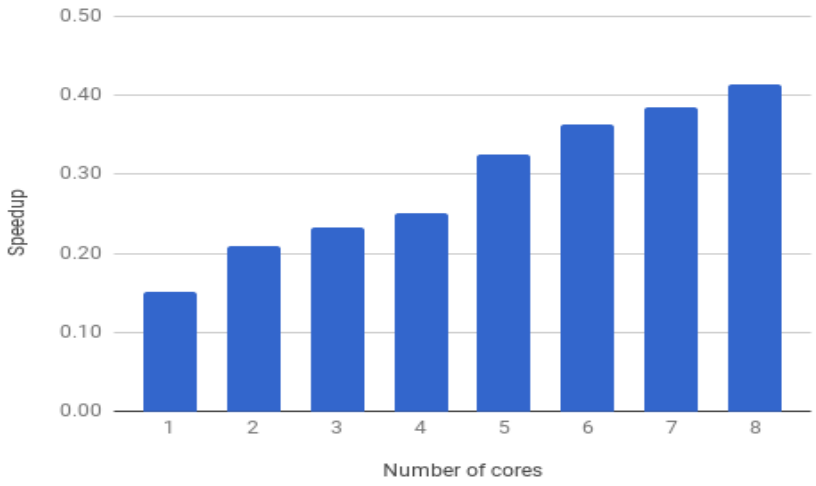

Figure 7: Comparison between hand coded version of VGG-16 and Caffe in OpenCL running on Odroid-XU4 board (CPU+GPU). The baseline is the CPU version of Caffe with 1 thread.

In summary, to address the previous limitations/observations we aim to split workloads in chunks that adequately fit specific hardware characteristics. For this, we identified a set of parameters that immediately impact the OpenCL kernel workload. Considering these parameters is essential to calibrating the workload and avoiding synchronisation penalties. These observations feed directly into the OpenCL task scheduler we are developing.

3.2 Latency sensitive methods (UCLM lead)

Even with optimizations, deep learning inference in embedded systems can still take too long. From our experience with embedded systems, in some cases inference can take >1s per image. This means that the input may change while inference is still being computed. A change in the input can in that case >2s to be reflected in the output (worst-case). Such latencies are not tolerable and in fact may render the application useless. We will focus on an scenario of a continuous recognition task (some of which appear in the project demonstrators), in which the system's decision must be given within a limited time.

In this context, methods will be developed that, based on observation of changes in input (and/or possibly in first computed layers), can stop inference as early as possible. Obviously, inference should be done only when we know that the current output may change. A very simple example of this is in the problem of facial emotion recognition. If the mouth and eyes regions have not changed "significantly" from previous frame, we should not attempt inference. That is a check at the signal level, although we can also think of a check at a more "semantic" level (i.e. from within the first layers of the network itself).

In the context of inference acceleration, multiple alternatives and solutions have been proposed. In the following we give a non-exhaustive taxonomy:

Computational:

- Caches
  - Parallel or otherwise specialized resource allocation/mapping
  - Bit precision reduction
  - Sparsity
- "Semantic":
  - Early stopping (e.g. BranchyNet). Stop at a layer when confidence may be enough
  - Paced layer execution (e.g. clockwork convnets). Different layers are updated according to their semantic stability

The methods that will be explored here pertain to the second category. We envision methods that provide the developer with a trade-off between reduced latency and accuracy. Other constraints (memory, for example) will not be considered at first. The methods will be first prototyped using flexible frameworks and environments (Matlab or similar) and then implemented in the context of a mainstream deep learning framework.

3.3 Data Augmentation Generative Adversarial Networks (UEDIN Lead)

One fundamental issue in deep learning for embedded systems is that they are deployed in changing environments. The scenarios that methods are used in vary, and do not necessarily match the training scenario. It is vital to develop approaches that are robust to these changes.

One setting that provides and abstract environment for this is the k-shot learning setting: we need to learn from a variety of related training environments how to do well in a previously unseen test environment for which we have little data. We tackle this approach via a mixture of supervised and unsupervised learning methods, where the unsupervised method captures the effective class-equivalent manifolds for each data point. Hence this can be applied to new settings with little data.

In this work we demonstrate a novel Generative Adversarial Network (GAN) training setup with which one can use an existing GAN framework (i.e. WGAN GP or Standard GAN) to learn to one-shot generate plausible augmentations of data samples using data-learnt augmentations. Intuitively the model learns a manifold around a data point within which a sample remains in the same class. Furthermore the concept of class is extracted directly from the image pairs passed to the discriminator and implicitly learned by the Generator network as a result of backpropagation. The definition of the classes themselves is learned implicitly by the GAN based on the provided image pairs with which the discriminator is trained. More specifically the discriminator is either presented two images from the same class but that are not the same

sample or the GAN conditional input and the GAN output, thus implicitly pushing the GAN to learn to augment the conditional input image such that it remains in the same class, but it is different enough to be considered a separate sample. One of the novelties of this Data Augmentation technique using GANs is that at generation time you are not restricted by the classes you have already learned (i.e. No Labels are passed to the generator) rather the generator can one shot generate from unseen-class data points. We call this form of GAN a Data Augmentation Generative Adversarial Network or DAGAN.

We evaluate the DAGAN in a number of ways. One way we can use it is to augment one-shot learning methods using pixel distance and matching networks as well as to augment vanilla classification. In all cases we can see improvements over the state-of-the-art baselines. The current preprint of this work is available at https://arxiv.org/abs/1711.04340.

3.4 Parameter cost reduction in RNN and Embeddings (TCD lead)

Image captioning and video captioning are more challenging tasks than object recognition in single images. The difference is that captioning models generate natural language sentences other than single object labels. Recurrent neural networks are typically leveraged in word sequence generation. In this case, one captioning model incorporates a CNN model and a RNN model. A NeuralTalk [1] model can have 90 million parameters and the DenseCap [2] model is even bigger. We are working toward parameter reduction in an integrated model of CNN and RNN, with emphasis on RNN and Embedding representations.

In the NeuralTalk model convolutional features of visual regions are mapped to a h-dimensional embedding. On the other hand, the word sequence embedding is encoded by Bidirectional RNN and the embedding is also in h-dimension. The objective function evaluates the alignment of image region to sentence and vice versa. DenseCap [2] dropped the visual region proposal network of NeuralTalk and a fully convolutional localization layer was applied to locate region features from convolutional features. DenseCap was trained with VisualGenome which covers much richer annotation data. In NeuralTalk, word embeddings are in dense vectors with 300 dimensions. In order to train a more portable captioning model, alternative embedding representations and RNN parameter reduction are necessary. Candidate approaches of model compression include:

Quantization - Reduced precision arithmetic has been applied in deep CNN and it achieved comparable classification accuracy as original models. However, quantization degrades RNN performance in many trials. He et al. [3] introduced robust RNN quantization approaches. Currently we are conservative to keep 32-bit precision for the NeuralTalk model, where the embedding and score ranking steps in a captioning model prefer high precision.

Sparsity - Narang et al. [4] applied weight pruning during RNN training and generated sparse model. RNN model size is reduced by 8X. Chen et al. [5] proposed sparse word representations over dense vectors of word embeddings. Rare words are represented by sparse linear combination of common words.

Rank reduction - A SVD rank reduction approach was introduced for RNN and MGRU (Cox [6]).

Knowledge distillation - As introduced by Hinton et al. [7], a student model can be distilled from a teacher model through softmax output matching. The student model can be smaller in size and help deployment. Distillation has been applied in classification models. On the other hand, word embedding models are also suitable for dimension reduction. Mou et al. [8] indicated two distillation approaches for embeddings: matching softmax and encoding embeddings. A distilled word embedding model can be applied inline with image captioning.

We work with available model compression approaches on RNN and embeddings for captioning tasks and also explore new methods.

References

[1] A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. IEEE Transactions on Pattern Analysis and Machine Intelligence, 39(4):664–676, April 2017.
[2] J. Johnson, A. Karpathy, and L. Fei-Fei. DenseCap: Fully Convolutional Localization Networks for Dense Captioning. ArXiv:1511.07571, November 2015.
[3] Q. He, H. Wen, S. Zhou, Y. Wu, C. Yao, X. Zhou and Y. Zou. Effective Quantization Methods for Recurrent Neural Networks. ArXiv:1611.10176, November 2016.
[4] S. Narang, E. Elsen, G. Diamos and S. Sengupta. Exploring Sparsity in Recurrent Neural Networks. ArXiv:1704.05119, April 2017.
[5] Y. Chen, L. Mou, Y. Xu, G. Li and Z. Jin. Compressing Neural Language Models by Sparse Word Representations. ArXiv:1610.03950, October 2016.
[6] J. Cox. Parameter Compression of Recurrent Neural Networks and Degradation of Short-term Memory. ArXiv:1612.00891, December 2016.
[7] G. Hinton, O. Vinyals and J. Dean. Distilling the Knowledge in a Neural Network. ArXiv:1503.02531, March 2015.
[8] L. Mou, R. Jia, Y. Xu, G. Li, L. Zhang and Z. Jin. Distilling Word Embeddings: An Encoding Approach. ArXiv:1506.04488, June 2015.