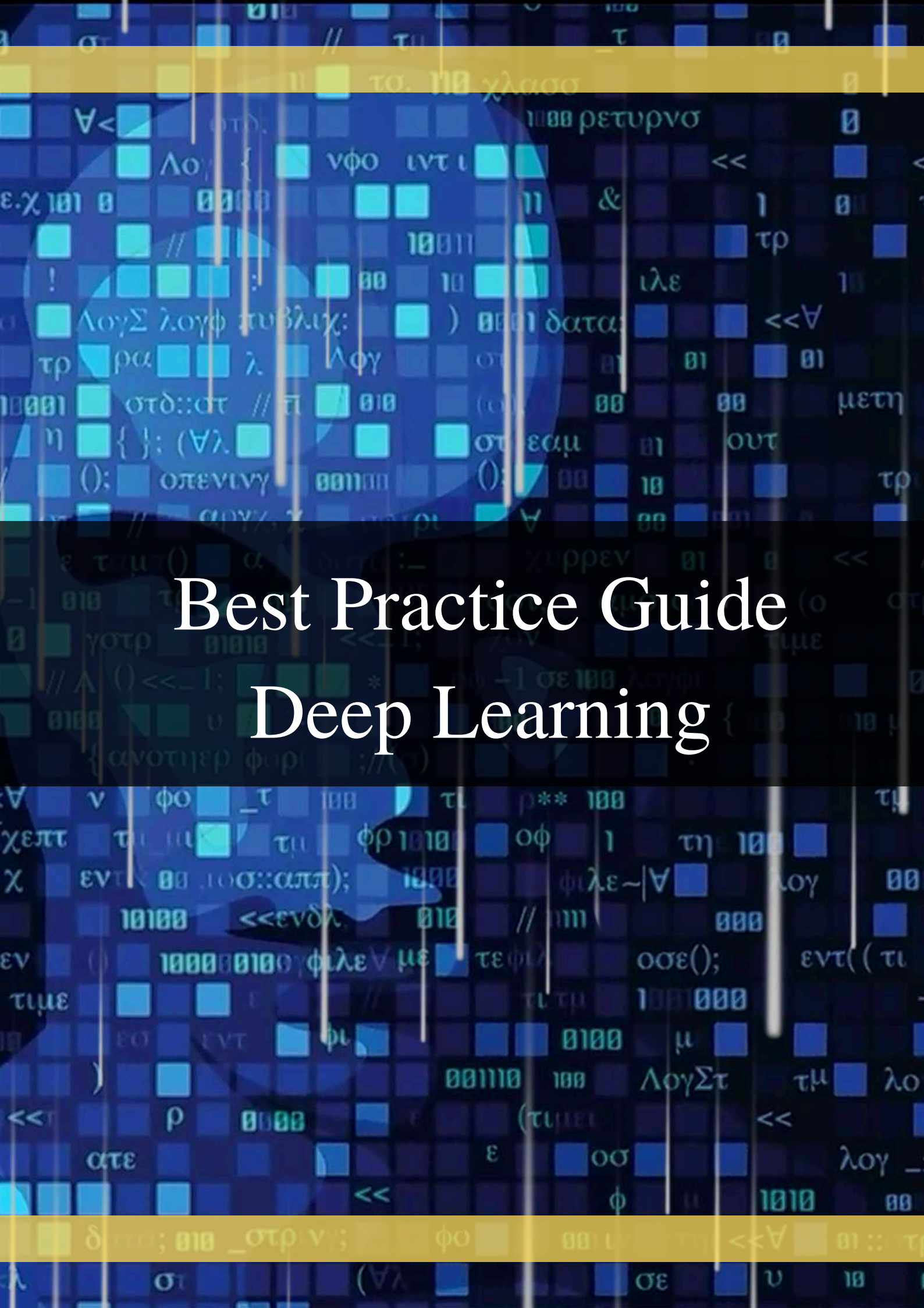


Best Practice Guide
Deep Learning

Table of Contents

1. Introduction

Artificial neural networks (ANNs) are a class of machine learning models that are loosely inspired by the biological neural networks that constitute animal brains. Artificial neural networks use multiple layers of nonlinear processing units for feature extraction and transformation. This allows models to represent multiple levels of abstraction from the data: an approach that works well for many types of problems, such as image, sound, and text analysis. Neural networks with many layers are known as deep neural networks, and the process of training such networks is known as ‘deep learning’.

Sparked by various inference and prediction challenges on publicly available large datasets (such as MS-COCO [1], ImageNet, Open Images, Yelp Reviews, the Wikipedia Corpus, WMT, Merk Molecular Activity, Million Songs and FMA) and by having available open-source frameworks (such as TensorFlow, Caffe and PyTorch), the deep learning field has evolved rapidly over the past decade. With more complex neural networks and larger input data sets, the scalability of deep learning algorithms is an increasingly important topic.

The main aim of this guide is to teach you how to perform deep learning at large scale. In the process, different algorithms, software frameworks and hardware platforms will be discussed. This should help you pick the most suitable framework and hardware platform for your deep learning problem. However, note that this guide only gives a broad overview and does not aim to replace software framework manuals or a deep learning course.

The guide is structured in five chapters: Hardware, Algorithms, HPC and scaling, Frameworks and Use cases. The separation into these different chapters is occasionally difficult, as the concepts are closely related: the type of framework you choose may be dependent upon the hardware you want to run on, which algorithms it supports, etc. We recommend reading the full guide if you want to get a complete picture.

2. Algorithms

This best practice guide focusses on Artificial Neural Networks (ANN). In this chapter, we will introduce various deep learning algorithms. The intention is to provide basic background knowledge of the algorithms employed in the use-cases – a full discussion of algorithms is outside the scope of this guide. Additionally, we discuss which algorithms are particularly computationally intensive and may require distributed training.

2.1. Supervised learning

In machine learning, supervised learning is the process of optimizing a function from a set of labeled samples (dataset) such that, given a sample, the function would return a value that approximates the label. It is assumed that both the dataset and other unobserved samples, originate from the same probability distribution.

We will use P and E as the probability and expectation of random variables; $z \sim D$ denotes that a random variable z is sampled from a probability distribution D ; and $E_{z\sim D}[f(z)]$ denotes the expected value of $f(z)$ for a random variable z . So, for a sample domain X , label domain Y , true labels of z denoted by $h(z)$, a set of functions $f : X \rightarrow Y$, we define a loss function $L_D(f) \equiv P[f(z) \neq h(z)]$, that minimizes the generalization error. In practice, in ANNs, f is defined as a vector of parameters w. This is defined in multiple layers.

$$w^* = \arg\min_w L_D(f_w) = \arg\min_w E_{z\sim D}[l(w,z)]$$

where l is the loss of an individual sample.

For the optimization purposes, if samples are independent and identically distributed, the loss function l should be continuous and differentiable. In regression problems, it is possible to use straightforward loss functions such as the squared difference, however in classification problems, a simple binary loss definition such as

$$l(w,z) = \begin{matrix} 0 \, if \, f(z) = h(z) \\ 1 \, if \, f(z) \neq h(z) \end{matrix}$$

does not match the continuity and differentiability criteria.

To provide this distribution for multi-class classification problems define Y as a probability distribution of the inferred class types, instead of a single label. The model output is typically normalized to a distribution using the softmax function. The loss function then computes the difference of the prediction from the true label “distribution”, e.g., using cross- entropy:

$$l(w,z) = -\sum_i h(z)_i \log \sigma(f_w(z))_i$$

The cross-entropy loss can be seen as a generalization of logistic regression, inducing a continuous loss function for multi-class classification. Minimizing the loss function can be performed by using different approaches, such as iterative methods (e.g. the Broyden-Fletcher-Goldfarb-Shanno algorithm, BFGS [2]) or meta-heuristics (e.g., evolutionary algorithms [3]). Optimization in machine learning is predominantly performed via gradient descent. Since the full distribution D is, however, never observed, it is necessary to obtain an unbiased estimator of the gradient. Observe that $\nabla L_D(w) = E_{z\sim D}[\nabla l(w,z)]$ (linearity of the derivative). Thus, in expectation, we can descend using randomly sampled data in each iteration, applying Stochastic Gradient Descent (SGD).

In the chapter “HPC and Scaling” there are descriptions of several distribution strategies for SGD-based training.

For data-parallel distribution, one should be aware that since workers observe the full dataset less often, it becomes important to use good parameters for smoothing techniques like decay rates, batch normalization, or moving averages over batches. Most of our use-cases are based on supervised learning algorithms with multiple classes and are optimized by SGD.

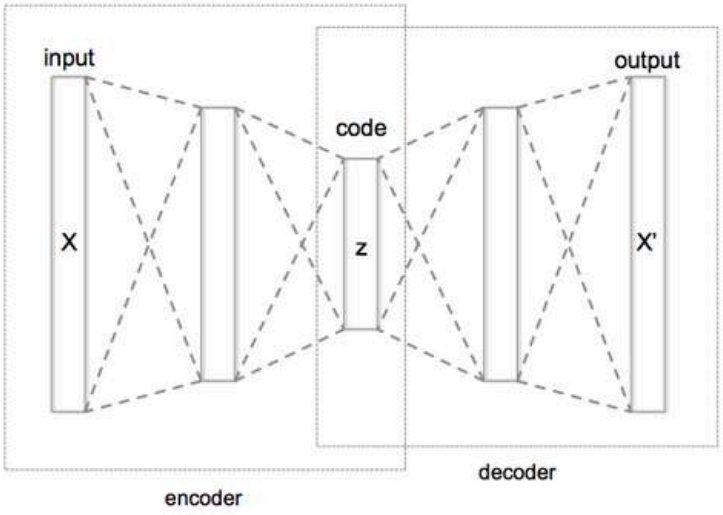
2.2. Unsupervised and Reinforcement Learning

Two other classes of algorithms in machine learning are unsupervised and reinforcement learning. In the former class, the dataset S is not labeled (i.e., h(z) does not exist) and training typically results in different objective functions, intended to infer structure from the unlabeled data. The latter class refers to tasks where an *environment*

is observed at given points in time, and training optimizes an *action policy function* to maximize the *reward* of the *observer*. As an example, consider a monkey (observer) that is presented with a light (environment) and has two buttons that he can press (action). When the light turns on (at some point in time) the monkey should press the left button (action) to obtain a banana (reward). This will train the monkey to always press the left button when the light turns on (action policy function), thus maximizing his reward.

In the context of deep learning, unsupervised learning has two useful implementations: auto-encoders, and Gen-
erative Adversarial Networks (GANs). Auto-encoders are neural networks that receive a sample X as input, map that to a lower dimensional representation z (the encoder) and subsequently reconstruct from z an output X' that is as close as possible to the original sample X (the decoder, see Figure 1). When training such networks, it is possible to, for instance, feed samples with artificially-added noise and optimize the network to return the original
sample (e.g., using a squared loss function), in order to learn de-noising filters. Alternatively, similar techniques can be used to learn image compression.

Figure 1. Schematic representation of an auto-encoder.



Source: Chervinskii (https://en.wikipedia.org/wiki/Autoencoder).

GANs (see Figure 2) are a recent development in machine learning. They employ deep neural networks to generate realistic data (typically images) by simultaneously training two networks. The first (discriminator) network is trained to distinguish "real" dataset samples from "fake" generated samples, while the second (generator) network is trained to generate samples that are as similar to the real dataset as possible.

Figure 2. Schematic representation a GAN.

Reinforcement Learning (RL) utilizes DNNs for different purposes, such as defining policy functions and reward functions. A recent and famous example of RL is Alpha Go [4], which defeated a professional human player and of which the latest iteration, AlphaZero, is perceived as the world's top player in Go [101]. Training algorithms for reinforcement learning differ from supervised and unsupervised learning and use methods such as Deep Q Learning and A3C. These algorithms are out of scope for this best practice guide, but their parallelization techniques are similar.

2.3. Convolutional based algorithms

A convolutional neural network [102] is an ANN architecture that uses the convolution operator (see Figure 3) in one of its layers. That is, two functions produce a third that describes how the shape of one is changed by the other. Commonly this is used as an encoding of some input space (e.g. images), with regularized linear activations and uses several layers of the same encoding (i.e. convolutional filters) and pooling layers (e.g. max pooling, see Figure 4) to define an information hierarchy. In practice convolutional layers tend to also be used for reducing the total number of connections needed by a layer as that can easily be overwhelming for a fully connected layer, especially in the case of inputs with large dimensions.

Figure 3. Schematic illustration of a convolutional operation. The convolutional kernel shifts over the source layer, filling the pixels in the destination layer.

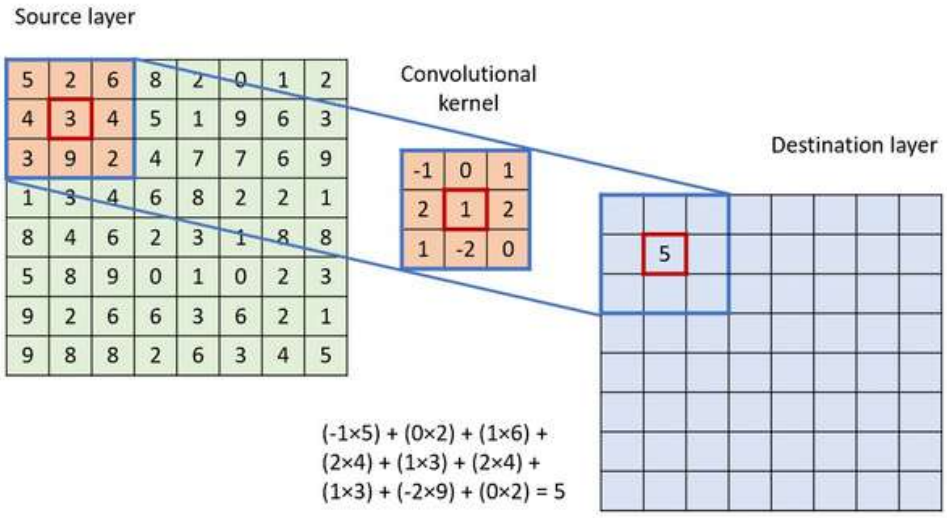
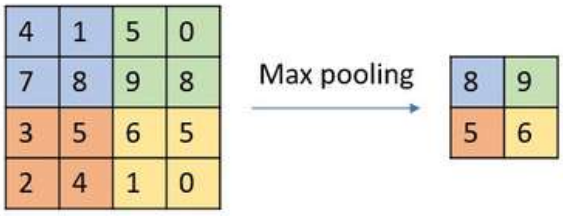


Figure 4. Schematic illustration of a max pooling operation with 2 × 2 filter and stride of 2.



Most of the use-cases in the last chapter of this guide employ convolutional based algorithms. For the reader's convenience we have listed the most utilized architectures: MobileNets, ResNeXt, Xception, DenseNet, ResNet, Inception-ResNet, SqueezeNet, Inception-v3, GoogLeNet, VGG, AlexNet, SENet, LeNet, DPN, NASNet. How- ever, there are many more candidates.

The number of neurons determines (through the associated number of weights and activations) the memory footprint of the network, this can lead to several implementation and design decisions. Please refer also to the section "Model parallelism" in the chapter "HPC and Scaling".

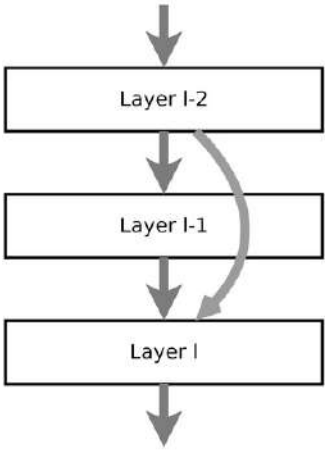
Currently, most deep learning software stacks have efficient implementations for convolutional kernels by exploiting hardware features through algebra libraries like cuDNN [5] and MKL-DNN [6].

2.3.1. Residual neural networks

Residual networks are networks that contain connections that skip one (or multiple) layers (Figure 5) [7] [103]. Normal ANNs suffer from the so-called vanishing gradients problem when they contain too many layers [8]. The

novelty of residual networks is that an identity function regularizes the desired mapping function. Thus, residual neural networks can be much deeper than traditional architectures. As such, residual neural networks typically require a lot of computational power to be trained.

Figure 5. Canonical form of a residual neural network, where a connection from layer l-2 skips layer l-1.



One of the most famous residual network architectures – and one employed also in the chapter “Use-cases” of this guide – is the ResNet architecture. Training a 50 layer residual network (ResNet-50) on the ImageNet-1K dataset takes around 10 days using an NVIDIA P100 GPU card. Training a larger ResNet-152 in the same setting would take roughly 3 weeks. As the size and complexity of the training datasets increases, supervised classification systems become even more accurate [104]. This of course increases the time-to-trained model. By continuing the previous example, it would take roughly one year to train a ResNet-152 model using the full ImageNet-22K dataset, which is about one order of magnitude larger than ImageNet-1K. Thus, there is a clear need for distributed training for such models.

2.3.2. 3D convolutions

A 3D convolutional neural network uses 3D convolutional kernels to operate on a 3D input space. Training these architectures is an even more time consuming task compared to their traditional 2D counterpart, as the extra kernel dimension triggers an exponential increase in weights. Such a task can take weeks. To speed up this research cycle, researchers leverage the power of modern supercomputers to decrease the time necessary for training these models. Due to the more controllable nature of synchronous stochastic gradient descent and relatively limited straggling effects, a lot of approaches opt for a synchronous instead of an asynchronous approach for 3D optimization.

2.4. Generative Adversarial Networks

A very exciting and productive subfield of ANNs are GANs. As previously mentioned, GANs are a set of ANNs with one ANN serving the role of a generator, which generates new artificial samples, and another ANN is a discriminator, which is tasked with distinguishing the artificial samples from the true samples. Because a network is needed for each of these tasks (generation and discrimination), such architectures are typically also computa- tionally intensive to train.

Fortunately this workload is easy to parallelize. Even more so, when thinking about stochastic optimization meth-

ods, recent research showed that large batch size might have a stabilizing effect on GANs [105].

Here is a list of some of the most relevant GAN algorithms published to date:

•Activation Maximization Generative Adversarial Nets [106]

•AdaGAN: Boosting Generative Models [107]

•Adversarial Autoencoders [108]

•Bayesian GAN [109]

•Conditional Generative Adversarial Nets [110]

•Energy-based Generative Adversarial Network [111]

•Generative Adversarial Networks [112]

•Generative Adversarial Parallelization [113]

•Generative Adversarial Residual Pairwise Networks for One Shot Learning [114]

•Geometric GAN [115]

•Good Semi-supervised Learning that Requires a Bad GAN [116]

•Gradient descent GAN optimization is locally stable [117]

•Improved Training of Wasserstein GANs [118]

- InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets [119]

•Loss-Sensitive Generative Adversarial Networks on Lipschitz Densities [120]

•Parametrizing filters of a CNN with a GAN [121]

•PixelGAN Autoencoders [122]

•Progressive Growing of GANs for Improved Quality, Stability, and Variation [123]

•SegAN: Adversarial Network with Multi-scale L1 Loss for Medical Image Segmentation [124]

•SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient [125]

•Simple Black-Box Adversarial Perturbations for Deep Networks [126]

•Unrolled Generative Adversarial Networks [127]

•Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks [128]

•Wasserstein GAN [129]

2.5. Recurrent based algorithms

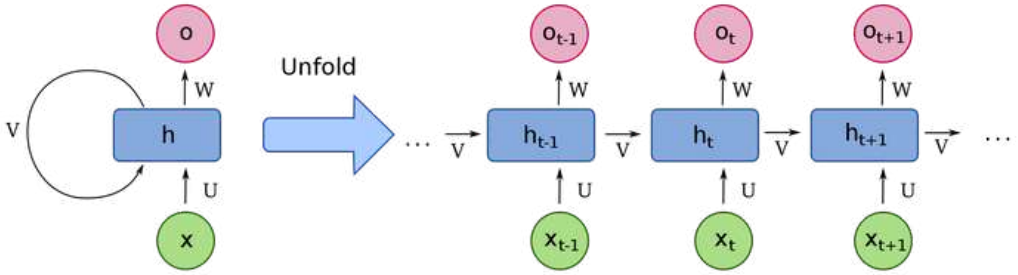
The simplest machine learning problem involving a sequence is a one-to-one problem. In this case, we have one data input or tensor to the model and the model generates a prediction with the given input. Linear regression, classification, and even image classification with convolutional network fall into this category. We can extend this formulation to allow for the model making use of the past values of the input and the output. This is known as the one-to-many problem.

The one-to-many problem starts like the one-to-one problem where we have an input to the model and the model generates one output. However, the output of the model is now fed back to the model as a new input. The model now can generate a new output and we can continue like this indefinitely. This is why these are known as recurrent neural networks (RNNs).

A recurrent neural network deals with sequence problems because their connections form a directed cycle. In other words, they can retain state from one iteration to the next by using their own output as input for the next step. In programming terms this is like running a fixed program with certain inputs and some internal variables. The simplest recurrent neural network can be viewed as a fully connected neural network if we unroll the time axes (see Figure 6). One can build a deep recurrent neural network by simply stacking units to one another. A simple

recurrent neural network works well only for a short-term memory. We will see that it suffers from a fundamental problem if we have a longer time dependency.

Figure 6. Schematic view of the loop structure of an RNN (left) and the same RNN unfolded (right).



Input is given by Xt , output by ot , the internal state by ht and U, V and W are the parameter matrices. Source: François Deloche (https://en.wikipedia.org/wiki/Recurrent_neural_network).

One of the classical applications of RNNs is time series prediction. Although there are many variants, some key RNN architectures are:

•Bi-directional RNN [130]

•LSTM [131]

•Multi-dimensional RNN [132]

•GRU (Gated Recurrent Unit) [133]

•GFRNN [134]

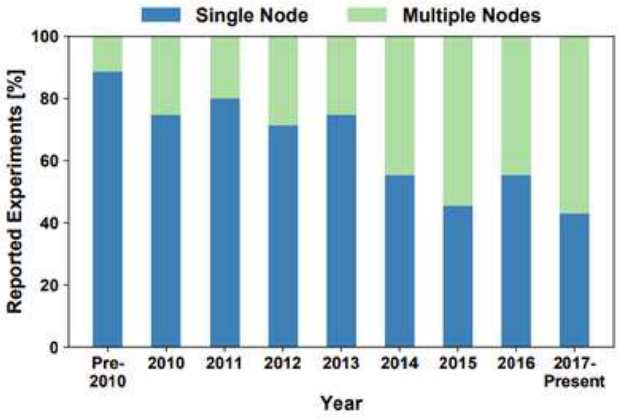
•Tree-Structured LSTM [135]

•Grid LSTM [136]

3. HPC and Scaling

As deep learning models become more complex, and input and output data becomes larger, deep learning is be- coming a compute intensive task that calls for distributed computation (Figure 7). Scaling out deep learning work- loads on HPC infrastructures promises to deliver faster research cycles and improved products. This can be ob- tained by pairing large, Internet-scale data to several types of HPC-scale datacenter, but there are many bottlenecks involved in reaching this. The bottlenecks are both problem-dependent, as well as system-dependent.

Figure 7. Fraction of non-distributed vs distributed deep learning over time.



Source: Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis [137].

3.1. Parallelism

In order to scale out neural network training one has to first consider the type of parallelism that would give the best benefits for the given problem. From a work sharing perspective, the main parallelization strategies are data and model parallelism.

3.1.1. Data parallelism

Data parallelism involves splitting a large batch of images across a collection of workers, each holding the full model. The forward pass involves no communication; however, the backward pass involves aggregating the gra- dients computed by each individual worker with respect to its separate part of the “global batch”. The ratio between the computational cost of the network layers and the size of the layers typically gives a rough indication of the scaling efficiency that a training process will have. A disadvantage of data parallelism is that by scaling out, the “global batch size” (i.e. the total number of examples across all workers that are seen in a single forward pass) increases. This can affect the convergence of deep learning algorithms and pose difficulties in achieving the same accuracy levels that can be obtained with smaller batch sizes.

3.1.2. Model parallelism

Model parallelism involves splitting the model layers across a collection of workers. Among other advantages of model parallelism are the fact that the batch size stays constant, and that large models that would not fit the memory of a single device can be trained. However, this scheme requires active communication in the forward pass, thus requiring a lot more communication than a data parallel approach. Therefore, unless the advantages of model parallelism are absolutely critical (e.g. because the model does not fit in a single worker’s memory), most research on large-scale training is done using data parallelism. Schemes involving a mix of data and model parallelism also exist and are known as hybrid parallelism.

3.2. Updating model parameters in distributed training

3.2.1. Synchronous SGD

SGD optimization is an iterative process that, in a distributed setting, can be performed either synchronously or asynchronously. Most research uses synchronous SGD, meaning that workers have to synchronize the updated model parameters based on the aggregated gradients in between each of the training steps. That way, all workers start each training step with the exact same set of model parameters. This guarantees consistent convergence behavior, but comes at the expense of introducing a blocking *barrier* between your training steps: workers will always have to wait for the slowest worker to finish, before they can continue with the next iteration.

Whether this is an issue in practice depends on the degree of load imbalance you expect. Examples of factors that can affect load balance are the homogeneity of your workers (e.g. do all your nodes contain identical hardware?) and the size of the worker pool (a large worker pool generally shows more variability in 'speed').

3.2.2. Asynchronous SGD

The asynchronous SGD scheme involves using a parameter server to which all workers send their updates and from which workers subsequently receive a certain model state, not necessarily updated by all gradients. This means that workers do not have to wait between training iterations for all other workers to send their gradient updates, they just get the model state as it is at that instant in time. This approach does not guarantee consistent convergence behaviour, but has the advantage that it naturally guards against the load imbalance issues that can occur when using large-scale synchronous SGD.

3.3. Large-scale training using synchronous data-parallel SGD

When performing distributed training at scale, the global batch size (i.e. examples seen in one iteration by all of the workers combined) can become very large, which potentially causes accuracy and generalization issues. In mid-2017 Facebook released a paper that gave some simple recipes on how to tackle the fact that networks trained in a large-batch regime have less generalization power than similar networks trained in a small-batch regime [138]. Actually, up to a global batch size of 8192 for training a ResNet-50 network on the ImageNet dataset, Facebook concluded that it is not necessarily a generalization issue, but more of an optimization one.

One of the techniques described in this paper is the momentum-corrected warmup scheme and the linear learning rate scaling rule. This combination allows the use of a large learning rate that allows for better optimization, while at the same time avoiding a potential divergent behavior if large learning rates are used from the start. The momentum-corrected warmup scheme does momentum correction after changing the learning rate and provides more stability. The Facebook paper presents two types of warmup, constant and gradual. In constant warm-up, the model is trained with a small learning rate for a few epochs and then the learning rate is increased by a constant for each epoch, untill it reaches k times the original learning rate. In gradual warm-up, training starts with a small learning rate which is then gradually increased.

There are several regularization techniques proposed in the literature [139] [140] that allow increasing the batch size even more, without the need for additional training epochs and without degrading the generalization abilities of the model. One such technique, Batch Normalization, computes statistics across the samples in a mini batch. When we calculate the losses of samples across multiple nodes, it is important that a loss of a sample is independent of the other. But when you use batch normalization the loss function for a sample is no more independent of others. The mini batch statistics that is computed is a key component of the loss. If the per worker batch size is changed, it affects the underlying loss function being optimized. Hence, when using Batch Normalization, it is suggested to keep the batch size same across all worker nodes. Regardless of whether or not regularization techniques are used, the maximum effective batch size and hence the minimum number of iterations per epoch seems to be dataset-dependent.

Besides these algorithmic optimization difficulties, depending on the network topology and dataset size, hardware bottlenecks may also be encountered. The most common bottlenecks are:

3.4. Hardware bottlenecks in distributed training

- I/O contention when large data sets, or data sets with many small examples are used and the data has to be physically stored on shared storage facilities.

•Communication bottlenecks during gradient-aggregation. Such bottlenecks are most likely when there is a high ratio between the number of parameters and amount of computation.

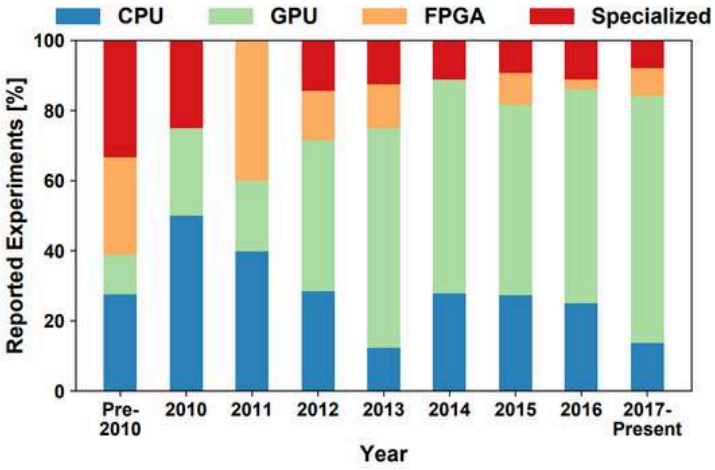
•Memory bottlenecks for large networks, particularly when using memory-limited GPUs.

Deep learning frameworks tend to have various approaches to avoid such bottlenecks, such as aggregate file formats (e.g. TFrecords [38]) or specialized gradient aggregation schemes (e.g. Horovod [10] uses the NCCL library to perform gradient aggregation on GPUs).

4. Hardware

Deep learning can be performed on various types of hardware, such as CPUs, GPUs, FPGAs or specialized hard- ware (e.g. TPUs). As can be seen from Figure 8, GPUs have become more popular for deep learning over the past decade. In this section, we will discuss the pro's and con's of various hardware for the purpose of deep learning.

Figure 8. Use of hardware for deep learning from over time.

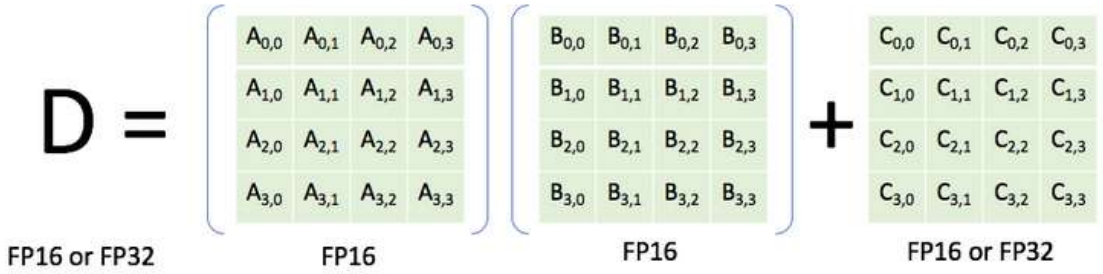


Source: [137].

4.1. NVIDIA Pascal / Volta GPUs

The NVIDIA Pascal architecture introduced support for reduced precision arithmetic, adding half-precision float- ing point support (FP16) to the basic streaming multiprocessor (SM) design [41]. This allowed for reduced pre- cision neural network training at twice the throughput of regular FP32 training. Moreover, using FP16 to store the neural network weights and activations is particularly beneficial for GPU-based training in general, as these devices are typically more memory-constrained, with access to 16GB of device memory for the P100 GPU. This allows the training of networks that are roughly twice the size of FP32 ones. Other innovations introduced with this architecture are the second generation high-bandwidth memory (HBM2) that allows for much higher memory bandwidth compared to the previous GDDR approaches, as well as the NVLink interconnect that allows five times faster communication between the GPUs compared to PCIe 3.0. The peak performance of the NVIDIA P100 is around 10 TFLOPS (tera floating point operations per second) with FP32 and 20 TFLOPS with FP16. These innovations continued with the Volta architecture that added even more domain-specific hardware. Ar- guably the feature most well-received by the deep learning community was the addition of the so-called Tensor Cores. These are well suited to the convolution and matrix-multiply operations that are heavily present in neur- al network training [42], as they can perform very fast FMA (fused multiply-add) operations on 4x4 matrices (Figure 9). Most deep learning convolutional layers can be transformed into this computational pattern. The peak performance of the V100 GPU reaches 15 TFLOPS of FP32, but with the Tensor Core functionality it can achieve eight times this rate, at 125 TOPS (tera operations per second). Moreover, the memory capacity of the V100 dou- bled to 32GB, allowing training of models twice as big. More information on how to use the FP16 and Tensor Core functionality with various Frameworks can be found in NVIDIA's Deep Learning Software Development Kit (SDK) documentation [43].

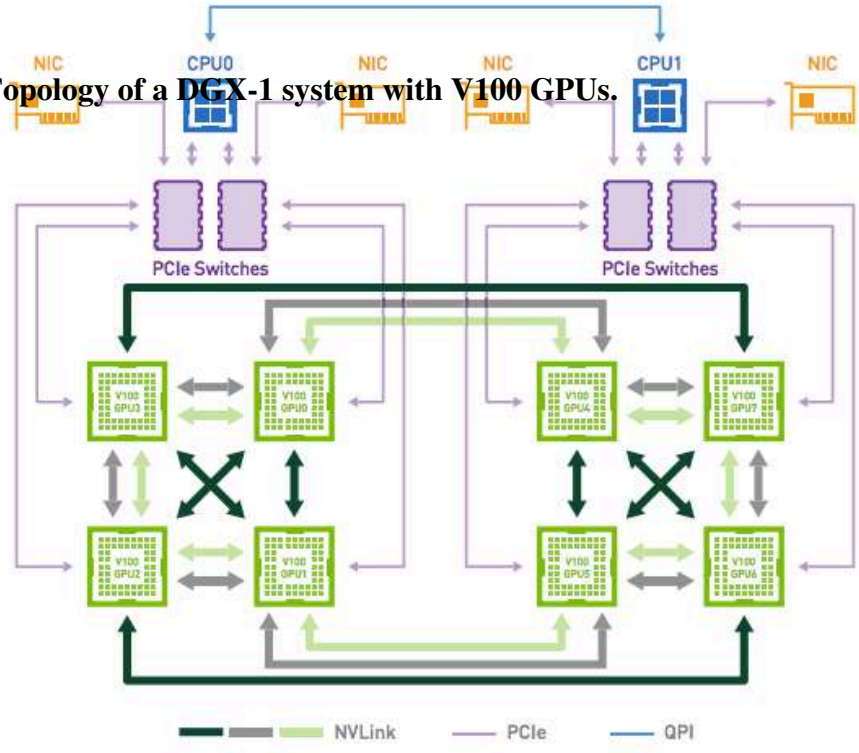
Figure 9. Mixed-precision fused multiply-add (D = AxB+C) operation supported by the NVIDIA Tensor Cores.



The memory bandwidth has also increased, peaking at around 900 GB/s, and so has the NVLink connectivity: NVLink 2.0 consists of six bidirectional links of 50 GB/s each, allowing for an aggregate bandwidth of 300 GB/s.

A reference architecture of a DGX-1 system equipped with V100 GPUs is depicted in Figure 10. This system has a peak performance of around 1 PetaFLOP of FP16 arithmetic. In this toplogy, the (direct) links between e.g. GPU 0 and GPU 1 in this system provide an aggregate bandwidth of 100 GB/s between these GPUs and the (direct) link between GPU 0 and GPU 3 provides 50 GB/s. Thus, while the specifications for NVLink 2.0 commonly (only) quote the aggregate of 300 GB/s, it should be realized that the bandwidth between pairs of GPUs is (typically) smaller.

Figure 10. Topology of a DGX-1 system with V100 GPUs.



Source: [44].

Low level libraries (often written in some form of assembly language) are an important component in exposing the full performance of an architecture to regular programmers. Nvidia provides several such libraries for use with their GPUs [45], some of which have been around for a long time and are very mature (e.g. cuBLAS has been around since 2007). Libraries that are particularly relevant to the deep learning community and that are also commonly used by the frameworks discussed later on in these guide are:

•cuBLAS, a GPU-accelerated library for basic linear algebra [46] (similar to the BLAS library [47]).

•cuSPARSE, a GPU-accelerated library for sparse matrices algebra [48].

•cuDNN, a GPU-accelerated library of primitives for deep neural networks (DNNs) [5].

•NCCL, the NVIDIA Collective Communications Library, for scaling applications across multiple GPUs and nodes [16].

4.2. AMD Vega / Vega20

Deep learning is one of the applications specifically targeted by AMD's Radeon Instinct accelerators product line [49]. Hence, the architecture of this product line contains a number of features specifically beneficial for deep learning.

The Radeon Instinct MI25, based on the Vega architecture, introduced support for half precision (FP16) computations in the hardware, allowing for two times higher throughput compared to FP32. Additionally, it offers 16 GB of HBM2, allowing for a total throughput of 484 GB/s. The MI25 is connected to the motherboard by a PCIe 3.0 x16 (15.75 GB/s) bus.

The newer Radeon Instinct MI50 and MI60, based on the Vega20 architecture, introduced support for INT8 computations in the hardware, allowing for four times higher throughput compared to FP32. The bandwidth of the HBM2 memory has been increased to 1024 GB/s for both models, and the amount of memory has been increased to 32GB for the MI60. Both cards support PCIe 3.0 x16 (15.75 GB/s) and PCIe 4.0 x16 (31.51 GB/s), and additionally offer two Infinity Fabric links (100 GB/s each) through which up to 4 GPUs can be connected in a ring configuration.

Libraries and tools to support usage of the Radeon Instinct accelerators for deep learning are mostly provided through AMDs involvement in the Radeon Open eCosystem (ROCm) platform [50].

A few notable tools/libraries are:

- Heterogeneous-computing Interface for Portability (HIP): a C++ dialect that was designed to ease conversion of CUDA applications to portable C++ code [52].
- Heterogeneous Compute Compiler (HCC): a C++ dialect with extensions for launching kernels and managing accelerator memory [51].
- MIOpen: a library that contains optimized machine learning primitives based on the OpenCL or HIP programming models [53].
- rocBLAS: AMD's library for BLAS on ROCm [54].

Practical performance not only depends on the theoretical capability of the hardware, but also on the supporting software stack. It should be realized that, since Nvidia's and Intel's deep learning related libraries have been around for longer, these are generally well supported by major deep learning frameworks. ROCm and the MIOpen library are (relatively) new and limited adaptation by current frameworks may therefore cause poor performance of such frameworks on AMD hardware.

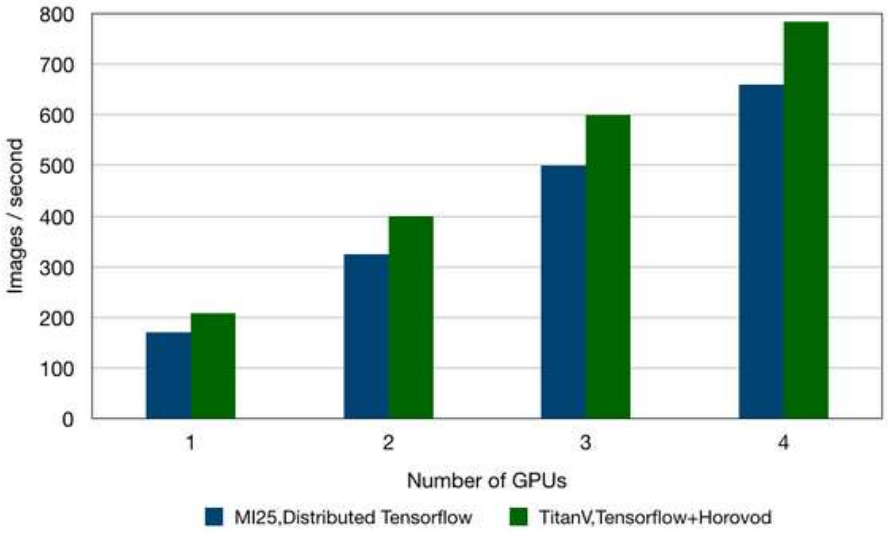
The ROCm community has created forks of popular frameworks such as TensorFlow, Caffe 2, PyTorch, MxNet and CNTK in order to stimulate integration of MIOpen support in these frameworks [55]. The intention of the ROCm community is to get this support integrated into the official frameworks. As long as that is not the case, using one of these forks instead of the original framework may provide better performance when running on AMD hardware.

As an example, Figure 11 compares the throughput (in images/second) of the MI25 and V100 architectures training a ResNet-50 model on synthetic data. The training on the MI25 was distributed using TensorFlow's native distribution layer, and used the TensorFlow docker container provided by ROCm [56]. The training on the Titan

V used Horovod as the distribution layer, and was run using an optimized manual build of TensorFlow with CUDA 10. The throughput was slightly lower for the AMD MI25, with a difference that is of similar order to the difference in theoretical FP32 performance. This shows that, despite the difference in maturity of the AMD and NVIDIA software stacks, a decent performance can be obtained provided that the ROCm-provided forks of the frameworks are used [57].

Note that this benchmark was run in (pure) FP32 in order to compare the software stacks: in a real-world scenario, one would probably try to exploit the Tensor Cores of the V100.

Figure 11. Deep learning performance comparison of MI25 (Vega architecture) and Titan V (V100) GPUs for the ResNet-50 model trained on synthetic image data in FP32.



4.3. Intel Xeon Scalable Processors

Intel Xeon Scalable Processors [58], codenamed Skylake, introduced support for AVX-512 instructions. This enables higher FMA throughput compared with AVX2 instructions supported by the previous general-purpose Xeon architectures. The Xeon Bronze, Silver family and part the Xeon Gold family have one AVX-512 unit per core, whereas the higher tier Xeon Gold and Xeon Platinum family have two AVX-512 units per core.

Unlike the GPUs from the previous section, Xeon Scalable Processors do not yet offer any benefits with regards to reduced-precision (FP16) training. However, AVX-512 has boosted Intel's single precision performance to around 3 TFLOPs per CPU for a Xeon Platinum 8180 (the top tier model). Thus, a dual socket system would achieve about 6 TFLOPs, approximately 20-30% of the performance of a GPU. The major advantage of CPU-based training is the fact that memory capacity is no longer a big constraint. Current dual-socket servers can be equipped with more than 1TB of system memory per node, allowing one to train very large models.

Intel offers the following low level libraries relevant to deep learning:

- The Intel MKL-DNN library, which contains implementations of common deep learning primitives that are optimized for use on Intel hardware. This library is supported by Tensorflow, PyTorch and (Intel) Caffe [6].
- The Intel Math Kernel Library (Intel MKL), which contains optimized implementation of BLAS, LAPACK and FFTW functions for Intel hardware [59].

4.4. AMD Zen

The AMD EPYC processor series contains the server processors from the Zen architecture [60]. The Zen architecture supports the AVX2 instruction set, but not the AVX-512 instruction set. A single top tier EPYC processor, the EPYC 7601, has a single precision floating point performance of about 1.1 TFLOPS [61] [62]

For the deep learning community, AMD mostly markets the Zen processor in combination with the Radeon Instinct accelerators. In other words: as a CPU to feed data to an accelerator (where the training is then performed), not as an architecture to also perform the training on.

AMD offers several general low level libraries [63]:

•BLIS: a BLAS implementation, optimized for the AMD EPYC processor family.

• libFLAME: a portable library for dense matrix multiplications, similar to LAPACK.

•AMD LibM: basic math functions optimized for x86-64 processor-based machines.

Although these generic libraries could potentially be used to accelerate training on AMD CPUs, they are not
commonly supported by deep learning frameworks. Also, AMD does not offer a specific library that implements optimized deep learning primitives.

More information on AMD EPYC can be found in the associated PRACE Best Practice Guide [64].

4.5. Choosing your architecture

An important factor in choosing your architecture for deep learning tasks will be the expected performance. How- ever, a one-to-one comparison of theoretical compute performance between different architectures is often diffi- cult to make. For example, current CPUs can often boost clock frequencies even if all cores are in use, but this depends on the workload and the amount of heat the workload generates. Thus, one may wonder, if it is fair to calculate peak performance using such clock frequencies.

Nevertheless, it is good to have an overview of the theoretical performance of the different architectures. Table 1
provides an overview of the theoretical performance of current-day top end model CPUs and GPUs. Note that it compares the performance of single GPUs / CPUs, but many HPC systems have multiple CPUs/GPUs per node.

As is clear, GPU architectures provide a high performance for floating point arithmetic, but have a limited amount
of memory available and the rate at which data can be moved from CPU to GPU memory may pose a limit. While the memory bandwidth for GPUs may seem a lot higher, this must be seen relative to the amount of FLOPS, if one has more floating point units, a higher bandwidth is needed to keep them occupied.

Table 1. Performance overview of various CPU and GPU architectures.

Architecture	INT8 [TOPS]	FP16 [TFLOPS]	FP32 [TFLOPS]	FP64 [TFLOPS]	Memory [GB]	Memory Bandwidth [GB/s]a	PCIe [GB/s]	Proprietary Interconnect [GB/s]a
AMD Instinct MI25 [65]	-	24.6	12.29	0.77	16	484	15.75	-
AMD Instinct MI50 [66]	53.6	26.8	13.4	6.7	16	1024	31.51	200 (2 x 100)
AMD Instinct MI60 [67]	58.9	29.5	14.7	7.4	32	1024	31.51	200 (2 x 100)
NVIDIA P100b [68] [69] [71]	-	21.2	10.6	5.3	16	732	15.75	160 (4 x 40)
NVIDIA V100b [71] [70] [72]	62.8c	31.4c / 125d	15.7	7.8	16 / 32	900	15.75	300 (6 x 50)

Architecture	INT8 [TOPS]	FP16 [TFLOPS]	FP32 [TFLOPS]	FP64 [TFLOPS]	Memory [GB]	Memory Bandwidth [GB/s]	PCIe [GB/s][a]	Proprietary Interconnect [GB/s][a]
Intel Xeon Scalable 8180 (per socket) [73] [74]	-	-	3.0 / 4.2[e]	1.5 / 2.1[e]	768 (max)	119 (max)	15.75	-
AMD EPYC 7601 (per socket) [61] [62]	-	-	1.1 / 1.4[f]	0.56 / 0.69[f]	2000 (max)	159	15.75	-

[a] PCIe bandwidths are unidirectional. NVlink speeds are aggregated bidirectional bandwidth. For Infinity Fabric (IF) the bandwidth for one Infinity Fabric link is quoted; it is unspecified whether this is unidirectional, or aggregated bidirectional bandwidth.

[b] Performance quoted here is for the SXM2 form factor, the PCIe edition has slightly different specifications.

[c] Only quoted in unofficial documentation. In practice, the FP16/FP32 tensor operations in Figure 9 have higher throughput on the Volta architecture and are therefore preferred.

[d] Only for FP16 tensor operations as shown in Figure 9.

[e] Based AVX512 operations at stock frequency (1.7 GHz) / all core boost frequency (2.3 GHz).

[f] Based AVX2 at stock frequency (2.2 GHz) / all core boost frequency (2.7 GHz).

So, how do you choose your architecture? There is no generic answer to this question, as it depends very much on the specific situation. In particular, it may depend on how much effort you are willing to spend on optimizing your run: GPUs may have a higher floating point performance, but it is often more difficult to keep all of those floating point units ‘occupied’.

In choosing your architecture, there are some general aspects to consider:

•Model size and type of parallelism: does your model fit in GPU memory? If not, one could

- tune the model e.g. by reducing its connectivity (if that is acceptable) to fit the hardware,

•use model parallelism to distribute the model over multiple GPUs or

•use a data parallel approach on CPUs (if the model does fit in CPU memory).

Note that the increase in communication for a model parallel GPU approach could mean that it becomes hard to fully exploit the GPUs floating point performance, in which case a switch to CPUs might be more efficient.

•Data size: does the dataset fit in GPU memory? If not, you may have to put in a lot of effort to ensure the data arrives at the GPU in time. Depending on how challenging that bottleneck proves to be, it may be faster and/or easier to use CPUs.

•Availability of resources: If you have access to a system with a few GPU nodes, but thousands of CPUs, that could affect your decision.

•Development stage: during development, you might run a limited number of examples (potentially even with a smaller model), e.g. to check code integrity. This could well fit in a single GPU, which allows quick development cycles. Your production run, with all examples and your full model, may require CPUs because of memory constraints.

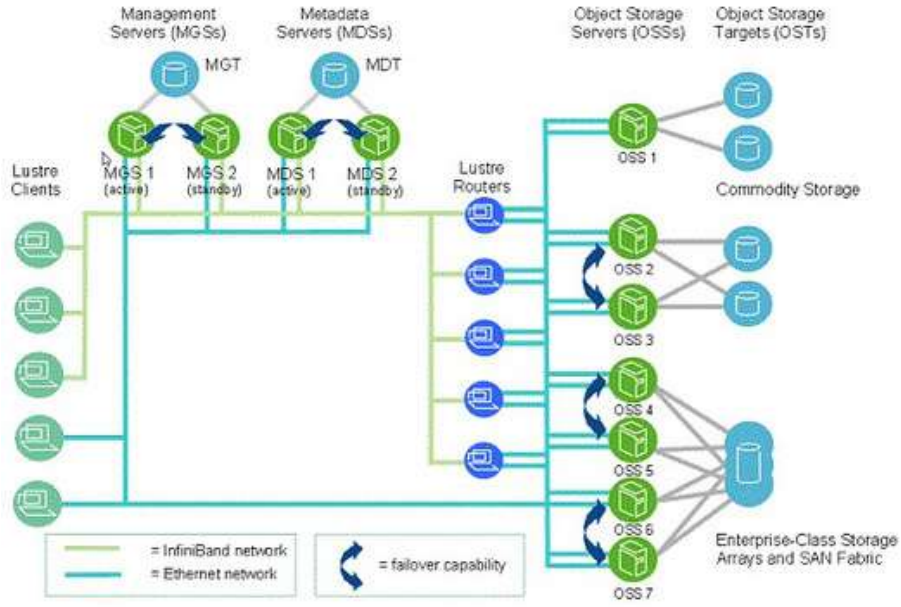
4.6. I/O considerations

As mentioned in most of the use-cases presented later in this guide, I/O tends to be an important concern for large- scale distributed deep learning. This section will illustrate potential causes of I/O bottlenecks and describe generic strategies to alleviate these.

Most supercomputers around the world are operated as shared machines, and they usually offer users access to a fast, parallel, shared filesystem. Typically, this is either Lustre or GPFS storage.

In the case of Lustre, there is the concept of object storage target (OST) and metadata servers (MDS). OSTs store the actual data, and MDSs store the links to the data. Typically, there are hundreds of OSTs, but much less MDSs (see Figure 12).

Figure 12. A diagram of a Lustre filesystem.



Source: [75].

GPFS Clients can access the storage via the Network Shared Disk (NSD) protocol. Worker nodes will be NSD clients and, as with Lustre, will see and access a coherent, synchronised namespace. Unlike Lustre, GPFS allows all nodes of the cluster to perform metadata operations. Therefore a separate metadata server is not required and so is not a single point of failure.

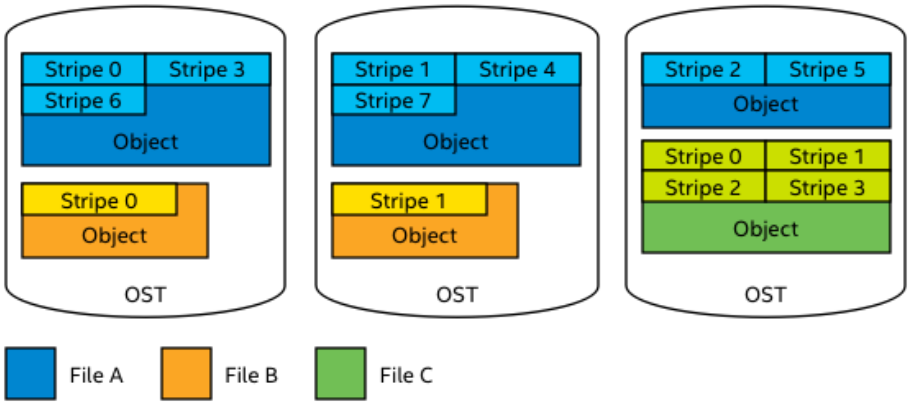
In computer vision, datasets are typically organized as a collection of files, distributed across a collection of directories. In the case of ImageNet-1K there are 1.28 million images used for training, distributed across 1000 different folders. When performing training, each image file is read around 90 times, leading to around 115.2 million images being processed. The order in which images are read is typically random. When performing this process distributed across many nodes, the MDSs in the case of Lustre tend to be overwhelmed by the amount of small-file access, and this usually leads to severe performance bottlenecks. Note that for deep learning tasks, the write performance of the filesystems is often less important, as the only data that is written are model checkpoints.

In order to get around this MDS limitation, most frameworks provide a database format that can be used to pack data efficiently and significantly reduce meta-data access. The Caffe framework uses the Lightning Memory-Mapped Database (LMDB) format [76], while Tensorflow uses TFRecords [38]. In the case of LMDB, the training data is packed usually inside a single-file, whereas in the case of Tensorflow there are still hundreds/thousands of TFRecords. One does not necessarily need to use these framework-specific file formats to pack multiple samples in one file. Other data formats such as the Hierarchical Data Format (HDF, commonly used in HPC and supported by several deep learning frameworks) that allow aggregation of multiple samples will provide similar performance benefits.

With these constructs, the MDS can often be eliminated as the bottleneck. However, one should be aware that the OSTs can still present a bottleneck in terms of bandwidth and latency, especially when using distributed training. There are multiple possible ways to resolve this bottleneck, depending on the size of the dataset. If the dataset is not that large and does fit into the system memory of each compute node (or SSD/local storage), then copying the entire dataset to each of the compute nodes prior to the actual training run should be the preferred solution. All accesses in subsequent epochs will then be local to each node (including the randomization of inter-epoch sample reads), and most of the Lustre contention will disappear. ImageNet-1K falls into this category, as the dataset size is around 42GB. If the dataset is much larger than the system memory/local storage, the most popular option is

to keep the data (TFRecords/LMDB/etc.) on the parallel filesystem itself. When doing so, it is important to stripe the dataset across a large number of OSTs with the Lustre striping commands, in order to allow reading in parallel from multiple OSTs (see Figure 13). This is particularly useful for very large files as is the case for LMDB.

Figure 13. Schematic illustration of how Lustre striping distributes files over multiple OSTs.



Source: [77].

GPFS implements striping in the file system, and hence striping is not a user-managed functionality. Managing its own striping affords GPFS the control it needs to achieve fault tolerance and to balance load across adapters, storage controllers, and disks.

However, even with these workarounds, for some deep learning tasks I/O is still the dominant bottleneck, and more efficient schemes need to be developed. This is currently an area of open research [142] [143].
More information on efficient usage of parallel file systems can be found in the associated PRACE Best Practice Guide [78].

5. Frameworks

This chapter provides a general overview of popular frameworks for deep learning, with a focus on the distribution and optimization aspects of those frameworks.

5.1. Caffe

Caffe [11], originally developed by the Berkeley Vision and Learning Center (BVLC) and by community contrib- utors, was one of the first successful deep learning frameworks. It was applied to various problems, particularly in the Computer Vision field. Although Caffe has both a C++ and a Python interface, these interfaces are only needed if custom layers are required. C++ or Python programming skills are not necessary for experimenting with various models or datasets, as users can specify these using human-readable protocol buffers (called prototxt files) [12].

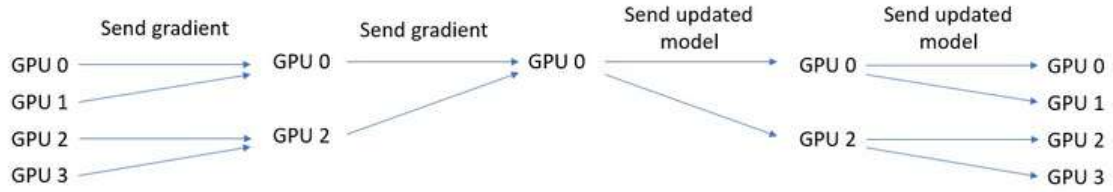
By default, Caffe does not support distributed training, however, both Intel [13] and NVIDIA [14] have forked this repository and have released their own versions (see sections “IntelCaffe” and “NVCaffe”) with optimized low-level kernels as well as distribution layers that support multi-node training.

Although BVLC Caffe does not support multi-node training, multi-GPU training is supported, albeit only via the C/C++ paths. This can be achieved using the --gpu flag when running the training:

```
build/tools/caffe train --solver=models/bvlc_alexnet/solver.prototxt
--gpu=0,1
```

Quoting the Caffe documentation [15]:
“The current implementation uses a tree reduction strategy. e.g. if there are four GPUs in the system, 0:1, 2:3 will exchange gradients, then 0:2 (top of the tree) will exchange gradients, 0 will calculate updated model, 0=>2, and then 0=>1, 2=>3. For best performance, P2P DMA access between devices is needed. Without P2P access, for example crossing PCIe root complex, data is copied through the host and effective exchange bandwidth is greatly reduced.”
This suggest a hierarchical update scheme like depicted in Figure 14 is used.

Figure 14. BVLC tree reduction scheme for gradient aggregation and model updating for multi-GPU training.

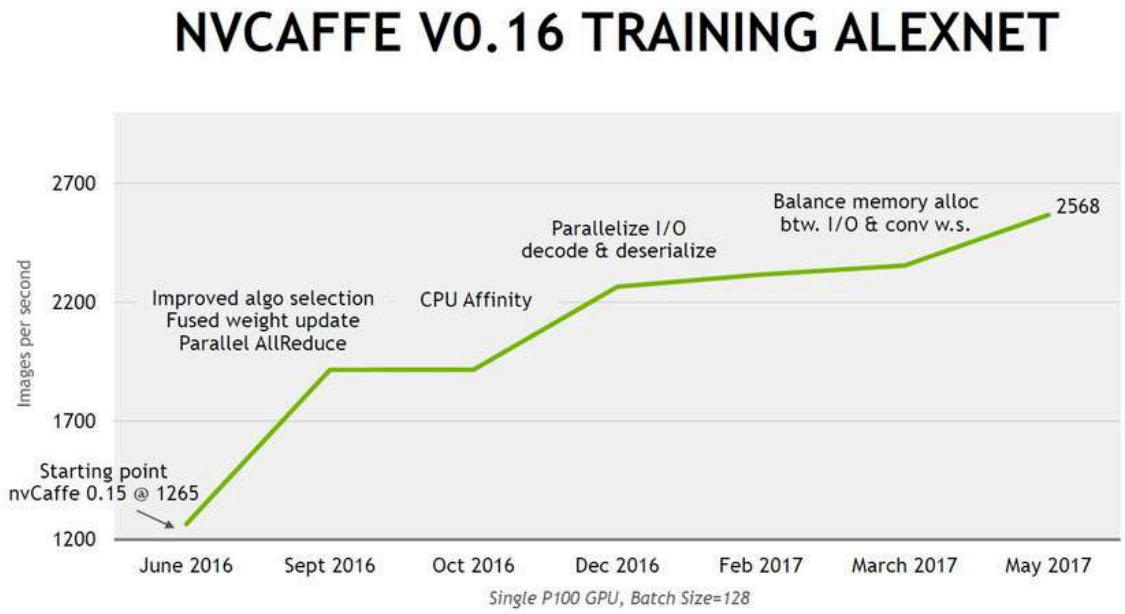


5.2. NVCaffe

NVCaffe [14] is an NVIDIA-maintained fork of BVLC Caffe tuned for NVIDIA GPUs. It features integration with both NVIDIA cuDNN’s [5] efficient primitive library, as well as with NCCL’s communication primitives [16]. This is particularly suited for multi-GPU configurations. NVCaffe also includes mixed precision support making full use of the Pascal and Volta architectures. NVCaffe is also offered as a pre-built, tested, ready-to-run Docker/Singularity container that includes all of the necessary dependencies [17]. As NVCaffe is mostly concerned with performance, most other tasks (e.g. dataset preparation, etc.) are done similarly as in the BVLC Caffe.

As can be seen in Figure 15 below, NVIDIA has doubled the performance of Caffe in about one year, compared to the original BVLC Caffe baseline.

Figure 15. Performance improvement of nvCaffe training AlexNet in 2016/2017.



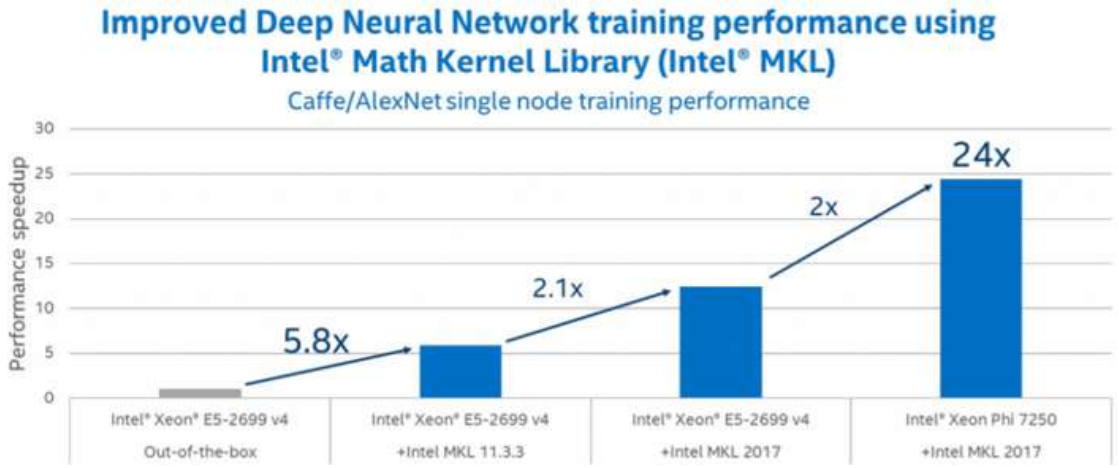
Source: NVIDIA [18].
As of the 0.17.1 release, NVCaffe supports training on multiple nodes using OpenMPI version 2.0 protocol, however, you cannot specify the number of threads per process because NVCaffe has its own thread manager (currently it runs one worker thread per GPU). As mentioned in the NVCaffe release notes [19], multi-node multi-GPU training can be invoked with

```
dgx job submit --name jobname --volume : --tasks 48 --clusterid --gpu 8
--cpu 64 --mem 480 --image --nc "mpirun -bind-to none -np 48 -pernode
--tag-output caffe train --solver solver.prototxt --gpu all >>
/logs/caffe.log 2>&1"
```

5.3. IntelCaffe

Another popular fork of the BVLC Caffe is the Intel-optimized version [13]. This fork has tight integration with two essential Intel-optimized libraries: MKL-DNN [6] and MLSL (Machine Learning Scaling Library [20]). MKL- DNN speeds up the convolution, matrix-multiplication, pooling, activation functions and other compute primitives used in deep neural network training and inference, whereas MLSL provides efficient communication primitives. Some single-node performance numbers compared to the original BVLC Caffe can be found in the Figure 16 (for training and inference).

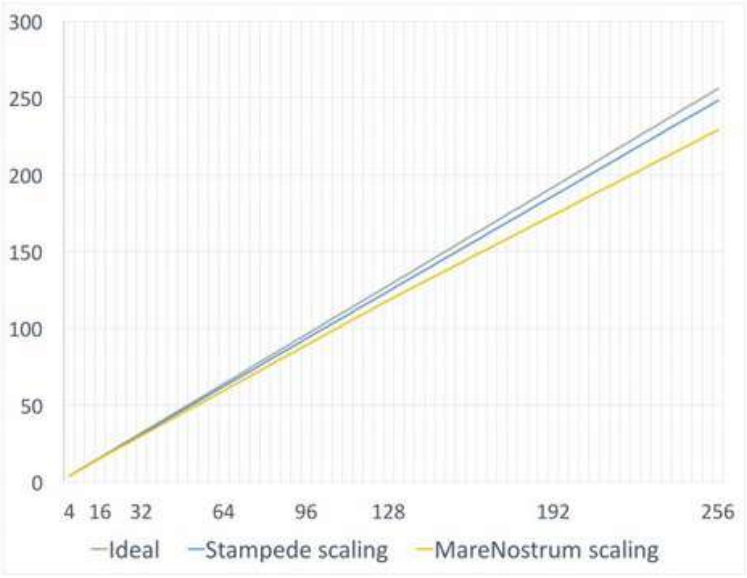
Figure 16. Performance improvement for IntelCaffe over BVLC Caffe on Intel hardware.



Source: Intel [21].

Intel's MLSL (Machine Learning Scaling Library) is a software library that efficiently deals with the communication involved in neural networks. It is tightly coupled to the training framework, allowing simultaneous compute and communication when performing the backward propagation pass. MLSL allows for both data and model parallelism, and does this while efficiently using the bandwidth offered by high-speed interconnects such as Intel's Omni-Path Architecture [22]. When using model parallelism the model is divided across the participating workers. This has the potential to accommodate larger models, as each worker will hold a part of the parameter set, but comes at the cost of additional communication, since workers have to communicate both in the forward and backward passes. In the case of data parallelism, the parameters are replicated on each worker. Thus, the size of the model is limited to the size of one node's memory, but communication happens only in the backward pass. MLSL gives users a lot of flexibility, such as setting the number and the affinity of the communication cores. Also, since it is relying on Intel MPI [23], the communication algorithms used for the AllReduce communication patterns can be selected (through the I_MPI_ADJUST environment variables [24]). These AllReduce patterns are heavily used in data-parallel distributed deep learning. By efficiently setting these parameters, users can expect scaling efficiency in excess of 90% when using MLSL on production-grade supercomputers (with either Knights Landing or Skylake nodes), as illustrated in Figure 17.

Figure 17. IntelCaffe scaling efficiency (speedup vs number of workers) on the Stampede2 [25] and MareNostrum [26] supercomputers.



Source: [140].

Most HPC systems are multi-socket systems. Each CPU can access all memory, but accessing the memory from the other socket takes more time. This is known as a Non-Uniform Memory Access (NUMA) architecture. In order to get the best performance, users should make sure that processes are not moved between sockets during runtime. That way, processes only have to access the memory of their own socket. Moreover, binding threads to individual cores may provide an even larger advantage, as keeping threads on the same core will reduce the number of cache misses.

MSLS allows the user to determine how many processes are started on a single node through the MLSL_NUM_SERVERS environment variable. Additionally, the MLSL_SERVER_AFFINITY variable allows the user to bind these processes to a particular core. Finally, one can set KMP_AFFINITY to determine the binding and set OMP_NUM_THREADS to determine the number of threads each Caffe uses. For example, for a 2 socket system, one can define

```
MLSL_NUM_SERVERS=2 MLSL_SERVER_AFFINITY="39,38,43,42"
OMP_NUM_THREADS=22 KMP_AFFINITY="granularity=fine,compact,1,0"
```

This will start two MLSL servers per socket, bind those processes to cores 39 and 38 (on socket 0) and 43 and 42 (on socket 1), allow each MLSL server to use 22 threads and bind each thread to a core (for more info on using KMP_AFFINITY, see [27]). Note that you should check the numbering of cores on your system using the lscpu or numactl –H commands and ensure that you distribute the MLSL servers equally over your sockets using the MLSL_SERVER_AFFINITY variable.

5.4. PyTorch

PyTorch [28] is a very popular open-source machine learning framework designed and maintained by Facebook. There are many interesting features in the PyTorch framework, however the most notable change is the adoption of a Dynamic Computational Graph. This capability is also referred to as “Define by Run” as opposed to the more conventional “Define and Run”.

Most DL frameworks maintain a computational graph that defines the order in which computations must be per-
formed. Thus, these frameworks typically use a language (e.g. Python) that sets up the computational graph and
an execution mechanism that is different from the host language. This somewhat unusual setup is motivated by

efficiency and optimization reasons. Since a computational graph fully specifies which computations will occur, the graph can be optimized and run in parallel on the target architectures (e.g. CPU/GPU).

Contrary to this static computation graph assumptions, dynamic computational graphs are valuable for situations where you cannot determine the computation a priori. An example of this are recursive computations that are based on variable data. Also, in the space of natural language processing, where language can come in various expression lengths, dynamic computational graphs are of high importance.

One other important feature of PyTorch is its versatility in performing distributed training. The distributed package included in PyTorch (i.e., torch.distributed) enables researchers and practitioners to easily parallelize their computations across processes and clusters of machines. To do so, it leverages the messaging passing semantics allowing each process to communicate data to any of the other processes. As opposed to the multiprocessing (torch.multiprocessing) package, which does not use message passing semantics and only allows distribution within a single node, torch.distributed processes can use different communication backends and are not restricted to being executed on the same machine.

The torch.distributed package supports both point-to-point and collective communication patterns, offering functions such as blocking/nonblocking send/receive, as well as reduce/all_reduce/broadcast/scatter/gather/etc, following the MPI communication interface [141].

One of the important aspects of torch.distributed is its ability to abstract and build on top of different backends [29]. There are currently four backends implemented in PyTorch: TCP, MPI, Gloo, and NCCL.

The Gloo backend provides an optimized implementation of collective communication routines, both for CPUs and GPUs. It is particularly suited for GPUs, as it can perform communication without transferring data to the CPU's memory using GPUDirect [33] (on Tesla-capable hardware). It is also capable of using NVIDIA's NCCL library to perform fast intra-node communication and implements its own algorithms for inter-node communication. The main difference between using Gloo with NCCL support, or the NCCL backend directly is that the latter was originally only for single node training. However, NCCL v 2.0 also introduces multi-node support, thereby blurring the line between these two approaches.

The Message Passing Interface (MPI) is the de-facto standard for communication in high-performance computing [141]. It allows point-to-point and collective communications and was the main inspiration for the API of torch.distributed. Several implementations of MPI exist (e.g. Open-MPI [30], MVAPICH2 [31], Intel MPI [23]) each optimized for different purposes. The advantage of using the MPI backend lies in MPI's wide availability and high-level of optimization – on large computer clusters. Some recent implementations are also able to take advantage of CUDA Interprocess Communication [32] and GPUDirect technologies in order to avoid memory copies through the CPU.

Unfortunately, the torch.distributed package, distributed training could be parallelism, although there is the most suitable solution if your model does not fit the memory of a single node.

Here, we discuss the native distributed TensorFlow; Horovod is discussed in a separate section.

There are two ways to run TensorFlow [34] in a distributed fashion: •Using the native distributed TensorFlow functionality [35]. •Using Horovod as a distribution layer for TensorFlow [10]. The choice between the two approaches is dependent on your situation. The main considerations are: •Horovod – scales well to a large number of nodes and has a strong focus on efficient training on GPUs. • If you already have serial TensorFlow code, Horovod only requires a handful of lines to be added. •Horovod was designed for data parallelism (although technically, model parallelism is possible [36]). Native distributed TensorFlow supports data parallelism and through the help of the Horovod library (also built on top of MPI) that is described in the section "Horovod".

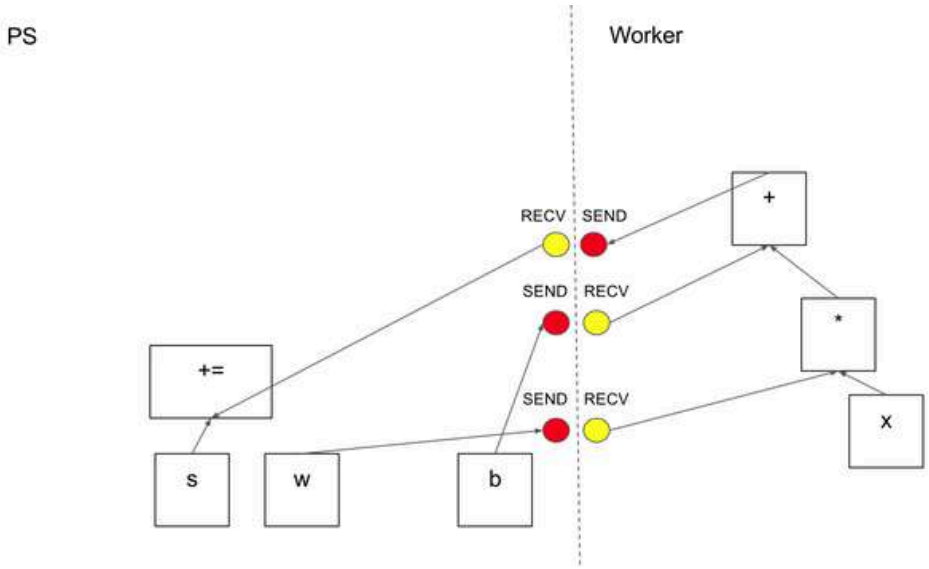
5.5. Tensorflow

5.5.1. Native distributed TensorFlow

The native distributed Tensorflow methods divides *tasks* (e.g. computations for one or more nodes in a graph) over different *jobs*. Typically, there are *worker* jobs that take care of compute intensive parts of the graph, while *parameter server* jobs store the model parameters (weights and biases) and do light computational operations. These *jobs* can be running on the same physical machine (e.g. a *parameter server* and *worker* on a single machine) or on different physical machines.

TensorFlow automatically takes care of communicating the correct variables, e.g. as depicted in Figure 18.

Figure 18. TensorFlow communication between Parameter Server (PS) and Worker.



Source: [37]. TensorFlow chooses optimized implementations for *send* and *recv* operations depending on the source and destination (e.g. if *PS* is running on a CPU and *worker* on a GPU in the same node, cudaMemcpyAsync() is used).

Since native distributed TensorFlow takes care of communicating the data from and to the right nodes, it is relatively easy for the programmer to distribute the model over multiple nodes (model parallelism). The advantage is that this allows one to run models that are too large for a single node's memory. A potential pitfall for the programmer is that he/she unintentionally creates code that performs a lot of communication, resulting in a lot of overhead and poor performance.

The centralized storage of the model in one (or a few) *parameter server* jobs means all of the communication goes through a limited set of nodes, potentially saturating their network connections. Although the number of nodes running *parameter server* jobs can be increased, scalibility of this approach (in terms of performance) is limited: there are more nodes that provide (part of) the model data, but it also increases the number of required *send* and *recv* operations.

5.5.2. IO in TensorFlow

To achieve the peak performance on your computing hardware, it is important that your processors receive the input data in an efficient way. The TensorFlow documentation contains a specific section on creating an efficient data input pipeline, that allows for concurrent computation and IO [9]. A number of key recommendations (all relating to functions from the tf.data.Dataset API) are:

•Use the *prefetch* transformation to overlap computation and IO. If the *prefetch* transformation is put at the end of the input pipeline (i.e. after any other potential transformations), also the preceding transformations are 'prefetched', i.e. done out of order with the actual training. This can be particularly beneficial when applying transformations on a CPU when the training is being run on an accelerator.

•Parallelize the *map* transformation to allow the input pipeline to use multiple threads.

- If your entire dataset can fit into memory, consider explicitly caching it using the *cache* transformation. If possible (i.e. the memory footprint allows it), cache the dataset after applying all other transformations.

•For very large batch sizes (hundreds of thousands) consider using the *map_and_batch* transformation to parallelize batch creation.

•Consider the order of *shuffle* and *repeat* operations if you apply both. If you apply *shuffle* before *repeat*, you may benefit from the fused *shuffle_and_repeat* transformation.

While creating an input pipeline can overlap IO and computation, it may also be important to reduce the IO time itself. In general, reading many small files is much slower than reading a single (or a few) big file(s). TensorFlow offers its own TFRecords file format, a record oriented binary file format [38]. If you have many small input files, and IO is a bottleneck in your application, consider converting your input files to the TFRecord file format [39].

5.5.3. Other performance considerations

The TensorFlow documentation has a specific section on performance that discusses topics like

•Performance considerations for RNNs.

•Building TensorFlow with Intel MKL DNN.

•Tuning MKL DNN settings for optimal TensorFlow performance.

•Differences between *channels_first* and *channels_last* configurations. *Channels_last* is the default in tensorflow, but *channels_first* is optimal for training on NVIDIA GPUs with cuDNN and/or training with Intel MKL.

•Checking / estimating if your input data pipeline is a bottleneck.

We will not repeat those here, but suggest you read the official documentation [40].

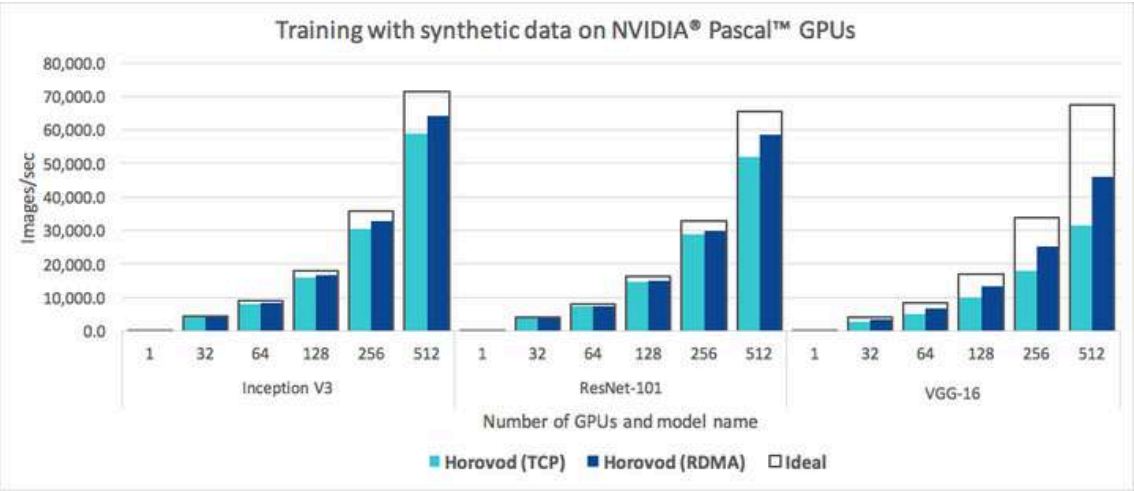
5.6. Horovod

Horovod is a distributed training framework for TensorFlow, Keras and PyTorch. It was designed from an HPC background and uses the MPI model for communication. Horovod was designed to

•Scale well to large numbers of nodes (Figure 19).

•Require minimal changes to the original (serial) TensorFlow/Keras/PyTorch code.

Figure 19. Horovod scaling on multiple GPU nodes.



Source: [10].

In an MPI program, the workload is divided over *MPI processes*. Each process runs the same code, except that the *MPI rank* is different among the processes. This rank is typically used to distribute the computational workload.

There are a minimal number of changes that the programmer will need to make to their initial, serial code, in order to parallelize it with Horovod:

- Initialize Horovod (assuming Horovod is imported as hvd: hvd.init())

•Optional: pin each MPI process to a physical device. For example, when using GPUs, it is typically most efficient to pin one MPI process per GPU (e.g. using hvd.local_rank() which returns the rank of an MPI process *within* a node).

•Distribute the training set. Typically, you can use the MPI rank (hvd.rank()) to make sure that each MPI process trains on a different part of the dataset.

•Scale the learning rate by the number of MPI processes (given by hvd.size() which returns the global MPI rank). This compensates for the increase in batch size that all MPI processes see together.

•Wrap the original optimizer in hvd.DistributedOptimizer. This distributed optimizer delegates gradient computation to the original optimizer, and uses so called *MPI collectives* such as *allreduce* (see Figure 20) or *allgather* to communicate and average gradients, and then applies those averaged gradients.

- Initialize variables consistently. This can be done by broadcasting one set of randomly initialized variables from (typically) rank 0 to all MPI processes, or by broadcasting the variables (also typically from rank 0) that were restored from a checkpoint. For if not using a TensorFlow use hvd.BroadcastGlobalVariablesHook(0), or alternatively, `tf.train.MonitoredTrainingSession, use hvd.broadcast_global_variables. For PyTorch, use hvd.broadcast_parameters.`
- If checkpoints are stored (e.g. if tf.train.MonitoredTrainingSession is used), make sure that only a single MPI process (typically rank 0) writes the checkpoints to avoid corrupting the checkpoint file.

Typically, between 5-10 lines of code need to be changed, compared to purely serial TensorFlow/Keras/PyTorch code.

Figure 20. Illustration of four MPI processes that perform an MPI AllReduce operation.

The use of MPI_AllReduce and MPI_AllGather routines gives Horovod a performance advantage over na- tive distributed TensorFlow in an HPC environment, as the implementation of these routines are gener- ally well-optimized one HPC systems. When computing on GPUs, one can set the environment variable HOROVOD_GPU_ALLREDUCE=NCCL to let Horovod use the NCCL library (which is optimzed for GPUs) as backend for the MPI_AllReduce routine.

Since Horovod uses MPI, your TensorFlow/Keras/PyTorch code needs to be launched just like any other MPI program. Typically, you do this using the mpirun command, though some batch schedulers may provide their own commands to launch MPI programs (e.g. srun for the SLURM scheduler).

For more specific information on how to adapt your code for Horovod and how to run it, please read the official documentation [10].

6. Use-cases

6.1. Predicting video frames for traffic camera's using GANs

Algorithm	GAN
Hardware	NVIDIA DGX-1 (8x Nvidia Tesla P100)
Framework	NVIDIA optimized PyTorch
Libraries	cuDNN
Scientific problem	Predict video frames

6.1.1. Research background

Prediction of traffic scenes in videos can help understand the behavior of traffic participants. Generative Adversarial Networks (GANs, implemented in Nvidia optimized PyTorch that is opensource and can be downloaded from Nvidia GPU Cloud) are used to predict the future frames of a video based on the previous frames of the same video. During GANs training, both the discriminator and the generator network have layers operating on different scales (frame resolutions). The GANs can predict six future frames at once, based on six previous frames of resolution 128x128 pixels.

6.1.2. Computational problem

Table 2. Size of the computational problem.

Input data	32-bit float, [128 128 3 6]
Output data	32-bit float, [128 128 3 6]
Generator Network Parameters	131M (~2.1 GB) 81M
Discriminator Network Parameters	(~1.3 GB) 657 8 (1 per
Training sequence [a]	worker) 2900
Batch size	
# Epochs	

[a] Kitti Tracking Training Set [79]

The main parameters describing the computational problem are described in Table 2. Because of the long training time on a single GPU (~14 days) and because of the large amount of GPU memory needed for storing the networks and the video frames in each training iteration (Table 3)), multi-GPU training was performed on a DGX-1 system [80] containing 8 NVDIA Tesla P100 GPUs.

Table 3. GPU memory usage during training.

DGX main GPU [a]	~15.5 GB
DGX other GPUs	~12 GB
Single P100	~15.5 GB

[a] GPU with id=0, where the models and gradients are stored.

6.1.3. Parallelization

Since the DGX-1 is essentially a single node with 8 GPUs, parallelization can be easily done using the torch.nn.DataParallel module. This module parallelizes the application by splitting the input across the specified GPUs by chunking in the batch dimension. In the forward pass, the module is replicated on each GPU, and each replica handles a portion of the input. During the backwards pass, gradients from each replica are summed into the original module.

In our example, we first check if the current system has a GPU with the following code snippet to support both GPU and CPU based training (and also set the default tensor type accordingly):

```
if torch.cuda.is_available():
 use_cuda = True
 torch.set_default_tensor_type('torch.cuda.FloatTensor')
else:
 use_cuda = False
 torch.set_default_tensor_type('torch.FloatTensor')
```

Then, we wrap the model (in our example: the generator `self.G` and discriminator `self.D`) using nn.DataParallel.

```
if self.use_cuda:
 # If there is only 1 GPU in the node, all training data should go there
 if config.ngpu==1:

    self.G = torch.nn.DataParallel(self.G).cuda(device=0)
    self.D = torch.nn.DataParallel(self.D).cuda(device=0)
 else:
 # If there are multiple GPUs in the node, the DataParallel method takes
 # the device id's as arguments, and arranges distributing the training
 # data over the GPUs
    gpus = []
    for i in range(config.ngpu):
      gpus.append(i)  self.G  =  torch.nn.DataParallel(self.G,
      device_ids=gpus) self.D = torch.nn.DataParallel(self.D,
      device_ids=gpus)
```

Then, the model is put on the GPU using

```
if self.use_cuda:  self.G = self.G.cuda()  self.D = self.D.cuda()
```

Finally, assuming one has a DataLoader object data_loader the data can be moved to the GPU using

```
for data in data_loader:  input = data.cuda()
```

After this step, the data can be passed to the model, just as one would for a conventional (non-parallel) training.

6.1.4. Results

Table 4. GPU memory usage during training.

	Time to train	Speedup
1x NVIDIA Tesla P100	~14 days	-
8x NVIDIA Tesla P100	~5 days	2.8

This use case employed a very large model. Thus, gradient aggregation is costly and this has probably resulted in the limited speedup when going from 1 to 8 GPUs Table 4, "GPU memory usage during training.". Nevertheless, a time to train of 5 days is more reasonable when exploring the effects of (a small number of) hyperparameters.

6.1.5. Further references

More information on how to use torch.nn.DataParallel can be found in the PyTorch documentation [81]. The full code, out of which the above code snippets were taken, can be found online [82], as can the associated publication [144].

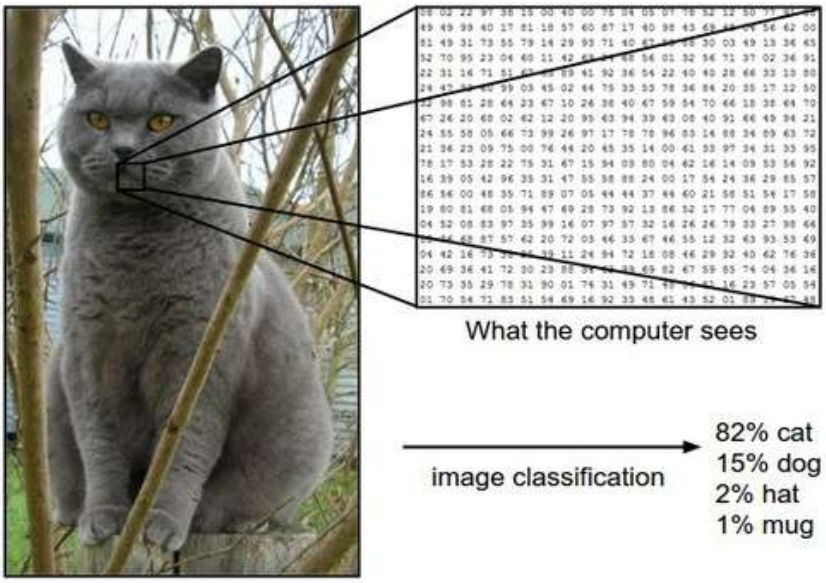
6.2. Large-scale Image classification on ImageNet-1k (CPU-based training)

Algorithm	CNN
Hardware	Intel Skylake
Framework	IntelCaffe
Libraries	MKL-DNN
Scientific problem	Image Classification

6.2.1. Research background

Image classification is the task of assigning an input image one label from a specific set of categories (Figure 21). This is one of the most important problems in Computer Vision and has a large number of potential applications. Being a fundamental Computer Vision problem, the core components of an image classification system can be extended to other Computer Vision tasks such as segmentation or object detection.

Figure 21. An image classification task from a computer perspective.



Source: [83].

6.2.2. Computational problem

Table 5. Size of the computational problem.

Input data	32-bit float, [224 224 3]
Output data	32-bit float, [1000]
# Weights	~25M
# Training examples	~1.28M
Batch size	14,336 (16 per worker)
# Epochs	90
# nodes / # workers	448 / 896

In this use case, we train a 50-layer neural network, ResNet-50, on 1.28 million examples to label images according to 1000 categories. The full computational problem is described in Table 5. Training such a network on a single

multi-core Intel Xeon Platinum 8160 CPU node (dual-socket node) would take ~400 hours, so distributed training is essential to reduce training time.

6.2.3. Parallelization

IntelCaffe coupled with Intel's MLSL (Machine Learning Scaling Library) is used for parallelization. It is impor- tant that IntelCaffe is linked against MKL-DNN in order to have efficient execution for the convolution (and other) operators). This parallelizes the training procedure by splitting the specified global batch size across the number of CPU nodes by chunking in the batch dimension (i.e. data parallelism).

We varied the number of Caffe processes per node and found that the hardware was used most efficiently when running two Caffe processes per node. This achieved relatively high throughput with a batch size of only 16 images
per worker. It is desirable to have efficient execution with small local batches, since when scaling out, the local batch is multiplied by the number of workers, and large global batches negatively impact training convergence.
Using the current settings, we get to a global batch size of 14336.

In the forward pass, the model is replicated on each worker, and each replica handles a portion of the input.

During

the backwards pass, gradients from each replica are aggregated using Intel's MLSL library (effectively doing very
efficient AllReduce operations) into the original model.

6.2.4. IO

With 90 epochs of 1.28M examples run on 448 nodes, each node passes approximately 250k examples. Each example is approximately 33 KB (42 GB / 1.28M examples). Since the complete run takes an hour, each node handles roughly 70 examples/s, or 70 x 33 = 2.3 MB/second. This should not be a bottleneck for any local storage. However, note that all nodes together read at an aggregated 1 GB/s, something that typical shared file systems will not be able to deliver.

Using local storage is almost always a good idea and is even essential for this example. The size of

ImageNet

LMDB is around 42 GB, which will generally fit on local storage.

6.2.5. Thread optimization

On systems with many cores, assigning dedicated cores for computation and for communication can improve performance. This can be achieved by setting the environment variables

```
MLSL_NUM_SERVERS=2  MLSL_SERVER_AFFINITY="39,38,43,42"  OMP_NUM_THREADS=22
KMP_AFFINITY="granularity=fine,compact,1,0"
```

- `MLSL_NUM_SERVERS=2` means that each Caffe process will have two cores that handle the gradient communication. Since we run two Caffe processes per node, this means a total of four cores per node are assigned to communication.

•MLSL_SERVER_AFFINITY="39,38,43,42" means that one process will use cores 39,38 for communication, and the other cores 43,42. The cores are selected such that the workload is distributed across the sockets.

•OMP_NUM_THREADS=22 means that each Caffe process can use up to 22 OpenMP threads.

•KMP_AFFINITY="granularity=fine,compact,1,0" is a setting from the Intel Thread Affinity Interface [27] that controls the binding of OpenMP threads to cores. This particular setting means that OpenMP* thread n+1 is bound to a thread context as close as possible to OpenMP* thread n, but on a different core. Once each core has been assigned one OpenMP thread, the subsequent OpenMP threads are assigned to the available cores in the same order, but they are assigned on different thread contexts.

6.2.6. Algorithmic adaptations for scaling

Very large global batch sizes, such as the ones used in this example, have a negative effect on the accuracy achieved. This is because the low number of times the gradient is updated during an epoch. Some adaptations can be made to mitigate this issue:

•Performing data augmentation typically helps in avoiding overfitting. Here, we outline the best-known methods for data augmentation on computer-vision problems such as ImageNet-1K. These are: random flip (mirror parameter), random crop (crop_size parameter), scale and aspect ratio augmentation (random_aspect_ratio_param parameter). In this use case, data augmentation was performed with

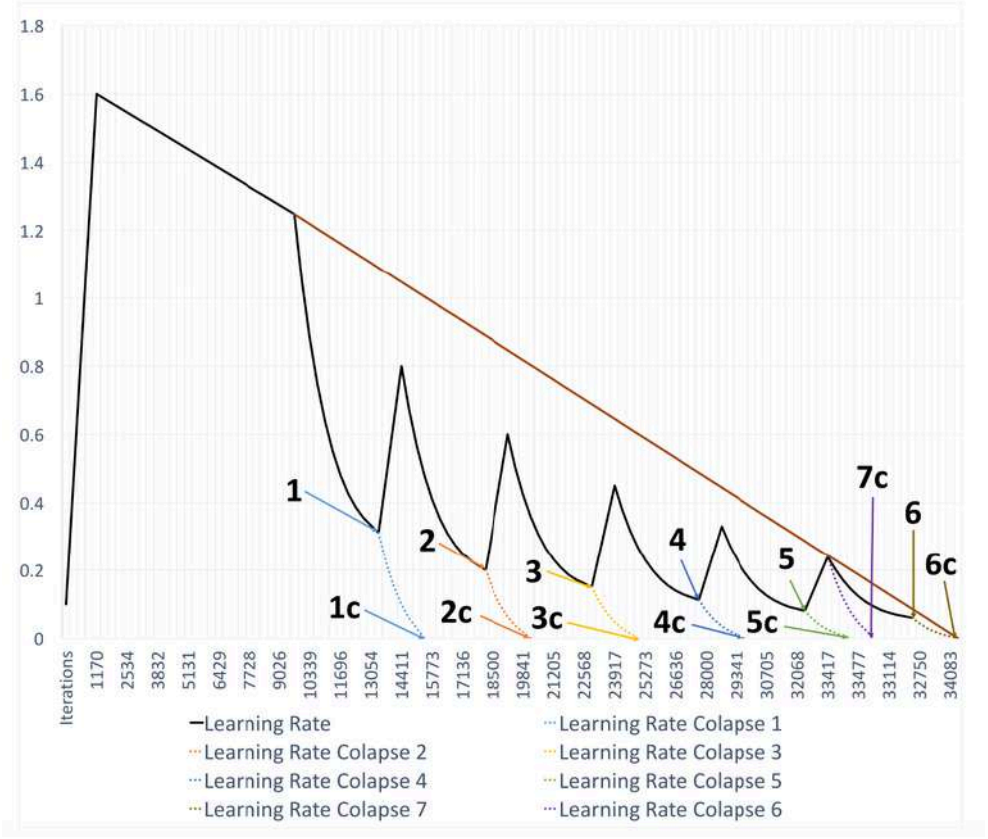
```
transform_param  {    mirror:
true   crop_size: 224   scale:
0.0078125    mean_value:  104
mean_value: 117   mean_value:
123
random_aspect_ratio_param {


    min_area_ratio: 0.08
    max_area_ratio: 1
    aspect_ratio_change: 0.75
    resize_param {
      interp_mode: CUBIC
    }
  }
}
```

•Batch normalization blocks are quite sensitive to large batches. In order to have more predictable results all batch normalization blocks in this use case have moving_average_fraction: 0.95.

•Gradual learning rate increase followed by polynomial decay. This procedure was described in more detail in [145].

•Collapsed decay, also described in more detail in [145].
Accuracy can be further improved, or the number of epochs (and hence time) to achieve the same accuracy can be reduced by using the cyclic learning rate schedule shown in Figure 22 (more details, see [145]).

Figure 22. Cyclic learning rate schedule.



Source: [145].

6.2.7. Training

Training is performed in two parts:

Part 1

•Warm-up for 5 epochs (446 iterations).

•Linear learning rate decay for the following 85 epochs (another 7594 iterations).

This is achieved by

```
MLSL_NUM_SERVERS=2 MLSL_SERVER_AFFINITY="39,38,43,42" OMP_NUM_THREADS=22
KMP_AFFINITY="granularity=fine,compact,1,0"
mpiexec.hydra -PSM2 -l -n 896 -ppn 2 -f hostfile -genv OMP_NUM_THREADS 22
-genv KMP_AFFINITY "granularity=fine,compact,1,0"
./build/tools/caffe train
--solver=models/intel_optimized_models/multinode/resnet_50_448nodes/
 solver.prototxt
```

`solver.prototxt` controls the training hyperparameters (number of warm-up iterations, number of decay iterations, learning rate, weight decay, etc.).

The resulting model achieves around 75.6%/92.7% top-1/top-5 accuracy. The process takes around 60 minutes.

Part 2

The model saved at epoch 85 is trained for 5 more epochs, in a collapsed fashion: scale/aspect ratio augmentation disabled, weight decay doubled, learning rate decay with a power of 2. We call this collapsed decay.

This is achieved by

```
MLSL_NUM_SERVERS=2 MLSL_SERVER_AFFINITY="39,38,43,42" OMP_NUM_THREADS=22
KMP_AFFINITY="granularity=fine,compact,1,0"
mpiexec.hydra -PSM2 -l -n 896 -ppn 2 -f hostfile -genv OMP_NUM_THREADS 22
-genv KMP_AFFINITY "granularity=fine,compact,1,0"
./build/tools/caffe train
--solver=models/intel_optimized_models/multinode/resnet50_448nodes/
 solver_collapse.prototxt --weights=resnet_50_448nodes_iter_7638.prototxt
```

`--weights=resnet_50_448nodes_iter_7638.prototxt` represents the fact that we load the model resulted fromthe section called “Part 1”, and that we continue the training processes from that generated set of weights.

The resulting model (resnet50_448nodes_coll_iter_402.caffemodel) achieves around 76.1%/93.2% top-1/top-5 accuracy. The process takes around 5 minutes.

```
I1024 08:35:26.892444 142045 solver.cpp:715]
Test net output #0: loss = 0.956108 (* 1 = 0.956108 loss)
I1024 08:35:26.893308 142045 solver.cpp:715]
Test net output #1: loss3/top-1 = 0.76152
I1024 08:35:26.893409 142045 solver.cpp:715]
Test net output #2: loss3/top-5 = 0.931762
```

6.2.8. Results

Table 6. Time to train and speedup with parallelization.

	Time to train	**Speedup**
1x Intel Xeon Platinum 8160 node	~400h	-
448x Intel Xeon Platinum 8160 nodes	1h	400x

The example above demonstrates how to train the ResNet-50 architecture on the ImageNet dataset within 1 hour using 448 Intel Xeon Platinum 8160 (dual-socket) nodes (Table 6). Since distributed training can reduce the validation accuracy that is obtained as a result of the large global batch size, it is important to check the effect of the distribution on the validation accuracy. In this example, an improved validation accuracy of 76.1% (top-1) was achieved (the baseline for ResNet-50 is 75.3% top-1 accuracy).

6.2.9. Further references

- A full working example can be found at [84]. This example demonstrates even higher accuracy (76.3%) using only 72 training epochs.
- More information on how to achieve good training performance on Intel Architecture can be found at [145].
- A blogpost with details on how to train CNNs on larger problems such as ImageNet-22K and Places-365 can be found at [85].

6.3. Large-scale Image classification on ImageNet-1k (GPU-based training)

Algorithm	CNN
Hardware	Intel Skylake Bronze + NVIDIA GTX1080 Ti
Framework	MXNet / PyTorch
Libraries	cuDNN, NCCL
Scientific problem	Image Classification

This section studies the same use case as the previous section, but the calculation is performed using different hardware (GPUs instead of CPUs) and with a different framework (MXNet). For the research background and description of the computational problem, please refer to previous section. Only the batch size, and number of nodes/workers differ, see Table 7.

Table 7. Distribution of computational problem.

Batch size	256-4096 (64 per worker)
# Nodes / # Workers	1-16 / 1-64

6.3.1. Parallelization

In this example we will use the Apache MXNet framework [86] to train the ResNet-50 network on the ImageNet-1K dataset. In order to benefit from high-performance GPU kernels and multi-GPU efficient communi- cation, MXNet needs to be properly linked against the cuDNN and NCCL. cuDNN provides the low-level prim- itive support (convolution, matrix multiplication, etc.), while NCCL provides efficient communication routines (e.g. topology-aware AllReduce).

In order to run distributed training with MXNet, one can make use of the ps-lite implementation that comes together with the framework. This is a parameter server based approach, as opposed to a pure all-reduce (Figure 20) one. This parallelizes the training procedure by splitting the specified global batch size across the number of GPU workers by chunking in the batch dimension (i.e. data parallelism). In the forward pass, the model is replicated on each worker, and each replica handles a portion of the input. During the backwards pass, gradients from each replica are aggregated using the ps-lite library into the original model. In a parameter-server based approach the workers send their computed gradients to the parameter server, and the parameter server adds the gradients from all workers to the set of weights. After receiving the contributions from all workers, the new set of weights is broadcasted to the participating workers. MXNet supports both synchronous and asynchronous SGD, and these can be controlled through the use of the *kvstore* property.

6.3.2. IO

In order to get scalable performance, IO has to be treated carefully. The recommended data format to use with MXNet is RecordIO [87], which concatenates multiple examples into seekable binary files for better read effi- ciency. The im2rec.py tool located in the tools subfolder of the incubator-mxnet repository [88] can be used to convert individual images into .rec files.

```
python tools/im2rec.py --resize 480 --quality 95 --num-thread 16 mydata img_data
```

6.3.3. Training

On systems with many cores, assigning dedicated cores for computation and for communication can improve performance. This can be achieved by setting some environment variables. In order to select between FP32 and FP16 training, we can pass the --dtype float16 to the training script, otherwise training defaults to FP32.

Since the node-to-node communication is not handled using MPI, MXNet needs to perform socket-based commu- nication. Also, the actual IP addresses of the executing nodes need to be provided to the launch.sh script. If the batch system being used is PBS, one can use

```
cat $PBS_NODEFILE > machine_file uniq ~/machine_file | cat > ~/machinefile
rm -rf ~/ipfile while read line; do  nslookup $line | grep Address | tail -
n 1 | cut -b 10-24 >> ~/ipfile done < ~/machinefile
```

to write the IP addresses to a file. For batch systems using SLURM, a similar script based on the $SLURM_JOB_NODELIST environment variable can be created.

We follow the ImageNet example from the *example/image-classification* directory. To launch the training script:

```
OMP_NUM_THREADS=1 python ../../tools/launch.py -n 16 -H ~/ipfile python
train_imagenet.py --gpus 0,1,2,3 --network resnet --batch-size 256 --data-
nthreads 2 --kv-store dist_device_sync --data-train $HOME/train.rec --num-
epochs 90 --lr-step-epochs 30,60,80 --model-prefix $TMPDIR/resnet-50 --
data-val $HOME/val.rec --lr 1.6
```

where the arguments have the following meaning:

•OMP_NUM_THREADS=1, each MXNet process will use a single OpenMP thread.

•-H ~/ipfile, this is the host file passed to launch.sh,

•--gpus 0,1,2,3, we instruct MXNet that we want to use 4 GPUs per node.

•--network resnet, the network topology used.

•--batch-size 256, this is the batch size per node, so we have 64 images per GPU.

•--data-nthreads 2, this is the number of threads handling preprocessing of the input images.

•--kv-store dist_device_sync, this is the type of SGD to use. dist_device_sync means that gradient aggregation is performed on the GPU in a synchronous fashion.

•--num-epochs 90, number of training epochs.

•--lr-step-epochs 30,60,80, we want to decrease the learning rate by a factor of 10 after 30, 60, and 80 epochs.

6.3.4. Results

Table 8. Throughput in images/second as function of the number of GPUs. Each node contained 4 1080Ti GPUs.

# GPUs	Throughput FP32 [img/s]	Speedup
1 4	182	
8	705	3.9
16	1402	7.7
32	2680	14.7
64	5120	28.1
	7080	38.9

In this example, a speedup of almost 28x could be obtained when moving from 1 to 32 GPUs, demonstrating a very reasonable scaling efficiency of 87.5% (see Table 8). Increasing to 64 GPUs, scaling efficiency drops to 61%.

The same neural network was also run using PyTorch, with Horovod as a distribution layer. This was run on

synthetic data. For comparison of performance, we here include a scaling test with that configuration. Note that Table 8 reports MXnet results on real data; on synthetic data, the MXnet implementation achieved 728 img/s with

4 GPUs (slightly higher than on the real data). Thus, performance is marginally better using PyTorch + Horovod, while the scaling behaviour is similar (Table 9).

Table 9. Throughput in images/second as function of the number of GPUs. Each node contained 4 1080Ti GPUs.

# GPUs	Throughput FP32 [img/s]	Speedup
1	208	

# GPUs	Throughput FP32 [img/s]	Speedup
4	768	3.7
8	1497	7.2
16	2942	14.1
32	5796	27.9

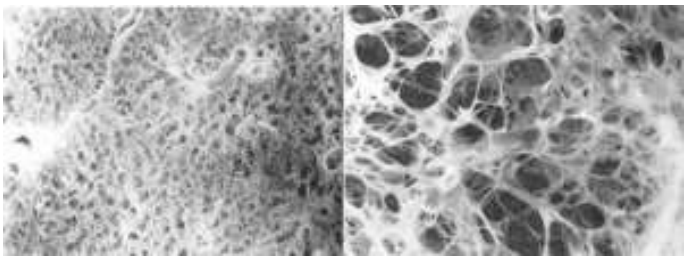
6.4. An AI Radiologist Trained using Deep Learning on Intel® Xeon® Scalable Processor HPC Supercomputer

Algorithm	CNN
Hardware	Intel Skylake / Knights Landing
Framework	IntelCaffe
Libraries	MKL-DNN
Scientific problem	Image Classification

6.4.1. Research background

Emphysema is (Figure 23) estimated to affect more than 3 million people in the U.S.1, and more than 65 million people worldwide [89]. Severe emphysema (types 3 / 4) are life threatening. Early detection is important to try to halt progression of emphysema. Also, pneumonia affects more than 1 million people each year in the U.S. [90], and more than 450 million [146] each year worldwide resulting in 1.4 million deaths per year worldwide. In many cases it is treatable with early detection.

Figure 23. Healthy Lung on the left and Lungs with severe Emphysema on the right.



Source: [91].

Figure 24. Chest X-Ray Images.

Source: [97].

Developed at Stanford University, CheXNet [92] is a deep learning Convolution Neural Network model for iden- tifying thoracic pathologies from the NIH ChestXray14 dataset. CheXNet is a 121-layer CNN that takes chest X- Ray images (e.g. Figure 24 as images and predicts the output probabilities of a pathology. It correctly detects pneu- monia localizing the areas in the image that are most indicative of the pathology. Stanford researchers demonstrate that they have been able to train the ChestX-Ray14 dataset using a pre-trained model of CheXNet-121 with Ima- geNet dataset. The NIH dataset consists of 112K frontal view of chest X-ray images from 30805 unique patients and annotated with up to 14 thoracic diseases including pneumonia and emphysema. CheXNet-121 outperforms the best published results on all 14 pathologies in the ChexX-Ray14 dataset. In this example, we tried to extend the Stanford research by using VGG-16 [147] [93] and ResNet-50 [94] scaled out on a large number of Intel Xeon HPC Dell Supercomputer and accurately trained on ImageNet2012 dataset [95]. Performance was compared against a single node training of a DenseNet topology [96]. We demonstrate that we are able to significantly reduce the training time and also outperform the CheXNet-121 published results in 4 pathological categories using VGG-16 and up to 10 categories using ResNet-50 including pneumonia and emphysema, two important categories, on a 200-Node 2 Socket Intel Xeon DellEMC HPC Su- percomputer.

6.4.2. Computational problem

In this example, we train two different networks: VGG-16 and ResNet-50. VGG-16 is a 16-layer convolutional neural network, while ResNet50 is a 50-layer convolutional neural network. Each network is first (pre-) trained on the ImageNet2012 dataset (containing approximately 1.4 million examples) to label images according to 1000 categories. Then, the pre-trained networks are trained further on the ChexX-Ray14 dataset (containing 111,000 examples) to distinguish 14 categories. Table 10, Table 11 and Table 12 summarize the size of the computational problem. Training on a single CPU would take ~300 hours, therefore, this training was distributed over 400 Intel Xeon 6148F CPUs.

Table 10. Size of the computational problem, dataset parameters.

	ImageNet2012	**ChestX-ray14**
Input data	32-bit float, [250 250 3]	32-bit float, [250 250 1]
Output data	32-bit float, [1000]	32-bit float, [14]
# Training examples	~1.4M	~111k

Table 11. Size of the computational problem, neural network parameters.

	VGG-16	**ResNet-50**
# Weights	~130M	~25M

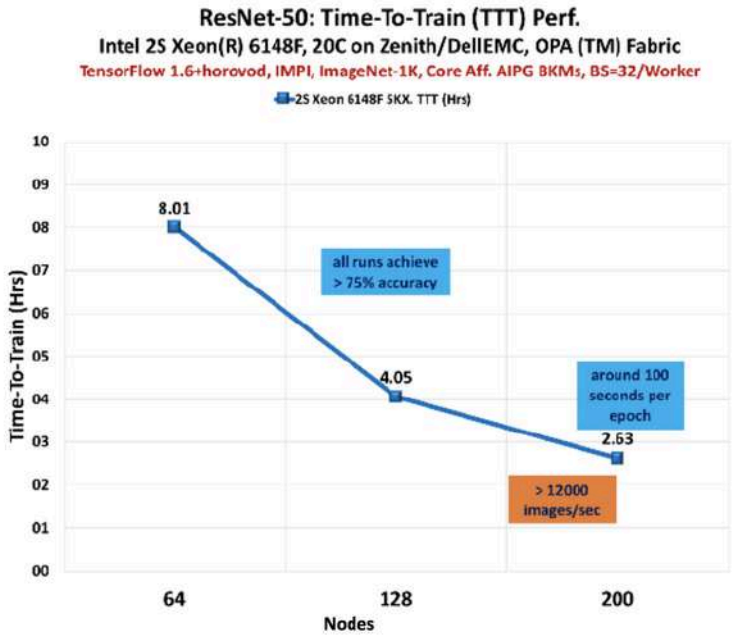
Table 12. Size of the computational problem, training parameters.

Batch size per worker	~16-32
# Epochs	~60
# Nodes / # Workers	Up to 200 / Up to 400

6.4.3. Transfer learning: pre-training on ImageNet2012

We first pre-Train the network on ImageNet2012 dataset on DellEMC 200-Node HPC Cluster using Intel opti- mized TensorFlow and Horovod. The chart below (Figure 25) shows the performance of ResNet-50 on 200-node DellEMC Zenith cluster pre-trained to > 75% Top-1 accuracy in about 2.6 Hrs.

Figure 25. DellEMC Zenith Intel(R) Xeon(R) HPC Supercomputer.



Source: [97].

The DellEMC compute cluster is used for this use case contains 2-socket nodes with Intel® Xeon® Gold 6148F CPUs, 96 GB of DDR4 memory, 200 GB local SSD, 10 Gbit Ethernet and a dual rail Intel® Omni-Path Host Fabric Interface interconnect. The following software was used: Intel® MPI Library 2017 Update 4, Intel® MPI Library 2019 Technical Preview OFI 1.5.0 PSM2 with multi-endpoint support, Red Hat* Enterprise Linux 6.7, TensorFlow 1.6 [98].

For completeness, we include here the command line used to launch the training:

```
OMP_NUM_THREADS=20 HOROVOD_FUSION_THRESHOLD=134217728
export I_MPI_FABRICS=tmi export I_MPI_TMI_PROVIDER=psm2 mpirun -np 512
-ppn 2 python resnet_main.py -train_batch_size 8192 --train_steps 14075
--num_intra_threads 20 --num_inter_threads 2 --mkl=True
--data_dir=/scratch/04611/valeriuc/tf-1.6/tpu_rec/train
--model_dir model_batch_8k_90ep --use_tpu=False --kmp_blocktime 1
```

Naturally, one would need to adjust the environment variables and arguments related to the fabric and number of threads to the architecture one is running on. The HOROVOD_FUSION_THRESHOLD affects the buffer size for the Tensor Fusion feature of Horovod [99], which allows fusing small allreduce operations for increased performance.

We use the following methodology to fine tune ResNet-50 on ImageNet2012 (more details can be found here [100]).

To further increase accuracy and efficient computing:

•When picking a pre-trained checkpoint do not pick the last one.

•Start with the learning rate at which the model was training when it was checkpointed.

•Perform gradual warmup of the learning rate, proportionally to the global batch size.

•Pack data in TF Records and consume them efficiently (asynchronously with compute) in Tensorflow.

We also observe that using TensorFlow directly rather than using Keras results in 4.7 times performance improve-
ment as shown by the Table 13 below on 128-Nodes on the DellEMC Zenith cluster.

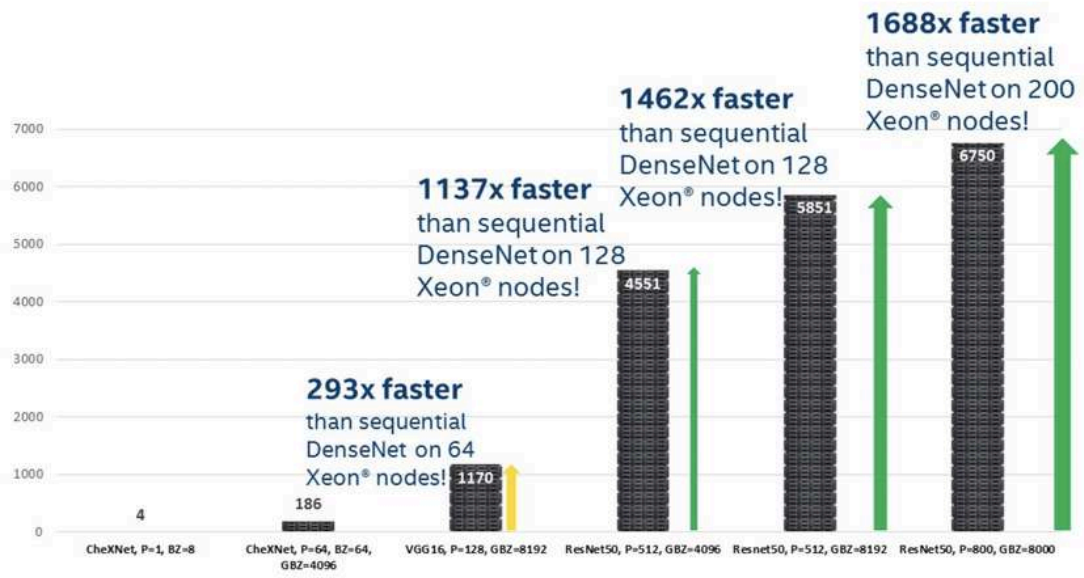
Table 13. Training Performance comparison between TensorFlow 1.6 and Keras.

Global batch size	Framework	# Nodes	Time per epoch
4096	Keras	128	85 s
4096	TensorFlow	128	18 s

6.4.4. Training on ChestX-Ray14 dataset

Figure 26 shows throughput for a pre-trained model with ImageNet2012 dataset, using Intel Xeon optimized Ten- sorFlow with MKL-DNN and exploiting NUMA domains with multiple workers per node. The VGG-16 through- put performance is 293X faster than sequential DenseNet on 64 nodes on Zenith cluster. The other three bar charts show that with a pre-trained ResNet-50 model, training scales even better: 1688 times faster than sequential DenseNet model.

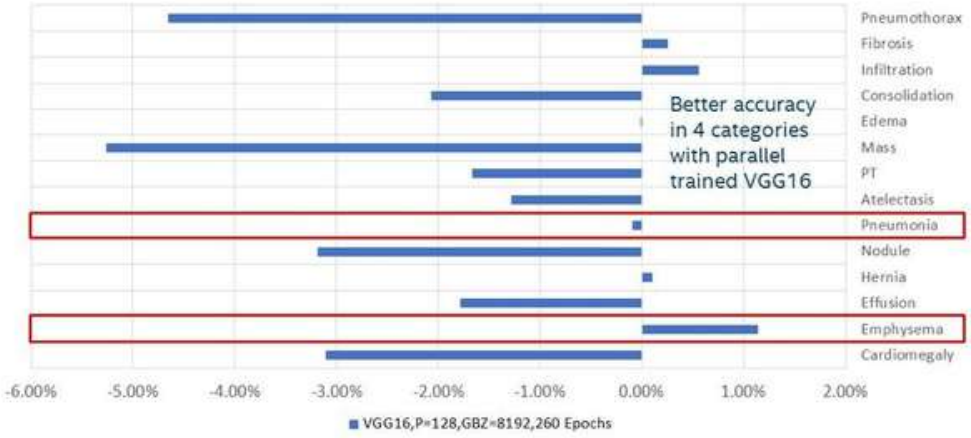
Figure 26. Scaleout Training Performance using ResNet-50 and VGG-16 compared to Single-Node DenseNet-121.



6.4.5. Training accuracy

In Figure 27 we show the accuracy measured with VGG-16 compared to the published CheXNet-121 for multiple pathologies. For the two important pathologies pneumonia and emphysema, VGG-16 is better or at most at par on AUROC metric compared to CheXNet.

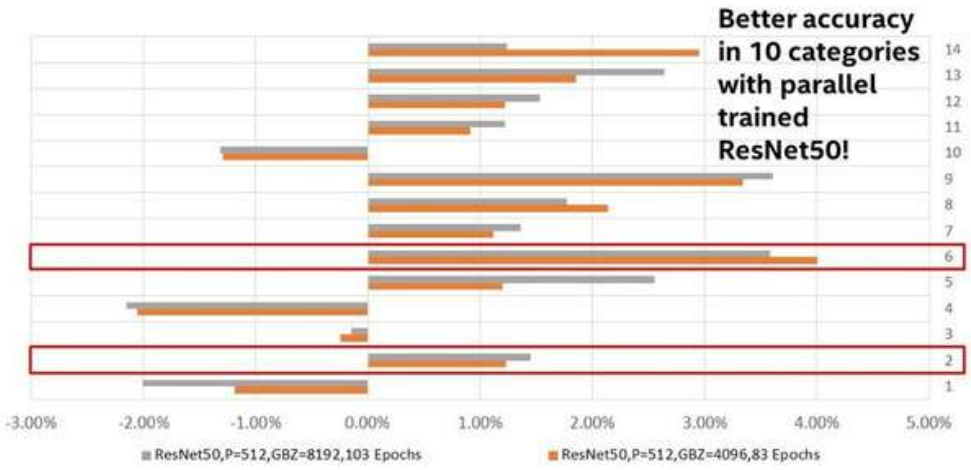
Figure 27. Training Accuracy using VGG-16 Relative to published CheXNet-121.



Source: [97].

Figure 28 shows the accuracy of ResNet-50 relative to CheXNet-121. We demonstrate that using the pre-trained ResNet-50 model against the ImageNet2012 dataset to a very high accuracy, we achieve better (positive) AUROC compared to the published CheXNet-121 in 10 categories out of 14 pathologies. Because of the complexity of the networks and the size of the datasets, these improved results could not have been obtained without the use of distributed deep learning.

Figure 28. Training Accuracy using ResNet-50 Relative to published CheXNet-121.



Source: [97].

7. List of abbreviations

BLVC	Berkeley Vision and Learning Center
CPU	Central Processing Unit
DNN	Deep Neural Network
FP16/32/64	16/32/64-bit precision floating point
(FL)OPS	(Floating Point) Operations per Second
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
HPC	High Performance Computing
FMA	Fused Multiply-Add
FPGA	Field-Programmable Gate Array
HBM(2)	High-Bandwidth memory (second generation)
MDS	Metadata Server
MPI	Message Parsing Interface
(Intel) MLSL	(Intel) Machine Learning Scaling Library
NCCL	NVIDIA Collective Communications Library
NSD	Network Shared Disk
OST	Object Storage Target
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
TPU	Tensor Processing Unit

Further documentation

Websites, forums, webinars

[1] *COCO dataset, http://cocodataset.org, https://arxiv.org/abs/1405.0312* .

[2] *Broyden-Fletcher-Goldfarb-Shanno algorithm, https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm* .

[3] *Wikipedia on evolutionary algorithms, https://en.wikipedia.org/wiki/Evolutionary_algorithm* .

[4] *Alpha Go, https://en.wikipedia.org/wiki/AlphaGo* .

[5] *The NVIDIA cuDNN library, https://developer.nvidia.com/cudnn* .

[6] *The Intel(R) Math Kernel Library for Deep Neural Networks, https://github.com/intel/mkl-dnn* .

[7] *Residual neural network, https://en.wikipedia.org/wiki/Residual_neural_network* .

[8] *Vanishing gradient problem, https://en.wikipedia.org/wiki/Vanishing_gradient_problem* .

[9] *TensorFlow data API, https://www.tensorflow.org/guide/datasets* .

[10] *Horovod, https://github.com/uber/horovod* .

[11] *BVLC Caffe, http://caffe.berkeleyvision.org/, https://github.com/BVLC/caffe* .

[12] *Caffe protocol buffer definition files (prototxt), http://caffe.berkeleyvision.org/tutorial/layers.html* .

[13] *Intel Caffe, https://software.intel.com/en-us/ai-academy/frameworks/caffe, https://github.com/intel/caffe* .

[14] *NVCaffe, https://docs.nvidia.com/deeplearning/dgx/caffe-user-guide/index.html* .

[15] *BVLC multigpu usage, https://github.com/BVLC/caffe/blob/master/docs/multigpu.md* .

[16] *NVIDIA Collective Communications Library (NCCL), https://developer.nvidia.com/nccl* .

[17] *NVCaffe docker container, https://ngc.nvidia.com/catalog/containers/nvidia:caffe* .

[18] *NVCaffe Training with mixed precision, http://on-demand.gputechconf.com/gtc/2018/presentation/s81012-training-neural-networks-with-mixed-precision.pdf* .

[19] *NVCaffe Release Notes, https://docs.nvidia.com/deeplearning/dgx/caffe-release-notes/running.html* .

[20] *Intel(R) Machine Learning Scaling Library (MLSL), https://github.com/intel/MLSL* .

[21] *Intel Caffe Performance, https://www.intel.com/content/dam/www/public/us/en/documents/presentation/ai-ml-ananth-sankaranarayanan-accelerating-ml-software-on-ia.pdf* .

[22] *Intel(R) Omni-Path Architecture, https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html* .

[23] *Intel(R) MPI Library, https://software.intel.com/en-us/mpi-library* .

[24] *Intel I_MPI_ADJUST environment variables, https://software.intel.com/en-us/mpi-developer-reference-linux-i-mpi-adjust-family-environment-variables* .

[25] *Stampede2 supercomputer, https://www.tacc.utexas.edu/systems/stampede2* .

[26] *Mare Nostrum supercomputer, https://www.bsc.es/marenostrum/marenostrum* .

[27] *Intel Thread Affinity Interface, https://software.intel.com/en-us/node/522691* .

[28] *PyTorch, https://pytorch.org/* .

[29] *Torch.distributed, https://pytorch.org/docs/stable/distributed.html* .

[30] *Open MPI library, https://www.open-mpi.org/* .

[31] *MVAPICH2 library, http://mvapich.cse.ohio-state.edu/* .

[32] *CUDA Interprocess Communication, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#interprocess-communication* .

[33] *NVIDIA GPUDirect, https://developer.nvidia.com/gpudirect* .

[34] *TensorFlow, https://www.tensorflow.org/* .

[35] *TensorFlow Distributed, https://www.tensorflow.org/deploy/distributed* .

[36] *Horovod model parallel, https://github.com/uber/horovod/issues/96* .

[37] *TensorFlow Architecture, https://www.tensorflow.org/guide/extend/architecture* .

[38] *TFRecord File Format, https://www.tensorflow.org/tutorials/load_data/tf_records* .

[39] *Converting to TFRecord File Format, https://github.com/tensorflow/models/tree/master/research/slim#downloading-and-converting-to-tfrecord-format* .

[40] *TensorFlow Performance Documentation, https://www.tensorflow.org/guide/performance/overview* .

[41] *NVIDIA Pascal Architecture, http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf* .

[42] *NVIDIA Volta Architecture, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf* .

[43] *NVIDIA Software Development Kit documentation on mixed precision training, https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html* .

[44] *NVIDIA DGX-1 system, http://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf* .

[45] *NVIDIA GPU accelerated libraries, https://developer.nvidia.com/gpu-accelerated-libraries* .

[46] *NVIDIA cuBLAS library, https://docs.nvidia.com/cuda/cublas/index.html* .

[47] *NETLIB BLAS library, http://www.netlib.org/blas/* .

[48] *NVIDIA cuSPARSE library, https://docs.nvidia.com/cuda/cusparse/index.html* .

[49] *AMD Radeon Instinct accelerators, https://www.amd.com/en/graphics/servers-radeon-instinct-mi* .

[50] *Radeon Open eCosystem (ROCm) platform, https://rocm.github.io/* .

[51] *ROCm programming languages, https://rocm.github.io/languages.html* .

[52] *ROCm HIP, https://github.com/ROCm-Developer-Tools/HIP* .

[53] *ROCm MIOpen library, https://github.com/ROCmSoftwarePlatform/MIOpen* .

[54] *ROCm BLAS library, https://github.com/ROCmSoftwarePlatform/rocBLAS* .

[55] *ROCm Software Platform repository, https://github.com/ROCmSoftwarePlatform* .

[56] *ROCm TensorFlow Docker container, https://hub.docker.com/r/rocm/tensorflow/* .

[57] *Deep Learning on ROCm, https://rocm.github.io/dl.html* .

[58] *Intel(R) Xeon Scalable processor family, https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-platform.html* .

[59] *Intel(R) MKL, https://software.intel.com/en-us/mkl* .

[60] *AMD EPYC processor family, https://www.amd.com/en/products/epyc* .

[61] *Wikichip on AMD EPYC 7601, https://en.wikichip.org/wiki/amd/epyc/7601* .

[62] *AMD EPYC 7601 FLOPS calculation, https://community.amd.com/thread/231240* .

[63] *AMD low level libraries, https://developer.amd.com/amd-cpu-libraries/* .

[64] *PRACE AMD EPYC best practice guide, http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-AMD.pdf* .

[65] *AMD Instinct MI25 datasheet, https://www.amd.com/en/products/professional-graphics/instinct-mi25* .

[66] *AMD Instinct MI50 datasheet, https://www.amd.com/system/files/documents/radeon-instinct-mi50-datasheet.pdf* .

[67] *AMD Instinct MI60 datasheet, https://www.amd.com/system/files/documents/radeon-instinct-mi60-datasheet.pdf* .

[68] *NVIDIA P100 datasheet, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-datasheet.pdf* .

[69] *NVIDIA P100 product page, https://www.nvidia.com/en-us/data-center/tesla-p100/* .

[70] *NVIDIA V100 datasheet, https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf* .

[71] *NVlink, https://en.wikichip.org/wiki/nvidia/nvlink* .

[72] *Microway Volta architecture description, https://www.microway.com/knowledge-center-articles/in-depth-comparison-of-nvidia-tesla-volta-gpu-accelerators/* .

[73] *Intel Xeon Platinum 8180 product page, https://ark.intel.com/products/120496/Intel-Xeon-Platinum-8180-Processor-38-5M-Cache-2-50-GHz-* .

[74] *Wikichip on Intel Xeon Platinum 8180, https://en.wikichip.org/wiki/intel/xeon_platinum/8180* .

[75] *Lustre website, http://lustre.org/about/* .

[76] *Lightning Memory-Mapped Database format, http://caffe.berkeleyvision.org/tutorial/data.html* .

[77] *Lustre wiki on Object Storage Service, http://wiki.lustre.org/Lustre_Object_Storage_Service_(OSS)* .

[78] *PRACE Parallel I/O best practice guide, http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Parallel-IO.pdf* .

[79] *Kitti Tracking Sequences Training Set, http://www.cvlibs.net/datasets/kitti/eval_tracking.php* .

[80] *NVIDIA DGX-1 product page, https://www.nvidia.com/en-us/data-center/dgx-1/* .

[81] *PyTorch documentation on torch.nn.DataParallel, https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html* .

[82] *FutureGAN code, https://github.com/TUM-LMF/FutureGAN* .

[83] *Convolutional Neural Networks for Visual Recognition introductory lecture, http://cs231n.github.io/classification/* .

[84] *ResNet50 on ImageNet-1k with custom learning rate, code, https://github.com/sara-nl/caffe/tree/master/models/intel_optimized_models/multinode/resnet50_custom_lr* .

[85] *Blogpost on training ImageNet-1k in less than 40 minutes, https://blog.surf.nl/en/imagenet-1k-training-on-intel-xeon-phi-in-less-than-40-minutes/* .

[86] *Apache MXNet, https://mxnet.apache.org/* .

[87] *RecordIO (MXNet recommended file format), mxnet.io/architecture/note_data_loading.html* .

[88] *MXNet repository, https://github.com/apache/incubator-mxnet* .

[89] *WHO Chronic respiratory diseases, http://www.who.int/respiratory/copd/burden/en/* .

[90] *CDC webpage on pneumonia, https://www.cdc.gov/features/pneumonia/index.html* .

[91] *Airway Bypass Stenting for Severe Emphysema, http://www.ctsnet.org/article/airway-bypass-stenting-severe-emphysema* .

[92] *CheXNet github, https://stanfordmlgroup.github.io/projects/chexnet/* .

[93] *TensorFlow implementation of VGG networks, https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/slim/python/slim/nets/vgg.py* .

[94] *TensorFlow implementation ResNet-50, https://github.com/tensorflow/tpu/tree/master/models/official/resnet* .

[95] *ImageNet2012 dataset, http://www.image-net.org/challenges/LSVRC/2012/* .

[96] *DenseNet source code, https://github.com/liuzhuang13/DenseNet* .

[97] *Intel/Dell Presentation: Efficient neural network training on Intel Xeon-based supercomputers, https://cdn.oreillystatic.com/en/assets/1/event/282/Efficient%20neural%20network%20training%20on%20Intel%20Xeon-based%20supercomputers%20Presentation.pdf* .

[98] *TensorFlow install from sources, https://www.tensorflow.org/install/install_sources* .

[99] *Horovod tensor fusion, https://github.com/uber/horovod/blob/master/docs/tensor-fusion.md* .

[100] *SURFsara description on finetuning ResNet-50 on ImageNet-2012 training, https://surfdrive.surf.nl/files/index.php/s/xrEFLPvo7IDRARs/download?path=%2F&files=SURFsara_Tensorflow_Perf_Convergence_BKM_April_2018_0.93.pdf* .

Manuals, papers

[101] *Silver D, Hubert T, Schrittwieser J, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play, Science, 2018. https://doi.org/10.1126/science.aar6404* .

[102] *Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks, Advances in neural information processing systems, 2012* .

[103] *He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. Proceedings of the IEEE conference on computer vision and pattern recognition, 2016* .

[104] *Sun C, Shrivastava A, Singh S, Gupta A. Deep residual learning for image recognition. Revisiting Unreasonable Effectiveness of Data in Deep Learning Era, arXiv:1707.02968, 2017. https://arxiv.org/abs/1707.02968* .

[105] *Brock A, Donahue F, Simonyan D. Large scale GAN training for high fidelity natural image synthesis, arXiv:1809.11096, 2018. https://arxiv.org/abs/1809.11096 .*

[106] *Zhou Z, Cai G, Rong S, et al. Activation Maximization Generative Adversarial Nets, arXiv:1703.02000, 2017. https://arxiv.org/abs/1703.02000 .*

[107] *Tolstikhin I, Gelly S, Bousquet O, Simon-Gabriel CJ, Schölkopf B. AdaGAN: Boosting Generative Models, arXiv:1701.02386, 2017. https://arxiv.org/abs/1701.02386 .*

[108] *Makhzani A, Shlens J, Jaitly N, Goodfellow I, Frey B. Adversarial Autoencoders, arXiv:1511.05644, 2015. https://arxiv.org/abs/1701.02386 .*

[109] *Saatchi Y, Wilson AG. Bayesian GAN, arXiv:1705.09558, 2017. https://arxiv.org/abs/1701.02386 .*

[110] *Mirza M, Osindero S. Conditional Generative Adversarial Nets, arXiv:1411.1784, 2014. https://arxiv.org/abs/1701.02386, https://github.com/wiseodd/generative-models .*

[111] *Zhao J, Mathieu M, LeCun Y. Energy-based Generative Adversarial Network, arXiv:1609.03126, 2016. https://arxiv.org/abs/1701.02386, https://github.com/wiseodd/generative-models .*

[112] *Goodfellow IJ, Pouget-Abadie J, Mirza M, et al. Generative Adversarial Networks, arXiv:1406.2661, 2014. https://arxiv.org/abs/1406.2661, https://github.com/goodfeli/adversarial, https://github.com/wiseodd/generative-models .*

[113] *Im DJ, Ma H, Kim CD, Taylor G. Generative Adversarial Parallelization, arXiv:1612.04021, 2016. https://arxiv.org/abs/1612.04021, https://github.com/wiseodd/generative-models .*

[114] *Mehrotra A, Dukkipati A. Generative Adversarial Residual Pairwise Networks for One Shot Learning, arXiv:1703.08033, 2017. https://arxiv.org/abs/1703.08033 .*

[115] *Lim JH, Ye JC. Geometric GAN, arXiv:1705.02894, 2017. https://arxiv.org/abs/1705.02894 .*

[116] *Dai Z, Yang Z, Yang F, Cohen WW, Salakhutdinov R. Good Semi-supervised Learning that Requires a Bad GAN, arXiv:1705.09783, 2017. https://arxiv.org/abs/1705.09783 .*

[117] *Nagarajan V, Kolter JZ. Gradient descent GAN optimization is locally stable, arXiv:1706.04156, 2017. https://arxiv.org/abs/1706.04156 .*

[118] *Gulrajani I, Ahmed F, Arjovsky M, Dumoulin V, Courville A. Improved Training of Wasserstein GANs, arXiv:1704.00028, 2017. https://arxiv.org/abs/1704.00028, https://github.com/wiseodd/generative-models .*

[119] *Chen X, Duan Y, Houthooft R, Schulman J, Sutskever I, Abbeel P. InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets, arXiv:1606.03657, 2016. https://arxiv.org/abs/1606.03657, https://github.com/wiseodd/generative-models .*

[120] *Qi GJ. Loss-Sensitive Generative Adversarial Networks on Lipschitz Densities, arXiv:1701.06264, 2017. https://arxiv.org/abs/1701.06264 .*

[121] *Kilcher Y, Becigneul G, Hofmann T. Parametrizing filters of a CNN with a GAN, arXiv:1710.11386, 2017. https://arxiv.org/abs/1710.11386 .*

[122] *Makhzani A, Frey B. PixelGAN Autoencoders, arXiv:1706.00531, 2017. https://arxiv.org/abs/1706.00531 .*

[123] *Karras T, Aila T, Laine S, Lehtinen J. Progressive Growing of GANs for Improved Quality, Stability, and Variation, arXiv:1710.10196, 2017. https://arxiv.org/abs/1710.10196, https://github.com/tkarras/progressive_growing_of_gans .*

[124] *Xue Y, Xu T, Zhang H, Long R, Huang X. SegAN: Adversarial Network with Multi-scale L1 Loss for Medical Image Segmentation, arXiv:1706.01805, 2017. https://arxiv.org/abs/1706.01805 .*

[125] *Yu L, Zhang W, Wang J, Yu Y. SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient, arXiv:1609.05473, 2016. https://arxiv.org/abs/1609.05473 .*

[126] *Narodytska N, Kasiviswanathan SP. Simple Black-Box Adversarial Perturbations for DeepNetworks, arXiv:1612.06299, 2016. https://arxiv.org/abs/1612.06299 .*

[127] *Metz L, Poole B, Pfau D, Sohl-Dickstein J. Unrolled Generative Adversarial Networks, arXiv:1611.02163, 2016. https://arxiv.org/abs/1611.02163 .*

[128] *Radford A, Metz L, Chintala S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, arXiv:1511.06434, 2015. https://arxiv.org/abs/1511.06434, https://github.com/Newmu/dcgan_code, https://github.com/pytorch/examples/tree/master/dcgan, https://github.com/carpedm20/DCGAN-tensorflow, https://github.com/soumith/dcgan.torch, https://github.com/jacobgil/keras-dcgan .*

[129] *Arjovsky M, Chintala S, Bottou L. Wasserstein GAN, arXiv:1701.07875, 2017. https://arxiv.org/abs/1701.07875, https://github.com/martinarjovsky/WassersteinGAN, https://github.com/wiseodd/generative-models .*

[130] *Schuster M, Paliwal KK. Bidirectional Recurrent Neural Networks, IEEE Transactions on Signal Processing, 1997. https://ieeexplore.ieee.org/document/650093 .*

[131] *Hochreiter S, Schmidhuber J. Long Short-Term Memory, Neural Computation, 1997. https://doi.org/10.1162/neco.1997.9.8.1735 .*

[132] *Graves A, Fernandez S, Schmidhuber J. Multi-Dimensional Recurrent Neural Networks, arXiv:0705.2011, 2007. https://arxiv.org/abs/0705.2011 .*

[133] *Cho K, van Merrienboer B, Gulcehre C, et al. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, arXiv:1406.1078, 2014. https://arxiv.org/abs/1406.1078 .*

[134] *Chung J, Gulcehre C, Cho K, Bengio Y. Gated Feedback Recurrent Neural Networks, arXiv:1502.02367, 2015. https://arxiv.org/abs/1502.02367, http://proceedings.mlr.press/v37/chung15.pdf, http://proceedings.mlr.press/v37/chung15-supp.pdf .*

[135] *Tai KS, Socher R, Manning CD. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks, arXiv:1503.00075, 2015. https://arxiv.org/abs/1503.00075 .*

[136] *Kalchbrenner N, Danihelka I, Graves A. Grid Long Short-Term Memory, arXiv:1507.01526, 2015. https://arxiv.org/abs/1507.01526 .*

[137] *Ben-Nun T, Hoefler T. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis, arXiv:1802.09941, 2018. https://arxiv.org/abs/1802.09941 .*

[138] *Goyal P, Dollár P, Noordhuis P, et al. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, arXiv:1706.02677, 2017. https://arxiv.org/abs/1706.02677 .*

[139] *Smith LN. A disciplined approach to neural network hyper-parameters: Part 1 -- learning rate, batch size, momentum, and weight decay, arXiv:1803.09820, 2018. https://arxiv.org/abs/1803.09820 .*

[140] *Codreanu V, Podareanu D, Saletore V. Large Minibatch Training on Supercomputers with Improved Accuracy and Reduced Time to Train, IEEE/ACM Machine Learning in HPC Environments, 2018. http://conferences.computer.org/scw/2018/pdfs/MLHPC2018-6dwpPKb3byMiMuAzcqSw8a/6e46BZtbGTtcfDkV7cHqhg/18h5Owu3Pce6f1Y6Vza2VF.pdf .*

[141] *MPI: A Message-Passing Interface Standard Version 3.1. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf .*

[142] *Pumma S, Si M, Feng WC, Balaji P, Parallel I/O Optimizations for Scalable Deep Learning, IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), 2017. https://ieeexplore.ieee.org/document/8368426 .*

[143] *Zhang Z, Huang L, Manor U, et al, FanStore: Enabling Efficient and Scalable I/O for Distributed Deep Learning, arXiv:1809.10799, 2018. https://arxiv.org/abs/1809.10799 .*

[144] *Aigner S, Körner M, FutureGAN: Anticipating the Future Frames of Video Sequences using Spatio-Temporal 3d Convolutions in Progressively Growing GANs, arXiv:1810.01325, 2018. https://arxiv.org/abs/1810.01325 .*

[145] *Codreanu V, Podareanu D, Saletore V, Scale out for large minibatch SGD:Residual network training on ImageNet-1K withimproved accuracy and reduced time to train, arXiv:1711.04291, 2017. https://arxiv.org/abs/1711.04291 .*

[146] *Ruuskanen O, Lahti E, Jennings LC, Murdoch DR, Viral pneumonia, The Lancet, 2011. https://doi.org/10.1016/S0140-6736(10)61459-6 .*

[147] *Simonyan K, Zisserman A, Very Deep Convolutional Networks for Large-Scale Image Recognition, arXiv:1409.1556, 2014. https://arxiv.org/abs/1409.1556 .*